Chap9

Unsupervised Learning Techniques

Written by Jin Kim

237~238 목차

- *Clustering* : 비슷한 그룹을 *clusters* 로 묶는것.
  - data analysis, customer segmentation, recommender systems, search engines, image segmentation, semi-supervised learning, dimensionality reduction 등에 좋음.
- *Anomaly detection* (이상 탐지) : 정상적인 데이터를 학습하여 비정상적인 데이터를 감지하는것.
- *Density estimation* : 데이터를 만들어낸 랜덤 프로세스로부터 *probability density function*(PDF)을 측정하는것.
  - anomaly detection에 흔히 쓰임. 밀집도가 낮을수록 비정상일 확률이 높음. data analysis and visualization 에 좋음.

238~239 Clustering

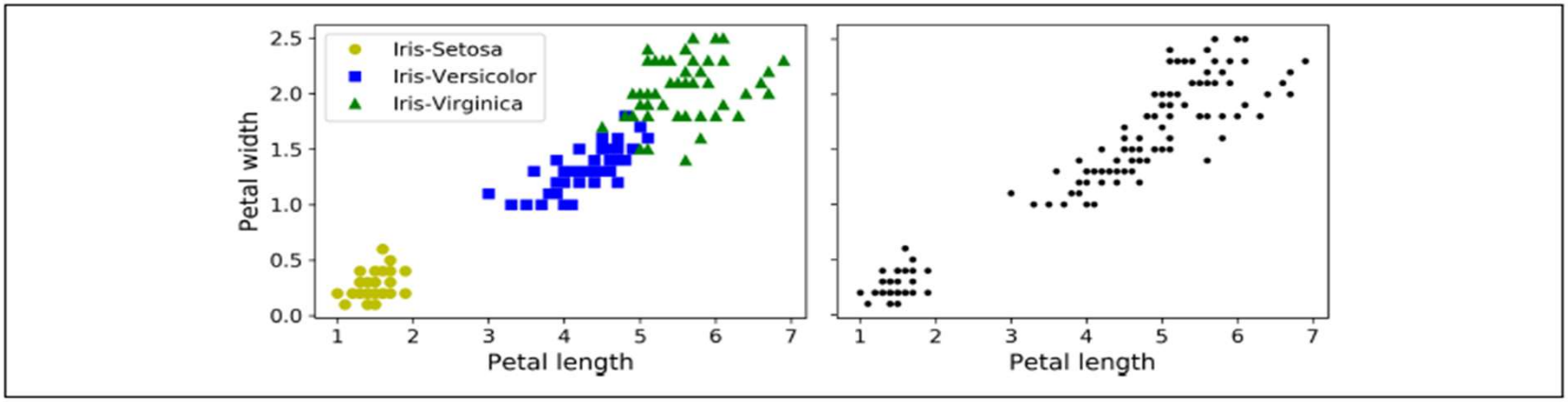


Figure 9-1. Classification (left) versus clustering (right)

비슷해 보이는 instance들을 *clusters*로 묶는것
좌측 그림은 Logistic Regression, SVMs, Random Forest classifiers 등과 같은 classification algorithms 들로 잘 구별이 가능하다.
근데 우측은 좌측과 같은 데이터셋이지만 label되어있지 않음. classification algorithm사용불가능.
-> Clustering algorithm 사용해야함.
좌하단의 cluster는 쉽게 구별이 가능하다. 그러나 우측의 cluster는 두개의 그룹으로 이루어진건지 잘 모를수도 있음. 사실 이 데이터는 여기엔 안나와있지만 sepal length / width 의 2개 feature가 더 있음.
clustering algorithms를 사용하면 3가지 label을 잘 구별 할 수 있음. Gaussian mixture model을 이용해 측정한 결과는 150개 중 5개의 instance만 잘못구별됨.

239 Clustering의 사용처

- 고객 집단 분류용
  - 고객별로 추천해주는 시스템 만들때 좋음
- 데이터 분석용
  - 비슷한 instance끼리 묶어서 cluster분류할때 좋음
- 차원축소용
  - Dataset이 Cluster 된 후에는 각 instance들이 cluster에 얼마나 잘 들어맞는지 쉽게 구별가능함.
- *anomaly detection(=outlier detection)*
  - Instance가 어떠한 cluster와도 잘 들어맞지 않는경우를 확인가능
  - manufacturing, or for *fraud detectio*에 좋음.
- semi-supervised learning
  - 몇가지의 label만 있다면 같은 cluster에 있는 모든 instance들에게도 쉽게 labeling이 가능함. 차후에 이 데이터로 supervised learning algorithm을 할 경우 더 나은 퍼포먼스를 가지게됨

240 Clustering의 사용처2

- 검색엔진
  - 이미지 검색을 예로 들면, 이미지를 넣었을때 clustering algorithm을 database에 적용하게 되면 비슷한 cluster에 있는 이미지들을 추출 할 수 있음.
- 이미지 분리
  - 이미지의 색으로 분류를 할 경우, cluster 집단의 색의 중간값으로 해당 색들을 모두 바꾸면 이미지의 색이 몇가지 안되게 바꿀 수 있음.
  - Object detection, tracking등에 사용됨.

240-241 clustering algorithm 1 : K-Means

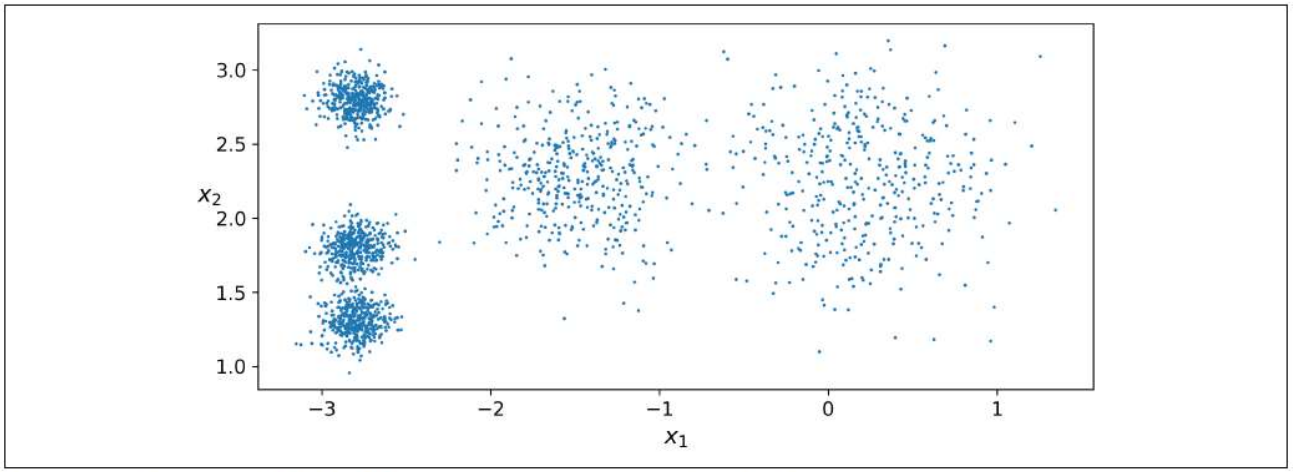


Figure 9-2. An unlabeled dataset composed of five blobs of instances

5가지 부분으로 나눠진 instance들이 있음.
먼저 부분들의 중심을 찾아볼것임.

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

먼저 몇 개의 클러스터를 찾을건지는 사용자가 정의해줘야함.
여기서는 k=5가 될것

각 instance는 5개의 cluster중 하나에 포함됨.
Instance의 *label*은 cluster의 index임. Instance는 algorithm에 의해 해당 index에 할당되게 됨.
Classification의 index와 혼동하지 말것.

242 K-Means

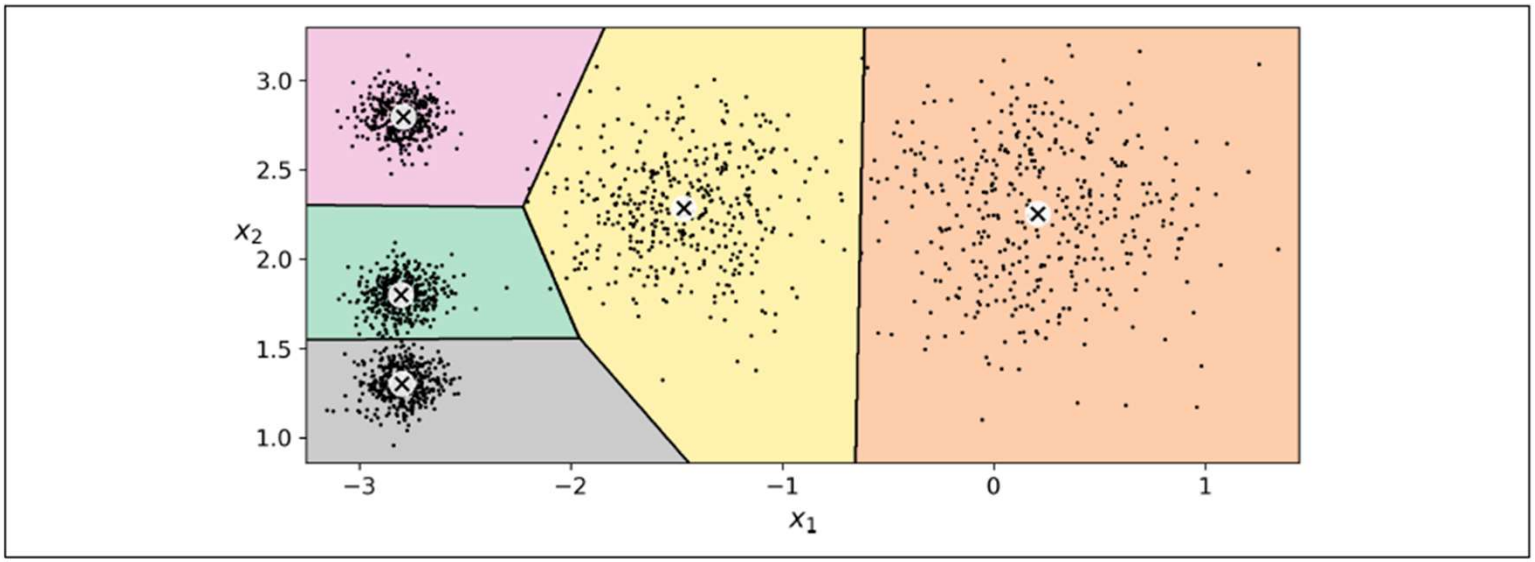


Figure 9-3. K-Means decision boundaries (Voronoi tessellation)

좌측은 K-Means의 Decision Boundary임.
X가 센터를 나타냄

K-Means는 집단들이 다른 사이즈를 가지고 있을 경우에는 잘 작동하지 않음.
-> 센터와 거리만으로 어느집단에 속할지 설정하기 때문

Hard clustering : instance가 어떤집단으로 속할지 정하는것
Soft clustering : instance가 어떤집단의 점수가 가장 높은지 측정하는것

242 K-Means

```
>>> 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]])
```

Transform 이라는 function을 사용하게 되면 각 instance별로 집단의 centroid와 거리를 나타내게 됨.
4개의 instance / 5개의 cluster가 있다면 위와 같이 나올 수 있음.
고차원일 경우 *k*-dimensional dataset 을 얻게될것임.

243 The K-Means Algorithm

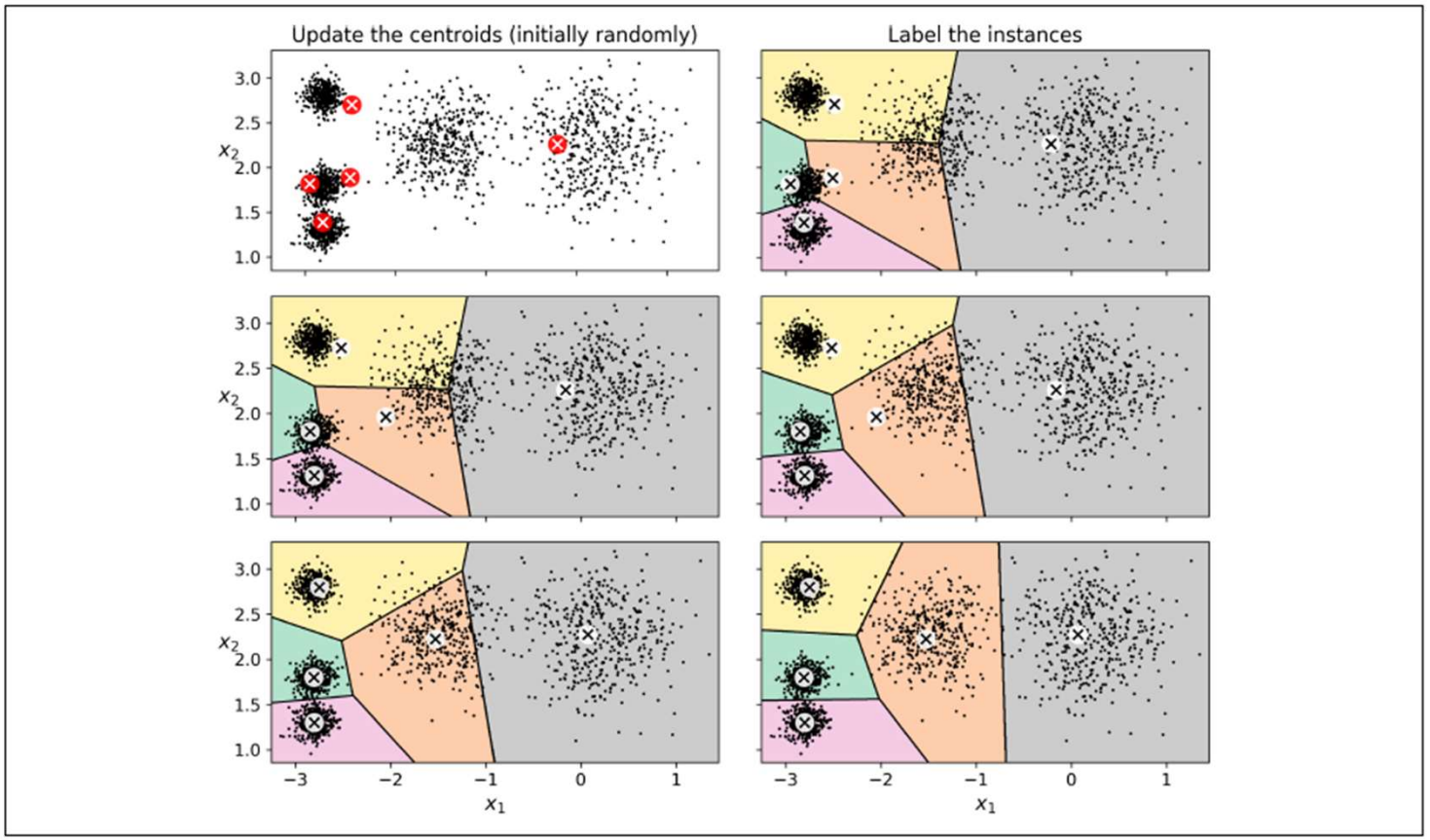


Figure 9-4. The K-Means algorithm

센터를 정하는 방법?
1. Centroid를 랜덤하게 설정해 봄.
2. 각 centroid와 가장 가까운 것부터 label을 붙임(상단) 이렇게되면 모든 instance는 label이 붙게됨.
3. 각 label이 붙은 instance의 평균값에 centroid의 위치로 새로 정함.
4. 위 2~3과정 반복

244 The K-Means Algorithm

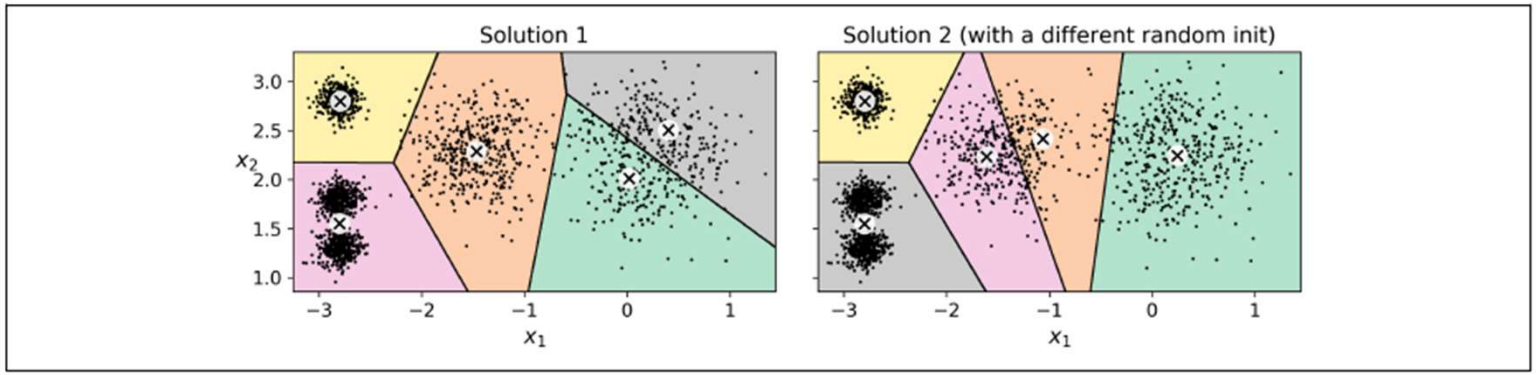


Figure 9-5. Sub-optimal solutions due to unlucky centroid initializations

Random하게 초기 centroid 를 정할때, 잘못되면 위처럼 clustering되는경우도 있음.
해결법은 다음페이지

244 Centroid Initialization Methods

```
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)
```

솔루션 1
대략적인 센터의 초기 위치를 알고 있다면 위와같이 init이라는 hyperparameter를 정하는 방법도 있음. n_init의 값은 1로 바꿔줘야함.

솔루션 2
Cluster 찾는 과정을 여러번 반복하여 가장 좋은 데이터를 사용하는것.
n_init의 값이 몇번 반복하냐임.
어떻게 좋은 데이터인지 아냐?
Inertia 라는것을 이용하는것임.
-> mean squared distance between each instance and its closest centroid
-> 각 인스턴스 & 가장 가까운 센터까지의 평균 제곱 거리
전페이지 좌측은 이 값이 223.3, 우측은 237.5, 잘 나눠졌을때는 211.6임
n_init의 값 만큼 반복하는동안 inertia의 값이 가장 작은것을 취하도록 함.

245 K-Means++

- 2006년에 David Arthur and Sergei Vassilvitskii로부터 제안된 방법. 초기 센터 설정을 다르게 하는것임. 방법은 다음과 같음.
- 1. 센터 한군데를 정함.
- 2. $\sum_{j=1}^{m} D\left(\mathbf{x}^{(j)}\right)^2$ 공식에 따라 새 센터를 정하는데, D(**x**(*j*))는 기존 instance와 1번에서 정해진 센터간의 거리임. 저 값이 큰 위치일수록 적절한 centroid일 확률이 높음.(This probability distribution ensures that instances further away from already chosen centroids are much more likely be selected as centroids.
- 3. 위 과정 반복
- K-means는 저 initialization method가 default임.

245-246 Accelerated K-Means and Mini-batch K-Means

Accelerated K-Means는 현재 K-Means의 default algorithm으로 사용되고 있음. 삼각부등식 등의 방법에 의해 필요없는 연산을 줄이는 방법임. 책에 자세한 설명은 안나와있음.

Mini-batch K-Means은 데이터셋이 너무 커서 메모리 위에 다 올리지 못할 경우 사용하는 방법인데, 보통은 일반 K-Means보다 쓰기가 귀찮다고 함. (여러 배치로 나누어서 한 후 사용자가 가장 좋은 데이터를 선택해야함)

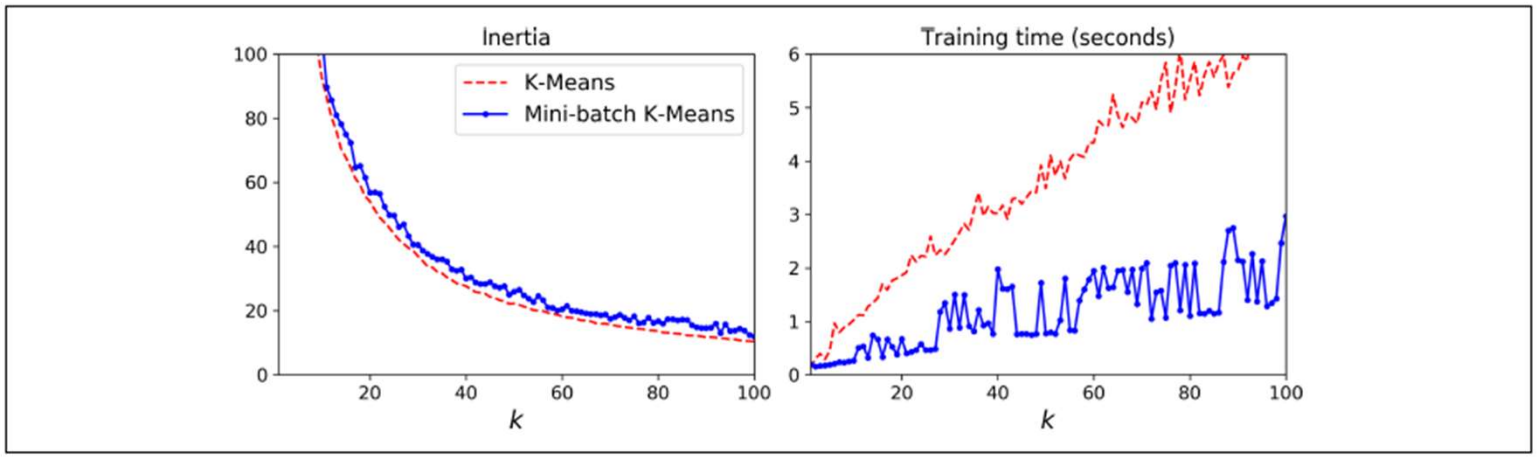


Figure 9-6. Mini-batch K-Means vs K-Means: worse inertia as k increases (left) but much faster (right)

247 Finding the Optimal Number of Clusters

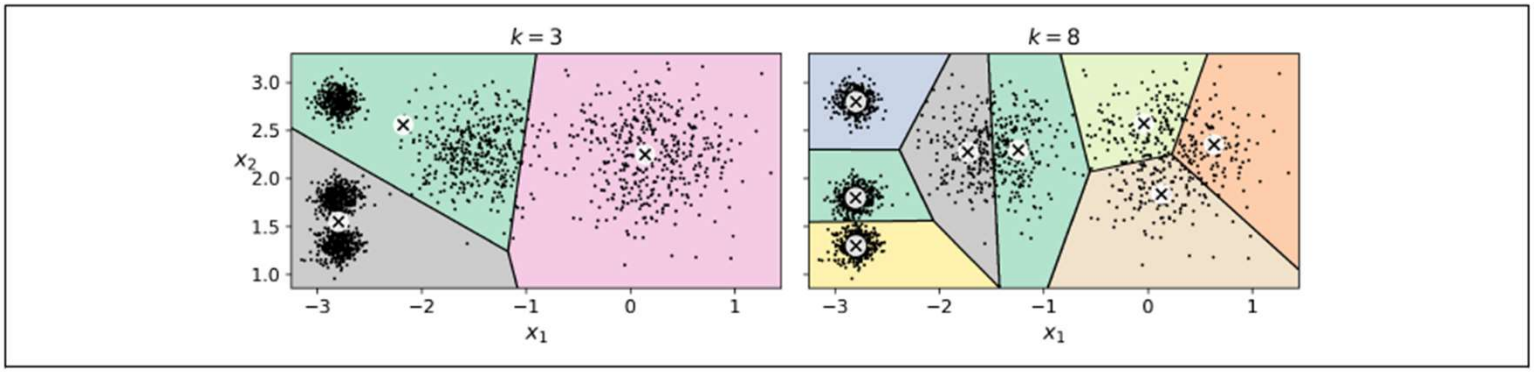


Figure 9-7. Bad choices for the number of clusters

위와같은 데이터는 cluster가 5개라는걸 알기 쉽지만, 실제 많은 데이터들은 그렇지 않음. Cluster를 3 또는 8로 하게되면 위와 같이 됨.
Inertia 값이 k=3일때는 653.2 , k=8일때는 119.1임. 여기서는 inertia로 판단하면 안됨.

248 Finding the Optimal Number of Clusters

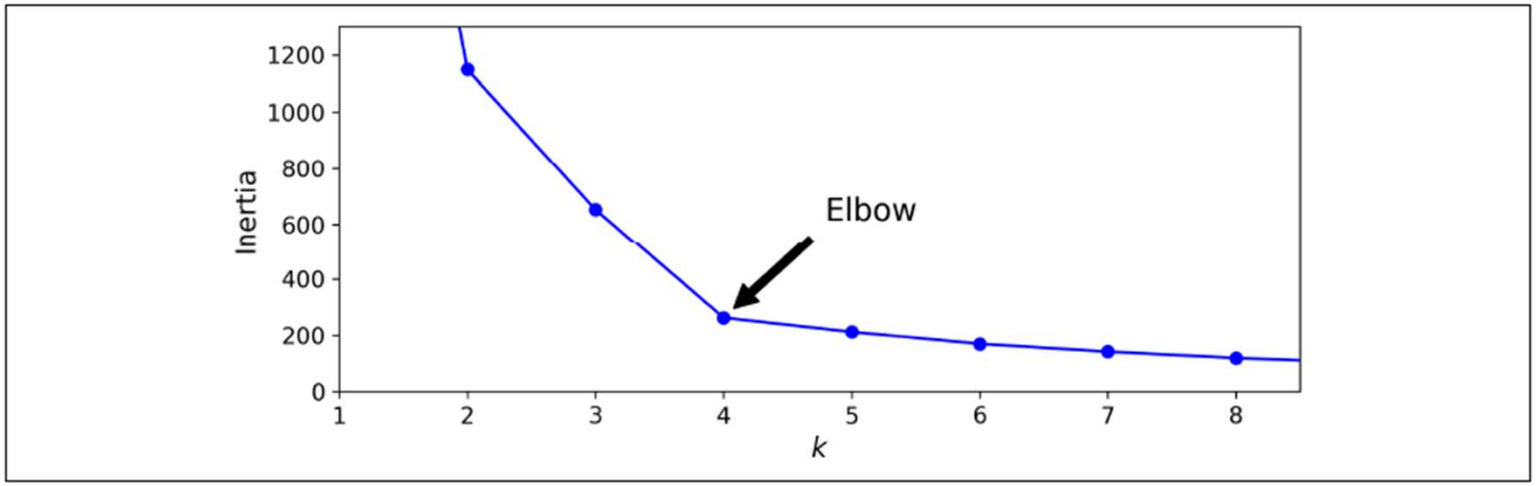


Figure 9-8. Selecting the number of clusters k using the "elbow rule"

보통 위 k-inertia 그래프는 팔 모양인데, 팔꿈치 부분을 찾으면 됨.

그래프 안그리고 $(b - a) / \max(a, b)$ 의 공식(=*silhouette score*)을 이용해서도 찾을 수 있음.

a = instance <-> instance (자기가 속한 cluster 내에서 instance끼리 거리의 평균)

b = instance <-> 자기가 속한 cluster외의 가장 가까운cluster 중심 거리의 평균

이 값은 -1부터 1까지의 값을 갖는다.

1에 가까울수록 자기 자신의 cluster에 머무르고 주변 다른 cluster까지의 거리가 멀다.

0에 가까울수록 다른 cluster와 거리가 가깝다

-1에 가까울수록 잘못된 cluster에 배치된 instance가 많음.

249 Finding the Optimal Number of Clusters

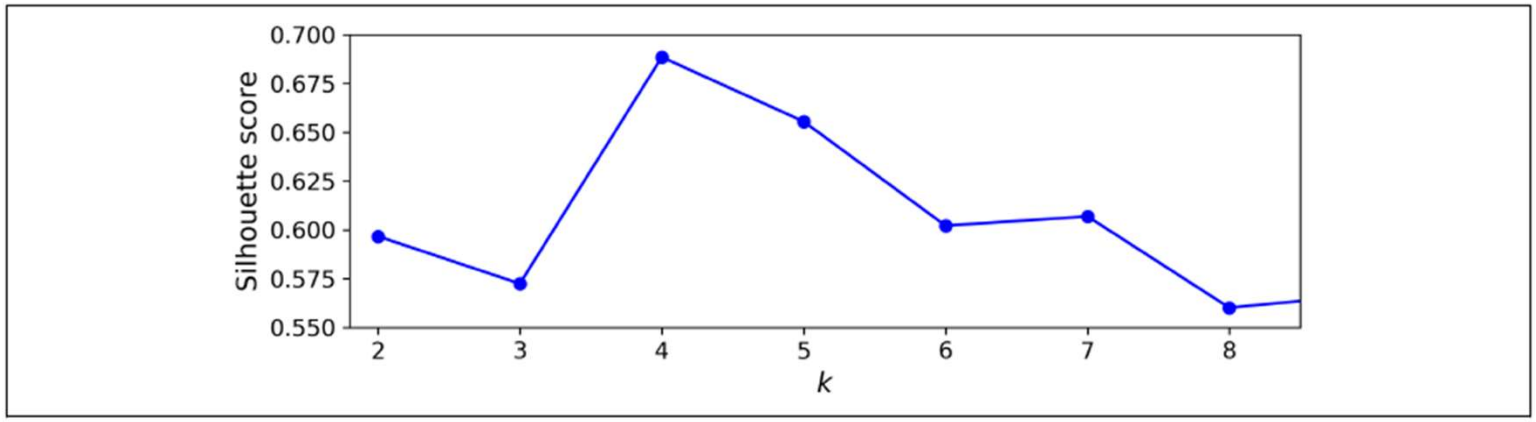


Figure 9-9. Selecting the number of clusters k using the silhouette score

좀 더 보기 쉬워짐

249-250 Finding the Optimal Number of Clusters

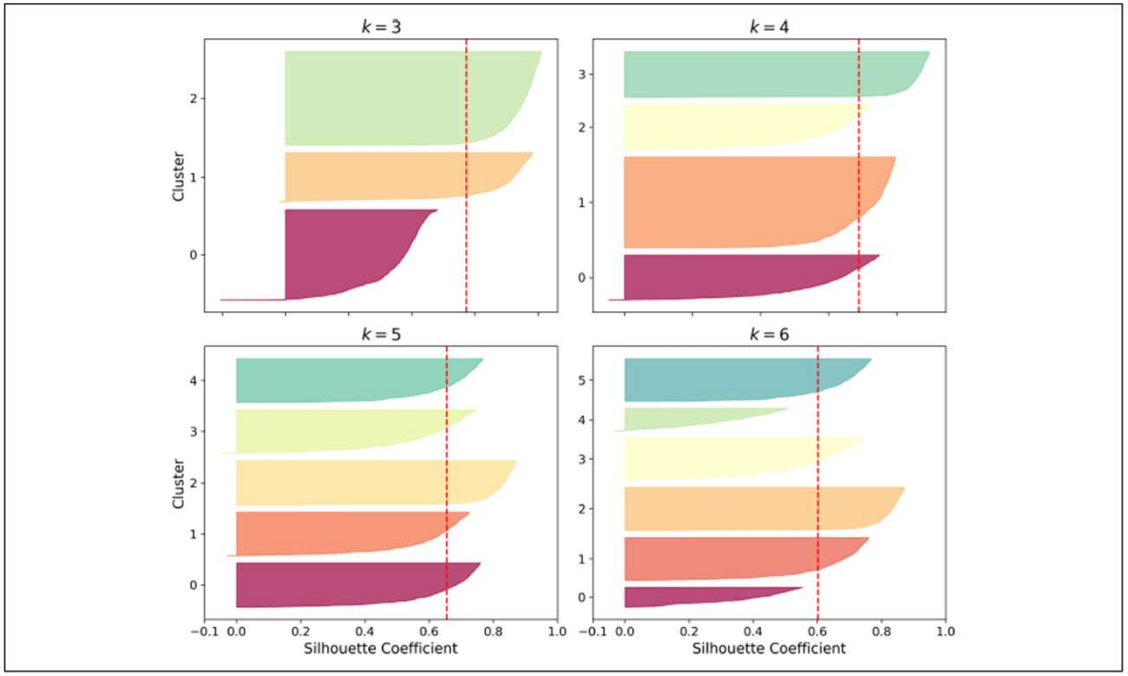


Figure 9-10. Silouhette analysis: comparing the silhouette diagrams for various values of k

Silhouette diagram (실루엣 다이어그램)을 통해 좀 더 나은 시각화를 할 수도 있음.

점선보다 cluster가 뒤에 있다면 instance들이 다른 cluster와 가깝다는 뜻임.
K=3일때와 k=6일때 다른 cluster와 가까운 cluster들이 있음. 이건 안좋은것임.

K=4일때는 모두 점선 밖으로 넘어가긴 하지만 중간에 큰 cluster가 하나 있음. 사이즈가 비슷한 것이 좋음.

250 Limits of K-Means

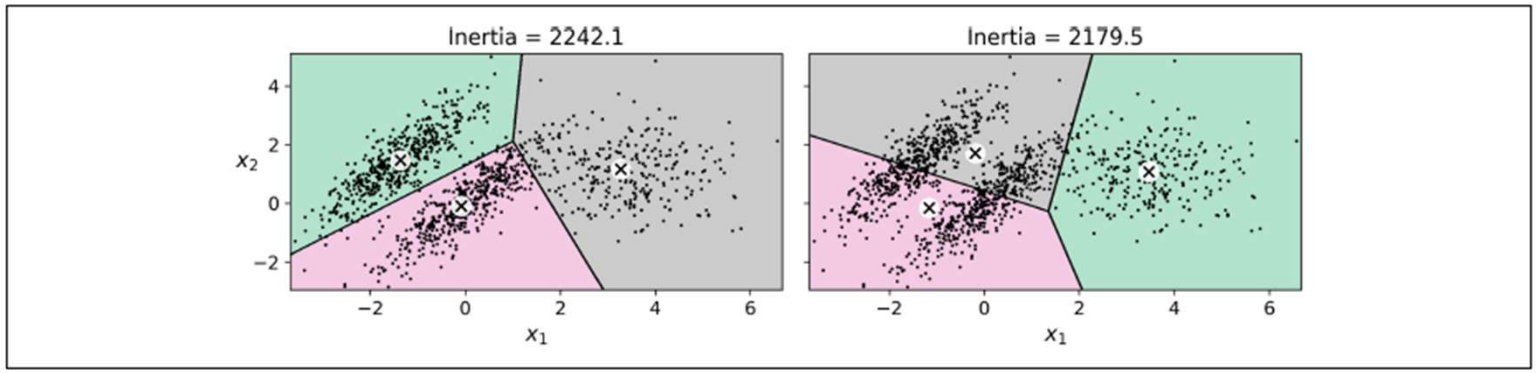


Figure 9-11. K-Means fails to cluster these ellipsoidal blobs properly

단점
1. 여러번 algorithm을 실행해야 optimal 을 얻을 수 있음.
2. Cluster의 갯수를 설정해줘야함
3. 위와 같은 데이터셋에서는 잘 동작하지 않음.

타원형과 같은 데이터셋에서는 Gaussian mixture가 잘 작동함.
우선, clustering의 장점에 대해 먼저 알아볼것임. K-means을 일단 사용할것임.

251 Using clustering for image segmentation

- *instance segmentation*에서는 label된 어떤 object의 모든 pixel은 같은 구역으로 분류되게 된다.
- 우선 *color segmentation*을 먼저 해보자. 같은색은 같은 구역으로 나뉘게 될것임. 몇몇 application에서는 이정도만 해도 될것임.
  - 예를들어 위성사진을 보고 숲의 면적 구할때 등.

```
>>> from matplotlib.image import imread  # you could also use `imageio.imread()`
>>> image = imread(os.path.join("images","clustering","ladybug.png"))
>>> image.shape
(533, 800, 3)
```

- 위와 같이 이미지를 불러온 후 다음페이지

251-252 Using clustering for image segmentation

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

전 페이지에서 불러온 데이터셋은 3D 배열임 : 세로, 가로, color channel의 수
1. RGB형태의 데이터로 바꿈.
2. 비슷한 색깔끼리 K-Means 를 사용하여 clustering시킴.
3. 각 cluster의 중간값으로 해당 cluster의 색을 바꿈.
4. 다시 원래형태의 데이터로 바꿈.

252 Using Clustering for Preprocessing

```
from sklearn.datasets import load_digits

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

Now, let's 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, let's fit a Logistic Regression model:

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_train, y_train)
```

Let's evaluate its accuracy on the test set:

```
>>> log_reg.score(X_test, y_test)
0.9666666666666667
```

Clustering은 preprocessing에도 적용 할 수 있다.
좌측의 MNIST 데이터로부터 Logistic Regression을 그냥 적용했을때 정확도는 0.967이다.

여기에 Clustering을 하게되면 정확도가 어떻게될지는 다음페이지.

253 Using Clustering for Preprocessing

```
from sklearn.pipeline import Pipeline

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

중간에 k-means만 추가했을뿐인데
정확도는 0.982로 상승함.

Now let's evaluate this classification pipeline:

```
>>> pipeline.score(X_test, y_test)
0.9822222222222222
```

```
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)
```

여기에 GridSearchCV를 통해 더 좋은
n_cluster값을 찾는다면
정확도는 또 0.984로 상승

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

```
>>> grid_clf.best_params_
{'kmeans__n_clusters': 90}
>>> grid_clf.score(X_test, y_test)
0.9844444444444445
```

254 Using Clustering for Semi-Supervised Learning

```
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)
0.8266666666666667
```

```
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]
```

Figure 9-13 shows these 50 representative images:

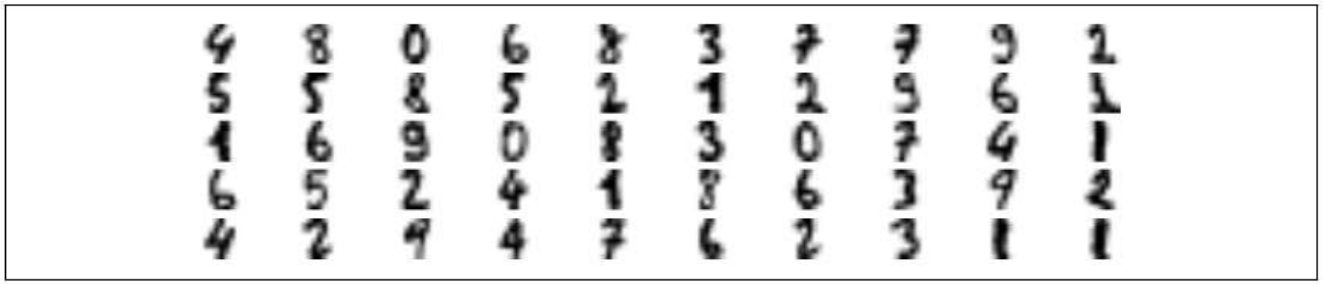


Figure 9-13. Fifty representative digit images (one per cluster)

Semi-Supervised Learning에 적용해보자.
기본 정확도는 0.827임

클러스터 50개로 K-Means적용함.
각 클러스터별 이미지는 아래와 같음.

254-255 Using Clustering for Semi-Supervised Learning

Now let's look at each image and manually label it:

```
y_representative_digits = np.array([4, 8, 0, 6, 8, 3, ..., 7, 6, 2, 3, 1, 1])
```

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

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_representative_digits, y_representative_digits)
>>> log_reg.score(X_test, y_test)
0.9244444444444444
```

각 이미지별 수동으로 라벨링을 한 후, 다시 피팅하여 점수를 내보면
0.924가 나옴
50개의 이미지만 트레인 했을 뿐인데 정확도는 굉장히 많이 올라갔음.

255 Using Clustering for Semi-Supervised Learning

```
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)
0.9288888888888889
```

정확도를 더 올리기 위해서 같은 cluster에 있는 모든 instance에게 label을 전파하여 다시 fit해봄.

-> 정확도는 더 상승

근데 boundary에 있는것들도 모두 같은 label로 매겨졌을것임.

255 Using Clustering for Semi-Supervised Learning

```
percentile_closest = 20

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

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

Now let's train the model again on this partially propagated dataset:

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
>>> log_reg.score(X_test, y_test)
0.9422222222222222
```

Centroid와 가까이 있는 20%에게만 label을 할당하여 다시 fit해봄

정확도는 0.942

기존 모든 instance들이 label데이터를 가지고 있는 supervised learning의 경우와 굉장히 비슷한 정확도가 나왔음.

256 Using Clustering for Semi-Supervised Learning

Active Learning

To continue improving your model and your training set, the next step could be to do a few rounds of *active learning*: this is when a human expert interacts with the learning algorithm, providing labels when the algorithm needs them. There are many different strategies for active learning, but one of the most common ones is called *uncertainty sampling*:

- The model is trained on the labeled instances gathered so far, and this model is used to make predictions on all the unlabeled instances.
- The instances for which the model is most uncertain (i.e., when its estimated probability is lowest) must be labeled by the expert.
- Then you just iterate this process again and again, 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 (e.g., an SVM, a Random Forest, and so on).

더욱 개선하기 위해선 Active Learning을 해야함.
사람이 뭔가를 배울때처럼, label이 필요한 경우에만 제공하는것임.
종류에는 여러가지가 있는데 가장 흔한건 *uncertainty sampling* 라고 불림.

1. 몇가지 label 된 instance들을 train하여 unlabel된 instance들을 prediction함.
2. 모호한 instance들은 사용자에 의해 label 되어야함
3. 위 과정 반복

다른방법으로는 가장 많은 모델을 바꿀만한 instance는 직접 label 하거나, 가장 크게 validation error에 영향이 갈 것 같은것은 직접 label 하는 방법 등이 있음.

256-257 DBSCAN

- 각 instance별로 ε 내의 거리에서 얼마나 많은 다른 instance들이 있는지 체크함. -> 이 거리를 *ε-neighborhood* 라고 부름.
- min_samples갯수 이상의 instance와 인접해있는 instance들은 *core instance*라고 부름. -> 보통 밀도가 높은곳에 있음.
- core instance의 neighborhood인 instance들은 같은 label로 분류됨.
- 주변에 어떠한 core instance도 없다면 anomaly(이상)로 분류됨.

257 DBSCAN

DBSCAN은 밀도가 높을때 잘 작동하는 모델임.
챕터 5에서 했던 달의 MNIST 를 이용하여 적용해봄.
-1로 label된것들은 이상데이터로 분류된것들임.

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

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

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

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

```
>>> len(dbscan.core_sample_indices_)
808
>>> dbscan.core_sample_indices_
array([ 0,  4,  5,  6,  7,  8, 10, 11, ..., 992, 993, 995, 997, 998, 999])
>>> dbscan.components_
array([[-0.02137124,  0.40618608],
       [-0.84192557,  0.53058695],
                  ...
       [-0.94355873,  0.3278936 ],
       [ 0.79419406,  0.60777171]])
```

총 1000개의 데이터 중에 808개가 core data로 분류되었음.

257-258 DBSCAN

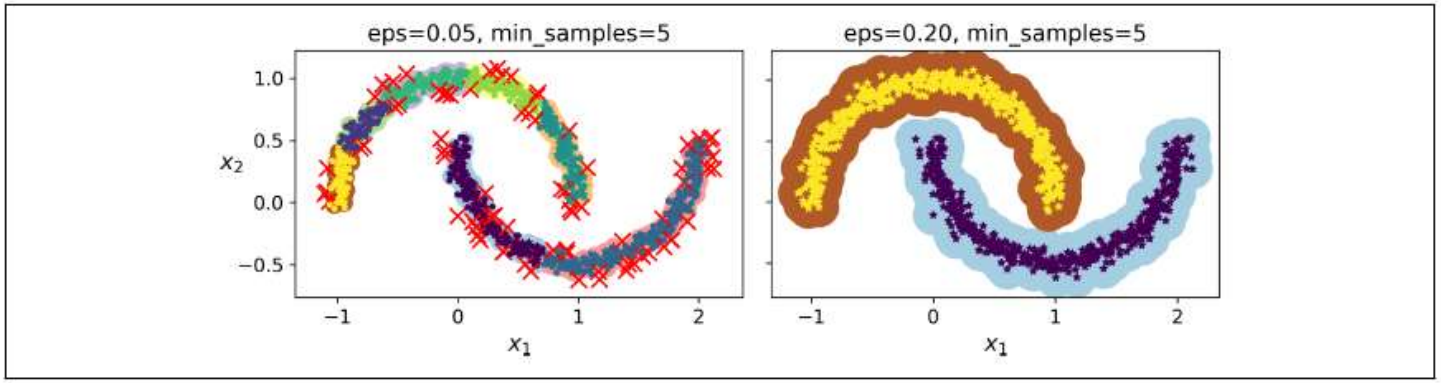


Figure 9-14. DBSCAN clustering using two different neighborhood radiuses

좌측은 hyperparameter 값 설정이 제대로 되지 않아 구역이 7개로 나뉘었음
앱실론값을 올릴수록 cluster가 커짐

258-259 DBSCAN

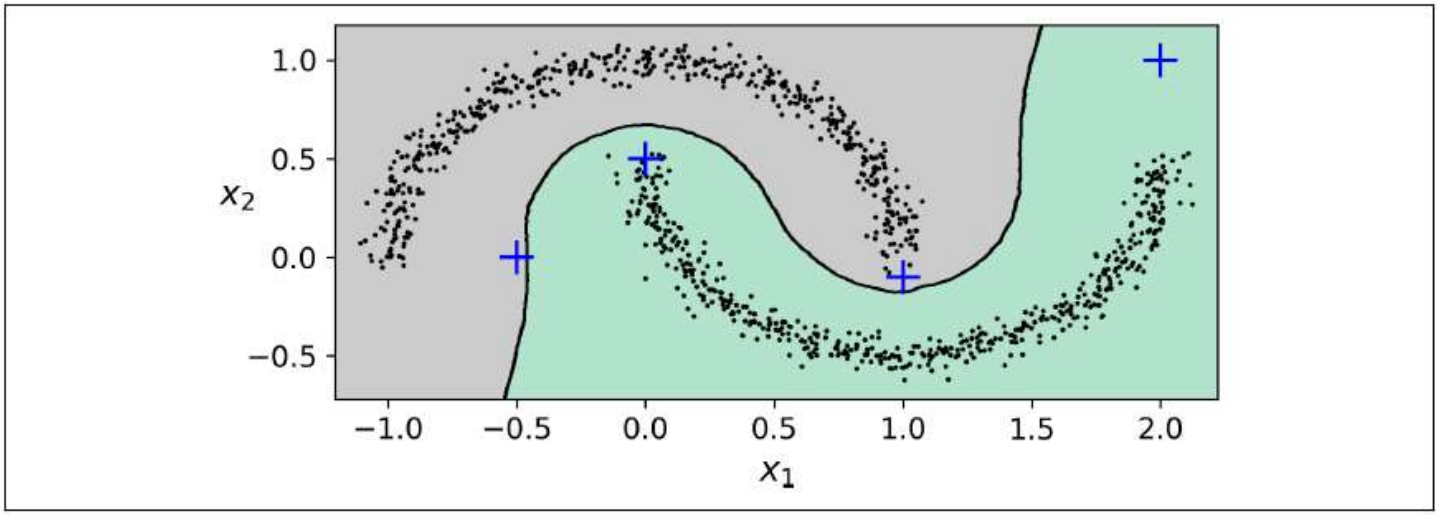


Figure 9-15. cluster_classification_diagram

십자마크는 새로 적용할 instance들임.

```
>>> 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])
```

Kneighbors 매소드를 이용하여 최대거리를 설정하여 이상데이터인지 아닌지 나눌 수 있음

259 DBSCAN의 문제점

- 밀도가 다양하면(작은 원, 큰 원들이 같이 있는 경우) 클러스터 찾기가 불가능
- 앱실론 값이 크면 계산양이 많아짐

259-260 Other Clustering Algorithms

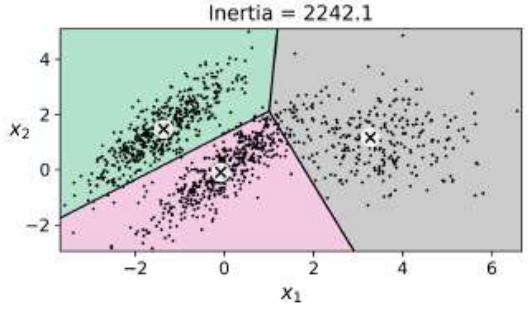
- 1. *Agglomerative clustering*
  - 작은 cluster 들을 큰 뭉치로 합치는것임.
  - Instance나 cluster들이 많을때 유용함
- 2. *Birch*
  - very large datasets 용으로 디자인됨
  - 나무형 구조임. 그때그때 새로운 instance들을 cluster에 할당시킴
  - 메모리는 적은데 다뤄야 하는 데이터는 클때 사용
- 3. *Mean-shift*
  - 각 instance별로 어떤 원의 중심이 되게끔 배치함. 그리고 원 안에 들어오는 다른 instance들까지 포함하여 원 위치의 평균값에 새로운 원을 또 만듦. 원 위치가 바뀌지 않을때까지 반복
  - DBSCAN과 비슷함. 어떤형태든 다 찾을 수 있음. 1개의 hyperparameter만 필요함. – 원의 반지름
  - 큰 데이터셋은 어울리지 않음

260 Other Clustering Algorithms

- 4. *Affinity propagation*
  - voting system을 이용함
  - 비슷한 성질을 가진 instance들중 대표값을 설정함.
  - 이 대표값을 중심으로 cluster을 형성함
  - 다른 크기의 cluster도 찾을 수 있음
  - 큰 데이터셋에는 어울리지 않음
- 5. *Spectral clustering*
  - Instance별 metrix을 만들고 low-dimensional embedding을 함.
  - 그렇게 해서 만들어진 저차원의 데이터에 다른 clustering algorith을 적용함 (K-means등)
  - 복잡한 cluster 구조도 잡아낼 수 있음.
  - 큰 데이터셋에는 어울리지 않음. 클러스터 사이즈가 달라도 잘 작동하지 않음.

260-261 Gaussian Mixtures

- Gaussian distribution의 혼합을 통하여 instance들이 만들어졌다고 가정함.
- Single gaussian distribution 에 의해 만들어진 데이터는 타원형 태임.
- 사진과 같은 데이터처럼 크기, 밀도, 형태, 방향등이 다 다름.
- 파라미터는 모르는상태
- 몇 가지 종류의 변형된 형태가 있음



261-262 Gaussian Mixtures

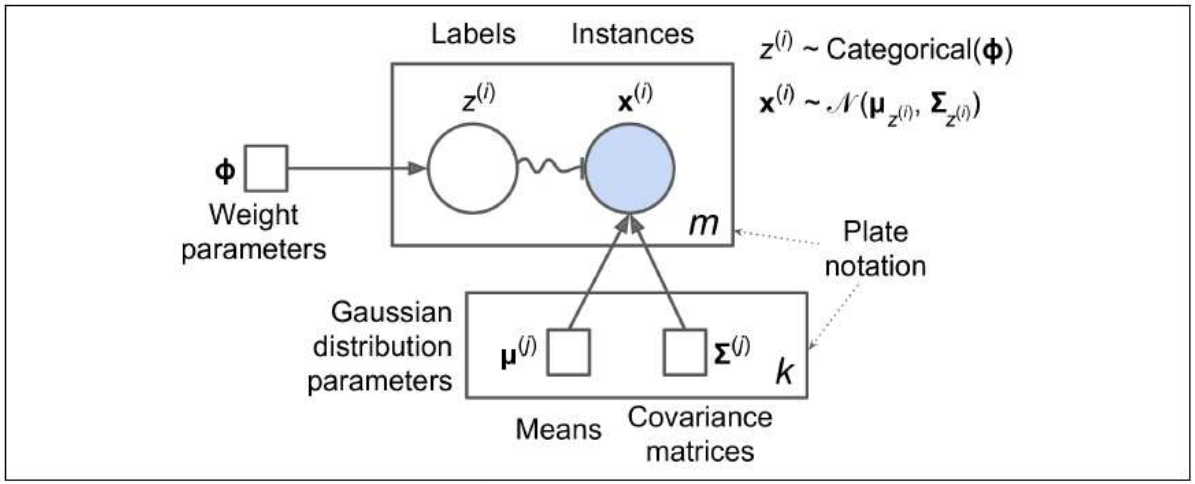


Figure 9-16. Gaussian mixture model

해석 : 1.원은 random value를 의미
2.작은 사각형은 fixed value를 의미
3.큰 사각형은 *plates*라고 불림. 이 내부 내용물은 계속 반복됨.
4.큰 사각형 우하단의 문자는 반복횟수.
5.즉 m개의 랜덤 변수 $z(i)$, $\mathbf{x}(i)$ 가 있음.
6.k는 $\boldsymbol{\mu}(j)$ 와 k covariance matrices $\boldsymbol{\Sigma}(j)$를 의미함
7.근데 $\boldsymbol{\phi}$는 하나밖에 없음. -> $\phi(1)$ 부터 $\phi(k)$ 까지 모두 포함

Gaussian Distribution number k 를 알아야함.
Dataset은 다음에 의해 만들어졌다는 가정이 필요하다.

1. 각 instance들은 k개의 cluster 중에 랜덤으로 정해진다. j번째 cluster가 골라질 가능성은 cluster weight 에 의해 정해진다. i번째 instance가 포함된 Cluster의 index는 $z(i)$라고 한다.
2. 만약 $z(i)$=j 라면(=i번째 instance가 j번째 cluster에 포함이 됐다면) instance i 의 위치$\mathbf{x}(i)$는 gaussian distribution(mean $\boldsymbol{\mu}(j)$ 와 covariance matrix $\boldsymbol{\Sigma}(j)$에 의해 정해짐)에 의해 랜덤하게 정해진다.

좌측은 생성 프로세스임.

262 Gaussian Mixtures

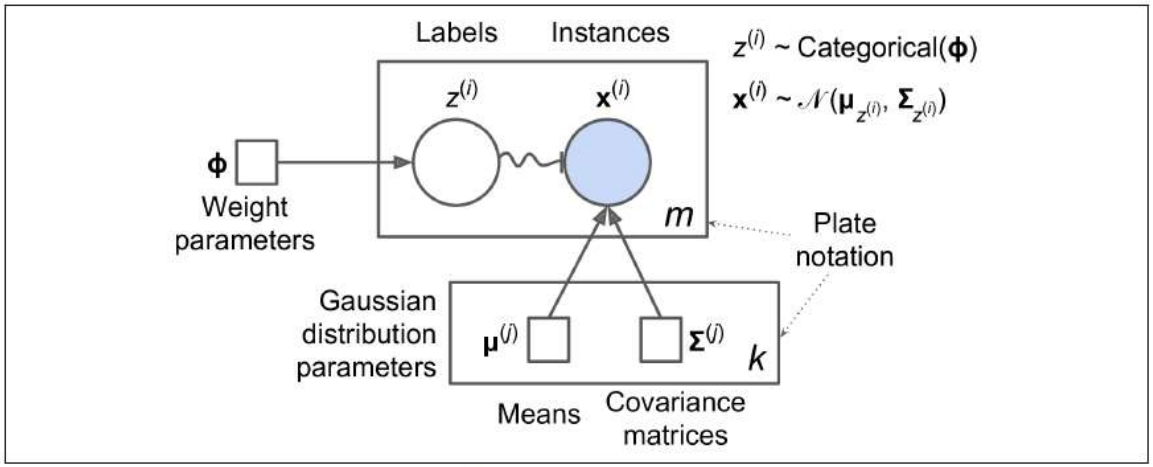


Figure 9-16. Gaussian mixture model

8.변수 $z(i)$는 weight $\boldsymbol{\phi}$에 의해 정해짐. $\mathbf{x}(i)$는 normal distribution에 의해 정해지는데, 이는 $z(i)$번째 cluste에 의해 정해지는 mean 과 covariance Matrix에 따름.

9.직선 화살표는 조건부 의존성(conditional dependencies)을 의미. 화살표가 plate의 경계를 넘는것은 모든 반복행위에도 동일하게 적용됨.

10.구불구불한 화살표는 스위치를 의미. $z(i)$값에 의해서 $\mathbf{x}(i)$가 샘플링 될 것임.

-> 예를 들자면 if $z^{(i)}=j$, then $\mathbf{x}^{(i)} \sim \mathcal{N}\left(\mu^{(j)}, \Sigma^{(j)}\right)$.

11. Shaded node는 해당 값을 모르는상태.

-> 여기서는 아는값이 $\mathbf{x}(i)$밖에 없음. *observed variables*라고 불림.

모르는값 $z(i)$는 *latent variables*라고 불림.

262-264 Gaussian Mixtures

```
from sklearn.mixture import GaussianMixture

gm = GaussianMixture(n_components=3, n_init=10)
gm.fit(X)
```

Let's look at the parameters that the algorithm estimated:

```
>>> gm.weights_
array([0.20965228, 0.4000662 , 0.39028152])
>>> gm.means_
array([[ 3.39909717,  1.05933727],
       [-1.40763984,  1.42710194],
       [ 0.05135313,  0.07524095]])
>>> gm.covariances_
array([[[ 1.14807234, -0.03270354],
        [-0.03270354,  0.95496237]],

       [[ 0.63478101,  0.72969804],
        [ 0.72969804,  1.1609872 ]],

       [[ 0.68809572,  0.79608475],
        [ 0.79608475,  1.21234145]]])
```

해당 알고리즘은 K-means와 비슷하다.

1. Cluster parameter 들을 랜덤하게 설정한다.
2. Cluster들을 업데이트 한다.
3. 위 과정 반복

각 instance별로 알고리즘이 각 cluster에 속할 확률을 계산한다.
그 후에 cluster의 업데이트시 해당 cluster에 속하게 될 가장 높은 가능성을 가지는 instance들에 영향을 받게 된다.

K-means와 같이 이상한 형태로 끝날수도 있음.
n_init을 10으로 하면 10번중 가장 괜찮은 결과가 나오게됨. 디폴트값은 1임

predict()를 통해 hard clustering
predict_proba()를 통해 soft clustering 가능함.

이는 *generative model* 라고 불림.
-> 새로운 instance를 샘플링 할 수 있음.

264 Gaussian Mixtures

```
>>> X_new, y_new = gm.sample(6)
>>> X_new
array([[ 2.95400315,  2.63680992],
       [-1.16654575,  1.62792705],
       [-1.39477712, -1.48511338],
       [ 0.27221525,  0.690366  ],
       [ 0.54095936,  0.48591934],
       [ 0.38064009, -0.56240465]])

>>> y_new
array([0, 1, 2, 2, 2, 2])

>>> gm.score_samples(X)
array([-2.60782346, -3.57106041, -3.33003479, ..., -3.51352783,
       -4.39802535, -3.80743859])
```

좌측은 새 instance 샘플링하는것임.

좌측 하단처럼 각 위치별 density도 측정가능.
값이 높을수록 밀도가 높은것임.
Exponential 을 위치별로 *probability density function* (PDF) 를 측정 할 수 있음.