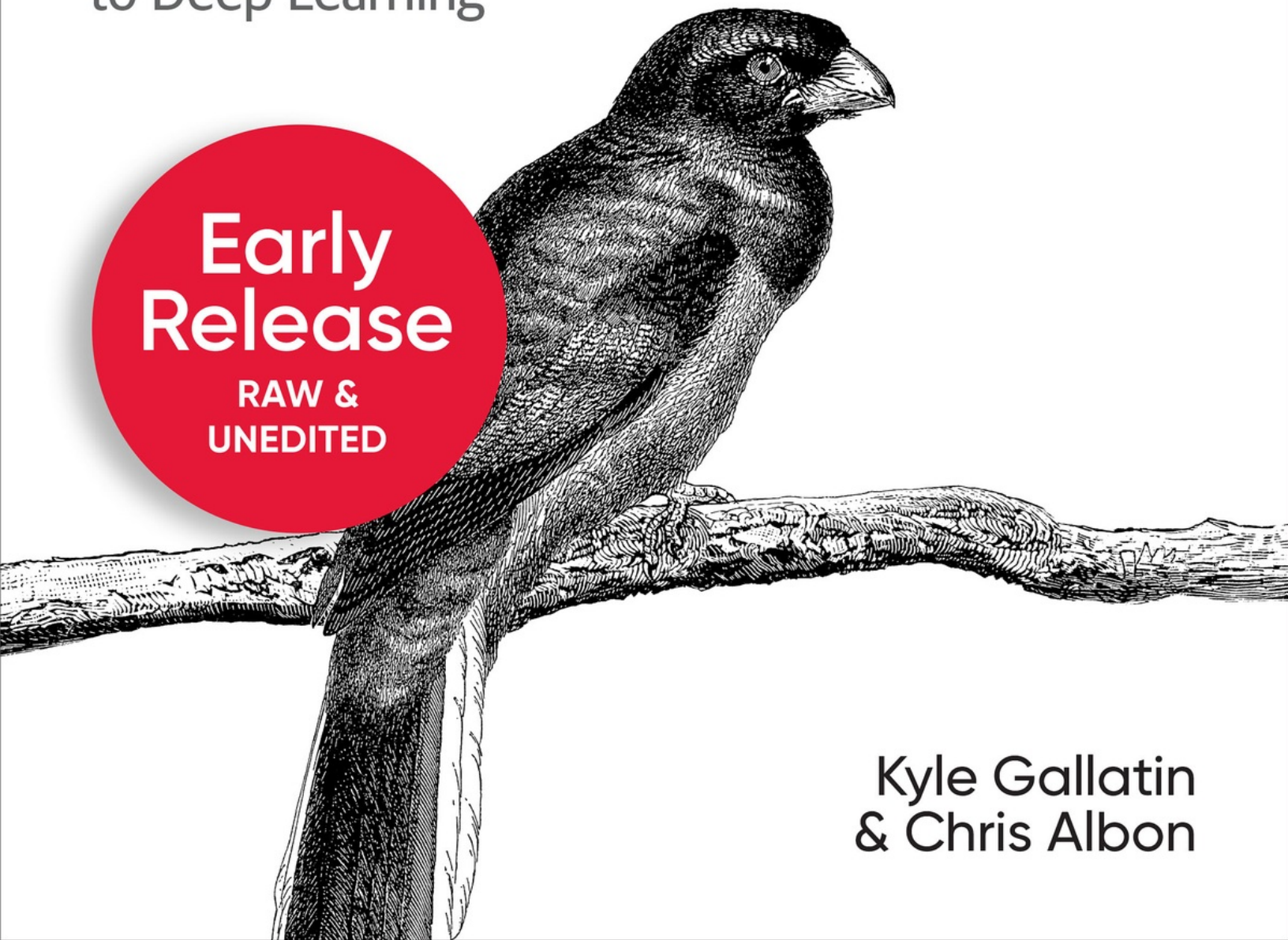
O'REILLY®

Second Edition

Machine Learning with Python Cookbook

Practical Solutions from Preprocessing to Deep Learning



Early Release

RAW & UNEDITED

Kyle Gallatin
& Chris Albon

Machine Learning with Python Cookbook

SECOND EDITION

Practical Solutions from Preprocessing to Deep Learning

Kyle Gallatin and Chris Albon

Beijing • Boston • Farnham • Sebastopol • Tokyo

Machine Learning with Python Cookbook

by Kyle Gallatin and Chris Albon

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Nicole Butterfield
- Development Editor Jeff Bleiel
- Production Editor: Christopher Faucher
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- April 2018: First Edition
- October 2023: Second Edition

Revision History for the Early Release

- 2022-08-24: First Release
- 2022-10-05: Second Release
- 2022-12-08: Third Release
- 2023-01-18: Fourth Release
- 2023-03-01: Fifth Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098135720 for release details.

978-1-098-13566-9

Chapter 1. Working with Vectors, Matrices and Arrays in NumPy

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at feedback.mlpythoncookbook@gmail.com.

1.0 Introduction

NumPy is a foundational tool of the Python machine learning stack. NumPy allows for efficient operations on the data structures often used in machine learning: vectors, matrices, and tensors. While NumPy is not the focus of this book, it will show up frequently throughout the following chapters. This chapter covers the most common NumPy operations we are likely to run into while working on machine learning workflows.

1.1 Creating a Vector

Problem

You need to create a vector.

Solution

Use NumPy to create a one-dimensional array:

```
# Load library
import numpy as np

# Create a vector as a row
vector_row = np.array([1, 2, 3])

# Create a vector as a column
vector_column = np.array([[1],
                          [2],
                          [3]])
```

Discussion

NumPy's main data structure is the multidimensional array. A vector is just an array with a single dimension. In order to create a vector, we simply create a one-dimensional array. Just like vectors, these

arrays can be represented horizontally (i.e., rows) or vertically (i.e., columns).

See Also

- Vectors, Math Is Fun
- Euclidean vector, Wikipedia

1.2 Creating a Matrix

Problem

You need to create a matrix.

Solution

Use NumPy to create a two-dimensional array:

```
# Load library
import numpy as np

# Create a matrix
matrix = np.array([[1, 2],
                   [1, 2],
                   [1, 2]])
```

Discussion

To create a matrix we can use a NumPy two-dimensional array. In our solution, the matrix contains three rows and two columns (a column of 1s and a column of 2s).

NumPy actually has a dedicated matrix data structure:

```
matrix_object = np.mat([[1, 2],
                        [1, 2],
                        [1, 2]])
```

```
matrix([[1, 2],
        [1, 2],
        [1, 2]])
```

However, the matrix data structure is not recommended for two reasons. First, arrays are the de facto standard data structure of NumPy. Second, the vast majority of NumPy operations return arrays, not matrix objects.

See Also

- Matrix, Wikipedia
- Matrix, Wolfram MathWorld

1.3 Creating a Sparse Matrix

Problem

Given data with very few nonzero values, you want to efficiently represent it.

Solution

Create a sparse matrix:

```
# Load libraries
import numpy as np
from scipy import sparse

# Create a matrix
matrix = np.array([[0, 0],
                   [0, 1],
                   [3, 0]])

# Create compressed sparse row (CSR) matrix
matrix_sparse = sparse.csr_matrix(matrix)
```

Discussion

A frequent situation in machine learning is having a huge amount of data; however, most of the elements in the data are zeros. For example, imagine a matrix where the columns are every movie on Netflix, the rows are every Netflix user, and the values are how many times a user has watched that particular movie. This matrix would have tens of thousands of columns and millions of rows! However, since most users do not watch most movies, the vast majority of elements would be zero.

A sparse matrix is a matrix in which most elements are 0. Sparse matrices only store nonzero elements and assume all other values will be zero, leading to significant computational savings. In our solution, we created a NumPy array with two nonzero values, then converted it into a sparse matrix. If we view the sparse matrix we can see that only the nonzero values are stored:

```
# View sparse matrix
print(matrix_sparse)
```

```
  (1, 1)    1
  (2, 0)    3
```

There are a number of types of sparse matrices. However, in *compressed sparse row* (CSR) matrices, `(1, 1)` and `(2, 0)` represent the (zero-indexed) indices of the non-zero values `1` and `3`, respectively. For example, the element `1` is in the second row and second column. We can see the advantage of sparse matrices if we create a much larger matrix with many more zero elements and then compare this larger matrix with our original sparse matrix:

```
# Create larger matrix
matrix_large = np.array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                         [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
                         [3, 0, 0, 0, 0, 0, 0, 0, 0, 0]])

# Create compressed sparse row (CSR) matrix
```

```
matrix_large_sparse = sparse.csr_matrix(matrix_large)

# View original sparse matrix
print(matrix_sparse)
```

```
  (1, 1)    1
  (2, 0)    3
```

```
# View larger sparse matrix
print(matrix_large_sparse)
```

```
  (1, 1)    1
  (2, 0)    3
```

As we can see, despite the fact that we added many more zero elements in the larger matrix, its sparse representation is exactly the same as our original sparse matrix. That is, the addition of zero elements did not change the size of the sparse matrix.

As mentioned, there are many different types of sparse matrices, such as compressed sparse column, list of lists, and dictionary of keys. While an explanation of the different types and their implications is outside the scope of this book, it is worth noting that while there is no "best" sparse matrix type, there are meaningful differences between them and we should be conscious about why we are choosing one type over another.

See Also

- Sparse matrices, SciPy documentation
- 101 Ways to Store a Sparse Matrix

1.4 Pre-allocating Numpy Arrays

Problem

You need to pre-allocate arrays of a given size with some value.

Solution

NumPy has functions for generating vectors and matrices of any size using 0s, 1s, or values of your choice.

```
# Load library
import numpy as np

# Generate a vector of shape (1,5) containing all zeros
vector = np.zeros(shape=5)

# View the vector
print(vector)
```

```
array([0., 0., 0., 0., 0.])
```

```
# Generate a matrix of shape (3,3) containing all ones
matrix = np.full(shape=(3,3), fill_value=1)

# View the vector
print(matrix)
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Discussion

Generating arrays prefilled with data is useful for a number of purposes, such as making code more performant or having synthetic data to test algorithms with. In many programming languages, pre-allocating an array of default values (such as 0s) is considered common practice.

1.5 Selecting Elements

Problem

You need to select one or more elements in a vector or matrix.

Solution

NumPy's arrays make it easy to select elements in vectors or matrices:

```
# Load library
import numpy as np

# Create row vector
vector = np.array([1, 2, 3, 4, 5, 6])

# Create matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Select third element of vector
vector[2]
```

```
3
```

```
# Select second row, second column
matrix[1,1]
```

```
5
```

Discussion

Like most things in Python, NumPy arrays are zero-indexed, meaning that the index of the first element is 0, not 1. With that caveat, NumPy offers a wide variety of methods for selecting (i.e., indexing and slicing) elements or groups of elements in arrays:

```
# Select all elements of a vector
vector[:]
```

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

```
# Select everything up to and including the third element
vector[:3]
```

```
array([1, 2, 3])
```

```
# Select everything after the third element
vector[3:]
```

```
array([4, 5, 6])
```

```
# Select the last element
vector[-1]
```

```
6
```

```
# Reverse the vector
vector[::-1]
```

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

```
# Select the first two rows and all columns of a matrix
matrix[:2,:]
```

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

```
# Select all rows and the second column
matrix[:,1:2]
```

```
array([[2],
       [5],
       [8]])
```

1.6 Describing a Matrix

Problem

You want to describe the shape, size, and dimensions of the matrix.

Solution

Use the `shape`, `size`, and `ndim` attributes of a NumPy object:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3, 4],
                   [5, 6, 7, 8],
                   [9, 10, 11, 12]])

# View number of rows and columns
matrix.shape
```

```
(3, 4)
```

```
# View number of elements (rows * columns)
matrix.size
```

```
12
```

```
# View number of dimensions
matrix.ndim
```

```
2
```

Discussion

This might seem basic (and it is); however, time and again it will be valuable to check the shape and size of an array both for further calculations and simply as a gut check after some operation.

1.7 Applying Functions Over Each Element

Problem

You want to apply some function to all elements in an array.

Solution

Use NumPy's `vectorize` method:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Create function that adds 100 to something
add_100 = lambda i: i + 100

# Create vectorized function
vectorized_add_100 = np.vectorize(add_100)

# Apply function to all elements in matrix
vectorized_add_100(matrix)
```

```
array([[101, 102, 103],
       [104, 105, 106],
       [107, 108, 109]])
```

Discussion

NumPy's `vectorize` class converts a function into a function that can apply to all elements in an array or slice of an array. It's worth noting that `vectorize` is essentially a `for` loop over the elements and does not increase performance. Furthermore, NumPy arrays allow us to perform operations between arrays even if their dimensions are not the same (a process called *broadcasting*). For example, we can

create a much simpler version of our solution using broadcasting:

```
# Add 100 to all elements
matrix + 100
```

```
array([[101, 102, 103],
       [104, 105, 106],
       [107, 108, 109]])
```

Broadcasting does not work for all shapes and situations, but a common way of applying simple operations over all elements of a numpy array.

1.8 Finding the Maximum and Minimum Values

Problem

You need to find the maximum or minimum value in an array.

Solution

Use NumPy's `max` and `min` methods:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Return maximum element
np.max(matrix)
```

```
9
```

```
# Return minimum element
np.min(matrix)
```

```
1
```

Discussion

Often we want to know the maximum and minimum value in an array or subset of an array. This can be accomplished with the `max` and `min` methods. Using the `axis` parameter we can also apply the operation along a certain axis:

```
# Find maximum element in each column
np.max(matrix, axis=0)
```

```
array([7, 8, 9])
```

```
# Find maximum element in each row
np.max(matrix, axis=1)
```

```
array([3, 6, 9])
```

1.9 Calculating the Average, Variance, and Standard Deviation

Problem

You want to calculate some descriptive statistics about an array.

Solution

Use NumPy's `mean`, `var`, and `std`:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Return mean
np.mean(matrix)
```

```
5.0
```

```
# Return variance
np.var(matrix)
```

```
6.666666666666667
```

```
# Return standard deviation
np.std(matrix)
```

```
2.5819888974716112
```

Discussion

Just like with `max` and `min`, we can easily get descriptive statistics about the whole matrix or do calculations along a single axis:

```
# Find the mean value in each column
np.mean(matrix, axis=0)
```

```
array([ 4.,  5.,  6.])
```

1.10 Reshaping Arrays

Problem

You want to change the shape (number of rows and columns) of an array without changing the element values.

Solution

Use NumPy's `reshape`:

```
# Load library
import numpy as np

# Create 4x3 matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9],
                   [10, 11, 12]])

# Reshape matrix into 2x6 matrix
matrix.reshape(2, 6)
```

```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]])
```

Discussion

`reshape` allows us to restructure an array so that we maintain the same data but it is organized as a different number of rows and columns. The only requirement is that the shape of the original and new matrix contain the same number of elements (i.e., the same size). We can see the size of a matrix using `size`:

```
matrix.size
```

```
12
```

One useful argument in `reshape` is `-1`, which effectively means "as many as needed," so `reshape(1, -1)` means one row and as many columns as needed:

```
matrix.reshape(1, -1)
```

```
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12]])
```

Finally, if we provide one integer, `reshape` will return a 1D array of that length:

```
matrix.reshape(12)
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

1.11 Transposing a Vector or Matrix

Problem

You need to transpose a vector or matrix.

Solution

Use the `T` method:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Transpose matrix
matrix.T
```

```
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

Discussion

Transposing is a common operation in linear algebra where the column and row indices of each element are swapped. One nuanced point that is typically overlooked outside of a linear algebra class is that, technically, a vector cannot be transposed because it is just a collection of values:

```
# Transpose vector
np.array([1, 2, 3, 4, 5, 6]).T
```

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

However, it is common to refer to transposing a vector as converting a row vector to a column vector (notice the second pair of brackets) or vice versa:

```
# Tranpose row vector
np.array([[1, 2, 3, 4, 5, 6]]).T
```

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

1.12 Flattening a Matrix

Problem

You need to transform a matrix into a one-dimensional array.

Solution

Use `flatten`:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3],
```

```
                    [4, 5, 6],
                    [7, 8, 9]])

# Flatten matrix
matrix.flatten()
```

```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Discussion

`flatten` is a simple method to transform a matrix into a one-dimensional array. Alternatively, we can use `reshape` to create a row vector:

```
matrix.reshape(1, -1)
```

```
array([[1, 2, 3, 4, 5, 6, 7, 8, 9]])
```

One more common method to flatten arrays is the `ravel` method. Unlike `flatten` which returns a copy of the original array, `ravel` operates on the original object itself and is therefore slightly faster. It also lets us flatten lists of arrays, which we can't do with the `flatten` method. This operation is useful for flattening very large arrays and speeding up code.

```
# Create one matrix
matrix_a = np.array([[1, 2],
                     [3, 4]])

# Create a second matrix
matrix_b = np.array([[5, 6],
                     [7, 8]])

# Create a list of matrices
matrix_list = [matrix_a, matrix_b]

# Flatten the entire list of matrices
np.ravel(matrix_list)
```

```
array([1, 2, 3, 4, 5, 6, 7, 8])
```

1.13 Finding the Rank of a Matrix

Problem

You need to know the rank of a matrix.

Solution

Use NumPy's linear algebra method `matrix_rank`:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 1, 1],
                   [1, 1, 10],
                   [1, 1, 15]])
```

```
# Return matrix rank
np.linalg.matrix_rank(matrix)
```

```
2
```

Discussion

The rank of a matrix is the dimensions of the vector space spanned by its columns or rows. Finding the rank of a matrix is easy in NumPy thanks to `matrix_rank`.

See Also

- The Rank of a Matrix, CliffsNotes

1.14 Getting the Diagonal of a Matrix

Problem

You need to get the diagonal elements of a matrix.

Solution

Use `diagonal`:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3],
                   [2, 4, 6],
                   [3, 8, 9]])

# Return diagonal elements
matrix.diagonal()
```

```
array([1, 4, 9])
```

Discussion

NumPy makes getting the diagonal elements of a matrix easy with `diagonal`. It is also possible to get a diagonal off from the main diagonal by using the `offset` parameter:

```
# Return diagonal one above the main diagonal
matrix.diagonal(offset=1)
```

```
array([2, 6])
```

```
# Return diagonal one below the main diagonal
matrix.diagonal(offset=-1)
```

```
array([2, 8])
```

1.15 Calculating the Trace of a Matrix

Problem

You need to calculate the trace of a matrix.

Solution

Use `trace`:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3],
                   [2, 4, 6],
                   [3, 8, 9]])

# Return trace
matrix.trace()
```

```
14
```

Discussion

The trace of a matrix is the sum of the diagonal elements and is often used under the hood in machine learning methods. Given a NumPy multidimensional array, we can calculate the trace using `trace`. We can also return the diagonal of a matrix and calculate its sum:

```
# Return diagonal and sum elements
sum(matrix.diagonal())
```

```
14
```

See Also

- The Trace of a Square Matrix

1.16 Calculating Dot Products

Problem

You need to calculate the dot product of two vectors.

Solution

Use NumPy's `dot`:

```
# Load library
import numpy as np
```

```
# Create two vectors
vector_a = np.array([1,2,3])
vector_b = np.array([4,5,6])

# Calculate dot product
np.dot(vector_a, vector_b)
```

```
32
```

Discussion

The dot product of two vectors, a and b, is defined as:

$$\sum_{i=1}^{n} a_i b_i$$

where a_i is the ith element of vector a and b_i is the ith element of vector b. We can use NumPy's `dot` function to calculate the dot product. Alternatively, in Python 3.5+ we can use the new `@` operator:

```
# Calculate dot product
vector_a @ vector_b
```

```
32
```

See Also

- Vector dot product and vector length, Khan Academy
- Dot Product, Paul's Online Math Notes

1.17 Adding and Subtracting Matrices

Problem

You want to add or subtract two matrices.

Solution

Use NumPy's `add` and `subtract`:

```
# Load library
import numpy as np

# Create matrix
matrix_a = np.array([[1, 1, 1],
                     [1, 1, 1],
                     [1, 1, 2]])

# Create matrix
matrix_b = np.array([[1, 3, 1],
                     [1, 3, 1],
                     [1, 3, 8]])

# Add two matrices
```

```
np.add(matrix_a, matrix_b)
```

```
array([[ 2,  4,  2],
       [ 2,  4,  2],
       [ 2,  4, 10]])
```

```
# Subtract two matrices
np.subtract(matrix_a, matrix_b)
```

```
array([[ 0, -2,  0],
       [ 0, -2,  0],
       [ 0, -2, -6]])
```

Discussion

Alternatively, we can simply use the `+` and `-` operators:

```
# Add two matrices
matrix_a + matrix_b
```

```
array([[ 2,  4,  2],
       [ 2,  4,  2],
       [ 2,  4, 10]])
```

1.18 Multiplying Matrices

Problem

You want to multiply two matrices.

Solution

Use NumPy's `dot`:

```
# Load library
import numpy as np

# Create matrix
matrix_a = np.array([[1, 1],
                     [1, 2]])

# Create matrix
matrix_b = np.array([[1, 3],
                     [1, 2]])

# Multiply two matrices
np.dot(matrix_a, matrix_b)
```

```
array([[2, 5],
       [3, 7]])
```

Discussion

Alternatively, in Python 3.5+ we can use the `@` operator:

```
# Multiply two matrices
```

```
matrix_a @ matrix_b
```

```
array([[2, 5],
       [3, 7]])
```

If we want to do element-wise multiplication, we can use the `*` operator:

```
# Multiply two matrices element-wise
matrix_a * matrix_b
```

```
array([[1, 3],
       [1, 4]])
```

See Also

- Array vs. Matrix Operations, MathWorks

1.19 Inverting a Matrix

Problem

You want to calculate the inverse of a square matrix.

Solution

Use NumPy's linear algebra `inv` method:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 4],
                   [2, 5]])

# Calculate inverse of matrix
np.linalg.inv(matrix)
```

```
array([[-1.66666667,  1.33333333],
       [ 0.66666667, -0.33333333]])
```

Discussion

The inverse of a square matrix, $\mathbf{A}$, is a second matrix $\mathbf{A}^{-1}$, such that:

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$$

where $\mathbf{I}$ is the identity matrix. In NumPy we can use `linalg.inv` to calculate $\mathbf{A}^{-1}$ if it exists. To see this in action, we can multiply a matrix by its inverse and the result is the identity matrix:

```
# Multiply matrix and its inverse
matrix @ np.linalg.inv(matrix)
```

```
array([[ 1.,  0.],
```

```
       [ 0.,  1.]])
```

See Also

- Inverse of a Matrix

1.20 Generating Random Values

Problem

You want to generate pseudorandom values.

Solution

Use NumPy's `random`:

```
# Load library
import numpy as np

# Set seed
np.random.seed(0)

# Generate three random floats between 0.0 and 1.0
np.random.random(3)
```

```
array([ 0.5488135 ,  0.71518937,  0.60276338])
```

Discussion

NumPy offers a wide variety of means to generate random numbers, many more than can be covered here. In our solution we generated floats; however, it is also common to generate integers:

```
# Generate three random integers between 0 and 10
np.random.randint(0, 11, 3)
```

```
array([3, 7, 9])
```

Alternatively, we can generate numbers by drawing them from a distribution (note this is not technically random):

```
# Draw three numbers from a normal distribution with mean 0.0
# and standard deviation of 1.0
np.random.normal(0.0, 1.0, 3)
```

```
array([-1.42232584,  1.52006949, -0.29139398])
```

```
# Draw three numbers from a logistic distribution with mean 0.0 and scale of 1.0
np.random.logistic(0.0, 1.0, 3)
```

```
array([-0.98118713, -0.08939902,  1.46416405])
```

```
# Draw three numbers greater than or equal to 1.0 and less than 2.0
np.random.uniform(1.0, 2.0, 3)
```

```
array([ 1.47997717,  1.3927848 ,  1.83607876])
```

Finally, it can sometimes be useful to return the same random numbers multiple times to get predictable, repeatable results. We can do this by setting the “seed” (an integer) of the pseudorandom generator. Random processes with the same seed will always produce the same output. We will use seeds throughout this book so that the code you see in the book and the code you run on your computer produces the same results.

Chapter 2. Loading Data

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at feedback.mlpythoncookbook@gmail.com.

2.0 Introduction

The first step in any machine learning endeavor is to get the raw data into our system. The raw data might be a logfile, dataset file, database, or cloud blob store such as Amazon S3. Furthermore, often we will want to retrieve data from multiple sources.

The recipes in this chapter look at methods of loading data from a variety of sources, including CSV files and SQL databases. We also cover methods of generating simulated data with desirable properties for experimentation. Finally, while there are many ways to load data in the Python ecosystem, we will focus on using the pandas library's extensive set of methods for loading external data, and using scikit-learn—an open source machine learning library in Python—for generating simulated data.

2.1 Loading a Sample Dataset

Problem

You want to load a preexisting sample dataset from the scikit-learn library.

Solution

scikit-learn comes with a number of popular datasets for you to use:

```
# Load scikit-learn's datasets
from sklearn import datasets

# Load digits dataset
digits = datasets.load_digits()

# Create features matrix
features = digits.data

# Create target vector
target = digits.target

# View first observation
features[0]
```

```
array([  0.,   0.,   5.,  13.,   9.,   1.,   0.,   0.,   0.,   0.,  13.,
        15.,  10.,  15.,   5.,   0.,   0.,   3.,  15.,   2.,   0.,  11.,
         8.,   0.,   0.,   4.,  12.,   0.,   0.,   8.,   8.,   0.,   0.,
         5.,   8.,   0.,   0.,   9.,   8.,   0.,   0.,   4.,  11.,   0.,
         1.,  12.,   7.,   0.,   0.,   2.,  14.,   5.,  10.,  12.,   0.,
         0.,   0.,   0.,   6.,  13.,  10.,   0.,   0.,   0.])
```

Discussion

Often we do not want to go through the work of loading, transforming, and cleaning a real-world dataset before we can explore some machine learning algorithm or method. Luckily, scikit-learn comes with some common datasets we can quickly load. These datasets are often called "toy" datasets because they are far smaller and cleaner than a dataset we would see in the real world. Some popular sample datasets in scikit-learn are:

`load_iris`

Contains 150 observations on the measurements of Iris flowers. It is a good dataset for exploring classification algorithms.

`load_digits`

Contains 1,797 observations from images of handwritten digits. It is a good dataset for teaching image classification.

To see more details on any of the datasets above, you can print the `DESCR` attribute:

```
# Load scikit-learn's datasets
from sklearn import datasets

# Load digits dataset
digits = datasets.load_digits()

# Print the attribute
print(digits.DESCR)
```

```
.. _digits_dataset:

Optical recognition of handwritten digits dataset
--------------------------------------------------

**Data Set Characteristics:**

    :Number of Instances: 1797
    :Number of Attributes: 64
    :Attribute Information: 8x8 image of integer pixels in the range 0..16.
    :Missing Attribute Values: None
    :Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)
    :Date: July; 1998
...
```

See Also

- scikit-learn toy datasets
- The Digit Dataset

2.2 Creating a Simulated Dataset

Problem

You need to generate a dataset of simulated data.

Solution

scikit-learn offers many methods for creating simulated data. Of those, three methods are particularly useful: `make_regression`, `make_classification`, and `make_blobs`.

When we want a dataset designed to be used with linear regression, `make_regression` is a good choice:

```
# Load library
from sklearn.datasets import make_regression

# Generate features matrix, target vector, and the true coefficients
features, target, coefficients = make_regression(n_samples = 100,
                                                 n_features = 3,
                                                 n_informative = 3,
                                                 n_targets = 1,
                                                 noise = 0.0,
                                                 coef = True,
                                                 random_state = 1)

# View feature matrix and target vector
print('Feature Matrix\n', features[:3])
print('Target Vector\n', target[:3])
```

```
Feature Matrix
 [[ 1.29322588 -0.61736206 -0.11044703]
 [-2.793085    0.36633201  1.93752881]
 [ 0.80186103 -0.18656977  0.0465673 ]]
Target Vector
 [-10.37865986  25.5124503   19.67705609]
```

If we are interested in creating a simulated dataset for classification, we can use `make_classification`:

```
# Load library
from sklearn.datasets import make_classification

# Generate features matrix and target vector
features, target = make_classification(n_samples = 100,
                                       n_features = 3,
                                       n_informative = 3,
                                       n_redundant = 0,
                                       n_classes = 2,
                                       weights = [.25, .75],
                                       random_state = 1)

# View feature matrix and target vector
print('Feature Matrix\n', features[:3])
print('Target Vector\n', target[:3])
```

```
Feature Matrix
 [[ 1.06354768 -1.42632219  1.02163151]
 [ 0.23156977  1.49535261  0.33251578]
 [ 0.15972951  0.83533515 -0.40869554]]
Target Vector
 [1 0 0]
```

Finally, if we want a dataset designed to work well with clustering techniques, scikit-learn offers `make_blobs`:

```
# Load library
from sklearn.datasets import make_blobs

# Generate feature matrix and target vector
features, target = make_blobs(n_samples = 100,
                              n_features = 2,
                              centers = 3,
                              cluster_std = 0.5,
                              shuffle = True,
                              random_state = 1)

# View feature matrix and target vector
print('Feature Matrix\n', features[:3])
print('Target Vector\n', target[:3])
```

```
Feature Matrix
 [[ -1.22685609   3.25572052]
 [ -9.57463218  -4.38310652]
 [-10.71976941  -4.20558148]]
Target Vector
 [0 1 1]
```

Discussion

As might be apparent from the solutions, `make_regression` returns a feature matrix of float values and a target vector of float values, while `make_classification` and `make_blobs` return a feature matrix of float values and a target vector of integers representing membership in a class.

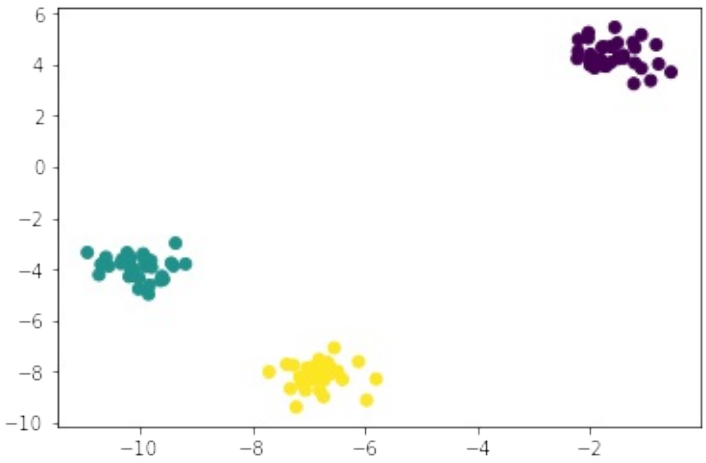
scikit-learn's simulated datasets offer extensive options to control the type of data generated. scikit-learn's documentation contains a full description of all the parameters, but a few are worth noting.

In `make_regression` and `make_classification`, `n_informative` determines the number of features that are used to generate the target vector. If `n_informative` is less than the total number of features (`n_features`), the resulting dataset will have redundant features that can be identified through feature selection techniques.

In addition, `make_classification` contains a `weights` parameter that allows us to simulate datasets with imbalanced classes. For example, `weights = [.25, .75]` would return a dataset with 25% of observations belonging to one class and 75% of observations belonging to a second class.

For `make_blobs`, the `centers` parameter determines the number of clusters generated. Using the `matplotlib` visualization library, we can visualize the clusters generated by `make_blobs`:

```
# Load library
import matplotlib.pyplot as plt

# View scatterplot
plt.scatter(features[:,0], features[:,1], c=target)
plt.show()
```



See Also

- `make_regression` documentation
- `make_classification` documentation
- `make_blobs` documentation

2.3 Loading a CSV File

Problem

You need to import a comma-separated values (CSV) file.

Solution

Use the pandas library's `read_csv` to load a local or hosted CSV file into a Pandas DataFrame:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/data.csv'

# Load dataset
dataframe = pd.read_csv(url)

# View first two rows
dataframe.head(2)
```

	integer	datetime	category
0	5	2015-01-01 00:00:00	0
1	5	2015-01-01 00:00:01	0

Discussion

There are two things to note about loading CSV files. First, it is often useful to take a quick look at the

contents of the file before loading. It can be very helpful to see how a dataset is structured beforehand and what parameters we need to set to load in the file. Second, `read_csv` has over 30 parameters and therefore the documentation can be daunting. Fortunately, those parameters are mostly there to allow it to handle a wide variety of CSV formats.

CSV files get their names from the fact that the values are literally separated by commas (e.g., one row might be `2,"2015-01-01 00:00:00",0`); however, it is common for "CSV" files to use other (termed "TSVs"). pandas' `sep` parameter allows us to define the delimiter used in the file. Although it is not always the case, a common formatting issue with CSV files is that the first line of the file is used to define column headers (e.g., `integer, datetime, category` in our solution). The `header` parameter allows us to specify if or where a header row exists. If a header row does not exist, we set `header=None`.

The `read_csv` function returns a Pandas DataFrame: a common and useful object for working with tabular data that we'll cover in more depth throughout this book.

2.4 Loading an Excel File

Problem

You need to import an Excel spreadsheet.

Solution

Use the pandas library's `read_excel` to load an Excel spreadsheet:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/data.xlsx'

# Load data
dataframe = pd.read_excel(url, sheet_name=0, header=1)

# View the first two rows
dataframe.head(2)
```

	5	2015-01-01 00:00:00	0
0	5	2015-01-01 00:00:01	0
1	9	2015-01-01 00:00:02	0

Discussion

This solution is similar to our solution for reading CSV files. The main difference is the additional parameter, `sheetname`, that specifies which sheet in the Excel file we wish to load. `sheetname` can accept both strings containing the name of the sheet and integers pointing to sheet positions (zero-indexed). If we need to load multiple sheets, include them as a list. For example, `sheetname=[0,1,2, "Monthly Sales"]` will return a dictionary of pandas DataFrames containing the first, second, and third

sheets and the sheet named `Monthly Sales`.

2.5 Loading a JSON File

Problem

You need to load a JSON file for data preprocessing.

Solution

The pandas library provides `read_json` to convert a JSON file into a pandas object:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/data.json'

# Load data
dataframe = pd.read_json(url, orient='columns')

# View the first two rows
dataframe.head(2)
```

	category	datetime	integer
0	0	2015-01-01 00:00:00	5
1	0	2015-01-01 00:00:01	5

Discussion

Importing JSON files into pandas is similar to the last few recipes we have seen. The key difference is the `orient` parameter, which indicates to pandas how the JSON file is structured. However, it might take some experimenting to figure out which argument (`split`, `records`, `index`, `columns`, and `values`) is the right one. Another helpful tool pandas offers is `json_normalize`, which can help convert semistructured JSON data into a pandas DataFrame.

See Also

- `json_normalize` documentation

2.6 Loading a parquet file

Problem

You need to load a parquet file.

Solution

The pandas `read_parquet` function allows us to read in parquet files:

```
# Load library
import pandas as pd

# Create URL
url = 'https://machine-learning-python-cookbook.s3.amazonaws.com/data.parquet'

# Load data
dataframe = pd.read_parquet(url)

# View the first two rows
dataframe.head(2)
```

	category	datetime	integer
0	0	2015-01-01 00:00:00	5
1	0	2015-01-01 00:00:01	5

Discussion

Paruqet is a popular data storage format in the large data space. It is often used with big data tools such as hadoop and spark. While Pyspark is outside the focus of this book, it's highly likely companies operating a large scale will use an efficient data storage format such as parquet and it's valuable to know how to read it into a dataframe and manipulate it.

See Also

- Apache Parquet Documentation

2.7 Loading a avro file

Problem

You need to load an avro file into a pandas dataframe.

Solution

The use the pandavro library's `read_avro` method:

```
# Load library
import pandavro as pdx

# Create URL
url = 'https://machine-learning-python-cookbook.s3.amazonaws.com/data.avro'

# Load data
dataframe = pdx.read_avro(url)

# View the first two rows
dataframe.head(2)
```

	category	datetime	integer

	category	datetime	integer
0	0	2015-01-01 00:00:00	5
1	0	2015-01-01 00:00:01	5

Discussion

Apache Avro is an open source, binary data format that relies on schemas for the data structure. At the time of writing it is not as common as parquet. However, large binary data formats such as avro, thrift and protocol buffers are growing in popularity due to the efficient nature of these formats. If you work with large data systems, you're likely to run into one of these formats (such as avro) in the near future.

See Also

- Apache Avro Docs

2.8 Loading a TFRecord file

Problem

You need to load a TFRecord file into a pandas dataframe.

Solution

	category	datetime	integer
0	0	2015-01-01 00:00:00	5
1	0	2015-01-01 00:00:01	5

Discussion

Like avro, TFRecord is a binary data format (in this case it is based on protocol buffers) - however it is specific to TensorFlow.

See Also

- TFRecord Docs

2.9 Querying a SQLite Database

Problem

You need to load data from a database using the structured query language (SQL).

Solution

pandas' `read_sql_query` allows us to make a SQL query to a database and load it:

```
# Load libraries
import pandas as pd
from sqlalchemy import create_engine

# Create a connection to the database
database_connection = create_engine('sqlite:///sample.db')

# Load data
dataframe = pd.read_sql_query('SELECT * FROM data', database_connection)

# View first two rows
dataframe.head(2)
```

	first_name	last_name	age	preTestScore	postTestScore
0	Jason	Miller	42	4	25
1	Molly	Jacobson	52	24	94

Discussion

SQL is the lingua franca for pulling data from databases. In this recipe we first use `create_engine` to define a connection to a SQL database engine called SQLite. Next we use pandas' `read_sql_query` to query that database using SQL and put the results in a DataFrame.

SQL is a language in its own right and, while beyond the scope of this book, it is certainly worth knowing for anyone wanting to learn machine learning. Our SQL query, `SELECT * FROM data`, asks the database to give us all columns (`*`) from the table called `data`.

Note that this is one of a few recipes in this book that will not run without extra code. Specifically, `create_engine('sqlite:///sample.db')` assumes that an SQLite database already exists.

See Also

- SQLite
- W3Schools SQL Tutorial

2.10 Querying a Remote SQL Database

Problem

You need to connect to, and read from, a remote SQL database.

Solution

Create a connection with `pymysql` and read it into a dataframe with `pandas`:

```
# Import libraries
import pymysql
import pandas as pd

# Create a DB connection
# Use the example below to start a DB instance
# https://github.com/kylegallatin/mysql-db-example
conn = pymysql.connect(
```

```
    host='localhost',
    user='root',
    password = "",
    db='db',
)

# Read the SQL query into a dataframe
dataframe = pd.read_sql("select * from data", conn)

# View the first 2 rows
dataframe.head(2)
```

	integer	datetime	category
0	5	2015-01-01 00:00:00	0
1	5	2015-01-01 00:00:01	0

Discussion

Out of all of the recipes presented in this chapter, this recipe is probably the one we will use most in the real world. While connecting and reading from an example `sqlite` database is useful, it's likely not representative of tables you'll need to connect to in the an enterprise environment. Most SQL instances that you'll connect to will require you to connect to the host and port of a remote machine, specifying a username and password for authentication. This example requires you to start a running SQL instance locally that mimics a remote server (the host `127.0.0.1` is actually your localhost) so that you can get a sense for the workflow.

See Also

- Pymysql Documentation
- Pandas Read SQL

2.11 Loading Data from a Google Sheet

Problem

You need to read data in directly from a Google Sheet.

Solution

Use Pandas read CSV and a URL that exports the Google Sheet as a CSV:

```
# Import libraries
import pandas as pd

# Google Sheet URL that downloads the sheet as a CSV
url = "https://docs.google.com/spreadsheets/d/1ehC-9otcAuitqnmWksqt1mOrTRCL38dv0K9UjhwzTOA/export?
format=csv"

# Read the CSV into a dataframe
dataframe = pd.read_csv(url)

# View the first 2 rows
```

```
dataframe.head(2)
```

	integer	datetime	category
0	5	2015-01-01 00:00:00	0
1	5	2015-01-01 00:00:01	0

Discussion

While Google Sheets can also easily be downloaded, it's sometimes helpful to be able to read them directly into Python without any intermediate steps. The `/export?format=csv` query parameter at the end of the URL above creates an endpoint from which we can either download the file or read it directly into pandas.

See Also

- Google Sheets API

2.12 Loading Data from an S3 Bucket

Problem

You need to read a CSV file from an S3 bucket you have access to.

Solution

Add storage options to pandas giving it access to the S3 object:

```
# Import libraries
import pandas as pd

# S3 path to csv
s3_uri = "s3://machine-learning-python-cookbook/data.csv"

# Set AWS credentails (replace with your own)
ACCESS_KEY_ID = "xxxxxxxxxxxx"
SECRET_ACCESS_KEY = "xxxxxxxxxxxxxxxx"

# Read the csv into a dataframe
dataframe = pd.read_csv(s3_uri,storage_options={
        "key": ACCESS_KEY_ID,
        "secret": SECRET_ACCESS_KEY,
  }
)

# View first two rows
dataframe.head(2)
```

	integer	datetime	category
0	5	2015-01-01 00:00:00	0
1	5	2015-01-01 00:00:01	0

Discussion

In the modern day, many enterprises keep data in cloud provider blob stores such as Amazon S3 or Google Cloud Storage (GCS). It's common for machine learning practitioners to connect to these sources in order to retrieve data. Although the S3 URI above (`s3://machine-learning-python-cookbook/data.csv`) is public, it still requires you to provide your own AWS access credentials in order to access it. It's worth noting that public objects also have http urls from which they can download files such as this one for the CSV file above.

See Also

- Amazon S3
- Setting up AWS access credentials

2.13 Loading Unstructured Data

Problem

You need to load in unstructured data like text or images.

Solution

Use the base Python `open` function to load the information:

```
# import libraries
import requests

# URL to download the txt file from
txt_url = "https://machine-learning-python-cookbook.s3.amazonaws.com/text.txt"

# Get the txt file
r = requests.get(txt_url)

# Write it to text.txt locally
with open('text.txt', 'wb') as f:
    f.write(r.content)

# Read in the file
with open('text.txt', 'r') as f:
    text = f.read()

# Print the content
print(text)
```

```
Hello there!
```

Discussion

While structured data can easily be read in from CSV, JSON, or various databases, unstructured data can be more challegning and may require custom processing down the line. Sometimes, it's helpful to open and read in files using Python's basic `open` function. This allows us to open files, and then read the content of that file.

See Also

- Python’s open function
- Context managers in Python

Chapter 3. Data Wrangling

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at feedback.mlpythoncookbook@gmail.com.

3.0 Introduction

Data wrangling is a broad term used, often informally, to describe the process of transforming raw data to a clean and organized format ready for use. For us, data wrangling is only one step in preprocessing our data, but it is an important step.

The most common data structure used to "wrangle" data is the data frame, which can be both intuitive and incredibly versatile. Data frames are tabular, meaning that they are based on rows and columns like you would see in a spreadsheet. Here is a data frame created from data about passengers on the *Titanic*:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data as a dataframe
dataframe = pd.read_csv(url)

# Show first 5 rows
dataframe.head(5)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.00	female	1	1
1	Allison, Miss Helen Loraine	1st	2.00	female	0	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.00	male	0	0
3	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25.00	female	0	1
4	Allison, Master Hudson Trevor	1st	0.92	male	1	0

There are three important things to notice in this data frame.

First, in a data frame each row corresponds to one observation (e.g., a passenger) and each column corresponds to one feature (gender, age, etc.). For example, by looking at the first observation we can see that Miss Elisabeth Walton Allen stayed in first class, was 29 years old, was female, and survived the disaster.

Second, each column contains a name (e.g., `Name`, `PClass`, `Age`) and each row contains an index number

(e.g., `0` for the lucky Miss Elisabeth Walton Allen). We will use these to select and manipulate observations and features.

Third, two columns, `Sex` and `SexCode`, contain the same information in different formats. In `Sex`, a woman is indicated by the string `female`, while in `SexCode`, a woman is indicated by using the integer `1`. We will want all our features to be unique, and therefore we will need to remove one of these columns.

In this chapter, we will cover a wide variety of techniques to manipulate data frames using the pandas library with the goal of creating a clean, well-structured set of observations for further preprocessing.

3.1 Creating a Data Frame

Problem

You want to create a new data frame.

Solution

pandas has many methods of creating a new DataFrame object. One easy method is to instantiate a `DataFrame` using a Python dictionary. In the dictionary, each key is a column name and the value is a list - where each item corresponds to a row:

```
# Load library
import pandas as pd

# Create a dictionary
dictionary = {
  "Name": ['Jacky Jackson', 'Steven Stevenson'],
  "Age": [38, 25],
  "Driver": [True, False]
}

# Create DataFrame
dataframe = pd.DataFrame(dictionary)

# Show DataFrame
dataframe
```

	Name	Age	Driver
0	Jacky Jackson	38	True
1	Steven Stevenson	25	False

It’s easy to add new columns to any dataframe using a list of values:

```
# Add a column for eye color
dataframe["Eyes"] = ["Brown", "Blue"]

# Show DataFrame
dataframe
```

	Name	Age	Driver	Eyes

0	Jacky Jackson	38	True	Brown
1	Steven Stevenson	25	False	Blue

Discussion

pandas offers what can feel like an infinite number of ways to create a DataFrame. In the real world, creating an empty DataFrame and then populating it will almost never happen. Instead, our DataFrames will be created from real data we have loading from other sources (e.g., a CSV file or database).

3.2 Getting Information about the Data

Problem

You want to view some characteristics of a DataFrame.

Solution

One of the easiest things we can do after loading the data is view the first few rows using `head`:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Show two rows
dataframe.head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

We can also take a look at the number of rows and columns:

```
# Show dimensions
dataframe.shape
```

```
(1313, 6)
```

We can get descriptive statistics for any numeric columns using `describe`:

```
# Show statistics
dataframe.describe()
```

	Age	Survived	SexCode
count	756.000000	1313.000000	1313.000000

mean	30.397989	0.342727	0.351866
std	14.259049	0.474802	0.477734
min	0.170000	0.000000	0.000000
25%	21.000000	0.000000	0.000000
50%	28.000000	0.000000	0.000000
75%	39.000000	1.000000	1.000000
max	71.000000	1.000000	1.000000

```
Additionally, the `info` method can also show some helpful information:
----
# Show info
dataframe.info()
----


<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1313 entries, 0 to 1312
Data columns (total 6 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Name      1313 non-null   object
 1   PClass    1313 non-null   object
 2   Age       756 non-null    float64
 3   Sex       1313 non-null   object
 4   Survived  1313 non-null   int64
 5   SexCode   1313 non-null   int64
dtypes: float64(1), int64(2), object(3)
memory usage: 61.7+ KB
```

Discussion

After we load some data, it is a good idea to understand how it is structured and what kind of information it contains. Ideally, we would view the full data directly. But with most real-world cases, the data could have thousands to hundreds of thousands to millions of rows and columns. Instead, we have to rely on pulling samples to view small slices and calculating summary statistics of the data.

In our solution, we are using a toy dataset of the passengers of the *Titanic* on her last voyage. Using `head` we can take a look at the first few rows (five by default) of the data. Alternatively, we can use `tail` to view the last few rows. With `shape` we can see how many rows and columns our DataFrame contains. And finally, with `describe` we can see some basic descriptive statistics for any numerical column.

It is worth noting that summary statistics do not always tell the full story. For example, pandas treats the columns `Survived` and `SexCode` as numeric columns because they contain 1s and 0s. However, in this case the numerical values represent categories. For example, if `Survived` equals 1, it indicates that the passenger survived the disaster. For this reason, some of the summary statistics provided don't make sense, such as the standard deviation of the `SexCode` column (an indicator of the passenger's gender).

3.3 Slicing DataFrames

Problem

You need to select individual data or slices of a DataFrame.

Solution

Use `loc` or `iloc` to select one or more rows or values:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Select first row
dataframe.iloc[0]
```

```
Name        Allen, Miss Elisabeth Walton
PClass                               1st
Age                                   29
Sex                               female
Survived                               1
SexCode                                1
Name: 0, dtype: object
```

We can use `:` to define a slice of rows we want, such as selecting the second, third, and fourth rows:

```
# Select four rows
dataframe.iloc[1:4]
```

	Name	PClass	Age	Sex	Survived	SexCode
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0
3	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25.0	female	0	1

We can even use it to get all rows up to a point, such as all rows up to and including the fourth row:

```
# Select three rows
dataframe.iloc[:4]
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0
3	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25.0	female	0	1

DataFrames do not need to be numerically indexed. We can set the index of a DataFrame to any value where the value is unique to each row. For example, we can set the index to be passenger names and then select rows using a name:

```
# Set index
```

```
dataframe = dataframe.set_index(dataframe['Name'])

# Show row
dataframe.loc['Allen, Miss Elisabeth Walton']
```

```
Name        Allen, Miss Elisabeth Walton
PClass                               1st
Age                                   29
Sex                               female
Survived                               1
SexCode                                1
Name: Allen, Miss Elisabeth Walton, dtype: object
```

Discussion

All rows in a pandas DataFrame have a unique index value. By default, this index is an integer indicating the row position in the DataFrame; however, it does not have to be. DataFrame indexes can be set to be unique alphanumeric strings or customer numbers. To select individual rows and slices of rows, pandas provides two methods:

- `loc` is useful when the index of the DataFrame is a label (e.g., a string).
- `iloc` works by looking for the position in the DataFrame. For example, `iloc[0]` will return the first row regardless of whether the index is an integer or a label.

It is useful to be comfortable with both `loc` and `iloc` since they will come up a lot during data cleaning.

3.4 Selecting Rows Based on Conditionals

Problem

You want to select DataFrame rows based on some condition.

Solution

This can be easily done in pandas. For example, if we wanted to select all the women on the *Titanic*:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Show top two rows where column 'sex' is 'female'
dataframe[dataframe['Sex'] == 'female'].head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

Take a second and look at the format of this solution. `dataframe['Sex'] == 'female'` is our

conditional statement; by wrapping that in `dataframe[]` we are telling pandas to "select all the rows in the DataFrame where the value of `dataframe['Sex']` is `'female'`. These conditions result in a Pandas series of booleans.

Multiple conditions are easy as well. For example, here we select all the rows where the passenger is a female 65 or older:

```
# Filter rows
dataframe[(dataframe['Sex'] == 'female') & (dataframe['Age'] >= 65)]
```

	Name	PClass	Age	Sex	Survived	SexCode
73	Crosby, Mrs Edward Gifford (Catherine Elizabet...	1st	69.0	female	1	1

Discussion

Conditionally selecting and filtering data is one of the most common tasks in data wrangling. You rarely want all the raw data from the source; instead, you are interested in only some subsection of it. For example, you might only be interested in stores in certain states or the records of patients over a certain age.

3.5 Sorting Values

Problem

You need to sort a dataframe by the values in a column.

Solution

Use the pandas `sort_values` function:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Sort the dataframe by Age
dataframe.sort_values(by=["Age"]).head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
763	Dean, Miss Elizabeth Gladys (Millvena)	3rd	0.17	female	1	1
751	Danbom, Master Gilbert Sigvard Emanuel	3rd	0.33	male	0	0

Discussion

During data analysis and exploration, it's often useful to sort a DataFrame by a particular column or set

of columns. The `by` argument to `sort_values` takes a list of columns by which to sort the DataFrame, and will sort based on the order of column names in the list.

By default, the `ascending` argument is set to `True` - so it will sort the values lowest to highest. If we wanted the oldest passengers instead of the youngest, we could set it so `False`.

3.6 Replacing Values

Problem

You need to replace values in a DataFrame.

Solution

pandas' `replace` is an easy way to find and replace values. For example, we can replace any instance of `"female"` in the `Sex` column with `"Woman"`:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Replace values, show two rows
dataframe['Sex'].replace("female", "Woman").head(2)
```

```
0    Woman
1    Woman
Name: Sex, dtype: object
```

We can also replace multiple values at the same time:

```
# Replace "female" and "male with "Woman" and "Man"
dataframe['Sex'].replace(["female", "male"], ["Woman", "Man"]).head(5)
```

```
0    Woman
1    Woman
2      Man
3    Woman
4      Man
Name: Sex, dtype: object
```

We can also find and replace across the entire `DataFrame` object by specifying the whole data frame instead of a single column:

```
# Replace values, show two rows
dataframe.replace(1, "One").head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29	female	One	One

1	Allison, Miss Helen Loraine	1st	2	female	0	One

`replace` also accepts regular expressions:

```
# Replace values, show two rows
dataframe.replace(r"1st", "First", regex=True).head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	First	29.0	female	1	1
1	Allison, Miss Helen Loraine	First	2.0	female	0	1

Discussion

`replace` is a tool we use to replace values that is simple and yet has the powerful ability to accept regular expressions.

3.7 Renaming Columns

Problem

You want to rename a column in a pandas DataFrame.

Solution

Rename columns using the `rename` method:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Rename column, show two rows
dataframe.rename(columns={'PClass': 'Passenger Class'}).head(2)
```

	Name	Passenger Class	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

Notice that the `rename` method can accept a dictionary as a parameter. We can use the dictionary to change multiple column names at once:

```
# Rename columns, show two rows
dataframe.rename(columns={'PClass': 'Passenger Class', 'Sex': 'Gender'}).head(2)
```

	Name	Passenger Class	Age	Gender	Survived	SexCode

0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

Discussion

Using `rename` with a dictionary as an argument to the `columns` parameter is my preferred way to rename columns because it works with any number of columns. If we want to rename all columns at once, this helpful snippet of code creates a dictionary with the old column names as keys and empty strings as values:

```
# Load library
import collections

# Create dictionary
column_names = collections.defaultdict(str)

# Create keys
for name in dataframe.columns:
    column_names[name]

# Show dictionary
column_names
```

```
defaultdict(str,
            {'Age': '',
             'Name': '',
             'PClass': '',
             'Sex': '',
             'SexCode': '',
             'Survived': ''})
```

3.8 Finding the Minimum, Maximum, Sum, Average, and Count

Problem

You want to find the min, max, sum, average, or count of a numeric column.

Solution

pandas comes with some built-in methods for commonly used descriptive statistics such as `min`, `max`, `mean`, `sum` and `count`:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Calculate statistics
print('Maximum:', dataframe['Age'].max())
print('Minimum:', dataframe['Age'].min())
print('Mean:', dataframe['Age'].mean())
print('Sum:', dataframe['Age'].sum())
print('Count:', dataframe['Age'].count())
```

```
Maximum: 71.0
Minimum: 0.17
Mean: 30.397989417989415
Sum: 22980.879999999997
Count: 756
```

Discussion

In addition to the statistics used in the solution, pandas offers variance (`var`), standard deviation (`std`), kurtosis (`kurt`), skewness (`skew`), standard error of the mean (`sem`), mode (`mode`), median (`median`), value counts, and a number of others.

Furthermore, we can also apply these methods to the whole DataFrame:

```
# Show counts
dataframe.count()
```

```
Name        1313
PClass      1313
Age          756
Sex         1313
Survived    1313
SexCode     1313
dtype: int64
```

3.9 Finding Unique Values

Problem

You want to select all unique values in a column.

Solution

Use `unique` to view an array of all unique values in a column:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Select unique values
dataframe['Sex'].unique()
```

```
array(['female', 'male'], dtype=object)
```

Alternatively, `value_counts` will display all unique values with the number of times each value appears:

```
# Show counts
dataframe['Sex'].value_counts()
```

```
male      851
```

```
female    462
Name: Sex, dtype: int64
```

Discussion

Both `unique` and `value_counts` are useful for manipulating and exploring categorical columns. Very often in categorical columns there will be classes that need to be handled in the data wrangling phase. For example, in the *Titanic* dataset, `PClass` is a column indicating the class of a passenger's ticket. There were three classes on the *Titanic*; however, if we use `value_counts` we can see a problem:

```
# Show counts
dataframe['PClass'].value_counts()
```

```
3rd    711
1st    322
2nd    279
*        1
Name: PClass, dtype: int64
```

While almost all passengers belong to one of three classes as expected, a single passenger has the class `*`. There are a number of strategies for handling this type of issue, which we will address in Chapter 5, but for now just realize that "extra" classes are common in categorical data and should not be ignored.

Finally, if we simply want to count the number of unique values, we can use `nunique`:

```
# Show number of unique values
dataframe['PClass'].nunique()
```

```
4
```

3.10 Handling Missing Values

Problem

You want to select missing values in a DataFrame.

Solution

`isnull` and `notnull` return booleans indicating whether a value is missing:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

## Select missing values, show two rows
dataframe[dataframe['Age'].isnull()].head(2)
```

Name	PClass	Age	Sex	Survived	SexCode

	Name	PClass	Age	Sex	Survived	SexCode
12	Aubert, Mrs Leontine Pauline	1st	NaN	female	1	1
13	Barkworth, Mr Algernon H	1st	NaN	male	1	0

Discussion

Missing values are a ubiquitous problem in data wrangling, yet many underestimate the difficulty of working with missing data. pandas uses NumPy's `NaN` ("Not A Number") value to denote missing values, but it is important to note that `NaN` is not fully implemented natively in pandas. For example, if we wanted to replace all strings containing `male` with missing values, we return an error:

```
# Attempt to replace values with NaN
dataframe['Sex'] = dataframe['Sex'].replace('male', NaN)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-7-5682d714f87d> in <module>()
      1 # Attempt to replace values with NaN
----> 2 dataframe['Sex'] = dataframe['Sex'].replace('male', NaN)


NameError: name 'NaN' is not defined
---------------------------------------------------------------------------
```

To have full functionality with `NaN` we need to import the NumPy library first:

```
# Load library
import numpy as np

# Replace values with NaN
dataframe['Sex'] = dataframe['Sex'].replace('male', np.nan)
```

Oftentimes a dataset uses a specific value to denote a missing observation, such as `NONE`, `-999`, or `..` pandas' `read_csv` includes a parameter allowing us to specify the values used to indicate missing values:

```
# Load data, set missing values
dataframe = pd.read_csv(url, na_values=[np.nan, 'NONE', -999])
```

We can also use pandas `fillna` function to impute the missing values of a column. Here, we show the places where `Age` is null using the `isna` function, and then fill those values with the mean age of passengers.

```
# Get a single null row
null_entry = dataframe[dataframe["Age"].isna()].head(1)

null_entry
```

	Name	PClass	Age	Sex	Survived	SexCode
12	Aubert, Mrs Leontine Pauline	1st	NaN	female	1	1

```
# Fill all null values with the mean Age of passengers
null_entry.fillna(dataframe["Age"].mean())
```

	Name	PClass	Age	Sex	Survived	SexCode
12	Aubert, Mrs Leontine Pauline	1st	30.397989	female	1	1

3.11 Deleting a Column

Problem

You want to delete a column from your DataFrame.

Solution

The best way to delete a column is to use `drop` with the parameter `axis=1` (i.e., the column axis):

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Delete column
dataframe.drop('Age', axis=1).head(2)
```

	Name	PClass	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	female	1	1
1	Allison, Miss Helen Loraine	1st	female	0	1

You can also use a list of column names as the main argument to drop multiple columns at once:

```
# Drop columns
dataframe.drop(['Age', 'Sex'], axis=1).head(2)
```

	Name	PClass	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	1	1
1	Allison, Miss Helen Loraine	1st	0	1

If a column does not have a name (which can sometimes happen), you can drop it by its column index using `dataframe.columns`:

```
# Drop column
dataframe.drop(dataframe.columns[1], axis=1).head(2)
```

	Name	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	29.0	female	1	1
1	Allison, Miss Helen Loraine	2.0	female	0	1

Discussion

`drop` is the idiomatic method of deleting a column. An alternative method is `del dataframe['Age']`, which works most of the time but is not recommended because of how it is called within pandas (the details of which are outside the scope of this book).

I recommend that you avoid using pandas' `inplace=True` argument. Many pandas methods include an `inplace` parameter, which when `True` edits the DataFrame directly. This can lead to problems in more complex data processing pipelines because we are treating the DataFrames as mutable objects (which they technically are). I recommend treating DataFrames as immutable objects. For example:

```
# Create a new DataFrame
dataframe_name_dropped = dataframe.drop(dataframe.columns[0], axis=1)
```

In this example, we are not mutating the DataFrame `dataframe` but instead are making a new DataFrame that is an altered version of `dataframe` called `dataframe_name_dropped`. If you treat your DataFrames as immutable objects, you will save yourself a lot of headaches down the road.

3.12 Deleting a Row

Problem

You want to delete one or more rows from a DataFrame.

Solution

Use a boolean condition to create a new DataFrame excluding the rows you want to delete:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Delete rows, show first two rows of output
dataframe[dataframe['Sex'] != 'male'].head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

Discussion

While technically you can use the `drop` method (for example, `df.drop([0, 1], axis=0)` to drop the first two rows), a more practical method is simply to wrap a boolean condition inside `df[]`. The reason is because we can use the power of conditionals to delete either a single row or (far more likely) many rows at once.

We can use boolean conditions to easily delete single rows by matching a unique value:

```
# Delete row, show first two rows of output
dataframe[dataframe['Name'] != 'Allison, Miss Helen Loraine'].head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0

And we can even use it to delete a single row by row index:

```
# Delete row, show first two rows of output
dataframe[dataframe.index != 0].head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0

3.13 Dropping Duplicate Rows

Problem

You want to drop duplicate rows from your DataFrame.

Solution

Use `drop_duplicates`, but be mindful of the parameters:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Drop duplicates, show first two rows of output
dataframe.drop_duplicates().head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

Discussion

A keen reader will notice that the solution didn't actually drop any rows:

```
# Show number of rows
```

```
print("Number Of Rows In The Original DataFrame:", len(dataframe))
print("Number Of Rows After Deduping:", len(dataframe.drop_duplicates()))
```

```
Number Of Rows In The Original DataFrame: 1313
Number Of Rows After Deduping: 1313
```

The reason is because `drop_duplicates` defaults to only dropping rows that match perfectly across all columns. Under this condition, every row in our DataFrame, `dataframe`, is actually unique. However, often we want to consider only a subset of columns to check for duplicate rows. We can accomplish this using the `subset` parameter:

```
# Drop duplicates
dataframe.drop_duplicates(subset=['Sex'])
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0

Take a close look at the preceding output: we told `drop_duplicates` to only consider any two rows with the same value for `Sex` to be duplicates and to drop them. Now we are left with a DataFrame of only two rows: one man and one woman. You might be asking why `drop_duplicates` decided to keep these two rows instead of two different rows. The answer is that `drop_duplicates` defaults to keeping the first occurrence of a duplicated row and dropping the rest. We can control this behavior using the `keep` parameter:

```
# Drop duplicates
dataframe.drop_duplicates(subset=['Sex'], keep='last')
```

	Name	PClass	Age	Sex	Survived	SexCode
1307	Zabour, Miss Tamini	3rd	NaN	female	0	1
1312	Zimmerman, Leo	3rd	29.0	male	0	0

A related method is `duplicated`, which returns a boolean series denoting if a row is a duplicate or not. This is a good option if you don't want to simply drop duplicates.

```
dataframe.duplicated()
```

```
0       False
1       False
2       False
3       False
4       False
        ...
1308    False
1309    False
1310    False
1311    False
1312    False
Length: 1313, dtype: bool
```

3.14 Grouping Rows by Values

Problem

You want to group individual rows according to some shared value.

Solution

`groupby` is one of the most powerful features in pandas:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Group rows by the values of the column 'Sex', calculate mean
# of each group
dataframe.groupby('Sex').mean()
```

	Age	Survived	SexCode
Sex			
female	29.396424	0.666667	1.0
male	31.014338	0.166863	0.0

Discussion

`groupby` is where data wrangling really starts to take shape. It is very common to have a DataFrame where each row is a person or an event and we want to group them according to some criterion and then calculate a statistic. For example, you can imagine a DataFrame where each row is an individual sale at a national restaurant chain and we want the total sales per restaurant. We can accomplish this by grouping rows by individual resturants and then calculating the sum of each group.

Users new to `groupby` often write a line like this and are confused by what is returned:

```
# Group rows
dataframe.groupby('Sex')
```

```
<pandas.core.groupby.DataFrameGroupBy object at 0x10efacf28>
```

Why didn't it return something more useful? The reason is because `groupby` needs to be paired with some operation we want to apply to each group, such as calculating an aggregate statistic (e.g., mean, median, sum). When talking about grouping we often use shorthand and say "group by gender," but that is incomplete. For grouping to be useful, we need to group by something and then apply a function to each of those groups:

```
# Group rows, count rows
dataframe.groupby('Survived')['Name'].count()
```

```
Survived
0    863
1    450
Name: Name, dtype: int64
```

Notice `Name` added after the `groupby`? That is because particular summary statistics are only meaningful to certain types of data. For example, while calculating the average age by gender makes sense, calculating the total age by gender does not. In this case we group the data into survived or not, then count the number of names (i.e., passengers) in each group.

We can also group by a first column, then group that grouping by a second column:

```
# Group rows, calculate mean
dataframe.groupby(['Sex','Survived'])['Age'].mean()
```

```
Sex     Survived
female  0           24.901408
        1           30.867143
male    0           32.320780
        1           25.951875
Name: Age, dtype: float64
```

3.15 Grouping Rows by Time

Problem

You need to group individual rows by time periods.

Solution

Use `resample` to group rows by chunks of time:

```
# Load libraries
import pandas as pd
import numpy as np

# Create date range
time_index = pd.date_range('06/06/2017', periods=100000, freq='30S')

# Create DataFrame
dataframe = pd.DataFrame(index=time_index)

# Create column of random values
dataframe['Sale_Amount'] = np.random.randint(1, 10, 100000)

# Group rows by week, calculate sum per week
dataframe.resample('W').sum()
```

	Sale_Amount
2017-06-11	86423
2017-06-18	101045
2017-06-25	100867
2017-07-02	100894
2017-07-09	100438

2017-07-16	10297

Discussion

Our standard *Titanic* dataset does not contain a datetime column, so for this recipe we have generated a simple DataFrame where each row is an individual sale. For each sale we know its date and time and its dollar amount (this data isn't realistic because every sale takes place precisely 30 seconds apart and is an exact dollar amount, but for the sake of simplicity let us pretend).

The raw data looks like this:

```
# Show three rows
dataframe.head(3)
```

	Sale_Amount
2017-06-06 00:00:00	7
2017-06-06 00:00:30	2
2017-06-06 00:01:00	7

Notice that the date and time of each sale is the index of the DataFrame; this is because `resample` requires the index to be datetime-like values.

Using `resample` we can group the rows by a wide array of time periods (offsets) and then we can calculate some statistic on each time group:

```
# Group by two weeks, calculate mean
dataframe.resample('2W').mean()
```

	Sale_Amount
2017-06-11	5.001331
2017-06-25	5.007738
2017-07-09	4.993353
2017-07-23	4.950481

```
# Group by month, count rows
dataframe.resample('M').count()
```

	Sale_Amount
2017-06-30	72000
2017-07-31	28000

You might notice that in the two outputs the datetime index is a date despite the fact that we are grouping by weeks and months, respectively. The reason is because by default `resample` returns the label of the right "edge" (the last label) of the time group. We can control this behavior using the `label` parameter:

```
# Group by month, count rows
```

```
dataframe.resample('M', label='left').count()
```

	Sale_Amount
2017-05-31	72000
2017-06-30	28000

See Also

- List of pandas time offset aliases

3.16 Aggregating Operations and Statistics

Problem

You need to aggregate an operation over each column (or a set of columns) in a dataframe.

Solution

Use the pandas `agg` method. Here, we can easily get the minimum value of every column:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Get the minimum of every column
dataframe.agg("min")
```

```
Name        Abbing, Mr Anthony
PClass                       *
Age                       0.17
Sex                     female
Survived                     0
SexCode                      0
dtype: object
```

Sometimes, we want to apply specific functions to specific sets of columns:

```
# Mean Age, min and max SexCode
dataframe.agg({"Age":["mean"], "SexCode":["min", "max"]})
```

	Age	SexCode
mean	30.397989	NaN
min	NaN	0.0
max	NaN	1.0

We can also apply aggregate functions to groups to get more specific, descriptive statistics:

```
# Number of people who survived and didn't survive in each class
dataframe.groupby(["PClass","Survived"]).agg({"Survived":["count"]}).reset_index()
```

	PClass	Survived
		count
0	*	0 1
1	1st	0 129
2	1st	1 193
3	2nd	0 160
4	2nd	1 119
5	3rd	0 573
6	3rd	1 138

Discussion

Aggregate functions are especially useful during exploratory data analysis to learn information about different subpopulations of data and the relationship between variables. By grouping the data applying aggregate statistics, you can view patterns in the data that may prove useful during the machine learning or feature engineering process. While visual charts are also helpful, it's often helpful to have such specific, descriptive statistics as a reference to better understand the data.

See Also

- pandas agg documentation

3.17 Looping Over a Column

Problem

You want to iterate over every element in a column and apply some action.

Solution

You can treat a pandas column like any other sequence in Python, and loop over it using the standard Python syntax:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Print first two names uppercased
for name in dataframe['Name'][0:2]:
    print(name.upper())
```

```
ALLEN, MISS ELISABETH WALTON
ALLISON, MISS HELEN LORAINE
```

Discussion

In addition to loops (often called `for` loops), we can also use list comprehensions:

```
# Show first two names uppercased
[name.upper() for name in dataframe['Name'][0:2]]
```

```
['ALLEN, MISS ELISABETH WALTON', 'ALLISON, MISS HELEN LORAINE']
```

Despite the temptation to fall back on `for` loops, a more Pythonic solution would use pandas' `apply` method, described in the next recipe.

3.18 Applying a Function Over All Elements in a Column

Problem

You want to apply some function over all elements in a column.

Solution

Use `apply` to apply a built-in or custom function on every element in a column:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Create function
def uppercase(x):
    return x.upper()

# Apply function, show two rows
dataframe['Name'].apply(uppercase)[0:2]
```

```
0    ALLEN, MISS ELISABETH WALTON
1     ALLISON, MISS HELEN LORAINE
Name: Name, dtype: object
```

Discussion

`apply` is a great way to do data cleaning and wrangling. It is common to write a function to perform some useful operation (separate first and last names, convert strings to floats, etc.) and then map that function to every element in a column.

3.19 Applying a Function to Groups

Problem

You have grouped rows using `groupby` and want to apply a function to each group.

Solution

Combine `groupby` and `apply`:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/sim_data/master/titanic.csv'

# Load data
dataframe = pd.read_csv(url)

# Group rows, apply function to groups
dataframe.groupby('Sex').apply(lambda x: x.count())
```

	Name	PClass	Age	Sex	Survived	SexCode
Sex						
female	462	462	288	462	462	462
male	851	851	468	851	851	851

Discussion

In Recipe 3.18 I mentioned `apply`. `apply` is particularly useful when you want to apply a function to groups. By combining `groupby` and `apply` we can calculate custom statistics or apply any function to each group separately.

3.20 Concatenating DataFrames

Problem

You want to concatenate two DataFrames.

Solution

Use `concat` with `axis=0` to concatenate along the row axis:

```
# Load library
import pandas as pd

# Create DataFrame
data_a = {'id': ['1', '2', '3'],
          'first': ['Alex', 'Amy', 'Allen'],
          'last': ['Anderson', 'Ackerman', 'Ali']}
dataframe_a = pd.DataFrame(data_a, columns = ['id', 'first', 'last'])

# Create DataFrame
data_b = {'id': ['4', '5', '6'],
          'first': ['Billy', 'Brian', 'Bran'],
          'last': ['Bonder', 'Black', 'Balwner']}
```

```
dataframe_b = pd.DataFrame(data_b, columns = ['id', 'first', 'last'])

# Concatenate DataFrames by rows
pd.concat([dataframe_a, dataframe_b], axis=0)
```

	id	first	last
0	1	Alex	Anderson
1	2	Amy	Ackerman
2	3	Allen	Ali
0	4	Billy	Bonder
1	5	Brian	Black
2	6	Bran	Balwner

You can use `axis=1` to concatenate along the column axis:

```
# Concatenate DataFrames by columns
pd.concat([dataframe_a, dataframe_b], axis=1)
```

	id	first	last	id	first	last
0	1	Alex	Anderson	4	Billy	Bonder
1	2	Amy	Ackerman	5	Brian	Black
2	3	Allen	Ali	6	Bran	Balwner

Discussion

Concatenating is not a word you hear much outside of computer science and programming, so if you have not heard it before, do not worry. The informal definition of *concatenate* is to glue two objects together. In the solution we glued together two small DataFrames using the `axis` parameter to indicate whether we wanted to stack the two DataFrames on top of each other or place them side by side.

3.21 Merging DataFrames

Problem

You want to merge two DataFrames.

Solution

To inner join, use `merge` with the `on` parameter to specify the column to merge on:

```
# Load library
import pandas as pd

# Create DataFrame
employee_data = {'employee_id': ['1', '2', '3', '4'],
                 'name': ['Amy Jones', 'Allen Keys', 'Alice Bees',
                 'Tim Horton']}
dataframe_employees = pd.DataFrame(employee_data, columns = ['employee_id',
                                                             'name'])
```

```
# Create DataFrame
sales_data = {'employee_id': ['3', '4', '5', '6'],
              'total_sales': [23456, 2512, 2345, 1455]}
dataframe_sales = pd.DataFrame(sales_data, columns = ['employee_id',
                                                      'total_sales'])

# Merge DataFrames
pd.merge(dataframe_employees, dataframe_sales, on='employee_id')
```

	employee_id	name	total_sales
0	3	Alice Bees	23456
1	4	Tim Horton	2512

`merge` defaults to inner joins. If we want to do an outer join, we can specify that with the `how` parameter:

```
# Merge DataFrames
pd.merge(dataframe_employees, dataframe_sales, on='employee_id', how='outer')
```

	employee_id	name	total_sales
0	1	Amy Jones	NaN
1	2	Allen Keys	NaN
2	3	Alice Bees	23456.0
3	4	Tim Horton	2512.0
4	5	NaN	2345.0
5	6	NaN	1455.0

The same parameter can be used to specify left and right joins:

```
# Merge DataFrames
pd.merge(dataframe_employees, dataframe_sales, on='employee_id', how='left')
```

	employee_id	name	total_sales
0	1	Amy Jones	NaN
1	2	Allen Keys	NaN
2	3	Alice Bees	23456.0
3	4	Tim Horton	2512.0

We can also specify the column name in each DataFrame to merge on:

```
# Merge DataFrames
pd.merge(dataframe_employees,
         dataframe_sales,
         left_on='employee_id',
         right_on='employee_id')
```

	employee_id	name	total_sales
0	3	Alice Bees	23456
1	4	Tim Horton	2512

If instead of merging on two columns we want to merge on the indexes of each DataFrame, we can replace the `left_on` and `right_on` parameters with `right_index=True` and `left_index=True`.

Discussion

Oftentimes, the data we need to use is complex; it doesn't always come in one piece. Instead in the real world, we're usually faced with disparate datasets, from multiple database queries or files. To get all that data into one place, we can load each data query or data file into pandas as individual DataFrames and then merge them together into a single DataFrame.

This process might be familiar to anyone who has used SQL, a popular language for doing merging operations (called *joins*). While the exact parameters used by pandas will be different, they follow the same general patterns used by other software languages and tools.

There are three aspects to specify with any `merge` operation. First, we have to specify the two DataFrames we want to merge together. In the solution we named them `dataframe_employees` and `dataframe_sales`. Second, we have to specify the name(s) of the columns to merge on—that is, the columns whose values are shared between the two DataFrames. For example, in our solution both DataFrames have a column named `employee_id`. To merge the two DataFrames we will match up the values in each DataFrame's `employee_id` column with each other. If these two columns use the same name, we can use the `on` parameter. However, if they have different names we can use `left_on` and `right_on`.

What is the left and right DataFrame? The simple answer is that the left DataFrame is the first one we specified in `merge` and the right DataFrame is the second one. This language comes up again in the next sets of parameters we will need.

The last aspect, and most difficult for some people to grasp, is the type of merge operation we want to conduct. This is specified by the `how` parameter. `merge` supports the four main types of joins:

Inner

> Return only the rows that match in both DataFrames (e.g., return any row with an `employee_id` value appearing in both `dataframe_employees` and `dataframe_sales`).

Outer

> Return all rows in both DataFrames. If a row exists in one DataFrame but not in the other DataFrame, fill NaN values for the missing values (e.g., return all rows in both `dataframe_employee` and `dataframe_sales`).

Left

> Return all rows from the left DataFrame but only rows from the right DataFrame that matched with the left DataFrame. Fill `NaN` values for the missing values (e.g., return all rows from `dataframe_employees` but only rows from `dataframe_sales` that have a value for `employee_id` that appears in `dataframe_employees`).

Right

> Return all rows from the right DataFrame but only rows from the left DataFrame that matched with

the right DataFrame. Fill `NaN` values for the missing values (e.g., return all rows from `dataframe_sales` but only rows from `dataframe_employees` that have a value for `employee_id` that appears in `dataframe_sales`).

If you did not understand all of that right now, I encourage you to play around with the `how` parameter in your code and see how it affects what `merge` returns.

See Also

- A Visual Explanation of SQL Joins
- pandas documentation on merging

Chapter 4. Handling Numerical Data

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at feedback.mlpythoncookbook@gmail.com.

4.0 Introduction

Quantitative data is the measurement of something—whether class size, monthly sales, or student scores. The natural way to represent these quantities is numerically (e.g., 29 students, $529,392 in sales). In this chapter, we will cover numerous strategies for transforming raw numerical data into features purpose-built for machine learning algorithms.

4.1 Rescaling a Feature

Problem

You need to rescale the values of a numerical feature to be between two values.

Solution

Use scikit-learn's `MinMaxScaler` to rescale a feature array:

```
# Load libraries
import numpy as np
from sklearn import preprocessing

# Create feature
feature = np.array([[-500.5],
                    [-100.1],
                    [0],
                    [100.1],
                    [900.9]])

# Create scaler
minmax_scale = preprocessing.MinMaxScaler(feature_range=(0, 1))

# Scale feature
scaled_feature = minmax_scale.fit_transform(feature)

# Show feature
scaled_feature
```

```
array([[ 0.        ],
       [ 0.28571429],
```

```
       [ 0.35714286],
       [ 0.42857143],
       [ 1.        ]])
```

Discussion

Rescaling is a common preprocessing task in machine learning. Many of the algorithms described later in this book will assume all features are on the same scale, typically 0 to 1 or –1 to 1. There are a number of rescaling techniques, but one of the simplest is called *min-max scaling*. Min-max scaling uses the minimum and maximum values of a feature to rescale values to within a range. Specifically, min-max calculates:

$$x'_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

where x is the feature vector, x_i is an individual element of feature x, and x'_i is the rescaled element. In our example, we can see from the outputted array that the feature has been successfully rescaled to between 0 and 1:

```
array([[ 0.        ],
       [ 0.28571429],
       [ 0.35714286],
       [ 0.42857143],
       [ 1.        ]])
```

scikit-learn's `MinMaxScaler` offers two options to rescale a feature. One option is to use `fit` to calculate the minimum and maximum values of the feature, then use `transform` to rescale the feature. The second option is to use `fit_transform` to do both operations at once. There is no mathematical difference between the two options, but there is sometimes a practical benefit to keeping the operations separate because it allows us to apply the same transformation to different *sets* of the data.

See Also

- Feature scaling, Wikipedia
- About Feature Scaling and Normalization, Sebastian Raschka

4.2 Standardizing a Feature

Problem

You want to transform a feature to have a mean of 0 and a standard deviation of 1.

Solution

scikit-learn's `StandardScaler` performs both transformations:

```
# Load libraries
```

```
import numpy as np
from sklearn import preprocessing

# Create feature
x = np.array([[-1000.1],
              [-200.2],
              [500.5],
              [600.6],
              [9000.9]])

# Create scaler
scaler = preprocessing.StandardScaler()

# Transform the feature
standardized = scaler.fit_transform(x)

# Show feature
standardized
```

```
array([[-0.76058269],
       [-0.54177196],
       [-0.35009716],
       [-0.32271504],
       [ 1.97516685]])
```

Discussion

A common alternative to min-max scaling discussed in Recipe 4.1 is rescaling of features to be approximately standard normally distributed. To achieve this, we use standardization to transform the data such that it has a mean, $\bar{x}$, of 0 and a standard deviation, σ, of 1. Specifically, each element in the feature is transformed so that:

$$x'_i = \frac{x_i - \overline{x}}{\sigma}$$

where x'_i is our standardized form of x_i. The transformed feature represents the number of standard deviations the original value is away from the feature's mean value (also called a *z-score* in statistics).

Standardization is a common go-to scaling method for machine learning preprocessing and in my experience is used more than min-max scaling. However, it depends on the learning algorithm. For example, principal component analysis often works better using standardization, while min-max scaling is often recommended for neural networks (both algorithms are discussed later in this book). As a general rule, I'd recommend defaulting to standardization unless you have a specific reason to use an alternative.

We can see the effect of standardization by looking at the mean and standard deviation of our solution's output:

```
# Print mean and standard deviation
print("Mean:", round(standardized.mean()))
print("Standard deviation:", standardized.std())
```

```
Mean: 0.0
Standard deviation: 1.0
```

If our data has significant outliers, it can negatively impact our standardization by affecting the feature's

mean and variance. In this scenario, it is often helpful to instead rescale the feature using the median and quartile range. In scikit-learn, we do this using the `RobustScaler` method:

```
# Create scaler
robust_scaler = preprocessing.RobustScaler()

# Transform feature
robust_scaler.fit_transform(x)
```

```
array([[ -1.87387612],
       [ -0.875     ],
       [  0.        ],
       [  0.125     ],
       [ 10.61488511]])
```

4.3 Normalizing Observations

Problem

You want to rescale the feature values of observations to have unit norm (a total length of 1).

Solution

Use `Normalizer` with a `norm` argument:

```
# Load libraries
import numpy as np
from sklearn.preprocessing import Normalizer

# Create feature matrix
features = np.array([[0.5, 0.5],
                     [1.1, 3.4],
                     [1.5, 20.2],
                     [1.63, 34.4],
                     [10.9, 3.3]])

# Create normalizer
normalizer = Normalizer(norm="l2")

# Transform feature matrix
normalizer.transform(features)
```

```
array([[ 0.70710678,  0.70710678],
       [ 0.30782029,  0.95144452],
       [ 0.07405353,  0.99725427],
       [ 0.04733062,  0.99887928],
       [ 0.95709822,  0.28976368]])
```

Discussion

Many rescaling methods (e.g., min-max scaling and standardization) operate on features; however, we can also rescale across individual observations. `Normalizer` rescales the values on individual observations to have unit norm (the sum of their lengths is 1). This type of rescaling is often used when we have many equivalent features (e.g., text classification when every word or *n*-word group is a feature).

`Normalizer` provides three norm options with Euclidean norm (often called L2) being the default argument:

$$\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

where x is an individual observation and x_n is that observation's value for the nth feature.

```
# Transform feature matrix
features_l2_norm = Normalizer(norm="l2").transform(features)

# Show feature matrix
features_l2_norm
```

```
array([[ 0.70710678,  0.70710678],
       [ 0.30782029,  0.95144452],
       [ 0.07405353,  0.99725427],
       [ 0.04733062,  0.99887928],
       [ 0.95709822,  0.28976368]])
```

Alternatively, we can specify Manhattan norm (L1):

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|.$$

```
# Transform feature matrix
features_l1_norm = Normalizer(norm="l1").transform(features)

# Show feature matrix
features_l1_norm
```

```
array([[ 0.5       ,  0.5       ],
       [ 0.24444444,  0.75555556],
       [ 0.06912442,  0.93087558],
       [ 0.04524008,  0.95475992],
       [ 0.76760563,  0.23239437]])
```

Intuitively, L2 norm can be thought of as the distance between two points in New York for a bird (i.e., a straight line), while L1 can be thought of as the distance for a human walking on the street (walk north one block, east one block, north one block, east one block, etc.), which is why it is called "Manhattan norm" or "Taxicab norm."

Practically, notice that `norm='l1'` rescales an observation's values so they sum to 1, which can sometimes be a desirable quality:

```
# Print sum
print("Sum of the first observation\'s values:",
   features_l1_norm[0, 0] + features_l1_norm[0, 1])
```

```
Sum of the first observation's values: 1.0
```

4.4 Generating Polynomial and Interaction Features

Problem

You want to create polynominal and interaction features.

Solution

Even though some choose to create polynomial and interaction features manually, scikit-learn offers a built-in method:

```
# Load libraries
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

# Create feature matrix
features = np.array([[2, 3],
                     [2, 3],
                     [2, 3]])

# Create PolynomialFeatures object
polynomial_interaction = PolynomialFeatures(degree=2, include_bias=False)

# Create polynomial features
polynomial_interaction.fit_transform(features)
```

```
array([[ 2.,  3.,  4.,  6.,  9.],
       [ 2.,  3.,  4.,  6.,  9.],
       [ 2.,  3.,  4.,  6.,  9.]])
```

The `degree` parameter determines the maximum degree of the polynomial. For example, `degree=2` will create new features raised to the second power:

$$x_1,\ x_2,\ x_1^2,\ x_1^x,\ x_2^2$$

while `degree=3` will create new features raised to the second and third power:

$$x_1,\ x_2,\ x_1^2,\ x_2^2,\ x_1^3,\ x_2^3,\ x_1^x,\ x_1^x,\ x_2^x$$

Furthermore, by default `PolynomialFeatures` includes interaction features:

$$x_1 x_2$$

We can restrict the features created to only interaction features by setting `interaction_only` to `True`:

```
interaction = PolynomialFeatures(degree=2,
              interaction_only=True, include_bias=False)

interaction.fit_transform(features)
```

```
array([[ 2.,  3.,  6.],
       [ 2.,  3.,  6.],
       [ 2.,  3.,  6.]])
```

Discussion

Polynomial features are often created when we want to include the notion that there exists a nonlinear

relationship between the features and the target. For example, we might suspect that the effect of age on the probability of having a major medical condition is not constant over time but increases as age increases. We can encode that nonconstant effect in a feature, *x*, by generating that feature's higher-order forms (x^2, x^3, etc.).

Additionally, often we run into situations where the effect of one feature is dependent on another feature. A simple example would be if we were trying to predict whether or not our coffee was sweet and we had two features: 1) whether or not the coffee was stirred and 2) if we added sugar. Individually, each feature does not predict coffee sweetness, but the combination of their effects does. That is, a coffee would only be sweet if the coffee had sugar and was stirred. The effects of each feature on the target (sweetness) are dependent on each other. We can encode that relationship by including an interaction feature that is the product of the individual features.

4.5 Transforming Features

Problem

You want to make a custom transformation to one or more features.

Solution

In scikit-learn, use `FunctionTransformer` to apply a function to a set of features:

```
# Load libraries
import numpy as np
from sklearn.preprocessing import FunctionTransformer

# Create feature matrix
features = np.array([[2, 3],
                     [2, 3],
                     [2, 3]])

# Define a simple function
def add_ten(x: int) -> int:
    return x + 10

# Create transformer
ten_transformer = FunctionTransformer(add_ten)

# Transform feature matrix
ten_transformer.transform(features)
```

```
array([[12, 13],
       [12, 13],
       [12, 13]])
```

We can create the same transformation in pandas using `apply`:

```
# Load library
import pandas as pd

# Create DataFrame
df = pd.DataFrame(features, columns=["feature_1", "feature_2"])

# Apply function
```

```
df.apply(add_ten)
```

	feature_1	feature_2
0	12	13
1	12	13
2	12	13

Discussion

It is common to want to make some custom transformations to one or more features. For example, we might want to create a feature that is the natural log of the values of the different feature. We can do this by creating a function and then mapping it to features using either scikit-learn's `FunctionTransformer` or pandas' `apply`. In the solution we created a very simple function, `add_ten`, which added 10 to each input, but there is no reason we could not define a much more complex function.

4.6 Detecting Outliers

Problem

You want to identify extreme observations.

Solution

Detecting outliers is unfortunately more of an art than a science. However, a common method is to assume the data is normally distributed and based on that assumption "draw" an ellipse around the data, classifying any observation inside the ellipse as an inlier (labeled as `1`) and any observation outside the ellipse as an outlier (labeled as `-1`):

```
# Load libraries
import numpy as np
from sklearn.covariance import EllipticEnvelope
from sklearn.datasets import make_blobs

# Create simulated data
features, _ = make_blobs(n_samples = 10,
                         n_features = 2,
                         centers = 1,
                         random_state = 1)

# Replace the first observation's values with extreme values
features[0,0] = 10000
features[0,1] = 10000

# Create detector
outlier_detector = EllipticEnvelope(contamination=.1)

# Fit detector
outlier_detector.fit(features)

# Predict outliers
outlier_detector.predict(features)
```

```
array([-1,  1,  1,  1,  1,  1,  1,  1,  1,  1])
```

In the above arrays, values of -1 refer to outliers whereas values of 1 refer to inliers. A major limitation of this approach is the need to specify a `contamination` parameter, which is the proportion of observations that are outliers—a value that we don't know. Think of `contamination` as our estimate of the cleanliness of our data. If we expect our data to have few outliers, we can set `contamination` to something small. However, if we believe that the data is very likely to have outliers, we can set it to a higher value.

Instead of looking at observations as a whole, we can instead look at individual features and identify extreme values in those features using interquartile range (IQR):

```
# Create one feature
feature = features[:,0]

# Create a function to return index of outliers
def indicies_of_outliers(x: int) -> np.array(int):
    q1, q3 = np.percentile(x, [25, 75])
    iqr = q3 - q1
    lower_bound = q1 - (iqr * 1.5)
    upper_bound = q3 + (iqr * 1.5)
    return np.where((x > upper_bound) | (x < lower_bound))

# Run function
indicies_of_outliers(feature)
```

```
(array([0]),)
```

IQR is the difference between the first and third quartile of a set of data. You can think of IQR as the spread of the bulk of the data, with outliers being observations far from the main concentration of data. Outliers are commonly defined as any value 1.5 IQRs less than the first quartile or 1.5 IQRs greater than the third quartile.

Discussion

There is no single best technique for detecting outliers. Instead, we have a collection of techniques all with their own advantages and disadvantages. Our best strategy is often trying multiple techniques (e.g., both `EllipticEnvelope` and IQR-based detection) and looking at the results as a whole.

If at all possible, we should take a look at observations we detect as outliers and try to understand them. For example, if we have a dataset of houses and one feature is number of rooms, is an outlier with 100 rooms really a house or is it actually a hotel that has been misclassified?

See Also

- Three ways to detect outliers (and the source of the IQR function used in this recipe)

4.7 Handling Outliers

Problem

You have outliers in your data that you want to identify, and then reduce their impact on the data

distribution.

Solution

Typically we have three strategies we can use to handle outliers. First, we can drop them:

```
# Load library
import pandas as pd

# Create DataFrame
houses = pd.DataFrame()
houses['Price'] = [534433, 392333, 293222, 4322032]
houses['Bathrooms'] = [2, 3.5, 2, 116]
houses['Square_Feet'] = [1500, 2500, 1500, 48000]

# Filter observations
houses[houses['Bathrooms'] < 20]
```

	Price	Bathrooms	Square_Feet
0	534433	2.0	1500
1	392333	3.5	2500
2	293222	2.0	1500

Second, we can mark them as outliers and include it as a feature:

```
# Load library
import numpy as np

# Create feature based on boolean condition
houses["Outlier"] = np.where(houses["Bathrooms"] < 20, 0, 1)

# Show data
houses
```

	Price	Bathrooms	Square_Feet	Outlier
0	534433	2.0	1500	0
1	392333	3.5	2500	0
2	293222	2.0	1500	0
3	4322032	116.0	48000	1

Finally, we can transform the feature to dampen the effect of the outlier:

```
# Log feature
houses["Log_Of_Square_Feet"] = [np.log(x) for x in houses["Square_Feet"]]

# Show data
houses
```

	Price	Bathrooms	Square_Feet	Outlier	Log_Of_Square_Feet
0	534433	2.0	1500	0	7.313220
1	392333	3.5	2500	0	7.824046
2	293222	2.0	1500	0	7.313220

3	4322032	116.0	48000	1	10.778956

Discussion

Similar to detecting outliers, there is no hard-and-fast rule for handling them. How we handle them should be based on two aspects. First, we should consider what makes them an outlier. If we believe they are errors in the data such as from a broken sensor or a miscoded value, then we might drop the observation or replace outlier values with `NaN` since we can't believe those values. However, if we believe the outliers are genuine extreme values (e.g., a house [mansion] with 200 bathrooms), then marking them as outliers or transforming their values is more appropriate.

Second, how we handle outliers should be based on our goal for machine learning. For example, if we want to predict house prices based on features of the house, we might reasonably assume the price for mansions with over 100 bathrooms is driven by a different dynamic than regular family homes. Furthermore, if we are training a model to use as part of an online home loan web application, we might assume that our potential users will not include billionaires looking to buy a mansion.

So what should we do if we have outliers? Think about why they are outliers, have an end goal in mind for the data, and, most importantly, remember that not making a decision to address outliers is itself a decision with implications.

One additional point: if you do have outliers standardization might not be appropriate because the mean and variance might be highly influenced by the outliers. In this case, use a rescaling method more robust against outliers like `RobustScaler`.

See Also

- `RobustScaler` documentation

4.8 Discretizating Features

Problem

You have a numerical feature and want to break it up into discrete bins.

Solution

Depending on how we want to break up the data, there are two techniques we can use. First, we can binarize the feature according to some threshold:

```
# Load libraries
import numpy as np
from sklearn.preprocessing import Binarizer

# Create feature
age = np.array([[6],
                [12],
                [20],
                [36],
                [65]])
```

```
# Create binarizer
binarizer = Binarizer(threshold=18)

# Transform feature
binarizer.fit_transform(age)
```

```
array([[0],
       [0],
       [1],
       [1],
       [1]])
```

Second, we can break up numerical features according to multiple thresholds:

```
# Bin feature
np.digitize(age, bins=[20,30,64])
```

```
array([[0],
       [0],
       [1],
       [2],
       [3]])
```

Note that the arguments for the `bins` parameter denote the left edge of each bin. For example, the `20` argument does not include the element with the value of 20, only the two values smaller than 20. We can switch this behavior by setting the parameter `right` to `True`:

```
# Bin feature
np.digitize(age, bins=[20,30,64], right=True)
```

```
array([[0],
       [0],
       [0],
       [2],
       [3]])
```

Discussion

Discretization can be a fruitful strategy when we have reason to believe that a numerical feature should behave more like a categorical feature. For example, we might believe there is very little difference in the spending habits of 19- and 20-year-olds, but a significant difference between 20- and 21-year-olds (the age in the United States when young adults can consume alcohol). In that example, it could be useful to break up individuals in our data into those who can drink alcohol and those who cannot. Similarly, in other cases it might be useful to discretize our data into three or more bins.

In the solution, we saw two methods of discretization—scikit-learn's `Binarizer` for two bins and NumPy's `digitize` for three or more bins—however, we can also use `digitize` to binarize features like `Binarizer` by only specifying a single threshold:

```
# Bin feature
np.digitize(age, bins=[18])
```

```
array([[0],
       [0],
       [1],
       [1],
```

```
       [1]])
```

See Also

- `digitize` documentation

4.9 Grouping Observations Using Clustering

Problem

You want to cluster observations so that similar observations are grouped together.

Solution

If you know that you have *k* groups, you can use k-means clustering to group similar observations and output a new feature containing each observation's group membership:

```
# Load libraries
import pandas as pd
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# Make simulated feature matrix
features, _ = make_blobs(n_samples = 50,
                         n_features = 2,
                         centers = 3,
                         random_state = 1)

# Create DataFrame
dataframe = pd.DataFrame(features, columns=["feature_1", "feature_2"])

# Make k-means clusterer
clusterer = KMeans(3, random_state=0)

# Fit clusterer
clusterer.fit(features)

# Predict values
dataframe["group"] = clusterer.predict(features)

# View first few observations
dataframe.head(5)
```

	feature_1	feature_2	group
0	–9.877554	–3.336145	0
1	–7.287210	–8.353986	2
2	–6.943061	–7.023744	2
3	–7.440167	–8.791959	2
4	–6.641388	–8.075888	2

Discussion

We are jumping ahead of ourselves a bit and will go much more in depth about clustering algorithms later in the book. However, I wanted to point out that we can use clustering as a preprocessing step.

Specifically, we use unsupervised learning algorithms like k-means to cluster observations into groups. The end result is a categorical feature with similar observations being members of the same group.

Don’t worry if you did not understand all of that right now: just file away the idea that clustering can be used in preprocessing. And if you really can’t wait, feel free to flip to [Link to Come] now.

4.10 Deleting Observations with Missing Values

Problem

You need to delete observations containing missing values.

Solution

Deleting observations with missing values is easy with a clever line of NumPy:

```
# Load library
import numpy as np

# Create feature matrix
features = np.array([[1.1, 11.1],
                     [2.2, 22.2],
                     [3.3, 33.3],
                     [4.4, 44.4],
                     [np.nan, 55]])

# Keep only observations that are not (denoted by ~) missing
features[~np.isnan(features).any(axis=1)]
```

```
array([[  1.1,  11.1],
       [  2.2,  22.2],
       [  3.3,  33.3],
       [  4.4,  44.4]])
```

Alternatively, we can drop missing observations using pandas:

```
# Load library
import pandas as pd

# Load data
dataframe = pd.DataFrame(features, columns=["feature_1", "feature_2"])

# Remove observations with missing values
dataframe.dropna()
```

	feature_1	feature_2
0	1.1	11.1
1	2.2	22.2
2	3.3	33.3
3	4.4	44.4

Discussion

Most machine learning algorithms cannot handle any missing values in the target and feature arrays. For this reason, we cannot ignore missing values in our data and must address the issue during preprocessing.

The simplest solution is to delete every observation that contains one or more missing values, a task quickly and easily accomplished using NumPy or pandas.

That said, we should be very reluctant to delete observations with missing values. Deleting them is the nuclear option, since our algorithm loses access to the information contained in the observation's non-missing values.

Just as important, depending on the cause of the missing values, deleting observations can introduce bias into our data. There are three types of missing data:

Missing Completely At Random (MCAR)

> The probability that a value is missing is independent of everything. For example, a survey respondent rolls a die before answering a question: if she rolls a six, she skips that question.

Missing At Random (MAR)

> The probability that a value is missing is not completely random, but depends on the information captured in other features. For example, a survey asks about gender identity and annual salary and women are more likely to skip the salary question; however, their nonresponse depends only on information we have captured in our gender identity feature.

Missing Not At Random (MNAR)

> The probability that a value is missing is not random and depends on information not captured in our features. For example, a survey asks about gender identity and women are more likely to skip the salary question, and we do not have a gender identity feature in our data.

It is sometimes acceptable to delete observations if they are MCAR or MAR. However, if the value is MNAR, the fact that a value is missing is itself information. Deleting MNAR observations can inject bias into our data because we are removing observations produced by some unobserved systematic effect.

See Also

- Identifying the Three Types of Missing Data
- Missing-Data Imputation

4.11 Imputing Missing Values

Problem

You have missing values in your data and want to impute them via some generic method or prediction.

Solution

You can impute missing values using k-means nearest neighbors (KNN) or scikit-learn's Simple Imputer class. If you have a small amount of data, predict and impute the missing values using k-nearest neighbors (KNN):

```
# Load libraries
import numpy as np
from sklearn.impute import KNNImputer
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs

# Make a simulated feature matrix
features, _ = make_blobs(n_samples = 1000,
                         n_features = 2,
                         random_state = 1)

# Standardize the features
scaler = StandardScaler()
standardized_features = scaler.fit_transform(features)

# Replace the first feature's first value with a missing value
true_value = standardized_features[0,0]
standardized_features[0,0] = np.nan



# Predict the missing values in the feature matrix
knn_imputer = KNNImputer(n_neighbors=5)
features_knn_imputed = knn_imputer.fit_transform(standardized_features)

# Compare true and imputed values
print("True Value:", true_value)
print("Imputed Value:", features_knn_imputed[0,0])


True Value: 0.8730186114
Imputed Value: 1.09553327131
```

Alternatively, we can use scikit-learn's `SimpleImputer` class from the `imputer` module to fill in missing values with the feature's mean, median, or most frequent value. However, we will typically get worse results than KNN:

```
# Load library
import numpy as np
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs

# Make a simulated feature matrix
features, _ = make_blobs(n_samples = 1000,
                         n_features = 2,
                         random_state = 1)

# Standardize the features
scaler = StandardScaler()
standardized_features = scaler.fit_transform(features)

# Replace the first feature's first value with a missing value
true_value = standardized_features[0,0]
standardized_features[0,0] = np.nan

# Create imputer using the "mean" strategy
mean_imputer = SimpleImputer(strategy="mean")

# Impute values
features_mean_imputed = mean_imputer.fit_transform(features)
```

```
# Compare true and imputed values
print("True Value:", true_value)
print("Imputed Value:", features_mean_imputed[0,0])
```

```
True Value: 0.8730186114
Imputed Value: -3.05837272461
```

Discussion

There are two main strategies for replacing missing data with substitute values, each of which has strengths and weaknesses. First, we can use machine learning to predict the values of the missing data. To do this we treat the feature with missing values as a target vector and use the remaining subset of features to predict missing values. While we can use a wide range of machine learning algorithms to impute values, a popular choice is KNN. KNN is addressed in depth later in [Link to Come], but the short explanation is that the algorithm uses the *k* nearest observations (according to some distance metric) to predict the missing value. In our solution we predicted the missing value using the five closest observations.

The downside to KNN is that in order to know which observations are the closest to the missing value, it needs to calculate the distance between the missing value and every single observation. This is reasonable in smaller datasets, but quickly becomes problematic if a dataset has millions of observations. In such cases, approximate nearest-neighbors (ANN) is a more feasible approach. We will discuss ANN later in the book.

An alternative and more scalable strategy than KNN is to fill in the missing values of numerical data with the mean, median or mode. For example, in our solution we used scikit-learn to fill in missing values with a feature's mean value. The imputed value is often not as close to the true value as when we used KNN, but we can scale mean-filling to data containing millions of observations more easily.

If we use imputation, it is a good idea to create a binary feature indicating whether or not the observation contains an imputed value.

See Also

- Imputation of missing values with scikit-learn
- A Study of K-Nearest Neighbour as an Imputation Method

Chapter 5. Handling Categorical Data

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at feedback.mlpythoncookbook@gmail.com.

5.0 Introduction

It is often useful to measure objects not in terms of their quantity but in terms of some quality. We frequently represent qualitative information in categories such as gender, colors, or brand of car. However, not all categorical data is the same. Sets of categories with no intrinsic ordering is called *nominal*. Examples of nominal categories include:

- Blue, Red, Green
- Man, Woman
- Banana, Strawberry, Apple

In contrast, when a set of categories has some natural ordering we refer to it as *ordinal*. For example:

- Low, Medium, High
- Young, Old
- Agree, Neutral, Disagree

Furthermore, categorical information is often represented in data as a vector or column of strings (e.g., `"Maine"`, `"Texas"`, `"Delaware"`). The problem is that most machine learning algorithms require inputs be numerical values.

The k-nearest neighbor algorithm is an example of an algorithm that requires numerical data. One step in the algorithm is calculating the distances between observations—often using Euclidean distance:

$$\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

where x and y are two observations and subscript i denotes the value for the observations' ith feature. However, the distance calculation obviously is impossible if the value of x_i is a string (e.g., `"Texas"`). Instead, we need to convert the string into some numerical format so that it can be inputted into the Euclidean distance equation. Our goal is to transform the data in a way that properly captures the information in the categories (ordinality, relative intervals between categories, etc.). In this chapter we will cover techniques for making this transformation as well as overcoming other challenges often

encountered when handling categorical data.

5.1 Encoding Nominal Categorical Features

Problem

You have a feature with nominal classes that has no intrinsic ordering (e.g., apple, pear, banana) and you want to encode the feature into numerical values.

Solution

One-hot encode the feature using scikit-learn's `LabelBinarizer`:

```
# Import libraries
import numpy as np
from sklearn.preprocessing import LabelBinarizer, MultiLabelBinarizer

# Create feature
feature = np.array([["Texas"],
                    ["California"],
                    ["Texas"],
                    ["Delaware"],
                    ["Texas"]])

# Create one-hot encoder
one_hot = LabelBinarizer()

# One-hot encode feature
one_hot.fit_transform(feature)
```

```
array([[0, 0, 1],
       [1, 0, 0],
       [0, 0, 1],
       [0, 1, 0],
       [0, 0, 1]])
```

We can use the `classes_` attribute to output the classes:

```
# View feature classes
one_hot.classes_
```

```
array(['California', 'Delaware', 'Texas'],
      dtype='<U10')
```

If we want to reverse the one-hot encoding, we can use `inverse_transform`:

```
# Reverse one-hot encoding
one_hot.inverse_transform(one_hot.transform(feature))
```

```
array(['Texas', 'California', 'Texas', 'Delaware', 'Texas'],
      dtype='<U10')
```

We can even use pandas to one-hot encode the feature:

```
# Import library
```

```
import pandas as pd

# Create dummy variables from feature
pd.get_dummies(feature[:,0])
```

	California	Delaware	Texas
0	0	0	1
1	1	0	0
2	0	0	1
3	0	1	0
4	0	0	1

One helpful ability of scikit-learn is to handle a situation where each observation lists multiple classes:

```
# Create multiclass feature
multiclass_feature = [("Texas", "Florida"),
                      ("California", "Alabama"),
                      ("Texas", "Florida"),
                      ("Delware", "Florida"),
                      ("Texas", "Alabama")]

# Create multiclass one-hot encoder
one_hot_multiclass = MultiLabelBinarizer()

# One-hot encode multiclass feature
one_hot_multiclass.fit_transform(multiclass_feature)
```

```
array([[0, 0, 0, 1, 1],
       [1, 1, 0, 0, 0],
       [0, 0, 0, 1, 1],
       [0, 0, 1, 1, 0],
       [1, 0, 0, 0, 1]])
```

Once again, we can see the classes with the `classes_` method:

```
# View classes
one_hot_multiclass.classes_
```

```
array(['Alabama', 'California', 'Delware', 'Florida', 'Texas'], dtype=object)
```

Discussion

We might think the proper strategy is to assign each class a numerical value (e.g., Texas = 1, California = 2). However, when our classes have no intrinsic ordering (e.g., Texas isn't "less" than California), our numerical values erroneously create an ordering that is not present.

The proper strategy is to create a binary feature for each class in the original feature. This is often called *one-hot encoding* (in machine learning literature) or *dummying* (in statistical and research literature). Our solution's feature was a vector containing three classes (i.e., Texas, California, and Delaware). In one-hot encoding, each class becomes its own feature with 1s when the class appears and 0s otherwise. Because our feature had three classes, one-hot encoding returned three binary features (one for each class). By using one-hot encoding we can capture the membership of an observation in a class while preserving the notion that the class lacks any sort of hierarchy.

Finally, it is often recommended that after one-hot encoding a feature, we drop one of the one-hot encoded features in the resulting matrix to avoid linear dependence.

See Also

- Dummy Variable Trap, Algosome
- Dropping one of the columns when using one-hot encoding, CrossValidated

5.2 Encoding Ordinal Categorical Features

Problem

You have an ordinal categorical feature (e.g., high, medium, low) and you want to transform them into numerical values.

Solution

Use pandas DataFrame's `replace` method to transform string labels to numerical equivalents:

```
# Load library
import pandas as pd

# Create features
dataframe = pd.DataFrame({"Score": ["Low", "Low", "Medium", "Medium", "High"]})

# Create mapper
scale_mapper = {"Low":1,
                "Medium":2,
                "High":3}

# Replace feature values with scale
dataframe["Score"].replace(scale_mapper)
```

```
0    1
1    1
2    2
3    2
4    3
Name: Score, dtype: int64
```

Discussion

Often we have a feature with classes that have some kind of natural ordering. A famous example is the Likert scale:

- Strongly Agree
- Agree
- Neutral
- Disagree
- Strongly Disagree

When encoding the feature for use in machine learning, we need to transform the ordinal classes into numerical values that maintain the notion of ordering. The most common approach is to create a dictionary that maps the string label of the class to a number and then apply that map to the feature.

It is important that our choice of numeric values is based on our prior information on the ordinal classes. In our solution, `high` is literally three times larger than `low`. This is fine in many instances, but can break down if the assumed intervals between the classes are not equal:

```
dataframe = pd.DataFrame({"Score": ["Low",
                                    "Low",
                                    "Medium",
                                    "Medium",
                                    "High",
                                    "Barely More Than Medium"]})

scale_mapper = {"Low":1,
                "Medium":2,
                "Barely More Than Medium": 3,
                "High":4}

dataframe["Score"].replace(scale_mapper)
```

```
0    1
1    1
2    2
3    2
4    4
5    3
Name: Score, dtype: int64
```

In this example, the distance between `Low` and `Medium` is the same as the distance between `Medium` and `Barely More Than Medium`, which is almost certainly not accurate. The best approach is to be conscious about the numerical values mapped to classes:

```
scale_mapper = {"Low":1,
                "Medium":2,
                "Barely More Than Medium": 2.1,
                "High":3}

dataframe["Score"].replace(scale_mapper)
```

```
0    1.0
1    1.0
2    2.0
3    2.0
4    3.0
5    2.1
Name: Score, dtype: float64
```

5.3 Encoding Dictionaries of Features

Problem

You have a dictionary and want to convert it into a feature matrix.

Solution

Use `DictVectorizer`:

```
# Import library
from sklearn.feature_extraction import DictVectorizer

# Create dictionary
data_dict = [{"Red": 2, "Blue": 4},
             {"Red": 4, "Blue": 3},
             {"Red": 1, "Yellow": 2},
             {"Red": 2, "Yellow": 2}]

# Create dictionary vectorizer
dictvectorizer = DictVectorizer(sparse=False)

# Convert dictionary to feature matrix
features = dictvectorizer.fit_transform(data_dict)

# View feature matrix
features
```

```
array([[ 4.,  2.,  0.],
       [ 3.,  4.,  0.],
       [ 0.,  1.,  2.],
       [ 0.,  2.,  2.]])
```

By default `DictVectorizer` outputs a sparse matrix that only stores elements with a value other than 0. This can be very helpful when we have massive matrices (often encountered in natural language processing) and want to minimize the memory requirements. We can force `DictVectorizer` to output a dense matrix using `sparse=False`.

We can get the names of each generated feature using the `get_feature_names` method:

```
# Get feature names
feature_names = dictvectorizer.get_feature_names()

# View feature names
feature_names
```

```
['Blue', 'Red', 'Yellow']
```

While not necessary, for the sake of illustration we can create a pandas DataFrame to view the output better:

```
# Import library
import pandas as pd

# Create dataframe from features
pd.DataFrame(features, columns=feature_names)
```

	Blue	Red	Yellow
0	4.0	2.0	0.0
1	3.0	4.0	0.0
2	0.0	1.0	2.0
3	0.0	2.0	2.0

Discussion

A dictionary is a popular data structure used by many programming languages; however, machine learning algorithms expect the data to be in the form of a matrix. We can accomplish this using scikit-learn's `DictVectorizer`.

This is a common situation when working with natural language processing. For example, we might have a collection of documents and for each document we have a dictionary containing the number of times every word appears in the document. Using `DictVectorizer`, we can easily create a feature matrix where every feature is the number of times a word appears in each document:

```
# Create word counts dictionaries for four documents
doc_1_word_count = {"Red": 2, "Blue": 4}
doc_2_word_count = {"Red": 4, "Blue": 3}
doc_3_word_count = {"Red": 1, "Yellow": 2}
doc_4_word_count = {"Red": 2, "Yellow": 2}

# Create list
doc_word_counts = [doc_1_word_count,
                   doc_2_word_count,
                   doc_3_word_count,
                   doc_4_word_count]

# Convert list of word count dictionaries into feature matrix
dictvectorizer.fit_transform(doc_word_counts)
```

```
array([[ 4.,  2.,  0.],
       [ 3.,  4.,  0.],
       [ 0.,  1.,  2.],
       [ 0.,  2.,  2.]])
```

In our toy example there are only three unique words (`Red`, `Yellow`, `Blue`) so there are only three features in our matrix; however, you can imagine that if each document was actually a book in a university library our feature matrix would be very large (and then we would want to set `spare` to `True`).

See Also

- How to use dictionaries in Python
- SciPy Sparse Matrices

5.4 Imputing Missing Class Values

Problem

You have a categorical feature containing missing values that you want to replace with predicted values.

Solution

The ideal solution is to train a machine learning classifier algorithm to predict the missing values, commonly a k-nearest neighbors (KNN) classifier:

```
# Load libraries
import numpy as np
```

```
from sklearn.neighbors import KNeighborsClassifier

# Create feature matrix with categorical feature
X = np.array([[0, 2.10, 1.45],
              [1, 1.18, 1.33],
              [0, 1.22, 1.27],
              [1, -0.21, -1.19]])

# Create feature matrix with missing values in the categorical feature
X_with_nan = np.array([[np.nan, 0.87, 1.31],
                       [np.nan, -0.67, -0.22]])

# Train KNN learner
clf = KNeighborsClassifier(3, weights='distance')
trained_model = clf.fit(X[:,1:], X[:,0])

# Predict missing values' class
imputed_values = trained_model.predict(X_with_nan[:,1:])

# Join column of predicted class with their other features
X_with_imputed = np.hstack((imputed_values.reshape(-1,1), X_with_nan[:,1:]))

# Join two feature matrices
np.vstack((X_with_imputed, X))
```

```
array([[ 0.  ,  0.87,  1.31],
       [ 1.  , -0.67, -0.22],
       [ 0.  ,  2.1 ,  1.45],
       [ 1.  ,  1.18,  1.33],
       [ 0.  ,  1.22,  1.27],
       [ 1.  , -0.21, -1.19]])
```

An alternative solution is to fill in missing values with the feature's most frequent value:

```
from sklearn.impute import SimpleImputer

# Join the two feature matrices
X_complete = np.vstack((X_with_nan, X))

imputer = SimpleImputer(strategy='most_frequent')

imputer.fit_transform(X_complete)
```

```
array([[ 0.  ,  0.87,  1.31],
       [ 0.  , -0.67, -0.22],
       [ 0.  ,  2.1 ,  1.45],
       [ 1.  ,  1.18,  1.33],
       [ 0.  ,  1.22,  1.27],
       [ 1.  , -0.21, -1.19]])
```

Discussion

When we have missing values in a categorical feature, our best solution is to open our toolbox of machine learning algorithms to predict the values of the missing observations. We can accomplish this by treating the feature with the missing values as the target vector and the other features as the feature matrix. A commonly used algorithm is KNN (discussed in depth later in this book), which assigns to the missing value the most frequent class of the *k* nearest observations.

Alternatively, we can fill in missing values with the most frequent class of the feature. While less sophisticated than KNN, it is much more scalable to larger data. In either case, it is advisable to include a binary feature indicating which observations contain imputed values.

See Also

- Imputation of missing values with scikit-learn
- Overcoming Missing Values in a Random Forest Classifier
- A Study of K-Nearest Neighbour as an Imputation Method

5.5 Handling Imbalanced Classes

Problem

You have a target vector with highly imbalanced classes, and you want to make adjustments so that you can handle the class imbalance.

Solution

Collect more data. If that isn't possible, change the metrics used to evaluate your model. If that doesn't work, consider using a model's built-in class weight parameters (if available), downsampling, or upsampling. We cover evaluation metrics in a later chapter, so for now let us focus on class weight parameters, downsampling, and upsampling.

To demonstrate our solutions, we need to create some data with imbalanced classes. Fisher's Iris dataset contains three balanced classes of 50 observations, each indicating the species of flower (*Iris setosa*, *Iris virginica*, and *Iris versicolor*). To unbalance the dataset, we remove 40 of the 50 *Iris setosa* observations and then merge the *Iris virginica* and *Iris versicolor* classes. The end result is a binary target vector indicating if an observation is an *Iris setosa* flower or not. The result is 10 observations of *Iris setosa* (class 0) and 100 observations of not Iris setosa (class 1):

```
# Load libraries
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris

# Load iris data
iris = load_iris()

# Create feature matrix
features = iris.data

# Create target vector
target = iris.target

# Remove first 40 observations
features = features[40:,:]
target = target[40:]

# Create binary target vector indicating if class 0
target = np.where((target == 0), 0, 1)

# Look at the imbalanced target vector
target
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

```
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

Many algorithms in scikit-learn offer a parameter to weight classes during training to counteract the effect of their imbalance. While we have not covered it yet, `RandomForestClassifier` is a popular classification algorithm and includes a `class_weight` parameter. You can pass an argument specifying the desired class weights explicitly:

```
# Create weights
weights = {0: .9, 1: 0.1}

# Create random forest classifier with weights
RandomForestClassifier(class_weight=weights)
```

```
RandomForestClassifier(class_weight={0: 0.9, 1: 0.1})
```

Or you can pass `balanced`, which automatically creates weights inversely proportional to class frequencies:

```
# Train a random forest with balanced class weights
RandomForestClassifier(class_weight="balanced")
```

```
RandomForestClassifier(class_weight='balanced')
```

Alternatively, we can downsample the majority class or upsample the minority class. In downsampling, we randomly sample without replacement from the majority class (i.e., the class with more observations) to create a new subset of observations equal in size to the minority class. For example, if the minority class has 10 observations, we will randomly select 10 observations from the majority class and use those 20 observations as our data. Here we do exactly that using our unbalanced Iris data:

```
# Indicies of each class' observations
i_class0 = np.where(target == 0)[0]
i_class1 = np.where(target == 1)[0]

# Number of observations in each class
n_class0 = len(i_class0)
n_class1 = len(i_class1)

# For every observation of class 0, randomly sample
# from class 1 without replacement
i_class1_downsampled = np.random.choice(i_class1, size=n_class0, replace=False)

# Join together class 0's target vector with the
# downsampled class 1's target vector
np.hstack((target[i_class0], target[i_class1_downsampled]))
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

```
# Join together class 0's feature matrix with the
# downsampled class 1's feature matrix
np.vstack((features[i_class0,:], features[i_class1_downsampled,:]))[0:5]
```

```
array([[ 5. ,  3.5,  1.3,  0.3],
       [ 4.5,  2.3,  1.3,  0.3],
       [ 4.4,  3.2,  1.3,  0.2],
       [ 5. ,  3.5,  1.6,  0.6],
       [ 5.1,  3.8,  1.9,  0.4]])
```

Our other option is to upsample the minority class. In upsampling, for every observation in the majority class, we randomly select an observation from the minority class with replacement. The end result is the same number of observations from the minority and majority classes. Upsampling is implemented very similarly to downsampling, just in reverse:

```
# For every observation in class 1, randomly sample from class 0 with replacement
i_class0_upsampled = np.random.choice(i_class0, size=n_class1, replace=True)

# Join together class 0's upsampled target vector with class 1's target vector
np.concatenate((target[i_class0_upsampled], target[i_class1]))
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

```
# Join together class 0's upsampled feature matrix with class 1's feature matrix
np.vstack((features[i_class0_upsampled,:], features[i_class1,:]))[0:5]
```

```
array([[ 5. ,  3.5,  1.6,  0.6],
       [ 5. ,  3.5,  1.6,  0.6],
       [ 5. ,  3.3,  1.4,  0.2],
       [ 4.5,  2.3,  1.3,  0.3],
       [ 4.8,  3. ,  1.4,  0.3]])
```

Discussion

In the real world, imbalanced classes are everywhere—most visitors don't click the buy button and many types of cancer are thankfully rare. For this reason, handling imbalanced classes is a common activity in machine learning.

Our best strategy is simply to collect more observations—especially observations from the minority class. However, this is often just not possible, so we have to resort to other options.

A second strategy is to use a model evaluation metric better suited to imbalanced classes. Accuracy is often used as a metric for evaluating the performance of a model, but when imbalanced classes are present accuracy can be ill suited. For example, if only 0.5% of observations have some rare cancer, then even a naive model that predicts nobody has cancer will be 99.5% accurate. Clearly this is not ideal. Some better metrics we discuss in later chapters are confusion matrices, precision, recall, F1 scores, and ROC curves.

A third strategy is to use the class weighing parameters included in implementations of some models. This allows us to have the algorithm adjust for imbalanced classes. Fortunately, many scikit-learn classifiers have a `class_weight` parameter, making it a good option.

The fourth and fifth strategies are related: downsampling and upsampling. In downsampling we create a random subset of the majority class of equal size to the minority class. In upsampling we repeatedly sample with replacement from the minority class to make it of equal size as the majority class. The decision between using downsampling and upsampling is context-specific, and in general we should try both to see which produces better results.

Chapter 6. Handling Text

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at feedback.mlpythoncookbook@gmail.com.

6.0 Introduction

Unstructured text data, like the contents of a book or a tweet, is both one of the most interesting sources of features and one of the most complex to handle. In this chapter, we will cover strategies for transforming text into information-rich features, and use some out-of-the-box features (termed embeddings) that have become increasingly ubiquitous in tasks that involve natural language processing (NLP).

This is not to say that the recipes covered here are comprehensive. There exist entire academic disciplines focused on handling unstructured data such as text. In this chapter, we will cover some commonly used techniques, and a knowledge of these will add valuable tools to our preprocessing toolbox. In addition to many generic text processing recipes, we'll also demonstrate how you can import and leverage some pre-trained machine learning models to generate richer text features.

6.1 Cleaning Text

Problem

You have some unstructured text data and want to complete some basic cleaning.

Solution

In the following example, we take a look at the text for three books and clean it by using Python's core string operations, in particular `strip`, `replace`, and `split`:

```
# Create text
text_data = ["   Interrobang. By Aishwarya Henriette     ",
             "Parking And Going. By Karl Gautier",
             "    Today Is The night. By Jarek Prakash   "]

# Strip whitespaces
strip_whitespace = [string.strip() for string in text_data]

# Show text
strip_whitespace
```

```
['Interrobang. By Aishwarya Henriette',
 'Parking And Going. By Karl Gautier',
 'Today Is The night. By Jarek Prakash']
```

```
# Remove periods
remove_periods = [string.replace(".", "") for string in strip_whitespace]

# Show text
remove_periods
```

```
['Interrobang By Aishwarya Henriette',
 'Parking And Going By Karl Gautier',
 'Today Is The night By Jarek Prakash']
```

We also create and apply a custom transformation function:

```
# Create function
def capitalizer(string: str) -> str:
    return string.upper()

# Apply function
[capitalizer(string) for string in remove_periods]
```

```
['INTERROBANG BY AISHWARYA HENRIETTE',
 'PARKING AND GOING BY KARL GAUTIER',
 'TODAY IS THE NIGHT BY JAREK PRAKASH']
```

Finally, we can use regular expressions to make powerful string operations:

```
# Import library
import re

# Create function
def replace_letters_with_X(string: str) -> str:
    return re.sub(r"[a-zA-Z]", "X", string)

# Apply function
[replace_letters_with_X(string) for string in remove_periods]
```

```
['XXXXXXXXXXX XX XXXXXXXXX XXXXXXXXX',
 'XXXXXXX XXX XXXXX XX XXXX XXXXXXX',
 'XXXXX XX XXX XXXXX XX XXXXX XXXXXXX']
```

Discussion

Some text data will need to be cleaned before we can use it to build features, or preprocessed in some way prior to being fed into an algorithm. Most basic text cleaning can be completed using Python's standard string operations. In the real world, we will most likely define a custom cleaning function (e.g., `capitalizer`) combining some cleaning tasks and apply that to the text data. Strings have many inherent methods that are useful for cleaning and processing, some additional examples can be found below:

```
# Define a string
s = "machine learning in python cookbook"

# Find the first index of the letter "n"
find_n = s.find("n")

# Whether or not the string starts with "m"
```

```
starts_with_m = s.startswith("m")

# Whether or not the string ends with "python"
ends_with_python = s.endswith("python")

# Is the string alphanumeric
is_alnum = s.isalnum()

# Is it composed of only alphabetical characters (not including spaces)
is_alpha = s.isalpha()

# Encode as utf-8
encode_as_utf8 = s.encode("utf-8")

# Decode the same utf-8
decode = encode_as_utf8.decode("utf-8")

print(find_n, starts_with_m, ends_with_python, is_alnum, is_alpha, encode_as_utf8, decode, sep = "|")
```

```
5|True|False|False|False|b'machine learning in python cookbook'|machine learning in python cookbook
```

See Also

- Beginners Tutorial for Regular Expressions in Python

6.2 Parsing and Cleaning HTML

Problem

You have text data with HTML elements and want to extract just the text.

Solution

Use Beautiful Soup's extensive set of options to parse and extract from HTML:

```
# Load library
from bs4 import BeautifulSoup

# Create some HTML code
html = """
       <div class='full_name'><span style='font-weight:bold'>Masego</span> Azra</div>"
       """

# Parse html
soup = BeautifulSoup(html, "lxml")

# Find the div with the class "full_name", show text
soup.find("div", { "class" : "full_name" }).text
```

```
'Masego Azra'
```

Discussion

Despite the strange name, Beautiful Soup is a powerful Python library designed for scraping HTML. Typically Beautiful Soup is used to process HTML during live web scraping processes, but we can just as easily use it to extract text data embedded in static HTML. The full range of Beautiful Soup operations is beyond the scope of this book, but even the method we use in our solution show how easy

it can be to parse HTML and extract information from specific tags using `find()`.

See Also

- Beautiful Soup

6.3 Removing Punctuation

Problem

You have a feature of text data and want to remove punctuation.

Solution

Define a function that uses `translate` with a dictionary of punctuation characters:

```
# Load libraries
import unicodedata
import sys

# Create text
text_data = ['Hi!!!! I. Love. This. Song....',
             '10000% Agree!!!! #LoveIT',
             'Right?!?!']

# Create a dictionary of punctuation characters
punctuation = dict.fromkeys(i for i in range(sys.maxunicode)
                            if unicodedata.category(chr(i)).startswith('P'))

# For each string, remove any punctuation characters
[string.translate(punctuation) for string in text_data]
```

```
['Hi I Love This Song', '10000 Agree LoveIT', 'Right']
```

Discussion

`translate` is a Python method that is popular due to its speed. In our solution, first we created a dictionary, `punctuation`, with all punctuation characters according to Unicode as its keys and `None` as its values. Next we translated all characters in the string that are in `punctuation` into `None`, effectively removing them. There are more readable ways to remove punctuation, but this somewhat hacky solution has the advantage of being far faster than alternatives.

It is important to be conscious of the fact that punctuation contains information (e.g., “Right?” versus “Right!”). Removing punctuation can be a necessary evil when we need to manually create features; however, if the punctuation is important we should make sure to take that into account.

6.4 Tokenizing Text

Problem

You have text and want to break it up into individual words.

Solution

Natural Language Toolkit for Python (NLTK) has a powerful set of text manipulation operations, including word tokenizing:

```
# Load library
from nltk.tokenize import word_tokenize

# Create text
string = "The science of today is the technology of tomorrow"

# Tokenize words
word_tokenize(string)
```

```
['The', 'science', 'of', 'today', 'is', 'the', 'technology', 'of', 'tomorrow']
```

We can also tokenize into sentences:

```
# Load library
from nltk.tokenize import sent_tokenize

# Create text
string = "The science of today is the technology of tomorrow. Tomorrow is today."

# Tokenize sentences
sent_tokenize(string)
```

```
['The science of today is the technology of tomorrow.', 'Tomorrow is today.']
```

Discussion

Tokenization, especially word tokenization, is a common task after cleaning text data because it is the first step in the process of turning the text into data we will use to construct useful features. Some pre-trained NLP models (such as Google's BERT) utilize model-specific tokenization techniques, however word-level tokenization is still a fairly common tokenization approach before getting features from individual words.

6.5 Removing Stop Words

Problem

Given tokenized text data, you want to remove extremely common words (e.g., *a*, *is*, *of*, *on*) that contain little informational value.

Solution

Use NLTK's `stopwords`:

```
# Load library
from nltk.corpus import stopwords
```

```
# You will have to download the set of stop words the first time
# import nltk
# nltk.download('stopwords')

# Create word tokens
tokenized_words = ['i',
                   'am',
                   'going',
                   'to',
                   'go',
                   'to',
                   'the',
                   'store',
                   'and',
                   'park']

# Load stop words
stop_words = stopwords.words('english')

# Remove stop words
[word for word in tokenized_words if word not in stop_words]
```

```
['going', 'go', 'store', 'park']
```

Discussion

While "stop words" can refer to any set of words we want to remove before processing, frequently the term refers to extremely common words that themselves contain little information value. Whether or not you choose to remove stop words will depend on your individual use case. NLTK has a list of common stop words that we can use to find and remove stop words in our tokenized words:

```
# Show stop words
stop_words[:5]
```

```
['i', 'me', 'my', 'myself', 'we']
```

Note that NLTK's `stopwords` assumes the tokenized words are all lowercased.

6.6 Stemming Words

Problem

You have tokenized words and want to convert them into their root forms.

Solution

Use NLTK's `PorterStemmer`:

```
# Load library
from nltk.stem.porter import PorterStemmer

# Create word tokens
tokenized_words = ['i', 'am', 'humbled', 'by', 'this', 'traditional', 'meeting']

# Create stemmer
porter = PorterStemmer()
```

```
# Apply stemmer
[porter.stem(word) for word in tokenized_words]
```

```
['i', 'am', 'humbl', 'by', 'thi', 'tradit', 'meet']
```

Discussion

Stemming reduces a word to its stem by identifying and removing affixes (e.g., gerunds) while keeping the root meaning of the word. For example, both "tradition" and "traditional" have "tradit" as their stem, indicating that while they are different words they represent the same general concept. By stemming our text data, we transform it to something less readable, but closer to its base meaning and thus more suitable for comparison across observations. NLTK's `PorterStemmer` implements the widely used Porter stemming algorithm to remove or replace common suffixes to produce the word stem.

See Also

- Porter Stemming Algorithm

6.7 Tagging Parts of Speech

Problem

You have text data and want to tag each word or character with its part of speech.

Solution

Use NLTK's pre-trained parts-of-speech tagger:

```
# Load libraries
from nltk import pos_tag
from nltk import word_tokenize

# Create text
text_data = "Chris loved outdoor running"

# Use pre-trained part of speech tagger
text_tagged = pos_tag(word_tokenize(text_data))

# Show parts of speech
text_tagged
```

```
[('Chris', 'NNP'), ('loved', 'VBD'), ('outdoor', 'RP'), ('running', 'VBG')]
```

The output is a list of tuples with the word and the tag of the part of speech. NLTK uses the Penn Treebank parts for speech tags. Some examples of the Penn Treebank tags are:

Tag	Part of speech
NNP	Proper noun, singular
NN	Noun, singular or mass
RB	Adverb

VBD	Verb, past tense
VBG	Verb, gerund or present participle
JJ	Adjective
PRP	Personal pronoun

Once the text has been tagged, we can use the tags to find certain parts of speech. For example, here are all nouns:

```
# Filter words
[word for word, tag in text_tagged if tag in ['NN','NNS','NNP','NNPS'] ]
```

```
['Chris']
```

A more realistic situation would be that we have data where every observation contains a tweet and we want to convert those sentences into features for individual parts of speech (e.g., a feature with `1` if a proper noun is present, and `0` otherwise):

```
# Import libraries
from sklearn.preprocessing import MultiLabelBinarizer

# Create text
tweets = ["I am eating a burrito for breakfast",
          "Political science is an amazing field",
          "San Francisco is an awesome city"]

# Create list
tagged_tweets = []

# Tag each word and each tweet
for tweet in tweets:
    tweet_tag = nltk.pos_tag(word_tokenize(tweet))
    tagged_tweets.append([tag for word, tag in tweet_tag])

# Use one-hot encoding to convert the tags into features
one_hot_multi = MultiLabelBinarizer()
one_hot_multi.fit_transform(tagged_tweets)
```

```
array([[1, 1, 0, 1, 0, 1, 1, 1, 0],
       [1, 0, 1, 1, 0, 0, 0, 0, 1],
       [1, 0, 1, 1, 1, 0, 0, 0, 1]])
```

Using `classes_` we can see that each feature is a part-of-speech tag:

```
# Show feature names
one_hot_multi.classes_
```

```
array(['DT', 'IN', 'JJ', 'NN', 'NNP', 'PRP', 'VBG', 'VBP', 'VBZ'], dtype=object)
```

Discussion

If our text is English and not on a specialized topic (e.g., medicine) the simplest solution is to use NLTK's pre-trained parts-of-speech tagger. However, if `pos_tag` is not very accurate, NLTK also gives us the ability to train our own tagger. The major downside of training a tagger is that we need a large corpus of text where the tag of each word is known. Constructing this tagged corpus is obviously labor

intensive and is probably going to be a last resort.

All that said, if we had a tagged corpus and wanted to train a tagger, the following is an example of how we could do it. The corpus we are using is the Brown Corpus, one of the most popular sources of tagged text. Here we use a backoff *n*-gram tagger, where *n* is the number of previous words we take into account when predicting a word's part-of-speech tag. First we take into account the previous two words using `TrigramTagger`; if two words are not present, we "back off" and take into account the tag of the previous one word using `BigramTagger`, and finally if that fails we only look at the word itself using `UnigramTagger`. To examine the accuracy of our tagger, we split our text data into two parts, train our tagger on one part, and test how well it predicts the tags of the second part:

```
# Load library
from nltk.corpus import brown
from nltk.tag import UnigramTagger
from nltk.tag import BigramTagger
from nltk.tag import TrigramTagger

# Get some text from the Brown Corpus, broken into sentences
sentences = brown.tagged_sents(categories='news')

# Split into 4000 sentences for training and 623 for testing
train = sentences[:4000]
test = sentences[4000:]

# Create backoff tagger
unigram = UnigramTagger(train)
bigram = BigramTagger(train, backoff=unigram)
trigram = TrigramTagger(train, backoff=bigram)

# Show accuracy
trigram.accuracy(test)
```

```
0.8174734002697437
```

See Also

- Penn Treebank
- Brown Corpus

6.8 Performing Named-Entity Recognition

Problem

You want to perform named-entity recognition in freeform text (such at "Person", "State", etc…).

Solution

Use spaCy's default named-entity recognition pipeline and models to extract entites from text:

```
# Import libraries
import spacy

# Load the spaCy package and use it to parse the text (make sure you have run "python -m spacy download en")
nlp = spacy.load("en_core_web_sm")
```

```
doc = nlp("Elon Musk offered to buy Twitter using $21B of his own money.")

# Print each entity
print(doc.ents)

# For each entity print the text and the entity label
for entity in doc.ents:
    print(entity.text, entity.label_, sep=",")
```

```
(Elon Musk, Twitter, 21B)
Elon Musk, PERSON
Twitter, ORG
21B, MONEY
```

Discussion

Named-entity recognition is the process of recognizing specific entities from text. Tools like spaCy offer pre-configured pipelines and pre-trained machine learning models that can easily identify these entities. In this case, we use spaCy to identify a person (“Elon Musk”), organization (“Twitter”) and money value (“21B”) from the raw text. Using this information, we can extract structured information from the unstructured textual data. This information can then be used in downstream machine learning models or data analysis.

Training a custom named-entity recognition model is outside the scope of this example, however it is often done using deep learning and other NLP techniques.

See Also

- spaCy named-entity recognition docs
- Named-entity recognition on Wikipedia

6.9 Encoding Text as a Bag of Words

Problem

You have text data and want to create a set of features indicating the number of times an observation’s text contains a particular word.

Solution

Use scikit-learn’s `CountVectorizer`:

```
# Load library
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer

# Create text
text_data = np.array(['I love Brazil. Brazil!',
                      'Sweden is best',
                      'Germany beats both'])



# Create the bag of words feature matrix
```

```
count = CountVectorizer()
bag_of_words = count.fit_transform(text_data)

# Show feature matrix
bag_of_words
```

```
<3x8 sparse matrix of type '<class 'numpy.int64'>'
    with 8 stored elements in Compressed Sparse Row format>
```

This output is a sparse array, which is often necessary when we have a large amount of text. However, in our toy example we can use `toarray` to view a matrix of word counts for each observation:

```
bag_of_words.toarray()
```

```
array([[0, 0, 0, 2, 0, 0, 1, 0],
       [0, 1, 0, 0, 0, 1, 0, 1],
       [1, 0, 1, 0, 1, 0, 0, 0]], dtype=int64)
```

We can use the `get_feature_names` method to view the word associated with each feature:

```
# Show feature names
count.get_feature_names_out()
```

```
array(['beats', 'best', 'both', 'brazil', 'germany', 'is', 'love',
       'sweden'], dtype=object)
```

This might be confusing, so for the sake of clarity here is what the feature matrix looks like with the words as column names (each row is one observation):

beats	best	both	brazil	germany	is	love	sweden
0	0	0	2	0	0	1	0
0	1	0	0	0	1	0	1
1	0	1	0	1	0	0	0

Discussion

One of the most common methods of transforming text into features is by using a bag-of-words model. Bag-of-words models output a feature for every unique word in text data, with each feature containing a count of occurrences in observations. For example, in our solution the sentence `I love Brazil. Brazil!` has a value of `2` in the "brazil" feature because the word *brazil* appears two times.

The text data in our solution was purposely small. In the real world, a single observation of text data could be the contents of an entire book! Since our bag-of-words model creates a feature for every unique word in the data, the resulting matrix can contain thousands of features. This means that the size of the matrix can sometimes become very large in memory. However, luckily we can exploit a common characteristic of bag-of-words feature matrices to reduce the amount of data we need to store.

Most words likely do not occur in most observations, and therefore bag-of-words feature matrices will contain mostly 0s as values. We call these types of matrices "sparse." Instead of storing all values of the matrix, we can only store nonzero values and then assume all other values are 0. This will save us memory when we have large feature matrices. One of the nice features of `CountVectorizer` is that the

output is a sparse matrix by default.

`CountVectorizer` comes with a number of useful parameters to make creating bag-of-words feature matrices easy. First, while by default every feature is a word, that does not have to be the case. Instead we can set every feature to be the combination of two words (called a 2-gram) or even three words (3-gram). `ngram_range` sets the minimum and maximum size of our *n*-grams. For example, `(2,3)` will return all 2-grams and 3-grams. Second, we can easily remove low-information filler words using `stop_words` either with a built-in list or a custom list. Finally, we can restrict the words or phrases we want to consider to a certain list of words using `vocabulary`. For example, we could create a bag-of-words feature matrix for only occurrences of country names:

```
# Create feature matrix with arguments
count_2gram = CountVectorizer(ngram_range=(1,2),
                              stop_words="english",
                              vocabulary=['brazil'])
bag = count_2gram.fit_transform(text_data)

# View feature matrix
bag.toarray()
```

```
array([[2],
       [0],
       [0]])
```

```
# View the 1-grams and 2-grams
count_2gram.vocabulary_
```

```
{'brazil': 0}
```

See Also

- *n*-gram
- Bag of Words Meets Bags of Popcorn

6.10 Weighting Word Importance

Problem

You want a bag of words, but with words weighted by their importance to an observation.

Solution

Compare the frequency of the word in a document (a tweet, movie review, speech transcript, etc.) with the frequency of the word in all other documents using term frequency-inverse document frequency (tf-idf). scikit-learn makes this easy with `TfidfVectorizer`:

```
# Load libraries
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer

# Create text
text_data = np.array(['I love Brazil. Brazil!',
```

```
                    'Sweden is best',
                    'Germany beats both'])

# Create the tf-idf feature matrix
tfidf = TfidfVectorizer()
feature_matrix = tfidf.fit_transform(text_data)

# Show tf-idf feature matrix
feature_matrix
```

```
<3x8 sparse matrix of type '<class 'numpy.float64'>'
    with 8 stored elements in Compressed Sparse Row format>
```

Just as in Recipe 6.9, the output is a spare matrix. However, if we want to view the output as a dense matrix, we can use `.toarray`:

```
# Show tf-idf feature matrix as dense matrix
feature_matrix.toarray()
```

```
array([[ 0.        ,  0.        ,  0.        ,  0.89442719,  0.        ,
         0.        ,  0.4472136 ,  0.        ],
       [ 0.        ,  0.57735027,  0.        ,  0.        ,  0.        ,
         0.57735027,  0.        ,  0.57735027],
       [ 0.57735027,  0.        ,  0.57735027,  0.        ,  0.57735027,
         0.        ,  0.        ,  0.        ]])
```

`vocabulary_` shows us the word of each feature:

```
# Show feature names
tfidf.vocabulary_
```

```
{'love': 6,
 'brazil': 3,
 'sweden': 7,
 'is': 5,
 'best': 1,
 'germany': 4,
 'beats': 0,
 'both': 2}
```

Discussion

The more a word appears in a document, the more likely it is that the word is important to that document. For example, if the word *economy* appears frequently, it is evidence that the document might be about economics. We call this *term frequency* (*tf*).

In contrast, if a word appears in many documents, it is likely less important to any individual document. For example, if every document in some text data contains the word *after* then it is probably an unimportant word. We call this document frequency (*df*).

By combining these two statistics, we can assign a score to every word representing how important that word is in a document. Specifically, we multiply *tf* to the inverse of document frequency (*idf*):

$$\text{tf-idf}(t, d) = tf(t, d) \times idf(t)$$

where *t* is a word and *d* is a document. There are a number of variations in how *tf* and *idf* are calculated. In scikit-learn, *tf* is simply the number of times a word appears in the document and *idf* is calculated as:

$$\text{idf}\,(t) = log\frac{1+n_d}{1+\text{df}(d,t)}+1$$

where n_d is the number of documents and df*(d,t)* is term, *t*'s document frequency (i.e., number of documents where the term appears).

By default, scikit-learn then normalizes the tf-idf vectors using the Euclidean norm (L2 norm). The higher the resulting value, the more important the word is to a document.

See Also

- scikit-learn documentation: tf–idf term weighting

6.11 Using Word Vectors to Calculate Text Similarity in a Search Query

Problem

You want to use tf-idf vectors to implement a text search function in Python.

Solution

Calculate the cosine similarity between tf-idf vecotrs using scikit-learn.

```
# Load libraries
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import linear_kernel

# Create searchable text data
text_data = np.array(['I love Brazil. Brazil!',
                      'Sweden is best',
                      'Germany beats both'])

# Create the tf-idf feature matrix
tfidf = TfidfVectorizer()
feature_matrix = tfidf.fit_transform(text_data)

# Create a search query and transform it into a tf-idf vector
text = "Brazil is the best"
vector = tfidf.transform([text])

# Calculate the cosine similarities between the input vector and all other vectors
cosine_similarities = linear_kernel(vector, feature_matrix).flatten()

# Get the index of the most relevent items in order
related_doc_indicies = cosine_similarities.argsort()[:-10:-1]

# Print the most similar texts to the search query along with the cosine similarity
print([(text_data[i], cosine_similarities[i]) for i in related_doc_indicies])
```

```
[('Sweden is best', 0.6666666666666666), ('I love Brazil. Brazil!', 0.5163977794943222), ('Germany beats
both', 0.0)]
```

Discussion

Word vectors are incredibly useful for NLP use cases such as search engines. After calculating the tf-idf vectors of a set of sentences or documents, we can use the same `tfidf` object to vectorize future sets of text. Then, we can compute cosine similarity between our input vector and the matrix of other vectors and sort by the most relevant documents.

Cosine similarities take on the range of [0, 1.0], with 0 being least similar and 1 being most similar. Since we're using tf-idf vectors to compute the similarity between vectors, the frequency of occurence of a word is also taken into account. However, with a small corpus (set of documents) even "frequent" words may not appear frequently. In the example above, "Sweden is best" is the most relevant text to our search query "Brazil is the best". Since the query mentions Brazil, we might expect that to be the most relevant - however "Sweden is best" is the most similar due to the words "is" and "best". As the amount of documents we add to our corpus increases, less important words will be weighted less and have less effect on our cosine similarity calculation.

See Also

- Geeks for geeks cosine similarity
- Building a pubmed search engine in Python

6.12 Using a Sentiment Analysis Classifier

Problem

You want to classify the sentiment of some text to use as a feature or in downstream data analysis.

Solution

Use the transformers library's sentiment classifier.

```
# Import libraries
from transformers import pipeline

# Create an NLP pipeline that runs sentiment analysis
classifier = pipeline("sentiment-analysis")

# Classify some text (this may download some data and models the first time you run it)
sentiment_1 = classifier("I hate machine learning! It's the absolute worst.")
sentiment_2 = classifier("Machine learning is the absolute bees knees I love it so much!")

# Print sentiment output
print(sentiment_1, sentiment_2)
```

```
[{'label': 'NEGATIVE', 'score': 0.9998020529747009}] [{'label': 'POSITIVE', 'score': 0.9990628957748413}]
```

Discussion

The transformers library is an extremely popular library for NLP tasks, and contains a number of easy-to-use APIs for training models, or utilizing pre-trained ones. We'll talk more about NLP and this

library in the chapter on NLP, but the example above serves as a high level introduction to the power of utilizing pre-trained classifiers in your machine learning pipelines to generate features, classify text, or analyze unstructured data.

See Also

- Huggingface transformers quick tour

Chapter 7. Handling Dates and Times

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at feedback.mlpythoncookbook@gmail.com.

7.0 Introduction

Dates and times (datetimes) (for example, the time of a particular sale or the date of some public health statistic) are frequently encountered during preprocessing for machine learning, whether the time of a particular sale or the year of some public health statistic. Longitudinal data (or time series data) are data that are collected repeatedly for the same variables over points in time. In this chapter, we will build a toolbox of strategies for handling time series data including tackling time zones and creating lagged time features. Specifically, we will focus on the time series tools in the pandas library, which centralizes the functionality of many other general libraries such as `datetime`.

7.1 Converting Strings to Dates

Problem

Given a vector of strings representing dates and times, you want to transform them into time series data.

Solution

Use pandas' `to_datetime` with the format of the date and/or time specified in the `format` parameter:

```
# Load libraries
import numpy as np
import pandas as pd

# Create strings
date_strings = np.array(['03-04-2005 11:35 PM',
                         '23-05-2010 12:01 AM',
                         '04-09-2009 09:09 PM'])

# Convert to datetimes
[pd.to_datetime(date, format='%d-%m-%Y %I:%M %p') for date in date_strings]
```

```
[Timestamp('2005-04-03 23:35:00'),
 Timestamp('2010-05-23 00:01:00'),
 Timestamp('2009-09-04 21:09:00')]
```

We might also want to add an argument to the `errors` parameter to handle problems:

```
# Convert to datetimes
[pd.to_datetime(date, format="%d-%m-%Y %I:%M %p", errors="coerce")
for date in date_strings]
```

```
[Timestamp('2005-04-03 23:35:00'),
 Timestamp('2010-05-23 00:01:00'),
 Timestamp('2009-09-04 21:09:00')]
```

If `errors="coerce"`, then any problem that occurs will not raise an error (the default behavior) but instead will set the value causing the error to `NaT` (a missing value). This allows you to deal with outliers by filling them with null values, as opposed to troubleshooting errors for individual records in the data.

Discussion

When dates and times come as strings, we need to convert them into a data type Python can understand. While there are a number of Python tools for converting strings to datetimes, following our use of pandas in other recipes we can use `to_datetime` to conduct the transformation. One obstacle to strings representing dates and times is that the format of the strings can vary significantly between data sources. For example, one vector of dates might represent March 23rd, 2015 as “03-23-15” while another might use “3|23|2015”. We can use the `format` parameter to specify the exact format of the string. Here are some common date and time formatting codes:

Code	Description	Example
`%Y`	Full year	`2001`
`%m`	Month w/ zero padding	`04`
`%d`	Day of the month w/ zero padding	`09`
`%I`	Hour (12hr clock) w/ zero padding	`02`
`%p`	AM or PM	`AM`
`%M`	Minute w/ zero padding	`05`
`%S`	Second w/ zero padding	`09`

See Also

- Complete List of Python String Time Codes

7.2 Handling Time Zones

Problem

You have time series data and want to add or change time zone information.

Solution

If not specified, pandas objects have no time zone. However, we can add a time zone using `tz` during creation:

```
# Load library
import pandas as pd

# Create datetime
pd.Timestamp('2017-05-01 06:00:00', tz='Europe/London')
```

```
Timestamp('2017-05-01 06:00:00+0100', tz='Europe/London')
```

We can add a time zone to a previously created datetime using `tz_localize`:

```
# Create datetime
date = pd.Timestamp('2017-05-01 06:00:00')

# Set time zone
date_in_london = date.tz_localize('Europe/London')

# Show datetime
date_in_london
```

```
Timestamp('2017-05-01 06:00:00+0100', tz='Europe/London')
```

We can also convert to a different time zone:

```
# Change time zone
date_in_london.tz_convert('Africa/Abidjan')
```

```
Timestamp('2017-05-01 05:00:00+0000', tz='Africa/Abidjan')
```

Finally, pandas' Series objects can apply `tz_localize` and `tz_convert` to every element:

```
# Create three dates
dates = pd.Series(pd.date_range('2/2/2002', periods=3, freq='M'))

# Set time zone
dates.dt.tz_localize('Africa/Abidjan')
```

```
0   2002-02-28 00:00:00+00:00
1   2002-03-31 00:00:00+00:00
2   2002-04-30 00:00:00+00:00
dtype: datetime64[ns, Africa/Abidjan]
```

Discussion

pandas supports two sets of strings representing timezones; however, I suggest using pytz library's strings. We can see all the strings used to represent time zones by importing `all_timezones`:

```
# Load library
from pytz import all_timezones

# Show two time zones
all_timezones[0:2]
```

```
['Africa/Abidjan', 'Africa/Accra']
```

7.3 Selecting Dates and Times

Problem

You have a vector of dates and you want to select one or more.

Solution

Use two boolean conditions as the start and end dates:

```
# Load library
import pandas as pd

# Create data frame
dataframe = pd.DataFrame()

# Create datetimes
dataframe['date'] = pd.date_range('1/1/2001', periods=100000, freq='H')

# Select observations between two datetimes
dataframe[(dataframe['date'] > '2002-1-1 01:00:00') &
          (dataframe['date'] <= '2002-1-1 04:00:00')]
```

	date
8762	2002-01-01 02:00:00
8763	2002-01-01 03:00:00
8764	2002-01-01 04:00:00

Alternatively, we can set the date column as the DataFrame's index and then slice using `loc`:

```
# Set index
dataframe = dataframe.set_index(dataframe['date'])

# Select observations between two datetimes
dataframe.loc['2002-1-1 01:00:00':'2002-1-1 04:00:00']
```

	date
	date
2002-01-01 01:00:00	2002-01-01 01:00:00
2002-01-01 02:00:00	2002-01-01 02:00:00
2002-01-01 03:00:00	2002-01-01 03:00:00
2002-01-01 04:00:00	2002-01-01 04:00:00

Discussion

Whether we use boolean conditions or index slicing is situation dependent. If we wanted to do some complex time series manipulation, it might be worth the overhead of setting the date column as the index of the DataFrame, but if we wanted to do some simple data wrangling, the boolean conditions might be easier.

7.4 Breaking Up Date Data into Multiple Features

Problem

You have a column of dates and times and you want to create features for year, month, day, hour, and minute.

Solution

Use pandas `Series.dt`'s time properties:

```
# Load library
import pandas as pd

# Create data frame
dataframe = pd.DataFrame()

# Create five dates
dataframe['date'] = pd.date_range('1/1/2001', periods=150, freq='W')

# Create features for year, month, day, hour, and minute
dataframe['year'] = dataframe['date'].dt.year
dataframe['month'] = dataframe['date'].dt.month
dataframe['day'] = dataframe['date'].dt.day
dataframe['hour'] = dataframe['date'].dt.hour
dataframe['minute'] = dataframe['date'].dt.minute

# Show three rows
dataframe.head(3)
```

	date	year	month	day	hour	minute
0	2001-01-07	2001	1	7	0	0
1	2001-01-14	2001	1	14	0	0
2	2001-01-21	2001	1	21	0	0

Discussion

Sometimes it can be useful to break up a column of dates into components. For example, we might want a feature that just includes the year of the observation or we might want only to consider the month of some observation so we can compare them regardless of year.

7.5 Calculating the Difference Between Dates

Problem

You have two datetime features and want to calculate the time between them for each observation.

Solution

Subtract the two date features using pandas:

```
# Load library
import pandas as pd

# Create data frame
dataframe = pd.DataFrame()

# Create two datetime features
dataframe['Arrived'] = [pd.Timestamp('01-01-2017'), pd.Timestamp('01-04-2017')]
dataframe['Left'] = [pd.Timestamp('01-01-2017'), pd.Timestamp('01-06-2017')]

# Calculate duration between features
dataframe['Left'] - dataframe['Arrived']
```

```
0   0 days
1   2 days
dtype: timedelta64[ns]
```

Often we will want to remove the `days` output and keep only the numerical value:

```
# Calculate duration between features
pd.Series(delta.days for delta in (dataframe['Left'] - dataframe['Arrived']))
```

```
0    0
1    2
dtype: int64
```

Discussion

There are times when the feature we want is the change (delta) between two points in time. For example, we might have the dates a customer checks in and checks out of a hotel, but the feature we want is the duration of his stay. pandas makes this calculation easy using the `TimeDelta` data type.

See Also

- Pandas documentation: Time Deltas

7.6 Encoding Days of the Week

Problem

You have a vector of dates and want to know the day of the week for each date.

Solution

Use pandas' `Series.dt` method `day_name()`:

```
# Load library
import pandas as pd

# Create dates
dates = pd.Series(pd.date_range("2/2/2002", periods=3, freq="M"))

# Show days of the week
dates.dt.day_name()
```

```
0    Thursday
1      Sunday
2     Tuesday
dtype: object
```

If we want the output to be a numerical value and therefore more usable as a machine learning feature, we can use `weekday` where the days of the week are represented as an integer (Monday is 0):

```
# Show days of the week
dates.dt.weekday
```

```
0    3
1    6
2    1
dtype: int64
```

Discussion

Knowing the weekday can be helpful if, for instance, we wanted to compare total sales on Sundays for the past three years. pandas makes creating a feature vector containing weekday information easy.

See Also

- pandas Series datetimelike properties

7.7 Creating a Lagged Feature

Problem

You want to create a feature that is lagged *n* time periods.

Solution

Use pandas' `shift`:

```
# Load library
import pandas as pd

# Create data frame
dataframe = pd.DataFrame()

# Create data
dataframe["dates"] = pd.date_range("1/1/2001", periods=5, freq="D")
dataframe["stock_price"] = [1.1,2.2,3.3,4.4,5.5]

# Lagged values by one row
dataframe["previous_days_stock_price"] = dataframe["stock_price"].shift(1)

# Show data frame
dataframe
```

	dates	stock_price	previous_days_stock_price
0	2001-01-01	1.1	NaN
1	2001-01-02	2.2	1.1

2	2001-01-03	3.3	2.2
3	2001-01-04	4.4	3.3
4	2001-01-05	5.5	4.4

Discussion

Very often data is based on regularly spaced time periods (e.g., every day, every hour, every three hours) and we are interested in using values in the past to make predictions (this is often called *lagging* a feature). For example, we might want to predict a stock's price using the price it was the day before. With pandas we can use `shift` to lag values by one row, creating a new feature containing past values.

In our solution, the first row for `previous_days_stock_price` is a missing value because there is no previous `stock_price` value.

7.8 Using Rolling Time Windows

Problem

Given time series data, you want to calculate some statistic for a rolling time.

Solution

```
# Load library
import pandas as pd

# Create datetimes
time_index = pd.date_range("01/01/2010", periods=5, freq="M")

# Create data frame, set index
dataframe = pd.DataFrame(index=time_index)

# Create feature
dataframe["Stock_Price"] = [1,2,3,4,5]

# Calculate rolling mean
dataframe.rolling(window=2).mean()
```

	Stock_Price
2010-01-31	NaN
2010-02-28	1.5
2010-03-31	2.5
2010-04-30	3.5
2010-05-31	4.5

Discussion

Rolling (also called moving) time windows are conceptually simple but can be difficult to understand at first. Imagine we have monthly observations for a stock's price. It is often useful to have a time window of a certain number of months and then move over the observations calculating a statistic for all

observations in the time window.

For example, if we have a time window of three months and we want a rolling mean, we would calculate:

1. mean(January, February, March)
2. mean(February, March, April)
3. mean(March, April, May)
4. etc.

Another way to put it: our three-month time window "walks" over the observations, calculating the window's mean at each step.

pandas' `rolling` allows us to specify the size of the window using `window` and then quickly calculate some common statistics, including the max value (`max()`), mean value (`mean()`), count of values (`count()`), and rolling correlation (`corr()`).

Rolling means are often used to smooth out time series data because using the mean of the entire time window dampens the effect of short-term fluctuations.

See Also

- pandas documentation: Rolling Windows
- What are Moving Average or Smoothing Techniques?

7.9 Handling Missing Data in Time Series

Problem

You have missing values in time series data.

Solution

In addition to the missing data strategies previously discussed, when we have time series data we can use interpolation to fill in gaps caused by missing values:

```
# Load libraries
import pandas as pd
import numpy as np

# Create date
time_index = pd.date_range("01/01/2010", periods=5, freq="M")

# Create data frame, set index
dataframe = pd.DataFrame(index=time_index)

# Create feature with a gap of missing values
dataframe["Sales"] = [1.0,2.0,np.nan,np.nan,5.0]

# Interpolate missing values
dataframe.interpolate()
```

	Sales
2010-01-31	1.0
2010-02-28	2.0
2010-03-31	3.0
2010-04-30	4.0
2010-05-31	5.0

Alternatively, we can replace missing values with the last known value (i.e., forward-filling):

```
# Forward-fill
dataframe.ffill()
```

	Sales
2010-01-31	1.0
2010-02-28	2.0
2010-03-31	2.0
2010-04-30	2.0
2010-05-31	5.0

We can also replace missing values with the latest known value (i.e., back-filling):

```
# Back-fill
dataframe.bfill()
```

	Sales
2010-01-31	1.0
2010-02-28	2.0
2010-03-31	5.0
2010-04-30	5.0
2010-05-31	5.0

Discussion

Interpolation is a technique for filling in gaps caused by missing values by, in effect, drawing a line or curve between the known values bordering the gap and using that line or curve to predict reasonable values. Interpolation can be particularly useful when the time intervals between are constant, the data is not prone to noisy fluctuations, and the gaps caused by missing values are small. For example, in our solution a gap of two missing values was bordered by `2.0` and `5.0`. By fitting a line starting at `2.0` and ending at `5.0`, we can make reasonable guesses for the two missing values in between of `3.0` and `4.0`.

If we believe the line between the two known points is nonlinear, we can use `interpolate`'s `method` to specify the interpolation method:

```
# Interpolate missing values
dataframe.interpolate(method="quadratic")
```

	Sales
2010-01-31	1.000000
2010-02-28	2.000000
2010-03-31	3.059808
2010-04-30	4.038069
2010-05-31	5.000000

Finally, there might be cases when we have large gaps of missing values and do not want to interpolate values across the entire gap. In these cases we can use `limit` to restrict the number of interpolated values and `limit_direction` to set whether to interpolate values forward from at the last known value before the gap or vice versa:

```
# Interpolate missing values
dataframe.interpolate(limit=1, limit_direction="forward")
```

	Sales
2010-01-31	1.0
2010-02-28	2.0
2010-03-31	3.0
2010-04-30	NaN
2010-05-31	5.0

Back-filling and forward-filling can be thought of as a form of naive interpolation, where we draw a flat line from a known value and use it to fill in missing values. One (minor) advantage back- and forward-filling have over interpolation is the lack of the need for known values on *both* sides of missing value(s).

Chapter 8. Handling Images

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at feedback.mlpythoncookbook@gmail.com.

8.0 Introduction

Image classification is one of the most exciting areas of machine learning. The ability of computers to recognize patterns and objects from images is an incredibly powerful tool in our toolkit. However, before we can apply machine learning to images, we often first need to transform the raw images to features usable by our learning algorithms. As with textual data, there are also many pre-trained classifiers available for images that we can use to extract features or objects of interest to use as inputs to our own models.

To work with images, we will primarily use the Open Source Computer Vision Library (OpenCV). While there are a number of good libraries out there, OpenCV is the most popular and documented library for handling images. It can be occassionally be challenging to install, but if you run into issues there are many guides online.

Throughout this chapter we will use a set of images as examples, which are available to download on GitHub.

8.1 Loading Images

Problem

You want to load an image for preprocessing.

Solution

Use OpenCV's `imread`:

```
# Load library
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image as grayscale
image = cv2.imread("images/plane.jpg", cv2.IMREAD_GRAYSCALE)
```

If we want to view the image, we can use the Python plotting library Matplotlib:

```
# Show image
plt.imshow(image, cmap="gray"), plt.axis("off")
plt.show()
```

Discussion

Fundamentally, images are data and when we use `imread` we convert that data into a data type we are very familiar with—a NumPy array:

```
# Show data type
type(image)
```

```
numpy.ndarray
```

We have transformed the image into a matrix whose elements correspond to individual pixels. We can even take a look at the actual values of the matrix:

```
# Show image data
image
```

```
array([[140, 136, 146, ..., 132, 139, 134],
       [144, 136, 149, ..., 142, 124, 126],
       [152, 139, 144, ..., 121, 127, 134],
       ...,
       [156, 146, 144, ..., 157, 154, 151],
       [146, 150, 147, ..., 156, 158, 157],
       [143, 138, 147, ..., 156, 157, 157]], dtype=uint8)
```

The resolution of our image was 3600 × 2270, the exact dimensions of our matrix:

```
# Show dimensions
image.shape
```

```
(2270, 3600)
```

What does each element in the matrix actually represent? In grayscale images, the value of an individual

element is the pixel intensity. Intensity values range from black (0) to white (255). For example, the intensity of the top-rightmost pixel in our image has a value of 140:

```
# Show first pixel
image[0,0]
```

```
140
```

In the matrix, each element contains three values corresponding to blue, green, red values (BGR):

```
# Load image in color
image_bgr = cv2.imread("images/plane.jpg", cv2.IMREAD_COLOR)

# Show pixel
image_bgr[0,0]
```

```
array([195, 144, 111], dtype=uint8)
```

One small caveat: by default OpenCV uses BGR, but many image applications—including Matplotlib—use red, green, blue (RGB), meaning the red and the blue values are swapped. To properly display OpenCV color images in Matplotlib, we need to first convert the color to RGB (apologies to hardcopy readers):

```
# Convert to RGB
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# Show image
plt.imshow(image_rgb), plt.axis("off")
plt.show()
```

See Also

- Difference between RGB and BGR
- RGB color model

8.2 Saving Images

Problem

You want to save an image for preprocessing.

Solution

Use OpenCV's `imwrite`:

```
# Load libraries
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image as grayscale
image = cv2.imread("images/plane.jpg", cv2.IMREAD_GRAYSCALE)

# Save image
cv2.imwrite("images/plane_new.jpg", image)
```

```
True
```

Discussion

OpenCV's `imwrite` saves images to the filepath specified. The format of the image is defined by the filename's extension (*.jpg*, *.png*, etc.). One behavior to be careful about: `imwrite` will overwrite existing files without outputting an error or asking for confirmation.

8.3 Resizing Images

Problem

You want to resize an image for further preprocessing.

Solution

Use `resize` to change the size of an image:

```
# Load image
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image as grayscale
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Resize image to 50 pixels by 50 pixels
image_50x50 = cv2.resize(image, (50, 50))

# View image
plt.imshow(image_50x50, cmap="gray"), plt.axis("off")
plt.show()
```

Discussion

Resizing images is a common task in image preprocessing for two reasons. First, images come in all shapes and sizes, and to be usable as features, images must have the same dimensions. This standardization of image size does come with costs, however; images are matrices of information and when we reduce the size of the image we are reducing the size of that matrix and the information it contains. Second, machine learning can require thousands or hundreds of thousands of images. When those images are very large they can take up a lot of memory, and by resizing them we can dramatically reduce memory usage. Some common image sizes for machine learning are 32 × 32, 64 × 64, 96 × 96, and 256 × 256. In essence, the method we choose for image resizing will often be a tradeoff between the stastical performance of our model and computational cost to train it. The Pillow library offers many different options for resizing images for this reason.

8.4 Cropping Images

Problem

You want to remove the outer portion of the image to change its dimensions.

Solution

The image is encoded as a two-dimensional NumPy array, so we can crop the image easily by slicing the array:

```
# Load image
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image in grayscale
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Select first half of the columns and all rows
image_cropped = image[:,:128]
```

```
# Show image
plt.imshow(image_cropped, cmap="gray"), plt.axis("off")
plt.show()
```

Discussion

Since OpenCV represents images as a matrix of elements, by selecting the rows and columns we want to keep we are able to easily crop the image. Cropping can be particularly useful if we know that we only want to keep a certain part of every image. For example, if our images come from a stationary security camera we can crop all the images so they only contain the area of interest.

See Also

- Slicing NumPy Arrays

8.5 Blurring Images

Problem

You want to smooth out an image.

Solution

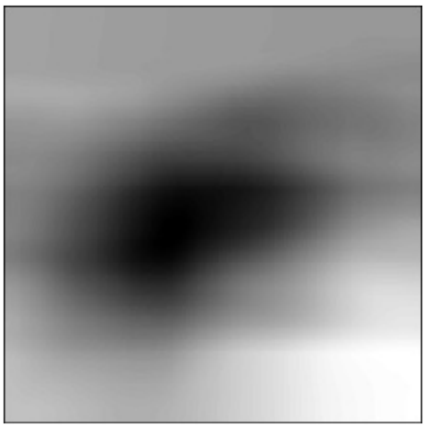
To blur an image, each pixel is transformed to be the average value of its neighbors. This neighbor and the operation performed are mathematically represented as a kernel (don't worry if you don't know what a kernel is). The size of this kernel determines the amount of blurring, with larger kernels producing smoother images. Here we blur an image by averaging the values of a 5 × 5 kernel around each pixel:

```
# Load libraries
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image as grayscale
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)
```

```
# Blur image
image_blurry = cv2.blur(image, (5,5))

# Show image
plt.imshow(image_blurry, cmap="gray"), plt.axis("off")
plt.show()
```

To highlight the effect of kernel size, here is the same blurring with a 100 × 100 kernel:

```
# Blur image
image_very_blurry = cv2.blur(image, (100,100))

# Show image
plt.imshow(image_very_blurry, cmap="gray"), plt.xticks([]), plt.yticks([])
plt.show()
```



Discussion

Kernels are widely used in image processing to do everything from sharpening to edge detection, and will come up repeatedly in this chapter. The blurring kernel we used looks like this:

```
# Create kernel
kernel = np.ones((5,5)) / 25.0

# Show kernel
kernel
```

```
array([[ 0.04,  0.04,  0.04,  0.04,  0.04],
       [ 0.04,  0.04,  0.04,  0.04,  0.04],
       [ 0.04,  0.04,  0.04,  0.04,  0.04],
       [ 0.04,  0.04,  0.04,  0.04,  0.04],
       [ 0.04,  0.04,  0.04,  0.04,  0.04]])
```

The center element in the kernel is the pixel being examined, while the remaining elements are its neighbors. Since all elements have the same value (normalized to add up to 1), each has an equal say in the resulting value of the pixel of interest. We can manually apply a kernel to an image using `filter2D` to produce a similar blurring effect:

```
# Apply kernel
image_kernel = cv2.filter2D(image, -1, kernel)

# Show image
plt.imshow(image_kernel, cmap="gray"), plt.xticks([]), plt.yticks([])
plt.show()
```

See Also

- Image Kernels Explained Visually
- Common Image Kernels

8.6 Sharpening Images

Problem

You want to sharpen an image.

Solution

Create a kernel that highlights the target pixel. Then apply it to the image using `filter2D`:

```
# Load libraries
import cv2
import numpy as np
from matplotlib import pyplot as plt
```

```
# Load image as grayscale
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Create kernel
kernel = np.array([[0, -1, 0],
                   [-1, 5,-1],
                   [0, -1, 0]])

# Sharpen image
image_sharp = cv2.filter2D(image, -1, kernel)

# Show image
plt.imshow(image_sharp, cmap="gray"), plt.axis("off")
plt.show()
```

Discussion

Sharpening works similarly to blurring, except instead of using a kernel to average the neighboring values, we constructed a kernel to highlight the pixel itself. The resulting effect makes contrasts in edges stand out more in the image.

8.7 Enhancing Contrast

Problem

We want to increase the contrast between pixels in an image.

Solution

Histogram equalization is a tool for image processing that can make objects and shapes stand out. When we have a grayscale image, we can apply OpenCV's `equalizeHist` directly on the image:

```
# Load libraries
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image
```

```
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Enhance image
image_enhanced = cv2.equalizeHist(image)

# Show image
plt.imshow(image_enhanced, cmap="gray"), plt.axis("off")
plt.show()
```

However, when we have a color image, we first need to convert the image to the YUV color format. The Y is the luma, or brightness, and U and V denote the color. After the conversion, we can apply `equalizeHist` to the image and then convert it back to BGR or RGB:

```
# Load image
image_bgr = cv2.imread("images/plane.jpg")

# Convert to YUV
image_yuv = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2YUV)

# Apply histogram equalization
image_yuv[:, :, 0] = cv2.equalizeHist(image_yuv[:, :, 0])

# Convert to RGB
image_rgb = cv2.cvtColor(image_yuv, cv2.COLOR_YUV2RGB)

# Show image
plt.imshow(image_rgb), plt.axis("off")
plt.show()
```

Discussion

While a detailed explanation of how histogram equalization works is beyond the scope of this book, the short explanation is that it transforms the image so that it uses a wider range of pixel intensities.

While the resulting image often does not look "realistic," we need to remember that the image is just a visual representation of the underlying data. If histogram equalization is able to make objects of interest more distinguishable from other objects or backgrounds (which is not always the case), then it can be a valuable addition to our image preprocessing pipeline.

8.8 Isolating Colors

Problem

You want to isolate a color in an image.

Solution

Define a range of colors and then apply a mask to the image:

```
# Load libraries
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image
image_bgr = cv2.imread('images/plane_256x256.jpg')

# Convert BGR to HSV
image_hsv = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2HSV)

# Define range of blue values in HSV
lower_blue = np.array([50,100,50])
upper_blue = np.array([130,255,255])

# Create mask
mask = cv2.inRange(image_hsv, lower_blue, upper_blue)

# Mask image
```

```
image_bgr_masked = cv2.bitwise_and(image_bgr, image_bgr, mask=mask)

# Convert BGR to RGB
image_rgb = cv2.cvtColor(image_bgr_masked, cv2.COLOR_BGR2RGB)

# Show image
plt.imshow(image_rgb), plt.axis("off")
plt.show()
```

Discussion

Isolating colors in OpenCV is straightforward. First we convert an image into HSV (hue, saturation, and value). Second, we define a range of values we want to isolate, which is probably the most difficult and time-consuming part. Third, we create a mask for the image (we will only keep the white areas):

```
# Show image
plt.imshow(mask, cmap='gray'), plt.axis("off")
plt.show()
```

Finally, we apply the mask to the image using `bitwise_and` and convert to our desired output format.

8.9 Binarizing Images

Problem

Given an image, you want to output a simplified version.

Solution

Thresholding is the process of setting pixels with intensity greater than some value to be white and less than the value to be black. A more advanced technique is *adaptive thresholding*, where the threshold value for a pixel is determined by the pixel intensities of its neighbors. This can be helpful when lighting conditions change over different regions in an image:

```
# Load libraries
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image as grayscale
image_grey = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Apply adaptive thresholding
max_output_value = 255
neighborhood_size = 99
subtract_from_mean = 10
image_binarized = cv2.adaptiveThreshold(image_grey,
                                        max_output_value,
                                        cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                        cv2.THRESH_BINARY,
                                        neighborhood_size,
                                        subtract_from_mean)

# Show image
plt.imshow(image_binarized, cmap="gray"), plt.axis("off")
plt.show()
```

Discussion

Our solution has four important arguments in `adaptiveThreshold`. `max_output_value` simply determines the maximum intensity of the output pixel intensities. `cv2.ADAPTIVE_THRESH_GAUSSIAN_C` sets a pixel's threshold to be a weighted sum of the neighboring pixel intensities. The weights are determined by a Gaussian window. Alternatively we could set the threshold to simply the mean of the

neighboring pixels with `cv2.ADAPTIVE_THRESH_MEAN_C`:

```
# Apply cv2.ADAPTIVE_THRESH_MEAN_C
image_mean_threshold = cv2.adaptiveThreshold(image_grey,
                                             max_output_value,
                                             cv2.ADAPTIVE_THRESH_MEAN_C,
                                             cv2.THRESH_BINARY,
                                             neighborhood_size,
                                             subtract_from_mean)

# Show image
plt.imshow(image_mean_threshold, cmap="gray"), plt.axis("off")
plt.show()
```

The last two parameters are the block size (the size of the neighborhood used to determine a pixel's threshold) and a constant subtracted from the calculated threshold (used to manually fine-tune the threshold).

A major benefit of thresholding is *denoising* an image—keeping only the most important elements. For example, thresholding is often applied to photos of printed text to isolate the letters from the page.

8.10 Removing Backgrounds

Problem

You want to isolate the foreground of an image.

Solution

Mark a rectangle around the desired foreground, then run the GrabCut algorithm:

```
# Load library
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image and convert to RGB
image_bgr = cv2.imread('images/plane_256x256.jpg')
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)
```

```
# Rectangle values: start x, start y, width, height
rectangle = (0, 56, 256, 150)

# Create initial mask
mask = np.zeros(image_rgb.shape[:2], np.uint8)

# Create temporary arrays used by grabCut
bgdModel = np.zeros((1, 65), np.float64)
fgdModel = np.zeros((1, 65), np.float64)

# Run grabCut
cv2.grabCut(image_rgb, # Our image
            mask, # The Mask
            rectangle, # Our rectangle
            bgdModel, # Temporary array for background
            fgdModel, # Temporary array for background
            5, # Number of iterations
            cv2.GC_INIT_WITH_RECT) # Initiative using our rectangle

# Create mask where sure and likely backgrounds set to 0, otherwise 1
mask_2 = np.where((mask==2) | (mask==0), 0, 1).astype('uint8')

# Multiply image with new mask to subtract background
image_rgb_nobg = image_rgb * mask_2[:, :, np.newaxis]

# Show image
plt.imshow(image_rgb_nobg), plt.axis("off")
plt.show()
```

Discussion

The first thing we notice is that even though GrabCut did a pretty good job, there are still areas of background left in the image. We could go back and manually mark those areas as background, but in the real world we have thousands of images and manually fixing them individually is not feasible. Therefore, we would do well by simply accepting that the image data will still contain some background noise.

In our solution, we start out by marking a rectangle around the area that contains the foreground. GrabCut assumes everything outside this rectangle to be background and uses that information to figure out what is likely background inside the square (to learn how the algorithm does this, check out external resources like *http://bit.ly/2wgbPIS*). Then a mask is created that denotes the different definitely/likely background/foreground regions:

```
# Show mask
plt.imshow(mask, cmap='gray'), plt.axis("off")
plt.show()
```

The black region is the area outside our rectangle that is assumed to be definitely background. The gray area is what GrabCut considered likely background, while the white area is likely foreground.

This mask is then used to create a second mask that merges the black and gray regions:

```
# Show mask
plt.imshow(mask_2, cmap='gray'), plt.axis("off")
plt.show()
```

The second mask is then applied to the image so that only the foreground remains.

8.11 Detecting Edges

Problem

You want to find the edges in an image.

Solution

Use an edge detection technique like the Canny edge detector:

```
# Load library
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image as grayscale
image_gray = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Calculate median intensity
median_intensity = np.median(image_gray)

# Set thresholds to be one standard deviation above and below median intensity
lower_threshold = int(max(0, (1.0 - 0.33) * median_intensity))
upper_threshold = int(min(255, (1.0 + 0.33) * median_intensity))

# Apply canny edge detector
image_canny = cv2.Canny(image_gray, lower_threshold, upper_threshold)

# Show image
plt.imshow(image_canny, cmap="gray"), plt.axis("off")
plt.show()
```

Discussion

Edge detection is a major topic of interest in computer vision. Edges are important because they are areas of high information. For example, in our image one patch of sky looks very much like another and is unlikely to contain unique or interesting information. However, patches where the background sky meets the airplane contain a lot of information (e.g., an object's shape). Edge detection allows us to remove low-information areas and isolate the areas of images containing the most information.

There are many edge detection techniques (Sobel filters, Laplacian edge detector, etc.). However, our solution uses the commonly used Canny edge detector. How the Canny detector works is too detailed for this book, but there is one point that we need to address. The Canny detector requires two parameters denoting low and high gradient threshold values. Potential edge pixels between the low and high thresholds are considered weak edge pixels, while those above the high threshold are considered strong edge pixels. OpenCV's `Canny` method includes the low and high thresholds as required

parameters. In our solution, we set the lower and upper thresholds to be one standard deviation below and above the image's median pixel intensity. However, there are often cases when we might get better results if we used a good pair of low and high threshold values through manual trial and error using a few images before running `Canny` on our entire collection of images.

See Also

- Canny Edge Detector
- Canny Edge Detection Auto Thresholding

8.12 Detecting Corners

Problem

You want to detect the corners in an image.

Solution

Use OpenCV's implementation of the Harris corner detector, `cornerHarris`:

```
# Load libraries
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image as grayscale
image_bgr = cv2.imread("images/plane_256x256.jpg")
image_gray = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)
image_gray = np.float32(image_gray)

# Set corner detector parameters
block_size = 2
aperture = 29
free_parameter = 0.04

# Detect corners
detector_responses = cv2.cornerHarris(image_gray,
                                      block_size,
                                      aperture,
                                      free_parameter)

# Large corner markers
detector_responses = cv2.dilate(detector_responses, None)

# Only keep detector responses greater than threshold, mark as white
threshold = 0.02
image_bgr[detector_responses >
          threshold *
          detector_responses.max()] = [255,255,255]

# Convert to grayscale
image_gray = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)

# Show image
plt.imshow(image_gray, cmap="gray"), plt.axis("off")
plt.show()
```

Discussion

The Harris corner detector is a commonly used method of detecting the intersection of two edges. Our interest in detecting corners is motivated by the same reason as for deleting edges: corners are points of high information. A complete explanation of the Harris corner detector is available in the external resources at the end of this recipe, but a simplified explanation is that it looks for windows (also called *neighborhoods* or *patches*) where small movements of the window (imagine shaking the window) creates big changes in the contents of the pixels inside the window. `cornerHarris` contains three important parameters that we can use to control the edges detected. First, `block_size` is the size of the neighbor around each pixel used for corner detection. Second, `aperture` is the size of the Sobel kernel used (don't worry if you don't know what that is), and finally there is a free parameter where larger values correspond to identifying softer corners.

The output is a grayscale image depicting potential corners:

```
# Show potential corners
plt.imshow(detector_responses, cmap='gray'), plt.axis("off")
plt.show()
```

We then apply thresholding to keep only the most likely corners. Alternatively, we can use a similar

detector, the Shi-Tomasi corner detector, which works in a similar way to the Harris detector (`goodFeaturesToTrack`) to identify a fixed number of strong corners. `goodFeaturesToTrack` takes three major parameters—the number of corners to detect, the minimum quality of the corner (0 to 1), and the minimum Euclidean distance between corners:

```
# Load images
image_bgr = cv2.imread('images/plane_256x256.jpg')
image_gray = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)

# Number of corners to detect
corners_to_detect = 10
minimum_quality_score = 0.05
minimum_distance = 25

# Detect corners
corners = cv2.goodFeaturesToTrack(image_gray,
                                  corners_to_detect,
                                  minimum_quality_score,
                                  minimum_distance)
corners = np.int16(corners)

# Draw white circle at each corner
for corner in corners:
    x, y = corner[0]
    cv2.circle(image_bgr, (x,y), 10, (255,255,255), -1)

# Convert to grayscale
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2GRAY)

# Show image
plt.imshow(image_rgb, cmap='gray'), plt.axis("off")
plt.show()
```

See Also

- OpenCV's cornerHarris
- OpenCV's goodFeaturesToTrack

8.13 Creating Features for Machine Learning

Problem

You want to convert an image into an observation for machine learning.

Solution

Use NumPy's `flatten` to convert the multidimensional array containing an image's data into a vector containing the observation's values:

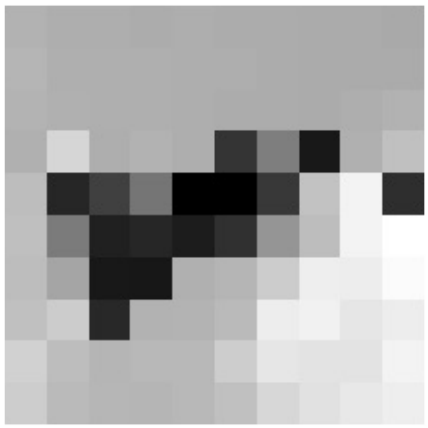
```
# Load image
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image as grayscale
image = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Resize image to 10 pixels by 10 pixels
image_10x10 = cv2.resize(image, (10, 10))

# Convert image data to one-dimensional vector
image_10x10.flatten()
```

```
array([133, 130, 130, 129, 130, 129, 129, 128, 128, 127, 135, 131, 131,
       131, 130, 130, 129, 128, 128, 128, 134, 132, 131, 131, 130, 129,
       129, 128, 130, 133, 132, 158, 130, 133, 130,  46,  97,  26, 132,
       143, 141,  36,  54,  91,   9,   9,  49, 144, 179,  41, 142,  95,
        32,  36,  29,  43, 113, 141, 179, 187, 141, 124,  26,  25, 132,
       135, 151, 175, 174, 184, 143, 151,  38, 133, 134, 139, 174, 177,
       169, 174, 155, 141, 135, 137, 137, 152, 169, 168, 168, 179, 152,
       139, 136, 135, 137, 143, 159, 166, 171, 175], dtype=uint8)
```

Discussion

Images are presented as a grid of pixels. If an image is in grayscale, each pixel is presented by one value (i.e., pixel intensity: `1` if white, `0` if black). For example, imagine we have a 10 × 10–pixel image:

```
plt.imshow(image_10x10, cmap="gray"), plt.axis("off")
plt.show()
```



In this case the dimensions of the images data will be 10 × 10:

```
image_10x10.shape
```

```
(10, 10)
```

And if we flatten the array, we get a vector of length 100 (10 multiplied by 10):

```
image_10x10.flatten().shape
```

```
(100,)
```

This is the feature data for our image that can be joined with the vectors from other images to create the data we will feed to our machine learning algorithms.

If the image is in color, instead of each pixel being represented by one value, it is represented by multiple values (most often three) representing the channels (red, green, blue, etc.) that blend to make the final color of that pixel. For this reason, if our 10 × 10 image is in color, we will have 300 feature values for each observation:

```
# Load image in color
image_color = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_COLOR)

# Resize image to 10 pixels by 10 pixels
image_color_10x10 = cv2.resize(image_color, (10, 10))

# Convert image data to one-dimensional vector, show dimensions
image_color_10x10.flatten().shape
```

```
(300,)
```

One of the major challenges of image processing and computer vision is that since every pixel location in a collection of images is a feature, as the images get larger, the number of features explodes:

```
# Load image in grayscale
image_256x256_gray = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_GRAYSCALE)

# Convert image data to one-dimensional vector, show dimensions
image_256x256_gray.flatten().shape
```

```
(65536,)
```

And the number of features only intensifies when the image is in color:

```
# Load image in color
image_256x256_color = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_COLOR)

# Convert image data to one-dimensional vector, show dimensions
image_256x256_color.flatten().shape
```

```
(196608,)
```

As the output shows, even a small color image has almost 200,000 features, which can cause problems when we are training our models because the number of features might far exceed the number of observations.

This problem will motivate dimensionality strategies discussed in a later chapter, which attempt to reduce the number of features while not losing an excessive amount of information contained in the data.

8.14 Encoding Convolutions as a Feature

Problem

You want to use convolutions as input to your machine learning model.

8.15 Encoding Color Histograms as Features

Problem

You want to create a set of features representing the colors appearing in an image.

Solution

Compute the histograms for each color channel:

```
# Load libraries
import cv2
import numpy as np
from matplotlib import pyplot as plt

# Load image
image_bgr = cv2.imread("images/plane_256x256.jpg", cv2.IMREAD_COLOR)

# Convert to RGB
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# Create a list for feature values
features = []

# Calculate the histogram for each color channel
colors = ("r","g","b")

# For each channel: calculate histogram and add to feature value list
for i, channel in enumerate(colors):
    histogram = cv2.calcHist([image_rgb], # Image
                             [i], # Index of channel
                             None, # No mask
                             [256], # Histogram size
                             [0,256]) # Range
    features.extend(histogram)

# Create a vector for an observation's feature values
observation = np.array(features).flatten()

# Show the observation's value for the first five features
observation[0:5]
```

```
array([ 1008.,   217.,   184.,   165.,   116.], dtype=float32)
```

Discussion

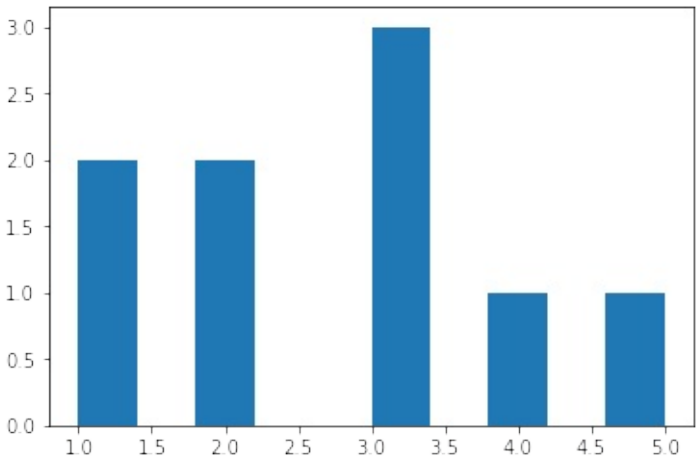
In the RGB color model, each color is the combination of three color channels (i.e., red, green, blue). In turn, each channel can take on one of 256 values (represented by an integer between 0 and 255). For example, the top-leftmost pixel in our image has the following channel values:

```
# Show RGB channel values
image_rgb[0,0]
```

```
array([107, 163, 212], dtype=uint8)
```

A histogram is a representation of the distribution of values in data. Here is a simple example:

```
# Import pandas
import pandas as pd

# Create some data
data = pd.Series([1, 1, 2, 2, 3, 3, 3, 4, 5])

# Show the histogram
data.hist(grid=False)
plt.show()
```

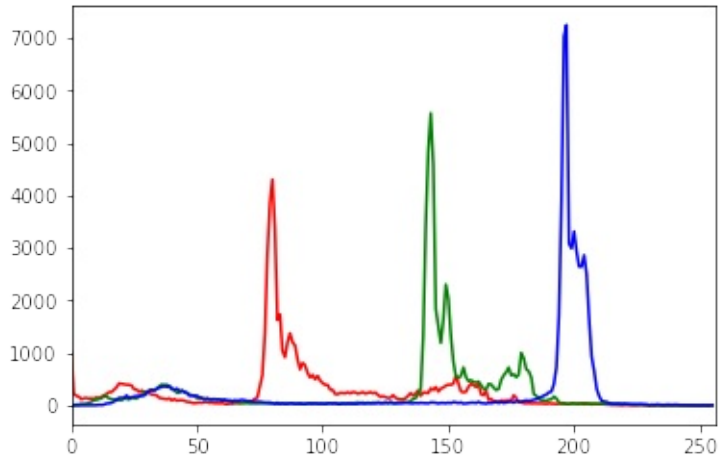


In this example, we have some data with two 1s, two 2s, three 3s, one 4, and one 5. In the histogram, each bar represents the number of times each value (1, 2, etc.) appears in our data.

We can apply this same technique to each of the color channels, but instead of five possible values, we have 256 (the range of possible values for a channel value). The x-axis represents the 256 possible channel values, and the y-axis represents the number of times a particular channel value appears across all pixels in an image:

```
# Calculate the histogram for each color channel
colors = ("r","g","b")

# For each channel: calculate histogram, make plot
for i, channel in enumerate(colors):
    histogram = cv2.calcHist([image_rgb], # Image
                        [i], # Index of channel
                        None, # No mask
                        [256], # Histogram size
                        [0,256]) # Range
    plt.plot(histogram, color = channel)
    plt.xlim([0,256])
```

```
# Show plot
plt.show()
```



As we can see in the histogram, barely any pixels contain the blue channel values between 0 and \~180, while many pixels contain blue channel values between ~190 and ~210. This distribution of channel values is shown for all three channels. The histogram, however, is not simply a visualization; it is 256 features for each color channel, making for 768 total features representing the distribution of colors in an image.

See Also

- Histogram
- pandas Histogram documentation
- OpenCV Histogram tutorial

8.16 Using Pretrained Embeddings as a Feature

Problem

You want to load pretrained embeddings from an existing model in Pytorch and use it as input to one of your own models.

Solution

Use `torchvision.models` to select a model and then retrieve an embedding from it for a given image:

```
import cv2
import numpy as np
import torch
from torchvision import transforms
import torchvision.models as models

# Load image
image_bgr = cv2.imread("images/plane.jpg", cv2.IMREAD_COLOR)
```

```
# Convert to pytorch data type
convert_tensor = transforms.ToTensor()
pytorch_image = convert_tensor(np.array(image_rgb))

# Load the pretrained model
model = models.resnet18(pretrained=True)

# Select the specific layer of the model we want output from
layer = model._modules.get('avgpool')

# Set model to evaluation mode
model.eval()

# Infer the embedding with the no_grad option
with torch.no_grad():
    embedding = model(pytorch_image.unsqueeze(0))

print(embedding.shape)
```

```
torch.Size([1, 1000])
```

Discussion

In the ML space, transfer learning is often defined as using information learned from one task as input to another task. Instead of starting from "0", we can use representations already learned from large pretrained image models (such as Resnet) to get a "head start" on our own machine learning models. More intuitively, you can understand how we could use the weights of a model trained to recognize cats as a good "start" for a model we want to train to recognize dogs. By sharing information form one model to another, we can leverage the information learned from other datasets and model architectures without the overhead of training a model from scratch.

The entire application of transfer learning in computer vision is outside the scope of this book, however there are many different ways we can extract embeddings-based representations of images outside of Pytorch. In TensorFlow, another common library for deep learning, we can use `tensorflow_hub`.

```
import cv2
import tensorflow as tf
import tensorflow_hub as hub

# Load image
image_bgr = cv2.imread("images/plane.jpg", cv2.IMREAD_COLOR)
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# Convert to tensorflow data type
tf_image = tf.image.convert_image_dtype([image_rgb], tf.float32)

# Create the model and get embeddings using the inception V1 model
embedding_model = hub.KerasLayer("https://tfhub.dev/google/imagenet/inception_v1/feature_vector/5")
embeddings = embedding_model(tf_image)

# Print the shape of the embedding
print(embeddings.shape)
```

```
(1, 1024)
```

See Also

- Transfer learning with Pytorch

- Tensorflow Hub

8.17 Detecting Objects with OpenCV

Problem

You want to detect objects in images using pretrained cascade classifiers with OpenCV.

Solution

Download and run one of OpenCV's Haar Cascade classifiers. In this case, we use a pretrained face detection model to detect and draw a rectangle around a face in an image:

```
# Import libraries
import cv2
from matplotlib import pyplot as plt

# first run : `mkdir models && cd models`
# `wget
https://raw.githubusercontent.com/opencv/opencv/4.x/data/haarcascades/haarcascade_frontalface_default.xml`
face_cascade = cv2.CascadeClassifier()
face_cascade.load(cv2.samples.findFile("models/haarcascade_frontalface_default.xml"))

# Load image
image_bgr = cv2.imread("images/kyle_pic.jpg", cv2.IMREAD_COLOR)
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# Detect faces and draw a rectangle
faces = face_cascade.detectMultiScale(image_rgb)
for (x,y,w,h) in faces:
    cv2.rectangle(image_rgb, (x, y),
                        (x + h, y + w),
                        (0, 255, 0), 5)

# Show the image
plt.subplot(1, 1, 1)
plt.imshow(image_rgb)
plt.show()
```

```
image::images/kyle_face_detected.png[]
```

Discussion

Haar Cascade classifiers are machine learning models used to learn a set of image features (specifically Haar features) that can be used to detect objects in images. The features themselves are simple rectangular features that determined by calulcating the difference in sums between rectangular regions. Subsequently, a gradient boosting algorithm is applied to learn the most important features and finally create a relatively strong model using cascading classifiers.

While the details of this process are out-of-scope for this book, it's noteworthy that these pretrained models can be easily downloaded from places such as the OpenCV Github as XML files, and applied to images without training a model yourself. This is useful in cases where you want to add simple binary image features such as `contains_face` (or any other object) to your data.

See Also

- OpenCV cascade classifier tutorial

8.18 Classifying Images with Pytorch

Problem

You want to classify images using pretrained deep learning models in Pytorch.

Solution

Use `torchvision.models` to select a pretrained image classification model and feed the image through it.

```
import cv2
import json
import numpy as np
import torch
from torchvision import transforms
from torchvision.models import resnet18
import urllib.request

# Get imagenet classes
with urllib.request.urlopen("https://raw.githubusercontent.com/raghakot/keras-
vis/master/resources/imagenet_class_index.json") as url:
    imagenet_class_index = json.load(url)

# Instantiate pretrained model
model = resnet18(pretrained=True)

# Load image
image_bgr = cv2.imread("images/plane.jpg", cv2.IMREAD_COLOR)
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

# Convert to pytorch data type
convert_tensor = transforms.ToTensor()
pytorch_image = convert_tensor(np.array(image_rgb))

# Set model to evaluation mode
model.eval()

# Make a prediction
prediction = model(pytorch_image.unsqueeze(0))

# Get the index of the highest predicted probability
_, index = torch.max(prediction, 1)

# Convert that to a percentage value
percentage = torch.nn.functional.softmax(prediction, dim=1)[0] * 100

# Print the name of the item at the index along with the percent confidence
print(imagenet_class_index[str(index.tolist()[0])][1], percentage[index.tolist()[0]].item())
```

```
airship 6.0569939613342285
```

Discussion

Many pretrained deep learning models for image classification are easily available via both Pytorch and TensorFlow. In the example above, we used ResNet18, a deep neural network architecture that was

trained on the ImageNet dataset that is 18 layers deep. Deeper ResNet models, such as ResNet101 and ResNet152 are also available in Pytorch - and beyond that there are many other image models to choose from. Models trained on the imagenet dataset are able output predicted probabilities for all classes defined in the `imagenet_class_index` variable above, which we download from Github.

Like the facial recognition example in OpenCV, we can use the predicted image classes as downstream features for future ML models, or handy metadata tags that add more information to our images.

See Also

- Pytorch models and pretrained weights

Chapter 9. Dimensionality Reduction Using Feature Extraction

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at feedback.mlpythoncookbook@gmail.com.

9.0 Introduction

It is common to have access to thousands and even hundreds of thousands of features. For example, in Chapter 8 we transformed a 256 × 256–pixel color image into 196,608 features. Furthermore, because each of these pixels can take one of 256 possible values, there ends up being 256^{196608} different configurations our observation can take. Many machine learning algorithms have trouble learning from such data, because we will practically never be able to collect enough observations for the algorithms to operate correctly. Even in more tabular, structured datasets we can easily end up with thousands of features after the feature engineering process.

Fortunately, not all features are created equal, and the goal of feature extraction for dimensionality reduction is to transform our set of features, $p_{original}$, such that we end up with a new set, p_{new}, where $p_{original} > p_{new}$, while still keeping much of the underlying information. Put another way, we reduce the number of features with only a small loss in our data's ability to generate high-quality predictions. In this chapter, we will cover a number of feature extraction techniques to do just this.

One downside of the feature extraction techniques we discuss is that the new features we generate will not be interpretable by humans. They will contain as much or nearly as much ability to train our models, but will appear to the human eye as a collection of random numbers. If we wanted to maintain our ability to interpret our models, dimensionality reduction through feature selection is a better option.

9.1 Reducing Features Using Principal Components

Problem

Given a set of features, you want to reduce the number of features while retaining the variance (important information) in the data.

Solution

Use principal component analysis with scikit's `PCA`:

```
# Load libraries
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn import datasets

# Load the data
digits = datasets.load_digits()

# Standardize the feature matrix
features = StandardScaler().fit_transform(digits.data)

# Create a PCA that will retain 99% of variance
pca = PCA(n_components=0.99, whiten=True)

# Conduct PCA
features_pca = pca.fit_transform(features)

# Show results
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_pca.shape[1])
```

```
Original number of features: 64
Reduced number of features: 54
```

Discussion

Principal component analysis (PCA) is a popular linear dimensionality reduction technique. PCA projects observations onto the (hopefully fewer) principal components of the feature matrix that retain the most *variance* in the data - which practically means we retain information. PCA is an unsupervised technique, meaning that it does not use the information from the target vector and instead only considers the feature matrix.

For a mathematical description of how PCA works, see the external resources listed at the end of this recipe. However, we can understand the intuition behind PCA using a simple example. In the following figure, our data contains two features, x_1 and x_2. Looking at the visualization, it should be clear that observations are spread out like a cigar, with a lot of length and very little height. More specifically, we can say that the variance of the "length" is significantly greater than the "height." Instead of length and height, we refer to the "directions" with the most variance as the first principal component and the "direction" with the second-most variance as the second principal component (and so on).

If we wanted to reduce our features, one strategy would be to project all observations in our 2D space onto the 1D principal component. We would lose the information captured in the second principal component, but in some situations that would be an acceptable trade-off. This is PCA.

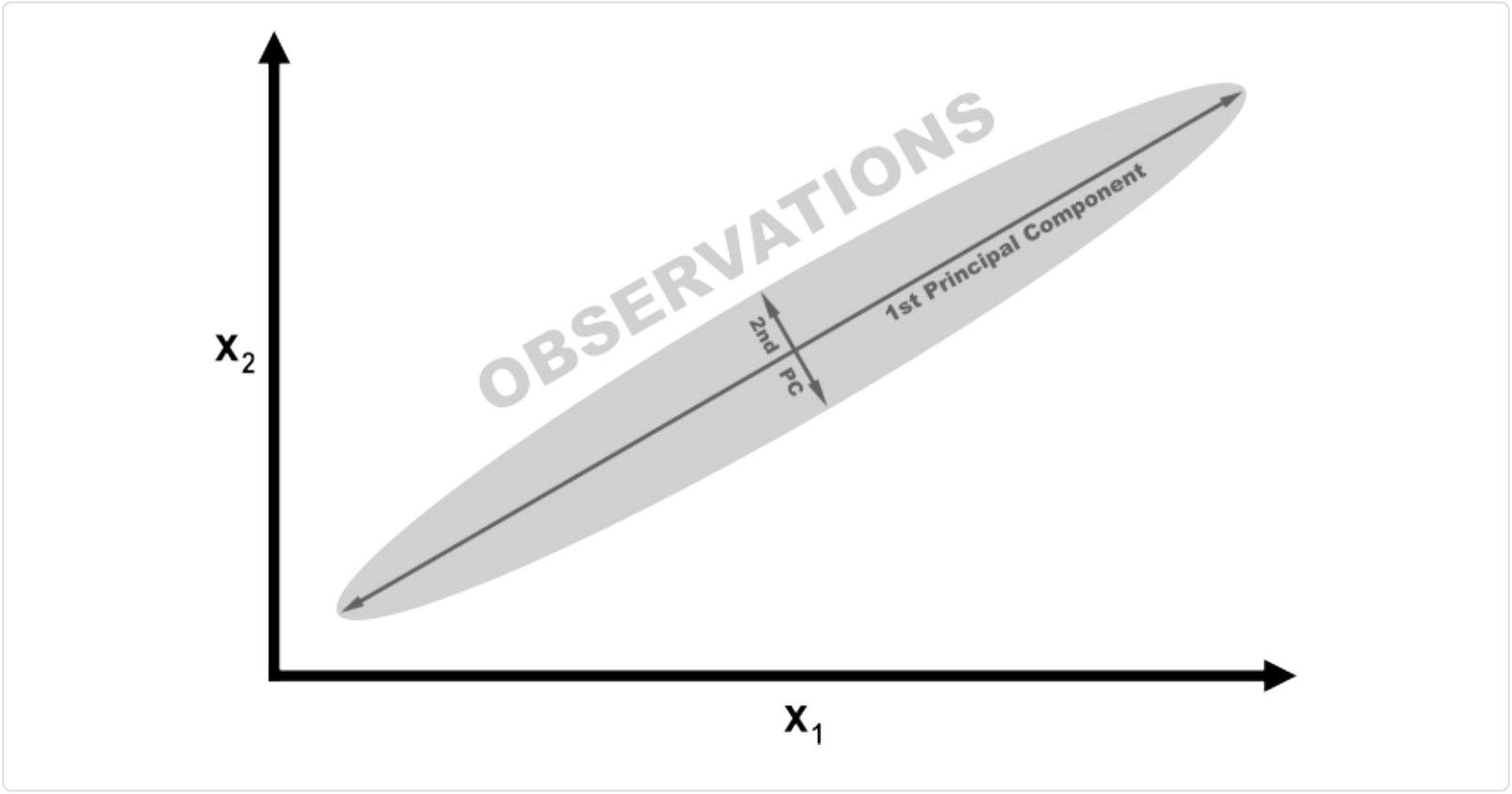


Figure 1: The first and second principal components of PCA

PCA is implemented in scikit-learn using the `PCA` class. `n_components` has two operations, depending on the argument provided. If the argument is greater than 1, `n_components` will return that many features. This leads to the question of how to select the number of features that is optimal. Fortunately for us, if the argument to `n_components` is between 0 and 1, `pca` returns the minimum amount of features that retain that much variance. It is common to use values of 0.95 and 0.99, meaning 95% and 99% of the variance of the original features has been retained, respectively. `whiten=True` transforms the values of each principal component so that they have zero mean and unit variance. Another parameter and argument is `svd_solver="randomized"`, which implements a stochastic algorithm to find the first principal components in often significantly less time.

The output of our solution shows that PCA let us reduce our dimensionality by 10 features while still retaining 99% of the information (variance) in the feature matrix.

See Also

- scikit-learn documentation on PCA
- Choosing the Number of Principal Components
- Principal component analysis with linear algebra

9.2 Reducing Features When Data Is Linearly Inseparable

Problem

You suspect you have linearly inseparable data and want to reduce the dimensions.

Solution

Use an extension of principal component analysis that uses kernels to allow for non-linear dimensionality reduction:

```
# Load libraries
from sklearn.decomposition import PCA, KernelPCA
from sklearn.datasets import make_circles

# Create linearly inseparable data
features, _ = make_circles(n_samples=1000, random_state=1, noise=0.1, factor=0.1)

# Apply kernal PCA with radius basis function (RBF) kernel
kpca = KernelPCA(kernel="rbf", gamma=15, n_components=1)
features_kpca = kpca.fit_transform(features)

print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_kpca.shape[1])
```

```
Original number of features: 2
Reduced number of features: 1
```

Discussion

PCA is able to reduce the dimensionality of our feature matrix (e.g., the number of features). Standard PCA uses linear projection to reduce the features. If the data is linearly separable (i.e., you can draw a straight line or hyperplane between different classes) then PCA works well. However, if your data is not linearly separable (e.g., you can only separate classes using a curved decision boundary), the linear transformation will not work as well. In our solution we used scikit-learn's `make_circles` to generate a simulated dataset with a target vector of two classes and two features. `make_circles` makes linearly inseparable data; specifically, one class is surrounded on all sides by the other class.

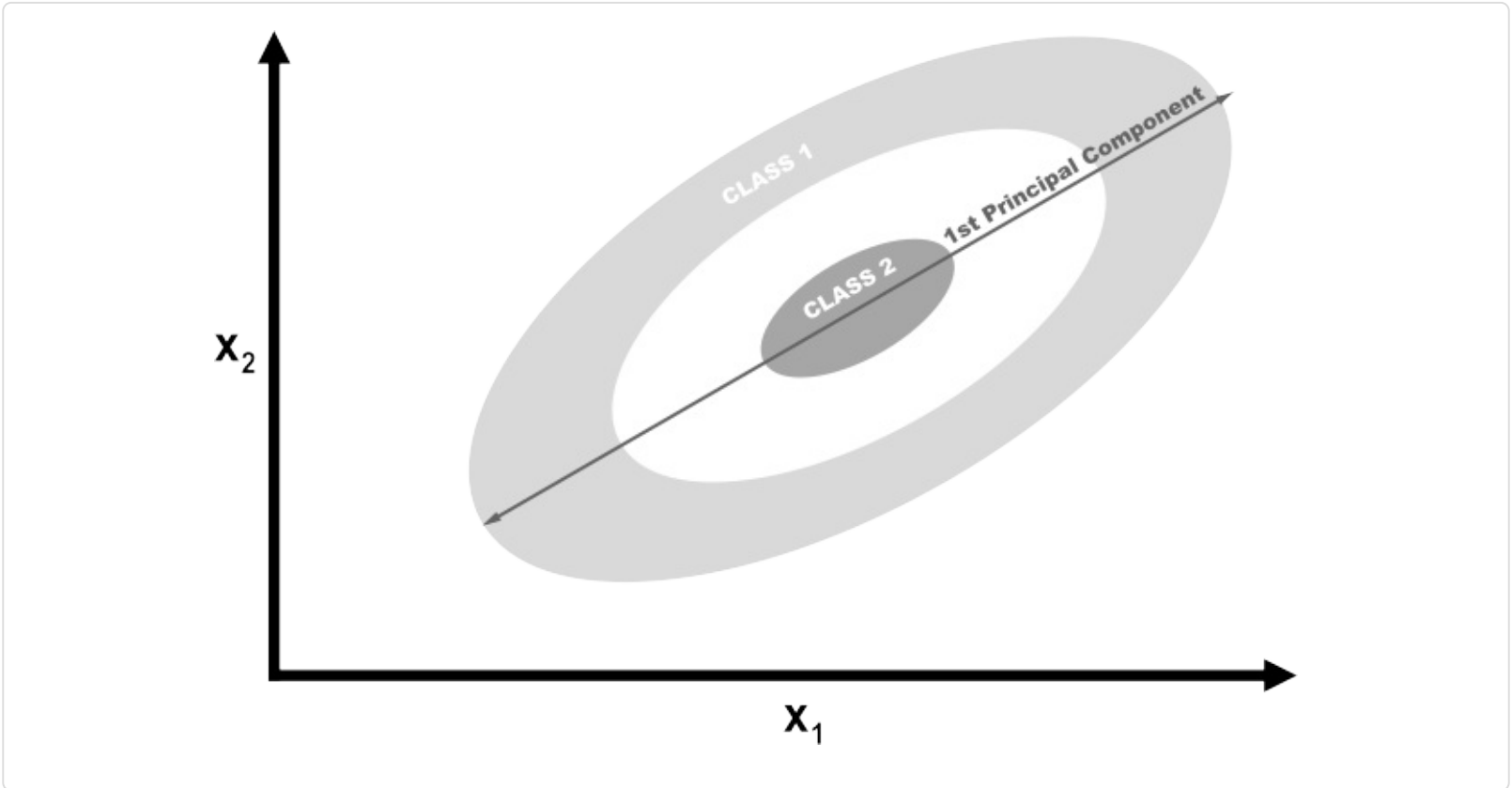


Figure 2: The first principle component projected on linearly inseparable data

If we used linear PCA to reduce the dimensions of our data, the two classes would be linearly projected onto the first principal component such that they would become intertwined.

Figure 3: The first principle component of linearly inseparable data without kernel PCA

Ideally, we would want a transformation that would both reduce the dimensions and also make the data linearly separable. Kernel PCA can do both.

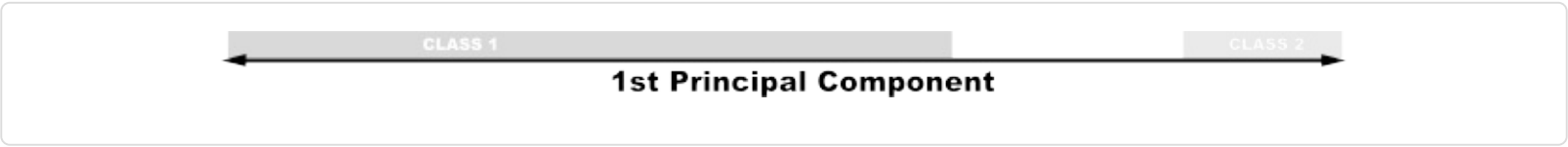


Figure 4: The first principle component of linearly inseparable data with kernel PCA

Kernels allow us to project the linearly inseparable data into a higher dimension where it is linearly separable; this is called the kernel trick. Don't worry if you don't understand the details of the kernel trick; just think of kernels as different ways of projecting the data. There are a number of kernels we can use in scikit-learn's `kernelPCA` class, specified using the `kernel` parameter. A common kernel to use is the Gaussian radial basis function kernel `rbf`, but other options are the polynomial kernel (`poly`) and sigmoid kernel (`sigmoid`). We can even specify a linear projection (`linear`), which will produce the same results as standard PCA.

One downside of kernel PCA is that there are a number of parameters we need to specify. For example, in Recipe 9.1 we set `n_components` to `0.99` to make `PCA` select the number of components to retain 99% of the variance. We don't have this option in kernel PCA. Instead we have to define the number of components (e.g., `n_components=1`). Furthermore, kernels come with their own hyperparameters that we will have to set; for example, the radial basis function requires a `gamma` value.

So how do we know which values to use? Through trial and error. Specifically we can train our machine learning model multiple times, each time with a different kernel or different value of the parameter. Once we find the combination of values that produces the highest quality predicted values, we are done. This is a common theme in machine learning, and we will learn about this strategy in depth in [Link to Come].

See Also

- scikit-learn documentation on Kernel PCA
- Kernel tricks and nonlinear dimensionality reduction via RBF kernel PCA

9.3 Reducing Features by Maximizing Class Separability

Problem

You want to reduce the number of features to be used by a classifier by maximizing the separation between the classes.

Solution

Try linear discriminant analysis (LDA) to project the features onto component axes that maximize the separation of classes:

```
# Load libraries
from sklearn import datasets
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Load Iris flower dataset:
iris = datasets.load_iris()
features = iris.data
target = iris.target

# Create and run an LDA, then use it to transform the features
lda = LinearDiscriminantAnalysis(n_components=1)
features_lda = lda.fit(features, target).transform(features)

# Print the number of features
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_lda.shape[1])
```

```
Original number of features: 4
Reduced number of features: 1
```

We can use `explained_variance_ratio_` to view the amount of variance explained by each component. In our solution the single component explained over 99% of the variance:

```
lda.explained_variance_ratio_
```

```
array([0.9912126])
```

Discussion

LDA is a classification that is also a popular technique for dimensionality reduction. LDA works similarly to principal component analysis (PCA) in that it projects our feature space onto a lower-dimensional space. However, in PCA we were only interested in the component axes that maximize the variance in the data, while in LDA we have the additional goal of maximizing the differences between classes. In this pictured example, we have data comprising two target classes and two features. If we project the data onto the y-axis, the two classes are not easily separable (i.e., they overlap), while if we project the data onto the x-axis, we are left with a feature vector (i.e., we reduced our dimensionality by one) that still preserves class separability. In the real world, of course, the relationship between the classes will be more complex and the dimensionality will be higher, but the concept remains the same.

In scikit-learn, LDA is implemented using `LinearDiscriminantAnalysis`, which includes a parameter, `n_components`, indicating the number of features we want returned. To figure out what argument value to use with `n_components` (e.g., how many parameters to keep), we can take advantage of the fact that `explained_variance_ratio_` tells us the variance explained by each outputted feature and is a sorted array. For example:

```
lda.explained_variance_ratio_
```

```
array([0.9912126])
```

Specifically, we can run `LinearDiscriminantAnalysis` with `n_components` set to `None` to return the ratio of variance explained by every component feature, then calculate how many components are required to get above some threshold of variance explained (often 0.95 or 0.99):

```
# Create and run LDA
lda = LinearDiscriminantAnalysis(n_components=None)
features_lda = lda.fit(features, target)

# Create array of explained variance ratios
lda_var_ratios = lda.explained_variance_ratio_

# Create function
def select_n_components(var_ratio, goal_var: float) -> int:
    # Set initial variance explained so far
    total_variance = 0.0

    # Set initial number of features
    n_components = 0

    # For the explained variance of each feature:
    for explained_variance in var_ratio:

        # Add the explained variance to the total
        total_variance += explained_variance
```

```
        # Add one to the number of components
        n_components += 1

        # If we reach our goal level of explained variance
        if total_variance >= goal_var:
            # End the loop
            break

    # Return the number of components
    return n_components

# Run function
select_n_components(lda_var_ratios, 0.95)
```

```
1
```

See Also

- Comparison of LDA and PCA 2D projection of Iris dataset
- Linear Discriminant Analysis

9.4 Reducing Features Using Matrix Factorization

Problem

You have a feature matrix of nonnegative values and want to reduce the dimensionality.

Solution

Use non-negative matrix factorization (NMF) to reduce the dimensionality of the feature matrix:

```
# Load libraries
from sklearn.decomposition import NMF
from sklearn import datasets

# Load the data
digits = datasets.load_digits()

# Load feature matrix
features = digits.data

# Create, fit, and apply NMF
nmf = NMF(n_components=10, random_state=4)
features_nmf = nmf.fit_transform(features)

# Show results
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_nmf.shape[1])
```

```
Original number of features: 64
Reduced number of features: 10
```

Discussion

NMF is an unsupervised technique for linear dimensionality reduction that factorizes (i.e., breaks up into multiple matrices whose product approximates the original matrix) the feature matrix into matrices

representing the latent relationship between observations and their features. Intuitively, NMF can reduce dimensionality because in matrix multiplication, the two factors (matrices being multiplied) can have significantly fewer dimensions than the product matrix. Formally, given a desired number of returned features, *r*, NMF factorizes our feature matrix such that:

$$\mathbf{V} \approx \mathbf{WH}$$

where $\mathbf{V}$ is our $n \times d$ feature matrix (i.e., d features, n observations), $\mathbf{W}$ is a $n \times r$, and $\mathbf{H}$ is an $r \times d$ matrix. By adjusting the value of r we can set the amount of dimensionality reduction desired.

One major requirement of NMA is that, as the name implies, the feature matrix cannot contain negative values. Additionally, unlike PCA and other techniques we have examined, NMA does not provide us with the explained variance of the outputted features. Thus, the best way for us to find the optimum value of `n_components` is by trying a range of values to find the one that produces the best result in our end model (see [Link to Come]).

See Also

- Non-Negative Matrix Factorization (NMF)

9.5 Reducing Features on Sparse Data

Problem

You have a sparse feature matrix and want to reduce the dimensionality.

Solution

Use Truncated Singular Value Decomposition (TSVD):

```
# Load libraries
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import TruncatedSVD
from scipy.sparse import csr_matrix
from sklearn import datasets
import numpy as np

# Load the data
digits = datasets.load_digits()

# Standardize feature matrix
features = StandardScaler().fit_transform(digits.data)

# Make sparse matrix
features_sparse = csr_matrix(features)

# Create a TSVD
tsvd = TruncatedSVD(n_components=10)

# Conduct TSVD on sparse matrix
features_sparse_tsvd = tsvd.fit(features_sparse).transform(features_sparse)

# Show results
print("Original number of features:", features_sparse.shape[1])
print("Reduced number of features:", features_sparse_tsvd.shape[1])
```

```
Original number of features: 64
Reduced number of features: 10
```

Discussion

TSVD is similar to PCA and in fact, PCA actually often uses non-truncated Singular Value Decomposition (SVD) in one of its steps. In regular SVD, given *d* features, SVD will create factor matrices that are *d* × *d*, whereas TSVD will return factors that are *n* × *n*, where *n* is previously specified by a parameter. The practical advantage of TSVD is that unlike PCA, it works on sparse feature matrices.

One issue with TSVD is that because of how it uses a random number generator, the signs of the output can flip between fittings. An easy workaround is to use `fit` only once per preprocessing pipeline, then use `transform` multiple times.

As with linear discriminant analysis, we have to specify the number of features (components) we want outputted. This is done with the `n_components` parameter. A natural question is then: what is the optimum number of components? One strategy is to include `n_components` as a hyperparameter to optimize during model selection (i.e., choose the value for `n_components` that produces the best trained model). Alternatively, because TSVD provides us with the ratio of the original feature matrix's variance explained by each component, we can select the number of components that explain a desired amount of variance (95% or 99% are common values). For example, in our solution the first three outputted components explain approximately 30% of the original data's variance:

```
# Sum of first three components' explained variance ratios
tsvd.explained_variance_ratio_[0:3].sum()
```

```
0.3003938537287226
```

We can automate the process by creating a function that runs TSVD with `n_components` set to one less than the number of original features and then calculate the number of components that explain a desired amount of the original data's variance:

```
# Create and run an TSVD with one less than number of features
tsvd = TruncatedSVD(n_components=features_sparse.shape[1]-1)
features_tsvd = tsvd.fit(features)

# List of explained variances
tsvd_var_ratios = tsvd.explained_variance_ratio_

# Create a function
def select_n_components(var_ratio, goal_var):
    # Set initial variance explained so far
    total_variance = 0.0

    # Set initial number of features
    n_components = 0

    # For the explained variance of each feature:
    for explained_variance in var_ratio:

        # Add the explained variance to the total
        total_variance += explained_variance

        # Add one to the number of components
        n_components += 1
```

```
        # If we reach our goal level of explained variance
        if total_variance >= goal_var:
            # End the loop
            break

    # Return the number of components
    return n_components

# Run function
select_n_components(tsvd_var_ratios, 0.95)
```

```
40
```

See Also

- scikit-learn documentation TruncatedSVD

Chapter 10. Dimensionality Reduction Using Feature Selection

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at feedback.mlpythoncookbook@gmail.com.

10.0 Introduction

In Chapter 9, we discussed how to reduce the dimensionality of our feature matrix by creating new features with (ideally) similar ability to train quality models but with significantly fewer dimensions. This is called feature extraction. In this chapter we will cover an alternative approach: selecting high-quality, informative features and dropping less useful features. This is called feature selection.

There are three types of feature selection methods: filter, wrapper, and embedded. Filter methods select the best features by examining their statistical properties. Methods where we explicitely set a threshold for a statistic or manually select the number of features we want to keep are examples of feature selection by filtering. Wrapper methods use trial and error to find the subset of features that produce models with the highest quality predictions. Wrapper methods are often the most effective method, as they find the best result through actual experimentation as opposed to naive assumptions. Finally, embedded methods select the best feature subset as part or as an extension of a learning algorithm's training process.

Ideally, we'd describe all three methods in this chapter. However, since embedded methods are closely intertwined with specific learning algorithms, they are difficult to explain prior to a deeper dive into the algorithms themselves. Therefore, in this chapter we cover only filter and wrapper feature selection methods, leaving the discussion of particular embedded methods until the chapters where those learning algorithms are discussed in depth.

10.1 Thresholding Numerical Feature Variance

Problem

You have a set of numerical features and want to filter out those with low variance (i.e., likely containing little information).

Solution

Select a subset of features with variances above a given threshold:

```
# Load libraries
from sklearn import datasets
from sklearn.feature_selection import VarianceThreshold

# import some data to play with
iris = datasets.load_iris()

# Create features and target
features = iris.data
target = iris.target

# Create thresholder
thresholder = VarianceThreshold(threshold=.5)

# Create high variance feature matrix
features_high_variance = thresholder.fit_transform(features)

# View high variance feature matrix
features_high_variance[0:3]
```

```
array([[ 5.1,  1.4,  0.2],
       [ 4.9,  1.4,  0.2],
       [ 4.7,  1.3,  0.2]])
```

Discussion

Variance thresholding (VT) is an example of feature selection by filtering, and one of the most basic approaches to feature selection. It is motivated by the idea that features with low variance are likely less interesting (and useful) than features with high variance. VT first calculates the variance of each feature:

$$Var\left(x\right) = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - \mu\right)^2$$

where x is the feature vector, x_i is an individual feature value, and μ is that feature's mean value. Next, it drops all features whose variance does not meet that threshold.

There are two things to keep in mind when employing VT. First, the variance is not centered; that is, it is in the squared unit of the feature itself. Therefore, the VT will not work when feature sets contain different units (e.g., one feature is in years while a different feature is in dollars). Second, the variance threshold is selected manually, so we have to use our own judgment for a good value to select (or use a model selection technique described in [Link to Come]). We can see the variance for each feature using `variances_`:

```
# View variances
thresholder.fit(features).variances_
```

```
array([0.68112222, 0.18871289, 3.09550267, 0.57713289])
```

Finally, if the features have been standardized (to mean zero and unit variance), then for obvious reasons variance thresholding will not work correctly:

```
# Load library
from sklearn.preprocessing import StandardScaler

# Standardize feature matrix
scaler = StandardScaler()
features_std = scaler.fit_transform(features)

# Caculate variance of each feature
selector = VarianceThreshold()
selector.fit(features_std).variances_
```

```
array([1., 1., 1., 1.])
```

10.2 Thresholding Binary Feature Variance

Problem

You have a set of binary categorical features and want to filter out those with low variance (i.e., likely containing little information).

Solution

Select a subset of features with a Bernoulli random variable variance above a given threshold:

```
# Load library
from sklearn.feature_selection import VarianceThreshold

# Create feature matrix with:
# Feature 0: 80% class 0
# Feature 1: 80% class 1
# Feature 2: 60% class 0, 40% class 1
features = [[0, 1, 0],
            [0, 1, 1],
            [0, 1, 0],
            [0, 1, 1],
            [1, 0, 0]]

# Run threshold by variance
thresholder = VarianceThreshold(threshold=(.75 * (1 - .75)))
thresholder.fit_transform(features)
```

```
array([[0],
       [1],
       [0],
       [1],
       [0]])
```

Discussion

Just like with numerical features, one strategy for selecting highly informative categorical features and filtering out less informative ones is to examine their variances. In binary features (i.e., Bernoulli random variables), variance is calculated as:

$$\text{Var}\ (x) = p(1-p)$$

where p is the proportion of observations of class 1. Therefore, by setting p, we can remove features where the vast majority of observations are one class.

10.3 Handling Highly Correlated Features

Problem

You have a feature matrix and suspect some features are highly correlated.

Solution

Use a correlation matrix to check for highly correlated features. If highly correlated features exist, consider dropping one of the correlated features:

```
# Load libraries
import pandas as pd
import numpy as np

# Create feature matrix with two highly correlated features
features = np.array([[1, 1, 1],
                     [2, 2, 0],
                     [3, 3, 1],
                     [4, 4, 0],
                     [5, 5, 1],
                     [6, 6, 0],
                     [7, 7, 1],
                     [8, 7, 0],
                     [9, 7, 1]])

# Convert feature matrix into DataFrame
dataframe = pd.DataFrame(features)

# Create correlation matrix
corr_matrix = dataframe.corr().abs()

# Select upper triangle of correlation matrix
upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape),
                          k=1).astype(bool))

# Find index of feature columns with correlation greater than 0.95
to_drop = [column for column in upper.columns if any(upper[column] > 0.95)]

# Drop features
dataframe.drop(dataframe.columns[to_drop], axis=1).head(3)
```

	0	2
0	1	1
1	2	0
2	3	1

Discussion

One problem we often run into in machine learning is highly correlated features. If two features are highly correlated, then the information they contain is very similar, and it is likely redundant to include both features. In the case of simple models like linear regression, failing to remove such features violates the assumptions of linear regression, and can result in an artificially inflated R-squared value. The solution to highly correlated features is simple: remove one of them from the feature set. Removing

highly correlated features in the manner outlined above is another example of filtering.

In our solution, first we create a correlation matrix of all features:

```
# Correlation matrix
dataframe.corr()
```

	0	1	2
0	1.000000	0.976103	0.000000
1	0.976103	1.000000	-0.034503
2	0.000000	-0.034503	1.000000

Second, we look at the upper triangle of the correlation matrix to identify pairs of highly correlated features:

```
# Upper triangle of correlation matrix
upper
```

	0	1	2
0	NaN	0.976103	0.000000
1	NaN	NaN	0.034503
2	NaN	NaN	NaN

Third, we remove one feature from each of those pairs from the feature set.

10.4 Removing Irrelevant Features for Classification

Problem

You have a categorical target vector and want to remove uninformative features.

Solution

If the features are categorical, calculate a chi-square (χ^2) statistic between each feature and the target vector:

```
# Load libraries
from sklearn.datasets import load_iris
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2, f_classif

# Load data
iris = load_iris()
features = iris.data
target = iris.target

# Convert to categorical data by converting data to integers
features = features.astype(int)

# Select two features with highest chi-squared statistics
```

```
chi2_selector = SelectKBest(chi2, k=2)
features_kbest = chi2_selector.fit_transform(features, target)

# Show results
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_kbest.shape[1])
```

```
Original number of features: 4
Reduced number of features: 2
```

If the features are quantitative, compute the ANOVA F-value between each feature and the target vector:

```
# Select two features with highest F-values
fvalue_selector = SelectKBest(f_classif, k=2)
features_kbest = fvalue_selector.fit_transform(features, target)

# Show results
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_kbest.shape[1])
```

```
Original number of features: 4
Reduced number of features: 2
```

Instead of selecting a specific number of features, we can also use `SelectPercentile` to select the top *n* percent of features:

```
# Load library
from sklearn.feature_selection import SelectPercentile

# Select top 75% of features with highest F-values
fvalue_selector = SelectPercentile(f_classif, percentile=75)
features_kbest = fvalue_selector.fit_transform(features, target)

# Show results
print("Original number of features:", features.shape[1])
print("Reduced number of features:", features_kbest.shape[1])
```

```
Original number of features: 4
Reduced number of features: 3
```

Discussion

Chi-square statistics examines the independence of two categorical vectors. That is, the statistic is the difference between the observed number of observations in each class of a categorical feature and what we would expect if that feature was independent (i.e., no relationship) with the target vector:

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

where O_i is the number of observations in class i and E_i is the number of observations in class i we would theoretically expect if there were no relationship between the feature and target vector.

A chi-squared statistic is a single number that tells you how much difference exists between your observed counts and the counts you would expect if there were no relationship at all in the population. By calculating the chi-squared statistic between a feature and the target vector, we obtain a

measurement of the independence between the two. If the target is independent of the feature variable, then it is irrelevant for our purposes because it contains no information we can use for classification. On the other hand, if the two features are highly dependent, they likely are very informative for training our model.

To use chi-squared in feature selection, we calculate the chi-squared statistic between each feature and the target vector, then select the features with the best chi-square statistics. In scikit-learn, we can use `SelectKBest` to select the features with the best statistics. The parameter `k` determines the number of features we want to keep - and filters out the least informative features.

It is important to note that chi-square statistics can only be calculated between two categorical vectors. For this reason, chi-squared for feature selection requires that both the target vector and the features are categorical. However, if we have a numerical feature we can use the chi-squared technique by first transforming the quantitative feature into a categorical feature. Finally, to use our chi-squared approach, all values need to be non-negative.

Alternatively, if we have a numerical feature we can use `f_classif` to calculate the ANOVA F-value statistic with each feature and the target vector. F-value scores examine if, when we group the numerical feature by the target vector, the means for each group are significantly different. For example, if we had a binary target vector, gender, and a quantitative feature, test scores, the F-value score would tell us if the mean test score for men is different than the mean test score for women. If it is not, then test score doesn't help us predict gender and therefore the feature is irrelevant.

10.5 Recursively Eliminating Features

Problem

You want to automatically select the best features to keep.

Solution

Use scikit-learn's `RFECV` to conduct recursive feature elimination (RFE) using cross-validation (CV). That is, use the wrapper feature selection method and repeatedly train a model, each time removing a feature until model performance (e.g., accuracy) becomes worse. The remaining features are the best:

```
# Load libraries
import warnings
from sklearn.datasets import make_regression
from sklearn.feature_selection import RFECV
from sklearn import datasets, linear_model

# Suppress an annoying but harmless warning
warnings.filterwarnings(action="ignore", module="scipy",
                        message="^internal gelsd")

# Generate features matrix, target vector, and the true coefficients
features, target = make_regression(n_samples = 10000,
                                   n_features = 100,
                                   n_informative = 2,
                                   random_state = 1)

# Create a linear regression
```

```
ols = linear_model.LinearRegression()

# Recursively eliminate features
rfecv = RFECV(estimator=ols, step=1, scoring="neg_mean_squared_error")
rfecv.fit(features, target)
rfecv.transform(features)
```

```
array([[ 0.00850799,  0.7031277 ,  1.52821875],
       [-1.07500204,  2.56148527, -0.44567768],
       [ 1.37940721, -1.77039484, -0.74675125],
       ...,
       [-0.80331656, -1.60648007,  0.52231601],
       [ 0.39508844, -1.34564911,  0.4228057 ],
       [-0.55383035,  0.82880112,  1.73232647]])
```

Once we have conducted RFE, we can see the number of features we should keep:

```
# Number of best features
rfecv.n_features_
```

```
3
```

We can also see which of those features we should keep:

```
# Which categories are best
rfecv.support_
```

```
array([False, False, False, False, False,  True, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False,  True, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False,  True, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False])
```

We can even view the rankings of the features:

```
# Rank features best (1) to worst
rfecv.ranking_
```

```
array([11, 92, 96, 87, 46,  1, 48, 23, 16,  2, 66, 83, 33, 27, 70, 75, 29,
       84, 54, 88, 37, 42, 85, 62, 74, 50, 80, 10, 38, 59, 79, 57, 44,  8,
       82, 45, 89, 69, 94,  1, 35, 47, 39,  1, 34, 72, 19,  4, 17, 91, 90,
       24, 32, 13, 49, 26, 12, 71, 68, 40,  1, 43, 63, 28, 73, 58, 21, 67,
        1, 95, 77, 93, 22, 52, 30, 60, 81, 14, 86, 18, 15, 41,  7, 53, 65,
       51, 64,  6,  9, 20,  5, 55, 56, 25, 36, 61, 78, 31,  3, 76])
```

Discussion

This is likely the most advanced recipe in this book up to this point, combining a number of topics we have yet to address in detail. However, the intuition is straightforward enough that we can address it here rather than holding off until a later chapter. The idea behind RFE is to train a model repeatedly, updating the *weights* or *coefficients* of that model each time. The first time we train the model, we include all the features. Then, we find the feature with the smallest parameter (notice that this assumes

the features are either rescaled or standardized), meaning it is less important, and remove the feature from the feature set.

The obvious question then is: how many features should we keep? We can (hypothetically) repeat this loop until we only have one feature left. A better approach requires that we include a new concept called cross-validation (CV). We will discuss cross-validation in detail in the next chapter, but here is the general idea.

Given data containing 1) a target we want to predict and 2) a feature matrix, first we split the data into two groups: a training set and a test set. Second, we train our model using the training set. Third, we pretend that we do not know the target of the test set, and apply our model to the test set's features in order to predict the values of the test set. Finally, we compare our predicted target values with the true target values to evaluate our model.

We can use CV to find the optimum number of features to keep during RFE. Specifically, in RFE with CV after every iteration, we use cross-validation to evaluate our model. If CV shows that our model improved after we eliminated a feature, then we continue on to the next loop. However, if CV shows that our model got worse after we eliminated a feature, we put that feature back into the feature set and select those features as the best.

In scikit-learn, RFE with CV is implemented using `RFECV` and contains a number of important parameters. The `estimator` parameter determines the type of model we want to train (e.g., linear regression). The `step` parameter sets the number or proportion of features to drop during each loop. The `scoring` parameter sets the metric of quality we use to evaluate our model during cross-validation.

See Also

- Recursive feature elimination with cross-validation

About the Authors

Kyle Gallatin is a software engineer for machine learning infrastructure with years of experience as a data analyst, data scientist and machine learning engineer. He is also a professional data science mentor, volunteer computer science teacher and frequently publishes articles at the intersection of software engineering and machine learning. Currently, Kyle is a software engineer on the machine learning platform team at Etsy.

Chris Albon is the Director of Machine Learning at the Wikimedia Foundation, the nonprofit that hosts Wikipedia.