NumPy 2

Thomas Schwarz, SJ

Free ebook

- https://riptutorial.com/ebook/numpy

NumPy Operations

- Numpy allows fast operations on array elements
- We can simply add, subtract, multiply or divide by a scalar

```
>>> vector = np.arange(20).reshape(4,5)
>>> vector
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> vector += 1
>>> vector
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20]])
```

NumPy Operations

- Numpy also allows operations between arrays

```
>>> mat = np.random.normal(0,1,(4,5))
>>> mat
array([[ 0.04646031, -1.32970787,  1.16764921, -0.48342653,  0.42295389],
       [ 0.70547825,  1.51980589,  1.46902433, -0.46742839,  1.42472386],
       [ 0.78756679, -0.39975927,  1.24411043, -0.67336526, -0.92416835],
       [ 0.4708628 , -0.29419976, -0.58634161,  0.29038393, -0.78814955]])
>>> vector + mat
array([[ 1.04646031,  0.67029213,  4.16764921,  3.51657347,  5.42295389],
       [ 6.70547825,  8.51980589,  9.46902433,  8.53257161, 11.42472386],
       [11.78756679, 11.60024073, 14.24411043, 13.32663474, 14.07583165],
       [16.4708628 , 16.70580024, 17.41365839, 19.29038393, 19.21185045]])
```

NumPy Operations

- What happens if there is an error?
  - Python would throw an exception, but not so NumPy
    - Example: Create two vectors, one with a zero

```
>>> vector = np.arange(5)
>>> vector2 = np.arange(2,7)
```

- If we divide, we get a warning
- But the result exists, with an inf value for infinity

```
>>> vec = vector2/vector
Warning (from warnings module):
  File "<pyshell#11>", line 1
RuntimeWarning: divide by zero encountered in true_divide
>>> vec
array([       inf, 3.        , 2.        , 1.66666667,
1.5       ])
```

NumPy Operations

- If we divide 0 by 0, we get an nan -- not a value

```
>>> vec=np.arange(4)
>>> vec
array([0, 1, 2, 3])
>>> vec/vec

Warning (from warnings module):
  File "<pyshell#15>", line 1
RuntimeWarning: invalid value encountered in
true_divide
array([nan,  1.,  1.,  1.])
```

NumPy Operations

- There are rules for how to define operations with nan and inf, that make intuitive sense
    - IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754)
- We can create inf directly by saying `np.inf`
    - Example: Infinity divided by infinity is not defined

```
>>> np.inf/np.inf
nan
```

Operations between Vectors and Matrices

- Adding two vectors:

```
>>> v1 = np.array([1,2,3])
>>> v2 = np.array([5,4,3])
>>> v1 + v2
array([6, 6, 6])
```

Operations between Vectors and Matrices

- Adding two matrices

```
>>> m1 = np.array([[1,2,3],[4,5,6],[9,10,0]])
>>> m1
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 9, 10,  0]])
>>> m2 = np.array([[4,2,0],[7,3,1],[5,1,2]])
>>> m2
array([[4, 2, 0],
       [7, 3, 1],
       [5, 1, 2]])
>>> m1+m2
array([[ 5,  4,  3],
       [11,  8,  7],
       [14, 11,  2]])
```

Operations between Vectors and Matrices

- Scalar multiplication

```
>>> v = np.array([5,3,-2,4])
>>> 5*v
array([ 25,  15, -10,  20])
```

Operations between Vectors and Matrices

- Scalar multiplication

```
>>> m1
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 9, 10,  0]])

>>> 3*m1
array([[ 3,  6,  9],
       [12, 15, 18],
       [27, 30,  0]])
```

Operations between Vectors and Matrices

- Element-wise multiplication **is not matrix multiplication**

```
>>> m1
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 9, 10,  0]])
>>> m2
array([[4, 2, 0],
       [7, 3, 1],
       [5, 1, 2]])
>>> m1*m2
array([[ 4,  4,  0],
       [28, 15,  6],
       [45, 10,  0]])
```

Operations between Vectors and Matrices

- **Matrix multiplication uses the (new) @ operator**
  - Python 3.5 and later

```
>>> m1
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 9, 10,  0]])
>>> m2
array([[4, 2, 0],
       [7, 3, 1],
       [5, 1, 2]])
>>> m1@m2
array([[ 33,  11,   8],
       [ 81,  29,  17],
       [106,  48,  10]])
```

Operations between Vectors and Matrices

- Can be used to multiply matrix and vector

```
>>> m = np.array([[2,3],[1,-1]])
>>> v = np.array([1,2])
>>> m@v
array([ 8, -1])
```

- Notice that the vector is in row form
- $\begin{pmatrix} 2 & 3 \\ 1 & -1 \end{pmatrix} \cdot (1, \quad 2) = (8, \quad -1)$
- Follows usage of matlab and Mathematica

Operations between Vectors and Matrices

- Transpose with np.transpose or the .T operator

```
>>> m
array([[ 2,  3],
       [ 1, -1]])
>>> m.T
array([[ 2,  1],
       [ 3, -1]])
```

Operations between Vectors and Matrices

- Thus, could have used

```
>>> m@ v.T
array([ 8, -1])
```

Operations between Vectors and Matrices

- We can use this to make a linear transform of a data set

```
def transform(matrix, dataset):
    return (matrix@ dataset.T).T
```

```
mat = np.array([[.1, .2, .3, .4],
                [.2, .2, .3, .4],
                [.1, -.1, .2, 3],
                [3, 2, 1, -2]
               ])
print(transform(mat, iris))
```

Operations between Vectors and Matrices

- Dot-product of two vectors:
  - ```
    v = np.array([1, 2, 3, 4, 5])
    >>> v@v.T
    55
    >>> np.vdot(v,v)55
    ```
    ```

Operations between Vectors and Matrices

- Can use linear algebra package in numpy
  - numpy.linalg
  - $\begin{pmatrix} 1 & 2 \\ 1 & -1 \end{pmatrix}^{10} = \begin{pmatrix} 243 & 0 \\ 0 & 243 \end{pmatrix}$

```
np.linalg.matrix_power(np.array([[1,2],[1,-1]]),10)
array([[243,   0],
       [  0, 243]])
```

Operations between Vectors and Matrices

- Can calculate matrix inverses
  - Throws LinAlgError if singular

```
>>> np.linalg.inv( np.array([1,-2],[-2,4]) )
Traceback (most recent call last):
…
numpy.linalg.LinAlgError: Singular matrix
```

Operations between Vectors and Matrices

- Can directly solve linear equations
  - Solving $x + 2y = 2, x - y = 3$
    - With solution $x = 8/9, y = -1/3$
  - Gives an error if matrix is not square or singular

```
>>> np.linalg.solve( np.array([[1,2],[1,-1]]) ,
      np.array([2,3]) )
array([ 2.66666667, -0.33333333])
```

NumPy: Universal Array Functions

- There is a plethora of functions that can be applied to a numpy array.
- These are much faster than the corresponding Python functions
- You can find a list in the numpy u-function manual
  - https://docs.scipy.org/doc/numpy/reference/ufuncs.html

NumPy: Universal Array Functions

- There are universal functions around which the operations are wrapped
  - np.add, np.subtract, np.negative, np.multiply, np.divide, np.floor_divide, np.power, np.mod
- The absolute function is
  - abs
  - np.absolute

NumPy: Universal Array Functions

- Trigonometric functions
  - np.sin, np.cos, np.tan, np.arcsin, np.arccos, np.arctan
- Exponents and logarithms
  - np.log, np.log2 (base 2), np.log10 (base 10)
  - np.expm1 (more exact for small arguments)
  - np.log1p (more exact for small arguments)

NumPy:
Universal Array Functions

- Special u-functions:
    - In addition, the submodule scipy.special contains many more specialized functions

NumPy: Universal Array Functions

- Avoid creating temporary arrays
  - If they are large, too much time spent on moving data
  - Specify the array using the 'out' parameter

```
>>> y = np.empty(10)
>>> x = np.arange(1,11)
>>> np.exp(x, out = y)
array([2.71828183e+00, 7.38905610e+00, 2.00855369e+01, 5.45981500e+01,
       1.48413159e+02, 4.03428793e+02, 1.09663316e+03, 2.98095799e+03,
       8.10308393e+03, 2.20264658e+04])
>>> y
array([2.71828183e+00, 7.38905610e+00, 2.00855369e+01, 5.45981500e+01,
       1.48413159e+02, 4.03428793e+02, 1.09663316e+03, 2.98095799e+03,
       8.10308393e+03, 2.20264658e+04])
```

NumPy: Universal Array Functions

- Can use np.min, np.max, sum
- Use np.argmin, np.argmax to find the index of the maximum / minimum element
- Can use np.mean, np.std, np.var, np.median, mp.percentile to get statistics
  - Not the only way, see the scipy module

NumPy: Broadcasting

- Operations can be also made between arrays of different sizes
    - Example 1: adding a scalar (zero-dimensional) to a vector

```
>>> x = np.full(5,1)
>>> x+1
array([2, 2, 2, 2, 2])
```

NumPy: Broadcasting

- Adding a vector to a matrix:
  - Create a matrix

```
>>> matrix = np.arange(1,11).resha
>>> matrix
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
```

  - Create a vector

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

  - Add them together: The vector has been broadcast to a 2 by 5 matrix by doubling the single row

```
>>> matrix+x
array([[ 2,  4,  6,  8, 10],
       [ 7,  9, 11, 13, 15]])
```

NumPy: Broadcasting

- The broadcast rules: Expand a single coordinate in a dimension in one operand to the value in the other

np.arrange(3) + 5

$$\begin{bmatrix}0 & 1 & 2\end{bmatrix} + \begin{bmatrix}5 & 5 & 5\end{bmatrix} = \begin{bmatrix}5 & 6 & 7\end{bmatrix}$$

np.arrange(9).reshape((3,3)) + np.arrange(3)

$$\begin{bmatrix}0 & 1 & 2\\ 3 & 4 & 5\\ 6 & 7 & 8\end{bmatrix} + \begin{bmatrix}0 & 1 & 2\\ 0 & 1 & 2\\ 0 & 1 & 2\end{bmatrix} = \begin{bmatrix}0 & 2 & 4\\ 3 & 5 & 6\\ 0 & 8 & 10\end{bmatrix}$$

np.arrange(3).reshape((3,1)) + np.arrange(3)

$$\begin{bmatrix}0 & 0 & 0\\ 1 & 1 & 1\\ 2 & 2 & 2\end{bmatrix} + \begin{bmatrix}0 & 1 & 2\\ 0 & 1 & 2\\ 0 & 1 & 2\end{bmatrix} = \begin{bmatrix}0 & 1 & 2\\ 1 & 2 & 3\\ 2 & 3 & 4\end{bmatrix}$$

NumPy: Broadcasting

- Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading site
- Rule 2: If the shape of two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape
- Rule 3: If in any dimensions the sizes disagree and neither is equal to 1, an error is raised

Neat Example

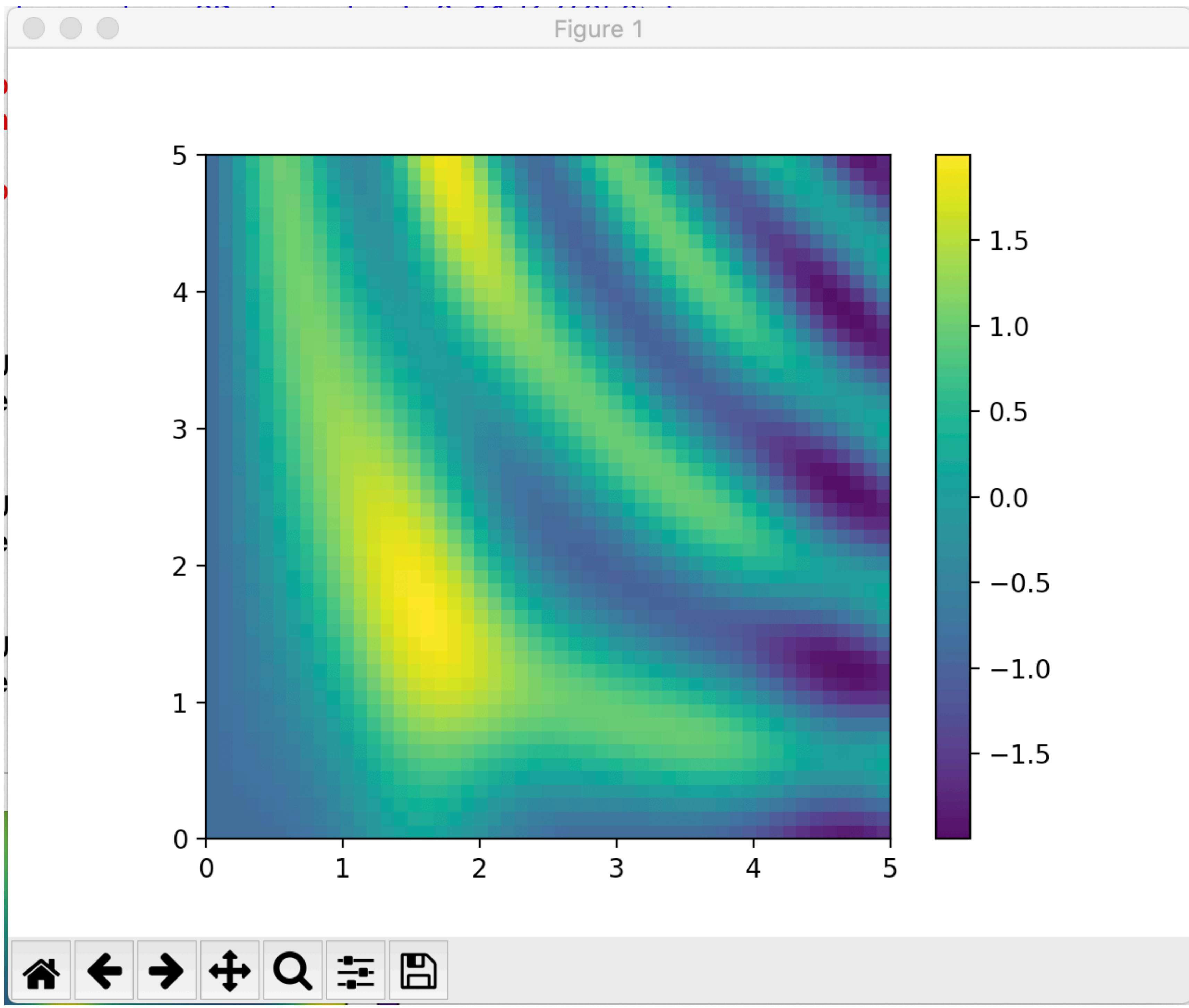
- We combine broadcasting with mathplotlib
    - Using IDLE, we need to call the show function at the end.

NumPy: Broadcasting

- Create a row and a column vector x and y
- Then use broadcasting to combine them for something two-dimensional
- This will get displayed

```
import matplotlib.pyplot as plt
def prob7():
    x = np.linspace(0,5,51)
    y = np.linspace(0,5,51).reshape(51,1)
    z = np.sin(x)**5+np.cos(10+x*y)
    plt.imshow(z, origin='lower', extent=[0, 5, 0
           cmap='viridis')
    plt.colorbar()
    plt.show()
```

NumPy: Broadcasting



NumPy: Fancy Indexing

- Fancy indexing:
  - Use an array of indices in order to access a number of array elements at once

NumPy: Fancy Indexing

- Example:
  - Create matrix

```
>>> mat = np.random.randint(0,10,(3,5))
>>> mat
array([[3, 2, 3, 3, 0],
       [9, 5, 8, 3, 4],
       [7, 5, 2, 4, 6]])
```

  - Fancy Indexing:

```
>>> mat[(1,2),(2,3)]
array([8, 4])
```

NumPy: Fancy Indexing

- Application:
  - Creating a sample of a number of points
- Create a large random array representing data points

```
>>> mat = np.random.normal(100,20, (200,2))
```

- Select the x and y coordinates by slicing

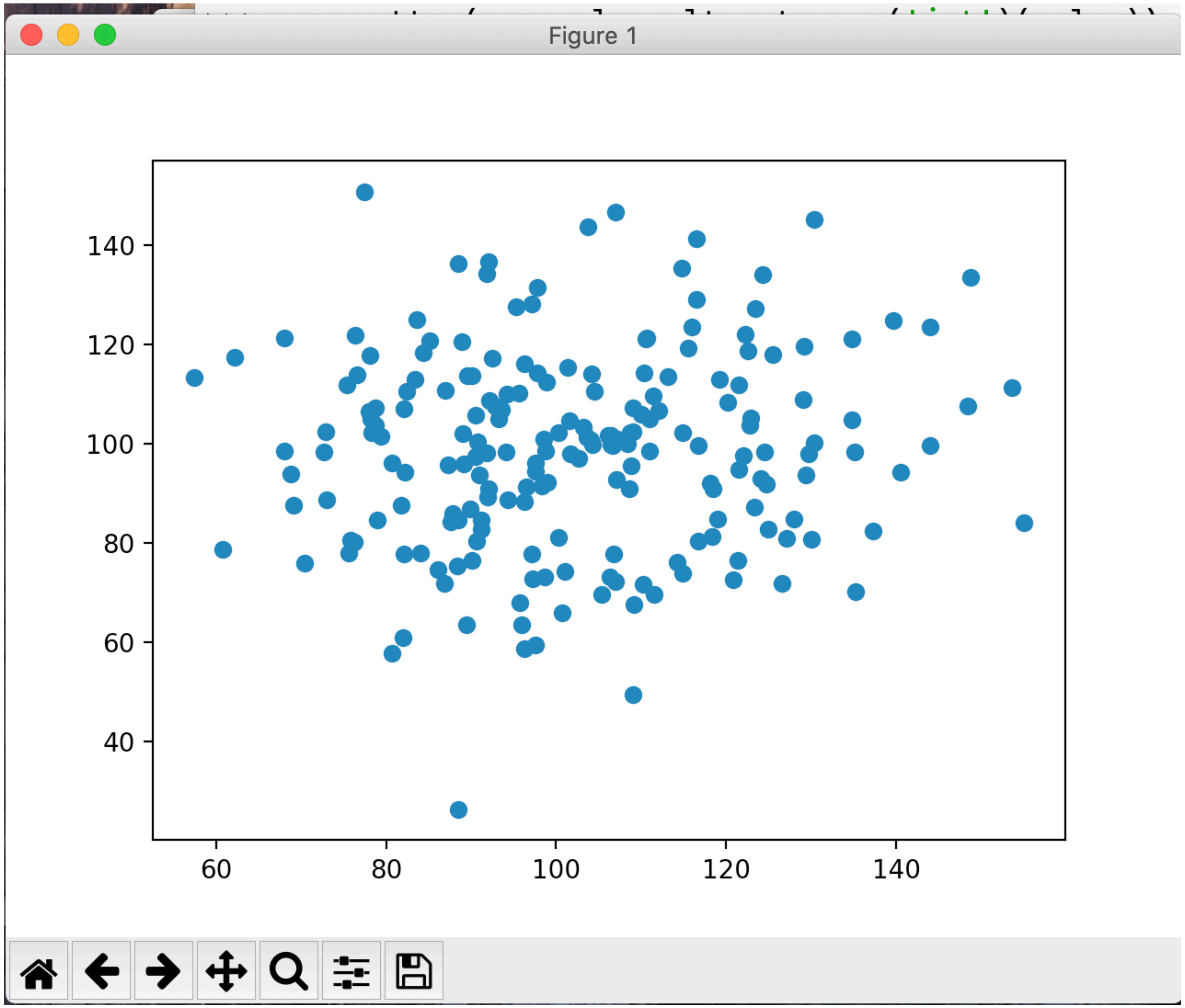
```
>>> x=mat[:,0]
>>> y=mat[:,1]
```

NumPy: Fancy Indexing

- Create a matplotlib figure with a plot inside it

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(1,1,1)
>>> ax.scatter(x,y)
>>> plt.show()
```

NumPy: Fancy Indexing



NumPy: Fancy Indexing

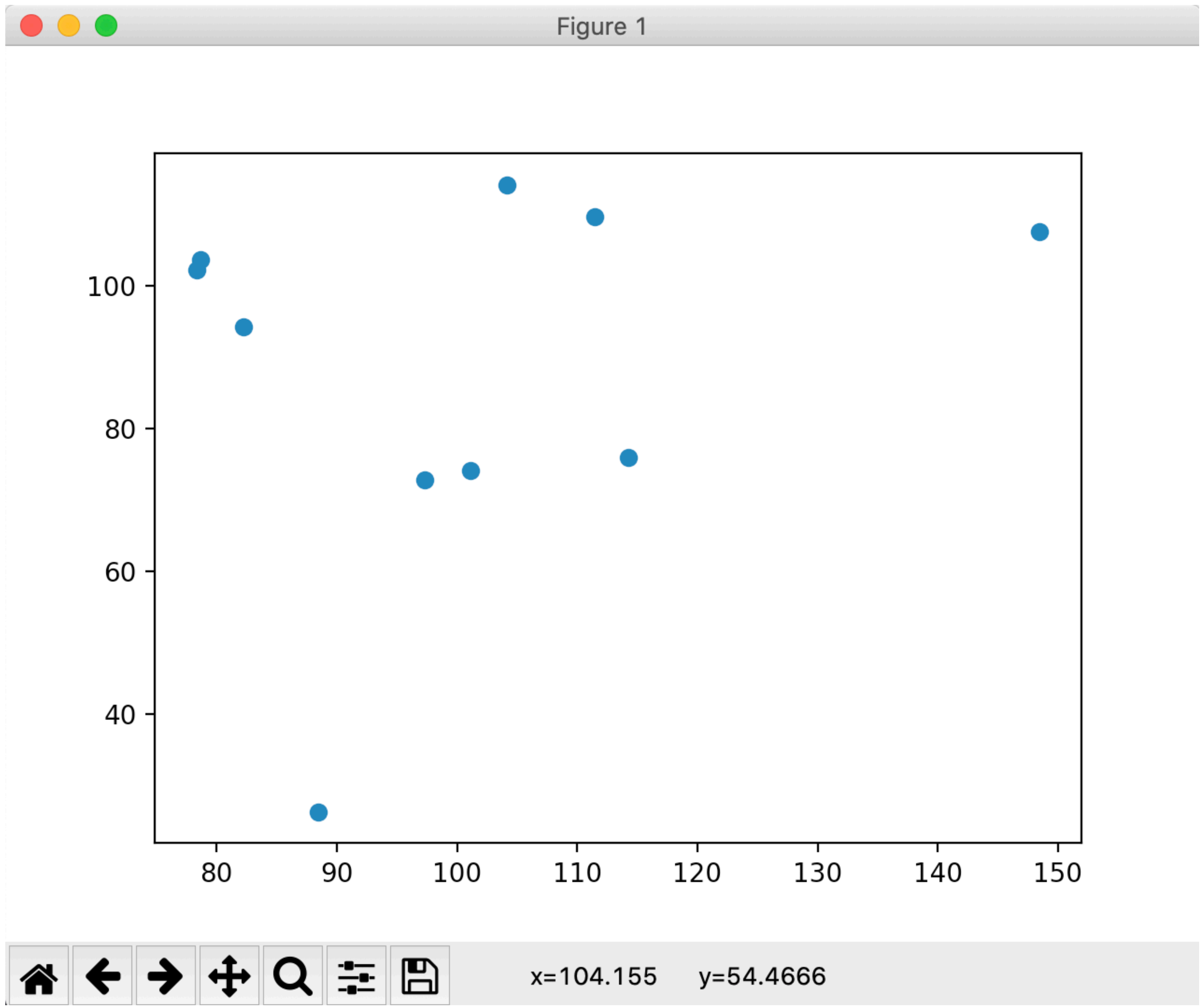
- Create a list of potential indices

```
>>> indices = np.random.choice(np.arange(0,200,1),10)
>>> indices
array([ 32,  93, 172, 134,  90,  66, 109, 158, 188,
30])
```

- Use fancy indexing to create the subset of points

```
>>> subset = mat[indices]
```

NumPy: Fancy Indexing



Simple Stats

- Recall iris data set
  - After normalization

```
>>> iris
array([[0.22222222, 0.625     , 0.06779661, 0.04166667],
       [0.16666667, 0.41666667, 0.06779661, 0.04166667],
       [0.11111111, 0.5       , 0.05084746, 0.04166667],
       [0.08333333, 0.45833333, 0.08474576, 0.04166667],
       [0.19444444, 0.66666667, 0.06779661, 0.04166667],
       [0.30555556, 0.79166667, 0.11864407, 0.125     ],
       [0.08333333, 0.58333333, 0.06779661, 0.08333333],
       [0.19444444, 0.58333333, 0.08474576, 0.04166667],
       [0.02777778, 0.375     , 0.06779661, 0.04166667],
```

Simple Stats

- Calculate average along of all values

```
>>> np.mean(iris)
0.4483046924042686
```

- Much more important: calculate average **along an axis**

```
>>> np.mean(iris, axis=0)
array([0.4287037 , 0.43916667, 0.46757062,
0.45777778])
```

Simple Stats

- Simularly: np.min, np.max, np.median
  - With version in case nan (not a value) is present
- Example: Normalizing the iris data set
-
```
def normalize(array):
    maxs = np.max(array, axis = 0)
    mins = np.min(array, axis = 0)
    return (array-mins)/(maxs-mins)
```

Simple Stats

- Or normalize to have mean 0 and standard deviation 1

```
def normalizeS(array):
    means = np.mean(array, axis = 0)
    stdevs = np.std(array, axis = 0)
    return (array - means)/stdevs
```

Simple Stats

- Can determine percentiles and quantiles

```
>>> iris[:5,:]
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2]])
   ])
>>> np.percentile(iris, 5, axis=0)
array([4.6  , 2.345, 1.3  , 0.2  ])
np.percentile(iris, 95, axis=0)
array([7.255, 3.8  , 6.1  , 2.3  ])
```

Broadcast Application

- Getting the difference matrix of a vector
  $(v_0, \quad v_1, \quad , \ldots, \quad v_{n-1})$

$$\begin{pmatrix} v_0 - v_0 & v_0 - v_1 & \cdots & v_0 - v_{n-1} \\ v_1 - v_0 & v_1 - v_1 & \cdots & v_1 - v_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ v_{n-1} - v_0 & v_{n-1} - v_1 & \cdots & v_{n-1} - v_{n-1} \end{pmatrix}$$

Broadcast Application

- Because of broadcast rules, this will not work

```
>>> v = np.array([1,2,3,4,5,6,7])
>>> v - v.T
array([0, 0, 0, 0, 0, 0, 0])
```

Broadcast Application

- But we can embed the vector into a two-dimensional vector in two different ways

```
>>> v[None,:]
array([[1, 2, 3, 4, 5, 6, 7]])
>>> v[:,None]
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6],
       [7]])
```

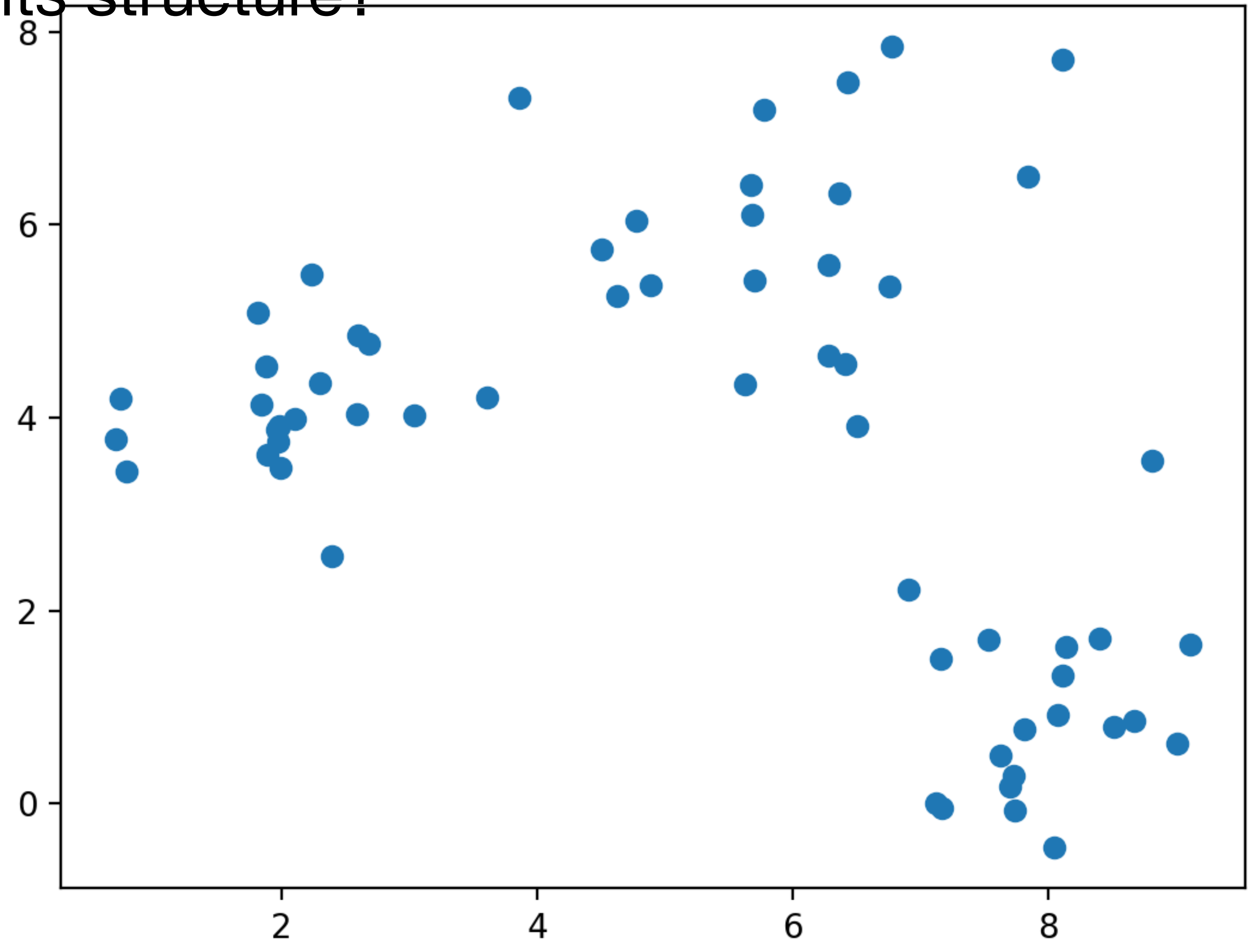
Broadcast Application

- Now we can use broadcasting

```
>>> v[:,None]-v[None,:]
array([[ 0, -1, -2, -3, -4, -5, -6],
       [ 1,  0, -1, -2, -3, -4, -5],
       [ 2,  1,  0, -1, -2, -3, -4],
       [ 3,  2,  1,  0, -1, -2, -3],
       [ 4,  3,  2,  1,  0, -1, -2],
       [ 5,  4,  3,  2,  1,  0, -1],
       [ 6,  5,  4,  3,  2,  1,  0]])
```

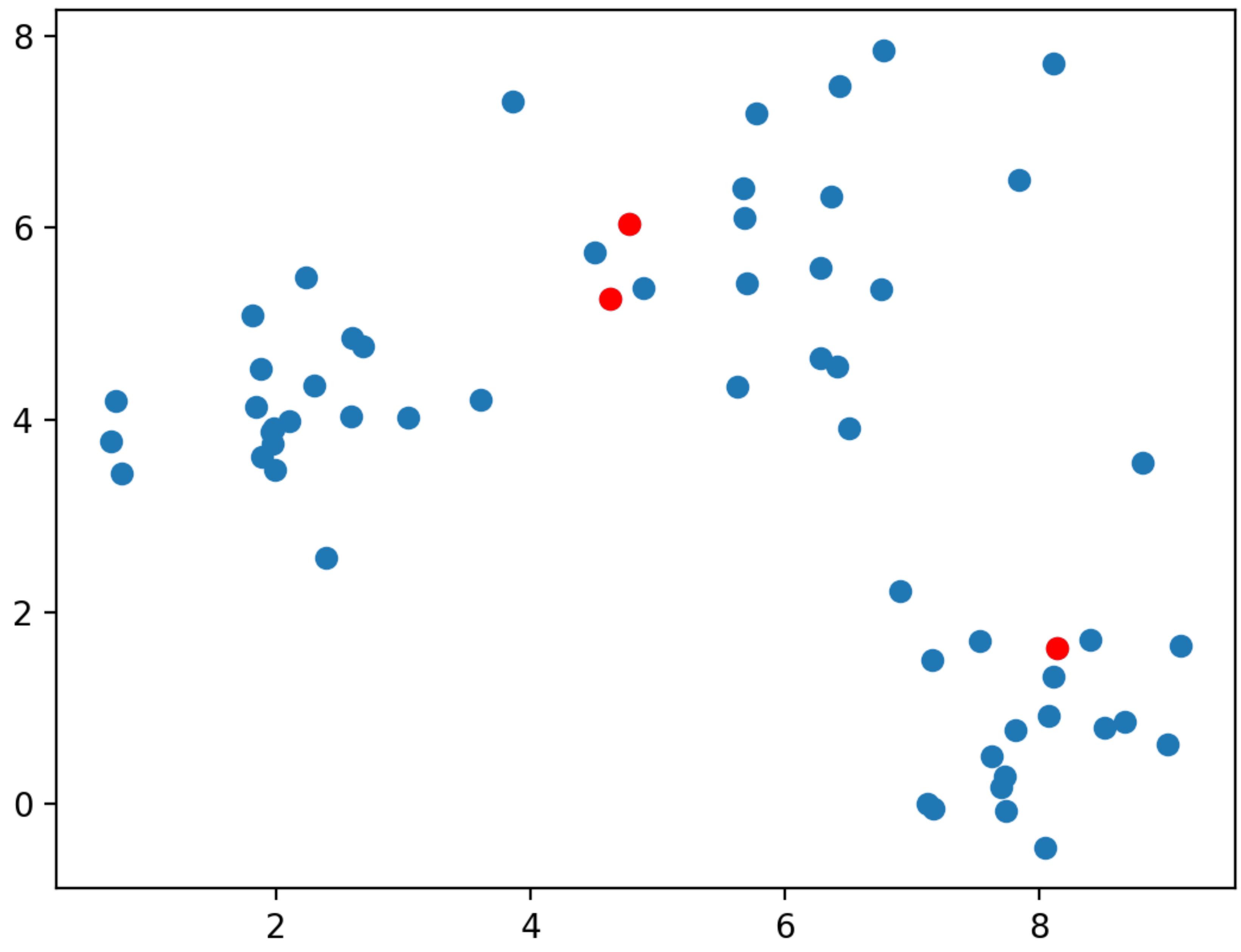
k-means clustering

- Given a set of data, can we cluster it even if we do not know its structure?



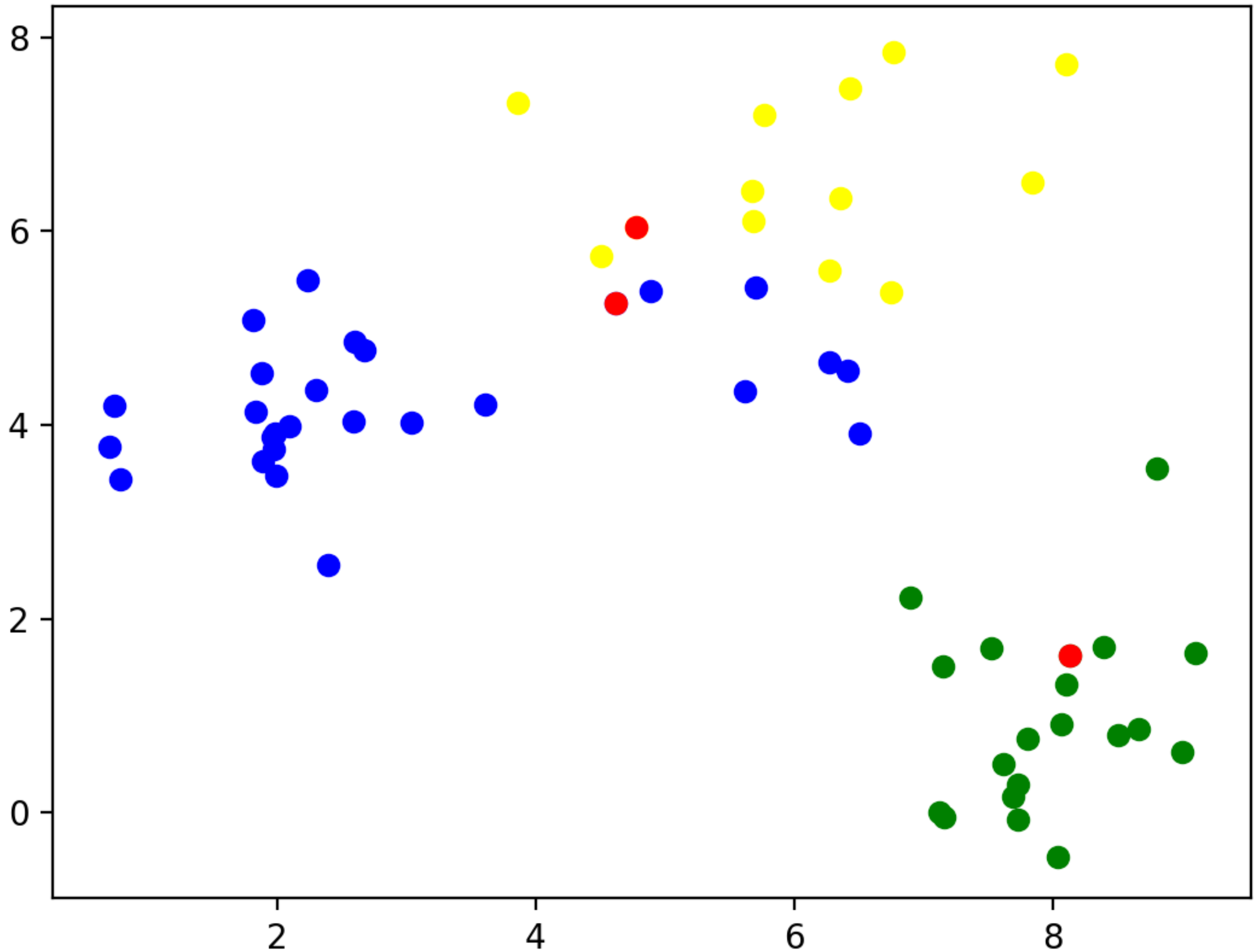
k-means clustering

- Guess a number of clusters and pick *k* arbitrary points



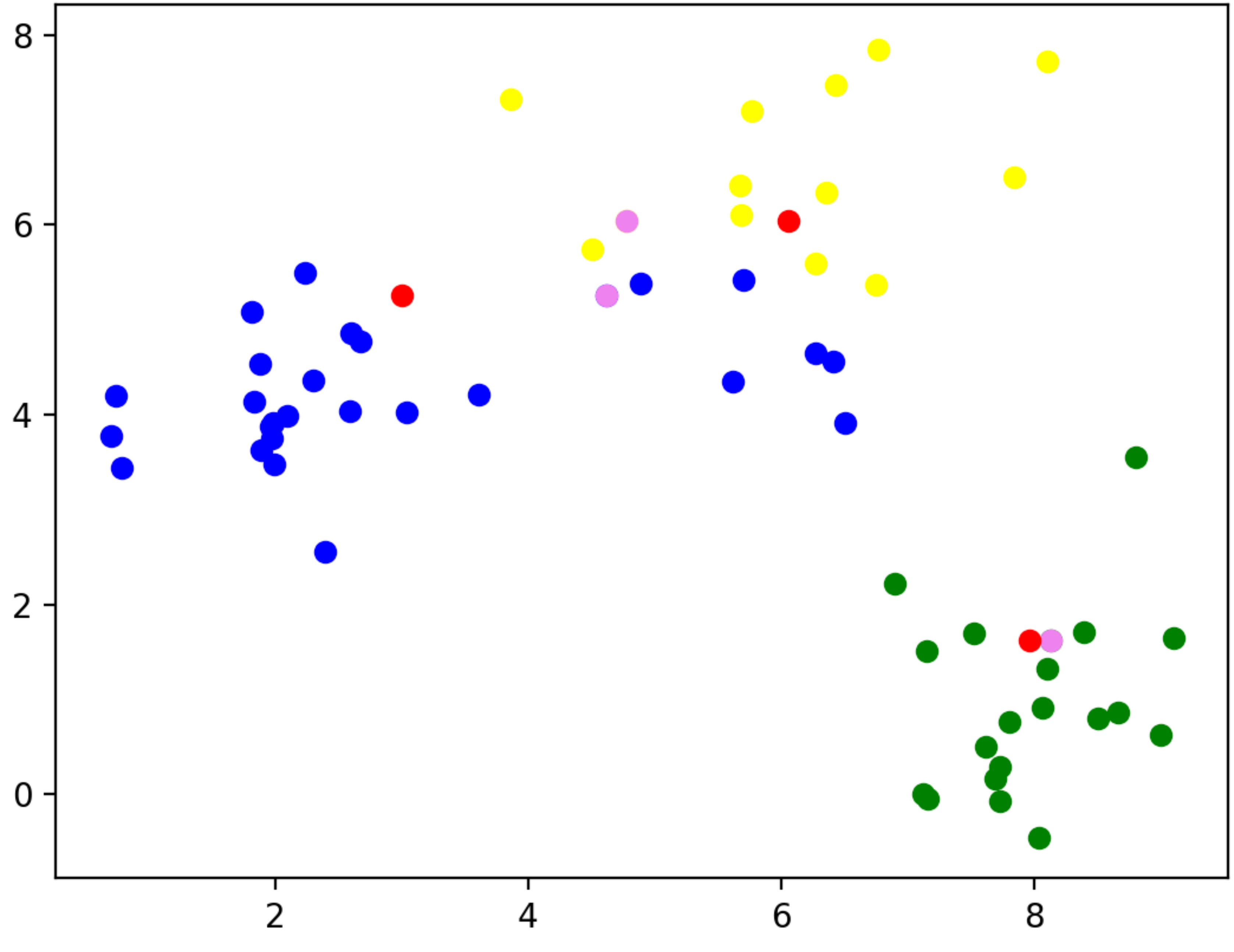
k-means clustering

- Classify all points according to which of the points they are closest



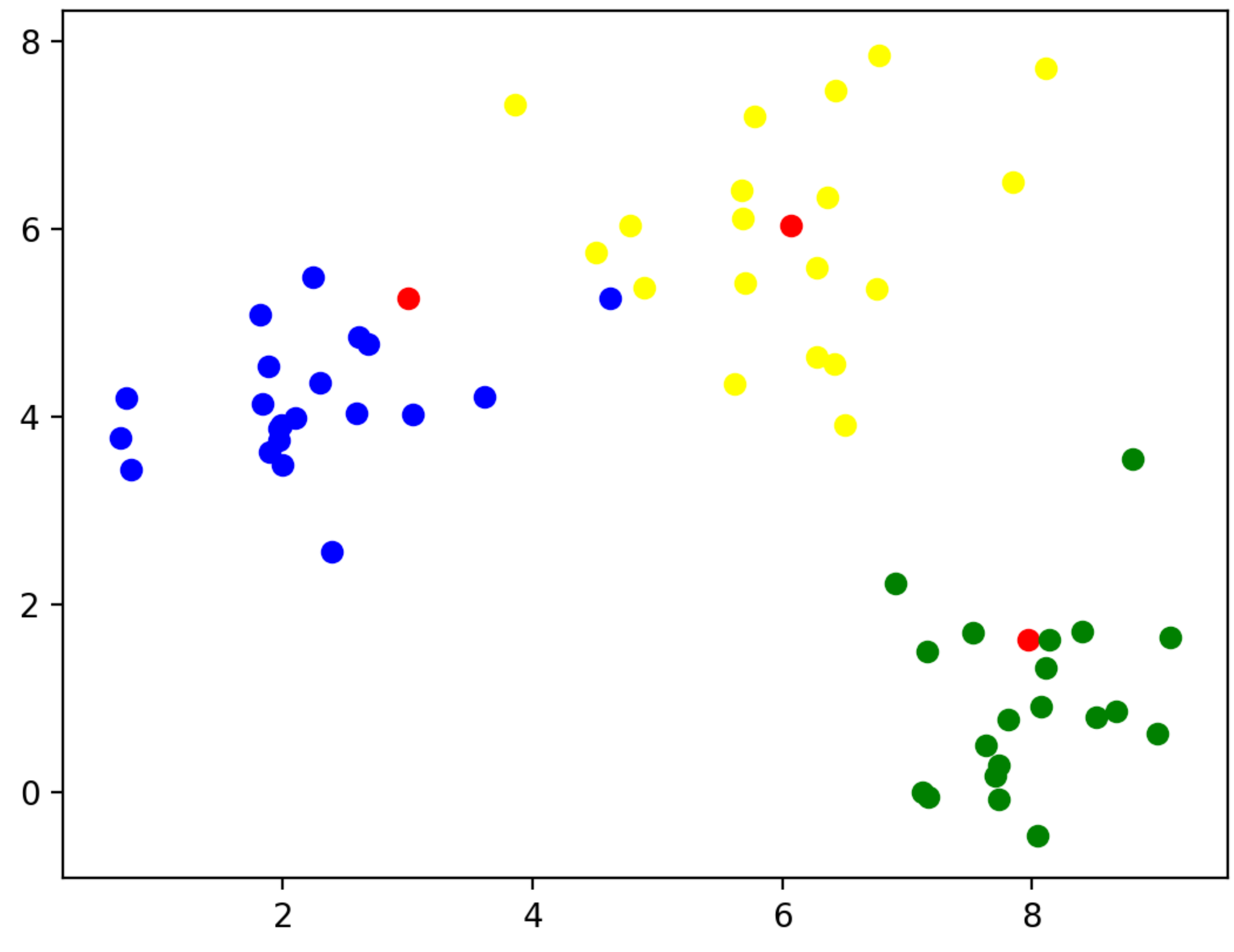
k-means clustering

- Calculate the mean of all the data points and set it as the new center



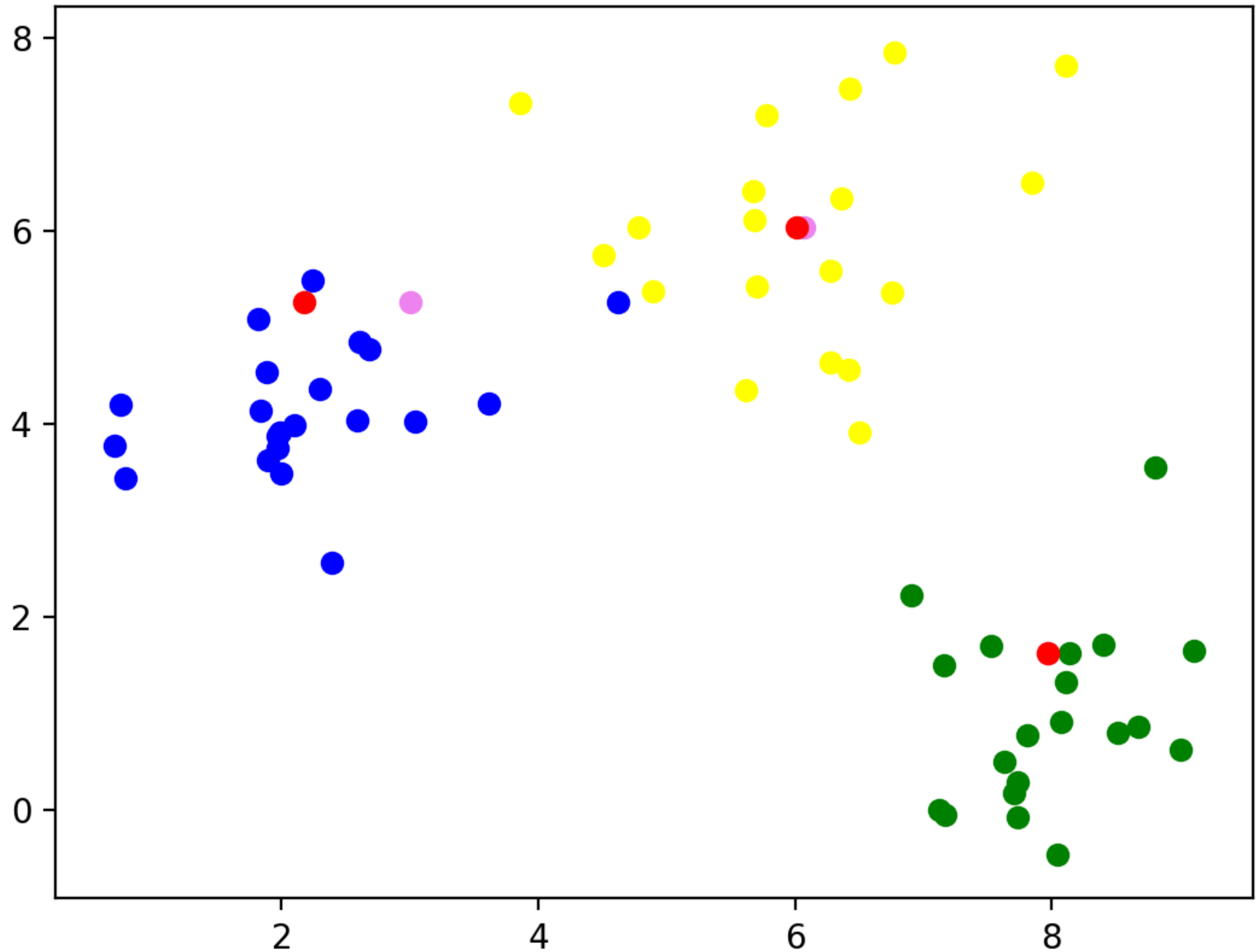
k-means clustering

- Reclassify all the points according to their closeness to the new centers



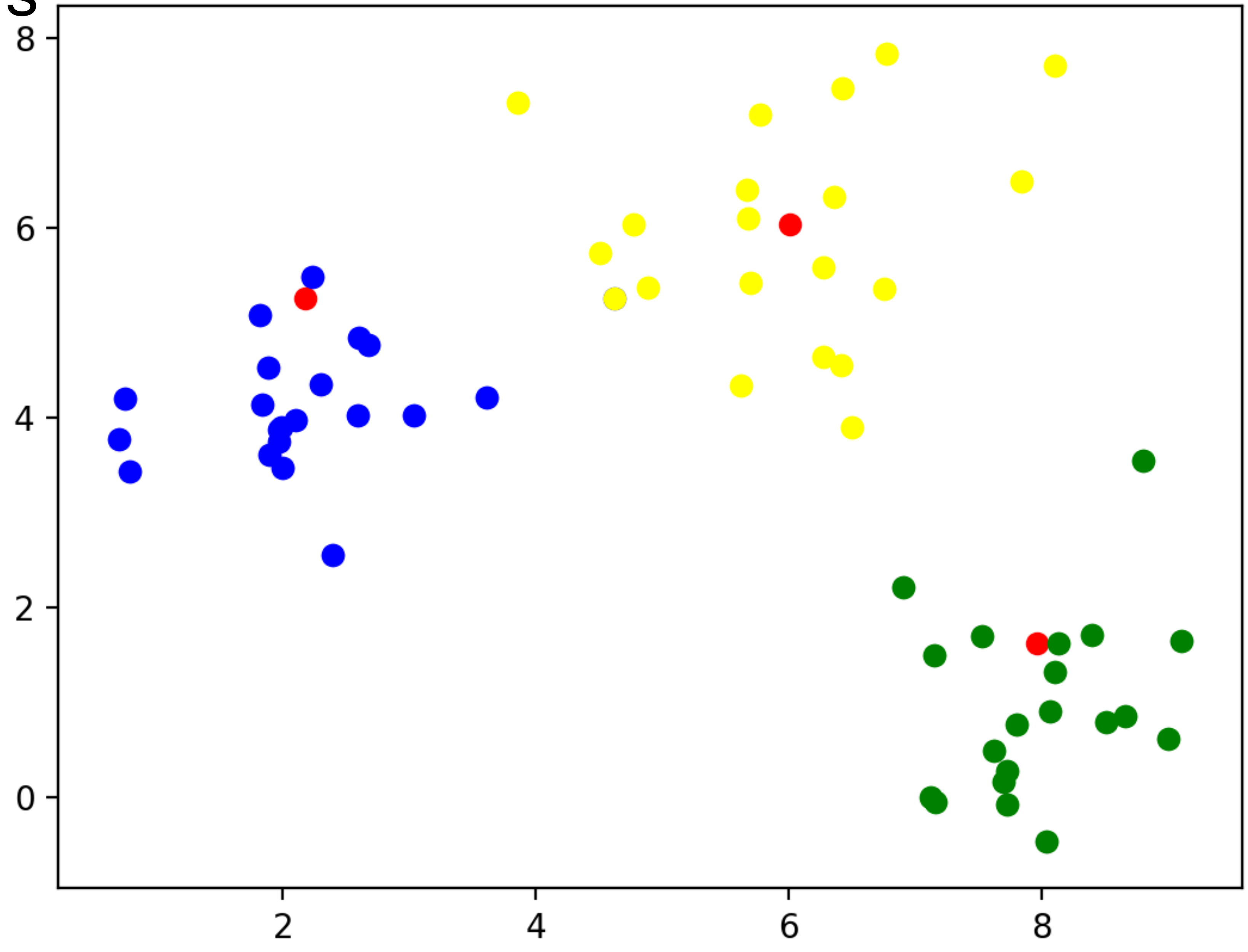
k-means clustering

- Now calculate the new centers of the groups



k-means clustering

- Repeat: Classify according to closeness to the new centers



k-means clustering

- Continue
  - The centers no longer move when points are no longer moved between different categories

k-means clustering

- Implementation
  - Find starting points by random selection

```
def cluster(data, k, limit):
    centers = data[ np.random.choice(np.arange(data.shape[0]), k, 
replace=False),: ]
    for _ in range(limit):
        distances = ((data[:,:,None] - 
centers.T[None, :, :])**2).sum(axis=1)
        classification = np.argmin( distances, axis=1)
        new_centers = np.array([data[classification==j,:].mean(axis=0) 
for j in range(k)])
        if np.max(np.abs(new_centers - centers)) < 0.01:
            break
        else:
            centers = new_centers
    else:  #loop did not end
        print('No convergence')
    return centers
```

k-means clustering

- Enter a limited loop:

```
def cluster(data, k, limit):
    centers = data[ np.random.choice(np.arange(data.shape[0]), k, 
replace=False),: ]
    for _ in range(limit):
        distances = ((data[:,:,None] - 
centers.T[None, :, :])**2).sum(axis=1)
        classification = np.argmin( distances, axis=1)
        new_centers = np.array([data[classification==j,:].mean(axis=0) 
for j in range(k)])
        if np.max(np.abs(new_centers - centers)) < 0.01:
            break
        else:
            centers = new_centers
    else:  #loop did not end
        print('No convergence')
    return centers
```

- Use the previous trick to calculate the difference between all points and the centers

```
def cluster(data, k, limit):
    centers = data[ np.random.choice(np.arange(data.shape[0]), k,
replace=False),: ]
    for _ in range(limit):
        distances = ((data[:,:,None] -
centers.T[None, :, :])**2).sum(axis=1)
        classification = np.argmin( distances, axis=1)
        new_centers = np.array([data[classification==j,:].mean(axis=0)
for j in range(k)])
        if np.max(np.abs(new_centers - centers)) < 0.01:
            break
        else:
            centers = new_centers
    else:  #loop did not end
        print('No convergence')
    return centers
```

- For each point, find the closest distance

```
def cluster(data, k, limit):
    centers = data[ np.random.choice(np.arange(data.shape[0]), k, 
replace=False),: ]
    for _ in range(limit):
        distances = ((data[:,:,None] - 
centers.T[None, :, :])**2).sum(axis=1)
        classification = np.argmin( distances, axis=1)
        new_centers = np.array([data[classification==j,:].mean(axis=0) 
for j in range(k)])
        if np.max(np.abs(new_centers - centers)) < 0.01:
            break
        else:
            centers = new_centers
    else:  #loop did not end
        print('No convergence')
    return centers
```

- The new centers are obtained by taking the mean of the points with a given classification

```
def cluster(data, k, limit):
    centers = data[ np.random.choice(np.arange(data.shape[0]), k,
replace=False),: ]
    for _ in range(limit):
        distances = ((data[:,:,None] -
centers.T[None, :, :])**2).sum(axis=1)
        classification = np.argmin( distances, axis=1)
        new_centers = np.array([data[classification==j,:].mean(axis=0)
for j in range(k)])
        if np.max(np.abs(new_centers - centers)) < 0.01:
            break
        else:
            centers = new_centers
    else:  #loop did not end
        print('No convergence')
    return centers
```

- If the centers do not move, we are done

```
def cluster(data, k, limit):
    centers = data[ np.random.choice(np.arange(data.shape[0]), k, 
replace=False),: ]
    for _ in range(limit):
        distances = ((data[:,:,None] - 
centers.T[None, :, :])**2).sum(axis=1)
        classification = np.argmin( distances, axis=1)
        new_centers = np.array([data[classification==j,:].mean(axis=0) 
for j in range(k)])
        if np.max(np.abs(new_centers - centers)) < 0.01:
            break
        else:
            centers = new_centers
    else:  #loop did not end
        print('No convergence')
    return centers
```

- Possible to not have convergence
    - For production quality code: consider raising an exception

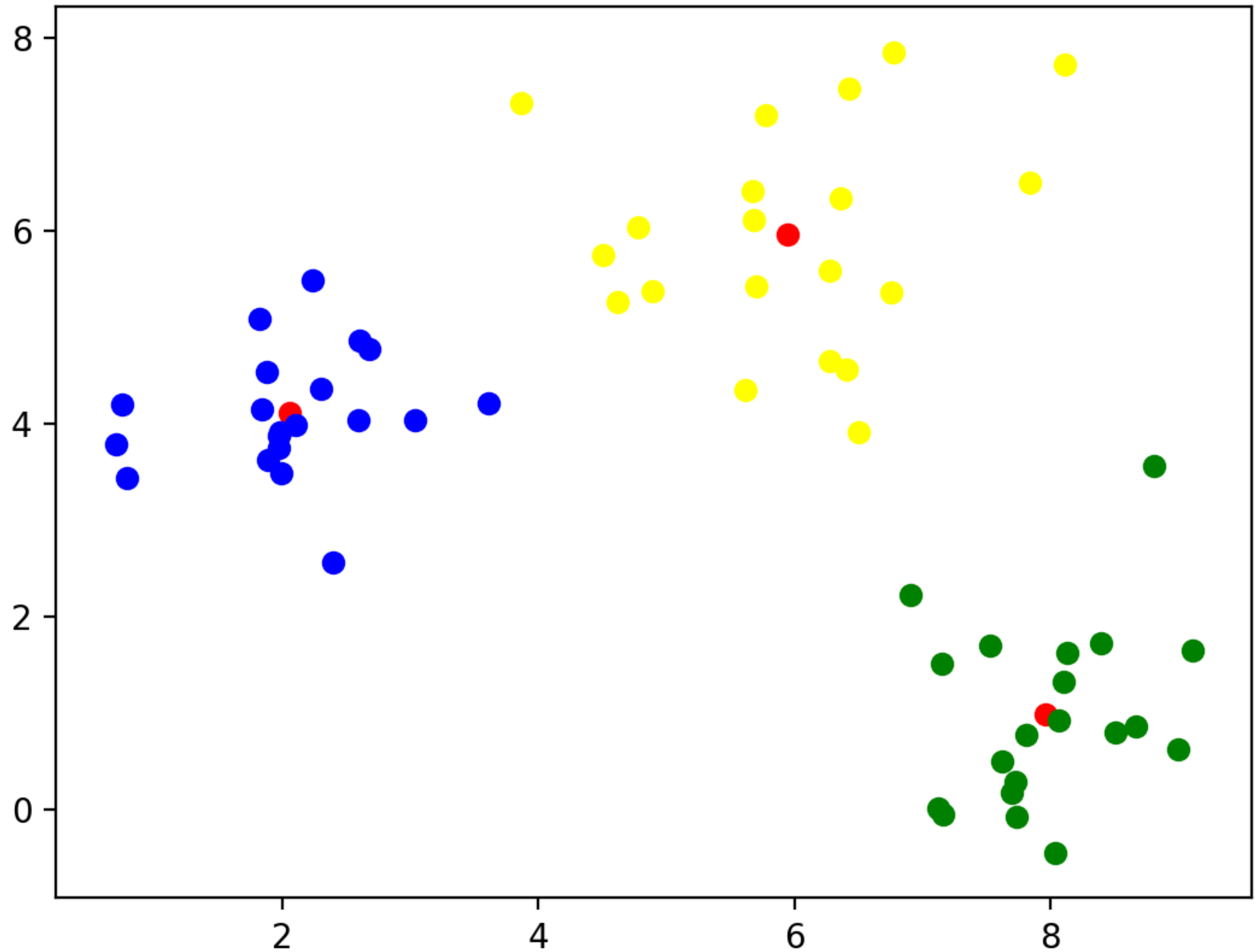
```
def cluster(data, k, limit):
    centers = data[ np.random.choice(np.arange(data.shape[0]), k, 
replace=False),: ]
    for _ in range(limit):
        distances = ((data[:,:,None] - 
centers.T[None, :, :])**2).sum(axis=1)
        classification = np.argmin( distances, axis=1)
        new_centers = np.array([data[classification==j,:].mean(axis=0) 
for j in range(k)])
        if np.max(np.abs(new_centers - centers)) < 0.01:
            break
        else:
            centers = new_centers
    else:  #loop did not end
        print('No convergence')
    return centers
```

- The loop stabilized, we are done

```
def cluster(data, k, limit):
    centers = data[ np.random.choice(np.arange(data.shape[0]), k,
replace=False),: ]
    for _ in range(limit):
        distances = ((data[:,:,None] -
centers.T[None, :, :])**2).sum(axis=1)
        classification = np.argmin( distances, axis=1)
        new_centers = np.array([data[classification==j,:].mean(axis=0)
for j in range(k)])
        if np.max(np.abs(new_centers - centers)) < 0.01:
            break
        else:
            centers = new_centers
    else:  #loop did not end
        print('No convergence')
    return centers
```

k-means clustering

- Final result

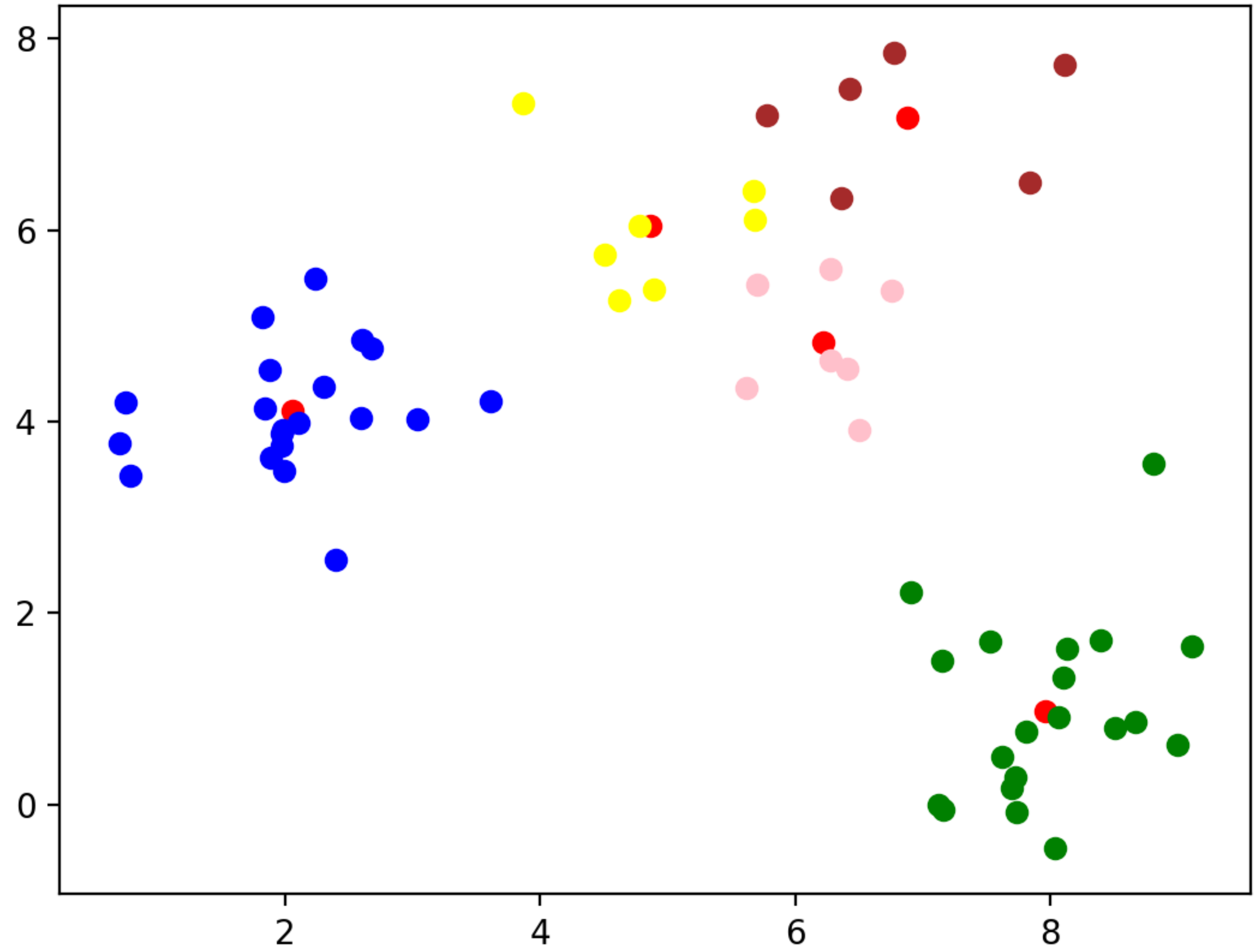


k-means clustering

- This worked because I used normalvariate to generate points around (2,4), (8,1), and (6,6)

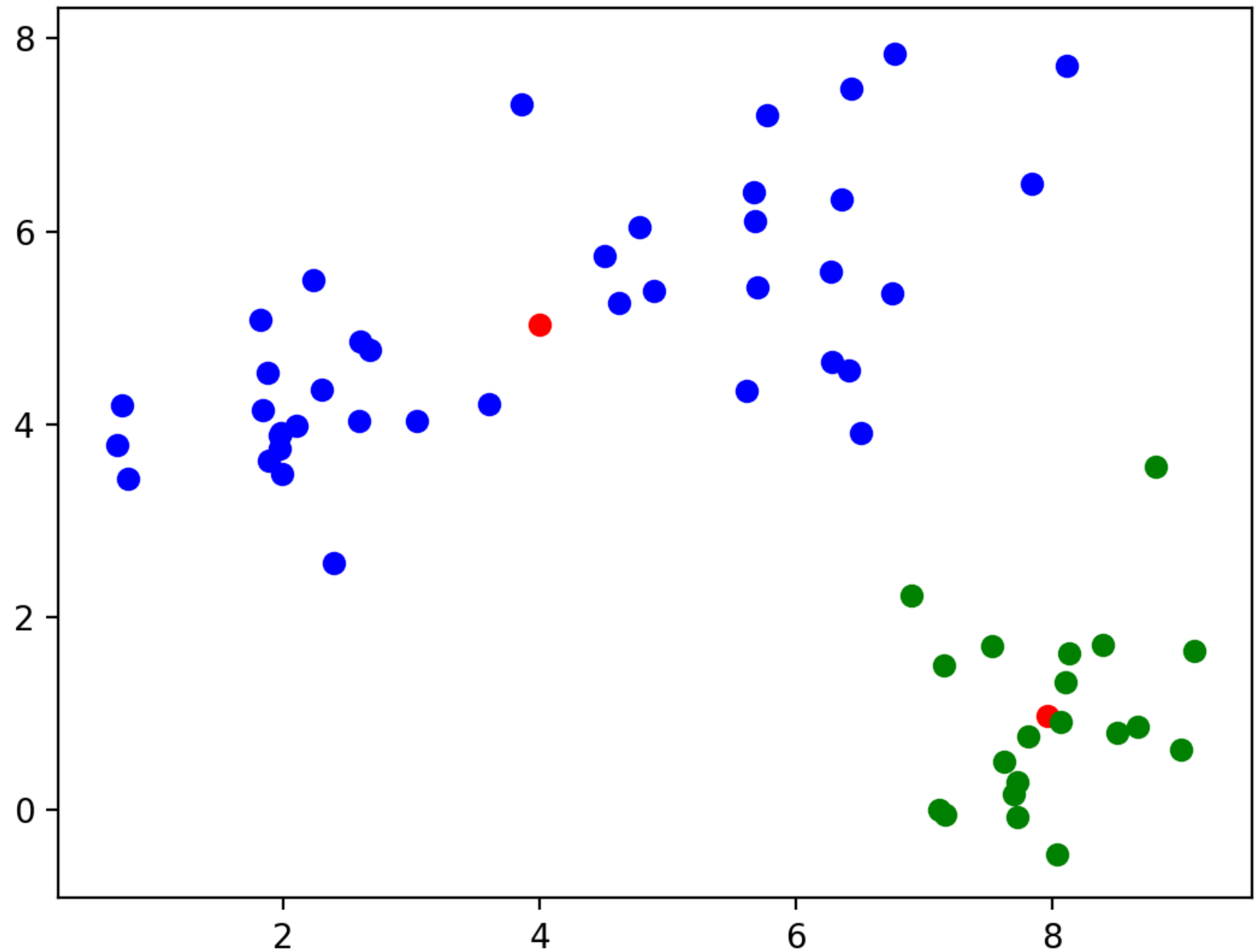
k-means clustering

- What happens if we use a different *k*?
- *k*=5: A cluster gets arbitrarily split



k-means clustering

- k=2 Two clusters get merged



k-means clustering

- Let's try this out on the Iris data set
  - We only keep the measurements
  - We can normalize data using the min-max method

```
def normalize(array):
    maxs = np.max(array, axis = 0)
    mins = np.min(array, axis = 0)
    return (array-mins)/(maxs-mins)
```

k-means clustering

- Now we try clustering without normalizing
    - The first 50 are 'Setosa', the next 50 are 'Virginica', then 'Variegata'
    - Sample with k = 5:
        - Recognizes 'Setosa' cluster, but not the other two

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

k-means clustering

- With *k* = 3: looks a bit better, but still cannot recognize Virginia and Variegata

```
[2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 1 0 1 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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 0 1 1 1 1
 1 1 0 0 1 1 1 1 0 1 0 1 0 1 1 0 0 1 1 1 1 1 0 1 1 1 1 0 1 1 1 0 1 1 1 0 1
 1 0]
```

k-means clustering

- Best results with k = 2:

```
[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 0 0 0 0 0 0 0 1 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 0 0 0 0 1 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]
```

k-means clustering

- With mean-std normalization, results are more encouraging, but still not satisfactory

```
[2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 0 2 2 2 2 2 2 2 2 1 1 1 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0
 0 1 1 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 0 1 1 1 1
 1 1 0 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 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 2 2 2 0 0 0 2 0 0 0 0 0 0 0 0 2 0 0 0 0 2 0 0 0
 0 2 2 2 0 0 0 0 0 0 0 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 2 2 2 2 0 2 0 2 2
 0 2 0 0 2 2 2 2 0 2 0 2 0 2 2 0 0 2 2 2 2 2 0 0 2 2 2 0 2 2 2 0 2 2 2 0 2
 2 0]
[1 2 2 2 1 1 2 2 2 2 1 2 2 2 1 1 1 1 1 1 2 1 2 2 2 2 2 1 2 2 2 2 1 1 2 2 1
 2 2 2 2 2 2 2 1 2 1 2 1 2 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 0 0 0
 0 0]
```

k-means clustering

- With *k* = 2, we cluster into Setosa and not-Setosa

```
[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 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 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 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 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 1 1
 1 1]
```

k-means clustering

- With *k*=4: still no separation

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

k-means clustering

- Morale:
  - With k-means clustering
    - Definitely need to normalize data set
    - Need to repeat method many times
      - Pick the one with the lowest sum of Euclidean distances