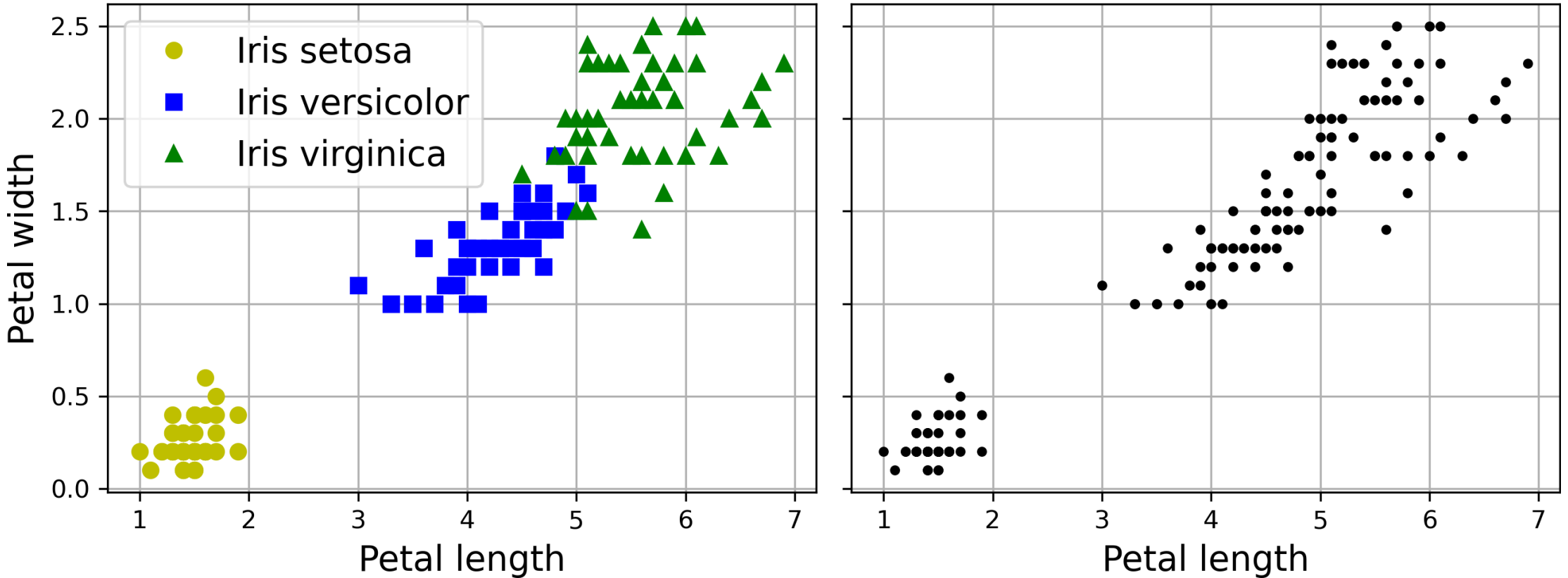
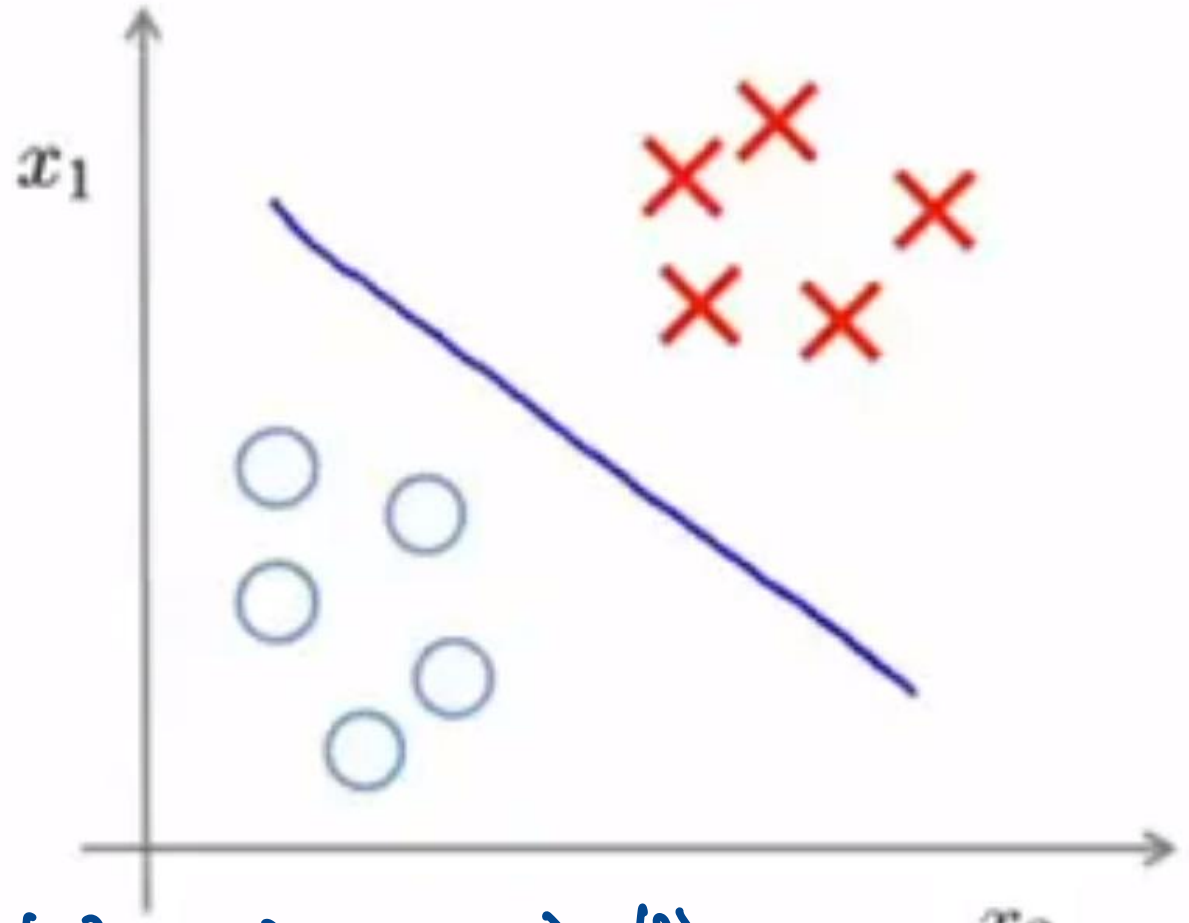
Unsupervised Learning

k-means Clustering

Introduction

- What if data is unlabelled ?
- Similar looking objects – clusters ?

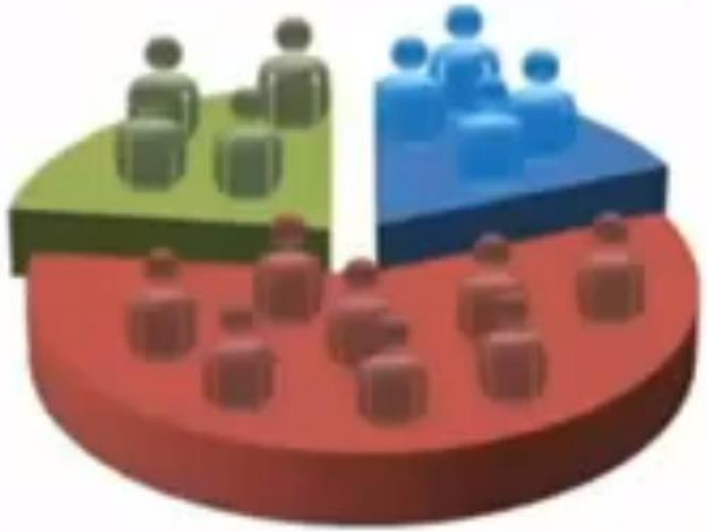




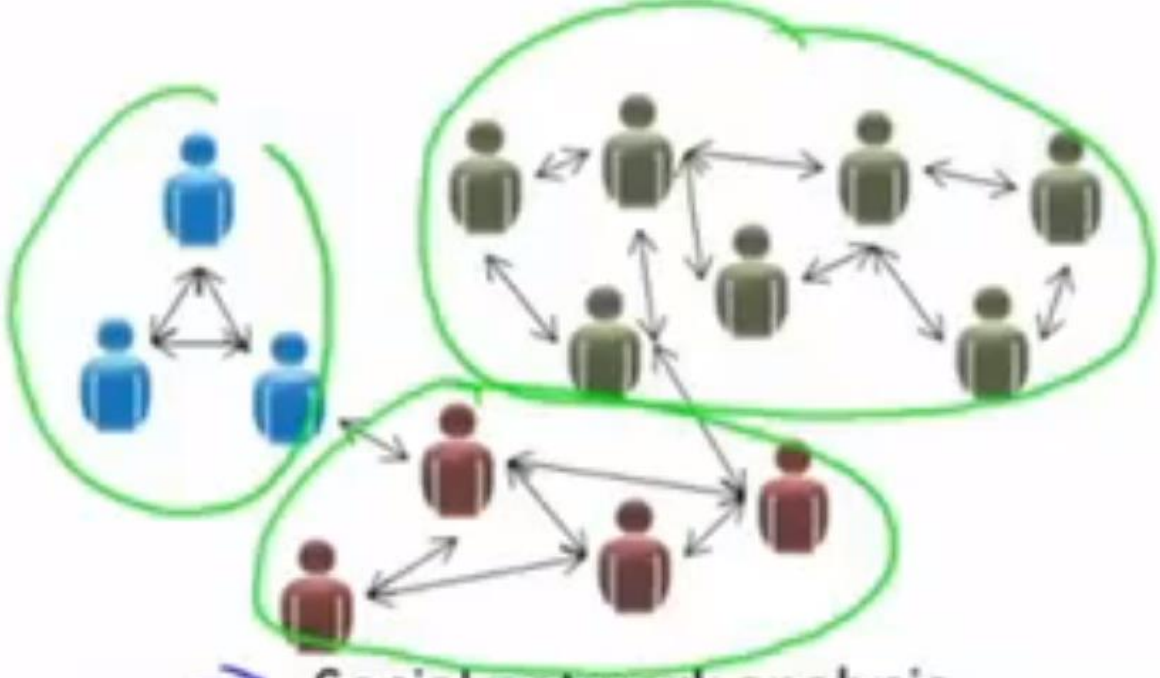
$(x_1^{(1)}, x_2^{(1)})$ $(x_1^{(2)}, x_2^{(2)})$ $(x_1^{(3)}, x_2^{(3)})$ ... (x_1^{m}, x_2^{m})

Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(3)}, y^{(3)}), \ldots, (x^{(m)}, y^{(m)})\}$.

Applications of clustering



→ Market segmentation



→ Social network analysis

Organize computing clusters

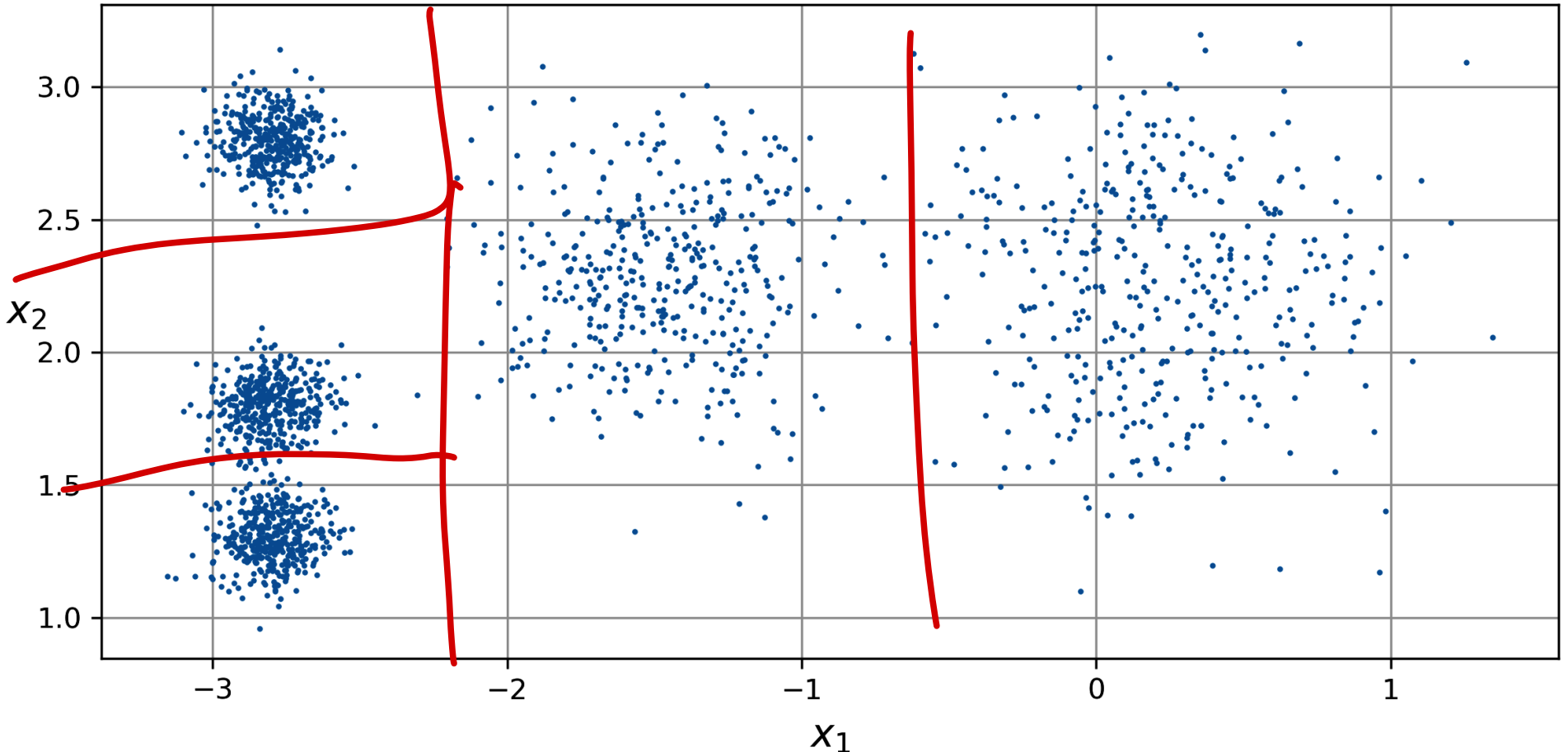
Astronomical data analysis

Applications

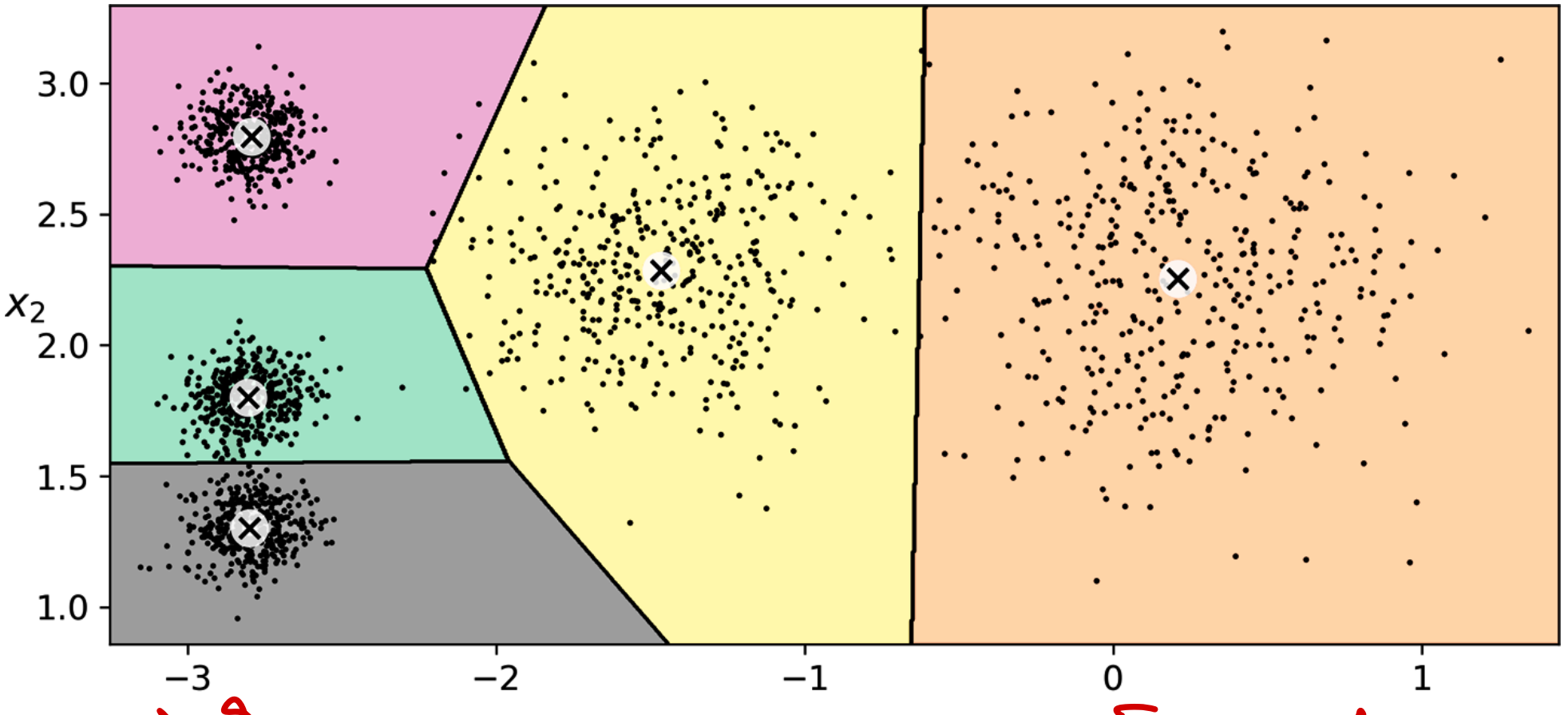
- Customer Segmentation
- Data Analysis
- Dimensionality Reduction
- Feature Engineering
- Anomaly Detection
- Semi-Supervised Learning
- Search Engines
- Image segmentation

K Means Clustering – Lloyd Forgy Algorithm

- An unlabelled dataset composed of five blobs of instances.



K Means Decision Boundaries – Voronoi Tessellation



Cluster centroid $(\bar{x}_1, \bar{x}_2)$

$\bar{x} = \frac{x_1 + x_2 + \cdots + x_m}{m}$

$\frac{y_1 + y_2 \cdots + y_m}{m}$

```
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

X, y = make_blobs([...])  # make the blobs: y contains the cluster IDs, but we
                          # will not use them; that's what we want to predict
k = 5
kmeans = KMeans(n_clusters=k, random_state=42)
y_pred = kmeans.fit_predict(X)
```

`KMeans` instance preserves the predicted labels of the instances it was trained on, available via the `labels_` instance variable:

```
>>> y_pred
array([4, 0, 1, ..., 2, 1, 0], dtype=int32)
>>> y_pred is kmeans.labels_
True
```

We can also take a look at the five centroids that the algorithm found:

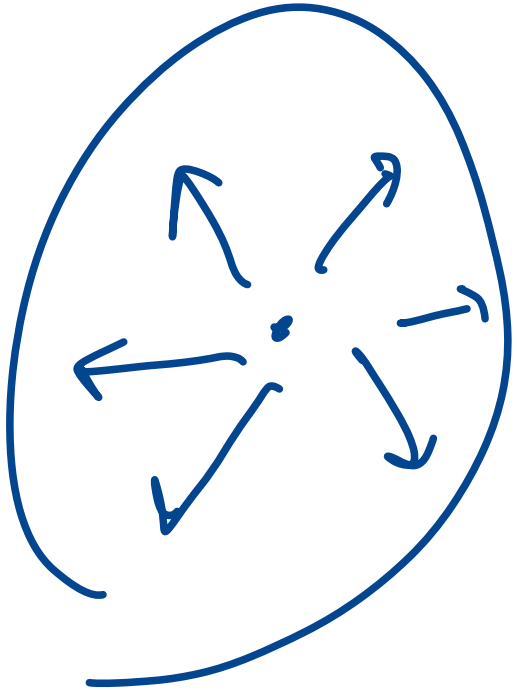
```
>>> kmeans.cluster_centers_
array([[-2.80389616,  1.80117999],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80037642,  1.30082566]])
```

You can easily assign new instances to the cluster whose centroid is closest:

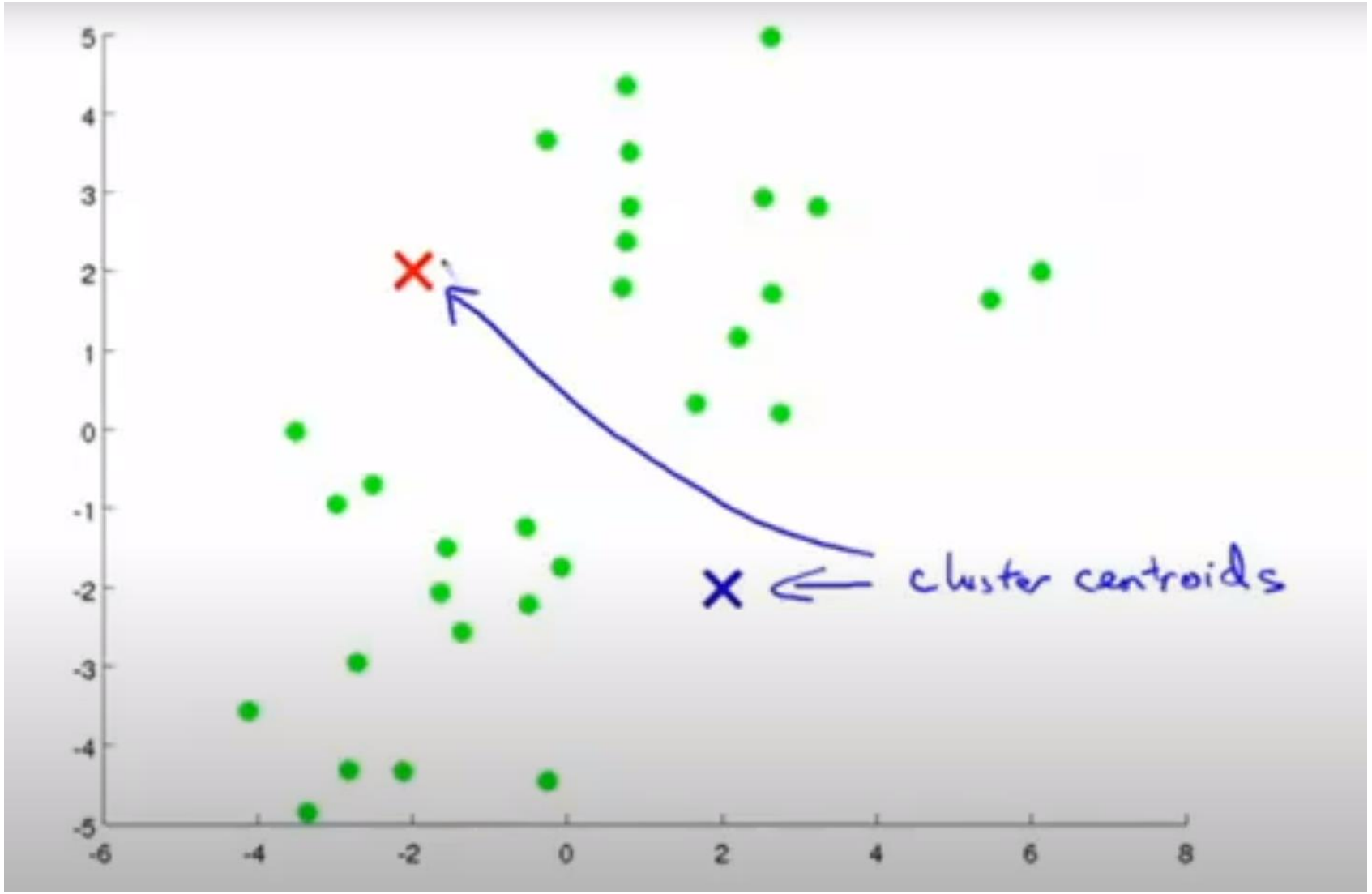
```
>>> import numpy as np
>>> X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
>>> kmeans.predict(X_new)
array([1, 1, 2, 2], dtype=int32)
```

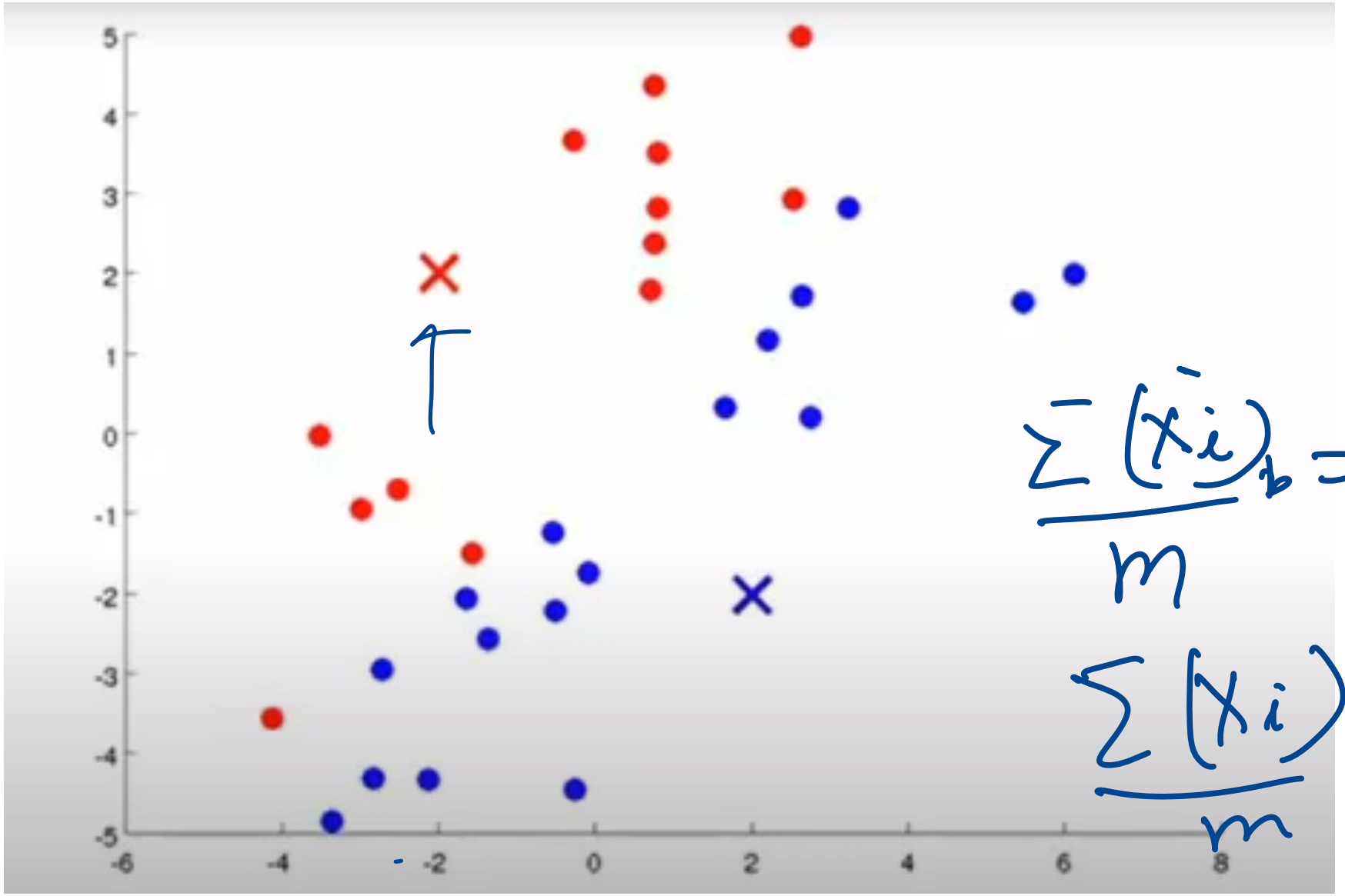
centroid $(\bar{x}_1, \bar{x}_2)$

- Hard clustering
- Soft Clustering



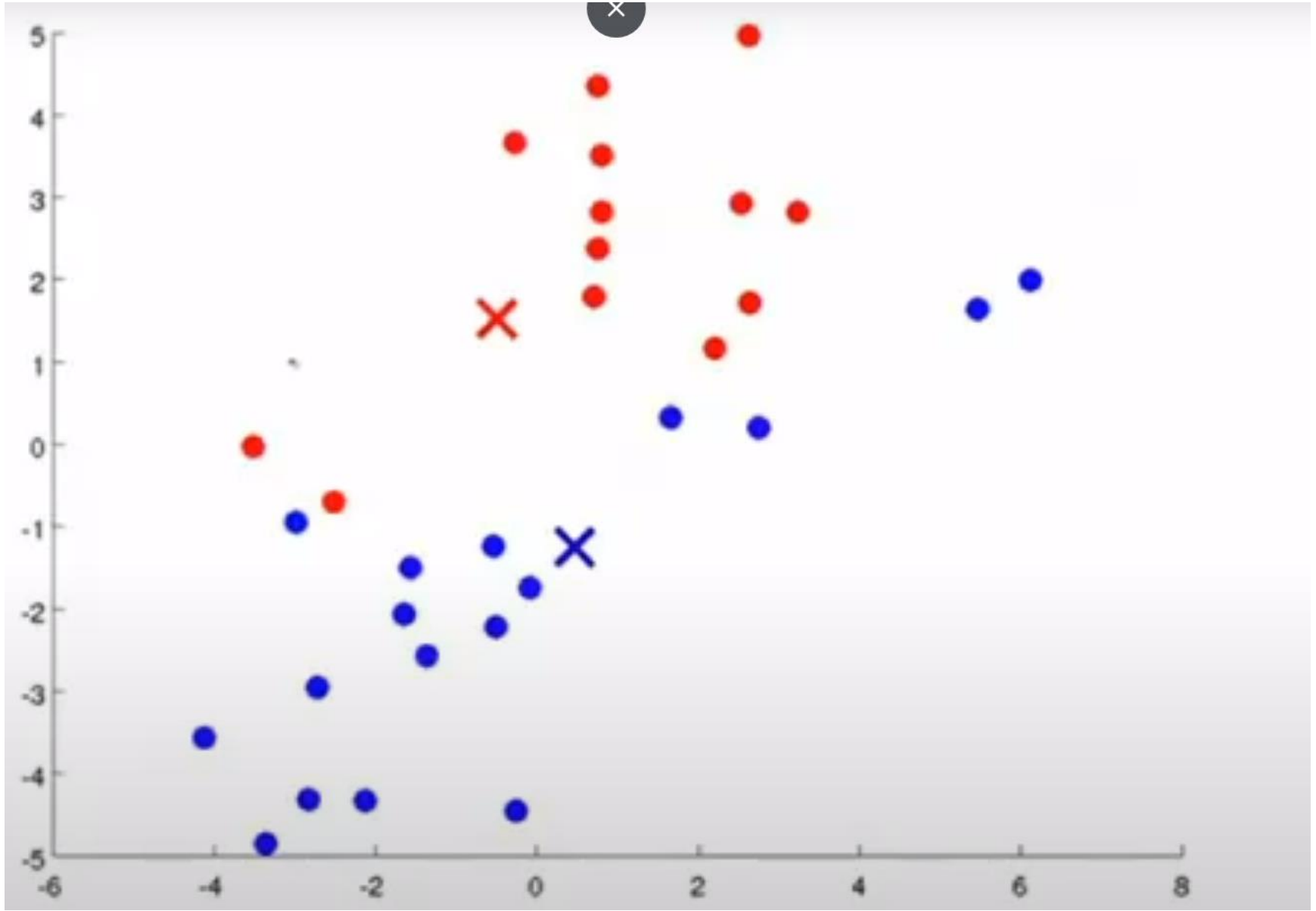
```
>>> kmeans.transform(X_new).round(2)
array([[2.81, 0.33, 2.9 , 1.49, 2.89],
       [5.81, 2.8 , 5.85, 4.48, 5.84],
       [1.21, 3.29, 0.29, 1.69, 1.71],
       [0.73, 3.22, 0.36, 1.55, 1.22]])
```

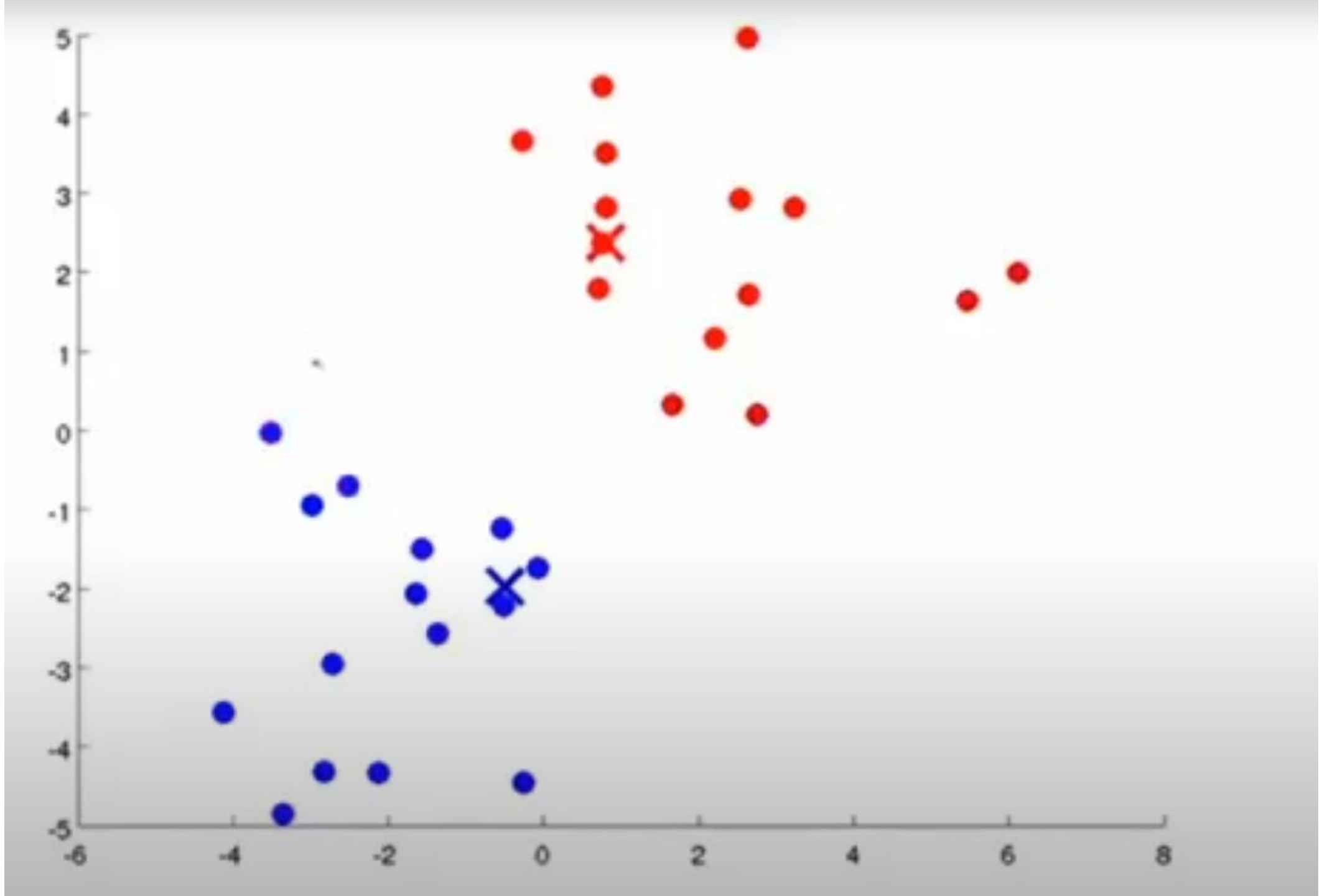

5
4
3
2
1
0
-1
-2
-3
-4
-5
-6
-4
-2
0
2
4
6
8
cluster centroids

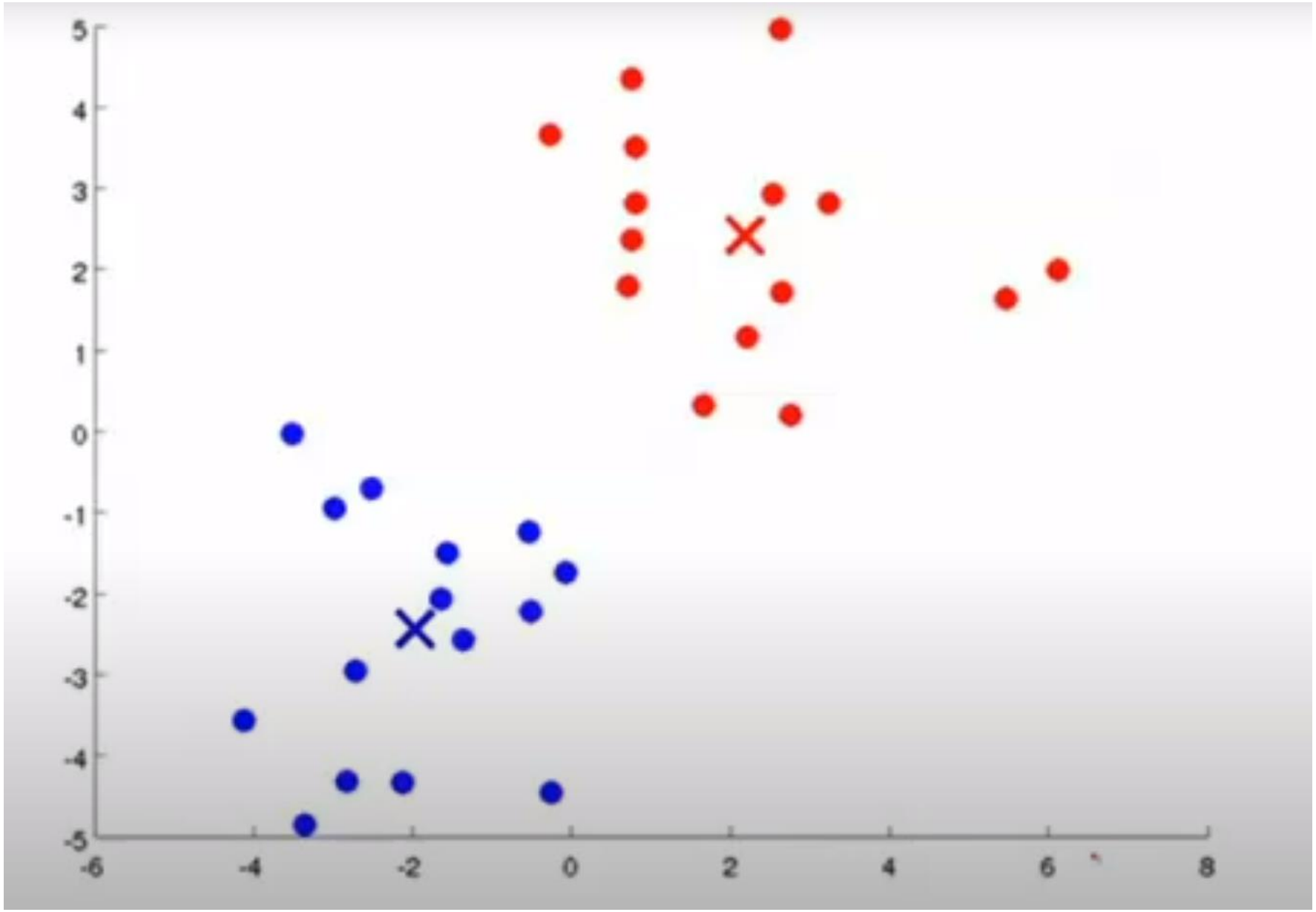


$$\frac{\sum (x_i)_b}{m} = \bar{x}_{blue}^{new}$$

$$\frac{\sum (x_i)_r}{m} = \bar{x}_{red}^{new}$$


5
4
3
2
1
0
-1
-2
-3
-4
-5
-6
-4
-2
0
2
4
6
8


5
4
3
2
1
0
-1
-2
-3
-4
-5
-6
-4
-2
0
2
4
6
8


5
4
3
2
1
0
-1
-2
-3
-4
-5
-6
-4
-2
0
2
4
6
8

Pseudo Code

1) Randomly initialise centroids => 2, 3 ... n
2) Assign points closest to respective centroid
3) move centroid based on new clusters
4) Repeat step (2) & (3)

Pseudo Code

K (no. of clusters)

Training set $(x^1, x^2 \dots x^m)$ $\quad x^i \in \mathbb{R}^n$

Randomly initialize K cluster centroids

$\mu_1 \; \mu_2 \dots \mu_K \in \mathbb{R}^n$

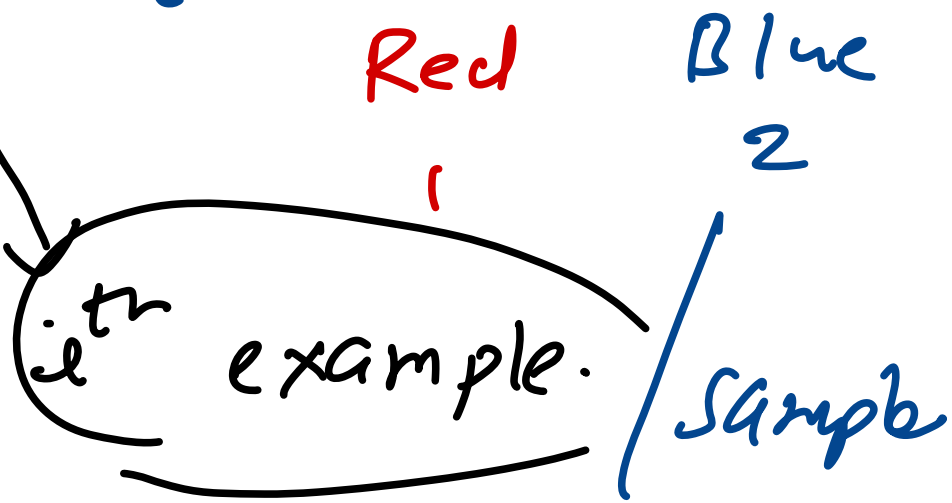
Pseudo Code

Step 2 cluster assignment.

for $i = 1$ to m

$c^{(i)}$ = index from (1 to K) of cluster centroids

$$\min_k \|x^{(i)} - \mu_k\|^2$$



K => total centroids.

k => index of centroid

Pseudo Code

(3) move centroid for $k = 1$ to K

$\mu_k \Rightarrow$ average of points assigned to that cluster (k)

$x^2\ x^3\ x^4$

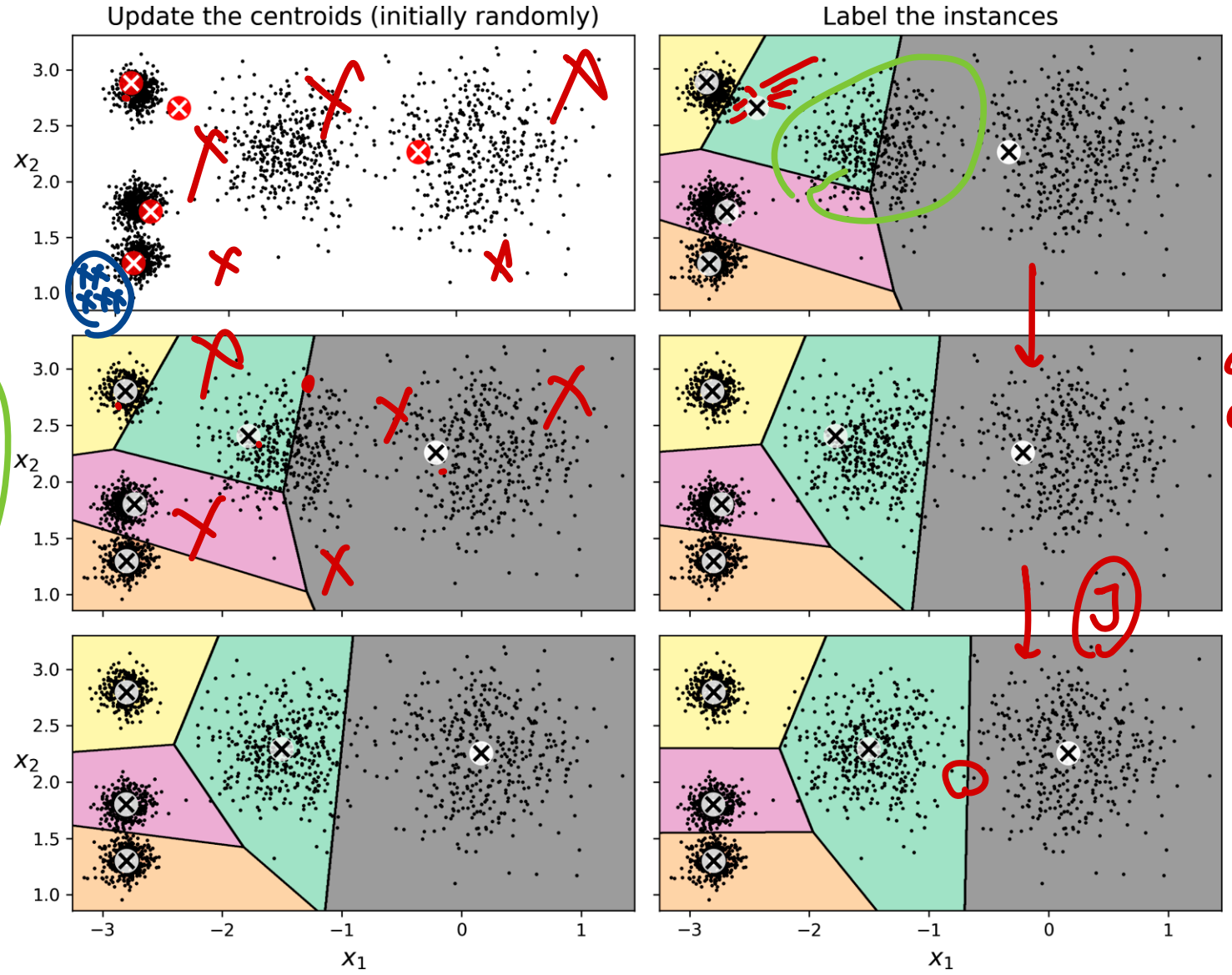
$x^7\ x^8\ x^9$

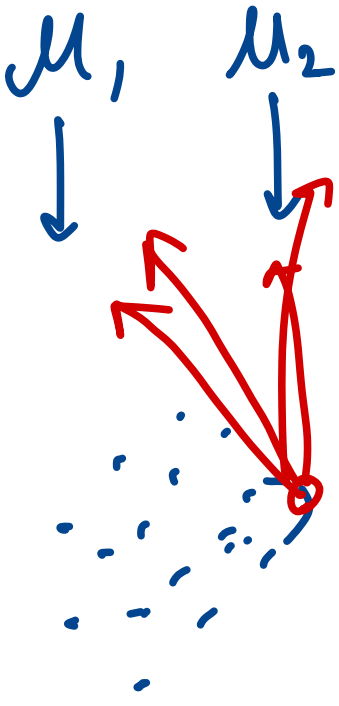
$\mu^1 = x^2 + x^3 + x^4 + x^7 + x^8 + x^9$

ex.

μ_2 { x^1 $c^1 = 2$; x^5 $c^5 = 2$; x^6 $c^6 = 2$; x^{10} $c^{10} = 2$ }

$$\mu_2 = \frac{x^1 + x^5 + x^6 + x^{10}}{4} \in \mathbb{R}^n$$


Update the centroids (initially randomly)
Label the instances
x_1
x_2



$\mu_1 \neq \mu_2$

μ

8

μ_2

9

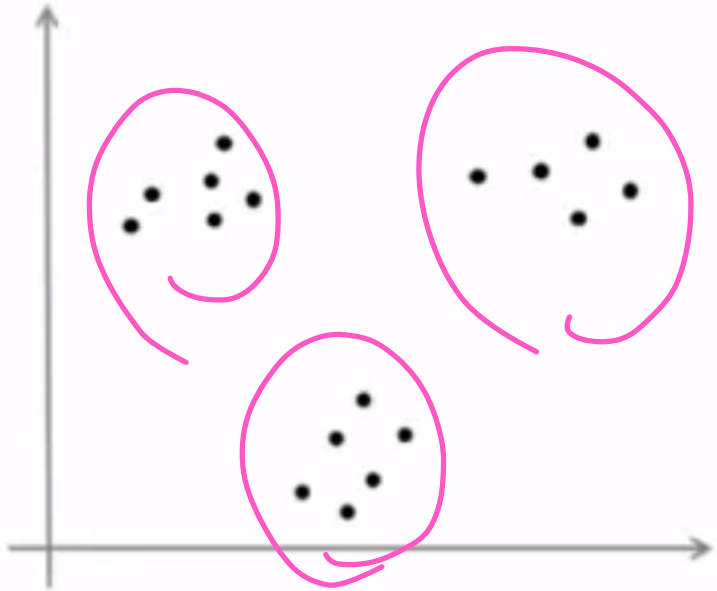
A

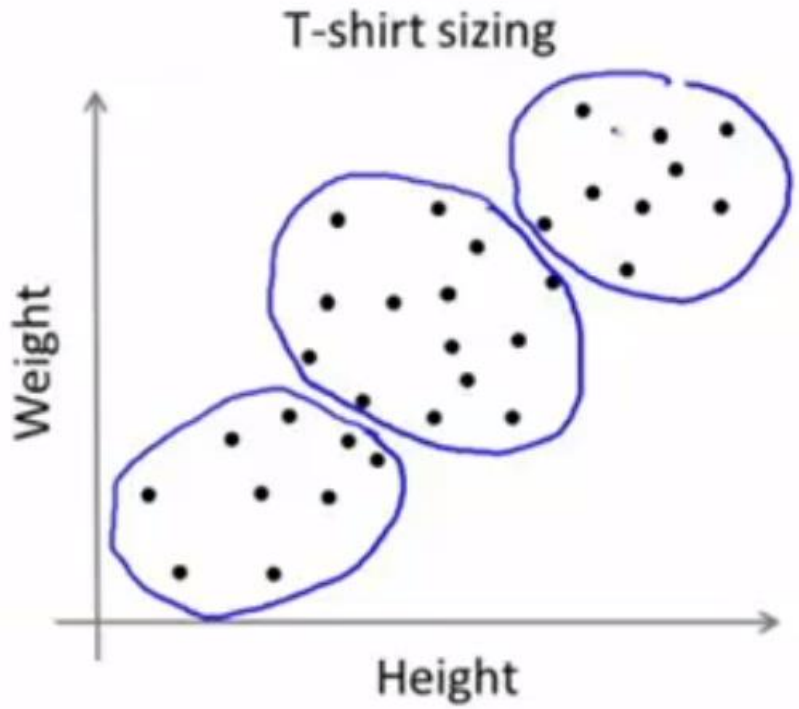
(8, 9)

$\Rightarrow$ ①

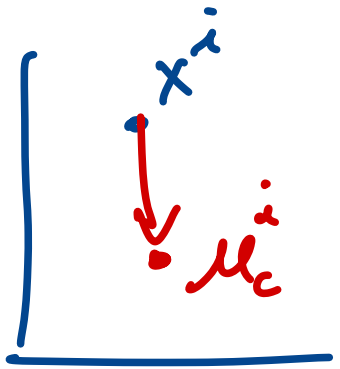
new centroid

K-means for non-separated clusters





Kmeans Cost Function



K means optimisation objective

$c^{(i)}$ ⇒ index of cluster (1 to K)

μ_K ⇒ cluster centroid

$\mu_c^{(i)}$ ⇒ cluster centroid to which x^i has been assigned.

$$J(c^1, \dots c^m, \mu_1, \dots \mu_K) = \frac{1}{m} \sum_{i=1}^{m} \|x^i - \mu_c^i\|^2$$

Minimise sum of distance btwn x^i & μ_c^i

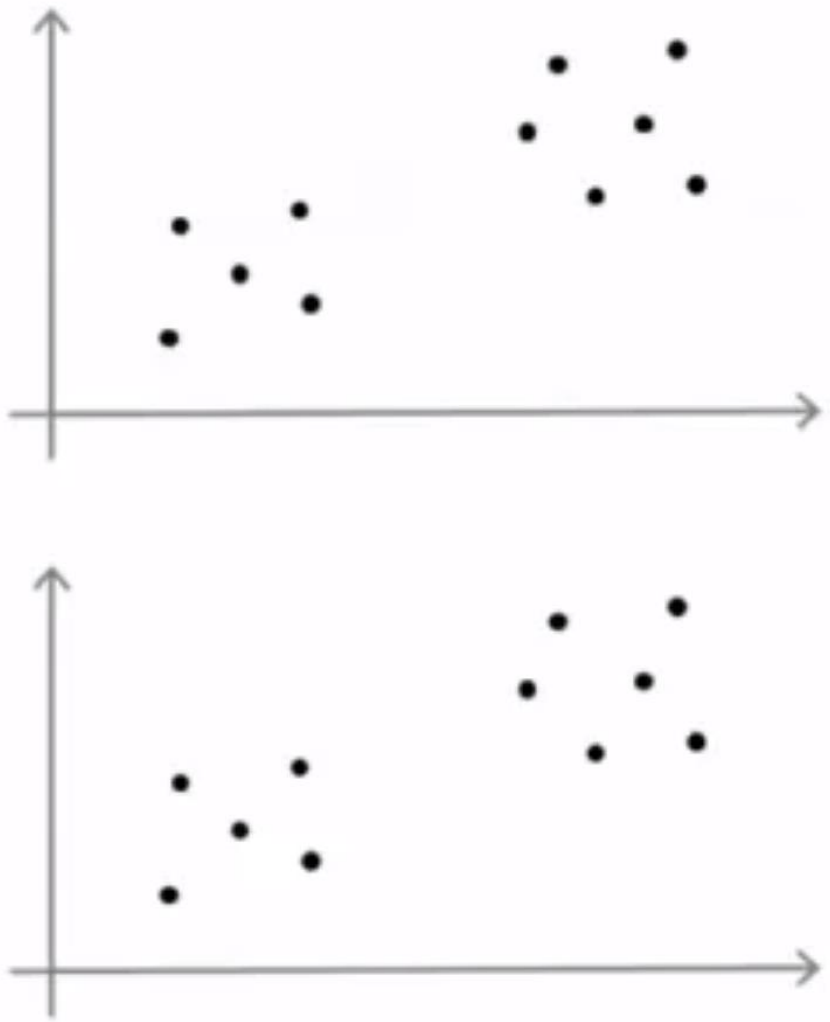
Random Initialisation

Random initialization

Should have $K < m$

Randomly pick K training examples.

Set $\mu_1, \ldots, \mu_K$ equal to these K examples.



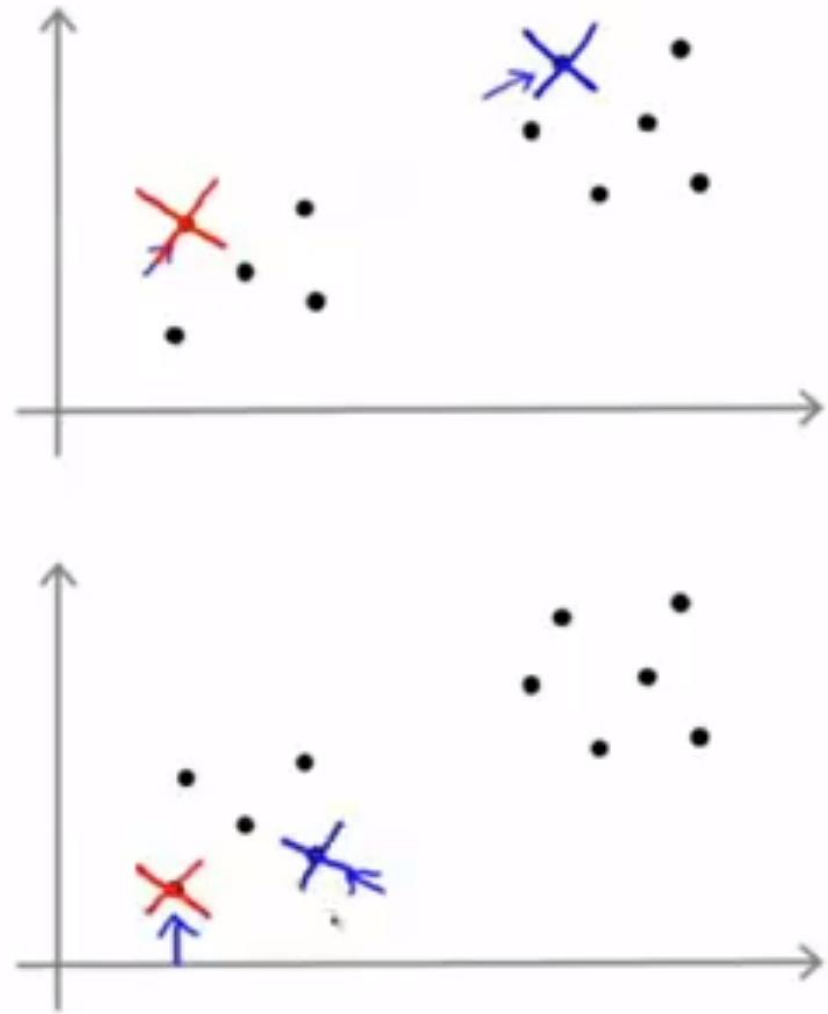
Random initialization

Should have $K < m$

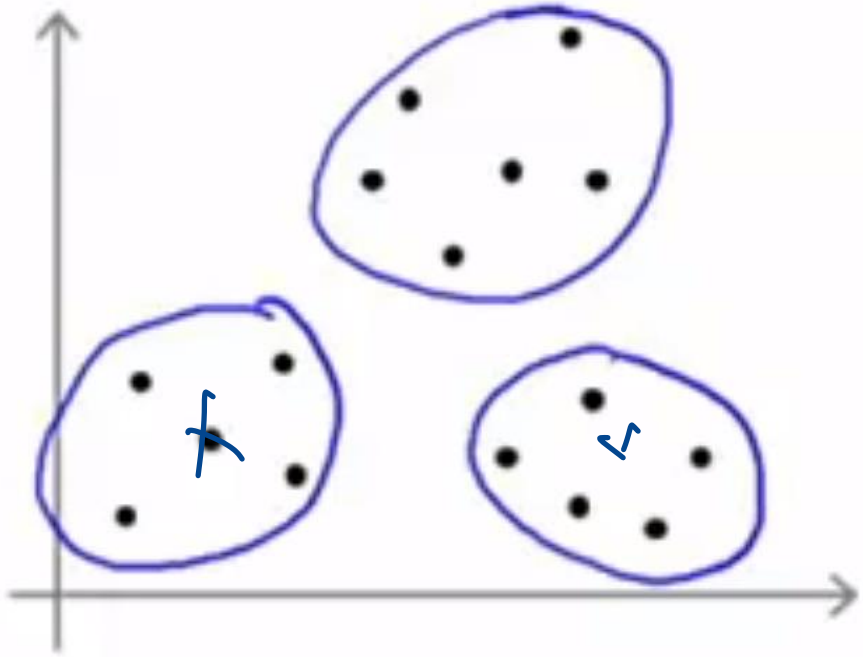
$K=2$

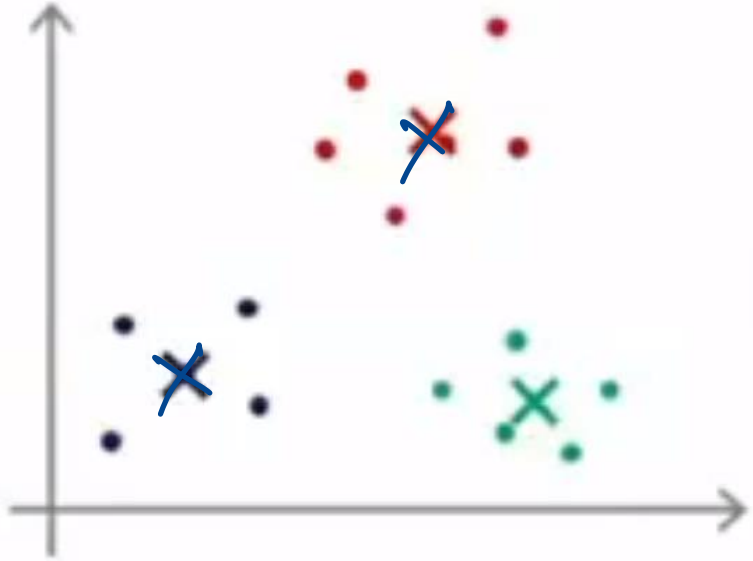
Randomly pick K training examples.

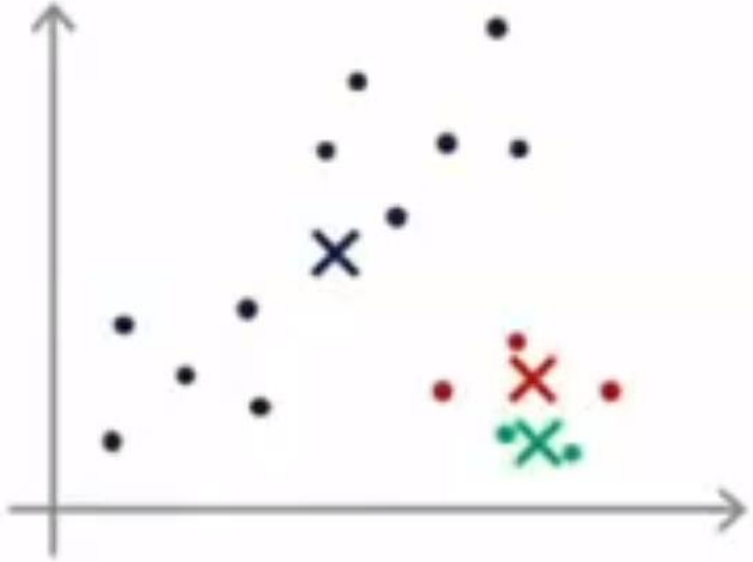
Set $\mu_1, \ldots, \mu_K$ equal to these K examples.

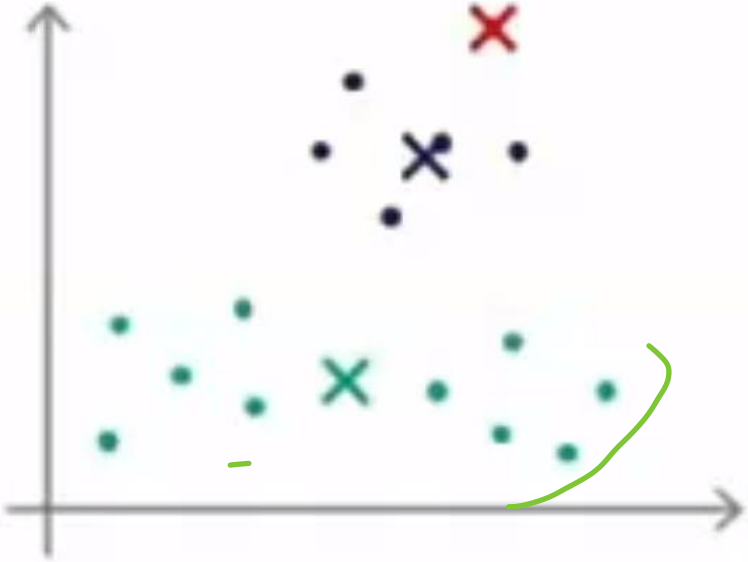


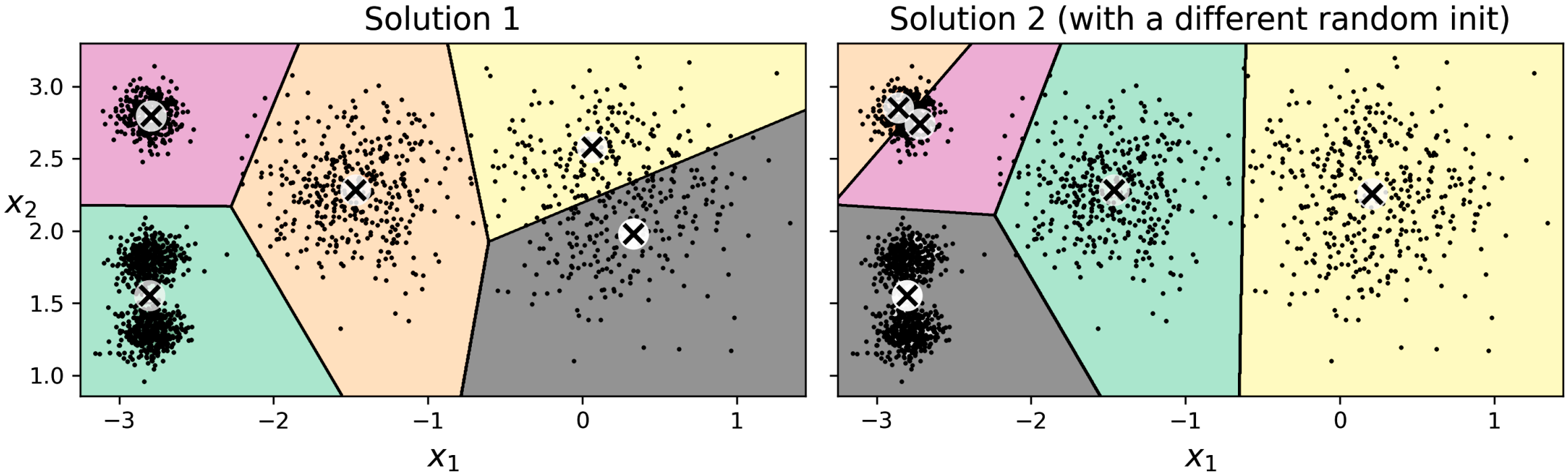
Local optima










Solution 1
Solution 2 (with a different random init)
3.0
2.5
2.0
1.5
1.0
x_2
−3
−2
−1
0
1
x_1
−3
−2
−1
0
1
x_1

Centroid Initialization

$$I = \sum_{i=1}^{m} \| x^i - \mu_c^i \|^2$$

- Inertia: Sum of squared distances between instances and their closest centroids

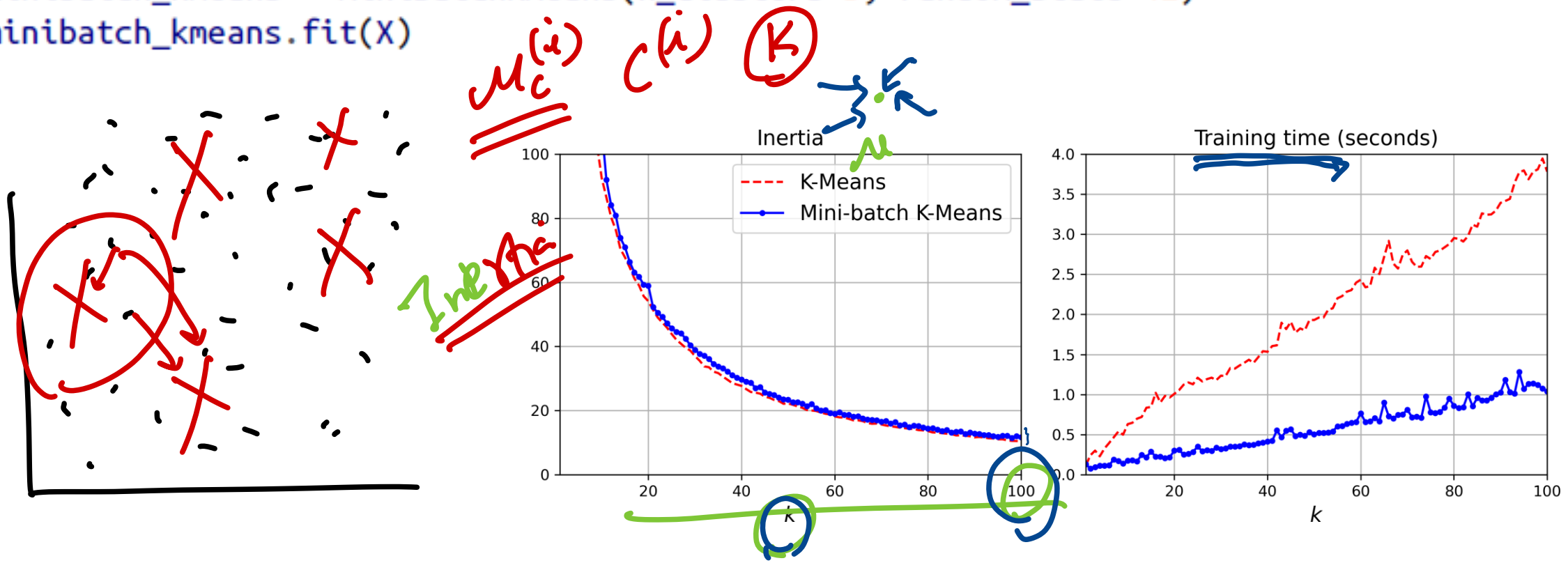
```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1, random_state=42)
kmeans.fit(X)
```

Run it multiple times

minimise inertia

Mini Batch

```
from sklearn.cluster import MiniBatchKMeans

minibatch_kmeans = MiniBatchKMeans(n_clusters=5, random_state=42)
minibatch_kmeans.fit(X)
```

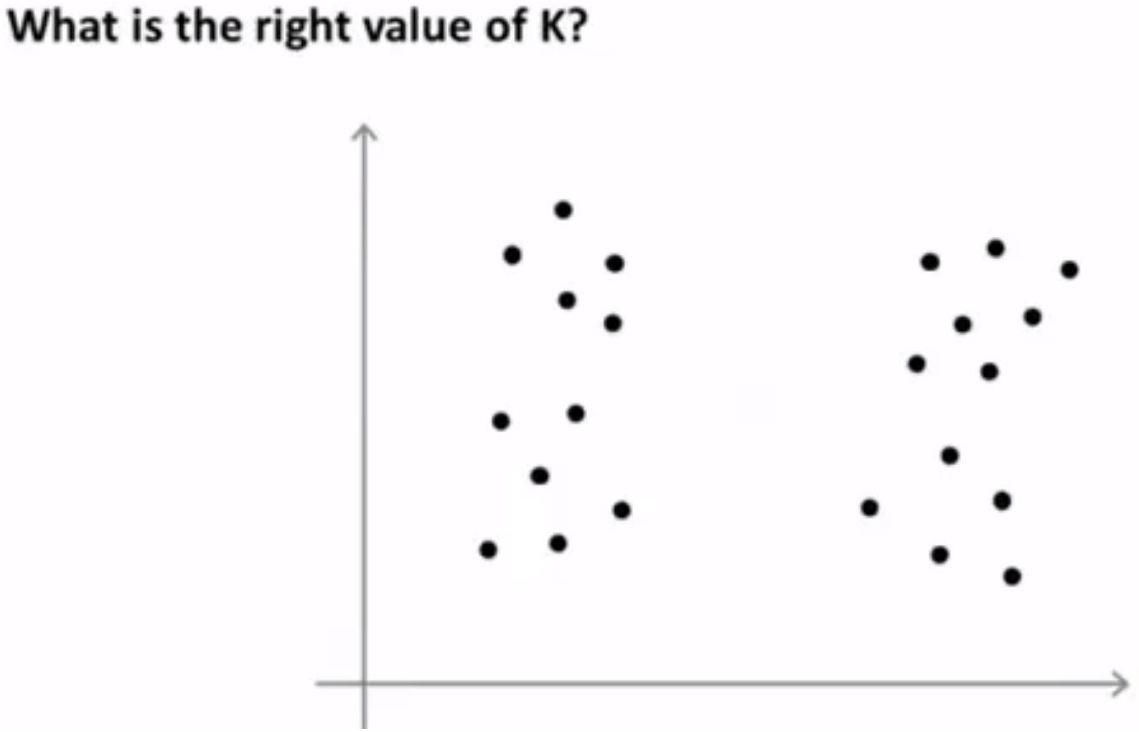


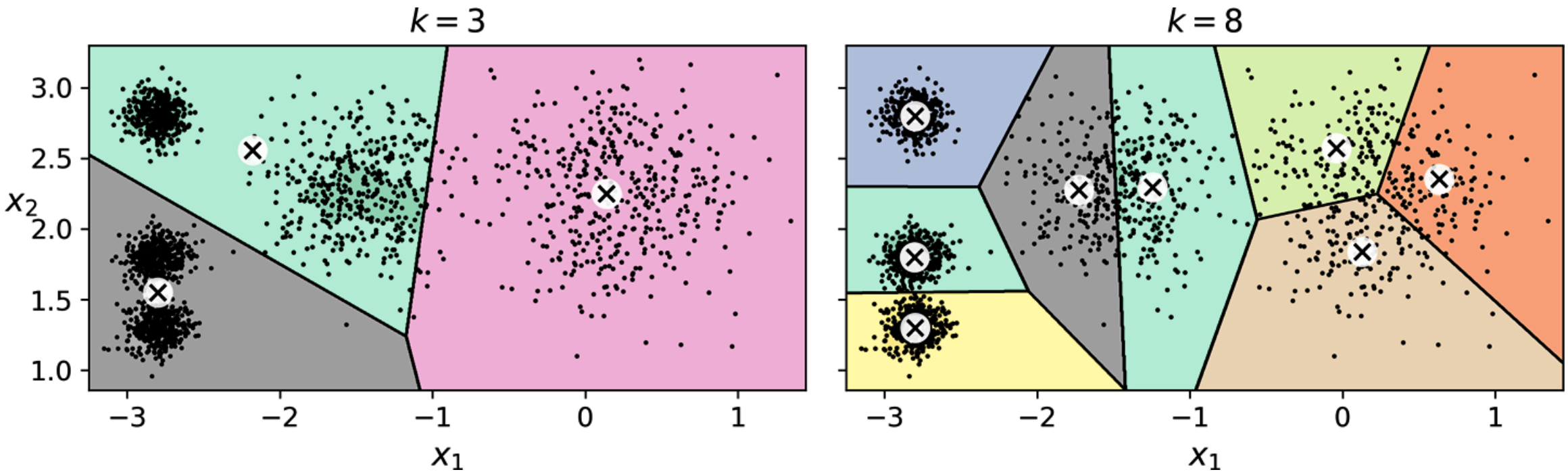
Q: What is the best hyperparameter?

A: There is no FOFY!

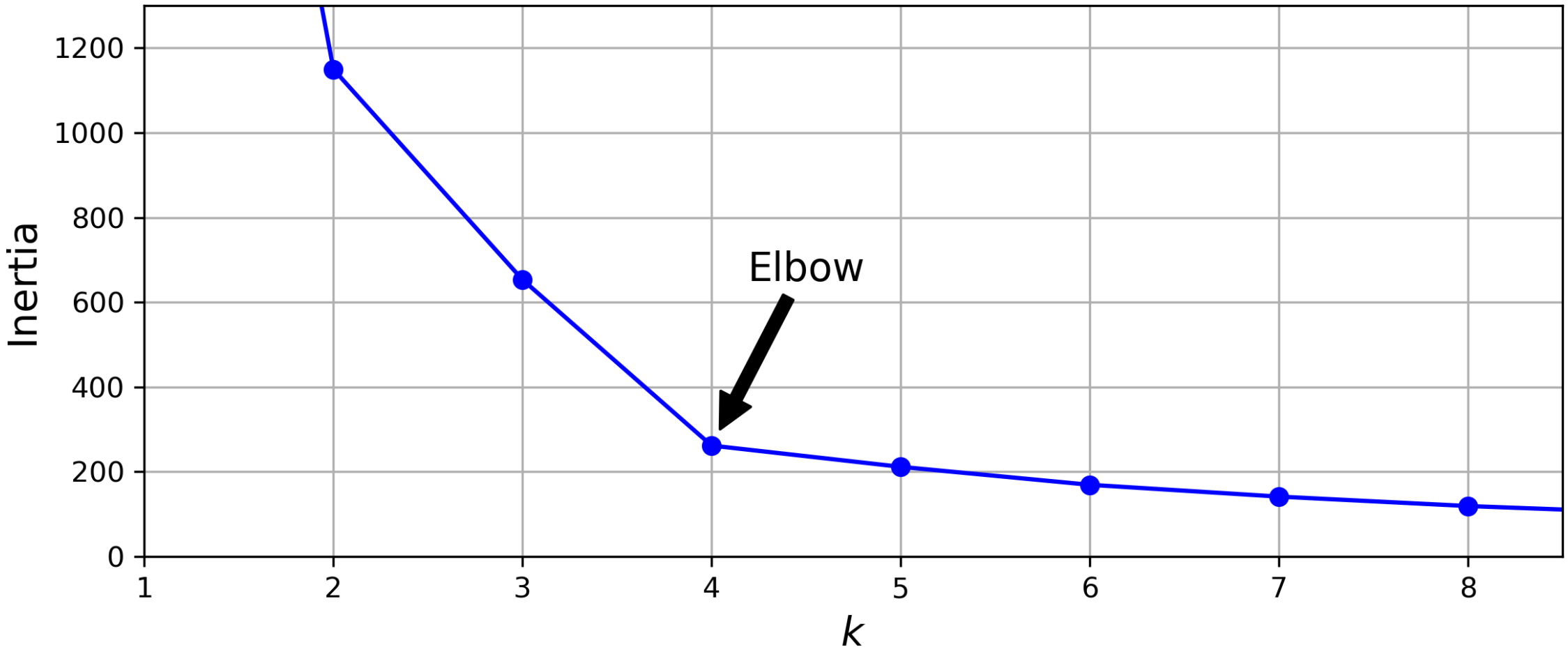
Choose the number of clusters

- There is no fixed way to do this …
- Manually, visual … most common is to choose the number of clusters by hands … CS229.
- No clear cut answer




$k = 3$
$k = 8$
3.0
2.5
2.0
1.5
1.0
x_2
−3
−2
−1
0
1
x_1

Elbow Method

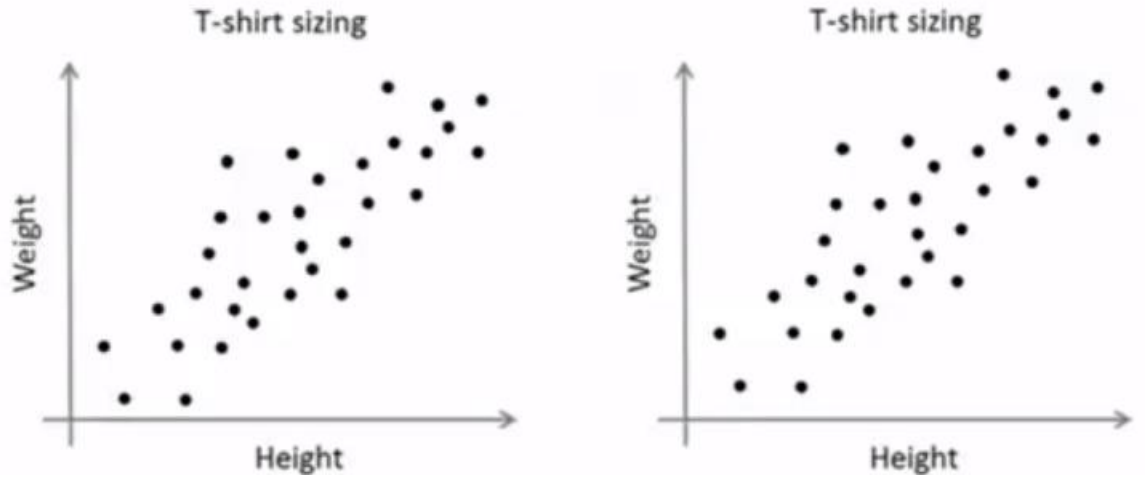


Elbow Method

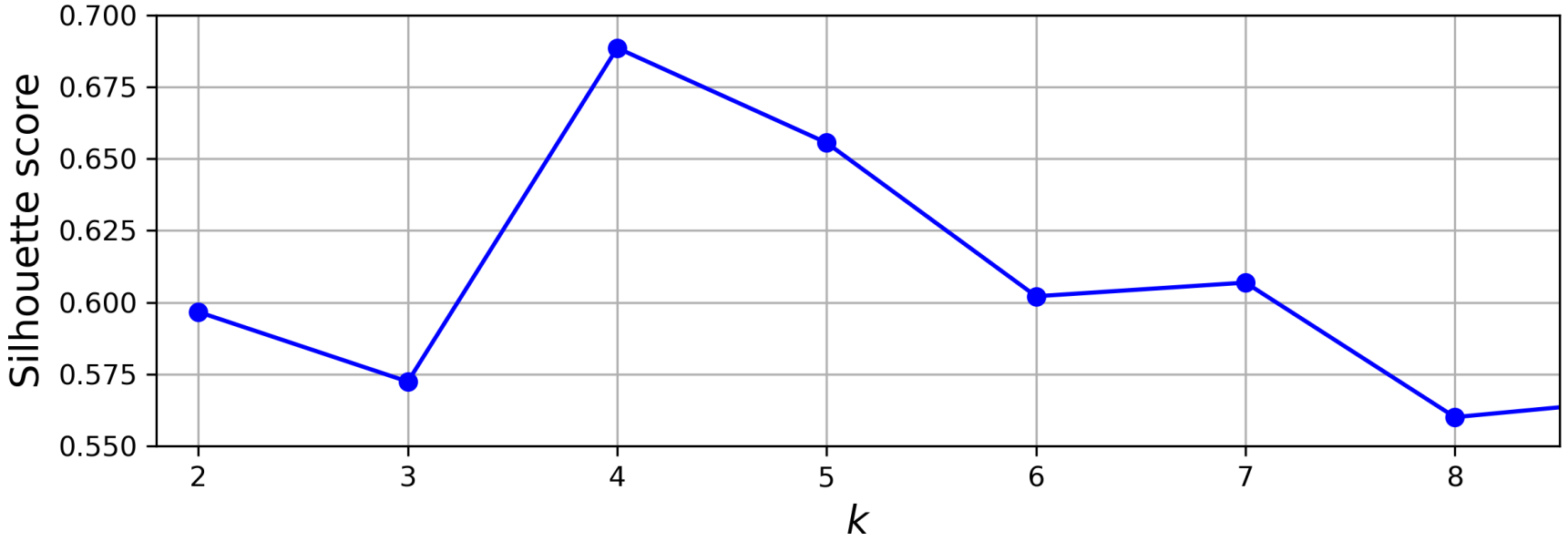
Choosing the value of K

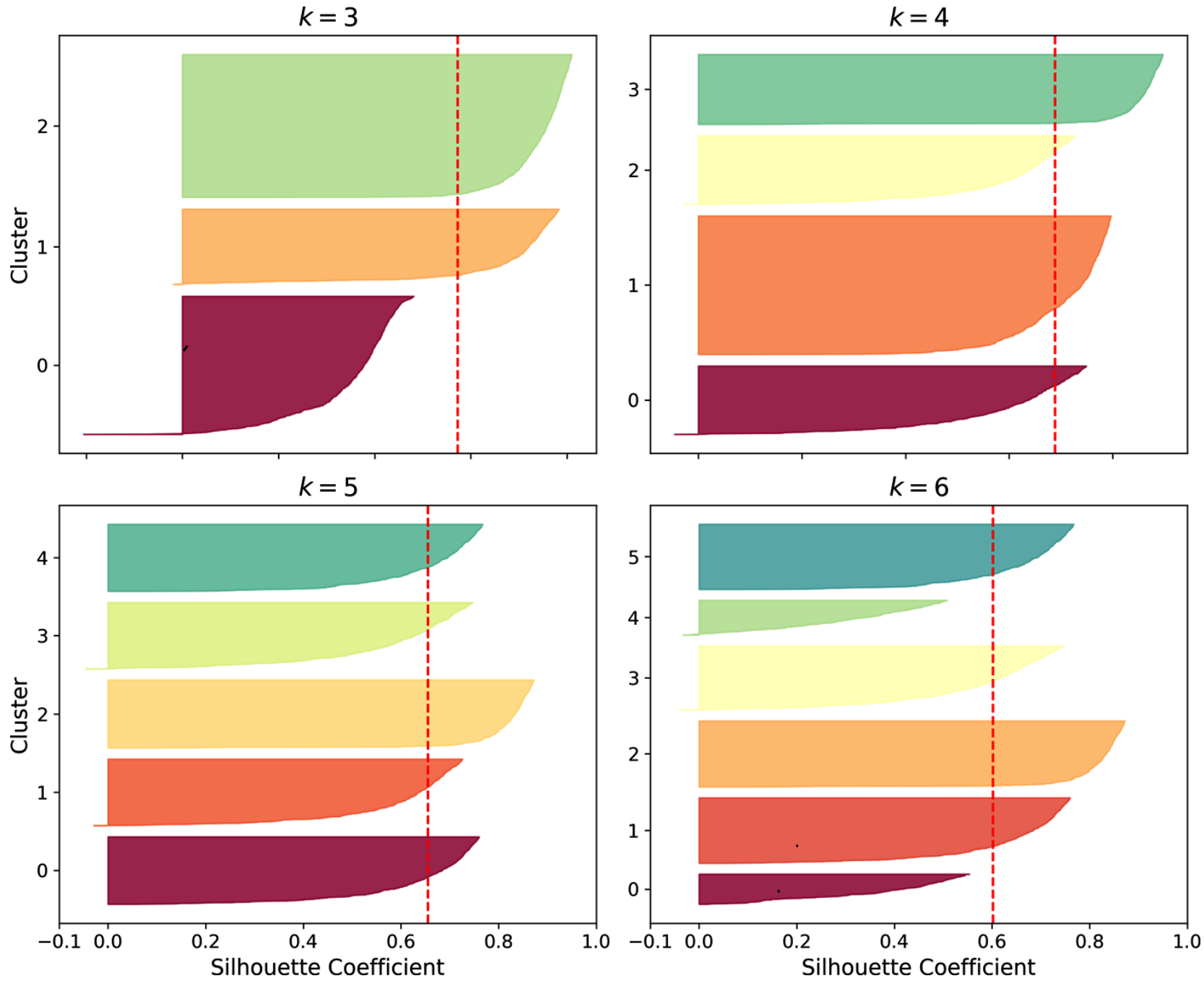
Sometimes, you're running K-means to get clusters to use for some later/downstream purpose. Evaluate K-means based on a metric for how well it performs for that later purpose.

E.g.



Silhouette Coefficient

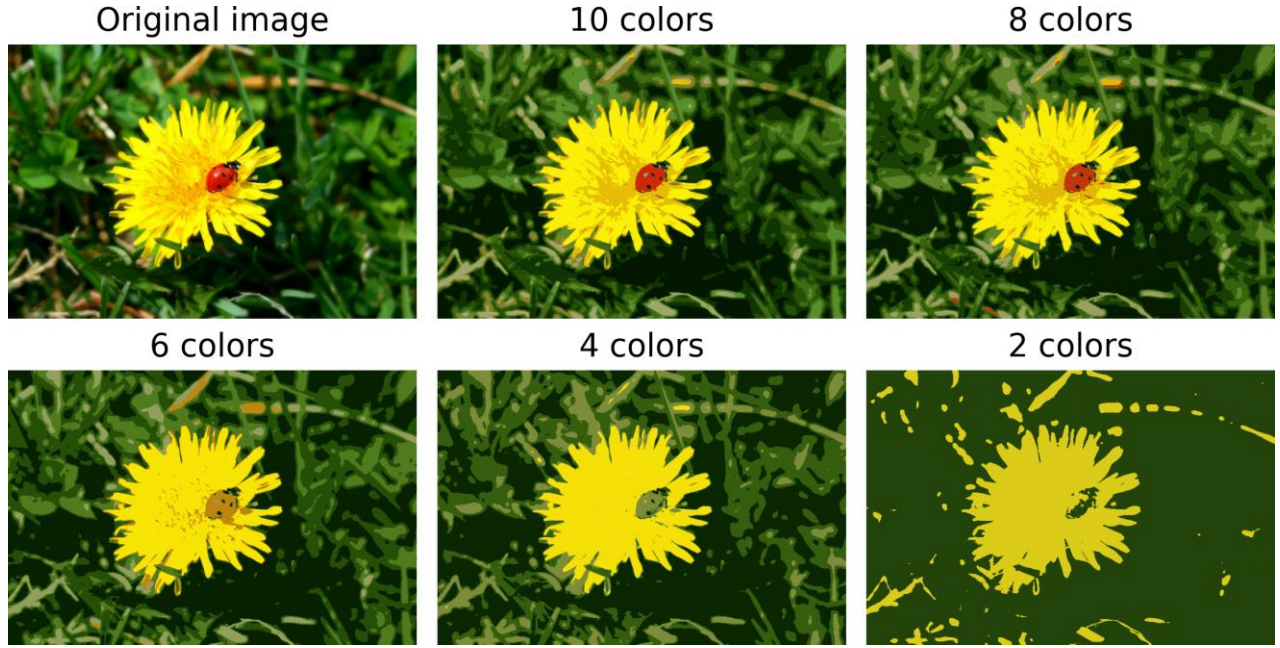



$k = 3$
$k = 4$
$k = 5$
$k = 6$
Cluster
Silhouette Coefficient
−0.1
0.0
0.2
0.4
0.6
0.8
1.0

Using Clustering for Image Segmentation

Image segmentation is the task of partitioning an image into multiple segments. There are several variants:

- In *color segmentation*, pixels with a similar color get assigned to the same segment. This is sufficient in many applications. For example, if you want to analyze satellite images to measure how much total forest area there is in a region, color segmentation may be just fine.
- In *semantic segmentation*, all pixels that are part of the same object type get assigned to the same segment. For example, in a self-driving car's vision system, all pixels that are part of a pedestrian's image might be assigned to the "pedestrian" segment (there would be one segment containing all the pedestrians).
- In *instance segmentation*, all pixels that are part of the same individual object are assigned to the same segment. In this case there would be a different segment for each pedestrian.



Using Clustering for Semi-Supervised Learning

Another use case for clustering is in semi-supervised learning, when we have plenty of unlabeled instances and very few labeled instances. In this section, we'll use the digits dataset, which is a simple MNIST-like dataset containing 1,797 grayscale 8×8 images representing the digits 0 to 9. First, let's load and split the dataset (it's already shuffled):

```
from sklearn.datasets import load_digits

X_digits, y_digits = load_digits(return_X_y=True)
X_train, y_train = X_digits[:1400], y_digits[:1400]
X_test, y_test = X_digits[1400:], y_digits[1400:]
```

We will pretend we only have labels for 50 instances. To get a baseline performance, let's train a logistic regression model on these 50 labeled instances:

```
from sklearn.linear_model import LogisticRegression

n_labeled = 50
log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

We can then measure the accuracy of this model on the test set (note that the test set must be labeled):

```
>>> log_reg.score(X_test, y_test)
0.7481108312342569
```

The model's accuracy is just 74.8%. That's not great: indeed, if you try training the model on the full training set, you will find that it will reach about 90.7% accuracy. Let's see how we can do better. First, let's cluster the training set into 50 clusters. Then, for each cluster, we'll find the image closest to the centroid. We'll call these images the *representative images*:

```
k = 50
kmeans = KMeans(n_clusters=k, random_state=42)
X_digits_dist = kmeans.fit_transform(X_train)
representative_digit_idx = np.argmin(X_digits_dist, axis=0)
X_representative_digits = X_train[representative_digit_idx]
```

Figure 9-13 shows the 50 representative images.

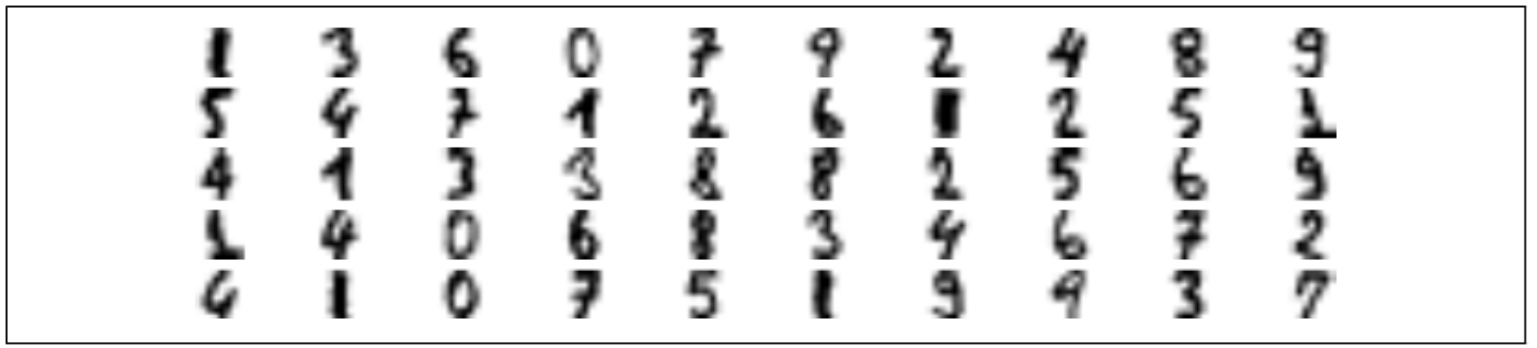


Figure 9-13. Fifty representative digit images (one per cluster)

Let's look at each image and manually label them:

```
y_representative_digits = np.array([1, 3, 6, 0, 7, 9, 2, ..., 5, 1, 9, 9, 3, 7])
```

Now we have a dataset with just 50 labeled instances, but instead of being random instances, each of them is a representative image of its cluster. Let's see if the performance is any better:

```
>>> log_reg = LogisticRegression(max_iter=10_000)
>>> log_reg.fit(X_representative_digits, y_representative_digits)
>>> log_reg.score(X_test, y_test)
0.8488664987405542
```

Wow! We jumped from 74.8% accuracy to 84.9%, although we are still only training the model on 50 instances. Since it is often costly and painful to label instances, especially when it has to be done manually by experts, it is a good idea to label representative instances rather than just random instances.

But perhaps we can go one step further: what if we propagated the labels to all the other instances in the same cluster? This is called *label propagation*:

```
y_train_propagated = np.empty(len(X_train), dtype=np.int64)
for i in range(k):
    y_train_propagated[kmeans.labels_ == i] = y_representative_digits[i]
```

Now let's train the model again and look at its performance:

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train, y_train_propagated)
>>> log_reg.score(X_test, y_test)
0.8942065491183879
```

We got another significant accuracy boost! Let's see if we can do even better by ignoring the 1% of instances that are farthest from their cluster center: this should eliminate some outliers. The following code first computes the distance from each instance to its closest cluster center, then for each cluster it sets the 1% largest distances to −1. Lastly, it creates a set without these instances marked with a −1 distance:

```
percentile_closest = 99

X_cluster_dist = X_digits_dist[np.arange(len(X_train)), kmeans.labels_]
for i in range(k):
    in_cluster = (kmeans.labels_ == i)
    cluster_dist = X_cluster_dist[in_cluster]
    cutoff_distance = np.percentile(cluster_dist, percentile_closest)
    above_cutoff = (X_cluster_dist > cutoff_distance)
    X_cluster_dist[in_cluster & above_cutoff] = -1
```

```
partially_propagated = (X_cluster_dist != -1)
X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]
```

Now let's train the model again on this partially propagated dataset and see what accuracy we get:

```
>>> log_reg = LogisticRegression(max_iter=10_000)
>>> log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
>>> log_reg.score(X_test, y_test)
0.9093198992443325
```

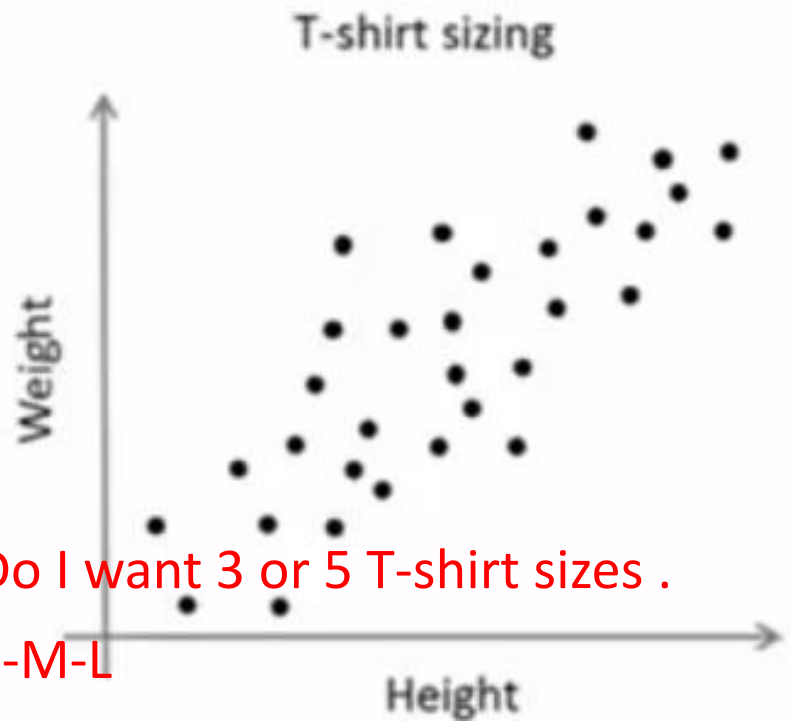
Nice! With just 50 labeled instances (only 5 examples per class on average!) we got 90.9% accuracy, which is actually slightly higher than the performance we got on the fully labeled digits dataset (90.7%). This is partly thanks to the fact that we dropped some outliers, and partly because the propagated labels are actually pretty good—their accuracy is about 97.5%, as the following code shows:

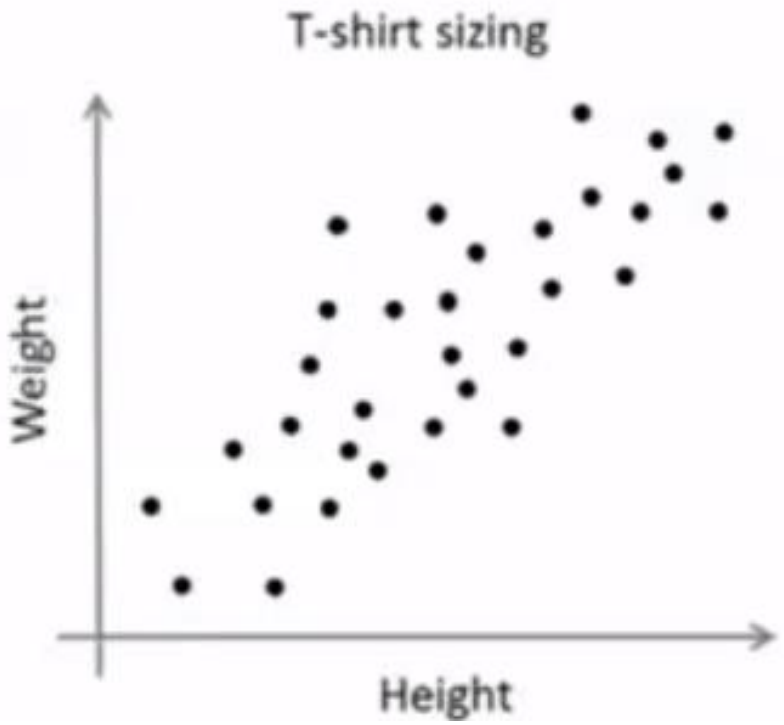
```
>>> (y_train_partially_propagated == y_train[partially_propagated]).mean()
0.9755555555555555
```

Choosing the value of K

Sometimes, you're running K-means to get clusters to use for some later/downstream purpose. Evaluate K-means based on a metric for how well it performs for that later purpose.

E.g.





- Do I want 3 or 5 T-shirt sizes .
- S-M-L
- XS, S, M, L, XL

Initialization Algorithm Functioning

1. Take one centroid $\mathbf{c}^{(1)}$, chosen uniformly at random from the dataset.

2. Take a new centroid $\mathbf{c}^{(i)}$, choosing an instance $\mathbf{x}^{(i)}$ with probability $D\left(\mathbf{x}^{(i)}\right)^2 / \sum_{j=1}^{m} D\left(\mathbf{x}^{(j)}\right)^2$, where $D(\mathbf{x}^{(i)})$ is the distance between the instance $\mathbf{x}^{(i)}$ and the closest centroid that was already chosen. This probability distribution ensures that instances farther away from already chosen centroids are much more likely to be selected as centroids.
3. Repeat the previous step until all k centroids have been chosen.

Initialization Algorithm Functioning

1. Take one centroid $\mathbf{c}^{(1)}$, chosen uniformly at random from the dataset.

2. Take a new centroid $\mathbf{c}^{(i)}$, choosing an instance $\mathbf{x}^{(i)}$ with probability $D\left(\mathbf{x}^{(i)}\right)^2 / \sum_{j=1}^{m} D\left(\mathbf{x}^{(j)}\right)^2$, where $D(\mathbf{x}^{(i)})$ is the distance between the instance $\mathbf{x}^{(i)}$ and the closest centroid that was already chosen. This probability distribution ensures that instances farther away from already chosen centroids are much more likely to be selected as centroids.
3. Repeat the previous step until all k centroids have been chosen.

DBSCAN

DBSCAN

The *density-based spatial clustering of applications with noise* (DBSCAN) algorithm defines clusters as continuous regions of high density. Here is how it works:

- For each instance, the algorithm counts how many instances are located within a small distance ε (epsilon) from it. This region is called the instance's *ε-neighborhood*.
- If an instance has at least `min_samples` instances in its ε-neighborhood (including itself), then it is considered a *core instance*. In other words, core instances are those that are located in dense regions.
- All instances in the neighborhood of a core instance belong to the same cluster. This neighborhood may include other core instances; therefore, a long sequence of neighboring core instances forms a single cluster.
- Any instance that is not a core instance and does not have one in its neighborhood is considered an anomaly.

This algorithm works well if all the clusters are well separated by low-density regions. The `DBSCAN` class in Scikit-Learn is as simple to use as you might expect. Let's test it on the moons dataset, introduced in Chapter 5:

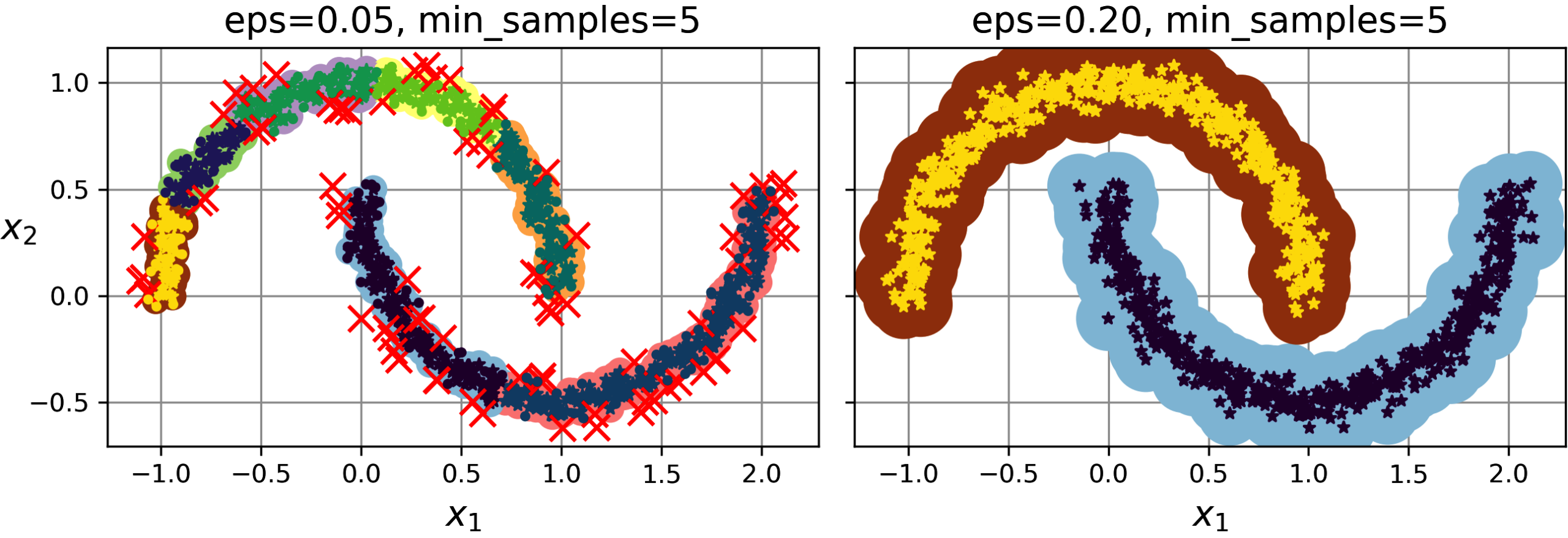
```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.05)
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
```

The labels of all the instances are now available in the `labels_` instance variable:

```
>>> dbscan.labels_
array([ 0,  2, -1, -1,  1,  0,  0,  0,  2,  5, [...], 3,  3,  4,  2,  6,  3])
```

Notice that some instances have a cluster index equal to –1, which means that they are considered as anomalies by the algorithm. The indices of the core instances are available in the `core_sample_indices_` instance variable, and the core instances themselves are available in the `components_` instance variable:

```
>>> dbscan.core_sample_indices_
array([  0,   4,   5,   6,   7,   8,  10,  11, [...], 993, 995, 997, 998, 999])
>>> dbscan.components_
array([[-0.02137124,  0.40618608],
       [-0.84192557,  0.53058695],
       [...],
       [ 0.79419406,  0.60777171]])
```


eps=0.05, min_samples=5
eps=0.20, min_samples=5
1.0
0.5
0.0
−0.5
x_2
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x_1

Other Clustering

- Agglomerative Clustering
- BIRCH
- Mean-shift
- Affinity Propagation
- Spectral Clustering

Agglomerative clustering

A hierarchy of clusters is built from the bottom up. Think of many tiny bubbles floating on water and gradually attaching to each other until there's one big group of bubbles. Similarly, at each iteration, agglomerative clustering connects the nearest pair of clusters (starting with individual instances). If you drew a tree with a branch for every pair of clusters that merged, you would get a binary tree of clusters, where the leaves are the individual instances. This approach can capture clusters of various shapes; it also produces a flexible and informative cluster tree instead of forcing you to choose a particular cluster scale, and it can be used with any pairwise distance. It can scale nicely to large numbers of instances if you provide a connectivity matrix, which is a sparse $m \times m$ matrix that indicates which pairs of instances are neighbors (e.g., returned by `sklearn.neighbors.kneighbors_graph()`). Without a connectivity matrix, the algorithm does not scale well to large datasets.

BIRCH

The balanced iterative reducing and clustering using hierarchies (BIRCH) algorithm was designed specifically for very large datasets, and it can be faster than batch *k*-means, with similar results, as long as the number of features is not too large (<20). During training, it builds a tree structure containing just enough information to quickly assign each new instance to a cluster, without having to store all the instances in the tree: this approach allows it to use limited memory while handling huge datasets.

Mean-shift

This algorithm starts by placing a circle centered on each instance; then for each circle it computes the mean of all the instances located within it, and it shifts the circle so that it is centered on the mean. Next, it iterates this mean-shifting step until all the circles stop moving (i.e., until each of them is centered on the mean of the instances it contains). Mean-shift shifts the circles in the direction of higher density, until each of them has found a local density maximum. Finally, all the instances whose circles have settled in the same place (or close enough) are assigned to the same cluster. Mean-shift has some of the same features as DBSCAN, like how it can find any number of clusters of any shape, it has very few hyperparameters (just one—the radius of the circles, called the *bandwidth*), and it relies on local density estimation. But unlike DBSCAN, mean-shift tends to chop clusters into pieces when they have internal density variations. Unfortunately, its computational complexity is $O(m^2n)$, so it is not suited for large datasets.

Affinity propagation

In this algorithm, instances repeatedly exchange messages between one another until every instance has elected another instance (or itself) to represent it. These elected instances are called *exemplars*. Each exemplar and all the instances that elected it form one cluster. In real-life politics, you typically want to vote for a candidate whose opinions are similar to yours, but you also want them to win the election, so you might choose a candidate you don't fully agree with, but who is more popular. You typically evaluate popularity through polls. Affinity propagation works in a similar way, and it tends to choose exemplars located near the center of clusters, similar to *k*-means. But unlike with *k*-means, you don't have to pick a number of clusters ahead of time: it is determined during training. Moreover, affinity propagation can deal nicely with clusters of different sizes. Sadly, this algorithm has a computational complexity of $O(m^2)$, so it is not suited for large datasets.

Spectral clustering

This algorithm takes a similarity matrix between the instances and creates a low-dimensional embedding from it (i.e., it reduces the matrix's dimensionality), then it uses another clustering algorithm in this low-dimensional space (Scikit-Learn's implementation uses *k*-means). Spectral clustering can capture complex cluster structures, and it can also be used to cut graphs (e.g., to identify clusters of friends on a social network). It does not scale well to large numbers of instances, and it does not behave well when the clusters have very different sizes.

GMM