9df35ce36ec13429fe1548b8faa1bac1
ebrary

# 6
# SciPy for Data Mining

9df35ce36ec13429fe1548b8faa1bac1
ebrary

This section deals with those branches of mathematics that treat the collection, organization, analysis, and interpretation of data. The different applications and operations spread over several modules and submodules – `scipy.stats` (for purely statistical tools), `scipy.ndimage.measurements` (for analysis and organization of data), `scipy.spatial` (for spatial algorithms and data structures), and finally the clustering package `scipy.cluster`, with its two submodules – `scipy.cluster.vq` (vector quantization) and `scipy.cluster.hierarchy` (for hierarchical and agglomerative clustering).

## Descriptive statistics

We often require the analysis of data in which certain features are grouped in different regions, each with different sizes, values, shapes, and so on. The `scipy.ndimage.measurements` submodule has the right tools for this task, and the best way to illustrate the capabilities of the module is by means of an exhaustive examples. For example, for binary images of zeros and ones, it is possible to label each blob (areas of contiguous pixels with value one) and obtain the number of these with the `label` command. If we desire to obtain the center of mass of the blobs, we may do so with the `center_of_mass` command. We may see these operations in action once again in the application to obtaining the structural model of oxides in next chapter.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

For nonbinary data, the `scipy.ndimage.measurements` submodule provides with the usual basic statistical measurements (value and location of extreme values, mean, standard deviation, sum, variance, histogram, and so on).

9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 106
http://site.ebrary.com/id/10672339?ppg=106
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary



For more advanced statistical measurements we must access functions from the `scipy.stats` module. We may now use geometric and harmonic means (`gmean`, `hmean`), median, mode, skewness, various moments, or kurtosis (`median`, `mode`, `skew`, `moment`, `kurtosis`). For an overview of the most significant statistical properties of the dataset, we prefer to use the `describe` routine. We may also compute item frequencies (`itemfreq`), percentiles (`scoreatpercentile`, `percentileofscore`), histograms (`histogram`, `histogram2`), cumulative and relative frequencies (`cumfreq`, `relfreq`), standard error (`sem`), and the signal-to-noise ratio (`signaltonoise`), which is always useful.

## Distributions

9df35ce36ec13429fe1548b8faa1bac1
ebrary

One of the main strengths of the `scipy.stats` module is the great number of distributions coded, both continuous and discrete. The list is impressively large and has 81 continuous distributions and 10 discrete distributions.

One of the most usual ways to employ these distributions is the generation of random numbers. We have been employing this technique to "contaminate" our images with noise, for example:

```
>>> from scipy.stats import norm     # Gaussian distribution
>>>lena=scipy.misc.lena().astype(float)
>>>lena+= norm.rvs(loc=0,scale=16,size=lena.shape)
>>>signaltonoise(lena,axis=None)
array(2.4578546916065163)
```

Let's see the SciPy way of handling distributions. First, a random variable class is created (in SciPy there is the `rv_continuous` class for continuous random variables, and the `rv_discrete` class for the discrete case). Each continuous random variable has associated a probability density function (`pdf`), a cumulative distribution function (`cdf`), a survival function along with its inverse (`sf`, `isf`), and all possible descriptive statistics. They also have associated the random variable per se, `rvs`, which is what we used to actually generate the random instances. For example, with a Pareto continuous random variable with parameter `b = 5`, to check these properties, we could issue the following:

9df35ce36ec13429fe1548b8faa1bac1
ebrary

```
>>> from scipy.stats import pareto
>>> import matplotlib.pyplot as plt
>>> x=numpy.linspace(1,10,1000)
>>>plt.subplot(131); plt.plot(pareto.pdf(x,5))
>>>plt.subplot(132); plt.plot(pareto.cdf(x,5))
>>>plt.subplot(133); plt.plot(pareto.rvs(5,size=1000))
```



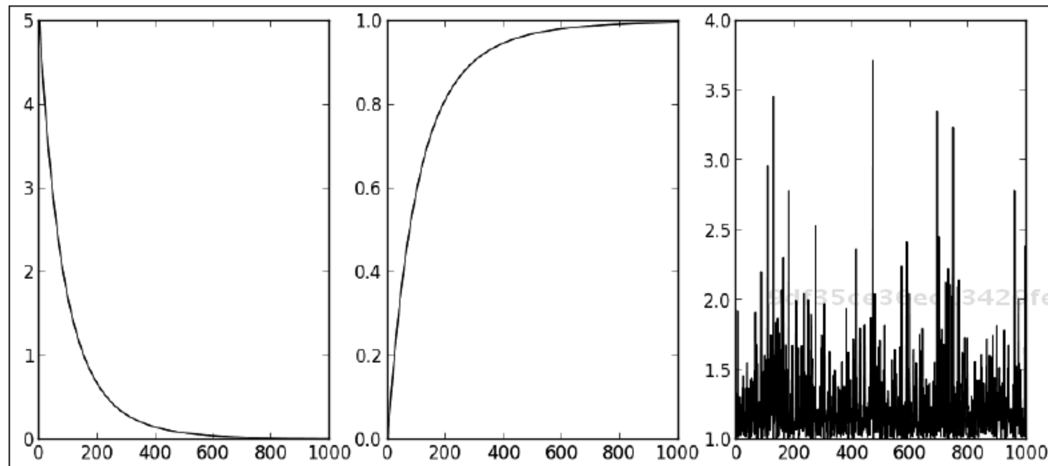
9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 107
http://site.ebrary.com/id/10672339?ppg=107
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary



This gives the following graphs showing probability density function (left), cumulative distribution function (center), and random generation (right):




9df35ce36ec13429fe1548b8faa1bac1
ebrary

## Interval estimation, correlation measures, and statistical tests

We briefly covered interval estimation as an introductory example of SciPy – `bayes_mvs`, in *Chapter 1, Introduction to SciPy*, with very simple syntax, as follows:

```
bayes_mvs(data, alpha=0.9)
```

It offers a tuple of three arguments, in which each argument has the form `(center, (lower, upper))`. The first argument refers to the mean, the second refers to the variance, and the third to the standard deviation. All intervals are computed according to the probability given by `alpha`, which is `0.9` by default.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

We may use the `linregress` routine to compute the regression line of some two-dimensional data `x`, or two sets of one-dimensional data, `x` and `y`. We may compute different correlation coefficients, with their corresponding p-values, as well. We have the Pearson correlation coefficient (`pearsonr`), Spearman's rank-order correlation (`spearmanr`), point biserial correlation (`pointbiserialr`), and Kendall's tau for ordinal data (`kendalltau`). In all cases, the syntax is the same, as it is only required either a two-dimensional array of data, or two one-dimensional arrays of data with the same length.



9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 108
http://site.ebrary.com/id/10672339?ppg=108
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary



SciPy also has most of the best-known statistical tests and procedures – t-tests (`ttest_1samp` for one group of scores, `ttest_ind` for two independent samples of scores, or `ttest_rel` for two related samples of scores), Kolmogorov-Smirnov tests for goodness of fit (`kstest`, `ks_2samp`), one-way Chi-square test (`chisquare`), and many more.

Let us illustrate some of the routines of this module with a textbook example, based on Timothy Sturm's studies on control design.

Twenty-five right-handed individuals were asked to use their right hands to turn a knob that moved an indicator by screw action. There were two identical instruments, one with a right-handed thread where the knob turned clockwise, and the other with a left-hand thread where the knob turned counter-clockwise. The following table gives the times in seconds each subject took to move the indicator to a fixed distance.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

| **Subject** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **Right thread** | 113 | 105 | 130 | 101 | 138 | 118 | 87 | 116 | 75 | 96 |
| **Left thread** | 137 | 105 | 133 | 108 | 115 | 170 | 103 | 145 | 78 | 107 |
| **Subject** | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| **Right thread** | 122 | 103 | 116 | 107 | 118 | 103 | 111 | 104 | 111 | 89 |
| **Left thread** | 84 | 148 | 147 | 87 | 166 | 146 | 123 | 135 | 112 | 93 |
| **Subject** | 21 | 22 | 23 | 24 | 25 | | | | | |
| **Right thread** | 78 | 100 | 89 | 85 | 88 | | | | | |
| **Left thread** | 76 | 116 | 78 | 101 | 123 | | | | | |

We may perform an analysis that leads to a conclusion about right-handed people finding right-hand threads easier to use, by a simple one-sample t-statistic. We will load the data in memory, as follows:

9df35ce36ec13429fe1548b8faa1bac1
ebrary

```
>>> data = numpy.array([[113,105,130,101,138,118,87,116,75,96, \
... 122,103,116,107,118,103,111,104,111,89,78,100,89,85,88], \
... [137,105,133,108,115,170,103,145,78,107, \
... 84,148,147,87,166,146,123,135,112,93,76,116,78,101,123]])
```

The difference of each row indicates which knob was faster, and for how much time. We can obtain that information easily, and perform some basic statistical analysis on it. We will start by computing the mean, standard deviation, and a histogram with 10 bins:

```
>>>dataDiff = data[1,:]-data[0,:]
>>>dataDiff.mean(), dataDiff.std()
(13.720000000000001, 21.62872164507186)
```



9df35ce36ec13429fe1548b8faa1bac1
ebrary

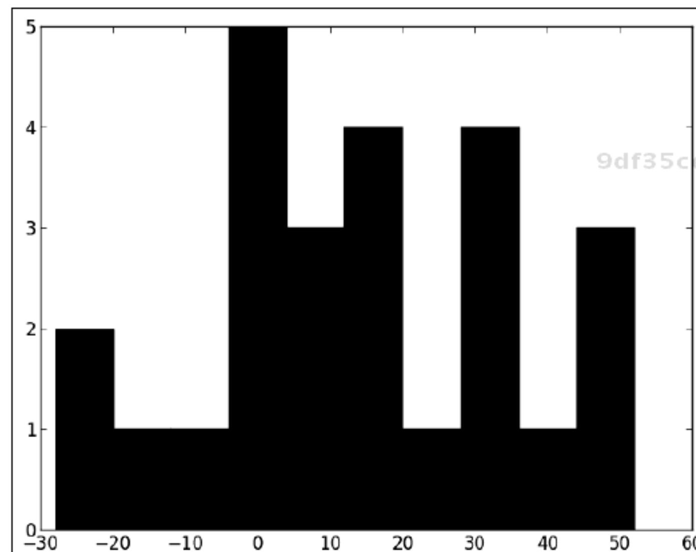
Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 109
http://site.ebrary.com/id/10672339?ppg=109
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary



```
>>>matplotlib.pyplot.hist(dataDiff)
(array([2, 1, 1, 5, 3, 4, 1, 4, 1, 3]),
 array([-28.,-20.,-12.,-4.,4.,12.,20.,28.,36.,44.,52.]),
<a list of 10 Patch objects>)
```

The following histogram is produced:




9df35ce36ec13429fe1548b8faa1bac1
ebrary

Under the light of this histogram, it is not too far fetched to assume a normal distribution. If we assume that this is a proper simple random sample, the use of t-statistics is justified. We would like to prove that it takes longer to turn the left thread than the right, so we set the mean of `dataDiff` to be contrasted against the zero mean (which would indicate that it takes the same time for both threads). The two-sample t-statistics and p-value for the two-sided test are computed by the simple command, as follows:

9df35ce36ec13429fe1548b8faa1bac1
ebrary

```
>>>t_stat,p_value=ttest_1samp(dataDiff)
```

The p-value for the one-sided test is then calculated:

```
>>> print p_value/2.0
0.00239943063239
```

Note that this p-value is much smaller than either of the usual thresholds `alpha = 0.05` or `alpha = 0.1`. We can thus guarantee that we have enough evidence to support the claim that right-handed threads take less time to turn than left-handed threads.



9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 110
http://site.ebrary.com/id/10672339?ppg=110
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary



# Distribution fitting

In Timothy Sturm's example we claim that the histogram of some data seemed to fit a normal distribution. SciPy has a few routines to help us approximate the best distribution to a random variable, together with the parameters that best approximate this fit. For example, for the data in that problem, the mean and standard deviation of the normal distribution that realizes the best fit can be found in the following way:

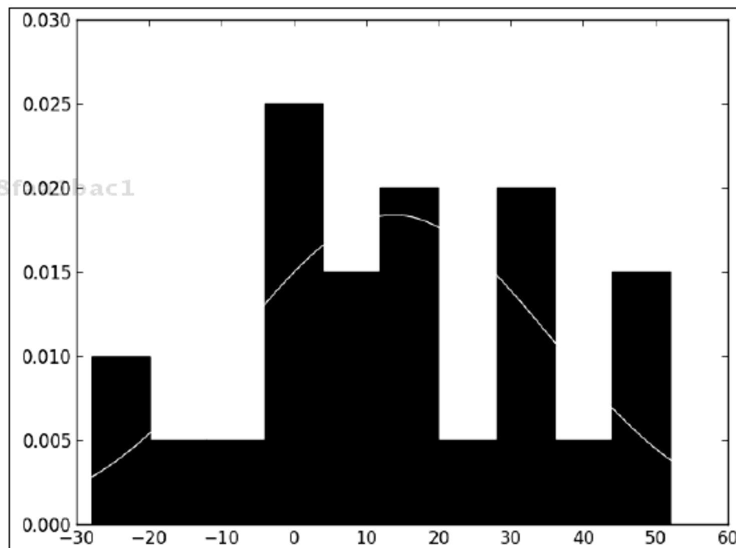
```
>>>mean,std=norm.fit(dataDiff)
```

We can now plot the (normed) histogram of the data, together with the computed probability density function, as follows:

9df35ce36ec13429fe1548b8faa1bac1
ebrary

```
>>>matplotlib.pyplot.hist(dataDiff, normed=1)
(array([ 0.01,0.005,0.005,0.025,0.015,0.02,0.005,0.02,
        0.005,  0.015]),
 array([-28.,-20.,-12.,-4.,4.,12.,20.,28.,36.,44.,52.]),
<a list of 10 Patch objects>)
>>> x=numpy.linspace(dataDiff.min(),dataDiff.max(),1000)
>>>pdf=norm.pdf(x,mean,std)
>>>matplotlib.pyplot.plot(x,pdf)
```

We will obtain the following graph showing the maximum likelihood estimate to the normal distribution that best fits `dataDiff`:




9df35ce36ec13429fe1548b8faa1bac1
ebrary



9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 111
http://site.ebrary.com/id/10672339?ppg=111
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

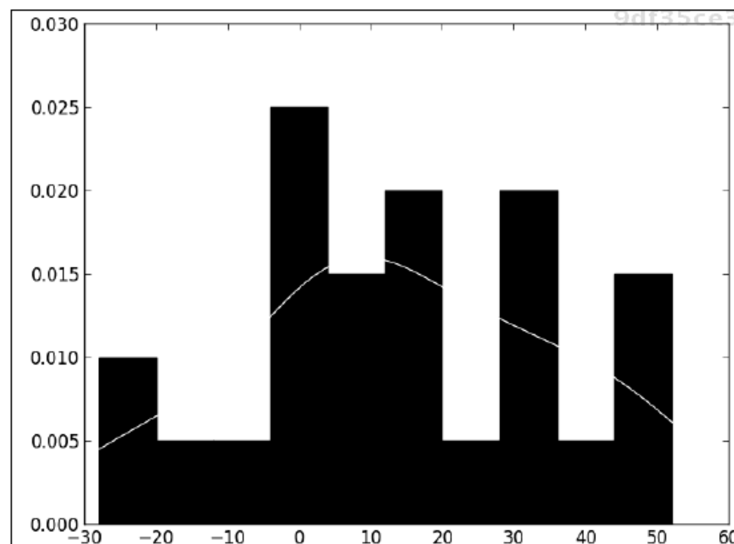
9df35ce36ec13429fe1548b8faa1bac1
ebrary



We may even fit the best probability density function without specifying any particular distribution, thanks to a non-parametric technique, kernel density estimation. We can find an algorithm to perform Gaussian kernel density estimation in the `scipy.stats.kde` submodule. Let us show by example with the same data as before:

```
>>> from scipy.stats.ked import gaussian_kde
>>>pdf=Gaussian_kde(dataDiff)
```

A similar plotting session as before, offers us the following graph, showing probability density function obtained by kernel density estimation on `dataDiff`:




9df35ce36ec13429fe1548b8faa1bac1
ebrary

# Distances

9df35ce36ec13429fe1548b8faa1bac1
ebrary

In the field of data mining, it is often required to determine which members of a training set are closest to unknown test instances. It is imperative to have a good set of different distance functions for any of the algorithms that perform the search, and SciPy has for this purpose a huge collection of optimally coded functions in the `distance` submoduleof the `scipy.spatial` module. The list is long. Besides Euclidean, squared Euclidean, or standardized Euclidean, we have many more – Bray-Curtis, Canberra, Chebyshev, Manhattan, correlation distance, cosine distance, dice dissimilarity, Hamming, Jaccard-Needham, Kulsinski, Mahalanobis, and so on. The syntax in most cases is simple:

```
distance_function(first_vector, second_vector)
```



9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 112
http://site.ebrary.com/id/10672339?ppg=112
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary



The only three cases in which the syntax is different are the Minkowski, Mahalanobis, and standarized Euclidean distances, in which the distance function requires either an integer number (for the order of the norm in the definition of Minkowski distance), a covariance for the Mahalanobis case (but this is an optional requirement), or a variance matrix to standardize the Euclidean distance.

Let us see now a fun exercise to visualize the unit balls in Minkowski metrics:

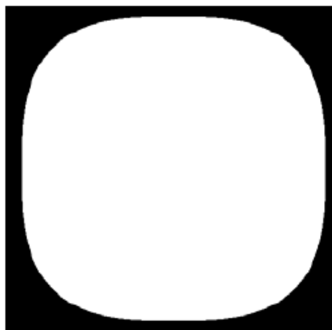
```
Square=numpy.meshgrid[-1.1:1.1:512j,-1.1,1.1:512j]
X=Square[0]; Y=Square[1]
f=lambda x,y,p: minkowski([x,y],[0.0,0.0],p)<=1.0
Ball=lambda p:numpy.vectorize(f)(X,Y,p)
```

9df35ce36ec13429fe1548b8faa1bac1
ebrary

We have created a function `Ball`, which creates a grid of 512 x 512 Boolean values. The grid represents a square of length 2.2 centered at the origin, with sides parallel to the coordinate axis, and the true values on it represent all those points of the grid inside of the unit ball for the Minkowksi metric, for the parameter `p`. All we have to do is show it graphically, like in the following example:

```
>>>matplotlib.pyplot.imshow(Ball(3)); plt.axis('off')
```

This produces the following, where `Ball(3)` is a unit ball in the Minkowski metric with parameter `p = 3`:



9df35ce36ec13429fe1548b8faa1bac1
ebrary

We feel the need to issue the following four important warnings:

- **First warning**: We must use these routines, instead of creating our own definitions of the corresponding distance functions whenever possible. They guarantee a faster result, and optimal coding to take care of situations in which the inputs are either too large or too small.



9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 113
http://site.ebrary.com/id/10672339?ppg=113
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary



- **Second warning**: These functions work great when comparing two vectors; however, for the pairwise computation of many vectors, we must resort to the `pdist` routine. This command takes an $m \times n$ array representing $m$ vectors of dimension $n$, and computes the distance of each of them to each other. We indicate the distance function to be used with the option metric, and additional parameters as needed. For example, for the Manhattan (cityblock) distance for five randomly selected four-dimensional vectors with integer values `1`, `0`, or `-1`, we could issue the following command:

```
>>> V=scipy.stats.randint.rvs(0.4,3,size=(5,4))-1
>>> print V
[[ 1  0  1 -1]
 [-1  0 -1  0]
 [ 1  1  1 -1]
 [ 1  1 -1  0]
 [ 0  0  1 -1]]
>>>pdist(V,metric='cityblock')
array([ 5.,  1.,  4.,  1.,  6.,  3.,  4.,  3.,  2.,  5.])
```

9df35ce36ec13429fe1548b8faa1bac1
ebrary

  This means, if `v1 = [1,0,1,-1]`, `v2 = [-1,0,-1,0]`, `v3 = [1,1,1,-1]`, `v4 = [1,1,-1,0]`, and `v5 = [0,0,1,-1]`, then the Manhattan distance of `v1` from `v2` is 5. The distance from `v1` to `v3` is 1; from `v1` to `v4` is 4; from `v1` to `v5` is 1. From `v2` to `v3` the distance is 6; from `v2` to `v4` is 3; from `v2` to `v5` is 4. From `v3` to `v4` the distance is 3; from `v3` to `v5` is 2. And finally, the distance from `v4` to `v5` is 5, which is the last entry of the output.

- **Third warning**: When computing the distance between each pair of two collections of inputs, we use the `cdist` routine, which has a similar syntax. For instance, for the two collections of three randomly selected four-dimensional Boolean vectors, the corresponding Jaccard-Needham dissimilarities are computed, as follows:

9df35ce36ec13429fe1548b8faa1bac1
ebrary

```
>>> V=scipy.stats.randint.rvs(0.4,2.size=(3,4)).astype(bool)
>>> W=scipy.stats.randint.rvs(0.4,3.size=(3,4)).astype(bool)
>>>cdist(V,W,'jaccard')
array([[ 0.75      ,  1.        ],
       [ 0.75      ,  1.        ],
       [ 0.33333333,  0.5       ]])
```

  That is, if the three vectors in `V` are labeled `v1` through `v3` and if the two vectors in `W` are labeled as `w1` and `w2`, then the dissimilarity between `v1` and `w1` is 0.75; between `v1` and `w2` is 1; and so on.



9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 114
http://site.ebrary.com/id/10672339?ppg=114
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary



- **Fourth warning**: When we have a large amount of data points, and we need to address the problem of nearest neighbors (for example, to locate the closest element of the data to a new instance point), we seldom do it by brute force. The optimal algorithm to perform this search is based in the idea of k-dimensional trees. SciPy has two classes to handle these objects – `KDTree` and `cKDTree`. The latter is a subset of the former, a little faster since it is wrapped from C code, but with very limited use. It only has the `query` method to find the nearest neighbors of the input. The syntax is simple, as follows:

  ```
  KDTree(data, leafsize=10)
  ```

  This creates a structure containing a binary tree, very apt for the design of fast search algorithms. The `leafsize` option indicates at what level the search based on the structure of binary tree must be abandoned in favor of brute force.

  9df35ce36ec13429fe1548b8faa1bac1
  ebrary

  The other methods associated to the `KDTree` class are – `count_neighbors`, to compute the number of nearby pairs that can be formed with another `KDTree`; `query_ball_point`, to find all points at a given distance from the input; `query_ball_tree` and `query_pairs`, to find all pairs of points within certain distance; and `sparse_distance_matrix`, that computes a sparse matrix with the distances between two `KDTree` classes.

  Let us see it in action, with a small dataset of 10 randomly generated four-dimensional points with integer entries:

  ```
  >>> data=scipy.stats.randint.rvs(0.4,10,size=(10,4))
  >>> print data
  [[8 6 1 1]
   [2 9 1 5]
   [4 8 8 9]
   [2 6 6 4]
   [4 1 2 1]
   [3 8 7 2]
   [1 1 3 6]
   [5 2 1 5]
   [2 5 7 3]
   [6 0 6 9]]
  >>> tree=KDTree(data)
  >>>tree.query([0,0,0,0])
  (4.6904157598234297, 4)
  ```

9df35ce36ec13429fe1548b8faa1bac1
ebrary



9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 115
http://site.ebrary.com/id/10672339?ppg=115
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary



This means, among all the points in the dataset, the closest one in the Euclidean distance to the origin is the fifth one (index 4), and the distance is precisely about 4.6 units.

We may input more than one point; the output will still be a tuple, where the first entry is an array that indicates the smallest distance to each of the input points. The second entry is another array that indicates the indices of the nearest neighbors.

# Clustering

Another technique used in data mining is clustering. SciPy has two modules to deal with any problem in this field, each of them addressing a different clustering tool – `scipy.cluster.vq` for k-means and `scipy.cluster.hierarchy` for hierarchical clustering.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

## Vector quantization and k-means

We have two routines to divide data into clusters using the k-means technique – `kmeans` and `kmeans2`. They correspond to two different implementations. The former has a very simple syntax:

```
kmeans(obs, k_or_guess, iter=20, thresh=1e-05)
```

The `obs` parameter is an `ndarray` with the data we wish to cluster. If the dimensions of the array are $m \times n$, the algorithm interprets this data as $m$ points in the $n$-dimensional Euclidean space. If we know the number of clusters in which this data should be divided, we input so with the `k_or_guess` option. The output is a tuple with two elements. The first is an `ndarray` of dimension $k \times n$, representing a collection of points – as many as clusters were indicated. Each of these locations indicates the centroid of the found clusters. The second entry of the tuple is a floating-point value indicating the distortion between the passed points, and the centroids generated previously.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

If we wish to impose an initial guess for the centroids of the clusters, we may do so with the `k_or_guess` parameter again, by sending a $k \times n$ `ndarray`.

The data we pass to `kmeans` need to be normalized with the `whiten` routine.

The second option is much more flexible, as its syntax indicates:

```
kmeans2(data, k, iter=10, thresh=1e-05,
minit='random', missing='warn')
```



9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 116
http://site.ebrary.com/id/10672339?ppg=116
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary



The data and k parameters are the same as obs and k_or_guess, respectively. The difference in this routine is the possibility of choosing among different initialization algorithms, hence providing us with the possibility to speed up process and use fewer resources if we know some properties of our data. We do so by passing to the minit parameter one of the strings such as 'random' (initialization centroids are constructed randomly using a Gaussian), 'points' (initialization is done by choosing points belonging to our data), or 'uniform' (if we prefer uniform distribution to Gaussian).

In case we would like to provide the initialization centroids ourselves with the k parameter, we must indicate our choice to the algorithm by passing 'matrix' to the minit option as well.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

In any case, if we wish to classify the original data by assigning to each point the cluster to which it belongs; we do so with the vq routine (for vector quantization). The syntax is pretty simple as well:

```
vq(obs, centroids)
```

The output is a tuple with two entries. The first entry is a one-dimensional ndarray of size $n$ holding for each point in obs, the cluster to which it belongs. The second entry is another one-dimensional ndarray of same size, but containing floating-point values indicating the distance from each point to the centroid of its cluster.

Let us illustrate with a classical example, the mouse dataset. We will create a big dataset with randomly generated points in three disks, as follows:

```
>>> from scipy.stats import norm
>>> from numpy import array,vstack
>>> data=norm.rvs(0,0.3,size=(10000,2))
>>>inside_ball=numpy.hypot(data[:,0],data[:,1])<1.0
>>> data=data[inside_ball]
>>> data = vstack((data, data+array([1,1]),data+array([-1,1])))
```

9df35ce36ec13429fe1548b8faa1bac1
ebrary

Once created, we request the data to be separated into three clusters:

```
>>> from scipy.cluster.vq import *
>>> centroids, distortion = kmeans(data,3)
>>>cluster_assignment, distances = vq(data,centroids)
```

Let us present the results:

```
>>> from matplotlib.pyplot import plot
>>> plot(data[cluster_assignment==0,0], \
```



9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 117
http://site.ebrary.com/id/10672339?ppg=117
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

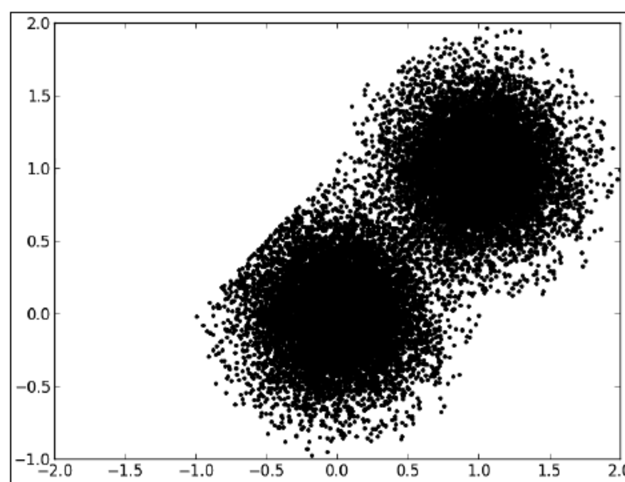
9df35ce36ec13429fe1548b8faa1bac1
ebrary



```
...       data[cluster_assignment==0,1], 'r.')
[<matplotlib.lines.Line2D at 0x10b84ad50>]
>>> plot(data[cluster_assignment==1,0], \
...       data[cluster_assignment==1,1], 'b.')
[<matplotlib.lines.Line2D at 0x10b84af50>]
>>> plot(data[cluster_assignment==2,0], \
...       data[cluster_assignment==2,1], 'k.')
[<matplotlib.lines.Line2D at 0x10b84e8d0>]
```

This gives the following plot showing the mouse dataset with three clusters from left to right – red (0), blue (1), and black (2):

9df35ce36ec13429fe1548b8faa1bac1
ebrary




## Hierarchical clustering

9df35ce36ec13429fe1548b8faa1bac1
ebrary

There are several different algorithms to perform hierarchical clustering. SciPy has routines for the following methods:

- **Single/min/nearest method**: `single`
- **Complete/max/farthest method**: `complete`
- **Average/UPGMA method**: `average`
- **Weighted/WPGMA method**: `weighted`
- **Centroid/UPGMC method**: `centroid`
- **Median/WPGMC method**: `median`
- **Ward's linkage method**: `ward`



9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 118
http://site.ebrary.com/id/10672339?ppg=118
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary



In any of the previous cases, the syntax is the same; the only input is the dataset, which can be either an $m \times n$ `ndarray` representing $m$ points in the $n$-dimensional Euclidean space, or a condensed distance matrix obtained from the previous data using the `pdist` routine from `scipy.spatial`. The output is always an `ndarray` representing the corresponding linkage matrix of the clustering obtained.

Alternatively, we may call the clustering with the generic routine, `linkage`. This routine accepts a dataset/distance matrix, and a string indicating the method to use. The strings coincide with the names introduced before. The advantage of `linkage` over the previous routines is that we are also allowed to indicate a different metric than the usual Euclidean distance. The complete syntax for `linkage` is then as follows:

```
linkage(data, method='single', metric='euclidean')
```

9df35ce36ec13429fe1548b8faa1bac1
ebrary

Different statistics on the resulting linkage matrices may be performed with the routines such as Cophenetic distances between observations (`cophenet`); inconsistency statistics (`inconsistent`); maximum inconsistency coefficient for each non-singleton cluster with its descendants (`maxdists`); and maximum statistic for each non-singleton cluster with its descendants (`maxRstat`).

It is customary to use binary trees to represent linkage matrices, and the `scipy.cluster.hierachy` submodule has a large number of different routines to manipulate and extract information from these trees. The most useful of these routines is the visualization of these trees, often called dendrograms. The corresponding routine in SciPy is dendrogram, and has the following imposing syntax:

```
dendrogram(Z, p=30, truncate_mode=None, color_threshold=None,
get_leaves=True, orientation='top', labels=None,
count_sort=False, distance_sort=False,
show_leaf_counts=True, no_plot=False, no_labels=False,
color_list=None, leaf_font_size=None,
leaf_rotation=None, leaf_label_func=None,
no_leaves=False, show_contracted=False,
link_color_func=None)
```

9df35ce36ec13429fe1548b8faa1bac1
ebrary

The first obvious parameter, `Z`, is a linkage matrix. This is the only non-optional variable. The other options control the style of the output (colors, labels, rotation, and so on), and since they are technically nonmathematical in nature, we will not explore them in detail in this monograph, other than through the simple application to animal clustering shown next.



9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 119
http://site.ebrary.com/id/10672339?ppg=119
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary



Clustering mammals by their dentition – Mammal's teeth are divided into four groups such as incisors, canines, premolars, and molars. The dentition of several mammals has been collected, and is available for download at `www.uni-koeln.de/themen/statistik/data/cluster/dentitio.dat`.

This file presents the name of the mammal, together with the number of top incisors, bottom incisors, top canines, bottom canines, top premolars, bottom premolars, top molars, and bottom molars.

We wish to use hierarchical clustering on that dataset to assess which species are closer to each other by these features.

We start by preparing the dataset and store the relevant data in ndarrays. The original data is given as a text file, where each line represents a different mammal. The first four lines are as follows:

9df35ce36ec13429fe1548b8faa1bac1
ebrary

```
OPOSSUM                     54113344
HAIRY TAIL MOLE             33114433
COMMON MOLE               32103333
STAR NOSE MOLE             33114433
```

The first twenty-seven characters of each line hold the name on the animal. The characters in positions twenty-eight to thirty-five are the number of respective kind of denture. We need to prepare this data into something that SciPy can handle. We collect the names apart, since we will be using them as labels in the dendrogram. The rest of the data will be forced into an array of integers:

```
file=open("dentitio.dat","r")    # open the file
lines=file.readlines()      # read each line in memory
file.close()        # close the file
mammals=[]                 # this stores the names
dataset=numpy.zeros((len(list),8))  # this stores the data
for index,line in enumerate(lines):
mammals.append( line[0:27].rstrip(" ").capitalize() )
    for tooth in range(8):
        dataset[index,tooth]=int(line[27+tooth])
```

9df35ce36ec13429fe1548b8faa1bac1
ebrary

We proceed to compute the linkage matrix and its posterior dendrogram, making sure to use the Python list mammals as labels:

```
>>> from scipy.cluster.hierachy import linkage, dendrogram
>>> Z=linkage(dataset)
>>>dendrogram(Z, labels=mammals, orientation="right")
>>>matplotlib.pyplot.show()
```



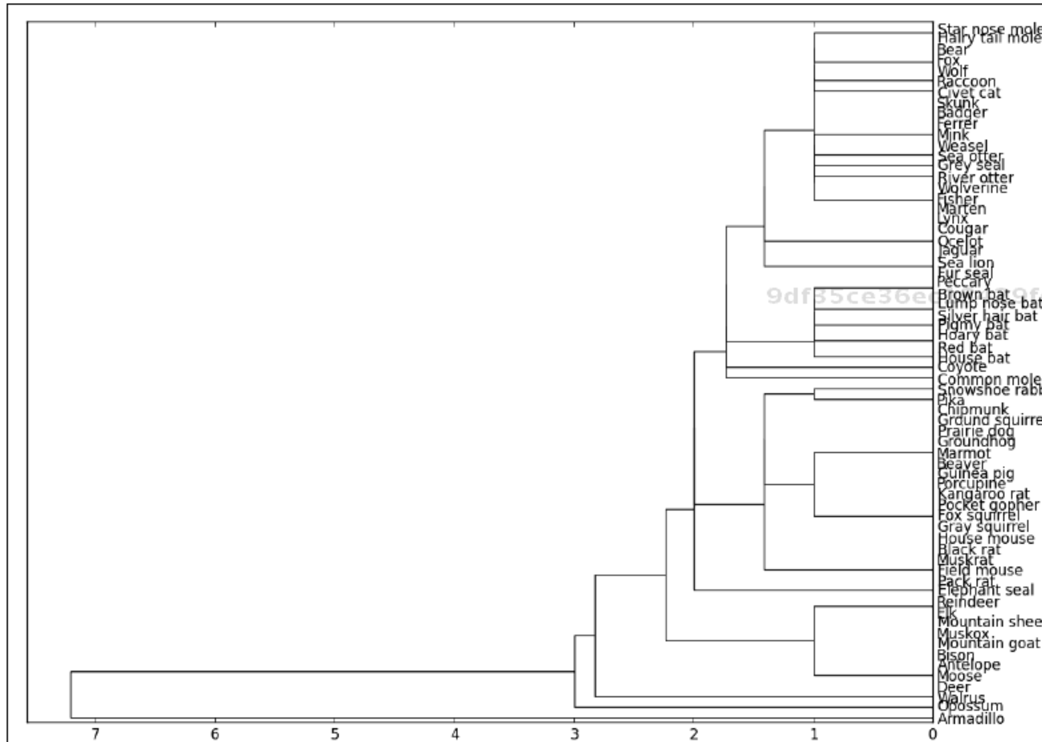
9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 120
http://site.ebrary.com/id/10672339?ppg=120
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

9df35ce36ec13429fe1548b8faa1bac1
ebrary



This gives us the following dendrogram showing clustering of mammals according to their dentition:




9df35ce36ec13429fe1548b8faa1bac1
ebrary

Note how all the bats are clustered together. The mice are also clustered together, but far from the bats. Sheep, goats, antelopes, deer, and moose have similar dentures too, and they appear clustered at the bottom of the tree, next to the opossum and the armadillo. Note how all felines are also clustered together, on the top of the tree.

9df35ce36ec13429fe1548b8faa1bac1
ebrary

Experts in data analysis can obtain more information from dendrograms; they are able to interpret the lengths of the branches or the different colors used in the composition, and give us more insightful explanations about the way the clusters differ from each other.

# Summary

This chapter dealt with tools appropriate for data mining, and explored the modules such as `stats` (for statistics), `spatial` (for data structures), and `cluster` (for clustering and vector quantization).



9df35ce36ec13429fe1548b8faa1bac1
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 121
http://site.ebrary.com/id/10672339?ppg=121
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.