Finding the closest objects in the feature space

Sometimes, the easiest thing to do is to just find the distance between two objects. We just need to find some distance metric, compute the pairwise distances, and compare the outcomes to what's expected.

Getting ready

- A lower-level utility in scikit-learn is `sklearn.metrics.pairwise`. This contains server functions to compute the distances between the vectors in a matrix X or the distances between the vectors in X and Y easily.
- This can be useful for information retrieval. For example, given a set of customers with attributes of X, we might want to take a reference customer and find the closest customers to this customer.
- In this scenario, we might want to rank customers by the notion of similarity measured by a distance function. The quality of the similarity depends upon the feature space selection as well as any transformation we might do on the space.

We'll walk through several different scenarios of measuring distance.

Implementation

We will use the `pairwise_distances` function to determine the "closeness" of objects. Remember that the closeness is really just similarity that we use our distance function to assess.

First, let's import the pairwise distance function from the **metrics** module and create a dataset to play with:

```
>>> from sklearn.metrics import pairwise
>>> from sklearn.datasets import make_blobs
>>> points, labels = make_blobs()
```

This simplest way to check the distances is `pairwise_distances`:

```
>>> distances = pairwise.pairwise_distances(points)
```

`distances` is an N x N matrix with 0s along the diagonals. In the simplest case, let's see the distances between each point and the first point:

```
>>> np.diag(distances) [:5]

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

	g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8	g_9	g_{10}
g_1	0.0	8.1	9.2	7.7	9.3	2.3	5.1	10.2	6.1	7.0
g_2	8.1	0.0	12.0	0.9	12.0	9.5	10.1	12.8	2.0	1.0
g_3	9.2	12.0	0.0	11.2	0.7	11.1	8.1	1.1	10.5	11.5
g_4	7.7	0.9	11.2	0.0	11.2	9.2	9.5	12.0	1.6	1.1
g_5	9.3	12.0	0.7	11.2	0.0	11.2	8.5	1.0	10.6	11.6
g_6	2.3	9.5	11.1	9.2	11.2	0.0	5.6	12.1	7.7	8.5
g_7	5.1	10.1	8.1	9.5	8.5	5.6	0.0	9.1	8.3	9.3
g_8	10.2	12.8	1.1	12.0	1.0	12.1	9.1	0.0	11.4	12.4
g_9	6.1	2.0	10.5	1.6	10.6	7.7	8.3	11.4	0.0	1.1
g_{10}	7.0	1.0	11.5	1.1	11.6	8.5	9.3	12.4	1.1	0.0

Now we can look for points that are closest to the first point in points:

```
>>> distances[0][:5]

array([ 0., 11.82643041,1.23751545, 1.17612135, 14.61927874])
```

Ranking the points by closeness is very easy with `np.argsort`:

```
>>> ranks = np.argsort(distances[0])
>>> ranks[:5]

array([ 0, 27, 98, 23, 67])
```

A useful characteristic of `argsort` is that now we can sort our points matrix to get the actual points:

```
>>> points[ranks][:5]
    array([[ 8.96147382, -1.90405304],
           [ 8.75417014, -1.76289919],
           [ 8.78902665, -2.27859923],
           [ 8.59694131, -2.10057667],
           [ 8.70949958, -2.30040991]])
```

It's useful to see what the closest points look like.

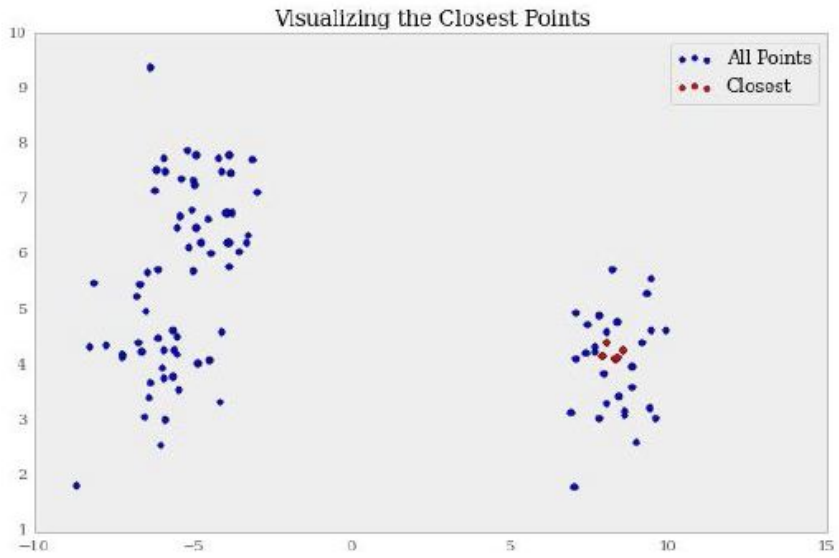

Figure 1

Theory : Euclidean Distance

Given some distance function, each point is measured in a pairwise function. The default is the Euclidian distance, which is as follows:

if $p = (p1, p2, ..., pn)$ and $q = (q1, q2, ..., qn)$ are two points in Euclidean $n-$space, then the distance (d) from p to q, or from q to p is given by the Pythagorean formula:

$$\mathrm{d}(\mathbf{p}, \mathbf{q}) = \mathrm{d}(\mathbf{q}, \mathbf{p})$$
$$= \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$
$$= \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}.$$

Verbally, this takes the difference between each component of the two vectors, squares the difference, sums them, and then takes the square root. This looks very familiar as we used something very similar to this when looking at the mean-squared error. If we take the square root, we have the same thing. In fact, a metric used often

is root-mean-square deviation (RMSE), which is just the applied distance function.

In Python, this looks like the following:

```
>>> def euclid_distances(x, y):

return np.power(np.power(x - y, 2).sum(), .5)
>>> euclid_distances(points[0], points[1])

11.826430406213145
```

Other Distance Measures

There are several other functions available in scikit-learn, but scikit-learn will also use distance functions of SciPy.

1. `cityblock`
2. `cosine`
3. `euclidean`
4. `l1`
5. `l2`
6. `manhattan`

Worked Example

We can now solve problems. For example, if we were standing on a grid at the origin, and the lines were the streets, how far will we have to travel to get to point (5, 5)?.

```
>>> pairwise.pairwise_distances([[0, 2], [6, 6]],
  metric='cityblock')[0]

array([ 0., 10.])
```