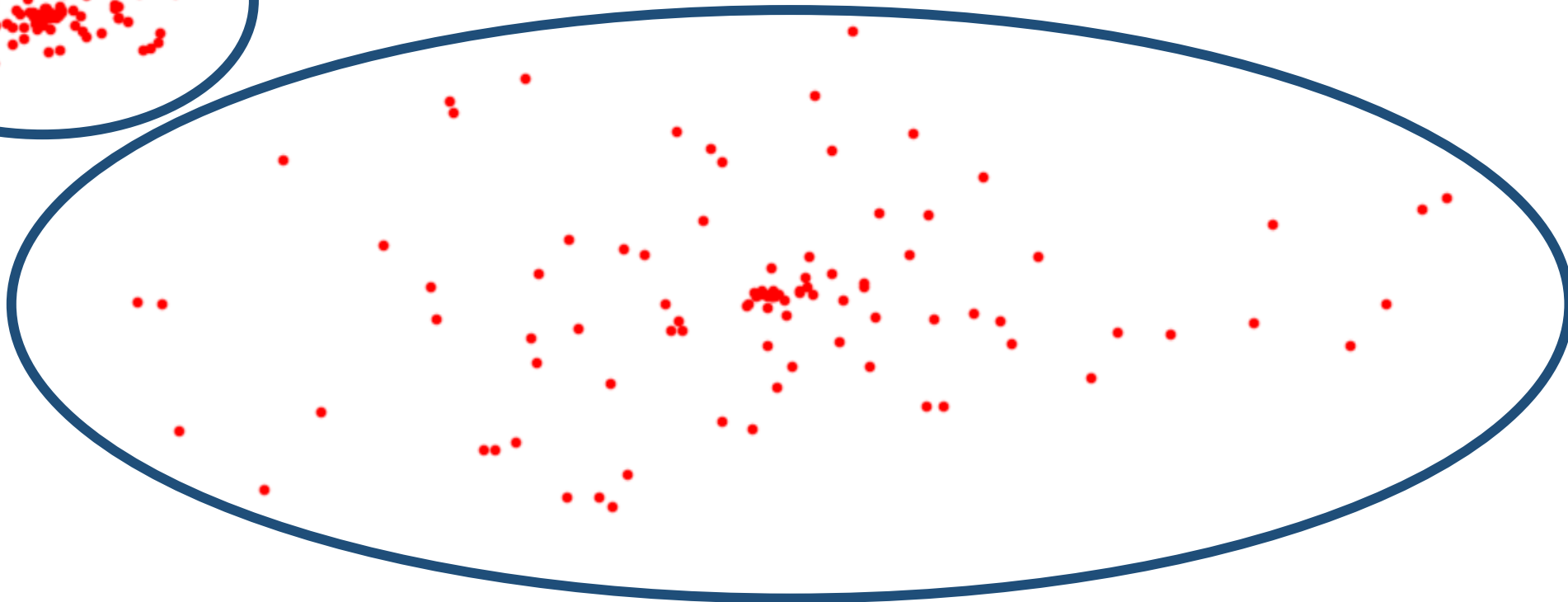
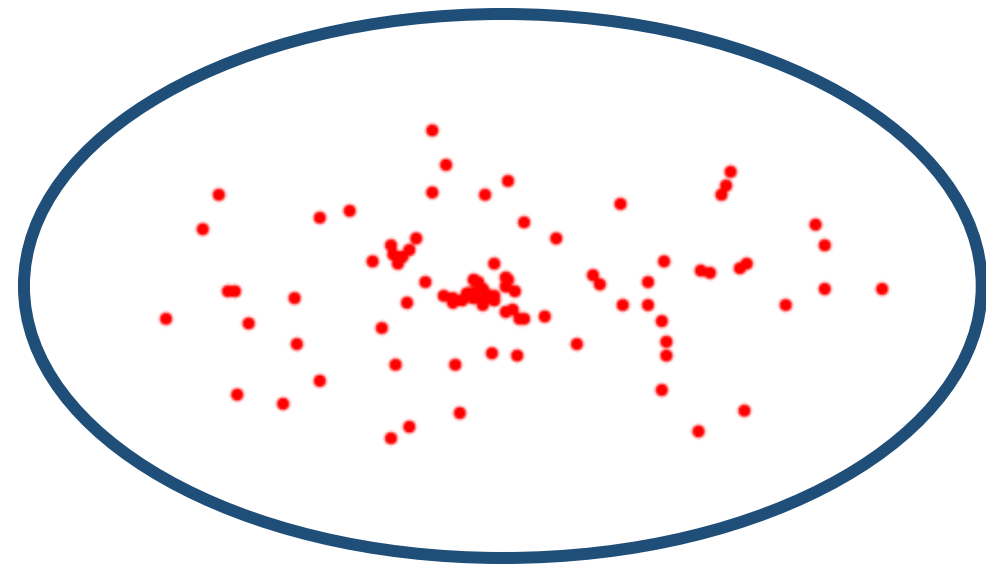
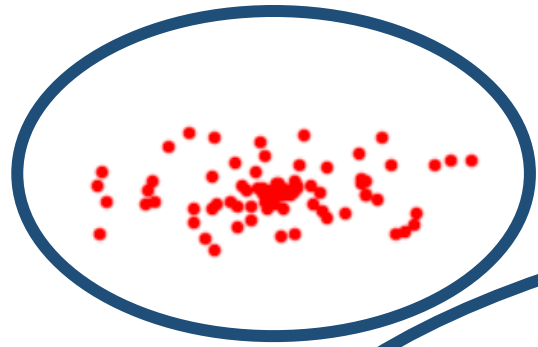


Clustering

- k-means
- `scipy.cluster.vq.kmeans`
- neural networks

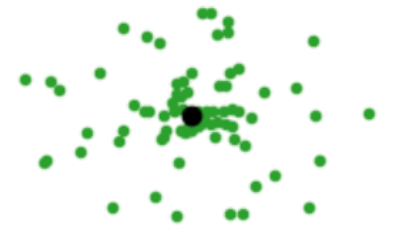
3 clusters / groups of points



Clustering = Optimization problem

Example: k-means

- Find k points *centroids*
- Assign each input point to nearest centroid $\rightarrow k$ 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

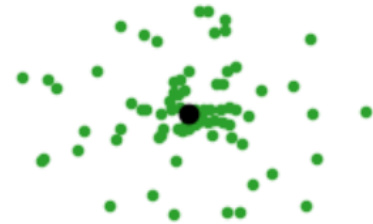


k-means for $k = 1$

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

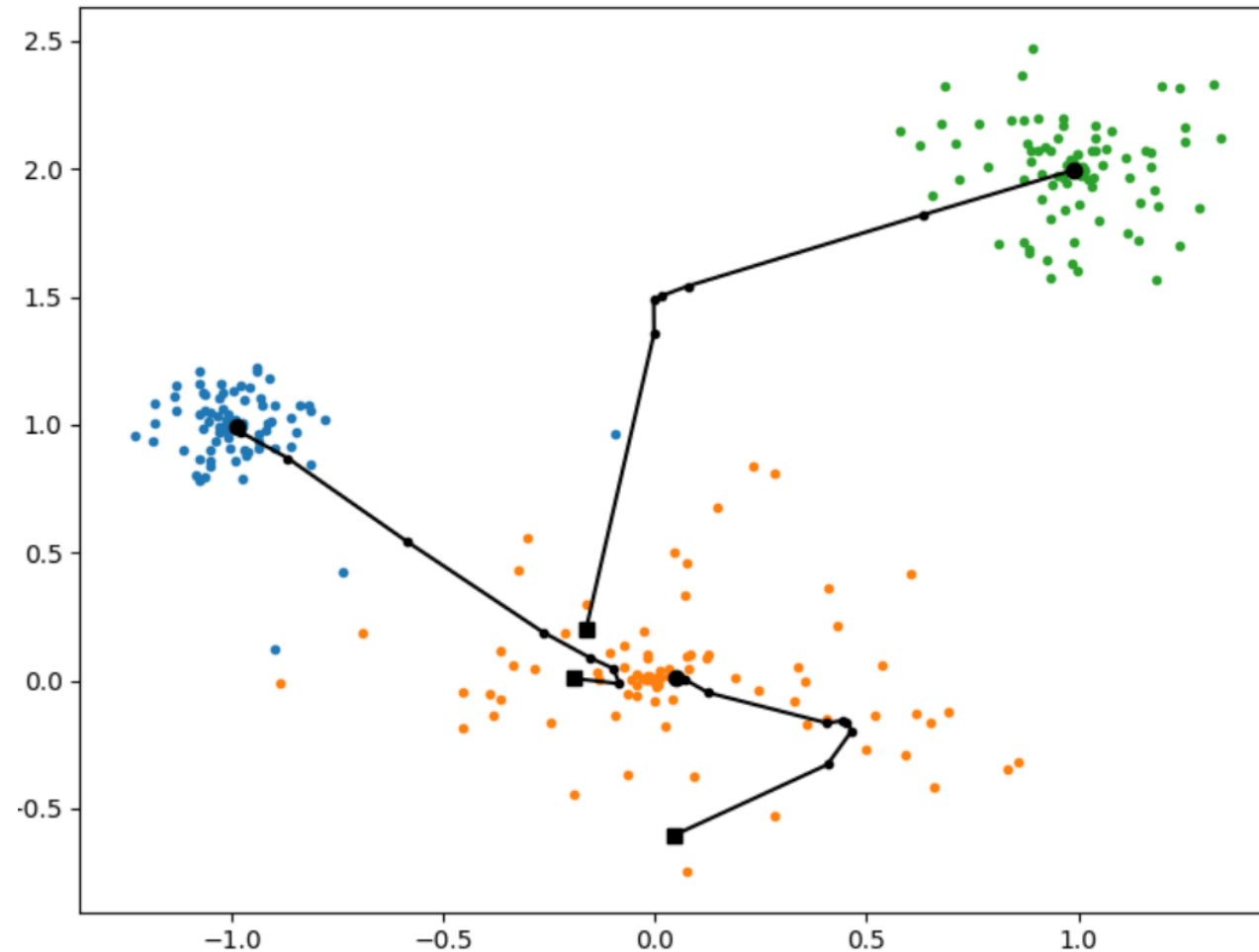
$$\text{distortion} = \sum_{p \in C} |p - c|^2$$

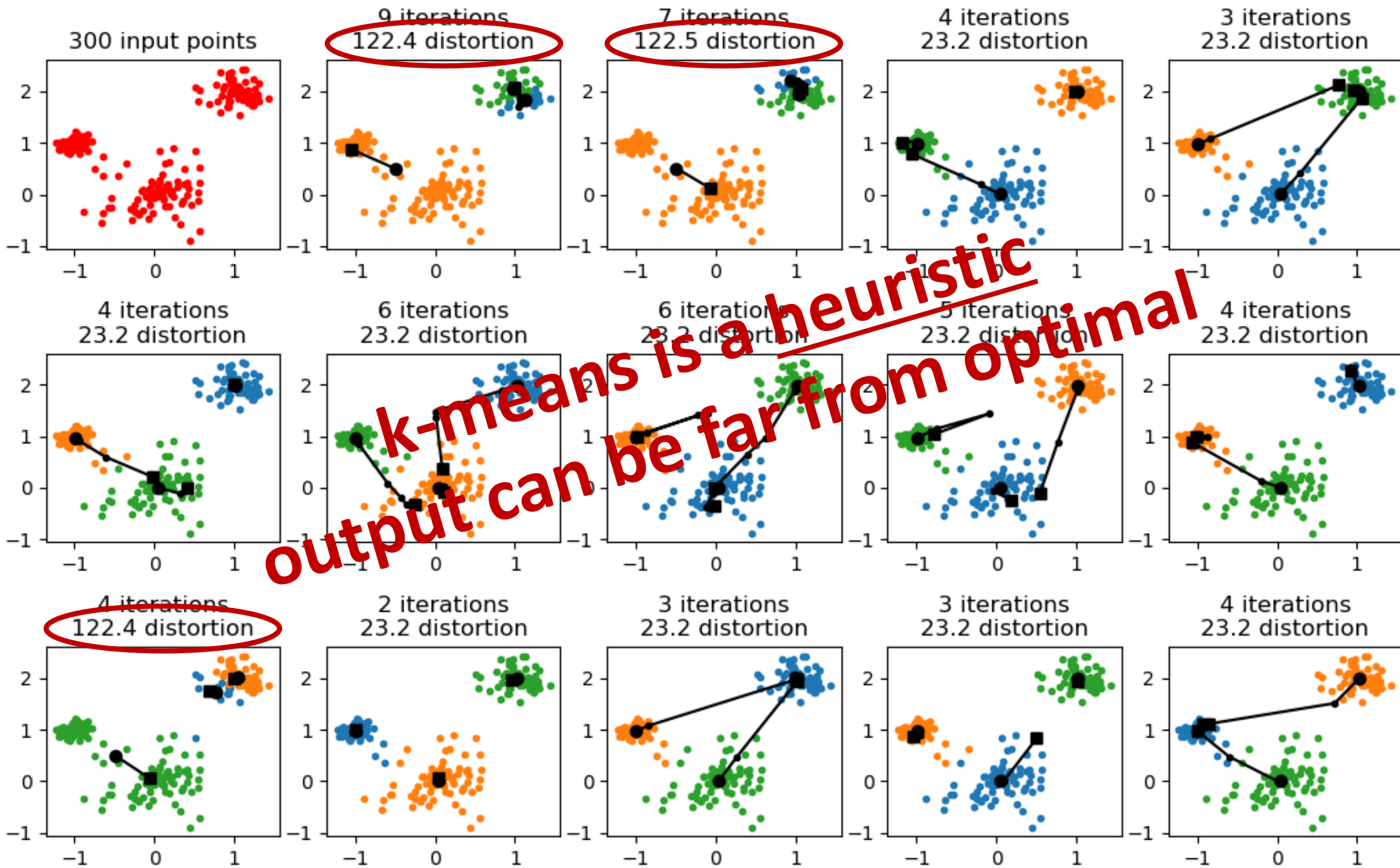
- Theorem $c = \text{average}(C)$



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
```





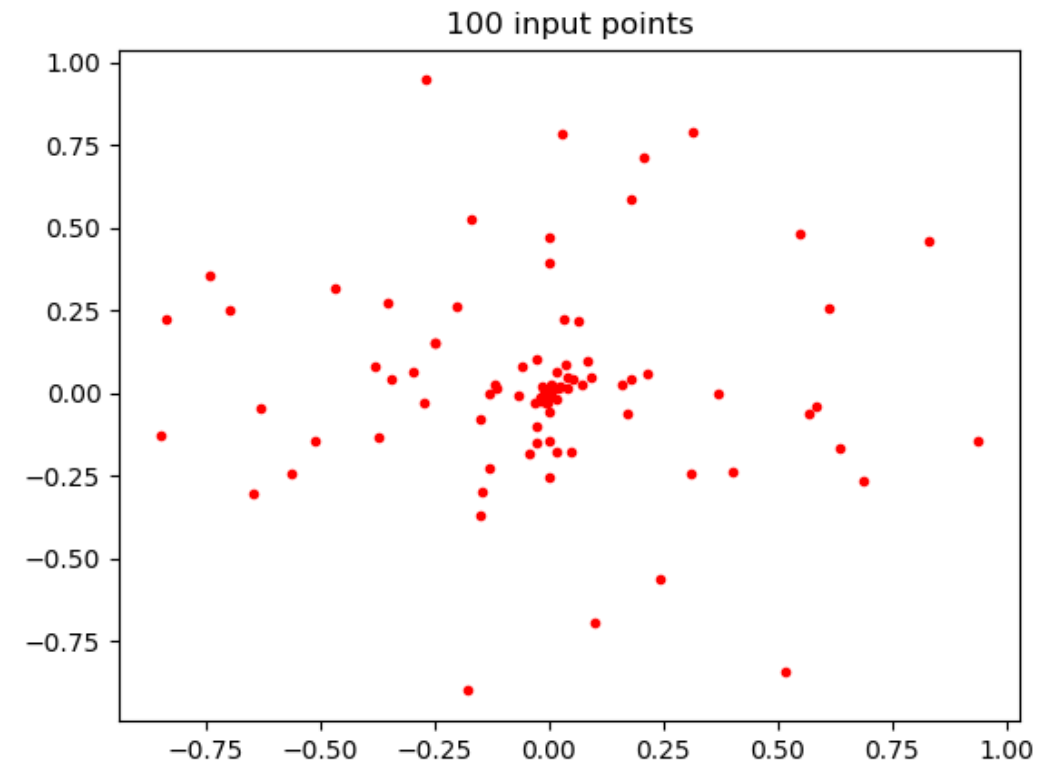
Generating random points (just one random approach)

`k_means.py`

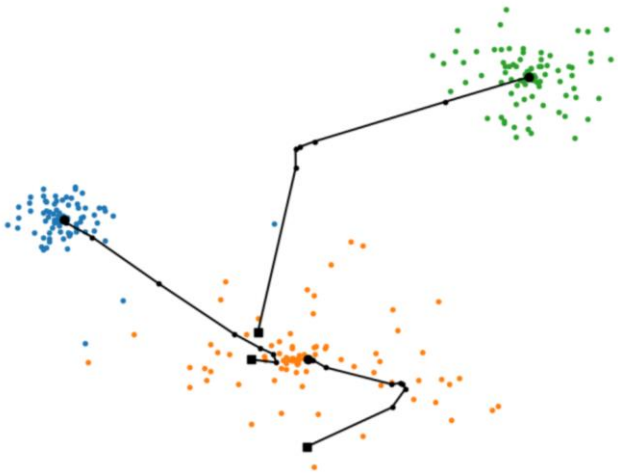
```
from random import random
from math import pi, cos, sin

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`

```
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 min(len(c) for c in clusters) == 0:
            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 $8(\ln k + 2)$ from the optimal [Vassilvitskii & Arthur]
- There exist **polynomial time approximation schemes** that find a solution that is guaranteed $1 + \epsilon$ 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`

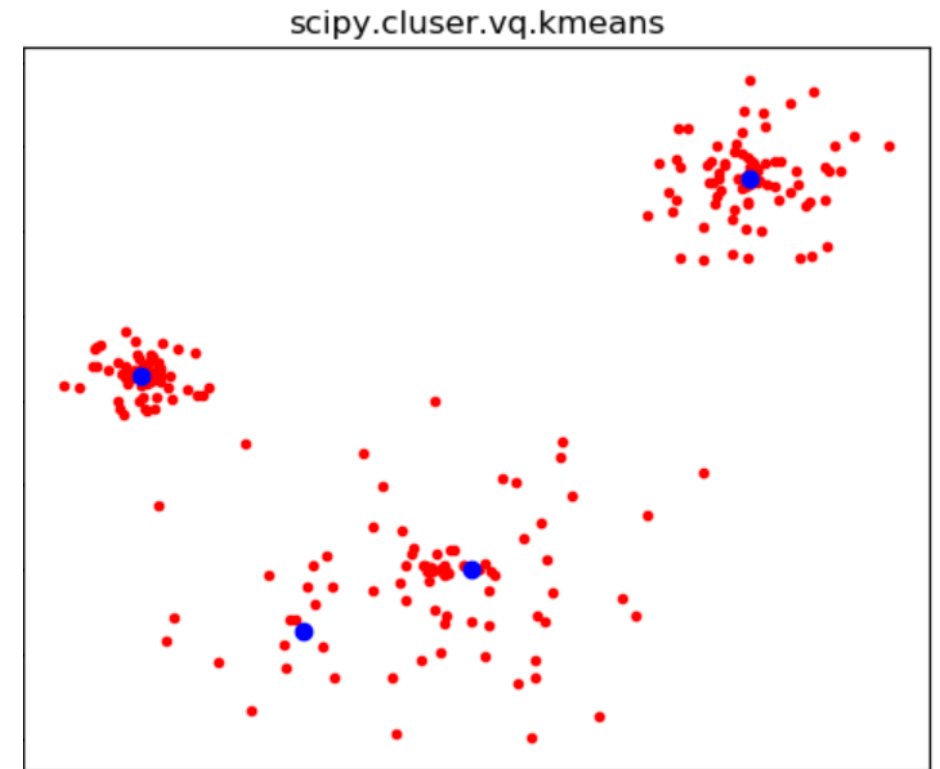
```
from scipy.cluster.vq import kmeans, whiten
import matplotlib.pyplot as plt

points = whiten(points) # normalize variance of points

plt.plot(*zip(*points), 'r.')
plt.plot(*zip(*kmeans(points, K)[0]), "bo")
plt.title("scipy.cluster.vq.kmeans")
```

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

kmeans returns tuple (centroids, distortion)



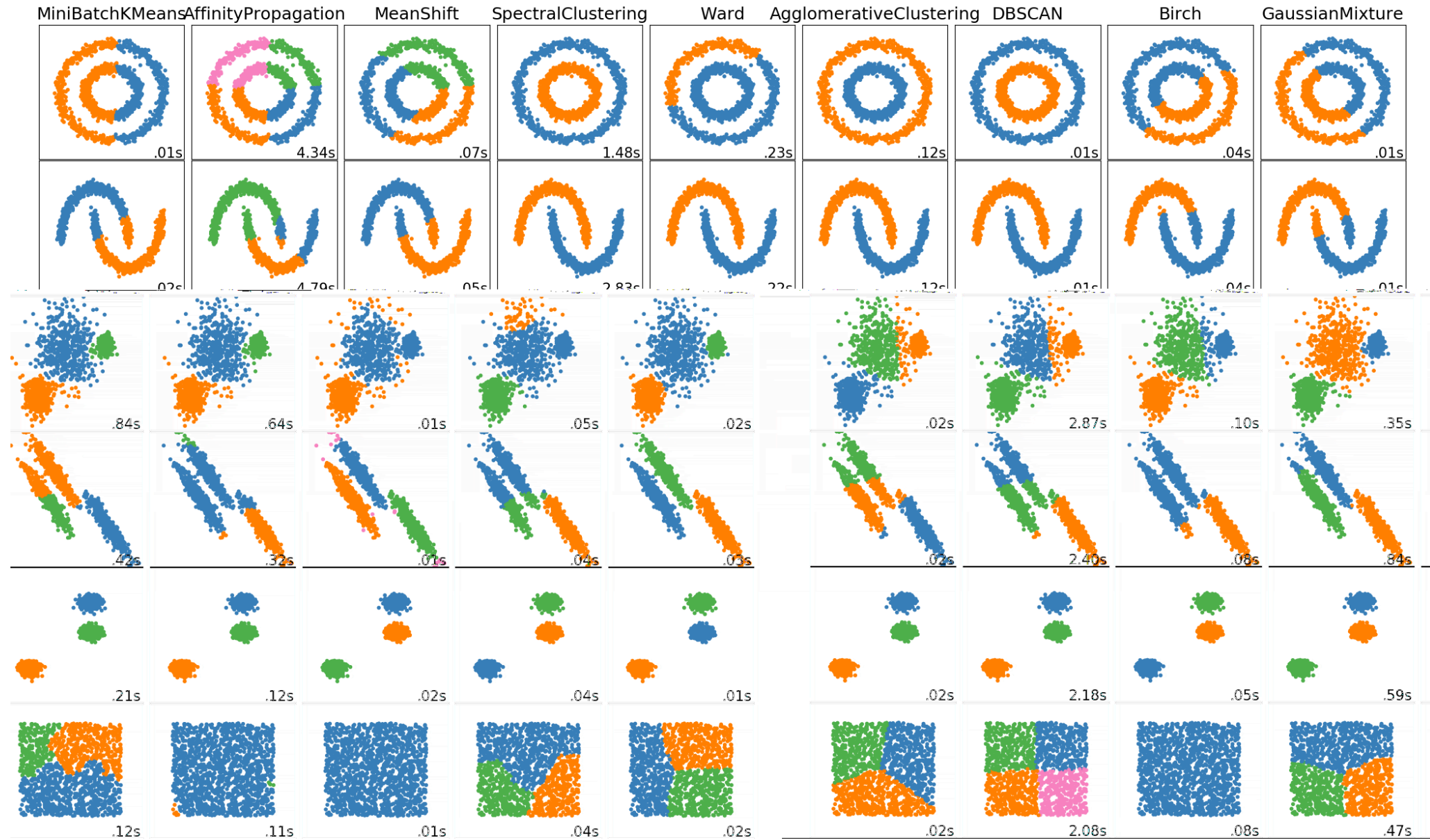
scipy.cluster.vq.whiten

- Normalizes / scales each dimension to have unit variance 1.0

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

$$\mu = \frac{1}{n} \sum_{i=1}^n x_i$$

Other Python clustering methods - `sklearn.cluster`



DBSCAN*

dbscan.py

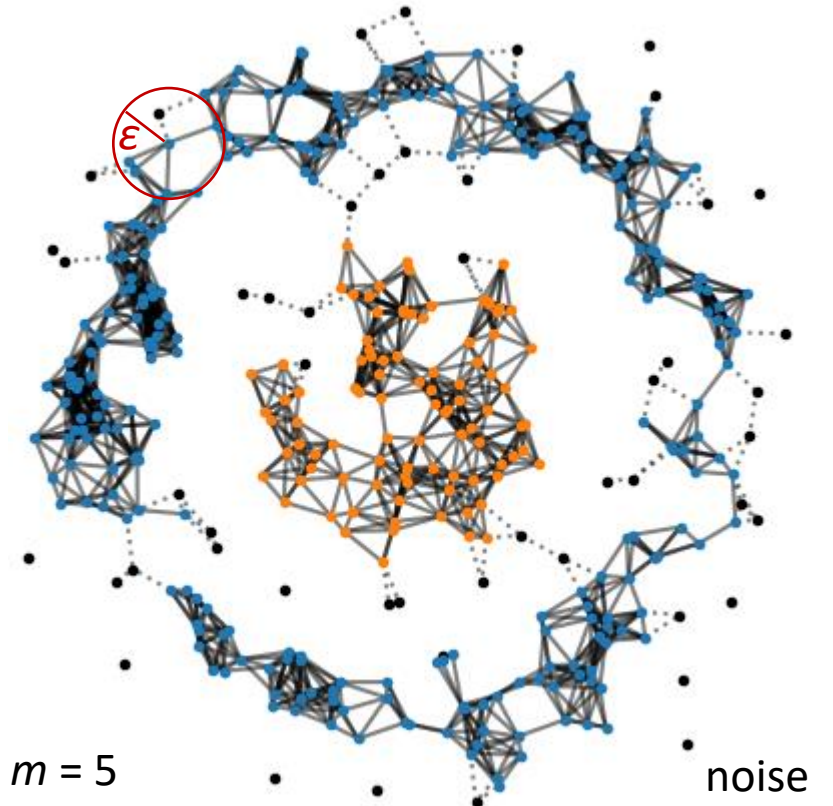
```
def dbscan(points, epsilon, m):
    def dist(p, q):
        return sum((pi - qi) ** 2 for pi, qi in zip(p, q))

    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
```

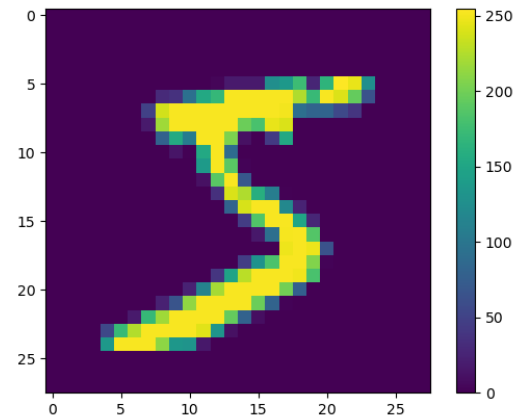
- Parameters ϵ and m
- p is a **core point** when
$$|\{q \mid |p - q| \leq \epsilon\}| \geq m$$
- Remaining points are **noise**
- Core points p and q are in the same **cluster** if $|p - q| \leq \epsilon$



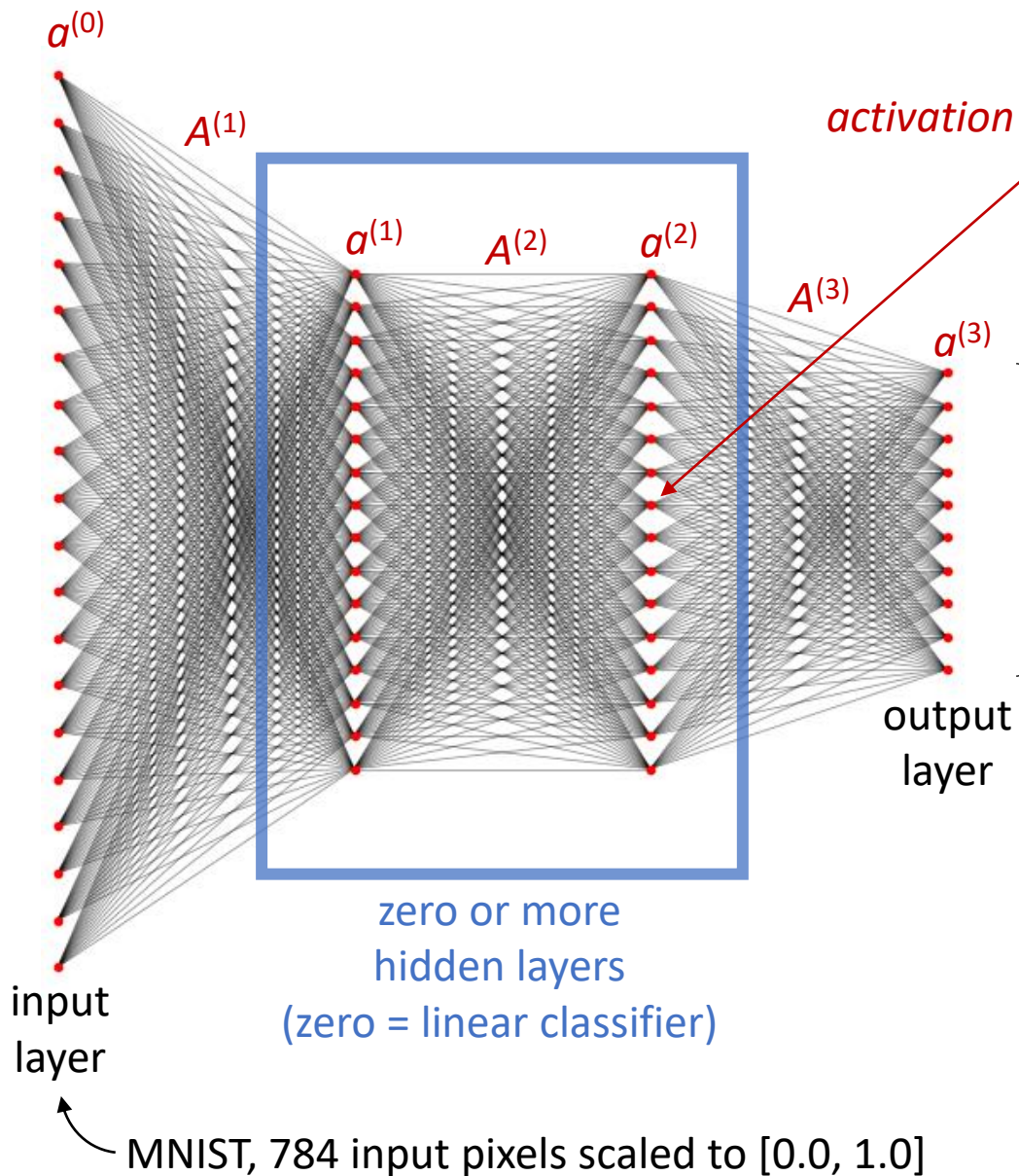
Data Mining Algorithms

- k-means, 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](#)

Neural networks (one slide introduction)



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

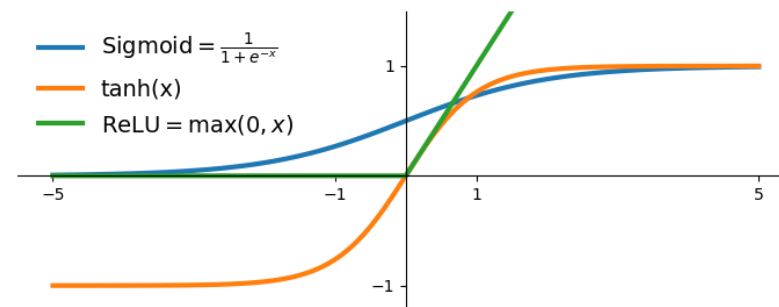


$$a_i^{(l)} = f^{(l)} \left(\sum_j a_j^{(l-1)} \cdot A_{ji}^{(l)} + b_i^{(l)} \right)$$

activation function weight bias

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

Common activation functions



e.g. mean squared error

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

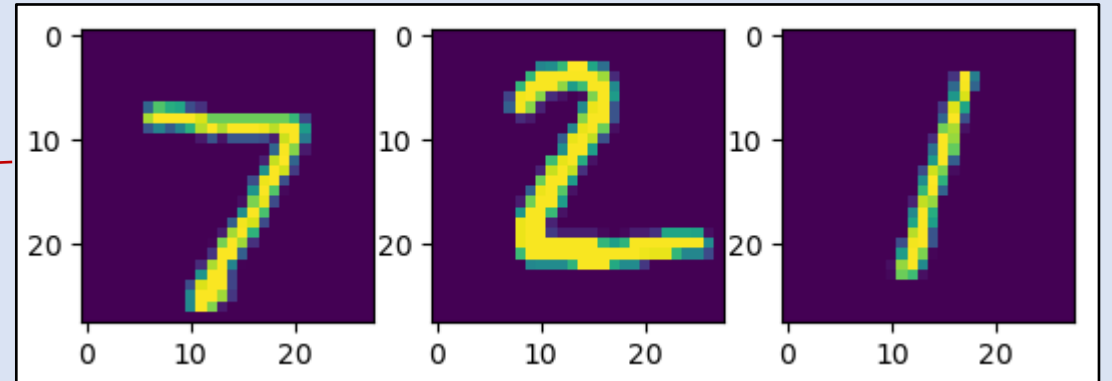
Learning

Find A 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 descent*

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

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) ← manually generated labels
| for i, image in zip(range(3), test_images):
|     plt.subplot(1, 3, i + 1)
|     plt.imshow(image)
| plt.show()
| A, b = map(np.array, eval(open('mnist_linear.weights').read())) # read A and b from file
| print(A.shape, A.dtype, b.shape, b.dtype)
> (784, 10) float64 (10,) float64
| print([np.argmax(image.reshape(28 * 28) @ A + b) for image in test_images[:3]])
> [7, 2, 1] ← network prediction # correct on 9_142 of the 10_000 images for the above file, ie accuracy 91%
```



THIS IS YOUR MACHINE LEARNING SYSTEM?

YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.

WHAT IF THE ANSWERS ARE WRONG?

JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.

