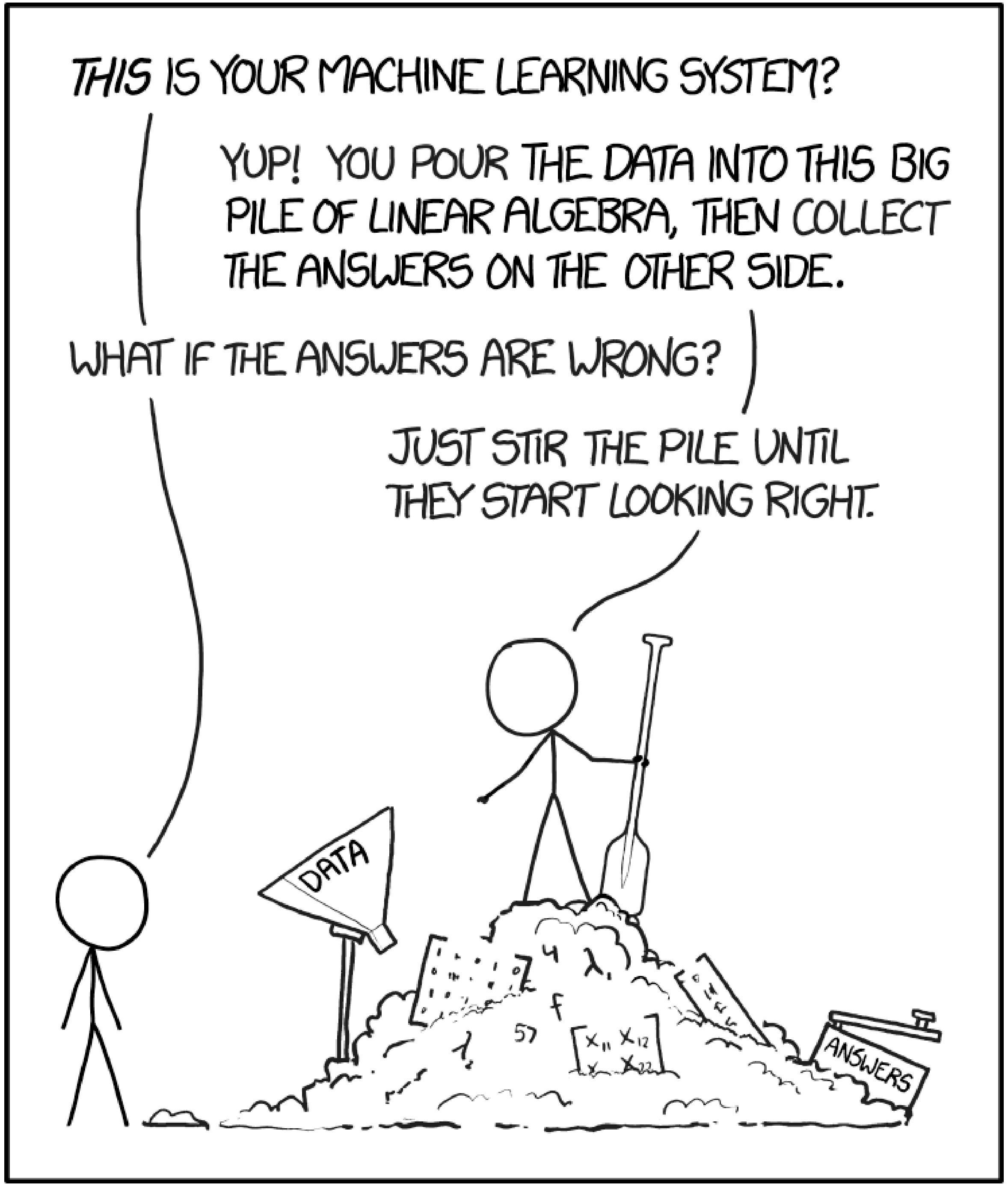
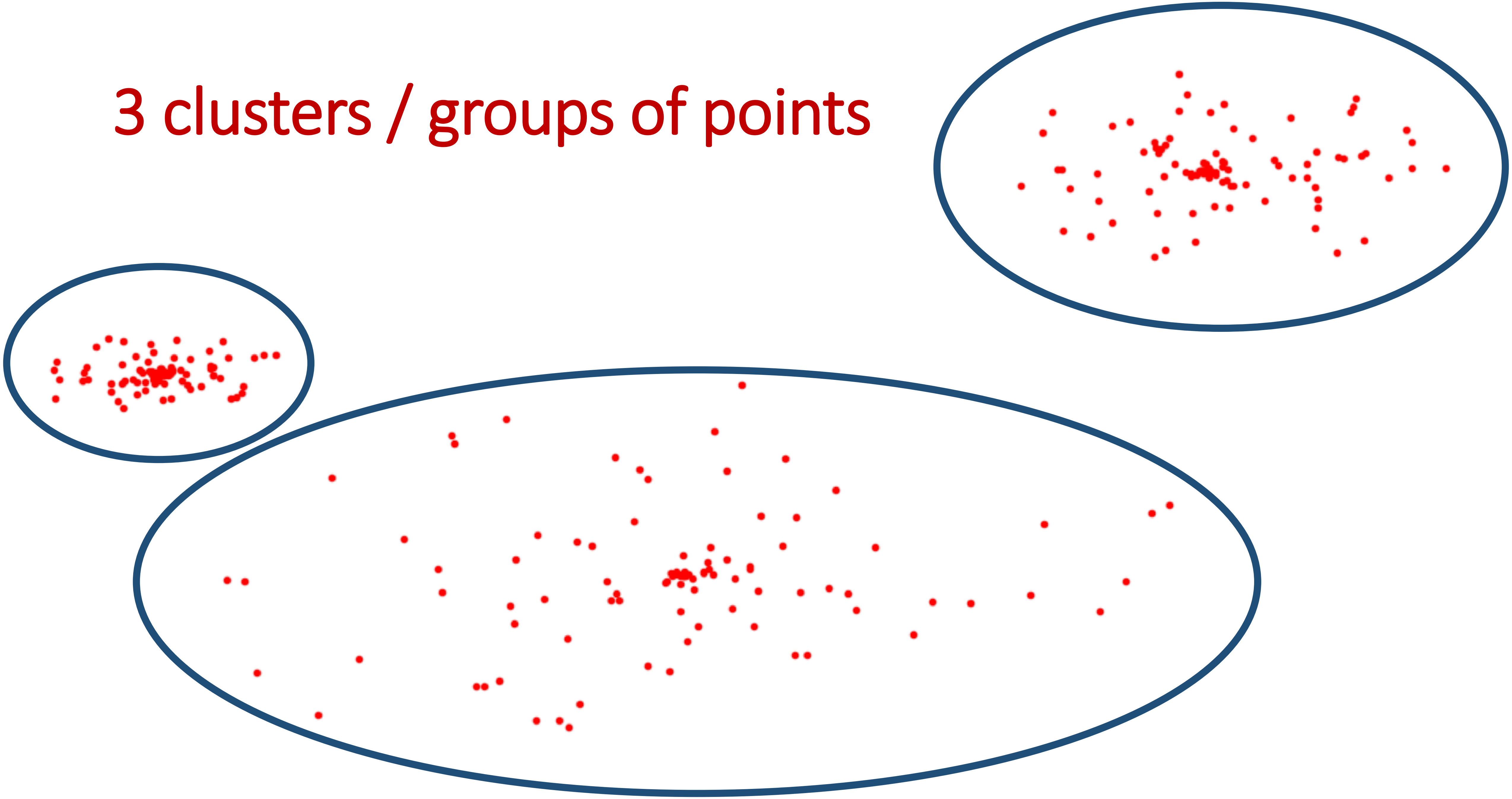


xkcd.com/1838/

Clustering

- k-means
- scipy.cluster.vq.kmeans
- DBSCAN*
- neural networks

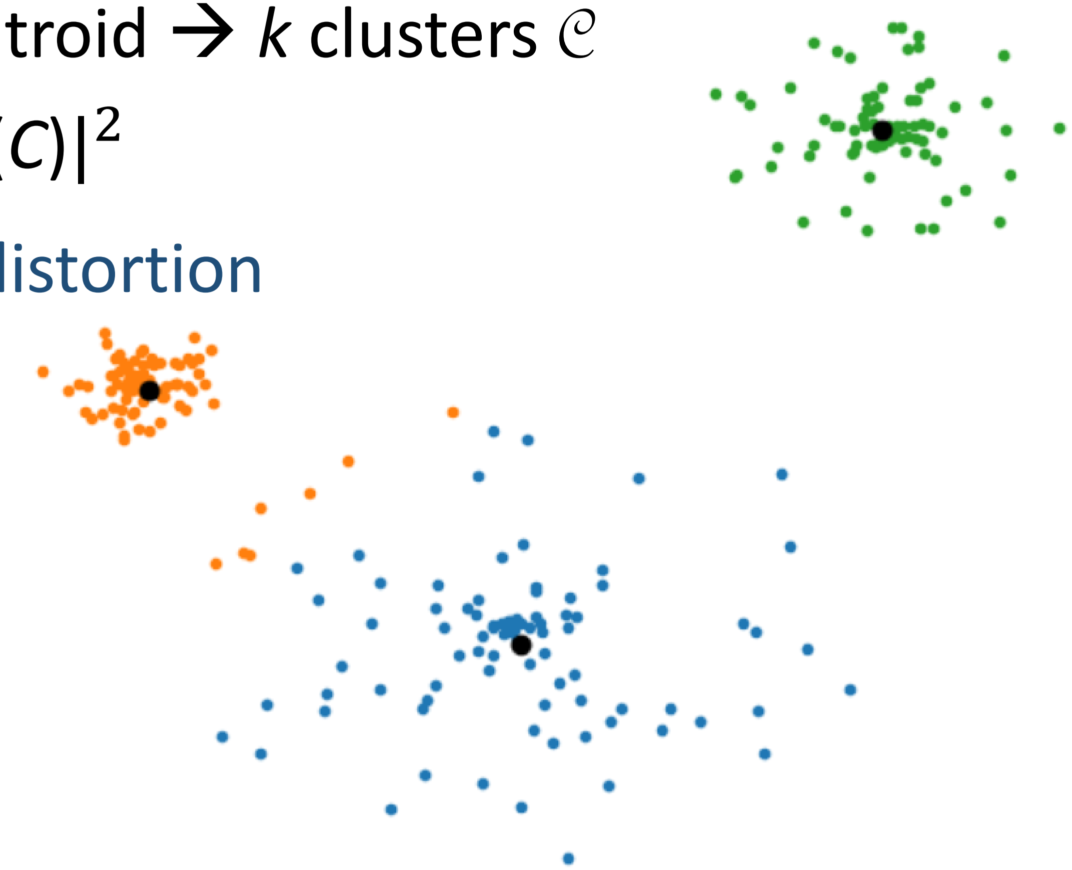
3 clusters / groups of points



Clustering = Optimization problem

Example: *k*-means

- Given n input points and an integer k, find k *centroid* points
- Assign each input point to nearest centroid → k clusters $\mathcal{C}$
- distortion = $\sum_{C \in \mathcal{C}} \sum_{p \in C} |p - \text{centroid}(C)|^2$
- Goal : Find k centroids that minimize distortion

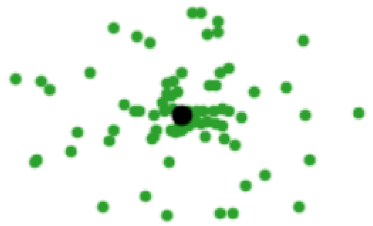


k-means for $k = 1$

- Let the centroid point c for a point set C be the point minimizing the

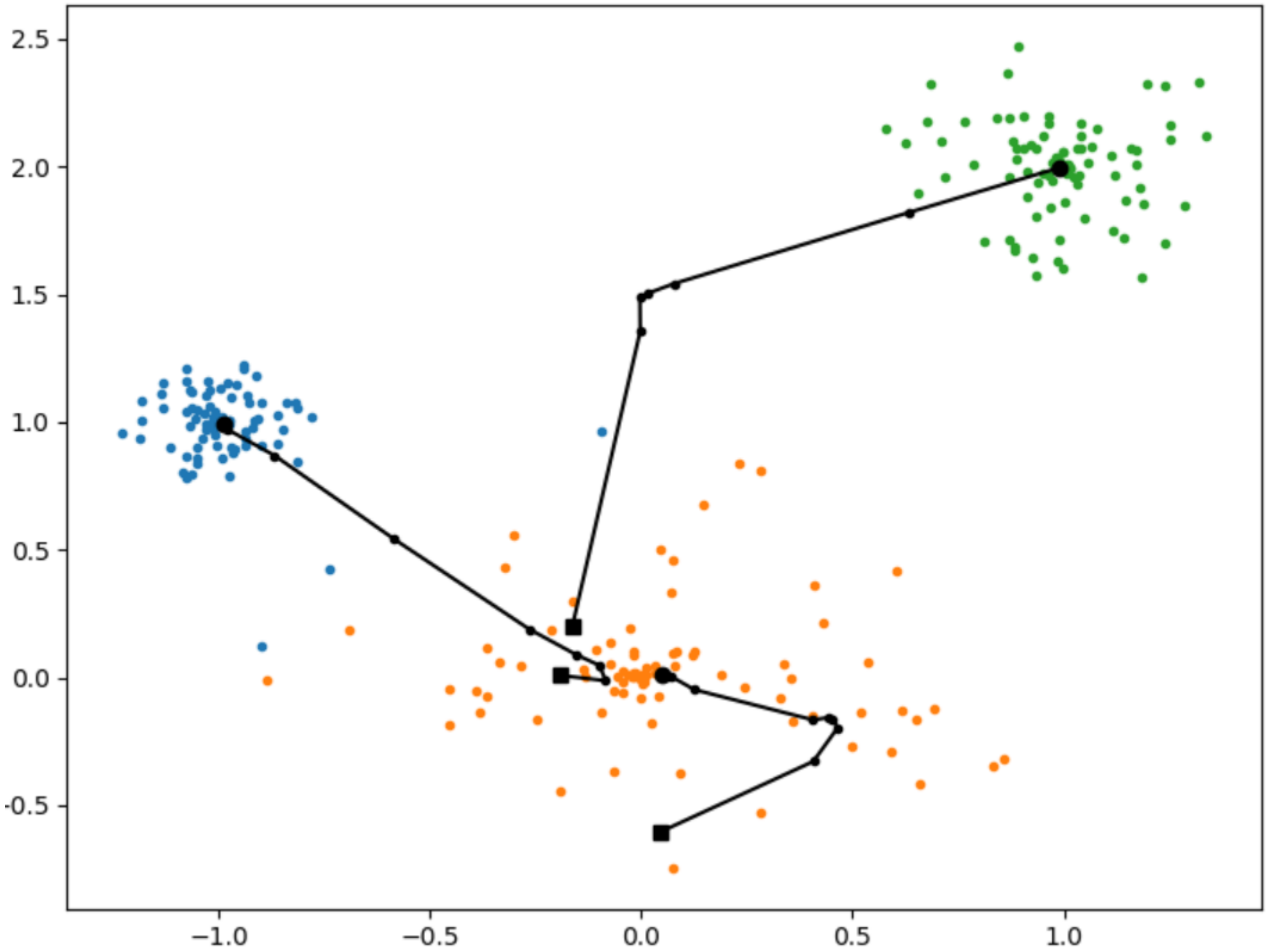
$$\text{distortion} = \sum_{p \in C} |p - c|^2$$

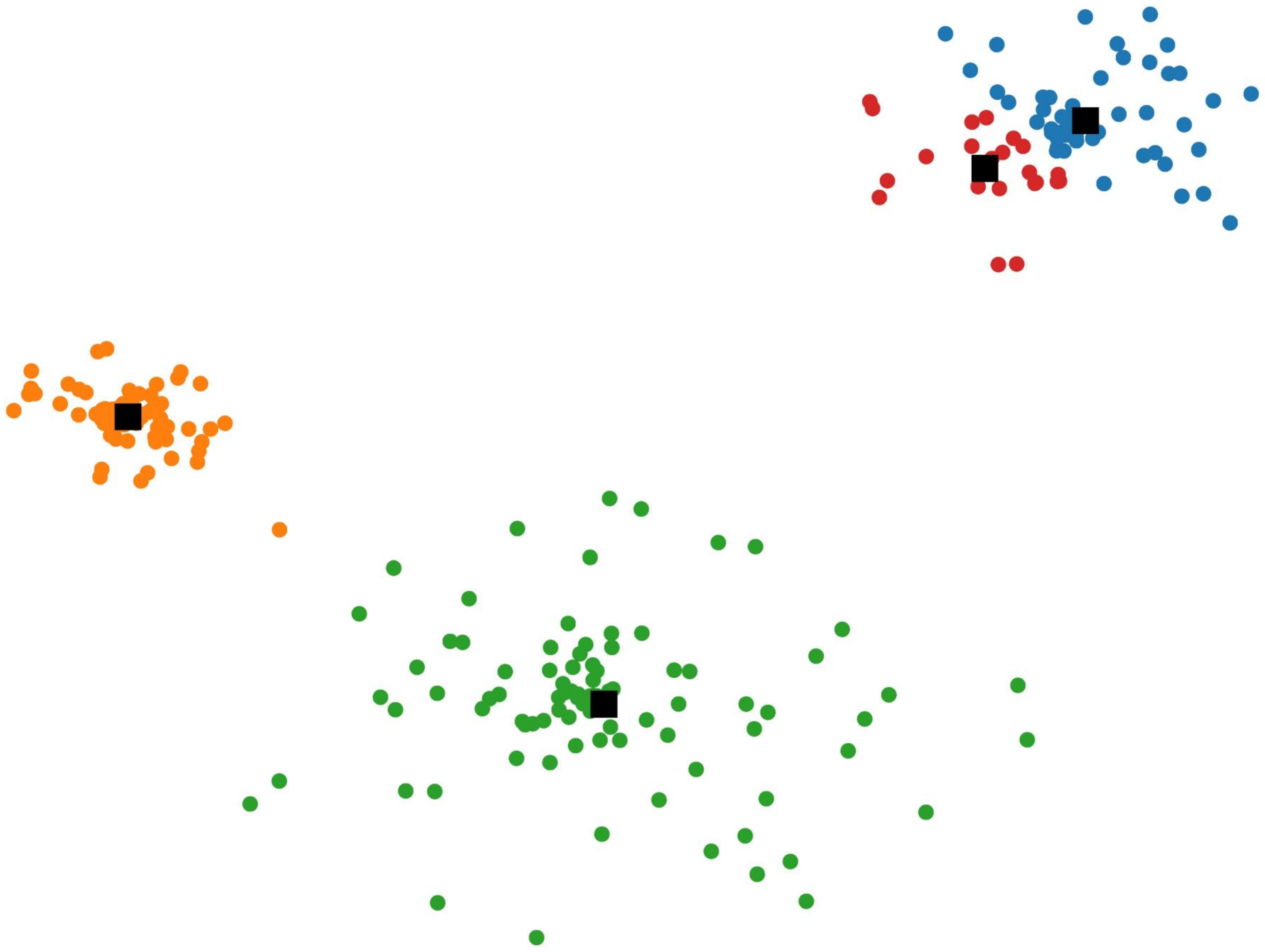
- Theorem $c = \text{average}(C)$

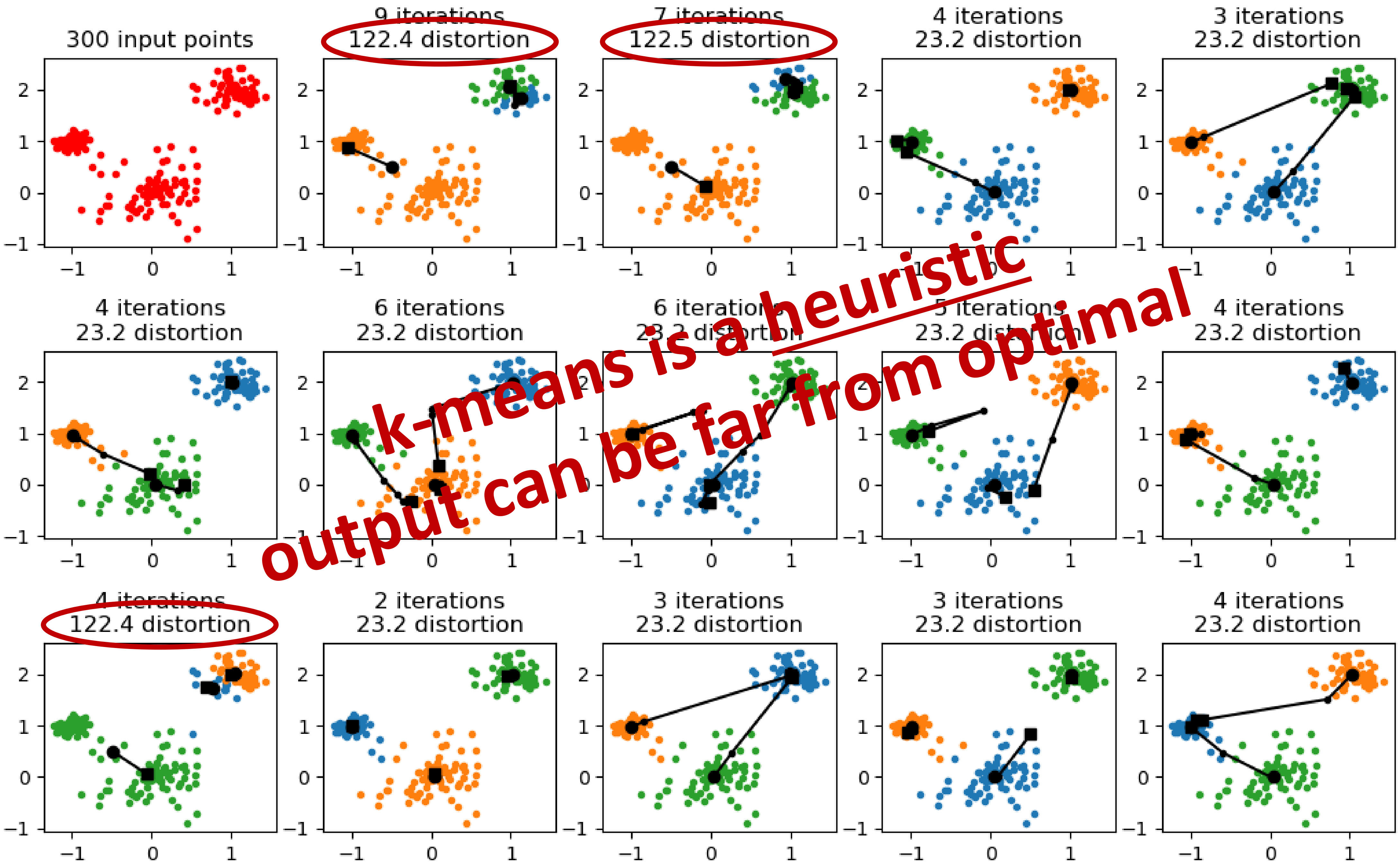


k-means - Lloyd's method (pseudo code)

```
centroids = k distinct random input points
while centroids change:
    create clusters C by assigning points to the nearest centroid
    centroids = average of each cluster
```







300 input points
9 iterations
122.4 distortion
7 iterations
122.5 distortion
4 iterations
23.2 distortion
3 iterations
23.2 distortion
4 iterations
23.2 distortion
6 iterations
23.2 distortion
6 iterations
23.2 distortion
5 iterations
23.2 distortion
4 iterations
23.2 distortion
4 iterations
122.4 distortion
2 iterations
23.2 distortion
3 iterations
23.2 distortion
3 iterations
23.2 distortion
4 iterations
23.2 distortion
k-means is a heuristic
output can be far from optimal

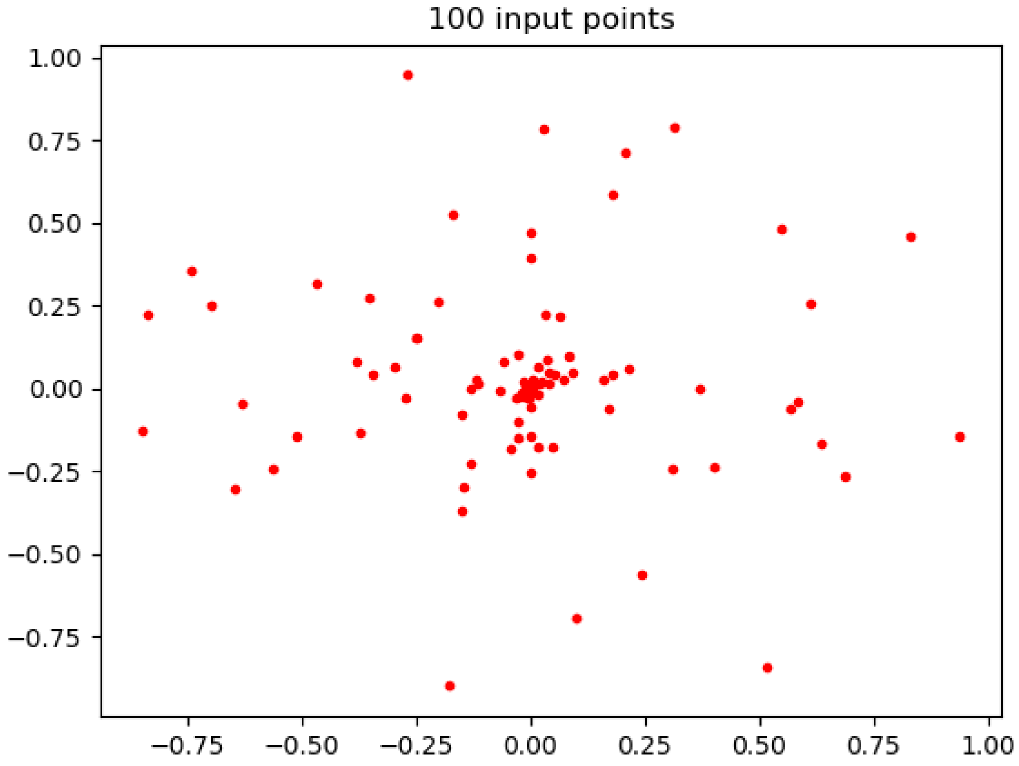
Generating random points (just one random approach)

k_means.py

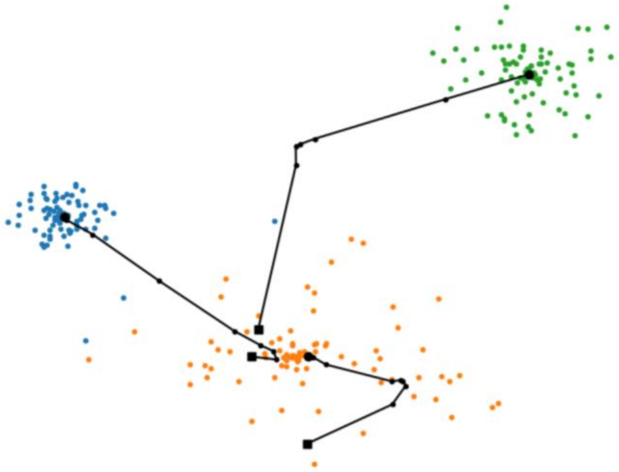
```
from random import random
from math import pi, cos, sin

def random_point(x, y, radius):
    angle = 2 * pi * random()
    r = radius * random() ** 2
    return x + r * cos(angle), y + r * sin(angle)

def random_points(n, x, y, radius):
    for _ in range(n):
        yield random_point(x, y, radius)
```



k-means



k_means.py

```python
from random import sample
from numpy import argmin, mean

def k_means(points, k):
    centroid = sample(points, k)
    centroids = [ centroid ]  # history for visualization
    while True:
        clusters = [[] for _ in centroid]
        for p in points:
            i = argmin([dist(p, c) for c in centroid])
            clusters[i].append(p)
        centroid = [tuple(map(mean, zip(*c))) for c in clusters]
        if centroid == centroids[-1]:
            break
        centroids.append(centroid)
        if any(len(c) == 0 for c in clusters):
            print('Not good - empty cluster')
            break
    return clusters
```

k-mean limitations

- Can easily converge to a solution far from a global minimum
  - Solution – try several times and take the best
    (possibly since we can measure the quality (= distortion) of a solution)
- Clusters can become empty
  - Solution – discard and restart / take a random point out as a new centroid / take point furthest away from existing centroids / ....
- Sensitive to the scales of the different dimensions
  - Solution – apply some kind of initial normalization of coordinates

k-means - better bounds

- The k-means++ algorithm achieves an expected guarantee to be at most a factor $8(2 + \ln k)$ from the optimal [Vassilvitskii & Arthur]

- There exist polynomial time approximation schemes that find a solution that is guaranteed $1 + \varepsilon$ of the optimal (but running time exponential in k and dimension of points) [Har Peled et al.]

- **In practice: A heuristic is most often the algorithm of choice**

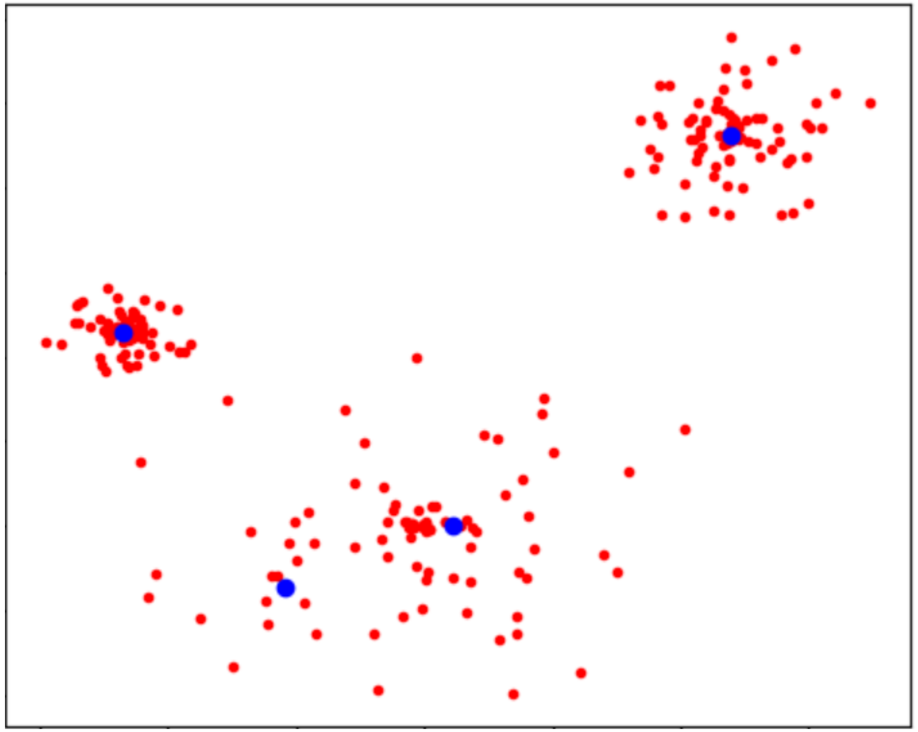
scipy.cluster.vq.kmeans

k_means.py

```
from scipy.cluster.vq import kmeans, whiten
import matplotlib.pyplot as plt

points = whiten(points)  # normalize variance of points
centroids, distortion = kmeans(points, K)

plt.plot(*zip(*points), 'r.')
plt.plot(*zip(*centroids), 'bo')
plt.title('scipy.cluster.vq.kmeans')
plt.show()
```



Note: According to the documentation "`whiten` must be called prior to passing an observation matrix to `kmeans`"

docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.vq.kmeans.html

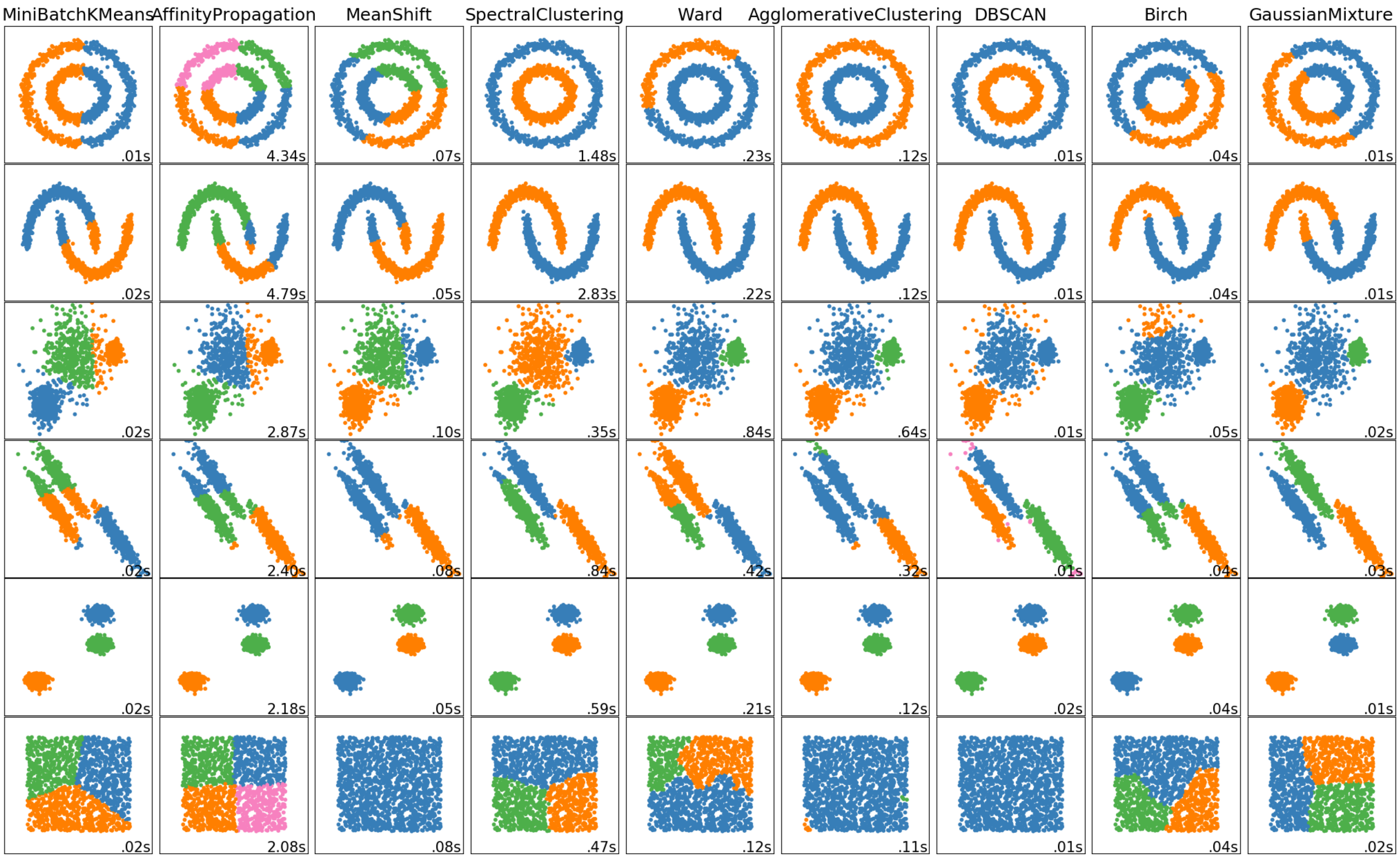
scipy.cluster.vq.whiten

- Normalizes / scales each dimension to have unit variance 1.0

$$\mathrm{Var}(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 \qquad \mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.vq.kmeans.html

Other Python clustering methods - `sklearn.cluster`



http://scikit-learn.org/stable/modules/clustering.html

DBSCAN*

dbscan.py

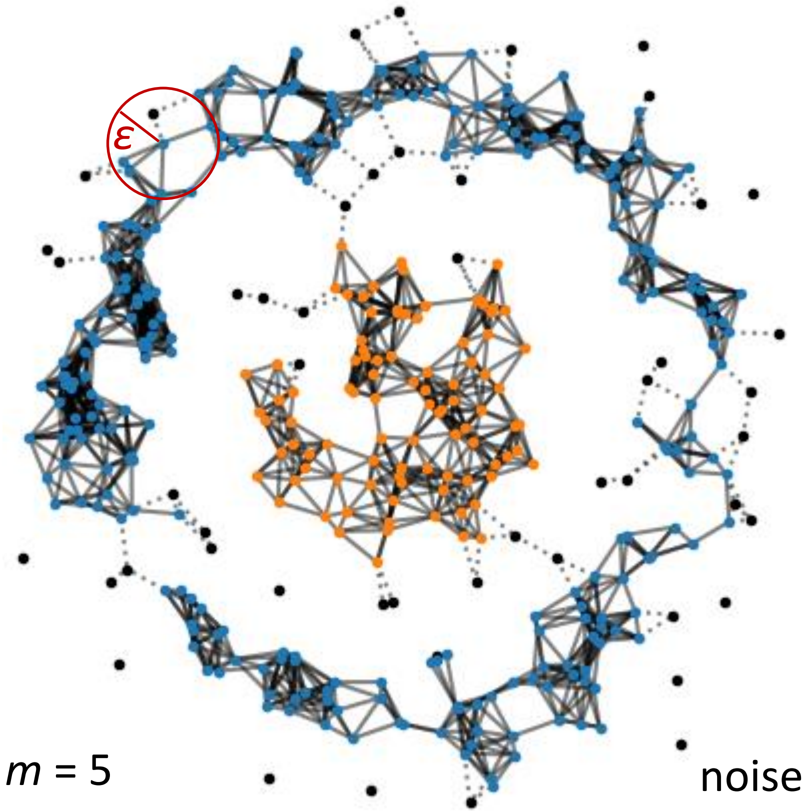
```
def dbscan(points, epsilon, m):
    def dist(p, q):
        return sum((pi - qi) ** 2 for pi, qi in zip(p, q))

    def close(p, q):
        return dist(p, q) <= epsilon ** 2

    core, noise, clusters = [], [], []
    for p in points:
        if sum(close(p, q) for q in points) >= m:
            core.append(p)
        else:
            noise.append(p)

    while core:
        cluster = [core.pop()]
        for p in cluster:
            for q in list(core):
                if close(p, q):
                    cluster.append(q)
                    core.remove(q)
        clusters.append(cluster)
    return clusters, noise
```

- Parameters ε and m
- p is a core point when
  $$|\{q \mid |p - q| \leq \varepsilon\}| \geq m$$
- Remaining points are noise
- Core points p and q are in the same cluster if $|p - q| \leq \varepsilon$

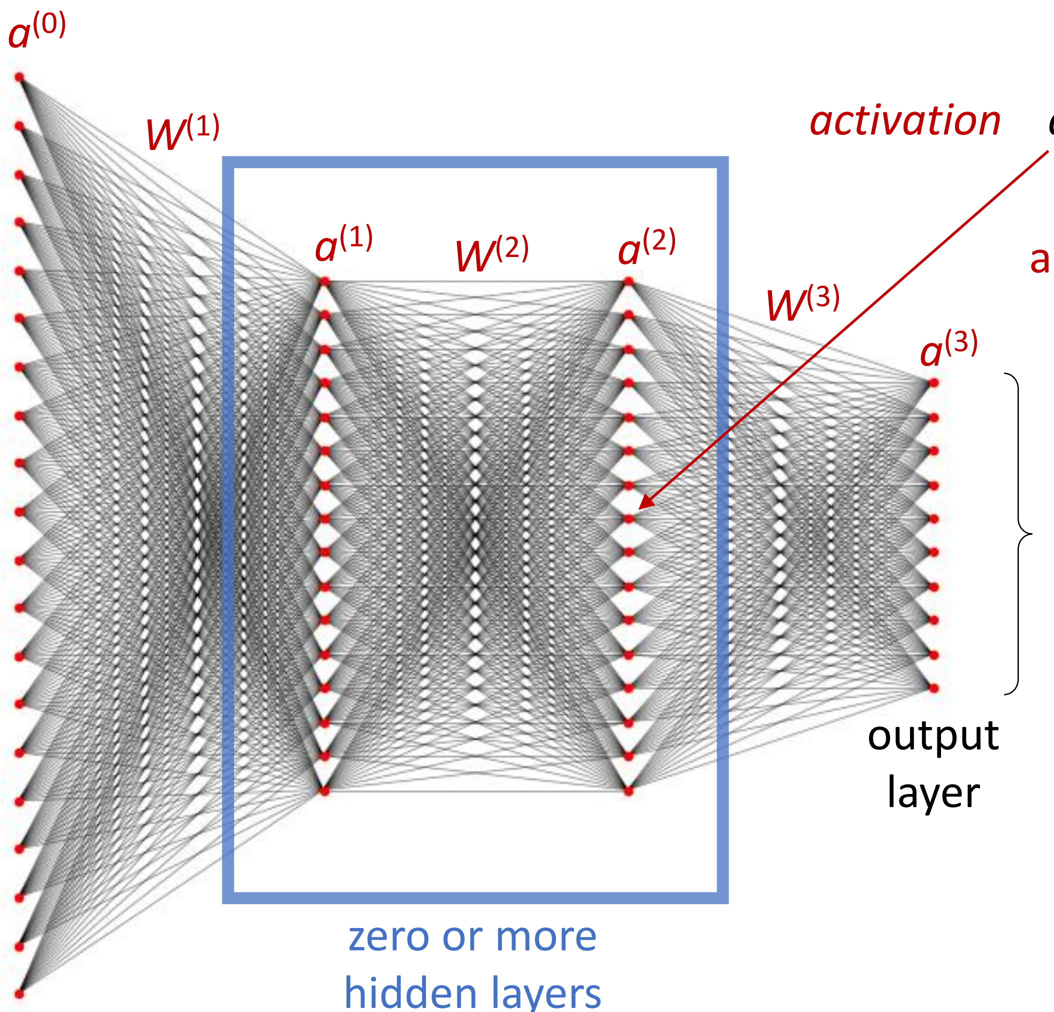


Data Mining Algorithms

- k-means, DBSCAN*, and more generally clustering, is just one field in the area of *Data Mining*
- For more information see the webpage
  Top 10 Data Mining Algorithms, Explained
  a follow up to the below paper
- X. Wu et al., *Top 10 algorithms in data mining*,
  Knowledge and Information Systems, 14(1):1–37, 2008.
  DOI 10.1007/s10115-007-0114-2

www.kdnuggets.com/2015/05/top-10-data-mining-algorithms-explained.html

Neural networks (one slide introduction)



input layer

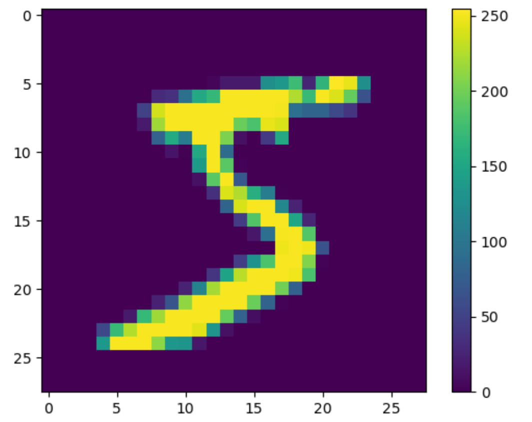
zero or more hidden layers (zero = linear classifier)

MNIST, 784 input pixels scaled to [0.0, 1.0]

activation $a_i^{(l)} = f^{(l)}\left(\sum_j a_j^{(l-1)} \cdot W_{ji}^{(l)} + b_i^{(l)} \right)$

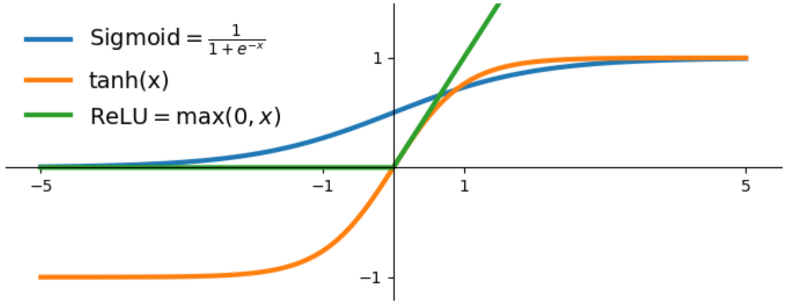
activation function (nonlinear) — weight — bias

Classification, like MNIST, prediction = index of node with maximum output



MNIST : 28 x 28 pixel values from [0, 255]

Common activation functions



e.g. mean squared error

$$\frac{1}{n}\sum_{(x,y)} |\mathrm{out}(x) - y|^2$$

Learning
Find *W*s and *b*s performing well (minimize a cost function) on a set of *n* training inputs *x* with known output *y* using *backpropagation / stochastic gradient descend*

Applying a linear classifier using Numpy: $x \cdot W + b$

Python shell

```
| import matplotlib.pyplot as plt
| import numpy as np
| from tensorflow import keras
| (train_images, train_labels), (test_images, test_labels) = keras.datasets.mnist.load_data()
| type(test_images)
> <class 'numpy.ndarray'>
| test_images.shape
> (10000, 28, 28)  # 10_000 images 28 x 28
| test_labels.shape
> (10000,)         # 10_000 labels
| test_labels[:3]
> array([7, 2, 1], dtype=uint8)
| for i, image in zip(range(3), test_images):
|     plt.subplot(1, 3, i + 1)
|     plt.imshow(image)
| plt.show()

| W, b = map(np.array, eval(open('mnist_linear.weights').read()))  # read W and b from file
| print(W.shape, W.dtype, b.shape, b.dtype)
> (784, 10) float64 (10,) float64
| print([np.argmax(image.reshape(28 * 28) @ W + b) for image in test_images[:3]])
> [7, 2, 1]  # correct on 9_142 of the 10_000 images for the above file, ie accuracy 91%
```

manually generated labels

network prediction