Hands-on Machine Learning

9. Unsupervised Learning

Unsupervised Learning

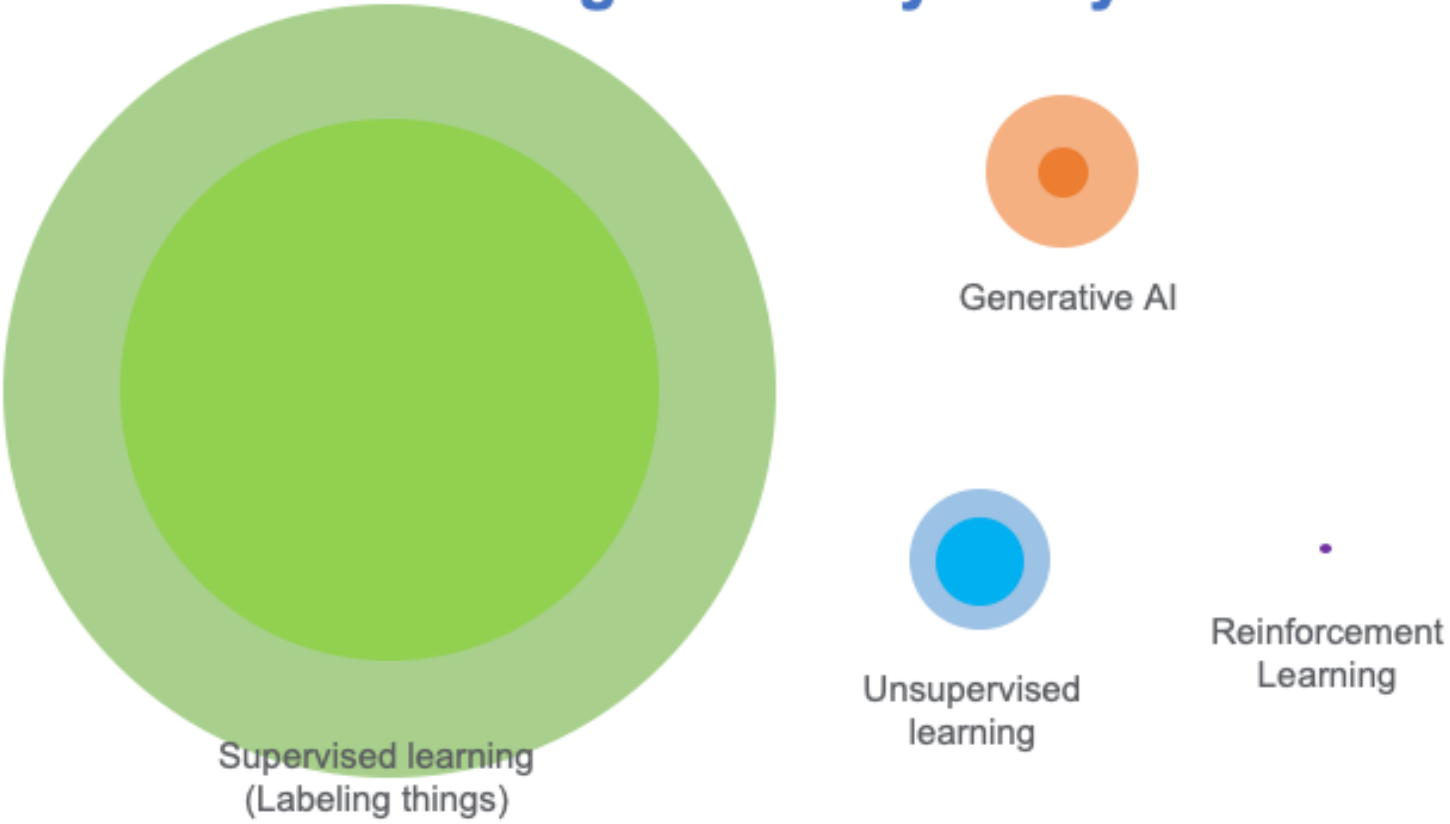
- Most of the applications of machine learning today are based on supervised learning, but the vast majority of the available data is unlabeled.

- Common unsupervised learning task:
  - *Clustering:* group similar instances together into *clusters.*
  - *Anomaly detection* (*outlier detection*)*:* learn what "normal" data looks like, and then use that to detect abnormal instances.
  - *Density estimation:* estimating the *probability density function* (PDF) of the random process that generated the dataset.

Question

- Difference between anomaly and outlier data?

Value from AI technologies: Today → 3 years

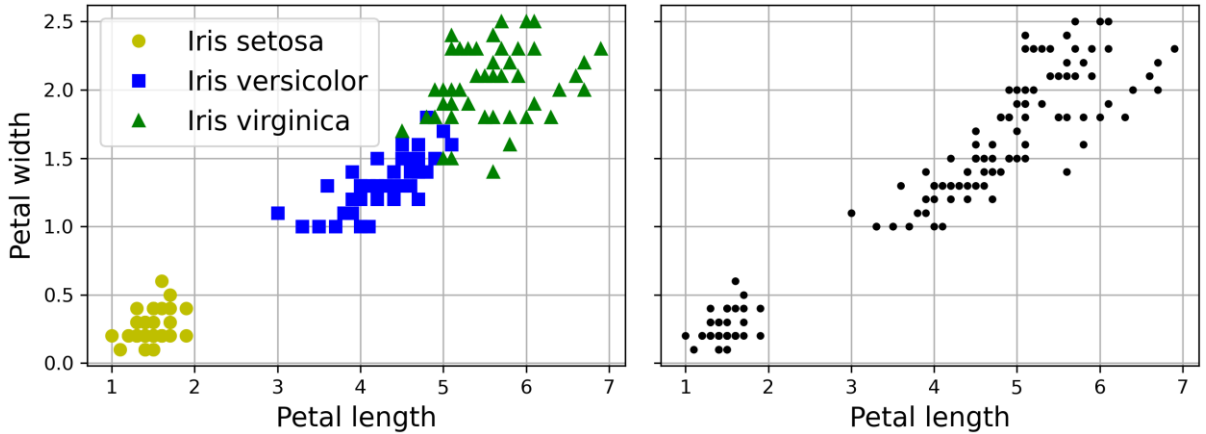


Andrew Ng

1.
Clustering

Clustering

- *Clustering*: the task of identifying similar instances and assigning them to *clusters*, or groups of similar instances.
- Similar to classification, each instance gets assigned to a group. But, unlike classification, clustering is an unsupervised task.



Applications of Clustering

- *Customer segmentation:* cluster customers based on their activity to adapt your products and marketing campaigns to each segment.
- *Data analysis:* when you analyze a new dataset, it can be helpful to run a clustering algorithm, and then analyze each cluster separately.
- *Dimensionality reduction:* each instance's feature vector $\mathbf{x}$ can be replaced with the vector of its cluster affinities.
  - affinity is any measure of how well an instance fits into a cluster.
- *Feature engineering:* the cluster affinities can often be useful as extra features.
- *Anomaly detection:* any instance that has a low affinity to all the clusters is likely to be an anomaly.

Applications of Clustering

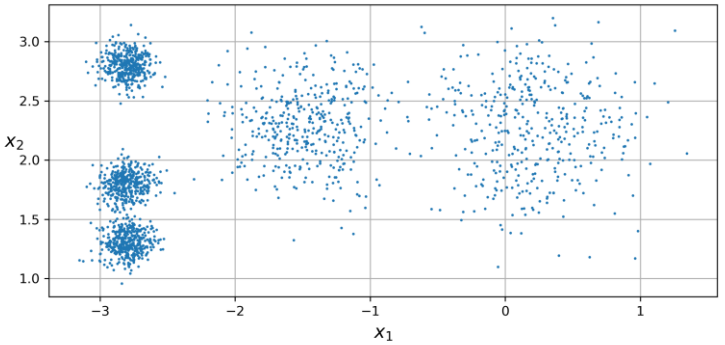
- *Semi-supervised learning:* if you only have a few labels, you could perform clustering and propagate the labels to all the instances in the same cluster.
- *Search engines:* to search for images that are similar to a reference image, apply a clustering algorithm to all the images in your database; and return all the images from the cluster of the searched image.
- *Image segmentation:* cluster pixels according to their color, then replace each pixel's color with the mean color of its cluster.
  - Image segmentation is used in many object detection and tracking systems, as it makes it easier to detect the contour of each object.

2.

K-means

K-means

- k-means is a simple algorithm capable of clustering many datasets quickly and efficiently.
  - We have to specify the number of clusters k that the algorithm must find.




```
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)
```

K-means

- Each instance will be assigned to one of the five clusters.
  - An instance's *label* is the index of its cluster.
- `KMeans` instance preserves the predicted labels of the instances it was trained on, available via the `labels_` instance variable:

```
y_pred
```

```
array([0, 0, 4, ..., 3, 1, 0])
```

```
y_pred is kmeans.labels_
```

```
True
```

- The centroids that the algorithm found:

```
kmeans.cluster_centers_
```

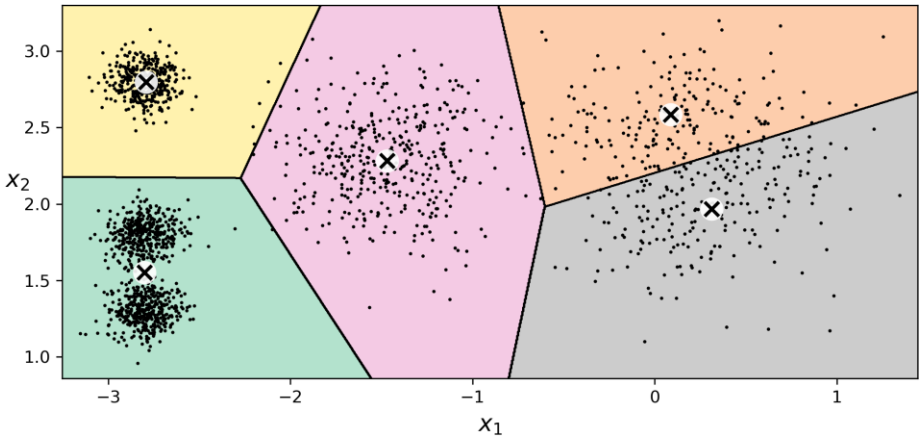
```
array([[-2.80389616,  1.80117999],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80037642,  1.30082566]])
```

K-means

- New instances get assigned 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)
```

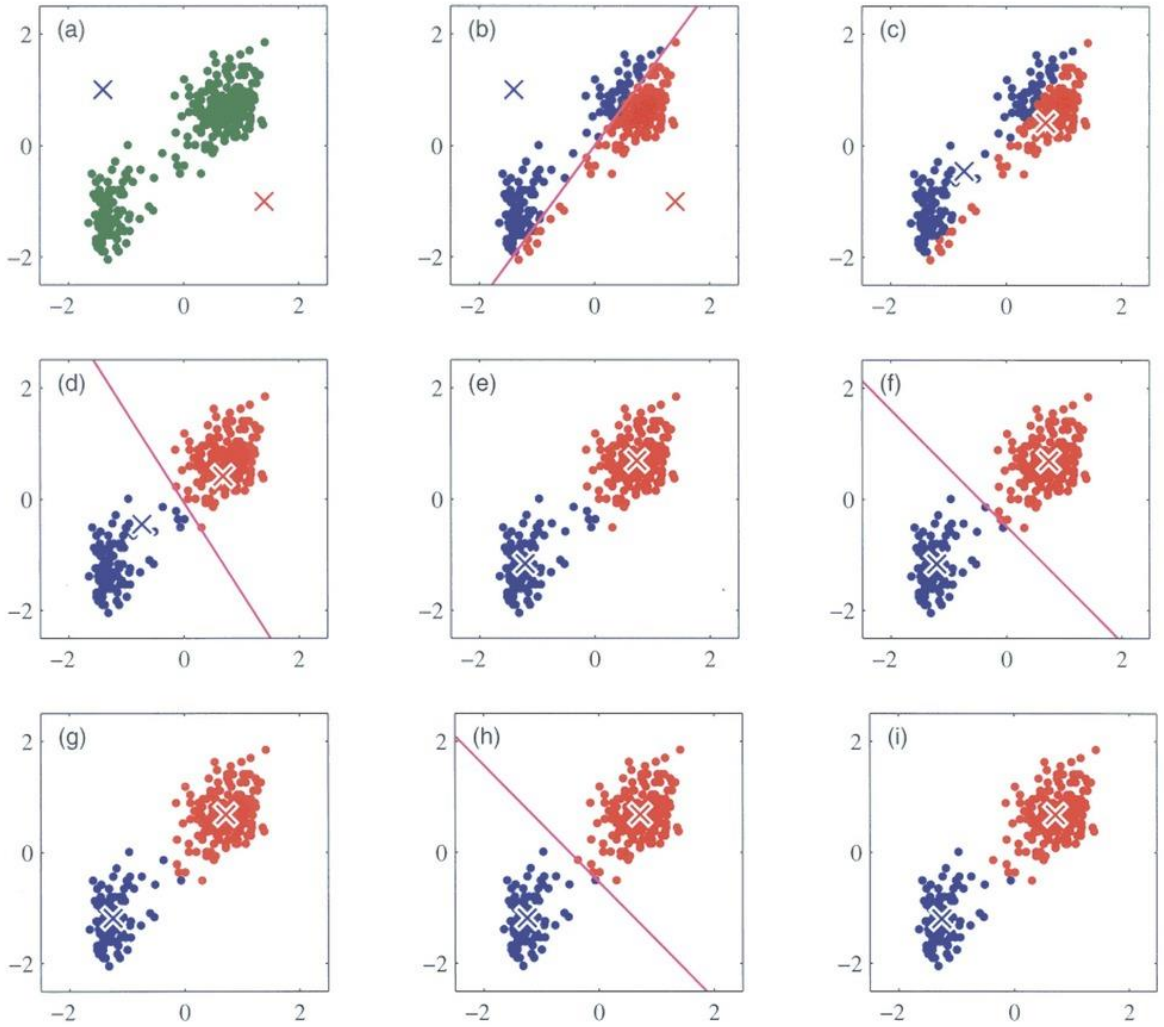


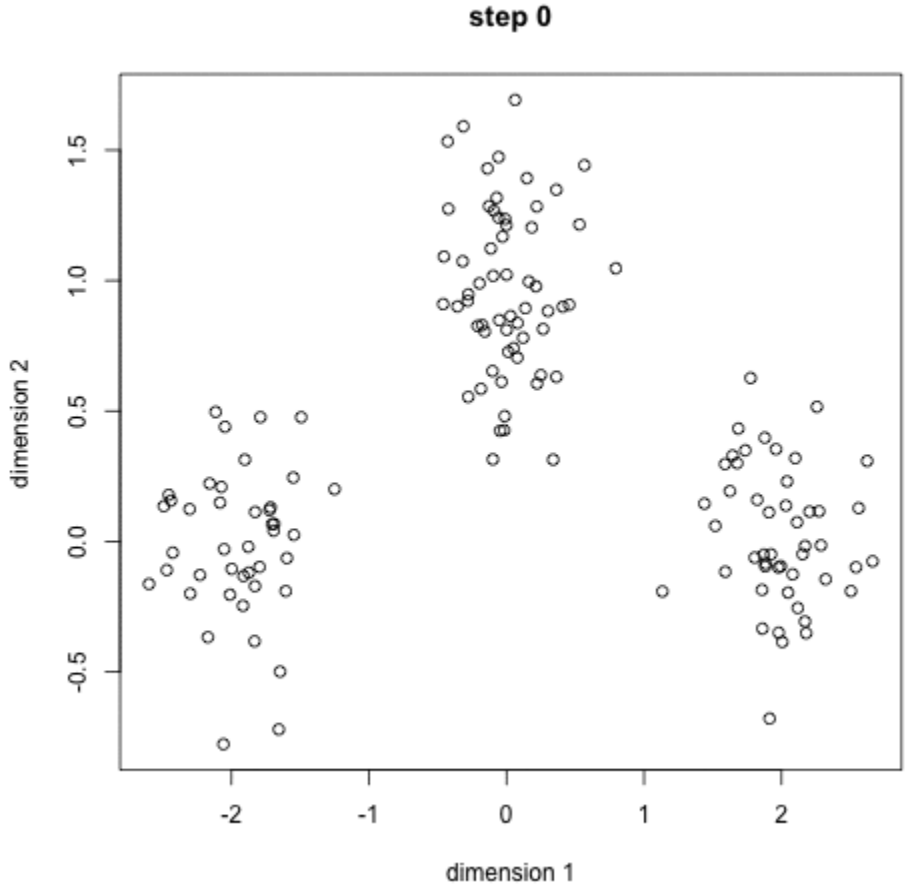
The k-means algorithm

- If you have the centroids, you could label the instances in the dataset by assigning them to the cluster whose centroid is closest.
- If you have all the instance labels, you could locate each cluster's centroid by computing the mean of the instances in that cluster.
- Start by placing the centroids randomly:
  1) label the instances
  2) update the centroids
  3) repeat (1) and (2) until the centroids stop moving.
- The algorithm is guaranteed to converge in a finite number of steps because the mean squared distance between the instances and their closest centroids can only go down at each step.

Visualization

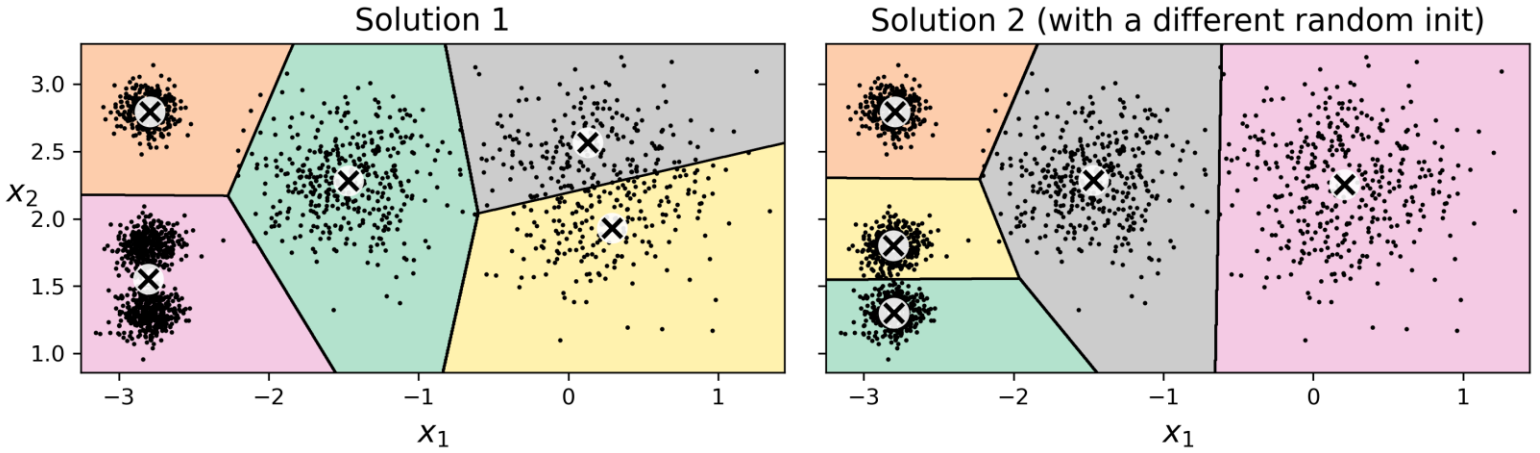
- Visualizing K-Means Clustering


(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)


step 0
dimension 2
1.5
1.0
0.5
0.0
-0.5
-2
-1
0
1
2
dimension 1

K-means disadvantage

- Although the algorithm is guaranteed to converge, it may not converge to the right solution
  - it may converge to a local optimum
  - whether it does or not depends on the centroid initialization.



Centroid initialization methods

- If you know approximately where the centroids should be (e.g., if you ran another clustering algorithm earlier), then you can set the `init` hyperparameter to a NumPy array containing the list of 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)
```

- Another solution is to run the algorithm multiple times with different random initializations and keep the best solution.
  - The number of random initializations is controlled by the `n_init`.
  - The default it is equal to 10.

Performance Metric

- Model's *inertia*: the sum of the squared distances between the instances and their closest centroids.
- The `KMeans` class runs the algorithm `n_init` times and keeps the model with the lowest inertia.
- A model's inertia is accessible via the `inertia_` instance variable:

```
kmeans.inertia_
```

```
211.5985372581684
```

- The `score()` method returns the negative inertia.
  - It is negative because a predictor's `score()` method must always respect Scikit-Learn's "greater is better" rule.

```
kmeans.score(X)
```

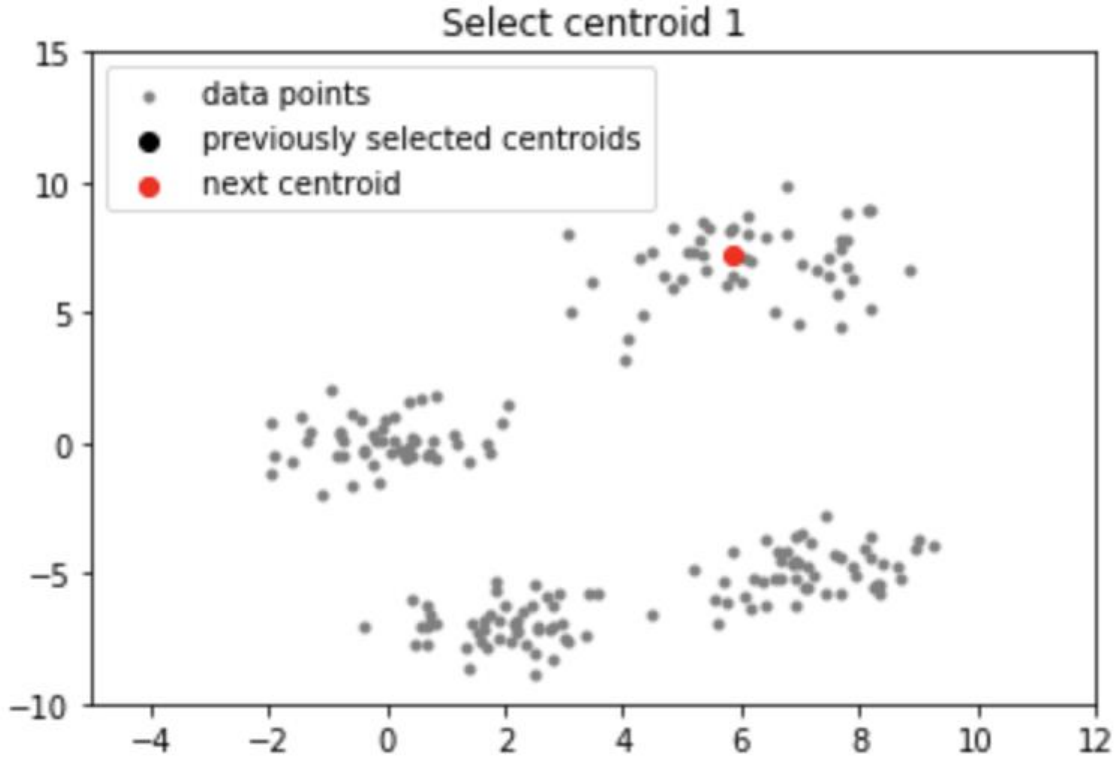
```
-211.59853725816836
```

K-means++

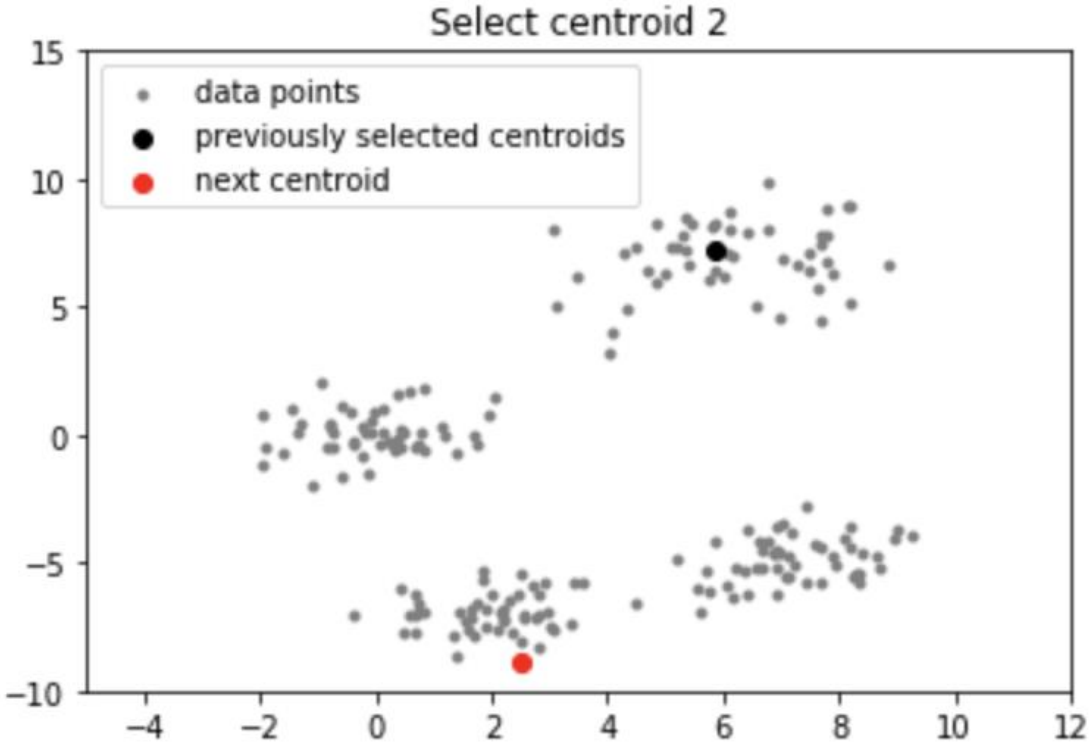
- A smarter initialization step that tends to select centroids that are distant from one another.
  - It makes the k-means algorithm much less likely to converge to a suboptimal solution.

1. Take one centroid $\boldsymbol{c}^{(1)}$, chosen uniformly at random from the dataset.
2. Take a new centroid $\boldsymbol{c}^{(i)}$, choosing an instance $\mathbf{x}^{(i)}$ with probability $\frac{D(\mathbf{x}^{(i)})^2}{\sum_{j=1}^{m} D(\mathbf{x}^{(j)})^2}$ where $D(\mathbf{x}^{(j)})$ is the distance between the instance $\mathbf{x}^{(j)}$ and the closest centroid that was already chosen.
3. Repeat the previous step until all k centroids have been chosen.

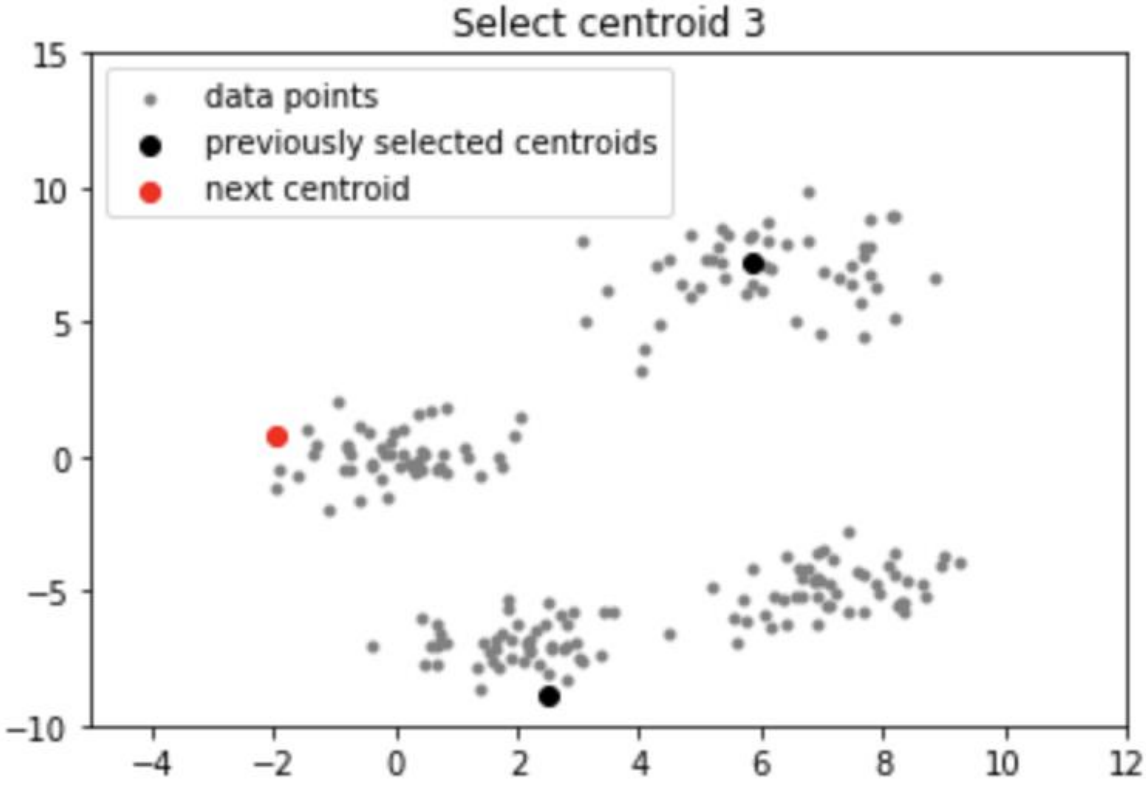
K-means++


Select centroid 1
data points
previously selected centroids
next centroid
15
10
5
0
−5
−10
−4
−2
0
2
4
6
8
10
12

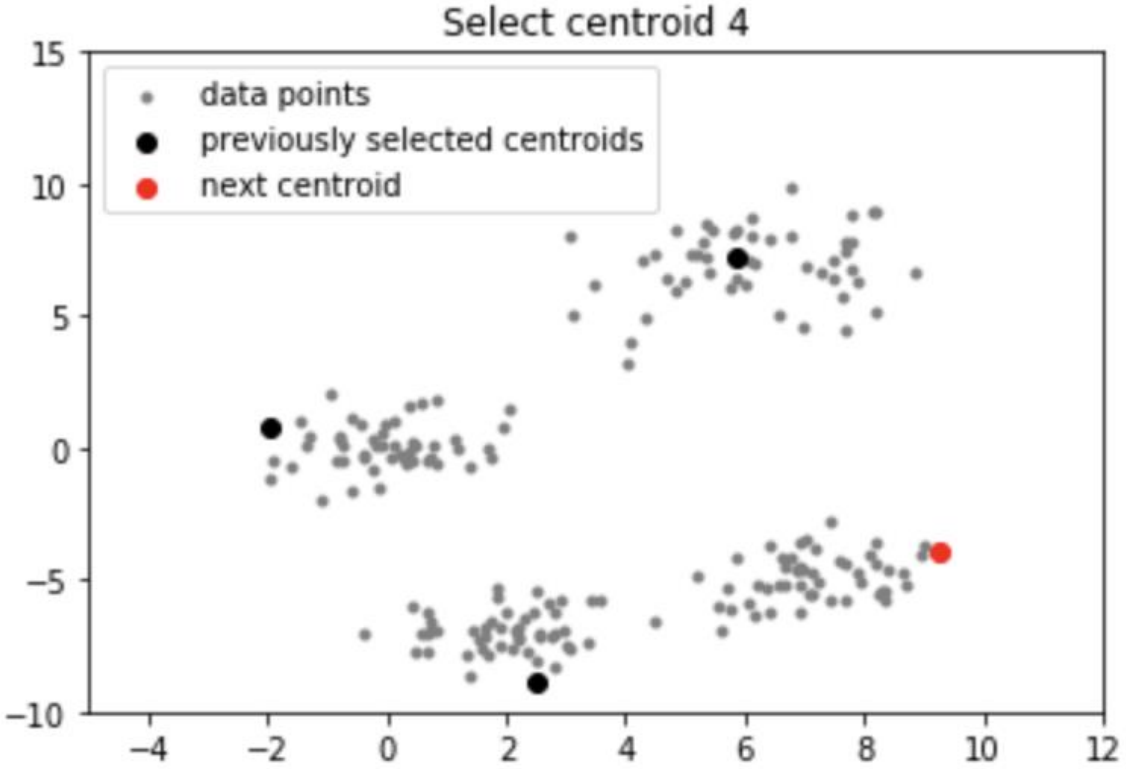
K-means++


Select centroid 2
data points
previously selected centroids
next centroid
15
10
5
0
−5
−10
−4
−2
0
2
4
6
8
10
12

K-means++


Select centroid 3
data points
previously selected centroids
next centroid
15
10
5
0
−5
−10
−4
−2
0
2
4
6
8
10
12

K-means++


Select centroid 4
data points
previously selected centroids
next centroid
15
10
5
0
−5
−10
−4
−2
0
2
4
6
8
10
12

Accelerated k-means

- Another improvement to the *k*-means algorithm was proposed in a 2003 paper by Charles Elkan.
- On some large datasets with many clusters, the algorithm can be accelerated by avoiding many unnecessary distance calculations.
  - Elkan achieved this by exploiting the triangle inequality and by keeping track of lower and upper bounds for distances between instances and centroids.
- Elkan's algorithm does not always accelerate training, and sometimes it can even slow down training significantly;
  - It depends on the dataset.
  - You can try it by setting `algorithm="elkan"`.

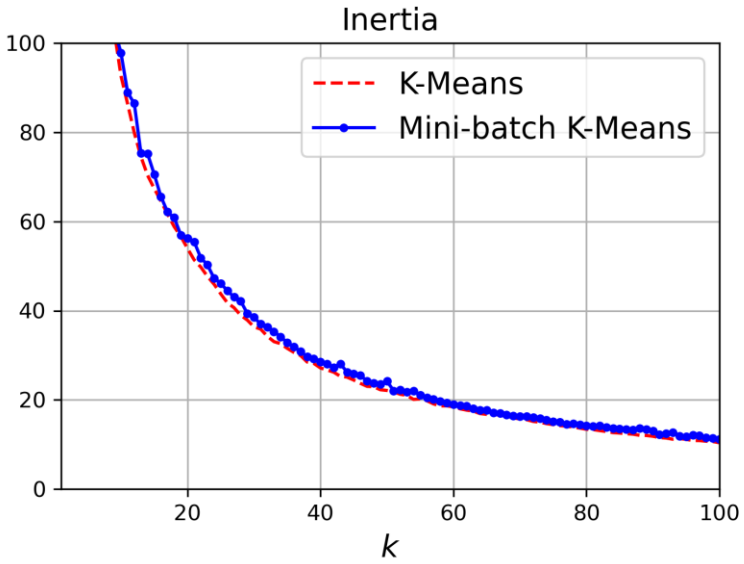
Mini-batch k-means

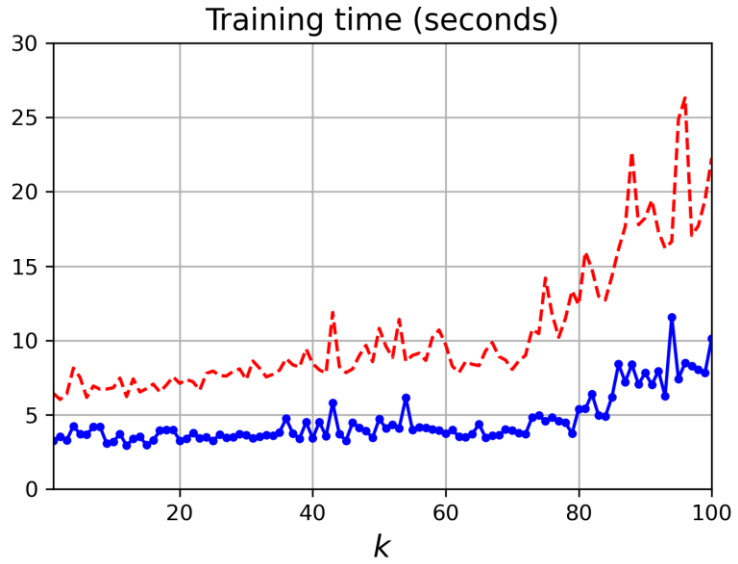
- Mini-batch k-means: instead of using the full dataset at each iteration, the algorithm is capable of using mini-batches, moving the centroids just slightly at each iteration.
- This speeds up the algorithm (typically by a factor of three to four) and makes it possible to cluster huge datasets that do not fit in memory.
- Scikit-Learn implements this algorithm in the `MiniBatchKMeans` class, which you can use just like the `KMeans` class:

```
from sklearn.cluster import MiniBatchKMeans

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

Mini-batch k-means

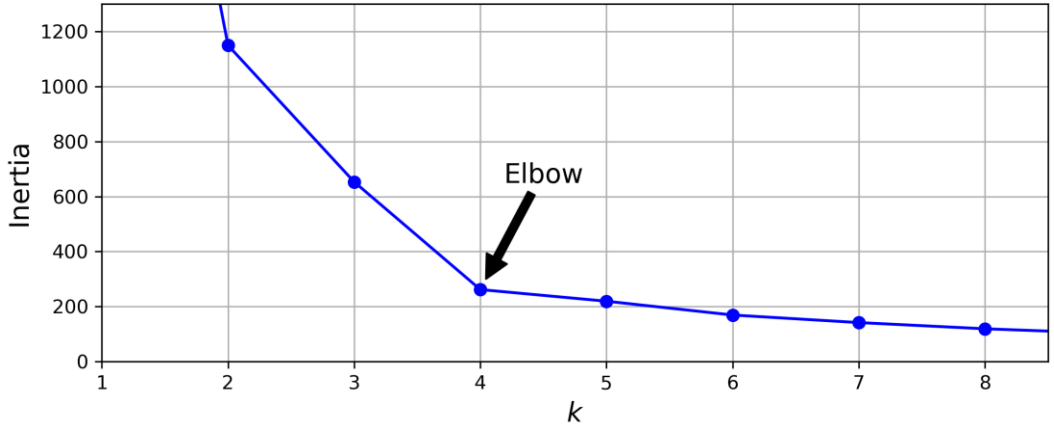
- Although the mini-batch k-means algorithm is much faster than the regular k-means algorithm, its inertia is generally slightly worse.





Finding the optimal number of clusters

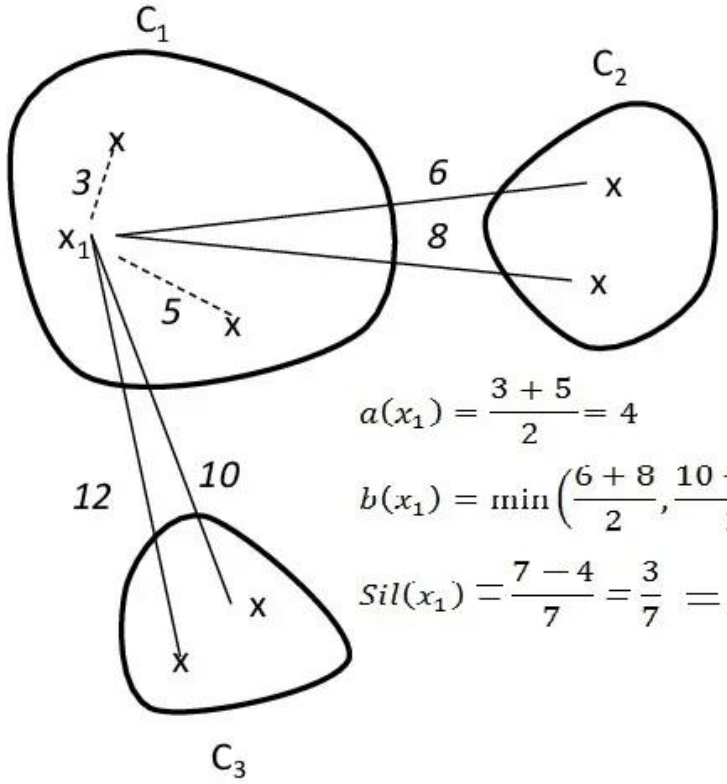
- The inertia is not a good performance metric when trying to choose k because it keeps getting lower as we increase k.
- *Elbow method*: plot the inertia as a function of k, the curve often contains an inflexion point called the *elbow*.



Silhouette Score

- *Silhouette score* is the mean *silhouette coefficient* over all the instances.
- An instance's silhouette coefficient is equal to $\frac{b - a}{\max(a,b)}$
  - a is the mean distance to the other instances in the same cluster (the mean intra-cluster distance).
  - b is the mean nearest-cluster distance.
  - The silhouette coefficient can vary between –1 and +1.
  - A coefficient close to +1 means that the instance is well inside its own cluster and far from other clusters.
  - A coefficient close to 0 means that it is close to a cluster boundary.
  - A coefficient close to –1 means that the instance may have been assigned to the wrong cluster.

Silhouette Coefficient



$$a(x_1) = \frac{3+5}{2} = 4$$

$$b(x_1) = \min\left(\frac{6+8}{2}, \frac{10+12}{2}\right) = 7$$

$$Sil(x_1) = \frac{7-4}{7} = \frac{3}{7} = 0.42$$

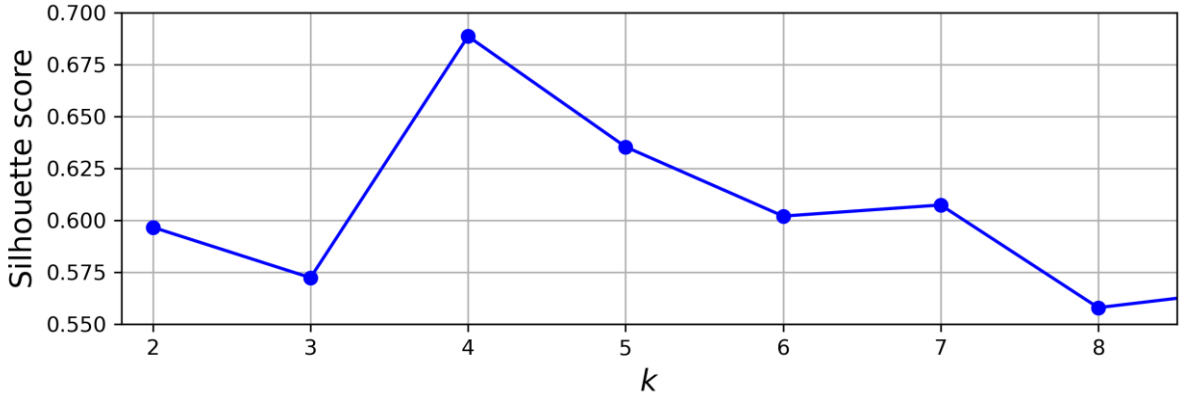
Silhouette Score

- You can use Scikit-Learn's `silhouette_score()` function, giving it all the instances in the dataset and the labels they were assigned:

```
from sklearn.metrics import silhouette_score
```

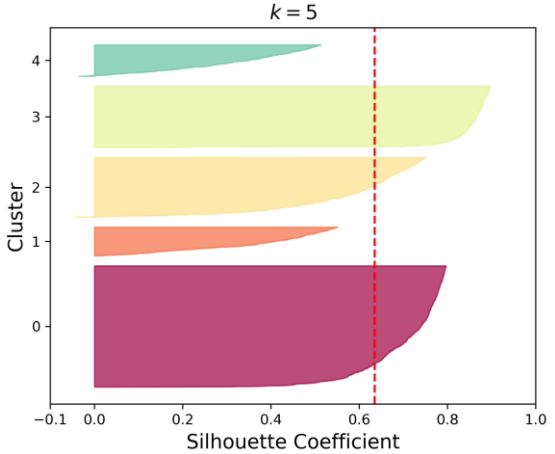
```
silhouette_score(X, kmeans.labels_)
```

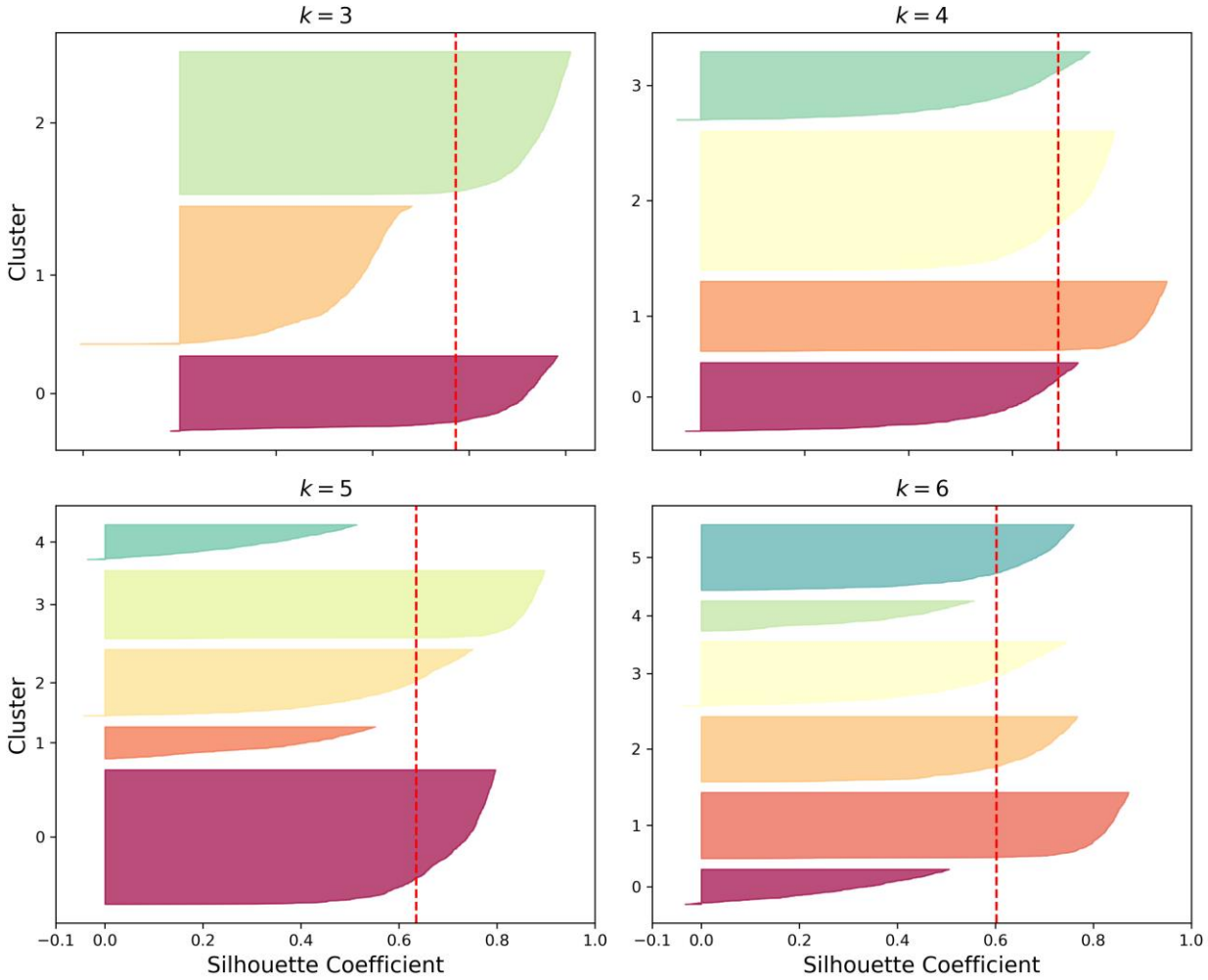
```
0.655517642572828
```



Silhouette Diagram

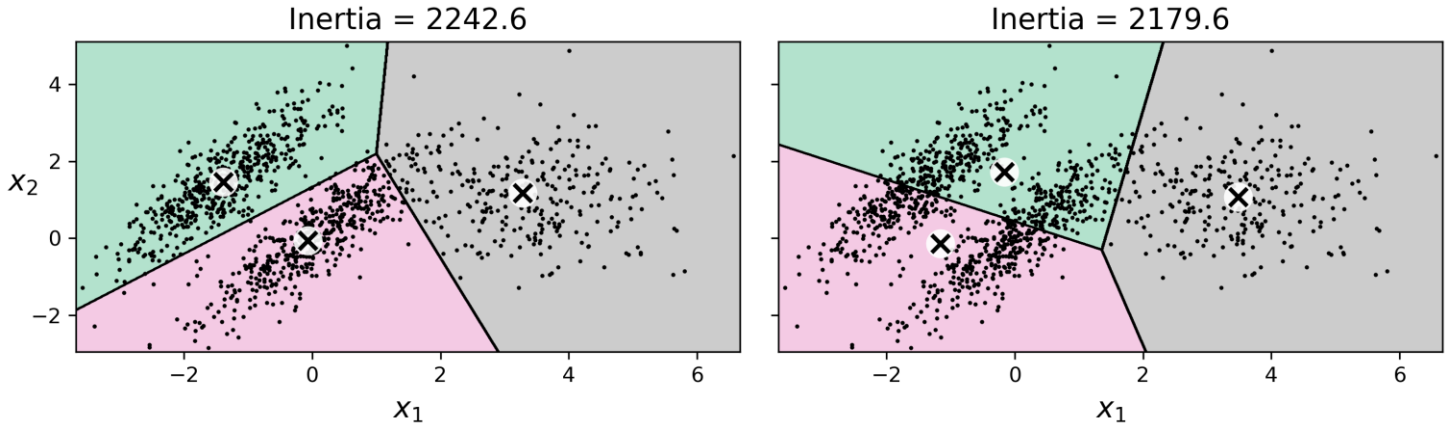
- *Silhouette diagram*: plot every instance's silhouette coefficient, sorted by the clusters they are assigned to and by the value of the coefficient.
- Each diagram contains one knife shape per cluster.
  - The shape's height indicates the number of instances in the cluster
  - Its width represents the sorted silhouette coefficients of the instances in the cluster (wider is better).
- The vertical dashed line is the silhouette score for the clustering.
- When most of the instances in a cluster have a lower coefficient than this score then the cluster is rather bad.




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

Limits of k-means

- We should run k-means several times to avoid suboptimal solutions.
- You must specify the number of clusters, which can be a problem.
- k-means does not behave very well when the clusters have varying sizes, different densities, or nonspherical shapes.



3.
Clustering Applications

Using Clustering for Image Segmentation

- *Image segmentation* is the task of partitioning an image into multiple segments:
  - *Color segmentation*: pixels with a similar color get assigned to the same segment.
  - *Semantic segmentation*: all pixels that are part of the same object type get assigned to the same segment.
  - *Instance segmentation*: all pixels that are part of the same individual object are assigned to the same segment.
- The state of the art in semantic or instance segmentation today is achieved using complex architectures based on convolutional neural networks.

Color Segmentation using k-means

- Import the Pillow package (successor to the Python Imaging Library, PIL), and load the *ladybug.png*, assuming it's located at `filepath`:

```
import PIL

image = np.asarray(PIL.Image.open(filepath))
image.shape
```

```
(533, 800, 3)
```

- The image is represented as a 3D array.
  - The first dimension's size is the height
  - The second is the width
  - The third is the number of color channels: red, green, and blue (RGB).
- For each pixel there is a 3D vector containing the intensities of red, green, and blue as unsigned 8-bit integers between 0 and 255.

Color Segmentation using k-means

- The following code:
  - reshapes the array to get a long list of RGB colors
  - clusters these colors using k-means with eight clusters
  - creates a `segmented_img` array containing the nearest cluster center for each pixel
  - reshapes this array to the original image shape.

```
X = image.reshape(-1, 3)
kmeans = KMeans(n_clusters=8, random_state=42).fit(X)
segmented_img = kmeans.cluster_centers_[kmeans.labels_]
segmented_img = segmented_img.reshape(image.shape)
```

Color Segmentation using k-means

Using Clustering for Semi-Supervised Learning

- In semi-supervised learning, we have plenty of unlabeled instances and very few labeled instances.
- The *digits* dataset is a simple MNIST-like dataset containing 1797 grayscale 8×8 images representing the digits 0 to 9.

```
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 pretend we only have labels for 50 instances. To get a baseline, 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])
```

Using Clustering for Semi-Supervised Learning

- Measure the accuracy of the baseline model on the test set:

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

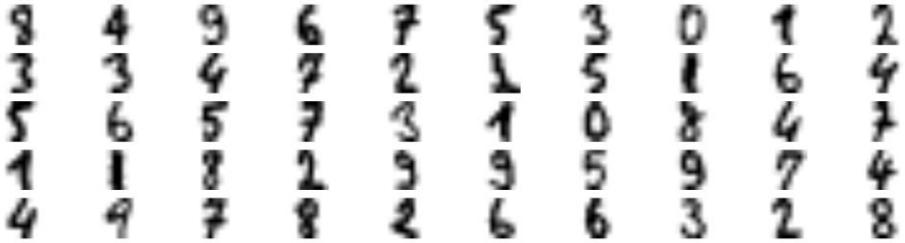
```
0.7481108312342569
```

- First, cluster the training set into 50 clusters.
- Then, for each cluster, 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 = X_digits_dist.argmin(axis=0)
X_representative_digits = X_train[representative_digit_idx]
```

Using Clustering for Semi-Supervised Learning

- The 50 representative images:



- 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])
```

- 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.

Using Clustering for Semi-Supervised Learning

- The performance jumps from 74.8% accuracy to 84.9%:

```
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
```

- *Label propagation:* propagate the labels to all the other instances in the same cluster.

```
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]
```

```
log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_train, y_train_propagated)
```

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

```
0.8942065491183879
```

Using Clustering for Semi-Supervised Learning

- Ignore the 1% of instances that are farthest from their cluster center.
  - This should eliminate some outliers.

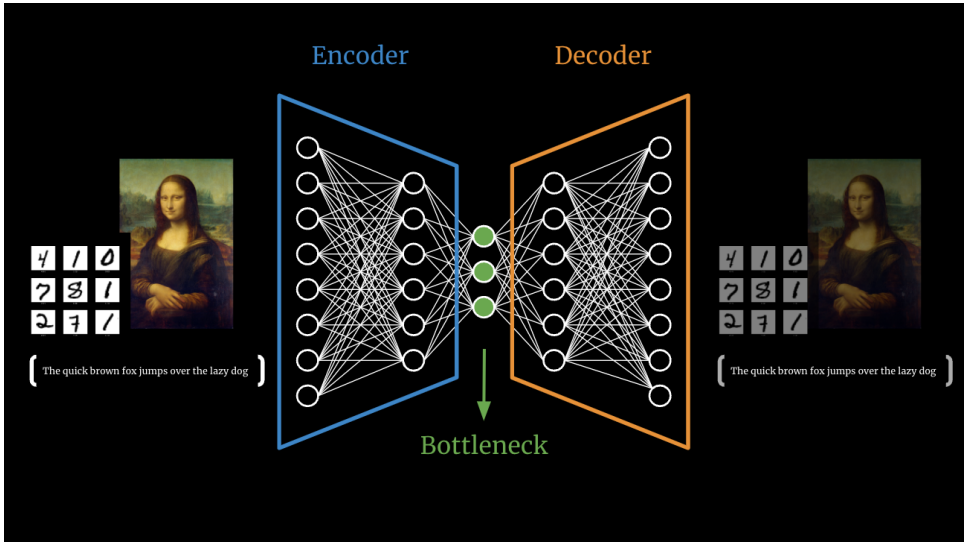
```
percentile_closest = 99

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]
```

```
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
```

Clustering High Dimensional Data

- What if we have a really large size image?
- Can we easily extract useful features?



Active Learning

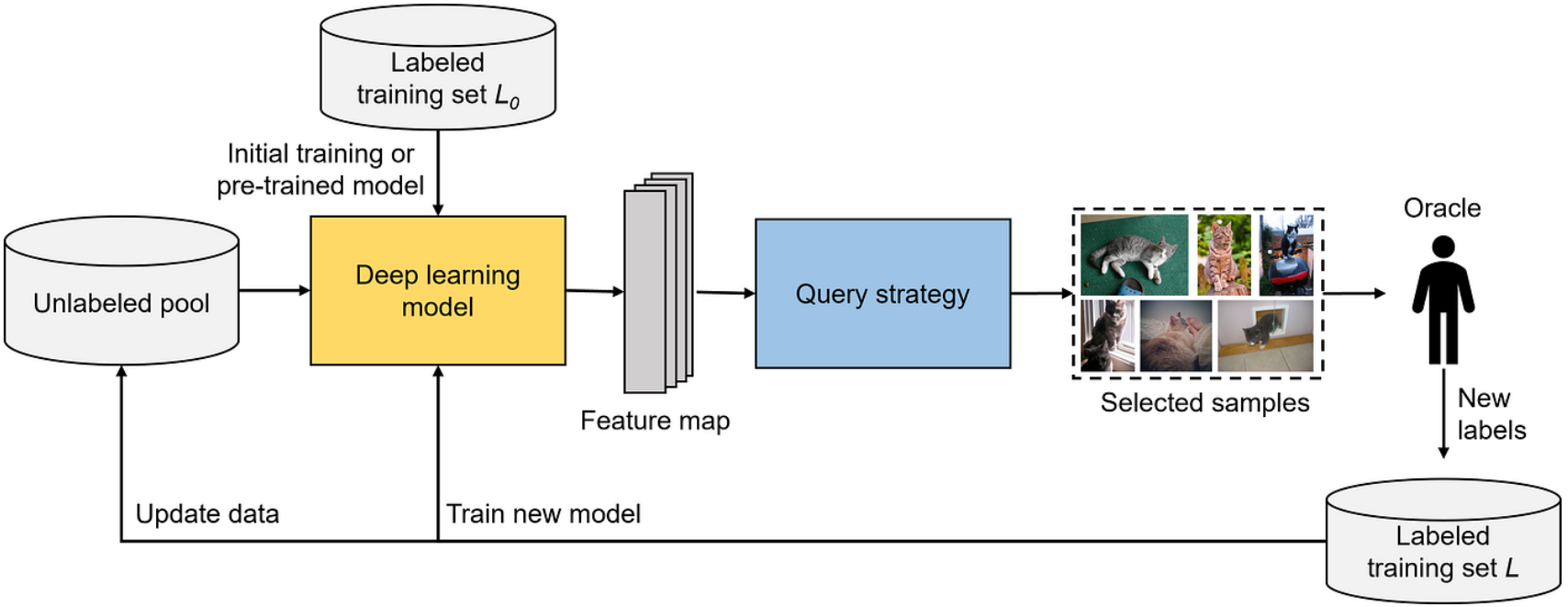


Image source: [How to Train Neural Networks With Fewer Data Using Active Learning | by Dr. Leon Eversberg | Towards AI]

Active Learning

- *Steps*
  - 1-Initial training
  - 2- Inference and selection
  - 3- Human Labeling
  - 4- Retrain the model
  - 5- Repeat steps 2 to 4 until the demanded results are achieved

Active Learning

- *Advantages*
  - Efficiency in data annotation
  - Improved model performance
  - Faster convergence
  - Cost Reduction

Active Learning

- *Active learning*: when a human expert provides labels for specific instances when the learning algorithm requests them.
- *Uncertainty sampling*: the most common strategy for active learning
  1. The model is trained on the labeled instances gathered so far, and this model is used to make predictions on all the unlabeled instances.
  2. The instances for which the model is most uncertain (i.e., where its estimated probability is lowest) are given to the expert for labeling.
  3. You iterate this process until the performance improvement stops being worth the labeling effort.
- Other strategies include labeling the instances that would result in the largest model change or the largest drop in the model's validation error, or the instances that different models disagree on.

4.
DBSCAN

DBSCAN

- The *density-based spatial clustering of applications with noise* algorithm defines clusters as continuous regions of high density:
  1. For each instance, count how many instances are located within a small distance ε from it. This region is called the instance's *ε-neighborhood*.
  2. 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.
  3. 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.
  4. Any instance that is not a core instance and does not have one in its neighborhood is considered an anomaly.

DBSCAN in Scikit-Learn

- DBSCAN works well if all the clusters are well separated by low-density regions.
- Let's test it on the moons dataset:

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

X, y = make_moons(n_samples=1000, noise=0.05, random_state=42)
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
```

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

```
dbscan.labels_[:10]
```

```
array([ 0,  2, -1, -1,  1,  0,  0,  0,  2,  5], dtype=int64)
```

DBSCAN in Scikit-Learn

- The indices of the core instances are in the `core_sample_indices_` instance variable, and the core instances themselves are available in the `components_` instance variable:

```
dbscan.core_sample_indices_[:10]
```

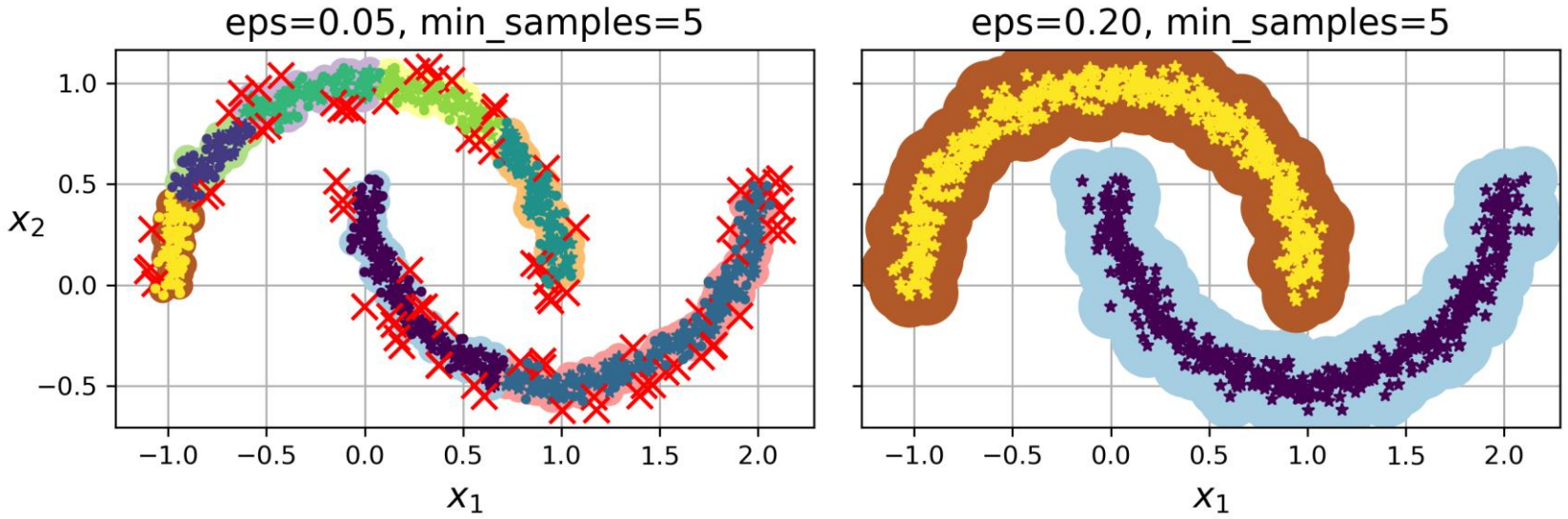
```
array([ 0,  4,  5,  6,  7,  8, 10, 11, 12, 13], dtype=int64)
```

```
dbscan.components_
```

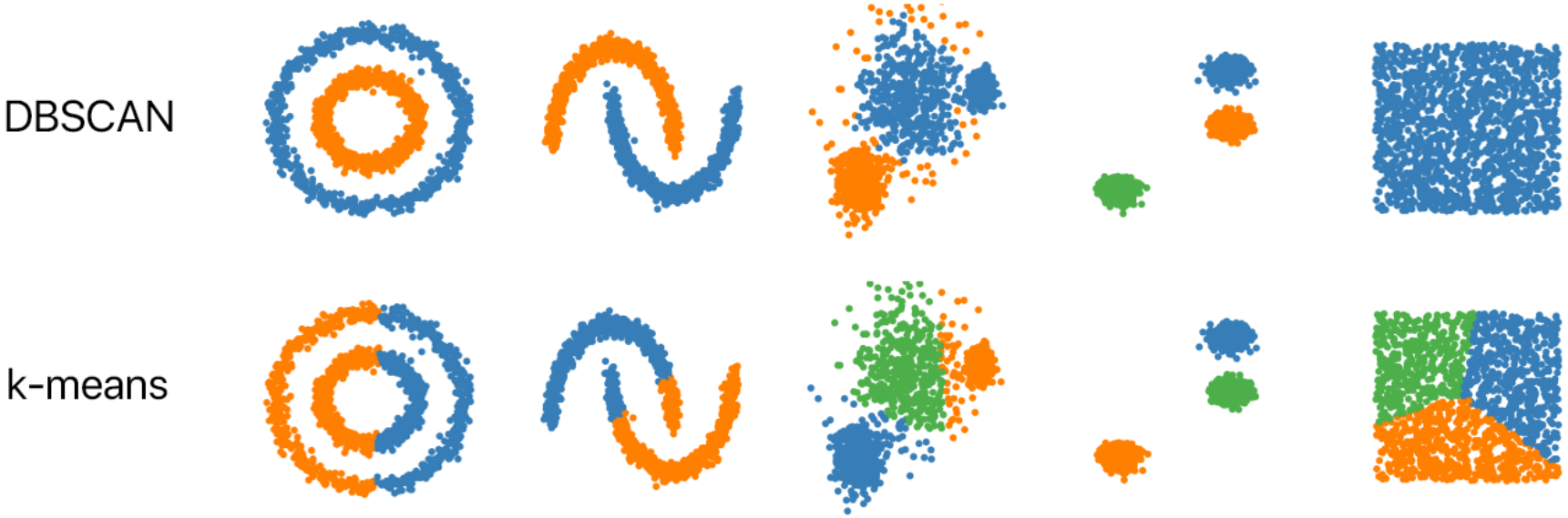
```
array([[-0.02137124,  0.40618608],
       [-0.84192557,  0.53058695],
       [ 0.58930337, -0.32137599],
       ...,
       [ 1.66258462, -0.3079193 ],
       [-0.94355873,  0.3278936 ],
       [ 0.79419406,  0.60777171]])
```

Neighborhood Radius Effect

- DBSCAN clustering using two different neighborhood radiuses



DBSCAN vs. K-means



DBSCAN in Scikit-Learn

- The `DBSCAN` class does not have a `predict()` method, although it has a `fit_predict()` method.
  - It cannot predict which cluster a new instance belongs to.
- But we can implement it (e.g. use a `KNeighborsClassifier`):

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbscan.components_, dbscan.labels_[dbscan.core_sample_indices_])
```

- Now, given a few new instances, we can predict which clusters they most likely belong to and even estimate a probability for each cluster:

```
X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
knn.predict(X_new)
```

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

```
knn.predict_proba(X_new)
```

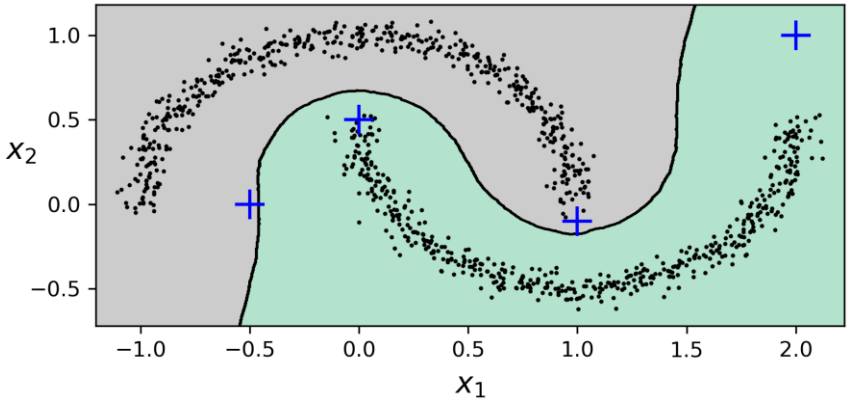
```
array([[0.18, 0.82],
       [1.  , 0.  ],
       [0.12, 0.88],
       [1.  , 0.  ]])
```

DBSCAN in Scikit-Learn

- You can introduce a maximum distance, in case the two instances that are far away from both clusters are classified as anomalies.

```
y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)
y_pred = dbscan.labels_[dbscan.core_sample_indices_][y_pred_idx]
y_pred[y_dist > 0.2] = -1
y_pred.ravel()
```

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

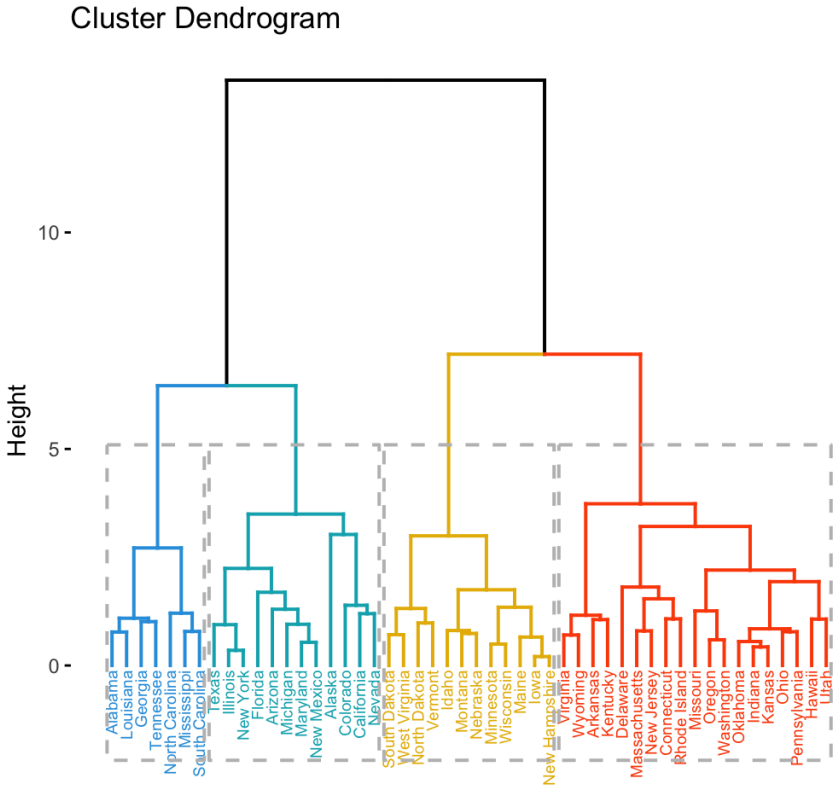


DBSCAN Pros and Cons

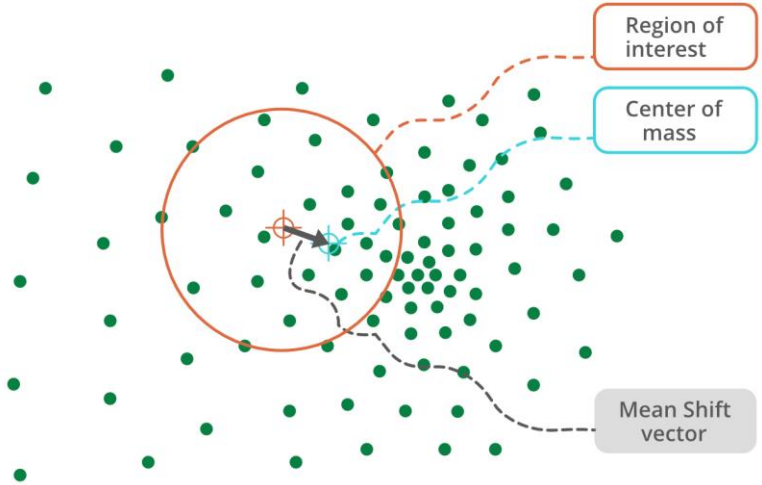
- DBSCAN is a simple yet powerful algorithm capable of identifying any number of clusters of any shape.
- It is robust to outliers, and it has just two hyperparameters (`eps` and `min_samples`).
- If the density varies significantly across the clusters, or if there's no sufficiently low-density region around some clusters, DBSCAN can struggle to capture all the clusters properly.
- Its computational complexity is roughly $O(m^2 n)$, so it does not scale well to large datasets.
- Also try *hierarchical DBSCAN* (HDBSCAN), which is usually better than DBSCAN at finding clusters of varying densities.

Other Clustering Algorithms

- *Agglomerative clustering* builds a hierarchy of clusters from the bottom up. At each iteration, the algorithm connects the nearest pair of clusters (starting with individual instances).
- *BIRCH* is designed specifically for very large datasets, and can be faster than batch k-means, if number of features is not too large (<20).
- *Mean-shift* places a circle centered on each instance; then for each circle it computes the mean of all the instances within it, and shifts the circle so that it is centered on the mean. It iterates this mean-shifting step until all the circles stop moving.
  - Computational complexity is $O(m^2 n)$ → not suited for large datasets.
  - Unlike DBSCAN, it tends to chop clusters into pieces when they have internal density variations.



Agglomerative clustering



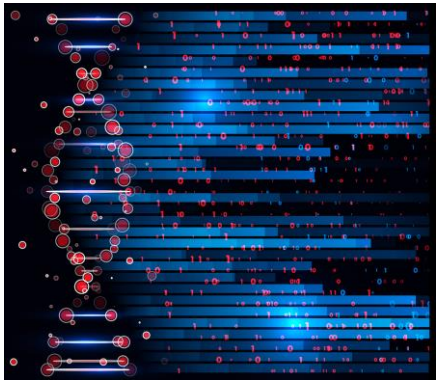
Mean-shift clustering

Hands-on Machine Learning

Dimensionality Reduction

The Curse of Dimensionality

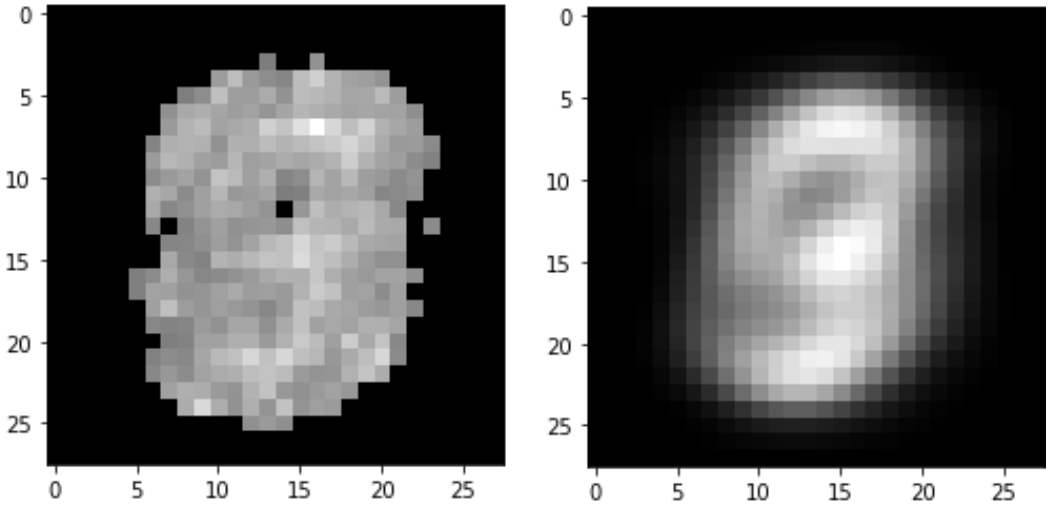
- Many machine learning problems involve thousands or even millions of features for each training instance.



- Not only do all these features make training extremely slow, but they can also make it much harder to find a good solution.
  - This problem is often referred to as the *curse of dimensionality*.

Dimensionality Reduction

- It is possible to reduce the number of features considerably, turning an intractable problem into a tractable one.



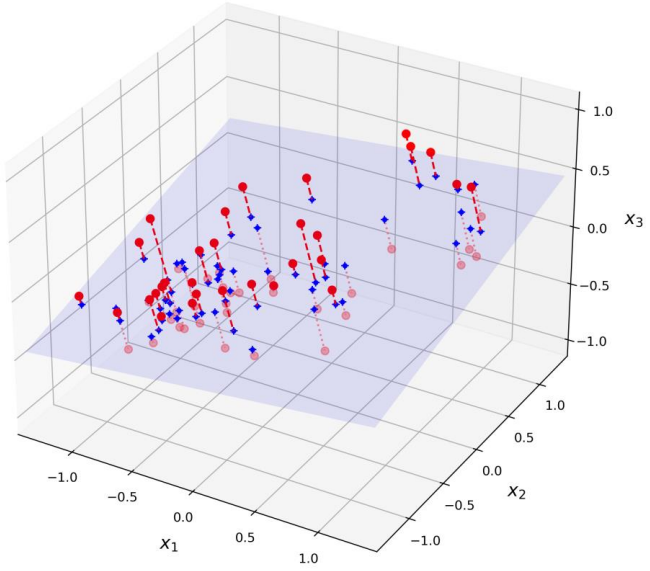
Pros and Cons of Dimensionality Reduction

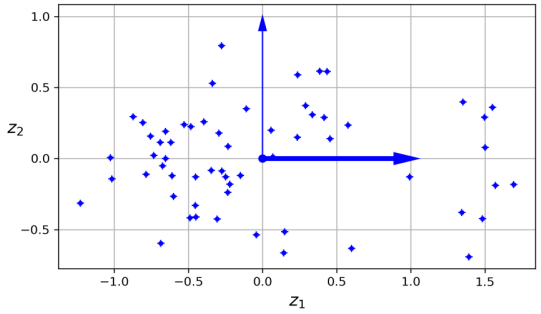
- Advantages:
  - Speed up training
  - May filter out some noise and unnecessary details and result in higher performance in some cases
  - Extremely useful for data visualization
  - Reduced overfitting
- Disadvantages:
  - Information loss and slightly worse performance
  - More complex pipelines
  - Interpretability challenges

1.
Main Approaches for Dimensionality Reduction

Projection

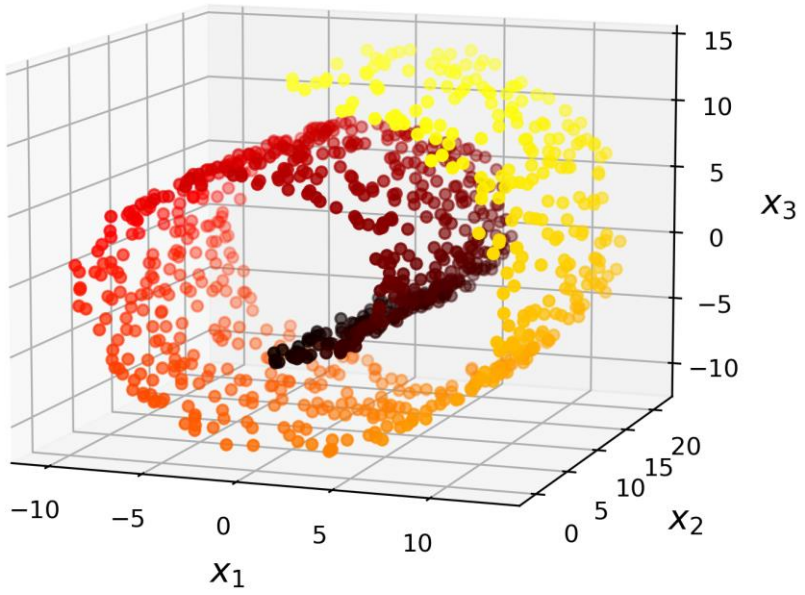
- In most real-world problems, training instances are *not* spread out uniformly across all dimensions.
  - Many features are almost constant.
  - Others are highly correlated.
- All training instances lie within (or close to) a much lower-dimensional *subspace* of the high-dimensional space.



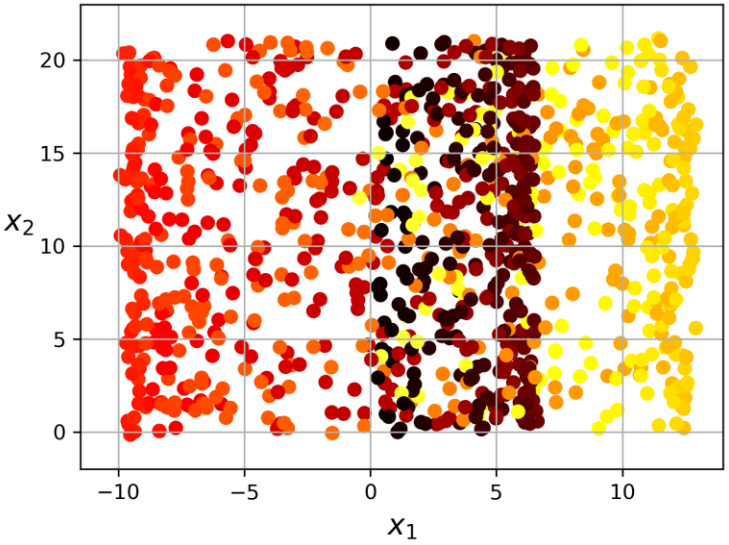


Manifold Learning

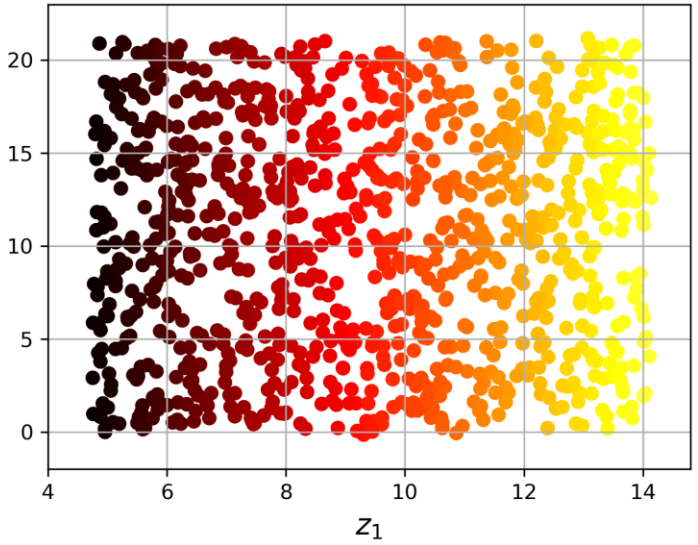
- In many cases the subspace may twist and turn.
- Simply projecting onto a plane (e.g., by dropping x_3) would squash different layers of the Swiss roll together.
- We want to unroll the Swiss roll to obtain the 2D dataset.



Manifold Learning



projecting onto a plane



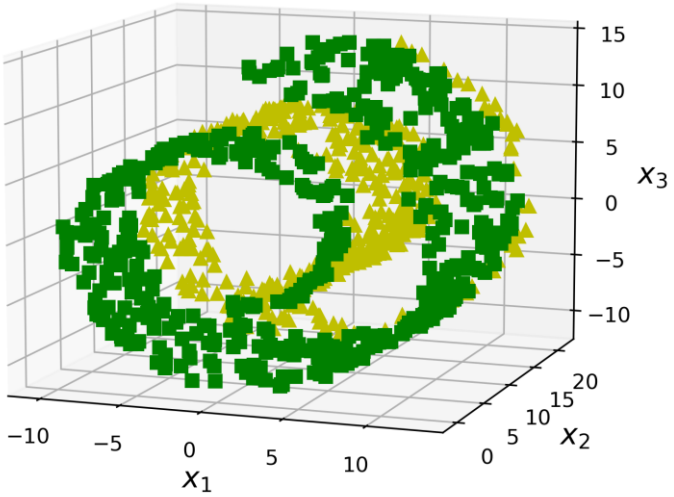
unrolling the Swiss roll

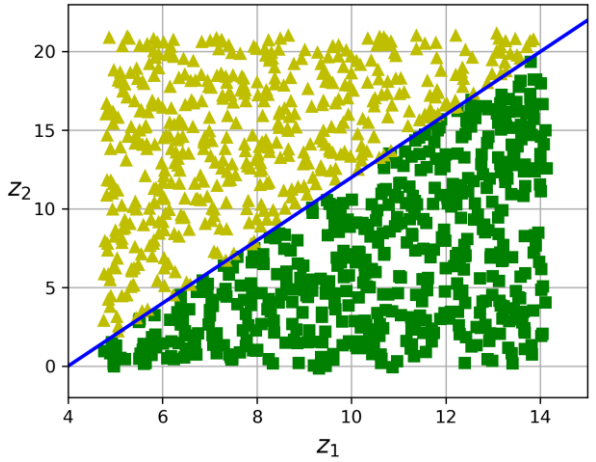
Manifold Learning

- In simple terms, a 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space.
  - The Swiss roll is an example of a 2D *manifold*.
- A d-dimensional manifold is a part of an n-dimensional space ($d < n$) that locally resembles a d-dimensional hyperplane.
- Many dimensionality reduction algorithms work by modeling the manifold on which the training instances lie;
  - This is called *manifold learning*.
- *Manifold assumption*: most real-world high-dimensional datasets lie close to a much lower-dimensional manifold.

Example

- Consider the MNIST dataset: all handwritten digit images have some similarities.
  - They are made of connected lines, the borders are white, and they are more or less centered.
- If you randomly generated images, only a tiny fraction of them would look like handwritten digits.
  - The degrees of freedom available to you if you try to create a digit image are dramatically lower than the degrees of freedom you have if you are allowed to generate any image you want.
- These constraints tend to squeeze the dataset into a lower-dimensional manifold.

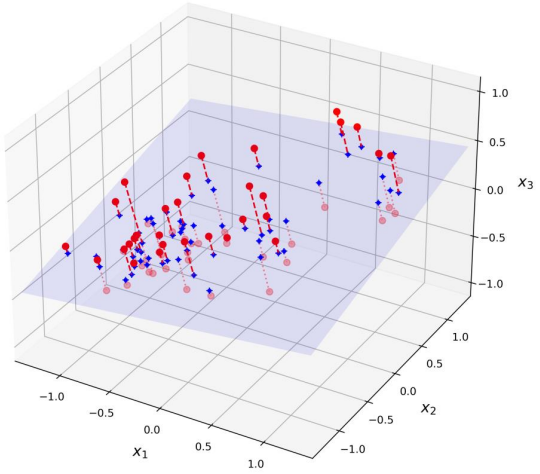

15
10
5
0
−5
−10
x_3
−10
−5
0
5
10
x_1
0
5
10
15
20
x_2


20
15
10
5
0
z_2
4
6
8
10
12
14
z_1

2.
Principal Component Analysis

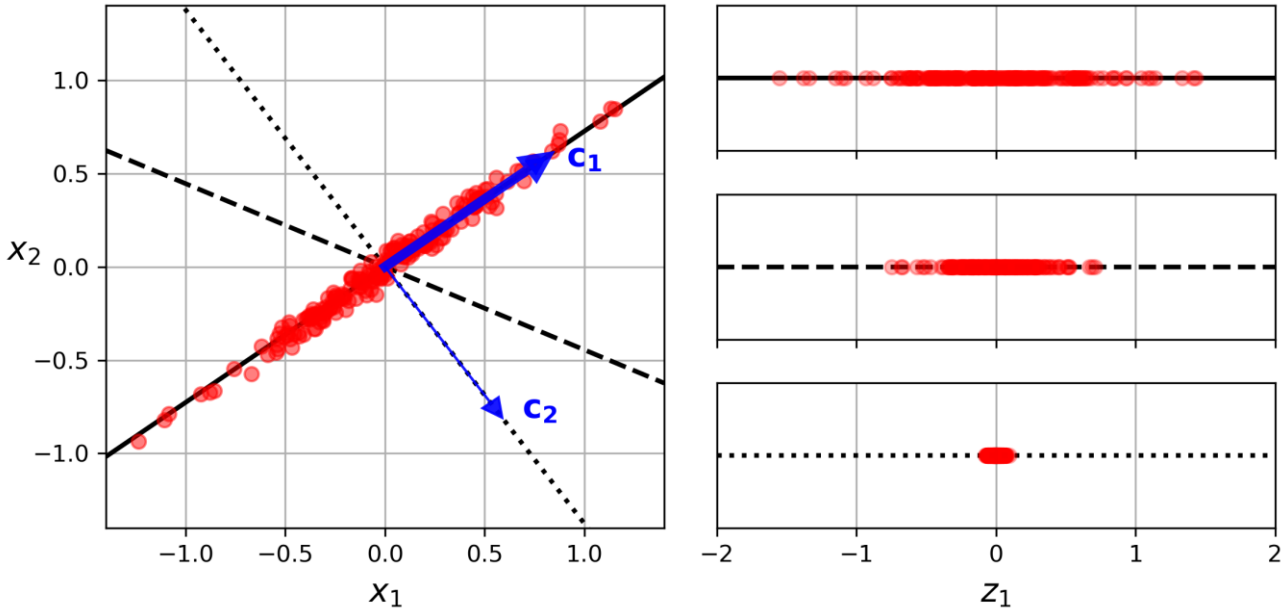
Principal Component Analysis

- *Principal component analysis* (PCA) is the most popular dimensionality reduction algorithm.
- First it identifies the hyperplane that lies closest to the data, and then it projects the data onto it.
- How to choose the right hyperplane?
  - Select the axis that preserves the maximum amount of variance



Preserving the Variance

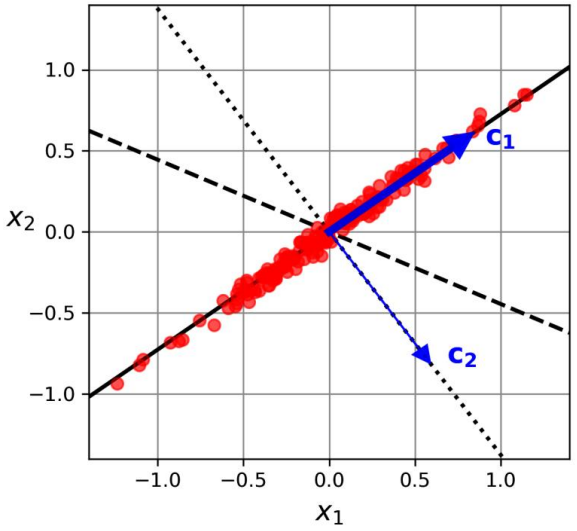
- Select the axis that preserves the maximum amount of variance, as it will lose less information than the other projections



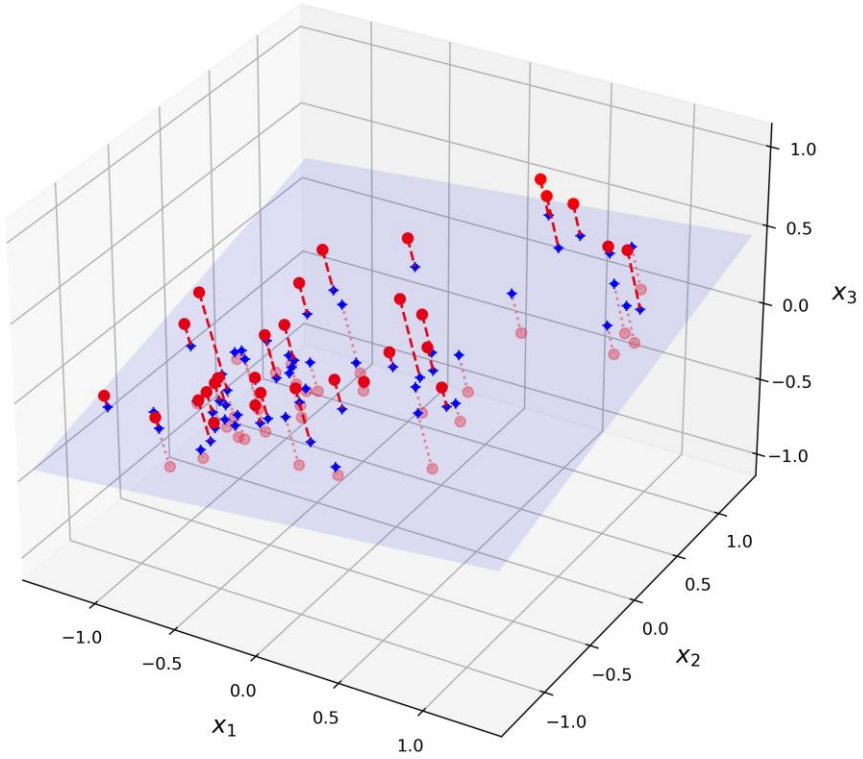
Principal Components

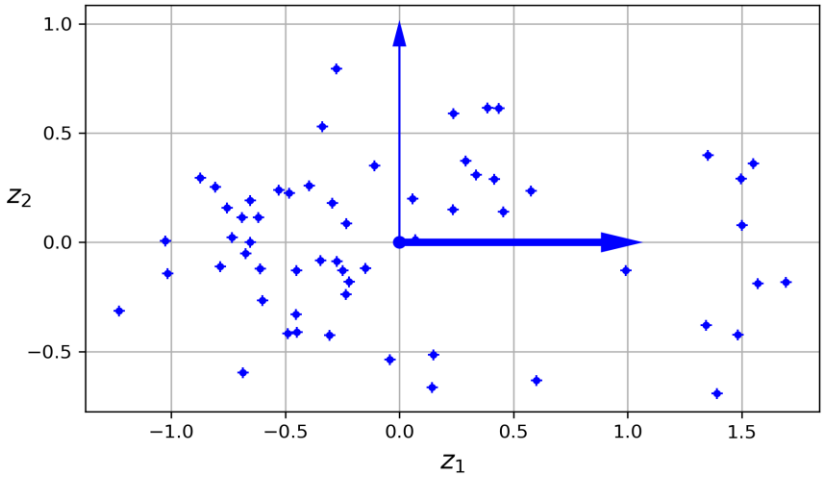
- PCA identifies the axis that accounts for the largest amount of variance in the training set (c_1), then a second axis, orthogonal to the first one (c_2), that accounts for the largest amount of the remaining variance.

- The i-th axis is called the *i-th principal component* (PC) of the data.



Principal Components





Choosing the Right Number of Dimensions

- Choose the number of dimensions that add up to a sufficiently large portion of the variance—say, 95%.

```
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', as_frame=False)
X_train, y_train = mnist.data[:60_000], mnist.target[:60_000]
X_test, y_test = mnist.data[60_000:], mnist.target[60_000:]

pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1  # d equals 154
```

- You could then set `n_components=d` and run PCA again.

Choosing the Right Number of Dimensions

- Instead of specifying the number of principal components you want to preserve, you can set `n_components` to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve:

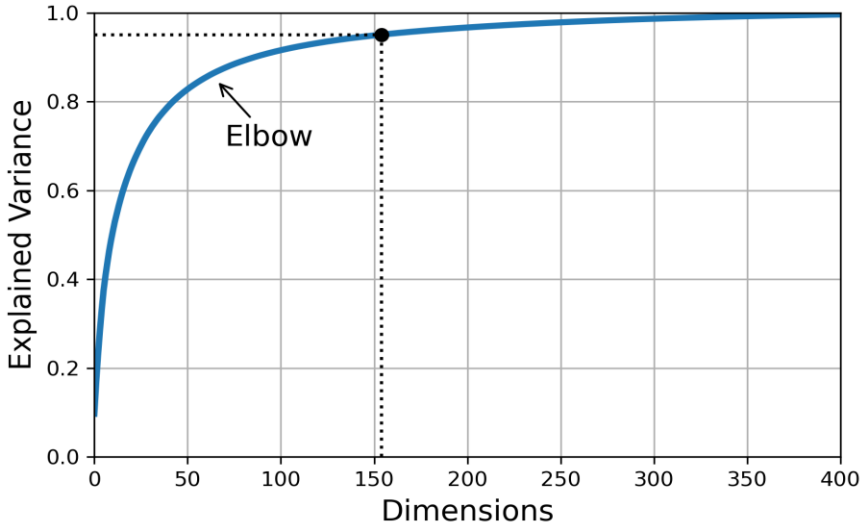
```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

```
pca.n_components_
```

```
154
```

Choosing the Right Number of Dimensions

- Plot the explained variance as a function of the number of dimensions, there will usually be an elbow in the curve, where the explained variance stops growing fast:



Choosing the Right Number of Dimensions

- If you are using dimensionality reduction as a preprocessing step for a supervised learning task, then you can tune the number of dimensions as you would any other hyperparameter.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.pipeline import make_pipeline

clf = make_pipeline(PCA(random_state=42),
                    RandomForestClassifier(random_state=42))
param_distrib = {
    "pca__n_components": np.arange(10, 80),
    "randomforestclassifier__n_estimators": np.arange(50, 500)
}
rnd_search = RandomizedSearchCV(clf, param_distrib, n_iter=10, cv=3, random_state=42)
rnd_search.fit(X_train[:1000], y_train[:1000])
```

```
print(rnd_search.best_params_)
```

```
{'randomforestclassifier__n_estimators': 465, 'pca__n_components': 23}
```

PCA for Compression

- After dimensionality reduction, the training set takes up less space.
  - E.g. MNIST dataset after PCA is less than 20% of its original size, and we only lost 5% of its variance!
- It is possible to decompress the reduced dataset by applying the inverse transformation of the PCA projection.
- The mean squared distance between the original data and the reconstructed data (compressed and then decompressed) is called the *reconstruction error*.
- Use the `inverse_transform`() method to decompress:

```
X_recovered = pca.inverse_transform(X_reduced)
```

PCA for Compression