Sorting Arrays

Up to this point we have been concerned mainly with **tools to access and operate** on array data with NumPy.

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

These algorithms are a **favorite topic in introductory computer science courses:**

Algorithms such as **insertion sorts**, **selection sorts**, **merge sorts**, **quick sorts**, **bubble sorts**, and many, many more.

All are means of accomplishing a similar task: **sorting the values in a list or array.**

Python has a couple of **built-in functions** and methods for **sorting lists and other iterable objects.**

The `sorted` function accepts a **list** and returns a sorted version of it:

In [100…
```
L = [3, 1, 4, 1, 5, 9, 2, 6]
sorted(L)  # returns a sorted copy
```

Out[100…
```
[1, 1, 2, 3, 4, 5, 6, 9]
```

By contrast, the `sort` **method of lists** will sort the list **in-place:**

In [102…
```
L.sort()  # acts in-place and returns None
print(L)
```

```
[1, 1, 2, 3, 4, 5, 6, 9]
```

Python's **sorting methods** are quite **flexible,** and can handle **any iterable** object. **For example,** here we sort a **string:**

In [104…
```
sorted('python')
```

Out[104…
```
['h', 'n', 'o', 'p', 't', 'y']
```

These **built-in sorting methods are convenient.**

But as previously discussed, the **dynamism of Python** values means they are **less performant than routines designed specifically for uniform arrays of numbers.**

This is where **NumPy's sorting routines** come in.

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

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

In [107…
```
import numpy as np

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

Out[107…
```
array([1, 2, 3, 4, 5])
```

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

In [109…
```
x.sort()
print(x)
```

```
[1 2 3 4 5]
```

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

In [111…

```
x = np.array([2, 1, 4, 3, 5])
i = np.argsort(x)
print(i)
```

```
[1 0 3 2 4]
```

The **first element of this result gives the index of the smallest element,** the second value gives the index of the second smallest, **and so on.**

These **indices can then be used** (via fancy indexing) to **construct the sorted array** if desired:

In [113…

```
x[i]
```

Out[113...

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

You'll see an application of `argsort` later in this chapter.

Sorting Along Rows or Columns

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

In [116...

```
rng = np.random.default_rng(seed=42)
X = rng.integers(0, 10, (4, 6))
print(X)
```

```
[[0 7 6 4 4 8]
 [0 6 2 0 5 9]
 [7 7 7 7 5 1]
 [8 4 5 3 1 9]]
```

In [117…

```python
# sort each column of X
np.sort(X, axis=0)
```

Out[117…

```
array([[0, 4, 2, 0, 1, 1],
       [0, 6, 5, 3, 4, 8],
       [7, 7, 6, 4, 5, 9],
       [8, 7, 7, 7, 5, 9]], dtype=int64)
```

In [118…

```python
# sort each row of X
np.sort(X, axis=1)
```

Out[118…

```
array([[0, 4, 4, 6, 7, 8],
       [0, 0, 2, 5, 6, 9],
       [1, 5, 7, 7, 7, 7],
       [1, 3, 4, 5, 8, 9]], dtype=int64)
```

Keep in mind that this **treats each row or column as an independent array.**

And any **relationships between the row or column values will be lost!**

Partial Sorts: Partitioning

Sometimes we're not interested in sorting the entire array, but simply want to find the **k** smallest values in the array.

NumPy enables this with the `np.partition` function.

`np.partition` takes an array and a number ***K.***

The result is a new array with the **smallest *K* values to the left of the partition** and **the remaining values to the right:**

In [121…

```
x = np.array([7, 2, 3, 1, 6, 5, 4])
np.partition(x, 3)
```

```
Out[121…  array([2, 1, 3, 4, 6, 5, 7])
```

Notice that the **first three values** in the resulting array are **the three smallest in the array,** and the remaining array positions contain the remaining values.

Within the two partitions, the elements have **arbitrary order.**

Similarly to sorting, we can **partition along an arbitrary axis** of a multidimensional array:

```
In [123…  X
```

```
Out[123…  array([[0, 7, 6, 4, 4, 8],
                 [0, 6, 2, 0, 5, 9],
                 [7, 7, 7, 7, 5, 1],
                 [8, 4, 5, 3, 1, 9]], dtype=int64)
```

```
In [124…  np.partition(X, 2, axis=1)
```

Out[124…

```
array([[0, 4, 4, 7, 6, 8],
       [0, 0, 2, 6, 5, 9],
       [1, 5, 7, 7, 7, 7],
       [1, 3, 4, 5, 8, 9]], dtype=int64)
```

The result is an array where the **first two slots in each row** contain the **smallest values from that row,** with the remaining values filling the remaining slots.

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

We'll see both of these in action in the following section.

Example: k-Nearest Neighbors

Let's quickly see how we might use the `argsort` function **along multiple axes** to **find the nearest neighbors of each point in a set.**

We'll start by **creating a random set of 10 points** on a **two-dimensional plane.**

Using the standard convention, we'll arrange these in a **10 × 2 array:**

In [127…

```
X = rng.random((10, 2))
```

In [128…

```
X
```

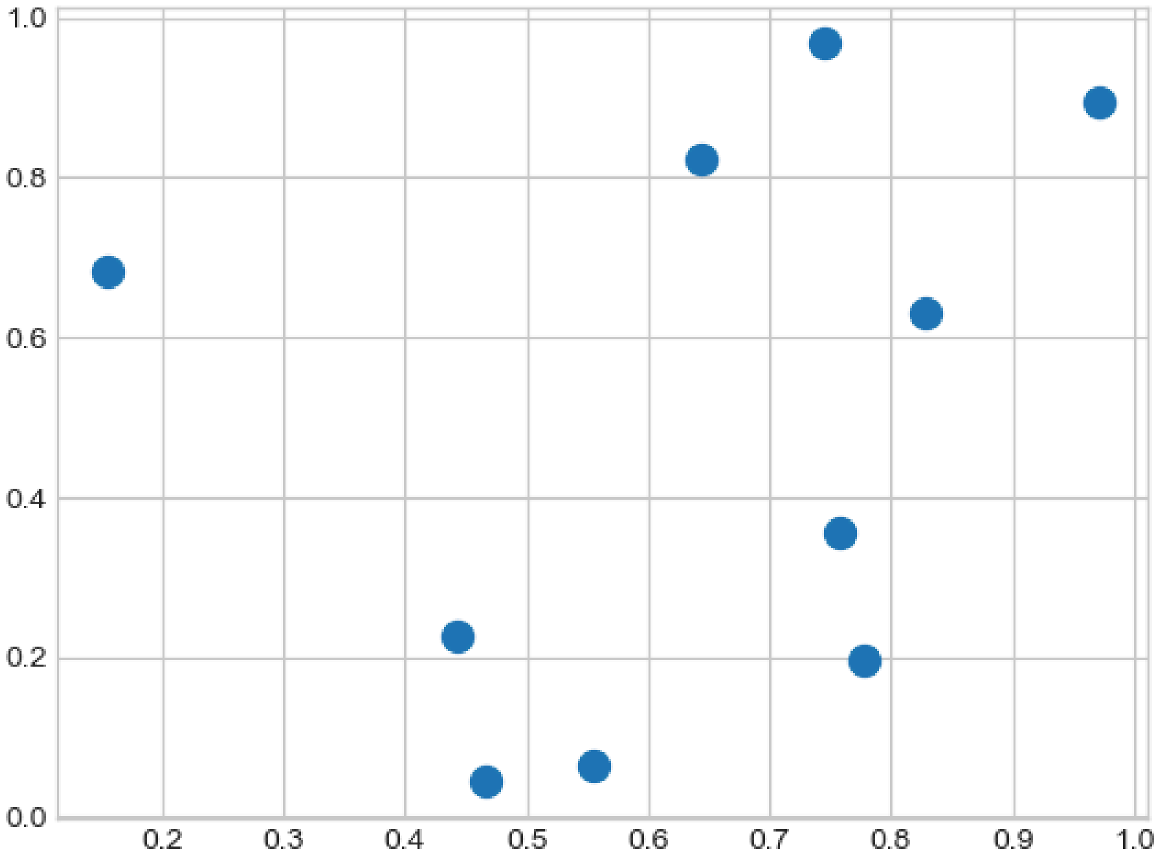
Out[128…

```
array([[0.64386512, 0.82276161],
       [0.4434142 , 0.22723872],
       [0.55458479, 0.06381726],
       [0.82763117, 0.6316644 ],
       [0.75808774, 0.35452597],
       [0.97069802, 0.89312112],
       [0.7783835 , 0.19463871],
       [0.466721  , 0.04380377],
       [0.15428949, 0.68304895],
       [0.74476216, 0.96750973]])
```

To **get an idea of how these points look,** let's generate a quick **scatter plot:**

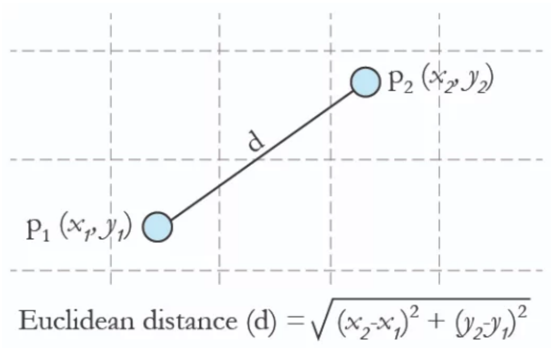
In [130…

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use("seaborn-v0_8-whitegrid")
plt.scatter(X[:, 0], X[:, 1], s=100);
```


1.0
0.8
0.6
0.4
0.2
0.0
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

Now we'll compute the **distance between each pair of points.**

Recall: **squared distance between two points is the sum of the squared differences in each dimension:**



Using the efficient broadcasting and aggregation routines provided by NumPy we can compute the **matrix of square distances** in a single line of code:

In [132…

```
dist_sq = np.sum((X[:, np.newaxis] - X[np.newaxis, :]) ** 2, a
```

In [133…

```
dist_sq.shape
```

Out[133…

```
(10, 10)
```

In [134…

```
dist_sq
```

```
Out[134…  array([[0.        , 0.39482809, 0.58396752, 0.07028811, 0.232
29143,
         0.11177021, 0.41263358, 0.63815537, 0.25920392, 0.031
13223],
        [0.39482809, 0.        , 0.03906548, 0.31118281, 0.115
22148,
         0.7214276 , 0.11326719, 0.03419159, 0.29135606, 0.638
81176],
        [0.58396752, 0.03906548, 0.        , 0.39700471, 0.125
92501,
         0.86089513, 0.06720011, 0.00812058, 0.54368422, 0.852
82752],
        [0.07028811, 0.31118281, 0.39700471, 0.        , 0.081
642  ,
         0.08882774, 0.19341679, 0.47583627, 0.45602939, 0.119
65936],
        [0.23229143, 0.11522148, 0.12592501, 0.081642  , 0.
,
         0.33528787, 0.02597585, 0.18144286, 0.47249968, 0.375
92667],
        [0.11177021, 0.7214276 , 0.86089513, 0.08882774, 0.335
```

```
28787,
        0.        , 0.52486256, 0.97533281, 0.71065321, 0.056
58068],
       [0.41263358, 0.11326719, 0.06720011, 0.19341679, 0.025
97585,
        0.52486256, 0.        , 0.11988469, 0.62803789, 0.598
46002],
       [0.63815537, 0.03419159, 0.00812058, 0.47583627, 0.181
44286,
        0.97533281, 0.11988469, 0.        , 0.50624786, 0.930
5396 ],
       [0.25920392, 0.29135606, 0.54368422, 0.45602939, 0.472
49968,
        0.71065321, 0.62803789, 0.50624786, 0.        , 0.429
5759 ],
       [0.03113223, 0.63881176, 0.85282752, 0.11965936, 0.375
92667,
        0.05658068, 0.59846002, 0.9305396 , 0.4295759 , 0.
]])
```

This **operation has a lot packed into it,** and it might be a bit confusing if you're unfamiliar with NumPy's broadcasting rules.

When you come across code like this, it can be useful to **break it down into its component steps:**

In [136...
```
# for each pair of points, compute differences in their coordi
differences = X[:, np.newaxis] - X[np.newaxis, :]
differences.shape
```

Out[136...
```
(10, 10, 2)
```

In [137...
```
differences
```

```
Out[137…  array([[[ 0.        ,  0.        ],
                  [ 0.20045092,  0.59552289],
                  [ 0.08928033,  0.75894436],
                  [-0.18376605,  0.19109721],
                  [-0.11422262,  0.46823565],
                  [-0.3268329 , -0.07035951],
                  [-0.13451838,  0.62812291],
                  [ 0.17714412,  0.77895785],
                  [ 0.48957563,  0.13971266],
                  [-0.10089704, -0.14474812]],

                 [[-0.20045092, -0.59552289],
                  [ 0.        ,  0.        ],
                  [-0.11117059,  0.16342147],
                  [-0.38421697, -0.40442568],
                  [-0.31467354, -0.12728725],
                  [-0.52728383, -0.6658824 ],
                  [-0.3349693 ,  0.03260001],
                  [-0.0233068 ,  0.18343496],
                  [ 0.28912471, -0.45581023],
                  [-0.30134796, -0.74027101]],
```

```
[[-0.08928033, -0.75894436],
 [ 0.11117059, -0.16342147],
 [ 0.        ,  0.        ],
 [-0.27304638, -0.56784714],
 [-0.20350295, -0.29070871],
 [-0.41611324, -0.82930387],
 [-0.22379871, -0.13082145],
 [ 0.08786378,  0.02001349],
 [ 0.40029529, -0.6192317 ],
 [-0.19017737, -0.90369248]],

[[ 0.18376605, -0.19109721],
 [ 0.38421697,  0.40442568],
 [ 0.27304638,  0.56784714],
 [ 0.        ,  0.        ],
 [ 0.06954343,  0.27713843],
 [-0.14306685, -0.26145672],
 [ 0.04924767,  0.43702569],
 [ 0.36091017,  0.58786063],
 [ 0.67334168, -0.05138455],
```

```
 [ 0.08286902, -0.33584533]],

[[ 0.11422262, -0.46823565],
 [ 0.31467354,  0.12728725],
 [ 0.20350295,  0.29070871],
 [-0.06954343, -0.27713843],
 [ 0.        ,  0.        ],
 [-0.21261028, -0.53859515],
 [-0.02029576,  0.15988726],
 [ 0.29136674,  0.3107222 ],
 [ 0.60379825, -0.32852299],
 [ 0.01332558, -0.61298376]],

[[ 0.3268329 ,  0.07035951],
 [ 0.52728383,  0.6658824 ],
 [ 0.41611324,  0.82930387],
 [ 0.14306685,  0.26145672],
 [ 0.21261028,  0.53859515],
 [ 0.        ,  0.        ],
 [ 0.19231453,  0.69848241],
 [ 0.50397702,  0.84931736],
```

```
 [ 0.81640853,  0.21007217],
 [ 0.22593587, -0.07438861]],

[[ 0.13451838, -0.62812291],
 [ 0.3349693 , -0.03260001],
 [ 0.22379871,  0.13082145],
 [-0.04924767, -0.43702569],
 [ 0.02029576, -0.15988726],
 [-0.19231453, -0.69848241],
 [ 0.        ,  0.        ],
 [ 0.31166249,  0.15083494],
 [ 0.62409401, -0.48841025],
 [ 0.03362134, -0.77287102]],

[[-0.17714412, -0.77895785],
 [ 0.0233068 , -0.18343496],
 [-0.08786378, -0.02001349],
 [-0.36091017, -0.58786063],
 [-0.29136674, -0.3107222 ],
 [-0.50397702, -0.84931736],
 [-0.31166249, -0.15083494],
```

```
 [ 0.        ,  0.        ],
 [ 0.31243151, -0.63924519],
 [-0.27804115, -0.92370597]],

[[-0.48957563, -0.13971266],
 [-0.28912471,  0.45581023],
 [-0.40029529,  0.6192317 ],
 [-0.67334168,  0.05138455],
 [-0.60379825,  0.32852299],
 [-0.81640853, -0.21007217],
 [-0.62409401,  0.48841025],
 [-0.31243151,  0.63924519],
 [ 0.        ,  0.        ],
 [-0.59047266, -0.28446078]],

[[ 0.10089704,  0.14474812],
 [ 0.30134796,  0.74027101],
 [ 0.19017737,  0.90369248],
 [-0.08286902,  0.33584533],
 [-0.01332558,  0.61298376],
 [-0.22593587,  0.07438861],
```

```
        [-0.03362134,  0.77287102],
        [ 0.27804115,  0.92370597],
        [ 0.59047266,  0.28446078],
        [ 0.        ,  0.        ]]])
```

In [138…

```
# square the coordinate differences
sq_differences = differences ** 2
sq_differences.shape
```

Out[138…

```
(10, 10, 2)
```

In [139…

```
sq_differences
```

```
Out[139…  array([[[0.00000000e+00, 0.00000000e+00],
                  [4.01805718e-02, 3.54647514e-01],
                  [7.97097787e-03, 5.75996537e-01],
                  [3.37699618e-02, 3.65181453e-02],
                  [1.30468069e-02, 2.19244619e-01],
                  [1.06819747e-01, 4.95046037e-03],
                  [1.80951937e-02, 3.94538384e-01],
                  [3.13800380e-02, 6.06775328e-01],
                  [2.39684296e-01, 1.95196274e-02],
                  [1.01802118e-02, 2.09520180e-02]],

                 [[4.01805718e-02, 3.54647514e-01],
                  [0.00000000e+00, 0.00000000e+00],
                  [1.23588997e-02, 2.67065754e-02],
                  [1.47622682e-01, 1.63560128e-01],
                  [9.90194376e-02, 1.62020431e-02],
                  [2.78028233e-01, 4.43399370e-01],
                  [1.12204431e-01, 1.06276091e-03],
                  [5.43207155e-04, 3.36483831e-02],
                  [8.35930961e-02, 2.07762967e-01],
                  [9.08105912e-02, 5.48001169e-01]],
```

```
[[7.97097787e-03, 5.75996537e-01],
 [1.23588997e-02, 2.67065754e-02],
 [0.00000000e+00, 0.00000000e+00],
 [7.45543283e-02, 3.22450378e-01],
 [4.14134519e-02, 8.45115552e-02],
 [1.73150226e-01, 6.87744901e-01],
 [5.00858626e-02, 1.71142522e-02],
 [7.72004441e-03, 4.00539795e-04],
 [1.60236323e-01, 3.83447895e-01],
 [3.61674316e-02, 8.16660092e-01]],

[[3.37699618e-02, 3.65181453e-02],
 [1.47622682e-01, 1.63560128e-01],
 [7.45543283e-02, 3.22450378e-01],
 [0.00000000e+00, 0.00000000e+00],
 [4.83628892e-03, 7.68057099e-02],
 [2.04681243e-02, 6.83596176e-02],
 [2.42533348e-03, 1.90991455e-01],
 [1.30256150e-01, 3.45580124e-01],
 [4.53389018e-01, 2.64037240e-03],
```

```
 [6.86727383e-03, 1.12792088e-01]],

[[1.30468069e-02, 2.19244619e-01],
 [9.90194376e-02, 1.62020431e-02],
 [4.14134519e-02, 8.45115552e-02],
 [4.83628892e-03, 7.68057099e-02],
 [0.00000000e+00, 0.00000000e+00],
 [4.52031330e-02, 2.90084739e-01],
 [4.11917752e-04, 2.55639360e-02],
 [8.48945751e-02, 9.65482870e-02],
 [3.64572324e-01, 1.07927352e-01],
 [1.77571194e-04, 3.75749095e-01]],

[[1.06819747e-01, 4.95046037e-03],
 [2.78028233e-01, 4.43399370e-01],
 [1.73150226e-01, 6.87744901e-01],
 [2.04681243e-02, 6.83596176e-02],
 [4.52031330e-02, 2.90084739e-01],
 [0.00000000e+00, 0.00000000e+00],
 [3.69848774e-02, 4.87877682e-01],
 [2.53992837e-01, 7.21339970e-01],
```

```
 [6.66522892e-01, 4.41303158e-02],
 [5.10470167e-02, 5.53366546e-03]],

[[1.80951937e-02, 3.94538384e-01],
 [1.12204431e-01, 1.06276091e-03],
 [5.00858626e-02, 1.71142522e-02],
 [2.42533348e-03, 1.90991455e-01],
 [4.11917752e-04, 2.55639360e-02],
 [3.69848774e-02, 4.87877682e-01],
 [0.00000000e+00, 0.00000000e+00],
 [9.71335098e-02, 2.27511797e-02],
 [3.89493327e-01, 2.38544568e-01],
 [1.13039458e-03, 5.97329621e-01]],

[[3.13800380e-02, 6.06775328e-01],
 [5.43207155e-04, 3.36483831e-02],
 [7.72004441e-03, 4.00539795e-04],
 [1.30256150e-01, 3.45580124e-01],
 [8.48945751e-02, 9.65482870e-02],
 [2.53992837e-01, 7.21339970e-01],
 [9.71335098e-02, 2.27511797e-02],
```

```
 [0.00000000e+00, 0.00000000e+00],
 [9.76134495e-02, 4.08634410e-01],
 [7.73068823e-02, 8.53232713e-01]],

[[2.39684296e-01, 1.95196274e-02],
 [8.35930961e-02, 2.07762967e-01],
 [1.60236323e-01, 3.83447895e-01],
 [4.53389018e-01, 2.64037240e-03],
 [3.64572324e-01, 1.07927352e-01],
 [6.66522892e-01, 4.41303158e-02],
 [3.89493327e-01, 2.38544568e-01],
 [9.76134495e-02, 4.08634410e-01],
 [0.00000000e+00, 0.00000000e+00],
 [3.48657967e-01, 8.09179349e-02]],

[[1.01802118e-02, 2.09520180e-02],
 [9.08105912e-02, 5.48001169e-01],
 [3.61674316e-02, 8.16660092e-01],
 [6.86727383e-03, 1.12792088e-01],
 [1.77571194e-04, 3.75749095e-01],
 [5.10470167e-02, 5.53366546e-03],
```

```
         [1.13039458e-03, 5.97329621e-01],
         [7.73068823e-02, 8.53232713e-01],
         [3.48657967e-01, 8.09179349e-02],
         [0.00000000e+00, 0.00000000e+00]]])
```

In [140…

```
# sum the coordinate differences to get the squared distance
dist_sq = sq_differences.sum(-1)
dist_sq.shape
```

Out[140…

```
(10, 10)
```

In [141…

```
dist_sq
```

```
Out[141…  array([[0.        , 0.39482809, 0.58396752, 0.07028811, 0.232
29143,
          0.11177021, 0.41263358, 0.63815537, 0.25920392, 0.031
13223],
         [0.39482809, 0.        , 0.03906548, 0.31118281, 0.115
22148,
          0.7214276 , 0.11326719, 0.03419159, 0.29135606, 0.638
81176],
         [0.58396752, 0.03906548, 0.        , 0.39700471, 0.125
92501,
          0.86089513, 0.06720011, 0.00812058, 0.54368422, 0.852
82752],
         [0.07028811, 0.31118281, 0.39700471, 0.        , 0.081
642   ,
          0.08882774, 0.19341679, 0.47583627, 0.45602939, 0.119
65936],
         [0.23229143, 0.11522148, 0.12592501, 0.081642  , 0.
,
          0.33528787, 0.02597585, 0.18144286, 0.47249968, 0.375
92667],
         [0.11177021, 0.7214276 , 0.86089513, 0.08882774, 0.335
```

```
28787,
        0.        , 0.52486256, 0.97533281, 0.71065321, 0.056
58068],
       [0.41263358, 0.11326719, 0.06720011, 0.19341679, 0.025
97585,
        0.52486256, 0.        , 0.11988469, 0.62803789, 0.598
46002],
       [0.63815537, 0.03419159, 0.00812058, 0.47583627, 0.181
44286,
        0.97533281, 0.11988469, 0.        , 0.50624786, 0.930
5396 ],
       [0.25920392, 0.29135606, 0.54368422, 0.45602939, 0.472
49968,
        0.71065321, 0.62803789, 0.50624786, 0.        , 0.429
5759 ],
       [0.03113223, 0.63881176, 0.85282752, 0.11965936, 0.375
92667,
        0.05658068, 0.59846002, 0.9305396 , 0.4295759 , 0.
]])
```

As **a quick check of our logic,** we should see that the **diagonal** of this matrix (i.e., the set of distances between each point and itself) is **all zeros:**

In [143…
```
dist_sq.diagonal()
```

Out[143…
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

With the **pairwise square distances** converted, we can now use `np.argsort` to sort **along each row.**

The **leftmost columns** will then give the **indices of the nearest neighbors:**

In [145…
```
nearest = np.argsort(dist_sq, axis=1)
print(nearest)
```

```
[[0 9 3 5 4 8 1 6 2 7]
 [1 7 2 6 4 8 3 0 9 5]
 [2 7 1 6 4 3 8 0 9 5]
 [3 0 4 5 9 6 1 2 8 7]
 [4 6 3 1 2 7 0 5 9 8]
 [5 9 3 0 4 6 8 1 2 7]
 [6 4 2 1 7 3 0 5 9 8]
 [7 2 1 6 4 3 8 0 9 5]
 [8 0 1 9 3 4 7 2 6 5]
 [9 0 5 3 4 8 6 1 2 7]]
```

Notice that the **first column** gives the **numbers 0 through 9 in order:**

This is due to the fact that **each point's closest neighbor is itself.**

By using a **full sort** here, we've actually done **more work than we need to in this case.**

If we're simply interested in the **nearest k neighbors:**

All we need to do is **partition each row** so that the **smallest *K+1* squared distances come first,** with larger distances filling the remaining positions of the array.

We can do this with the `np.argpartition` function:

In [147…
```
K = 2
nearest_partition = np.argpartition(dist_sq, K + 1, axis=1)
```

In [148…
```
nearest_partition.shape
```

Out[148…
```
(10, 10)
```

In [149…
```
nearest_partition
```

Out[149…

```
array([[0, 9, 3, 5, 4, 1, 6, 7, 8, 2],
       [1, 7, 2, 6, 4, 8, 3, 0, 5, 9],
       [1, 7, 2, 6, 4, 8, 3, 0, 5, 9],
       [3, 0, 4, 5, 9, 6, 1, 7, 8, 2],
       [4, 6, 3, 1, 2, 7, 0, 5, 8, 9],
       [3, 9, 5, 0, 1, 2, 6, 7, 8, 4],
       [4, 6, 2, 1, 7, 3, 0, 5, 8, 9],
       [1, 7, 2, 6, 4, 8, 3, 0, 5, 9],
       [1, 0, 8, 9, 3, 5, 6, 7, 2, 4],
       [9, 0, 5, 3, 4, 1, 6, 7, 8, 2]], dtype=int64)
```

In order to **visualize this network of neighbors:**

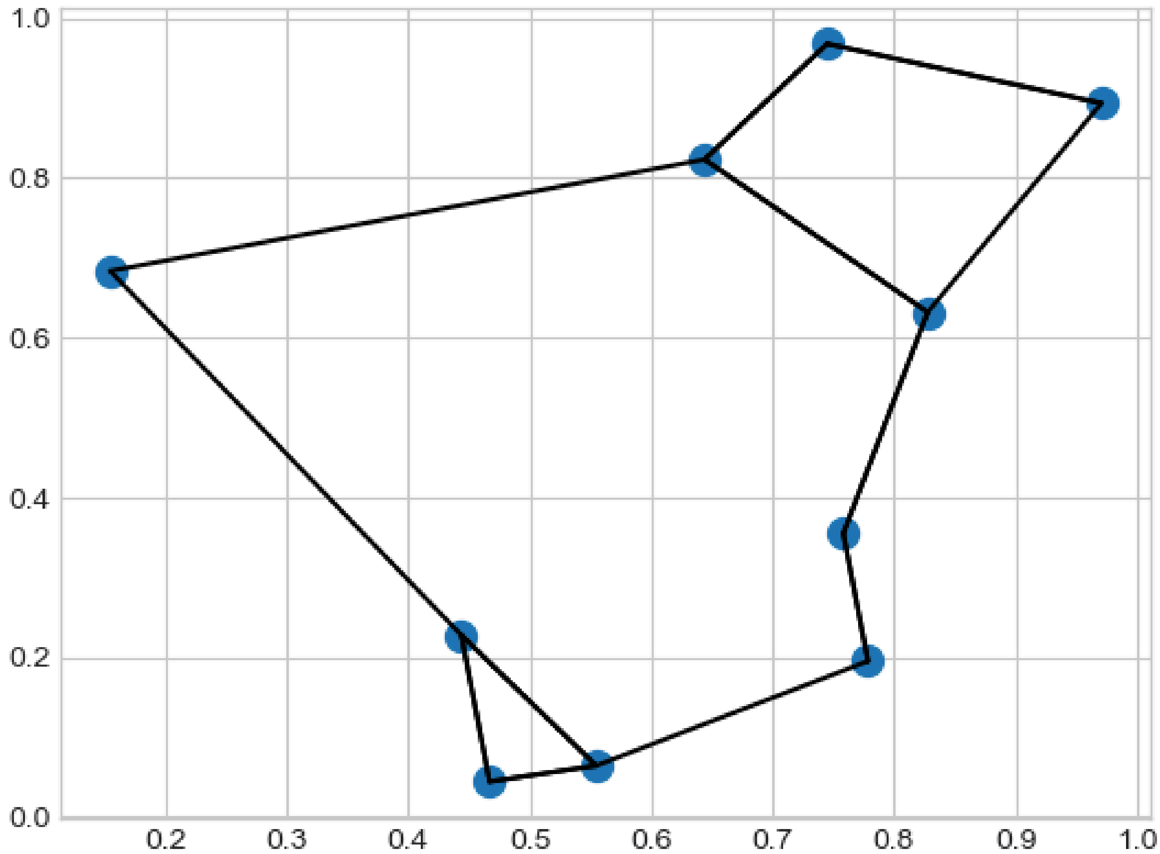
Let's quickly **plot the points** along with **lines** representing the **connections from each point to its two nearest neighbors:**

In [206…

```
plt.scatter(X[:, 0], X[:, 1], s=100)

# draw lines from each point to its two nearest neighbors
K = 2
```

```
for i in range(X.shape[0]):
    for j in nearest_partition[i, :K+1]:
        # plot a line from X[i] to X[j]
        # use some zip magic to make it happen:
        plt.plot(*zip(X[j], X[i]), color='black')
```



1.0
0.8
0.6
0.4
0.2
0.0
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

Each **point** in the plot has **lines drawn to its two nearest neighbors.**

At first glance, it might seem strange that **some of the points have more than two lines** coming out of them:

This is due to the fact that **if point A is one of the two nearest neighbors of point B.**

This **does not necessarily imply** that **point B is one of the two nearest neighbors of point A.**

Although the **broadcasting and row-wise sorting** of this approach might seem **less straightforward** than **writing a loop,** it turns out to be a very **efficient** way of operating on this data in Python.

You might be tempted to do the same type of operation by **manually looping through** the data and **sorting each set of**

neighbors individually.

But this would almost certainly **lead to a slower algorithm** than the **vectorized version** we used.

The **beauty of this approach** is that it's written in a way that's **agnostic to the size of the input data:**

We could as easily compute the neighbors among **100 or 1,000,000 points** in any number of dimensions, and the **code would look the same.**

Finally, when doing **very large nearest neighbor searches,** there are tree-based and/or approximate algorithms that can scale better.