THE UNIVERSITY OF CHICAGO

Machine Learning and Predictive Analytics, MSCA-31009-4

Lecture 4

Igor Yakushin

April 22, 2021

Overview I

1. Dimensionality Reduction
   - The curse of the dimensionality
   - Main approaches for Dimensionality Reduction
     - Projection
     - Manifold Learning
   - PCA
     - Principal Components
     - Projecting Down to d Dimensions
     - Choosing the Right Number of Dimensions
     - PCA for Compression
     - Randomized PCA
     - Incremental PCA
     - Kernel PCA
     - Selecting a Kernel and Tuning Hyperparameters
   - LLE
   - Other Dimensionality Reduction Techniques
2. Unsupervised Learning
   - Clustering

Overview II

- K-Means
- Using Clustering for Image Segmentation
- Using Clustering for Preprocessing
- Using Clustering for Semi-Supervised Learning
  - Active Learning
- DBSCAN
- Other clustering algorithms

Dimensionality Reduction

Dimensionality Reduction

- 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.
- It is often possible to reduce the number of features considerably
- For example, consider the MNIST images: the pixels on the image borders are almost always white, so you could completely drop these pixels from the training set without losing much information.
- Additionally, two neighboring pixels are often highly correlated: if you merge them into a single pixel (e.g., by taking the mean of the two pixel intensities), you will not lose much information.
- Apart from speeding up training, dimensionality reduction is also extremely useful for data visualization

The curse of the dimensionality

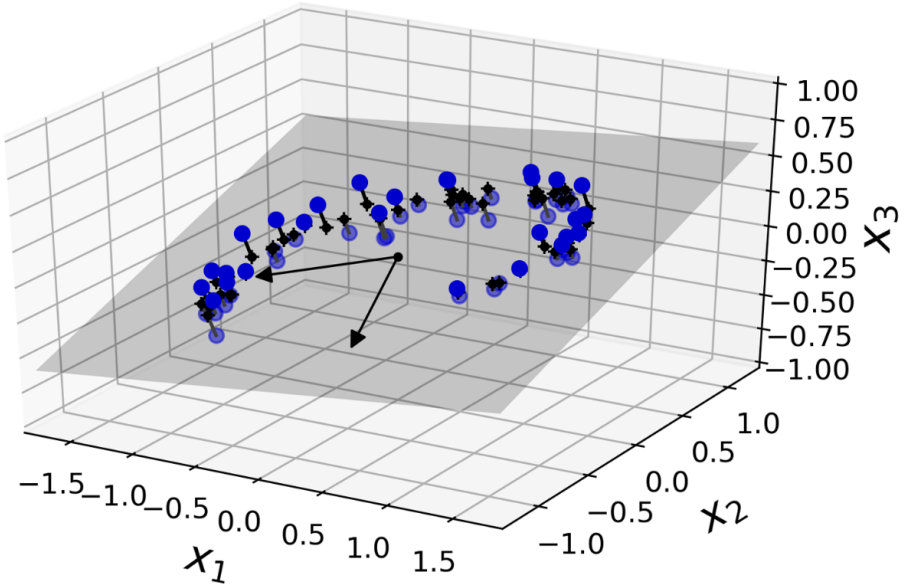
- It turns out that many things behave very differently in high-dimensional space.
- For example, if you pick a random point in a unit square (a 1×1 square), it will have only about a 0.4% chance of being located less than 0.001 from a border
- But in a 10,000-dimensional unit hypercube, this probability is greater than 99.999999%
- Most points in a high-dimensional hypercube are very close to the border
- If you pick two points randomly in a unit square, the distance between these two points will be, on average, roughly 0.52
- If you pick two random points in a unit 3D cube, the average distance will be roughly 0.66
- In a 1,000,000-dimensional hypercube the average distance between the points is about 408.25

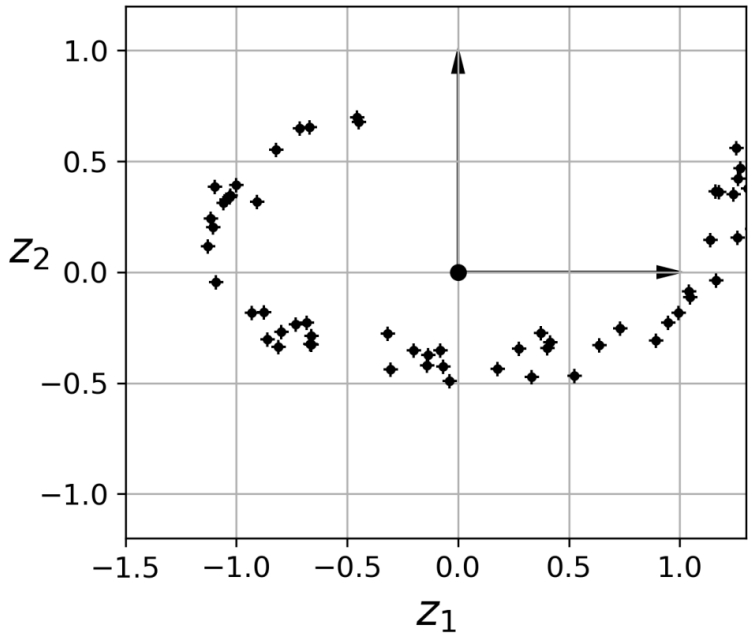
The curse of the dimensionality

- As a result, high-dimensional datasets are at risk of being very sparse: most training instances are likely to be far away from each other.
- This also means that a new instance will likely be far away from any training instance, making predictions much less reliable than in lower dimensions, since they will be based on much larger extrapolations.
- In short, the more dimensions the training set has, the greater the risk of overfitting it.

Main approaches for Dimensionality Reduction: projection

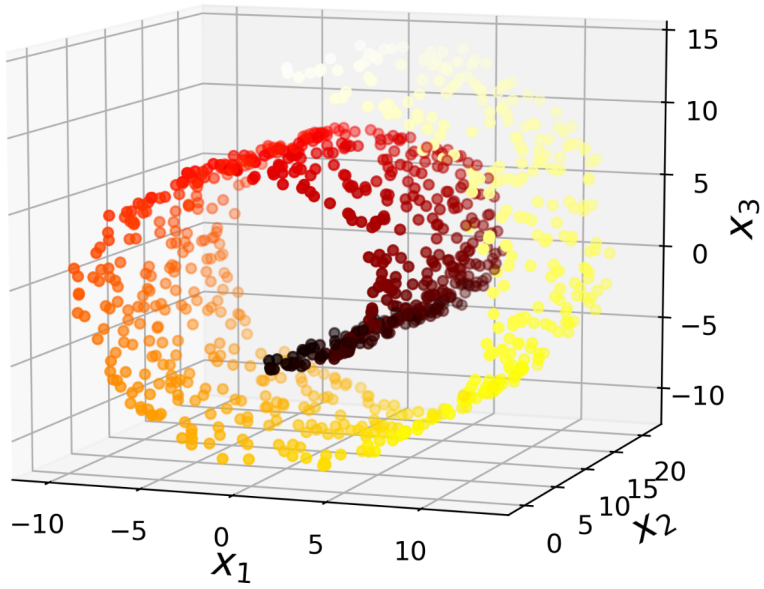
- In most real-world problems, training instances are not spread out uniformly across all dimensions. Many features are almost constant, while others are highly correlated. As a result, all training instances lie within (or close to) a much lower-dimensional subspace of the high-dimensional space and it might be possible to project them to this subspace

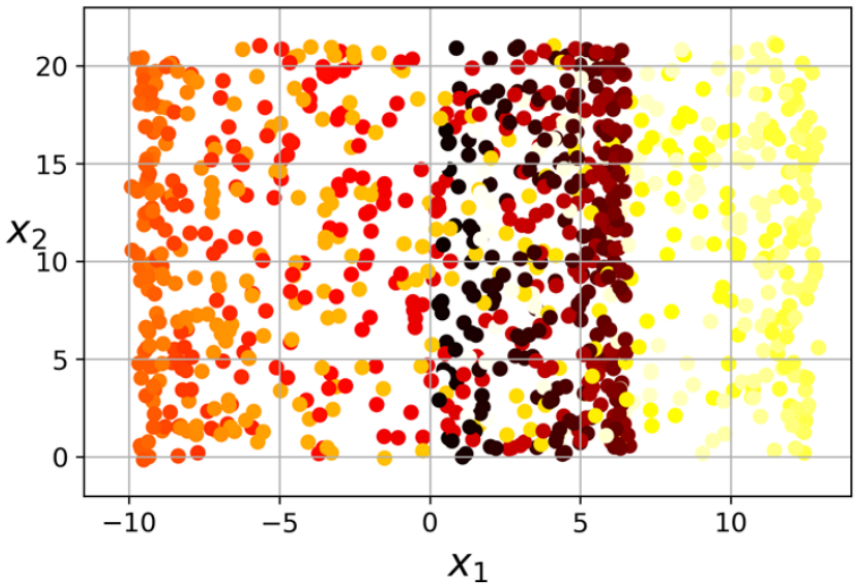


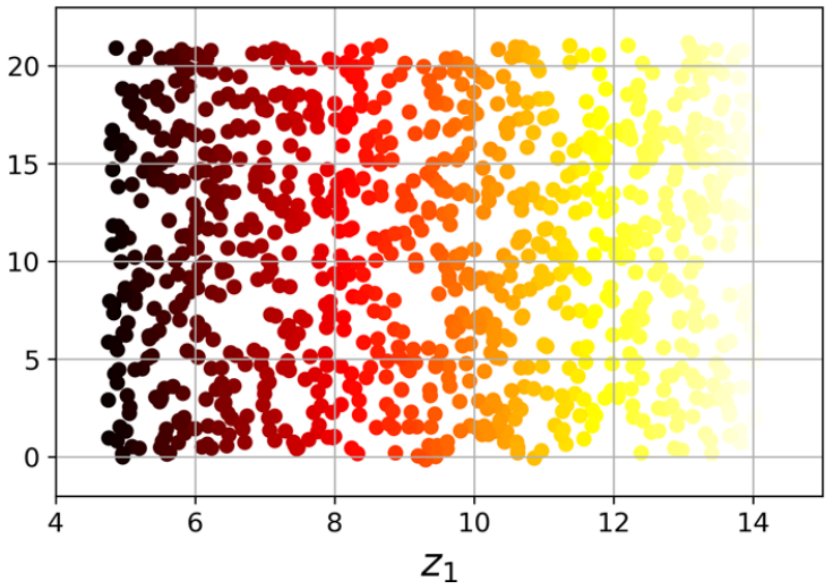


Main approaches for Dimensionality Reduction: projection

- However, projection is not always the best approach to dimensionality reduction. In many cases the subspace may twist and turn, such as in the famous Swiss roll toy dataset







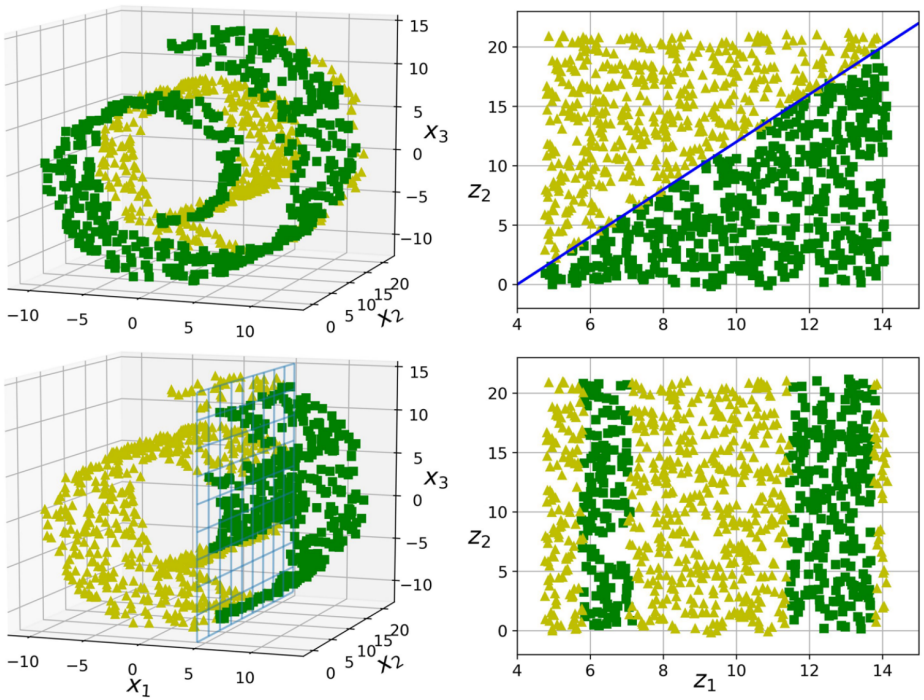
Main approaches for Dimensionality Reduction: Manifold Learning

- The Swiss roll is an example of a 2D manifold - a 2D shape that can be bent and twisted in a higher-dimensional space.
- More generally, a d-dimensional manifold is a part of an n-dimensional space (where $d < n$) that locally resembles a d-dimensional hyperplane.
- In the case of the Swiss roll, $d = 2$ and $n = 3$: it locally resembles a 2D plane, but it is rolled in the third dimension.
- Many dimensionality reduction algorithms work by modeling the manifold on which the training instances lie; this is called Manifold Learning.
- It relies on the manifold assumption, also called the manifold hypothesis, which holds that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold.

Main approaches for Dimensionality Reduction: Manifold Learning

- 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 would have if you were allowed to generate any image you wanted.
- These constraints tend to squeeze the dataset into a lower-dimensional manifold.

Main approaches for Dimensionality Reduction: Manifold Learning

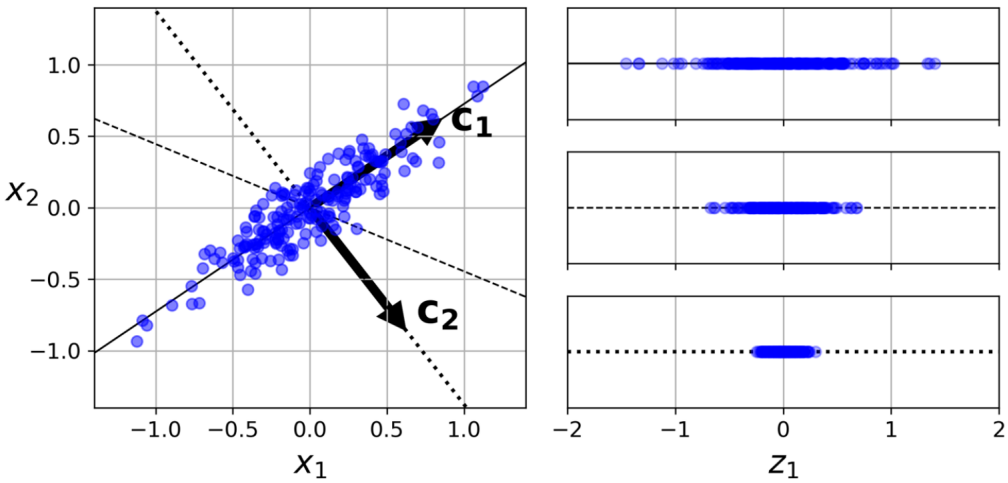


- The manifold assumption is often accompanied by another implicit assumption: that the task at hand will be simpler if expressed in the lower-dimensional space of the manifold.
- Not always the case

- Reducing the dimensionality of your training set before training a model will usually speed up training, but it may not always lead to a better or simpler solution; it all depends on the dataset.

PCA

- Principal Component Analysis (PCA) is by far the most popular dimensionality reduction algorithm.



- It identifies the hyperplane that lies closest to the data, and then it projects the data onto it
- The projection onto the solid line preserves the maximum variance, while the projection onto the dotted line preserves very little variance

- It seems reasonable to select the axis that preserves the maximum amount of variance, as it will most likely lose less information than the other projections.
- Also this axis minimizes the mean squared distance between the original dataset and its projection onto that axis.

Principal Components

- PCA identifies the axis that accounts for the largest amount of variance in the train- ing set.
- Then the next one and so on.
- The ith axis is called the ith principal component (PC) of the data
- For each principal component, PCA finds a zero-centered unit vector pointing in the direction of the PC.
- Since two opposing unit vectors lie on the same axis, the direction of the unit vectors returned by PCA is not stable: if you perturb the training set slightly and run PCA again, the unit vectors may point in the opposite direction as the original vectors. However, they will generally still lie on the same axes.
- In some cases, a pair of unit vectors may even rotate or swap (if the variances along these two axes are close), but the plane they define will generally remain the same.

Principal Components

- How can you find the principal components of a training set?
- There is a standard matrix factorization technique called Singular Value Decomposition (SVD) that can decompose the training set matrix $\mathbf{X}$ into the matrix multiplication of three matrices: $\mathbf{U\Sigma V^T}$ where $\mathbf{V}$ contains the unit vectors that define the principal componenets in the order of their importance:

$$\mathbf{V} = \begin{pmatrix} | & | & & | \\ c_1 & c_2 & \dots & c_n \\ | & | & & | \end{pmatrix} \tag{1}$$

Principal Components

- The following Python code uses NumPy's svd() function to obtain all the principal components of the training set, then extracts the two unit vectors that define the first two PCs:

```
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
```

- Cells 3-7, l4_n1.ipynb
- PCA assumes that the dataset is centered around the origin.
- As we will see, Scikit-Learn's PCA classes take care of centering the data for you.

Projecting Down to d Dimensions

- Once you have identified all the principal components, you can reduce the dimensionality of the dataset down to d dimensions by projecting it onto the hyperplane defined by the first d principal components.
- Selecting this hyperplane ensures that the projection will preserve as much variance as possible.
- To project the training set onto the hyperplane and obtain a reduced dataset $\mathbf{X}_{d\text{-}proj}$ of the dimensionality d, compute the matrix multiplication of the training set matrix $\mathbf{X}$ by the matrix $\mathbf{W}_d$, defined as the matrix containing the first d columns of $\mathbf{V}$:

$$\mathbf{X}_{d\text{-}proj} = \mathbf{X}\mathbf{W}_d \tag{2}$$

Projecting Down to d Dimensions

- The following Python code projects the training set onto the plane defined by the first two principal components:

```
W2 = Vt.T[:, :2]
X2D = X_centered.dot(W2)
```

- If you use Scikit-Learn:

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
```

- `pca.explained_variance_ratio_` contains explained variance ratio of each principal component
- Cells 8-11, `l4_n1.ipynb`

Choosing the Right Number of Dimensions

- Instead of arbitrarily choosing the number of dimensions to reduce down to, it is simpler to choose the number of dimensions that add up to a sufficiently large portion of the variance
- You can do it as follows:

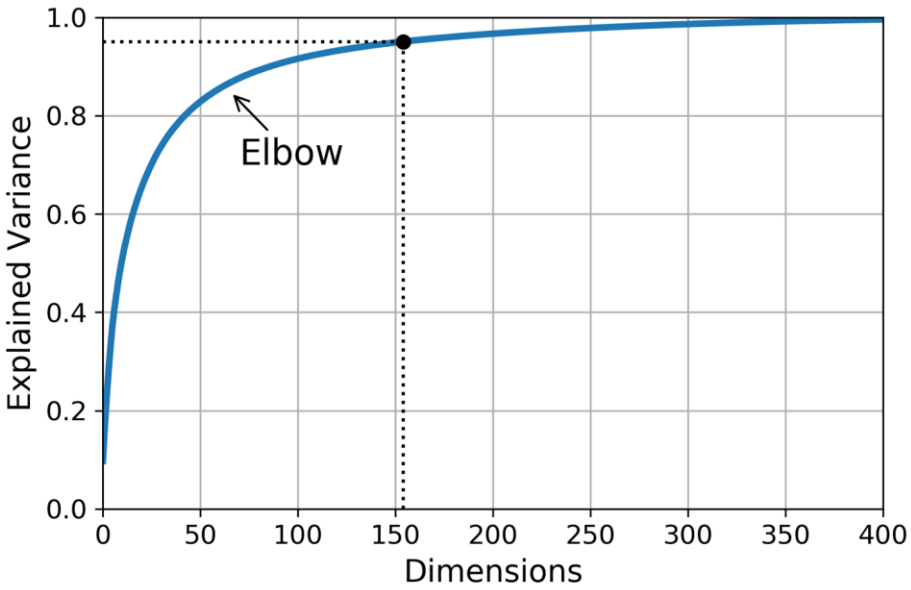
```
pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
pca = PCA(n_components=d)
X_reduced = pca.fit_transform(X_train)
```

- Or even simpler:

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

Choosing the Right Number of Dimensions

- Yet another option is to 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. In this case, you can see that reducing the dimensionality down to about 100 dimensions wouldn't lose too much explained variance



PCA for Compression

- After dimensionality reduction, the training set takes up much less space.
- As an example, try applying PCA to the MNIST dataset while preserving 95% of its var- iance.
- You should find that each instance will have just over 150 features, instead of the original 784 features.
- So, while most of the variance is preserved, the dataset is now less than 20% of its original size!
- This is a reasonable compression ratio, and you can see how this size reduction can speed up a classification algorithm (such as an SVM classifier) tremendously.
- Cells 31-42, `l4_n1.ipynb`

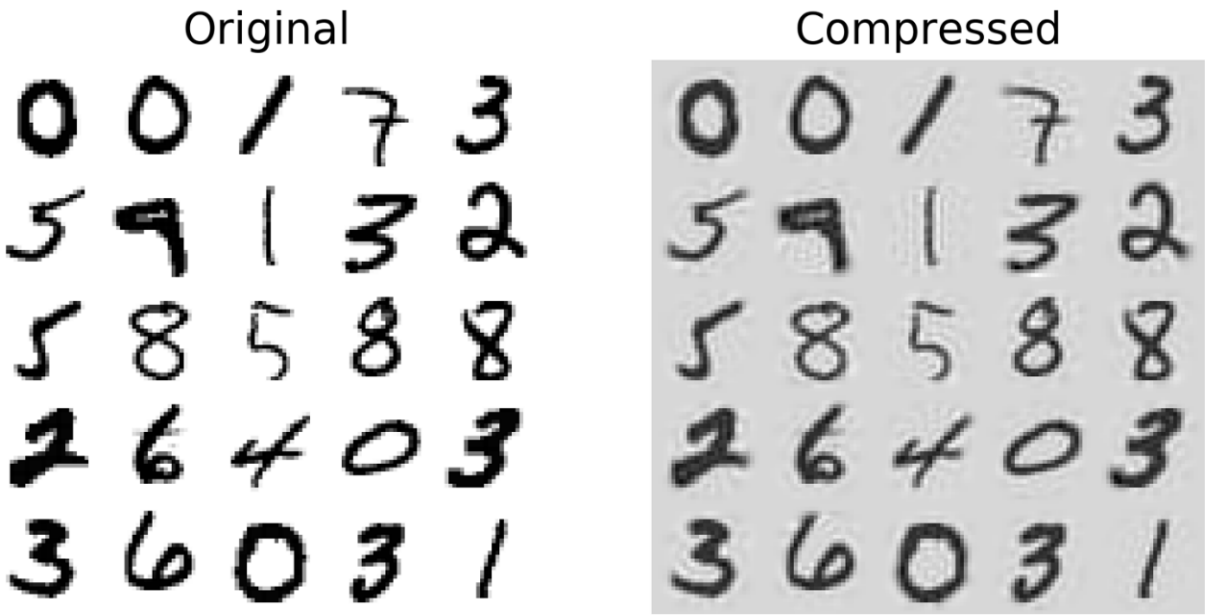
PCA for Compression

- It is also possible to decompress the reduced dataset back to 784 dimensions by applying the inverse transformation of the PCA projection. This won't give you back the original data, since the projection lost a bit of information (within the 5% variance that was dropped), but it will likely be close to the original data. The mean squared distance between the original data and the reconstructed data (compressed and then decompressed) is called the reconstruction error.
- The following code compresses the MNIST dataset down to 154 dimensions, then uses the `inverse_transform()` method to decompress it back to 784 dimensions:

```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

PCA for Compression

- You can see that there is a slight image quality loss, but the digits are still mostly intact:



- The equation of the inverse transformation:

$$\mathbf{X}_{recovered} = \mathbf{X}_{d\text{-}proj}\mathbf{W}_d^T \tag{3}$$

Randomized PCA

- If you set the `svd_solver` hyperparameter to "randomized" , Scikit-Learn uses a stochastic algorithm called Randomized PCA that quickly finds an approximation of the first d principal components.
- Its computational complexity is $O(m \times d^2) + O(d^3)$, instead of $O(m \times n^2) + O(n^3)$ for the full SVD approach, so it is dramatically faster than full SVD when d is much smaller than n:

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_train)
```

- By default, `svd_solver` is actually set to "auto" : Scikit-Learn automatically uses the randomized PCA algorithm if m or n is greater than 500 and d is less than 80% of m or n, or else it uses the full SVD approach.
- If you want to force Scikit-Learn to use full SVD, you can set the `svd_solver` hyperparameter to "full" .

Incremental PCA

- One problem with the preceding implementations of PCA is that they require the whole training set to fit in memory in order for the algorithm to run.
- Incremental PCA (IPCA) algorithms have been developed.
- They allow you to split the training set into mini-batches and feed an IPCA algorithm one mini-batch at a time.
- This is useful for large training sets and for applying PCA online.
- Cells 43-48, `l4_n1.ipynb`

```
from sklearn.decomposition import IncrementalPCA
n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
inc_pca.partial_fit(X_batch)
X_reduced = inc_pca.transform(X_train)
```

Incremental PCA

- Alternatively, you can use NumPy's memmap class, which allows you to manipulate a large array stored in a binary file on disk as if it were entirely in memory; the class loads only the data it needs in memory, when it needs it.
- Since the IncrementalPCA class uses only a small part of the array at any given time, the memory usage remains under control.
- This makes it possible to call the usual `fit()` method:

```
X_mm = np.memmap(filename, dtype="float32",
                 mode="readonly", shape=(m, n))
batch_size = m // n_batches
inc_pca = IncrementalPCA(n_components=154,
                         batch_size=batch_size)
inc_pca.fit(X_mm)
```

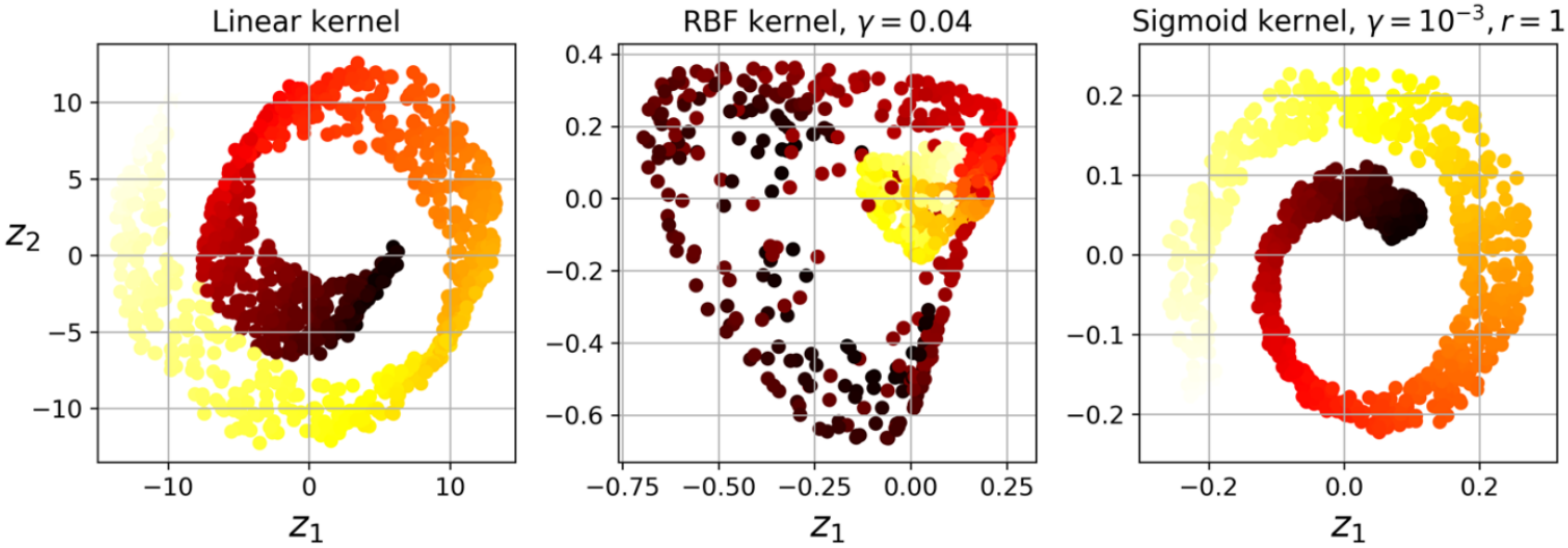
Cells 49-52, `l4_n1.ipynb`

Kernel PCA

- In last lecture we discussed the kernel trick, a mathematical technique that implicitly maps instances into a very high-dimensional space (called the feature space), enabling nonlinear classification and regression with Support Vector Machines.
- Recall that a linear decision boundary in the high-dimensional feature space corresponds to a complex nonlinear decision boundary in the original space.
- It turns out that the same trick can be applied to PCA, making it possible to perform complex nonlinear projections for dimensionality reduction. This is called Kernel PCA (kPCA).
- It is often good at preserving clusters of instances after projection, or sometimes even unrolling datasets that lie close to a twisted manifold.

```
from sklearn.decomposition import KernelPCA
rbf_pca = KernelPCA(n_components = 2, kernel="rbf",
                    gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

Kernel PCA



Cells 56-58, `14_n1.ipynb`

Selecting a Kernel and Tuning Hyperparameters

- As kPCA is an unsupervised learning algorithm, there is no obvious performance measure to help you select the best kernel and hyperparameter values.
- The dimensionality reduction is often a preparation step for a supervised learning task (e.g., classification), so you can use grid search to select the kernel and hyperparame- ters that lead to the best performance on that task.
- Cells 61-64, `l4_n1.ipynb`

Selecting a Kernel and Tuning Hyperparameters

- Another approach, this time entirely unsupervised, is to select the kernel and hyperparameters that yield the lowest reconstruction error.

```
rbf_pca = KernelPCA(n_components = 2, kernel="rbf",
                    gamma=0.0433,
                    fit_inverse_transform=True)
X_reduced = rbf_pca.fit_transform(X)
X_preimage = rbf_pca.inverse_transform(X_reduced)
from sklearn.metrics import mean_squared_error
mean_squared_error(X, X_preimage)
```

- Now you can use grid search with cross-validation to find the kernel and hyperparameters that minimize this error.

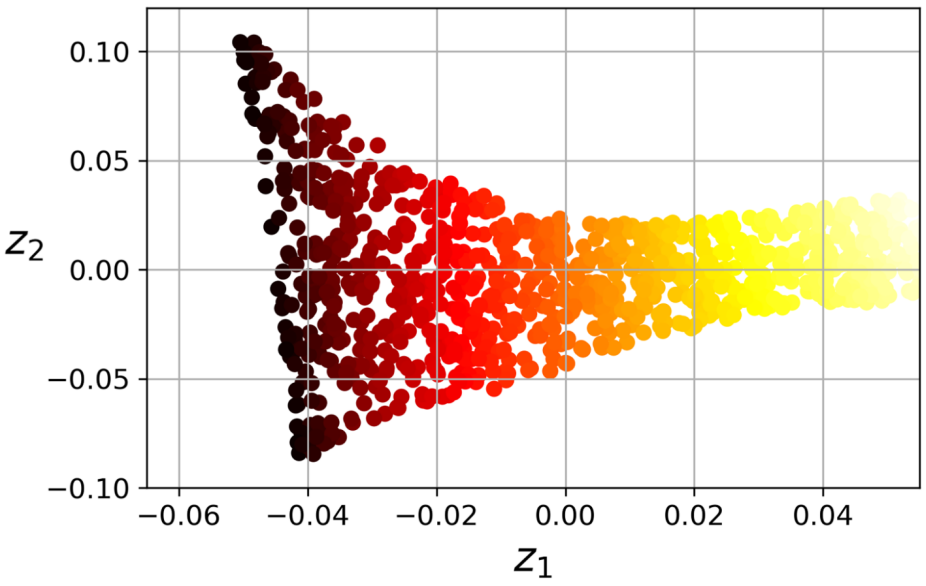
LLE

- Locally Linear Embedding (LLE) is another powerful nonlinear dimensionality reduction (NLDR) technique.
- It is a Manifold Learning technique that does not rely on projections, like the previous algorithms do.
- LLE works by first measuring how each training instance linearly relates to its closest neighbors (c.n.), and then looking for a low-dimensional representation of the training set where these local relationships are best preserved.
- This approach makes it particularly good at unrolling twisted manifolds, especially when there is not too much noise.

```
from sklearn.manifold import LocallyLinearEmbedding
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)
X_reduced = lle.fit_transform(X)
```

Cells 65-67, `l4_n1.ipynb`

LLE

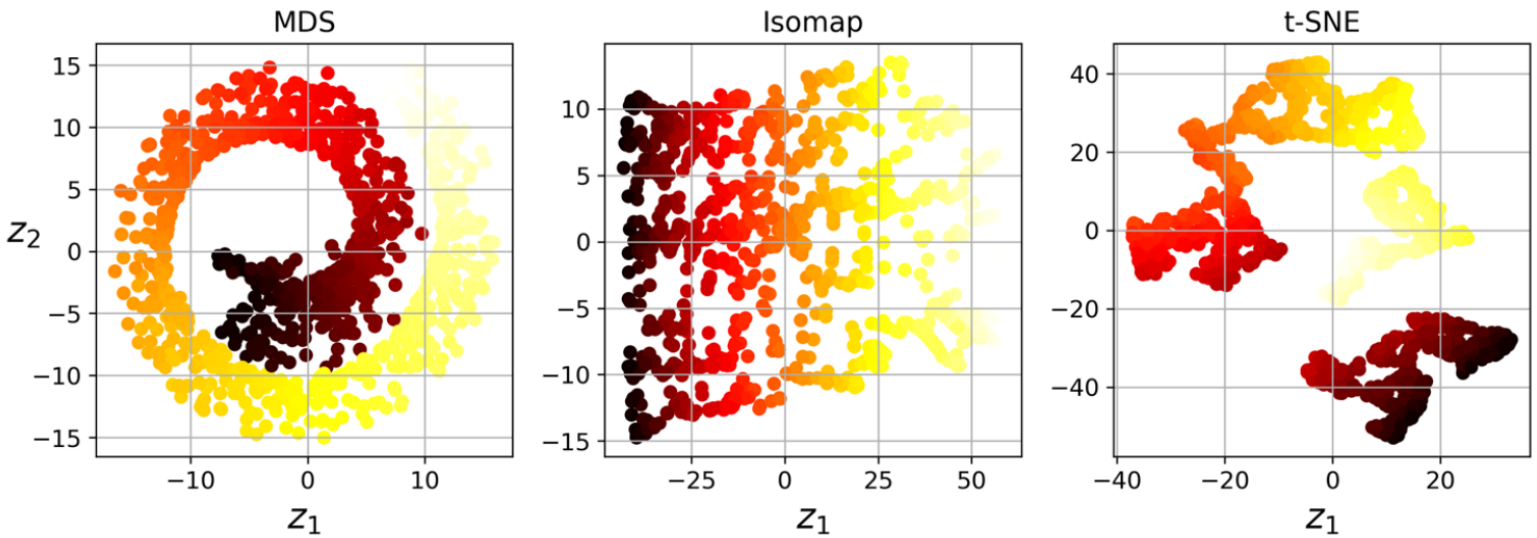


As you can see, the Swiss roll is completely unrolled, and the distances between instances are locally well preserved. However, distances are not preserved on a larger scale: the left part of the unrolled Swiss roll is stretched, while the right part is squeezed. Nevertheless, LLE did a pretty good job at modeling the manifold.

- Scikit-Learn's LLE implementation has the following computational complexity: $O(mlog(m)nlog(k))$ for finding the k nearest neighbors, $O(mnk^3)$ for optimizing the weights, and $O(dm^2)$ for constructing the low-dimensional representations. Unfortunately, the m^2 in the last term makes this algorithm scale poorly to very large datasets.

Other Dimensionality Reduction Techniques

- Random Projections
- Multidimensional Scaling (MDS)
- Isomap
- t-Distributed Stochastic Neighbor Embedding (t-SNE)
- Linear Discriminant Analysis (LDA)



Cells 68-72, `14_n1.ipynb`

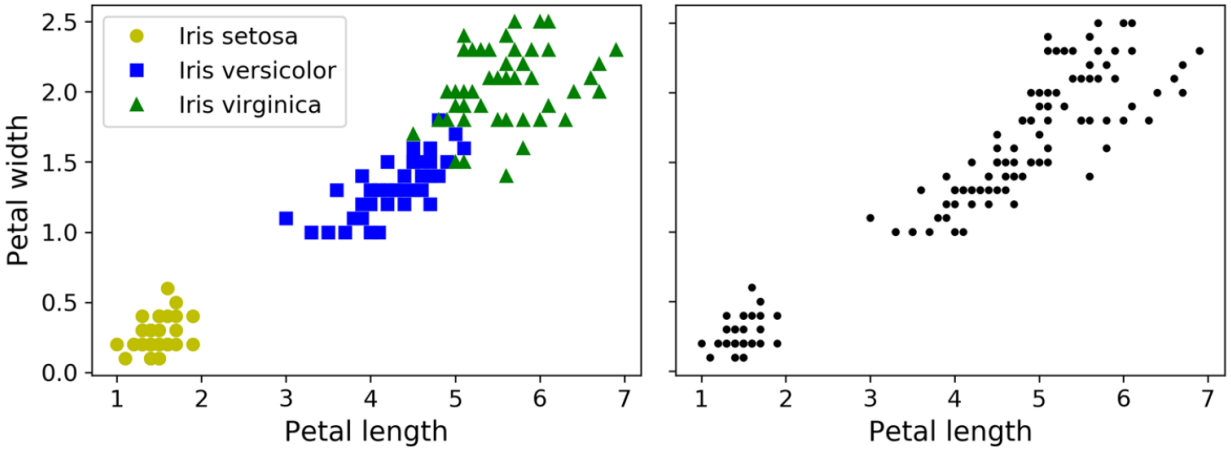
Unsupervised Learning

Usupervised Learning

- Although most of the applications of Machine Learning today are based on supervised learning, the vast majority of the available data is unlabeled
- Previously we looked into dimensionality reduction - this is an example of unsupervised learning
- Now we are going to look into several more unsupervised learning tasks
  - Clustering - used for for data analysis, customer segmentation, recommender systems, search engines, image segmentation, semi-supervised learning, dimensionality reduc- tion
  - Anomaly detection - the objective is to learn what "normal" data looks like, and then use that to detect abnormal instances, such as defective items on a production line, fraud detection
  - Density estimation - estimating the probability density function (PDF) of the random process that generated the dataset. Density estimation is commonly used for anomaly detection: instances located in very low-density regions are likely to be anomalies. It is also useful for data analysis and visualization

Clustering

- Clustering is the task of identifying similar instances and assigning them to clusters, or groups of similar instances.



Unlike classification, clustering is an unsupervised task.

- The left plot corresponds to the classification task: each instance has a label, the right plot - to the clustering task: no labels, one needs to assign similar instances to the same group.
- While Iris setosa instances clearly form a separate cluster, the other two classes seem to be merged into a single cluster if one considers only two features - petal length and width, adding two other features - sepal length and width - allows to separate Iris versicolor and Iris virginica into two groups

Clustering

Clustering is used in a wide variety of applications

- For customer segmentation
  You can cluster your customers based on their purchases and their activity on your website. This is useful to understand who your customers are and what they need, so you can adapt your products and marketing campaigns to each segment. For example, customer segmentation can be useful in recommender systems to suggest content that other users in the same cluster enjoyed.
- For data analysis
  When you analyze a new dataset, it can be helpful to run a clustering algorithm, and then analyze each cluster separately.

Clustering

- As a dimensionality reduction technique
  Once a dataset has been clustered, it is usually possible to measure each instance's affinity with each cluster (affinity is any measure of how well an instance fits into a cluster). Each instance's feature vector $\mathbf{x}$ can then be replaced with the vector of its cluster affinities. If there are k clusters, then this vector is k-dimensional. This vector is typically much lower-dimensional than the original feature vector, but it can preserve enough information for further processing.
- For anomaly detection (also called outlier detection)
  Any instance that has a low affinity to all the clusters is likely to be an anomaly. For example, if you have clustered the users of your website based on their behavior, you can detect users with unusual behavior, such as an unusual number of requests per second. Anomaly detection is particularly useful in detecting defects in manufacturing, or for fraud detection.

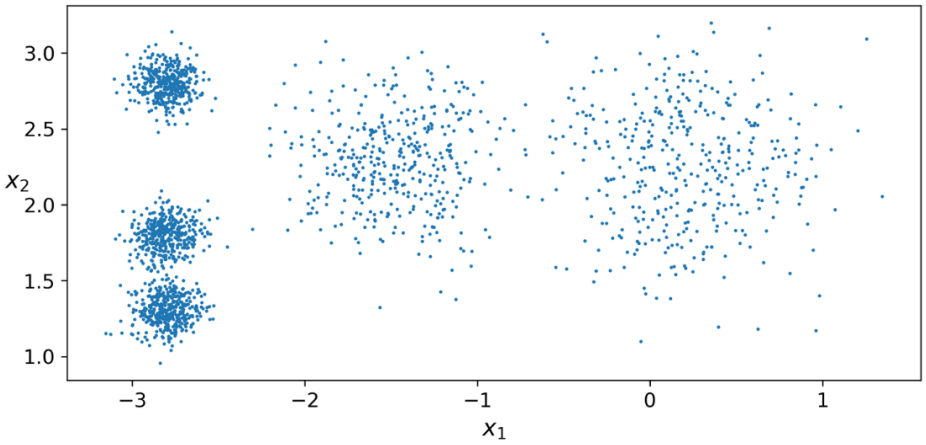
Clustering

- For 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. This technique can greatly increase the number of labels available for a subsequent supervised learning algorithm, and thus improve its performance.
- For search engines
  Some search engines let you search for images that are similar to a reference image. To build such a system, you would first apply a clustering algorithm to all the images in your database; similar images would end up in the same cluster. Then when a user provides a reference image, all you need to do is use the trained clustering model to find this image's cluster, and you can then simply return all the images from this cluster.

Clustering

- To segment an image
  By clustering pixels according to their color, then replacing each pixel's color with the mean color of its cluster, it is possible to considerably reduce the num- ber of different colors in the image. Image segmentation is used in many object detection and tracking systems, as it makes it easier to detect the contour of each object.
- There is no universal definition of what a cluster is: it really depends on the context, and different algorithms will capture different kinds of clusters. Some algorithms look for instances centered around a particular point, called a centroid. Others look for continuous regions of densely packed instances: these clusters can take on any shape. Some algorithms are hierarchical, looking for clusters of clusters.

K-Means



Let's train a K-Means clusterer on this dataset. It will try to find each blob's center and assign each instance to the closest blob:

```
from sklearn.cluster import KMeans
k = 5
kmeans = KMeans(n_clusters=k)
y_pred = kmeans.fit_predict(X)
```

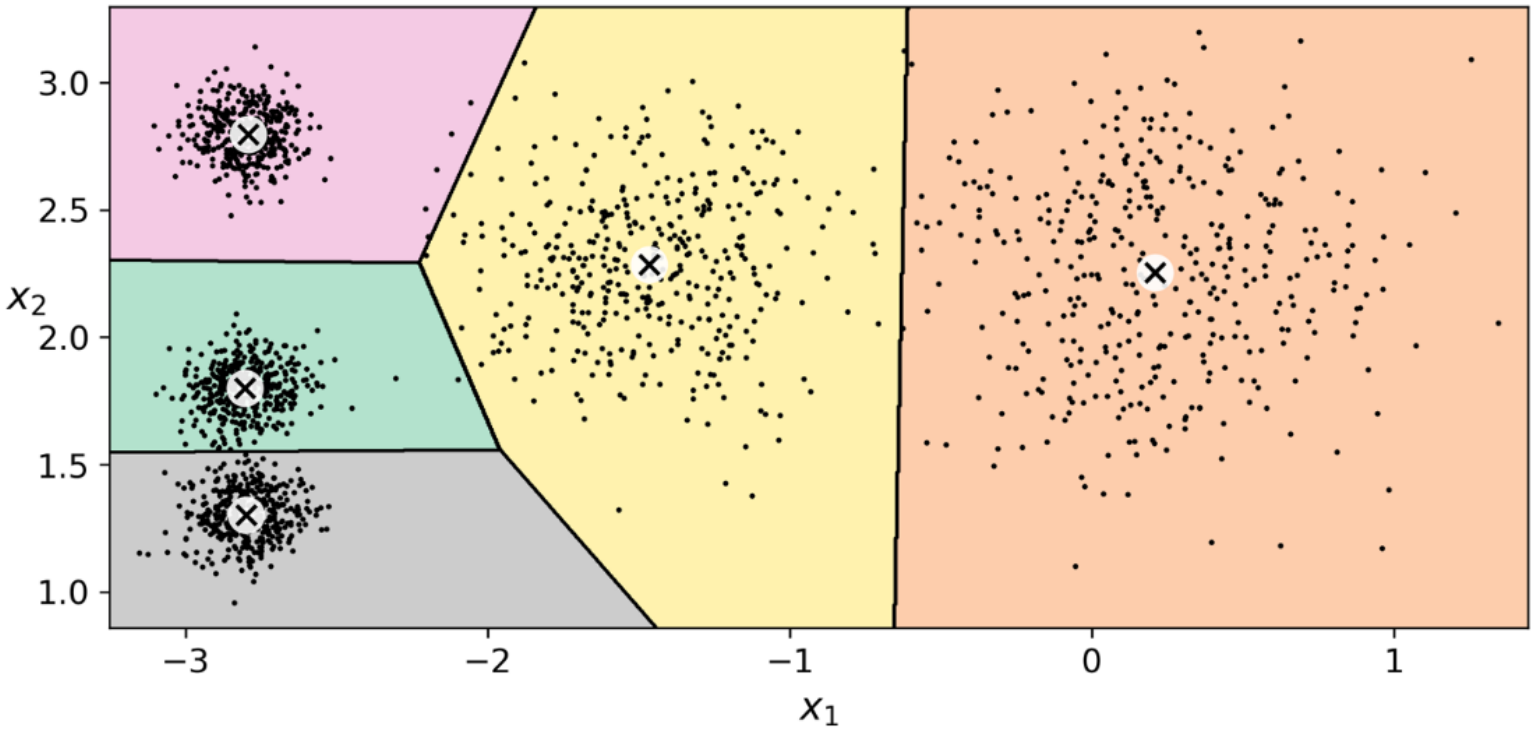
- Note that you have to specify the number of clusters k that the algorithm must find. In this example, it is pretty obvious from looking at the data that k should be set to 5, but in general it is not that easy.
- Cells 13-24, l4_n2.ipynb

K-Means

- `kmeans.labels_` contains indices of the clusters assigned to the instances of the training set
- To predict the corresponding cluster indices for the new data: `kmeans.predict(X_new)`
- To find the coordinates of the corresponding centroids: `kmeans.cluster_centers_`
- `kmeans.transform(X_new)` - measures the distances of each instance to each cluster's centroid
  - If you have a high-dimensional dataset and you transform it this way, you end up with a k-dimensional dataset: this transformation can be a very efficient nonlinear dimensionality reduction technique.

K-Means

- If you plot the cluster's decision boundaries, you get a Voronoi tessellation, where each centroid is represented with an X:



- The vast majority of the instances were clearly assigned to the appropriate cluster, but a few instances were probably mislabeled, especially near the boundary between the top-left cluster and the central cluster.

K-Means

- K-Means algorithm does not behave very well when the blobs have very different diameters because all it cares about when assigning an instance to a cluster is the distance to the centroid.
- Instead of assigning each instance to a single cluster, which is called hard clustering, it can be useful to give each instance a score per cluster, which is called soft clustering.
- The score can be the distance between the instance and the centroid; conversely, it can be a similarity score (or affinity), such as the Gaussian Radial Basis Function
- Cells 27-28, l4_n1.ipynb

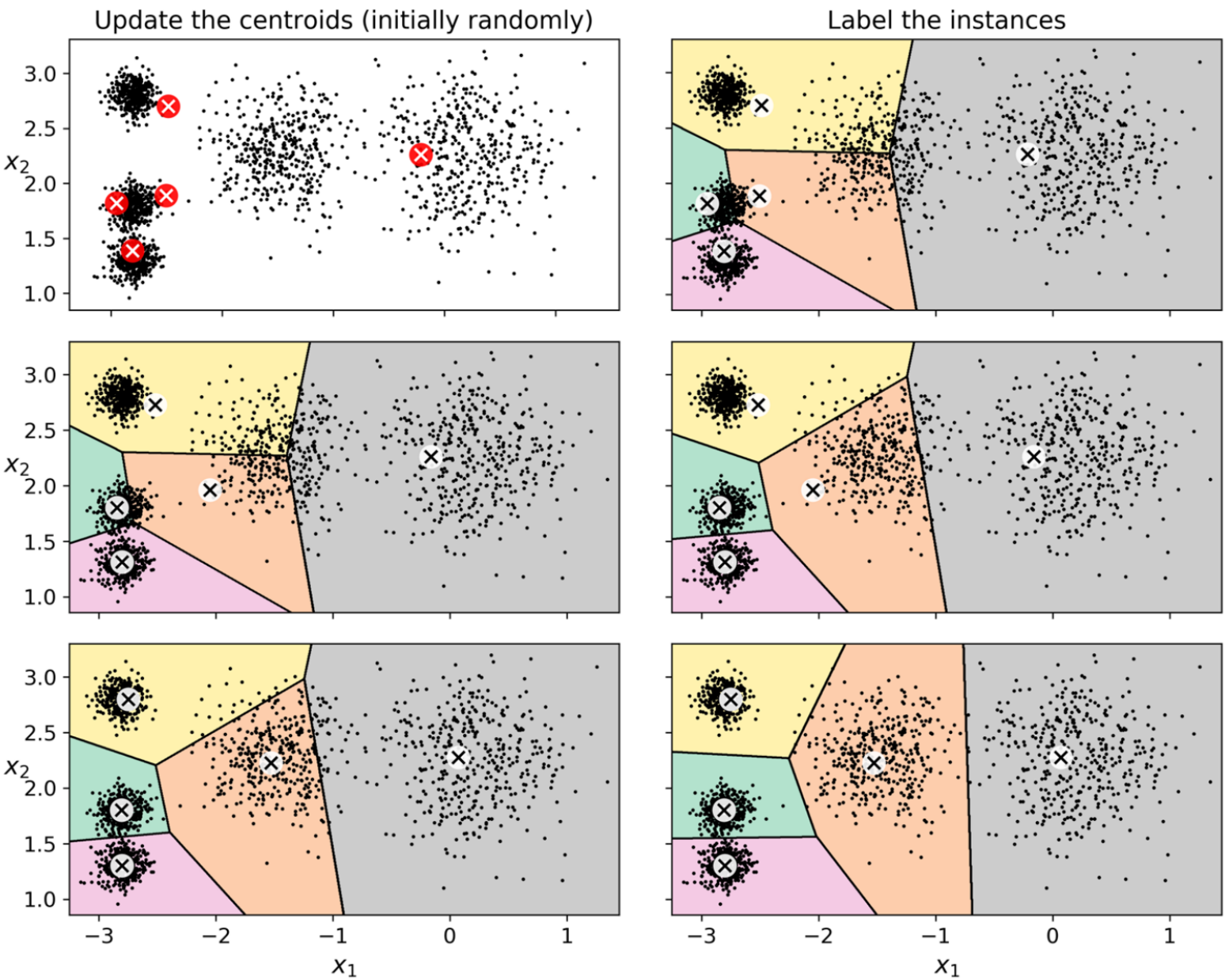
K-Means: algorithm

1. Place the centroids randomly, for example, by picking k instances at random and using their locations as centroids
2. Label the instances by assigning them to the cluster of the closest centroid
3. Update centroids by computing the mean of the instances for each cluster.
4. Keep repeating the last two steps until the centroids stop moving

---

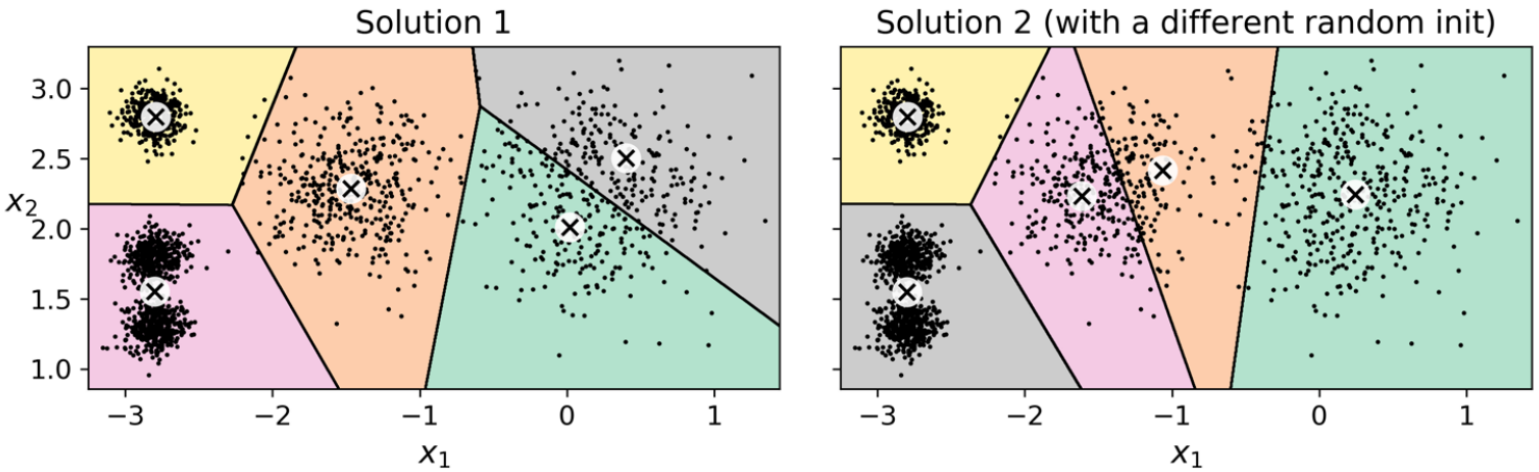
- The algorithm is guaranteed to converge in a finite number of steps (usually quite small); it will not oscillate forever

K-Means: algorithm



K-Means: algorithm

- 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.



K-Means: centroid initialization methods

- If you happen to 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, and set `n_init` to 1

```
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)
```

- 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` hyperparameter: by default, it is equal to 10: the whole algorithm runs 10 times when you call `fit()` and Scikit-Learn keeps the best solution
- The best solution is selected in terms of model's intertia: the mean squared distance between each instance and its closest centroid

K-Means: centroid initialization methods

- `kmeans.inertia_` contains model's intertia
- `kmeans.score(X)` returns negative intertia: a predictor's `score()` method must always respect Scikit-Learn's "greater is better" rule: if a predictor is better than another, its `kmeans.score(X)` method should return a greater score.
- `K-Means++` modification of K-Means is using a smarter initialization step that tends to select centroids that are distant from one another, and this improvement makes the K-Means algorithm much less likely to converge to a suboptimal solution:
  1. Select the first centroid uniformly randomly
  2. Select the next centroid among the instances with the probability that decreases with the distance to the closest selected centroid
  3. Keep repeating until k centroids are chosen
- `K-Means++` centroid intialization is used by default unless `init = ``random''`

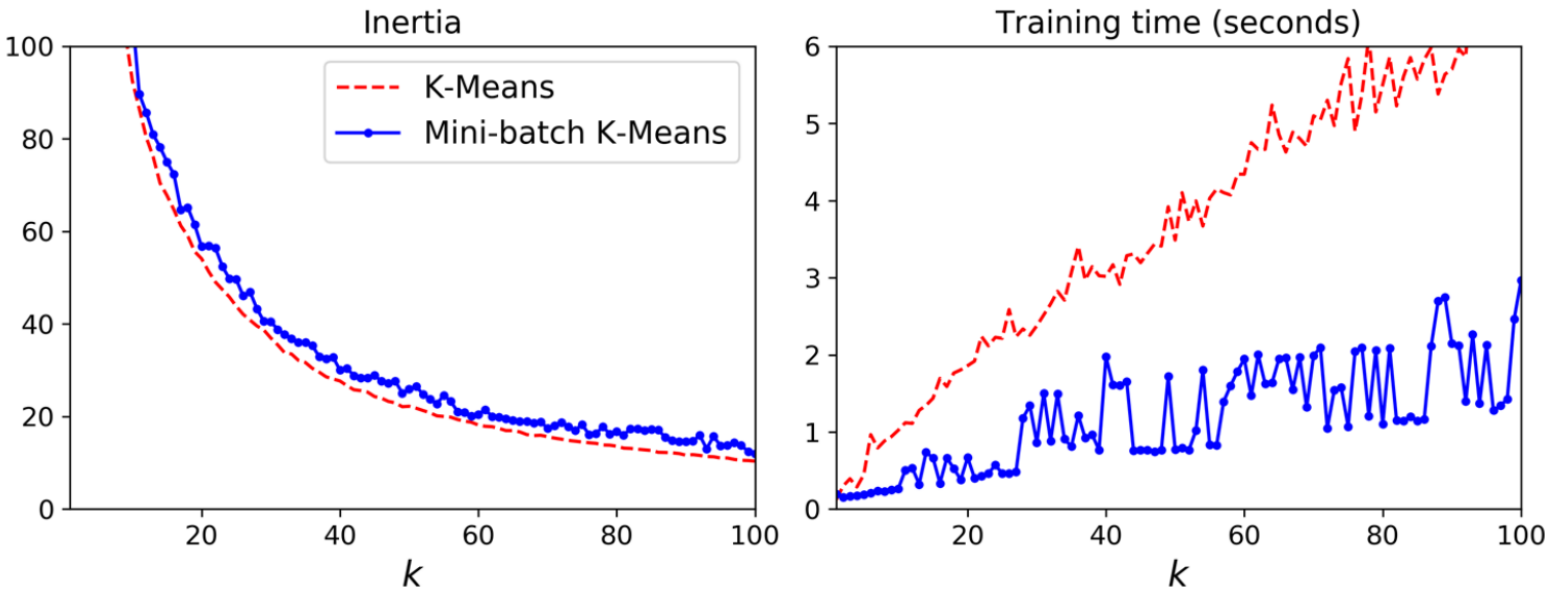
K-Means: MiniBatchKMeans

- 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 or four and makes it possible to cluster huge datasets that do not fit in memory.
- Scikit-Learn implements this algorithm in the MiniBatchKMeans class.

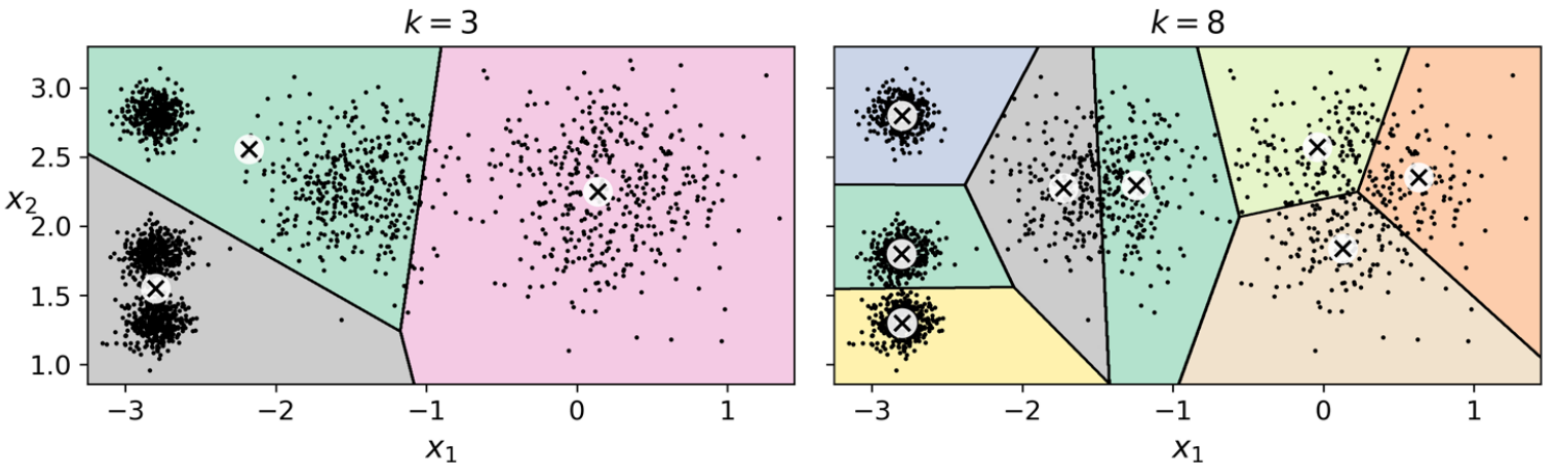
```
from sklearn.cluster import MiniBatchKMeans
minibatch_kmeans = MiniBatchKMeans(n_clusters=5)
minibatch_kmeans.fit(X)
```

- If the dataset does not fit in memory, the simplest option is to use the `memmap` class
- Alternatively, you can pass one mini-batch at a time to the `partial_fit()` method, but this will require much more work, since you will need to perform multiple initializations and select the best one yourself. Cells 44-59, l4_n2.ipynb

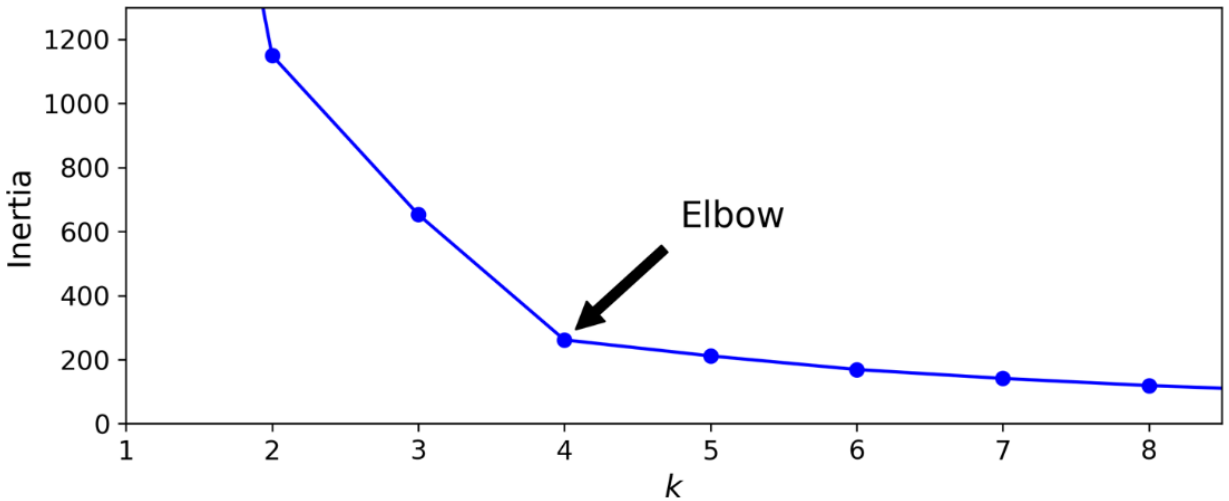
K-Means: MiniBatchKMeans



K-Means: Finding the optimal number of clusters



- How to select k?
- The inertia gets smaller as k increases



k near the elbow would be a good choice.

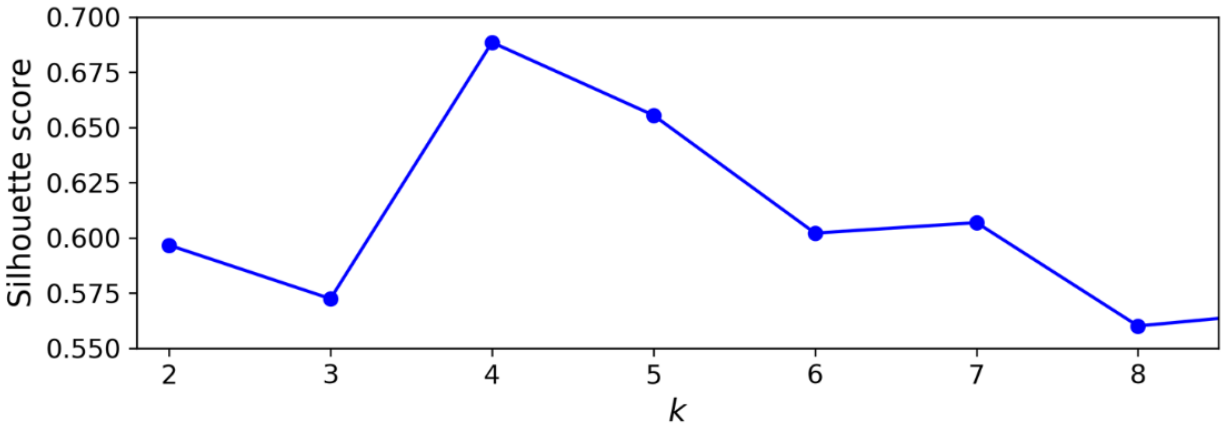
K-Means: Finding the optimal number of clusters

- A more precise approach (but also more computationally expensive) is to use the silhouette score, which is the mean silhouette coefficient over all the instances
- An instance's silhouette coefficient is equal to $\frac{b-a}{max(a,b)}$, where a is the mean distance to the other instances in the same cluster (i.e., the mean intra-cluster distance) and b is the mean nearest-cluster distance (i.e., the mean distance to the instances of the next closest cluster, defined as the one that minimizes b, excluding the instance's own cluster).
- 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, while a coefficient close to 0 means that it is close to a cluster boundary, and finally a coefficient close to -1 means that the instance may have been assigned to the wrong cluster.

K-Means: Finding the optimal number of clusters

```
from sklearn.metrics import silhouette_score
score = silhouette_score(X, kmeans.labels_)
```

Let's compare the silhouette scores for different numbers of clusters



- As you can see, this visualization is much richer than the previous one: although it confirms that $k = 4$ is a very good choice, it also underlines the fact that $k = 5$ is quite good as well, and much better than $k = 6$ or $k = 7$. This was not visible when comparing inertias.

K-Means: Limitations

- It is necessary to run the algorithm several times to avoid suboptimal solutions
- You need to specify the number of clusters
- K-Means does not behave very well when the clusters have varying sizes, different densities, or nonspherical shapes.
- It is important to scale the input features before you run K-Means, or the clusters may be very stretched and K-Means will perform poorly. Scaling the features does not guarantee that all the clusters will be nice and spherical, but it generally improves things.

Using Clustering for Image Segmentation

- Cells 75-79, l4_n2.ipynb
- Image segmentation is the task of partitioning an image into multiple segments.
- 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 Image Segmentation

- Here, we are going to do something much simpler: color segmentation. We will simply assign pixels to the same segment if they have a similar color

Using Clustering for Preprocessing

- Cells 80-94, l4_n2.ipynb
- Clustering can be an efficient approach to dimensionality reduction, in particular as a preprocessing step before a supervised learning algorithm.
- As an example of using clustering for dimensionality reduction, let's tackle the digits dataset, which is a simple MNIST-like dataset containing 1,797 grayscale 8×8 images representing the digits 0 to 9
- If we apply Logistic Regression, we get 0.969 accuracy.
- Let's see if we can do better by using K-Means as a preprocessing step.
- We create a pipeline that will first cluster the training set into 50 clusters and replace the images with their distances to these 50 clusters, then apply a Logistic Regression model
- The accuracy becomes 0.978
- Next let us do a grid search to select k that gives the best accuracy: we find that with k=99 we get 0.982 accuracy

Using Clustering for Semi-Supervised Learning

- Cells 95-110, l4_n2.ipynb
- Another use case for clustering is in semi-supervised learning, when we have plenty of unlabeled instances and very few labeled instances. Let's train a Logistic Regression model on a sample of 50 labeled instances from the digits dataset. The performance of this model on the test set is 0.83
- Let us cluster the training set into 50 groups, select best representative from each group, label those manually, train the model on those. The performance on the test set is 0.92.
- Let us propagate the labels to each instance of the cluster and train on the whole labelled train set. The performance on the test set is 0.93.

Using Clustering for Semi-Supervised Learning

- In order to avoid using instances on the border between clusters, instead of training on the whole train set, let us select in each cluster top 20% instances closest to the centroid and train the model only on those. The performance on the test set is 0.94.
- Just using 50 labelled instances out of 1,797, we managed to achieve performance close to what Logistic Regression would achieve (0.96) when trained on the whole train set.

Active Learning

- To continue improving your model and your training set, the next step could be to do a few rounds of active learning, which is when a human expert interacts with the learning algorithm, providing labels for specific instances when the algorithm requests them.
- There are many different strategies for active learning, but one of the most common ones is called uncertainty sampling:
  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 are given to the expert to be labeled.
  3. You iterate this process until the performance improvement stops being worth the labeling effort.
- Another strategy: label manually the instances on which different models (SVM, Random Forest, etc.) disagree

DBSCAN

- DBSCAN is another popular clustering algorithm that illustrates a very different approach based on local density estimation.
- This approach allows the algorithm to identify clusters of arbitrary shapes.
- This algorithm defines clusters as continuous regions of high density:
  1. For each instance, the algorithm counts 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

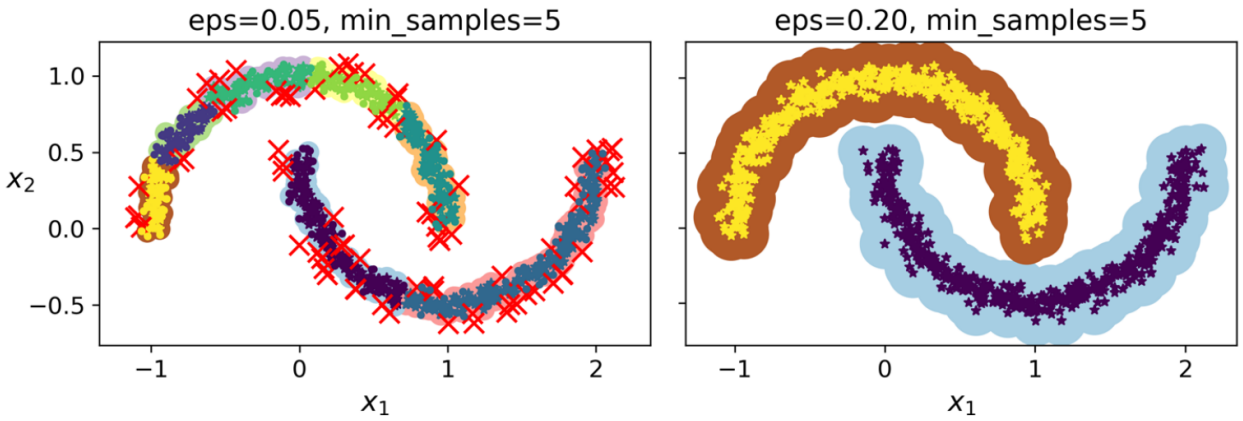
- This algorithm works well if all the clusters are dense enough and if they are well sep- arated by low-density regions.
- Cells 111-129, l4_n2.ipynb

```
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)
```

- After training one can find cluster labels for the instances in `dbscan.labels_`
- -1 label corresponds to an outlier that does not belong to any cluster

DBSCAN

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



- The red crosses on the left plot correspond to outliers.
- If we increase ϵ we get much better clustering on the right plot.

DBSCAN

- Surprisingly, the DBSCAN class does not have a `predict()` method, although it has a `fit_predict()` method. In other words, it cannot predict which cluster a new instance belongs to. To do that, train some classifier on the instances labelled by DBSCAN. For example:

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbscan.components_,
        dbscan.labels_[dbscan.core_sample_indices_])
X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
knn.predict(X_new)
```

- DBSCAN's computational complexity is roughly $O(m \log(m))$, Scikit-Learn's implementation might require $O(m^2)$ memory if ϵ is large.

DBSCAN

- DBSCAN is a very 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: ϵ and `min_samples`.
- If the density varies significantly across the clusters, however, it can be impossible for it to capture all the clusters properly.

Other clustering algorithms

Other clustering algorithms available in Scikit-Learn, Cells 130-140, l4_n2.ipynb

- Agglomerative clustering
- BIRCH
- Mean-Shift
- Affinity propagation
- Spectral clustering