numpy-notes

February 13, 2025

0.1 NumPy Notes

NumPy is a fundamental package for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. This notebook provides a brief introduction to NumPy as well as some explanations of how it has been used in k-means-clustering-demo.ipynb.

For a more methodical introduction, you may find the W3Schools tutorial useful: https://www.w3schools.com/python/numpy/numpy_intro.asp

0.2 NumPy Arrays

A NumPy array is a grid of values, all of the same type, indexed by a tuple of non-negative integers. The number of dimensions is the rank of the array, and the shape of an array is a tuple of integers giving the size of the array along each dimension.

0.3 Creating NumPy Arrays

[1]:
```python
import numpy as np

# Create a NumPy array from a Python list
a = np.array([1, 2, 3])
print("1D array:\n", a)

# Create a 2D NumPy array (matrix)
b = np.array([[1, 2, 3], [4, 5, 6]])
print("\n2D array:\n", b)
```

```
1D array:
 [1 2 3]

2D array:
 [[1 2 3]
 [4 5 6]]
```

[2]:
```python
a.shape
```

[2]:
```
(3,)
```

[3]:
```python
b.shape
```

[3]:
```
(2, 3)
```

0.4 Broadcasting

Broadcasting is a powerful mechanism that allows NumPy to work with arrays of different shapes when performing arithmetic operations. Frequently, we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

For example, if we want to add a constant vector to each row of a matrix, we can do this:

[4]:
```
# Add a vector to each row of a matrix
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
vector = np.array([1, 0, 1])

# Broadcasting happens here
result = matrix + vector

result
```

[4]:
```
array([[ 2,  2,  4],
       [ 5,  5,  7],
       [ 8,  8, 10]])
```

0.5 Understanding np.newaxis

np.newaxis is used here to increase the dimension of the existing array by one more dimension. This is often used in combination with broadcasting.

[5]:
```
# Create an array called v, print it, and print its shape
v = np.array([1, 2, 3])
print(v)
print(v.shape)  # (3,)

print()  # Print blank line

# Use np.newaxis to print v a column vector and then print its shape of the␣
 ↪column vector
print(v[:, np.newaxis])
print(v[:, np.newaxis].shape)  # (3, 1)
```

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

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

0.6 Vectorized Operations

NumPy provides a way to perform operations on arrays without writing explicit loops. This is not only more syntactically convenient but also leads to code that is more efficient.

The k-means clustering demo uses Pythagoras' theorem to calculate the Euclidean distance (https://en.wikipedia.org/wiki/Euclidean_distance) between each datapoint and each centroid using the expression ___np.sqrt(((data - centroids[:, np.newaxis])**2).sum(axis=2))___

Let's break this expression down and see how it works step by step.

0.6.1 Setting up our data and centroid arrays

First we'll create a 2D array representing 4 points, similar to the **data** variable in the k-means clustering demo.

```
[6]: data = np.array([[5.49, 4.86], [5.64, 6.52], [4.76, 4.76], [6.57, 5.76]])
     data
```

```
[6]: array([[5.49, 4.86],
            [5.64, 6.52],
            [4.76, 4.76],
            [6.57, 5.76]])
```

And then create a 2D array representing the coordinates of two centroids, similar to the **centroids** variable in the k-means clustering demo.

```
[7]: centroids = np.array([[5.49, 4.86], [5.64, 6.52]])
     centroids
```

```
[7]: array([[5.49, 4.86],
            [5.64, 6.52]])
```

0.6.2 centroids[:, np.newaxis]

This transforms the centroids array into a shape that's compatible with data for broadcasting. If centroids is a 2D array with shape (n_centroids, n_features), this operation makes it (n_centroids, 1, n_features).

```
[8]: centroids[:, np.newaxis]
```

```
[8]: array([[[5.49, 4.86]],

            [[5.64, 6.52]]])
```

0.6.3 data - centroids[:, np.newaxis]

Thanks to broadcasting, this subtracts each centroid from each point in data, despite their shapes being different. If the shape of data is (n_points, n_features), this results in an array of shape (n_centroids, n_points, n_features), (2, 4, 2) in this case. These are the horizontal and vertical distances from each point to the corresponding centroid.

```
[9]: data - centroids[:, np.newaxis]
```

```
[9]: array([[[ 0.  ,  0.  ],
             [ 0.15,  1.66],
             [-0.73, -0.1 ],
             [ 1.08,  0.9 ]],

            [[-0.15, -1.66],
             [ 0.  ,  0.  ],
             [-0.88, -1.76],
             [ 0.93, -0.76]]])
```

0.6.4 ** 2

This squares each element of the resulting array. Our horizontal and vertical distances will now be squared.

```
[10]: (data - centroids[:, np.newaxis]) ** 2
```

```
[10]: array([[[0.    , 0.    ],
              [0.0225, 2.7556],
              [0.5329, 0.01  ],
              [1.1664, 0.81  ]],

             [[0.0225, 2.7556],
              [0.    , 0.    ],
              [0.7744, 3.0976],
              [0.8649, 0.5776]]])
```

0.6.5 .sum(axis=2)

Sums over the last dimension (the squared horizontal and vertical distances from datapoint to centroid), resulting in the squared Euclidean distance from each point to the corresponding centroid. The shape now is (n_centroids, n_points).

```
[11]: ((data - centroids[:, np.newaxis]) ** 2).sum(axis=2)
```

```
[11]: array([[0.    , 2.7781, 0.5429, 1.9764],
             [2.7781, 0.    , 3.872 , 1.4425]])
```

0.6.6 np.sqrt(...)

This takes the square root of each element, finally giving the Euclidean distance from each point to each centroid.

The distances variable will then refer to an array with the following structure: [[distance from point 0 to centroid 0, distance from point 1 to centroid 0, distance from point 2 to centroid 0, ...] [distance from point 0 to centroid 1, distance from point 1 to centroid 1, distance from point 2 to centroid 1, ...] ...]

```
[12]: distances = np.sqrt(((data - centroids[:, np.newaxis]) ** 2).sum(axis=2))
      distances
```

```
[12]: array([[0.        , 1.66676333, 0.73681748, 1.40584494],
             [1.66676333, 0.        , 1.96773982, 1.20104121]])
```

0.6.7 Deciding which centroid each point is closest to (assigning points to a cluster)

We can now use np.argmin to choose the centroid closest to each point, giving us 0, 1, 0, and 1.

array([[**0.** , 1.66676333 , **0.73681748** , 1.40584494], # distances from each point to cluster 0 [1.66676333, **0.** , 1.96773982 , **1.20104121**] # distances from each point to cluster 1])

Point 0 is closest to centroid 0, point 1 is closest to centroid 1, point 2 is closest to centroid 0, and point 3 is closest to centroid 1.

```
[13]: assignments = np.argmin(distances, axis=0)
      assignments
```

```
[13]: array([0, 1, 0, 1], dtype=int64)
```

0.7 Selecting Initial Centroids

In the k-means clustering demo I indicated that the initial centroids could be chosen randomly using the following code: initial_centroids = data[np.random.choice(data.shape[0], NUMBER_OF_CLUSTERS, replace=False), :]

Run the code cell below several times and you will see that two points from our data are being selected randomly each time.

```
[14]: NUMBER_OF_CLUSTERS = 2

      initial_centroids = data[np.random.choice(
          data.shape[0], NUMBER_OF_CLUSTERS, replace=False), :]

      initial_centroids
```

```
[14]: array([[6.57, 5.76],
             [5.49, 4.86]])
```

This section breaks down the expression step by step.

Here we remind ourselves what the **data** variable refers to: a 2D array representing the x and y coordinates of our data points.

```
[15]: data
```

```
[15]: array([[5.49, 4.86],
             [5.64, 6.52],
             [4.76, 4.76],
        [6.57, 5.76]])
```

We can use .shape to confirm that the array has two dimensions sized 4 and 2.

```
[16]: data.shape
```

```
[16]: (4, 2)
```

.shape[0] gives us the size of the first dimension - this is the number of data points.

```
[17]: data.shape[0]
```

```
[17]: 4
```

np.random.choice is used below to select 2 values in range 0, 1, 2, 3.

```
[18]: np.random.choice(4, 2)
```

```
[18]: array([3, 3])
```

Since data.shape[0] equals 4 and NUMBER_OF_CLUSTERS equals 2, the following line is equivalent.

```
[19]: np.random.choice(data.shape[0], NUMBER_OF_CLUSTERS)
```

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

If you run the above cell enough times, you will see that np.random.choice sometimes picks the same values for both numbers. We can prevent this by setting replace=False. Run the cell below several times and you'll see that it always picks two different numbers. In the context of k-means clustering, this enables us to avoid picking the same point twice as an initial centroid.

```
[20]: np.random.choice(data.shape[0], NUMBER_OF_CLUSTERS, replace=False)
```

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

We can select specific rows from our data array by indexing with a list or array of indexes. Here we select the first and last rows (indexes 0 and 3) using a list.

```
[21]: data[[0, 3]]
```

```
[21]: array([[5.49, 4.86],
             [6.57, 5.76]])
```

And here we do the same thing but using the array of indexes randomly selected by np.random.choice

```
[22]: data[np.random.choice(data.shape[0], NUMBER_OF_CLUSTERS, replace=False)]
```

```
[22]: array([[4.76, 4.76],
             [6.57, 5.76]])
```

In NumPy indexing, the colon : operator means "select all" in the given dimension. In the context of 2D arrays it doesn't change the result, but it is more explicit and therefore often preferred. You can examine the output from the code below to see the effects.

[23]:
```
# Print the first and last rows of the data array.
print(data[[0, 3]])
print()

# Print only the x-coordinate of the first and last rows of the data array.
print(data[[0, 3], 0])
print()

# Print only the y-coordinate of the first and last rows of the data array.
print(data[[0, 3], 1])
print()

# Print the first and last rows of the data array, explicitly selecting the␣
 ↪whole row for printing.
print(data[[0, 3], :])
```

```
[[5.49 4.86]
 [6.57 5.76]]

[5.49 6.57]

[4.86 5.76]

[[5.49 4.86]
 [6.57 5.76]]
```

And now we have arrived back at the final expression for selecting the initial centroids for our clusters:

[24]:
```
data[np.random.choice(data.shape[0], NUMBER_OF_CLUSTERS, replace=False), :]
```

[24]:
```
array([[4.76, 4.76],
       [5.49, 4.86]])
```