Unsupervised Learning

Clustering

Outline

- **Clustering: K-Means**
- Clustering: K-Means application and other methods

Key topics

- **Clustering: group similar instances together into clusters.** E.g.: data analysis, customer segmentation, recommender systems, search engine, image segmentation, semi-supervised learning, dimensionality reduction, and more.
- Anomaly 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 generated the dataset. E.g.: anomaly detection and data analysis & visualization.

Clustering

- Iris data

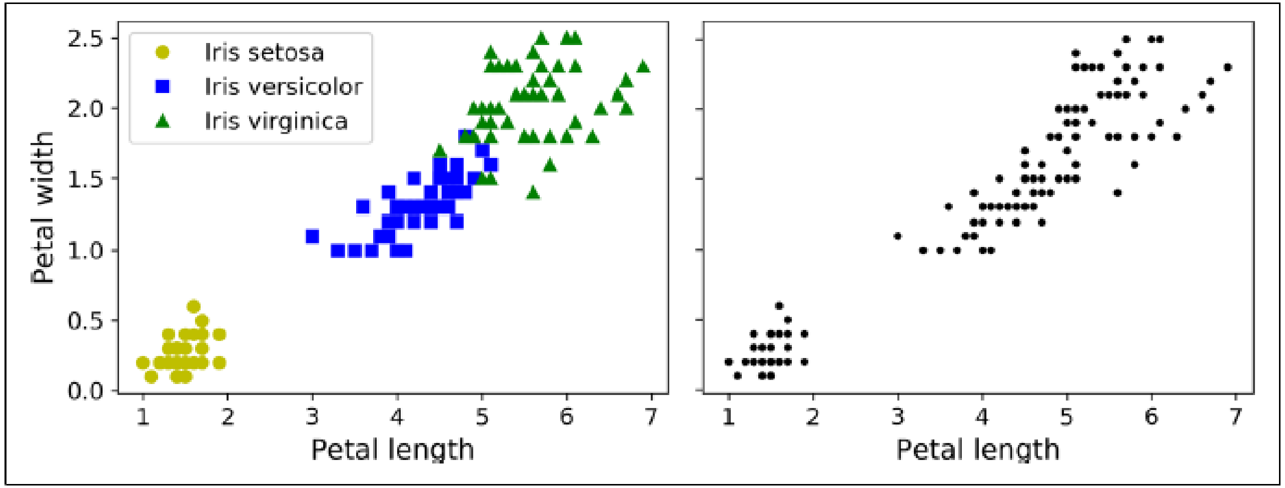


Figure 9-1. Classification (left) versus clustering (right)

Clustering: K-Means (Lloyd–Forgy, 1957, Bell Labs)

- Unlabeled dataset

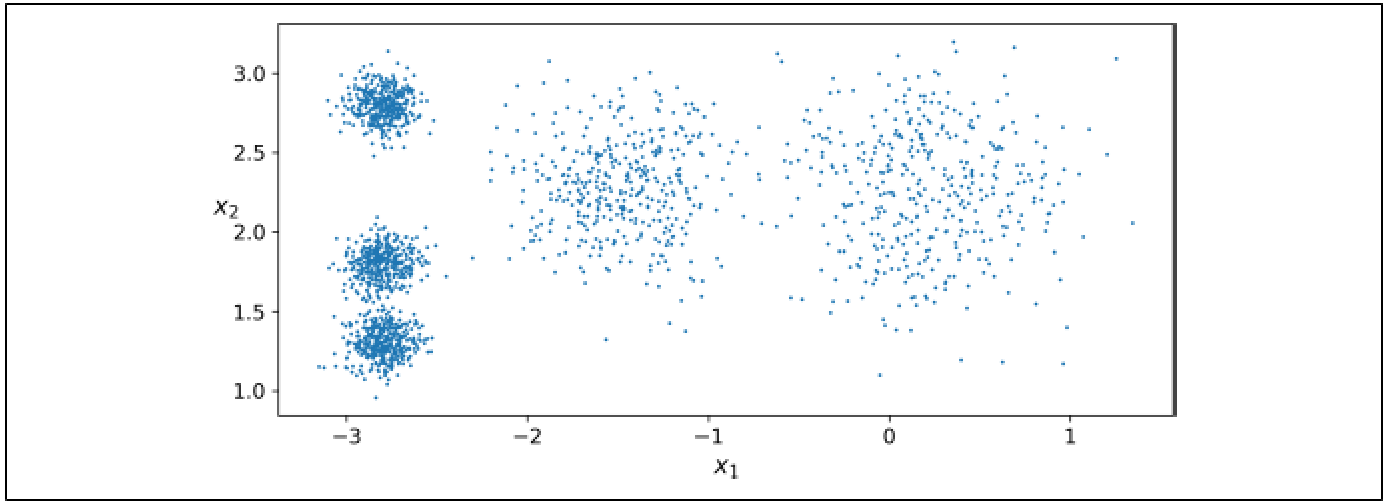


Figure 9-2. An unlabeled dataset composed of five blobs of instances

Clustering: K-Means (Lloyd–Forgy, 1957, Bell Labs)

- Unlabeled dataset

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

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

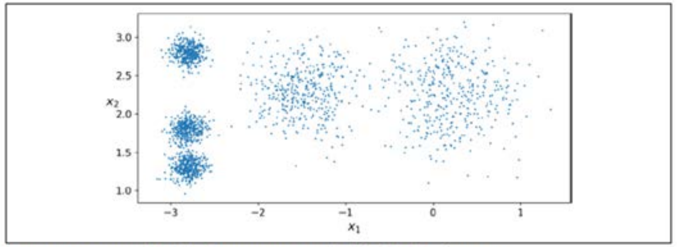


Figure 9-2. An unlabeled dataset composed of five blobs of instances

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

Clustering: K-Means (Lloyd–Forgy, 1957, Bell Labs)

- Assigning new instances

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

- Transform() method measures the distance from each instance to every centroid.

```
>>> kmeans.transform(X_new)
array([[2.81093633, 0.32995317, 2.9042344 , 1.49439034, 2.88633901],
       [5.80730058, 2.80290755, 5.84739223, 4.4759332 , 5.84236351],
       [1.21475352, 3.29399768, 0.29040966, 1.69136631, 1.71086031],
       [0.72581411, 3.21806371, 0.36159148, 1.54808703, 1.21567622]])
```

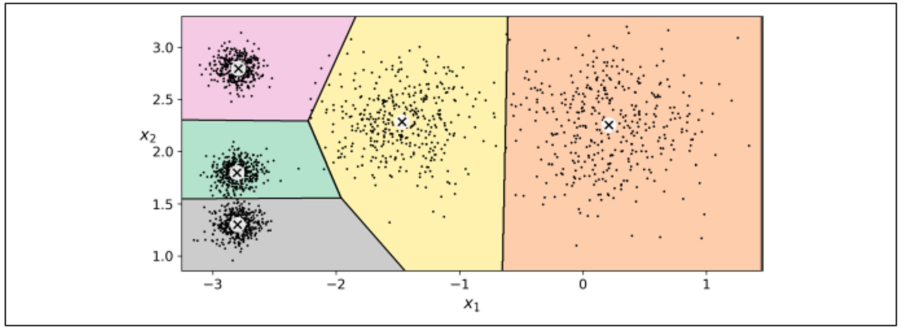


Figure 9-3. K-Means decision boundaries (Voronoi tessellation)

Clustering: K-Means Algorithms

- Initialize the centroids randomly
- Assign the instances
- Update the centroids based on distances
- Relabel the instances
- ……

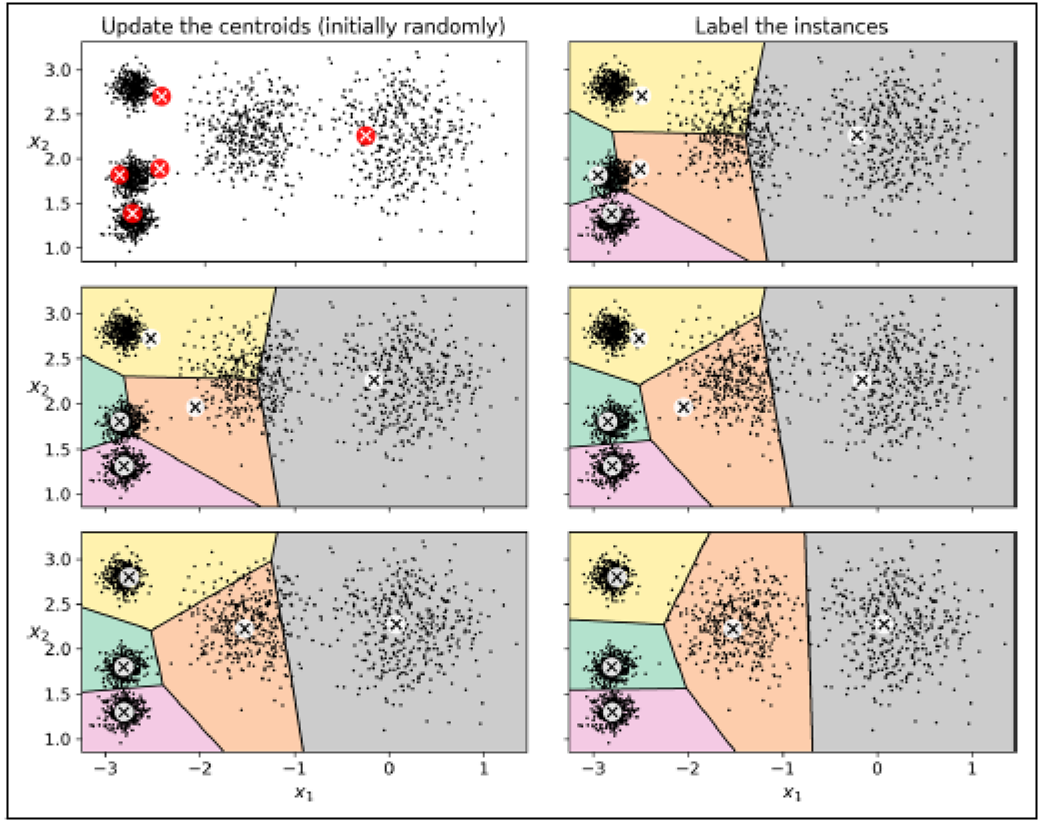


Figure 9-4. The K-Means algorithm

Clustering: K-Means Algorithms (risk of random initialization)

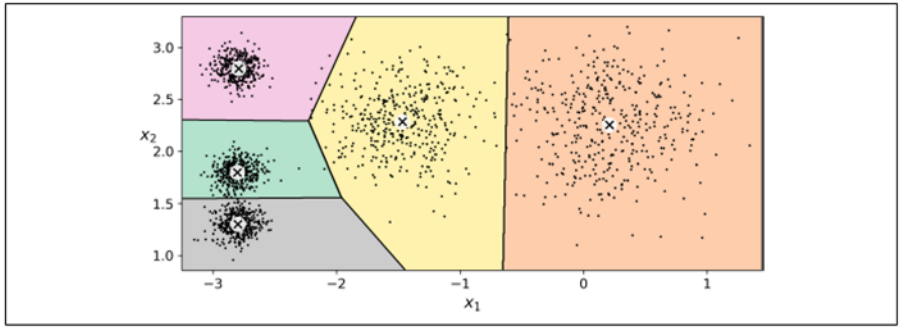


Figure 9-3. K-Means decision boundaries (Voronoi tessellation)

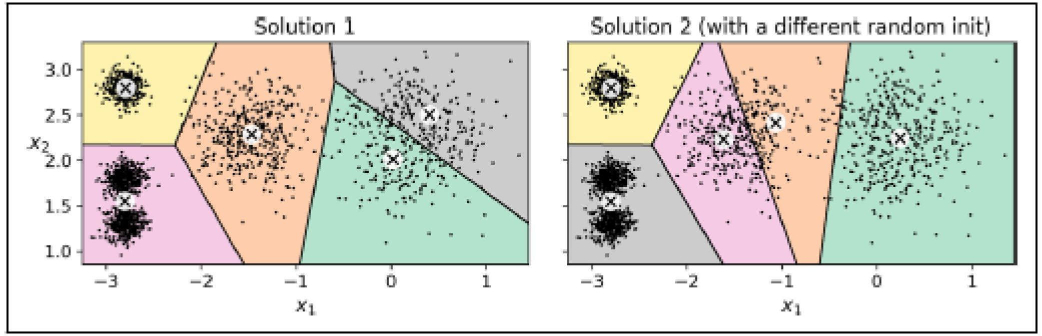
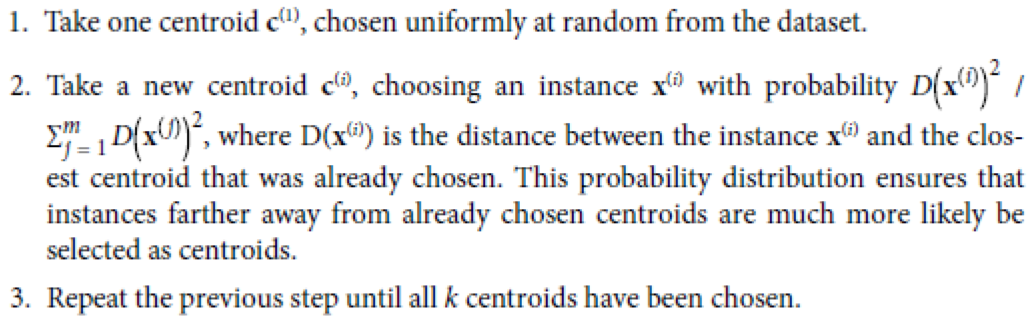


Figure 9-5. Suboptimal solutions due to unlucky centroid initializations

Clustering: K-Means Algorithms (centroid initialization methods)

- Base on empirical evidence/other algorithms (*init*)
- Multiple runs (*n_init*) (*inertia_*) (*score: negative of inertia_*)
- K-Means ++



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 $\mathrm{D}(\mathbf{x}^{(i)})$ 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 be selected as centroids.
3. Repeat the previous step until all k centroids have been chosen.

Clustering: accelerated K-Means and mini-batch K-Means

- Accelerated K-Means: avoid many unnecessary distance calculations. (default in Scikit-Learn) (**algorithm = 'full'** will force to calculate all distances)
- 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.)

```
from sklearn.cluster import MiniBatchKMeans

minibatch_kmeans = MiniBatchKMeans(n_clusters=5)
minibatch_kmeans.fit(X)
```

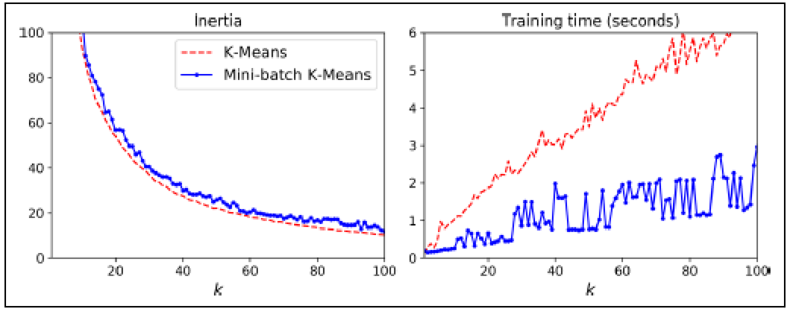


Figure 9-6. Mini-batch K-Means has a higher inertia than K-Means (left) but it is much faster (right), especially as k increases

Clustering: finding the optimal number of clusters (inertia)

- Bad choices for the number of clusters
- “elbow” of inertia (inflection point)

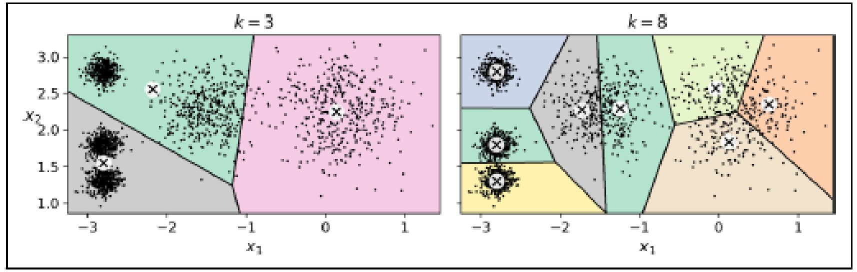


Figure 9-7. Bad choices for the number of clusters: when k is too small, separate clusters get merged (left), and when k is too large, some clusters get chopped into multiple pieces (right)

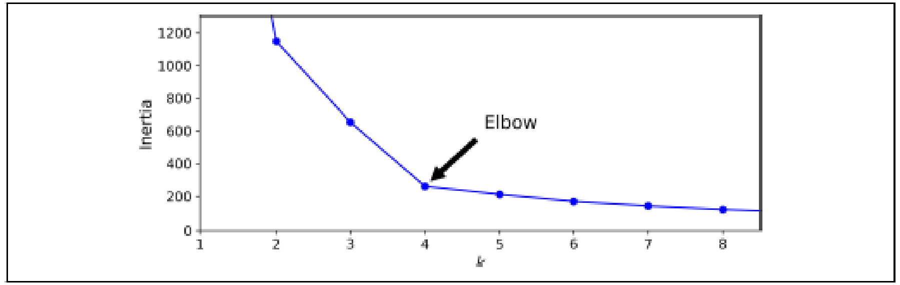


Figure 9-8. When plotting the inertia as a function of the number of clusters k, the curve often contains an inflexion point called the “elbow”

Clustering: finding the optimal number of clusters (sihouette score)

- Silhouette coefficient is equal to (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**).
- **Sihouette score** is the mean of *silhouette coefficient* over all the instances.
  - Between -1 to 1
  - Close to +1 means the instance is well inside its own cluster and far from other clusters
  - Close to 0 means that it is close to a cluster boundary
  - Close to -1 means the instance may have been assigned to the wrong cluster

Clustering: finding the optimal number of clusters (sihouette score)

- K=4 is a very good choice
- K=5 is good as well

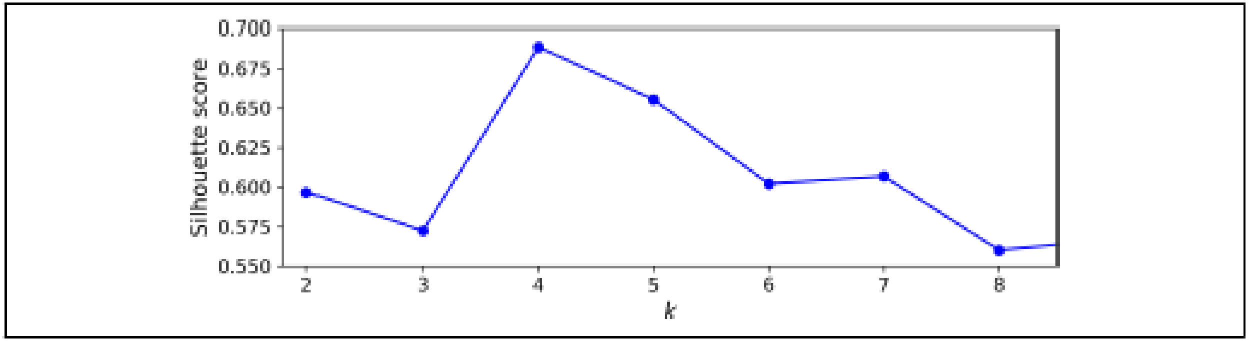


Figure 9-9. Selecting the number of clusters k using the silhouette score

Clustering: finding the optimal number of clusters (sihouette diagram)

- Sihouette diagram
- Height indicates the number of instances the cluster contains
- Width represents the sorted silhouette coefficients of the instances in the cluster (wider is better)
- Dashed red line indicates the mean silhouette coefficient.
- K = 5 to get cluster of similar sizes.

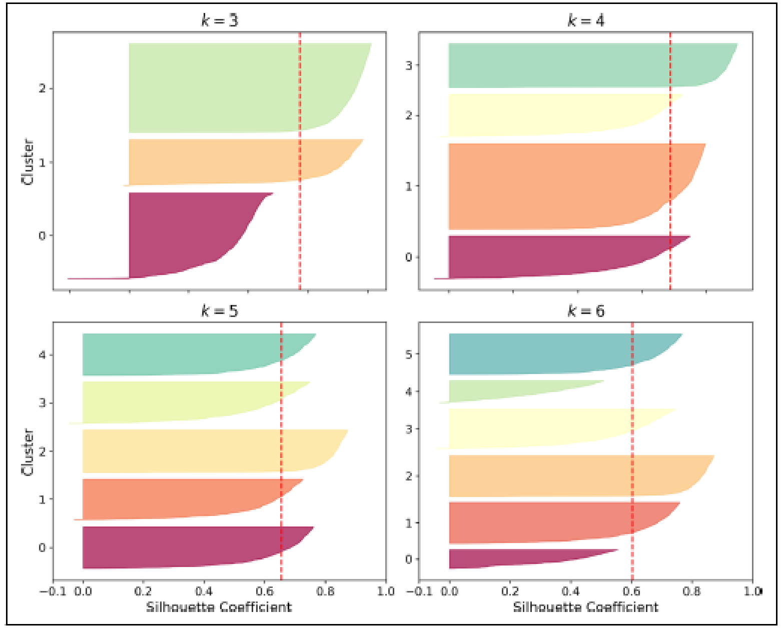


Figure 9-10. Analyzing the silhouette diagrams for various values of k

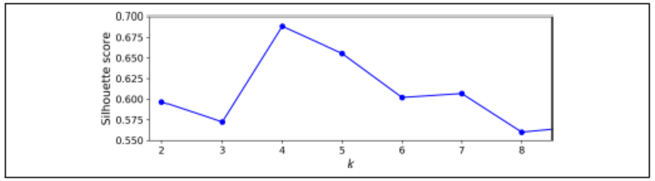


Figure 9-9. Selecting the number of clusters k using the silhouette score

Limits of K-Means clustering

- Run multiple times to avoid suboptimal solutions
- Specify number of clusters
- K-Means does not behave very well when the clusters have varying sizes, different densities, or nonspecial shapes
- May worth looking into other clustering algorithms when this happens

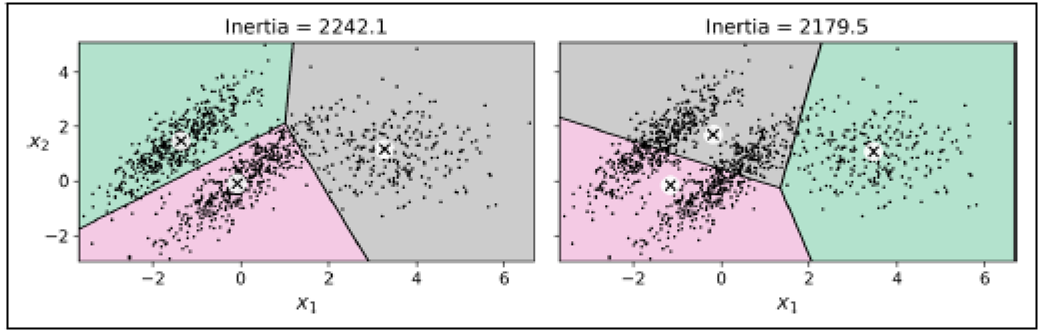


Figure 9-11. K-Means fails to cluster these ellipsoidal blobs properly

Using clustering for image segmentation

- Assign pixels to the same segment if they have a similar color
    - Matplotlib’s imread() to load images
    - Images are reshaped to a 3D array (height, weight, number of color channels)
    - Reshape this long list of colors to get the same shape as the original image

```
>>> from matplotlib.image import imread  # or `from imageio import imread`
>>> image = imread(os.path.join("images","unsupervised_learning","ladybug.png"))
>>> image.shape
(533, 800, 3)
```

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

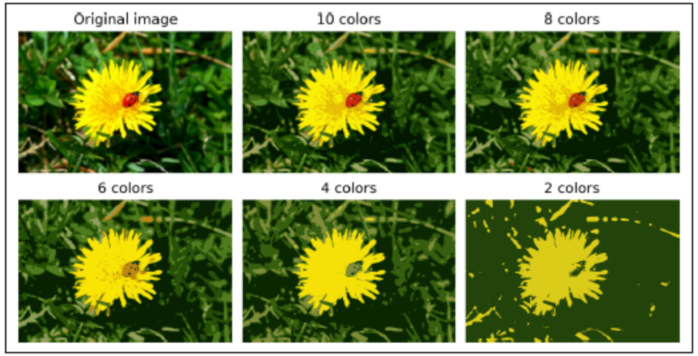
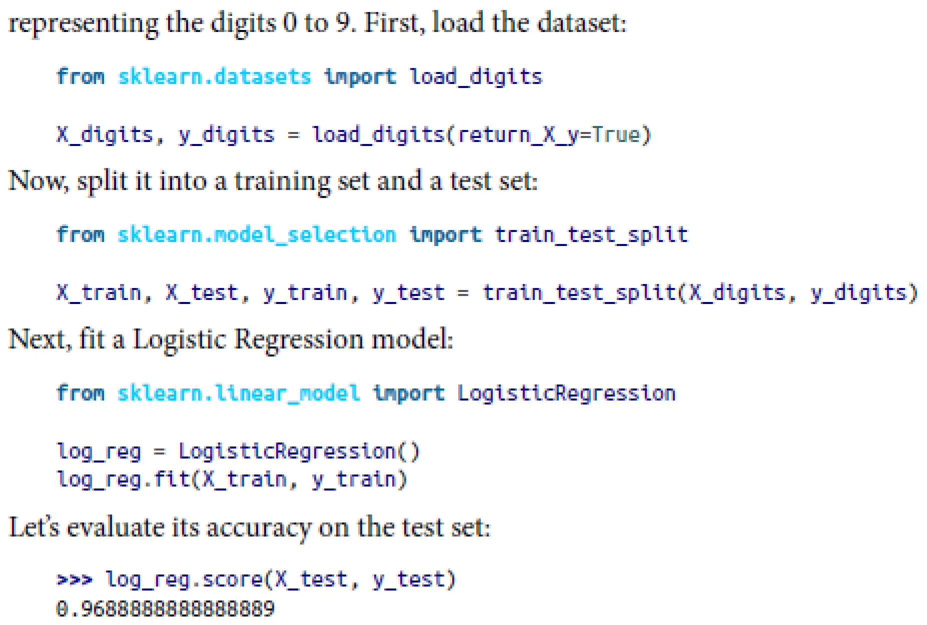


Figure 9-12. Image segmentation using K-Means with various numbers of color clusters

Using clustering for preprocessing

- A MNIST-like dataset containing 1797 grayscale 8*8 images



representing the digits 0 to 9. First, load the dataset:

```
from sklearn.datasets import load_digits

X_digits, y_digits = load_digits(return_X_y=True)
```

Now, split it into a training set and a test set:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_digits, y_digits)
```

Next, fit a Logistic Regression model:

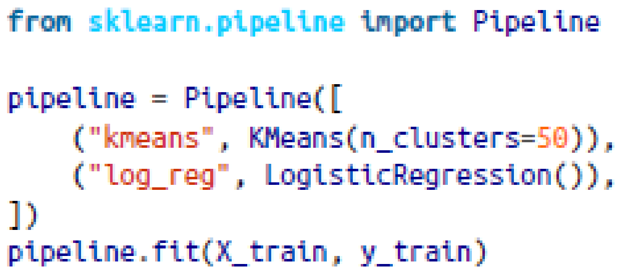
```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
```

Let's evaluate its accuracy on the test set:

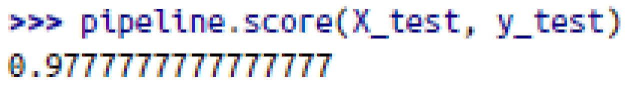
```
>>> log_reg.score(X_test, y_test)
0.9688888888888889
```

Using clustering for preprocessing

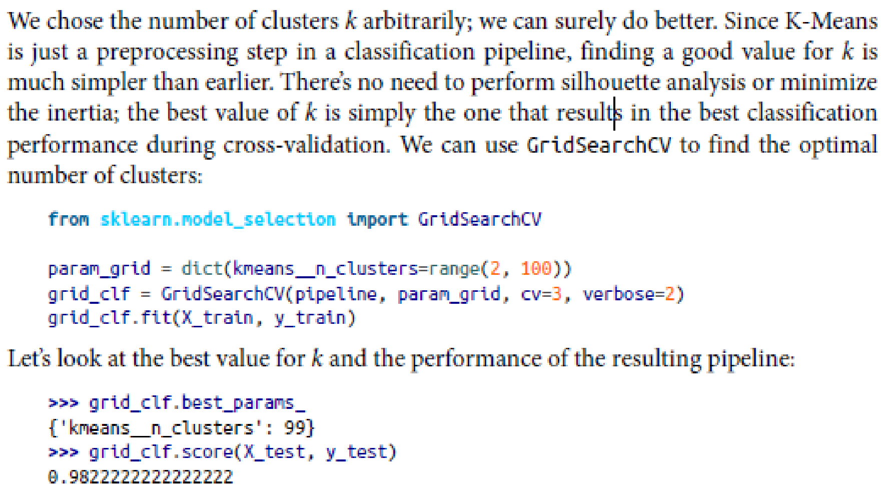
- Create a pipeline that will first cluster the training set into 50 clusters and replace the images with their distance to these 50 clusters, then apply a logistic regression model:



```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ("kmeans", KMeans(n_clusters=50)),
    ("log_reg", LogisticRegression()),
])
pipeline.fit(X_train, y_train)
```



```
>>> pipeline.score(X_test, y_test)
0.9777777777777777
```



We chose the number of clusters *k* arbitrarily; we can surely do better. Since K-Means is just a preprocessing step in a classification pipeline, finding a good value for *k* is much simpler than earlier. There's no need to perform silhouette analysis or minimize the inertia; the best value of *k* is simply the one that results in the best classification performance during cross-validation. We can use `GridSearchCV` to find the optimal number of clusters:

```
from sklearn.model_selection import GridSearchCV

param_grid = dict(kmeans__n_clusters=range(2, 100))
grid_clf = GridSearchCV(pipeline, param_grid, cv=3, verbose=2)
grid_clf.fit(X_train, y_train)
```

Let's look at the best value for *k* and the performance of the resulting pipeline:

```
>>> grid_clf.best_params_
{'kmeans__n_clusters': 99}
>>> grid_clf.score(X_test, y_test)
0.9822222222222222
```

Using clustering for semi-supervised learning

- Train a Logistic Regression model on a sample of 50 labeled instances from the digits dataset.

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

What is the performance of this model on the test set?

```
>>> log_reg.score(X_test, y_test)
0.8333333333333334
```

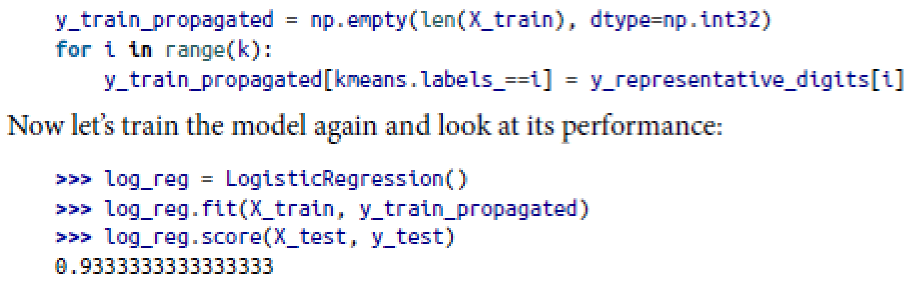
- Clustering, labeling, and then Logistic Regression

```
k = 50
kmeans = KMeans(n_clusters=k)
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]
```

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_representative_digits, y_representative_digits)
>>> log_reg.score(X_test, y_test)
0.9222222222222223
```

Using clustering for semi-supervised learning

- Propagate the labels to all the other instances in the same cluster (label propagation).

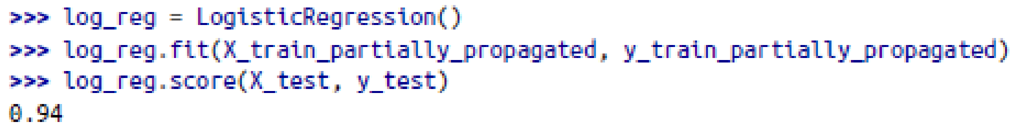


```
y_train_propagated = np.empty(len(X_train), dtype=np.int32)
for i in range(k):
    y_train_propagated[kmeans.labels_==i] = y_representative_digits[i]
```

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)
0.9333333333333333
```

- Propagate only 20% of the instances that are closest to the centroids.



```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
>>> log_reg.score(X_test, y_test)
0.94
```

Clustering: DBSCAN

The algorithm defines clusters as continuous regions of high density

- 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, then it is considered a core instance.
- All insances in the neighborhood of a core instance belong to the same cluster.
- Any instance that is not a core instance and does not have one in its neighborhood is considered to be an anomaly.

Clustering: DBSCAN

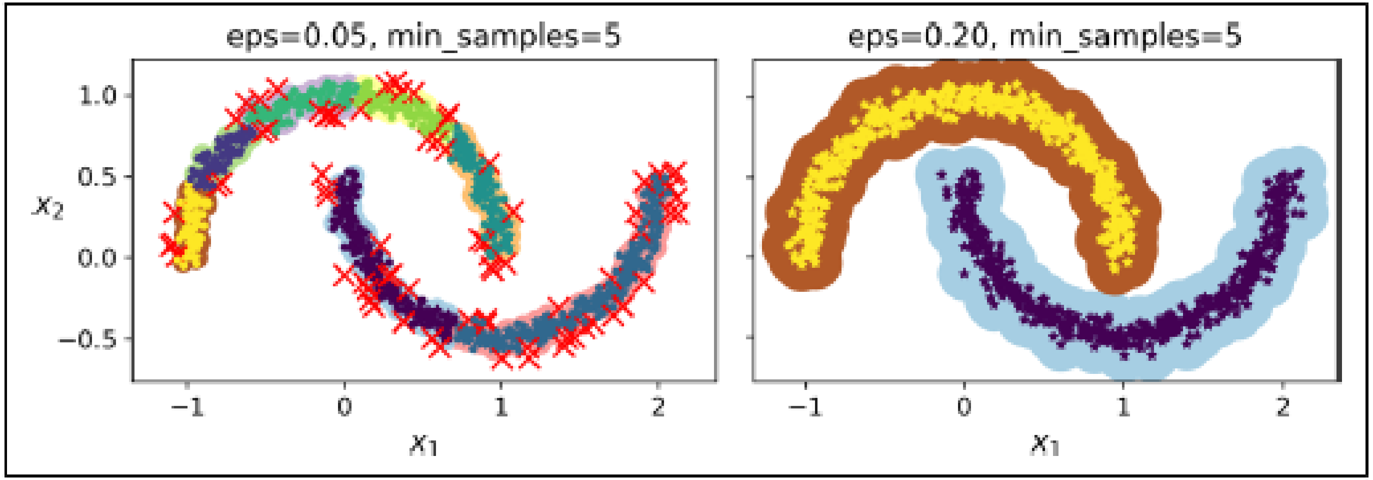


Figure 9-14. DBSCAN clustering using two different neighborhood radiuses

Clustering: DBSCAN

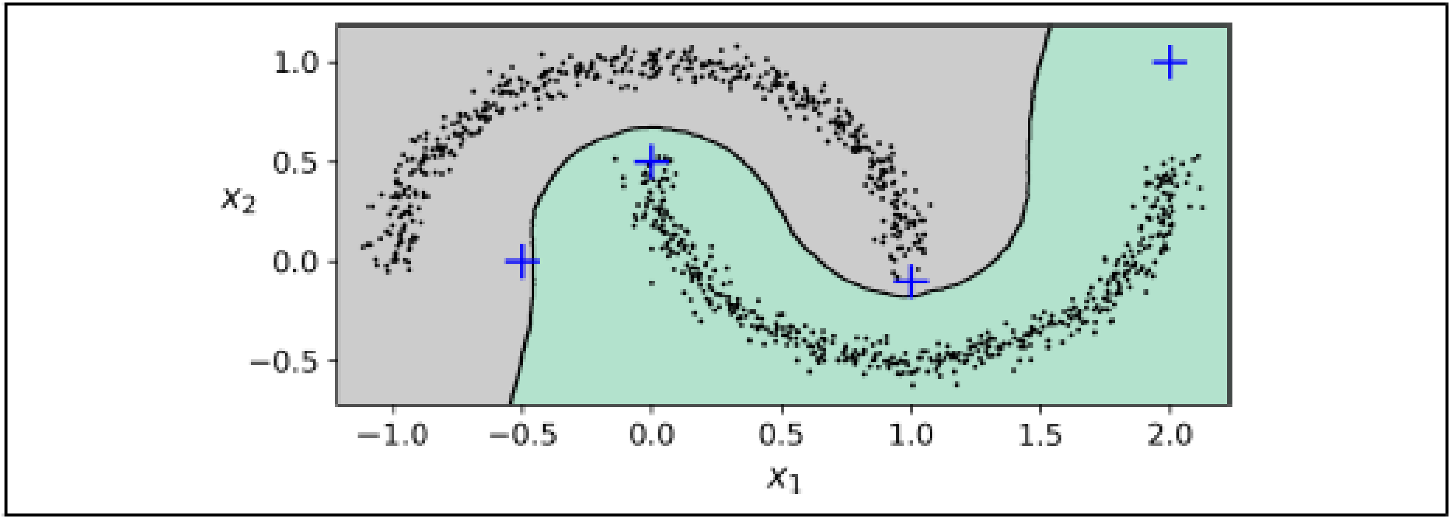


Figure 9-15. Decision boundary between two clusters

Other clustering algorithms

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