비지도학습

In [2]:

```
from matplotlib import font_manager, rc
import matplotlib as mpl
font_path = "C:WWUsersWW이혜림WWDesktopWWBita5/malgun.ttf"
font_name = font_manager.FontProperties(fname=font_path).get_name()
rc("font",family=font_name)
mpl.rcParams["axes.unicode_minus"]=False

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
import os
```

1. 군집
   - 비슷한 샘플을 클러스터로 모음
   - 군집은 데이터 분석, 고객 분류, 추천 시스템, 검색 엔진, 이미지 분할, 준지도 학습 차원 축소 등에 사용
2. 이상치 탐지
   - '정상' 데이터가 어떻게 보이는지 학습한 후 비정상 샘플을 감지하는 데 사용
   - 제조 라인에서 결함 제품을 감지하거나 시계열 데이터에서 새로운 트렌드를 찾음.
3. 밀도 추정
   - PDF(확률 밀도 함수)를 추정하여 밀도가 매우 낮은 영역에 놓인 샘플을 이상치로 탐지

1. 군집

- 군집 : 비슷해 보이는 샘플을 구별해 각 클러스터에 할당하는 것
- 클러스터 : 비슷한 샘플을 모아놓은 그룹

- 활용 애플리케이션
  - 고객분류 : 고객을 구매 이력이나 웹사이트 내 행동 등을 기반으로 클러스터로 모음. 클러스터마다 제품 추천이나 마케팅 전략을 다르게 적용.
    - ex) 동일한 클러스터 내의 사용자가 좋아하는 컨텐츠를 추천하는 추천 시스템을 만들 수 있음
  - 데이터 분석 : 새로운 데이터셋을 분석할 때 군집 알고리즘을 실행하고 각 클러스터를 따로 분석함으로써 도움
  - 차원 축소 기법 : 군집 알고리즘을 적용하여 각 샘플의 클러스터에 대한 친화성(샘플이 클러스터에 얼마나 잘 맞는지를 측정)을 측정. 각 샘플의 feature을 k개의 클러스터에 대한 친화성으로 바꿈.
    - ex) k개의 클러스터 -> 각 샘플 당 k개의 feature
  - 이상치 탐지 : 모든 클러스터에 친화성이 낮은 샘플-> 이상치일 가능성이 높음
    - 활용 : 제조분야의 결함 / 부정 거래 감지
  - 준지도 학습 : 레이블된 샘플이 적다면 군집을 수행하고 동일한 클러스터에 있는 모든 샘플에 레이블을 전파.

- 이미지 검색 엔진 : 데이터 베이스의 모든 이미지에 군집 알고리즘을 적용한 후 사용자가 찾으려는 이미지를 제공하면 훈련된 군집 모델을 사용해 해당 이미지의 클러스터를 찾음. 그 후 이 클러스터의 모든 이미지를 반환
- 이미지 분할 : 색을 기반으로 픽셀을 클러스터로 모음. 각 픽셀의 색을 해당 클러스터의 평균 색으로 바꿈으로써 이미지에 있는 색상의 종류를 크게 줄임. 물체의 윤곽을 감지하기 쉬워져 물체 탐지 및 추적 시스템에서 많이 사용

1.1 k-means

- 하드 군집 : 샘플을 하나의 클러스터에 할당
- 소프트 군집 : 클러스터마다 샘플에 점수를 부여
  - 점수 : 유사도점수(가우시안 방사 기저 함수와 같은) / 거리

가우시안 방사 기저 함수 : 각 데이터를 기준으로 하여 유사도를 측정하는 방법

- gamma는 하나의 데이터 샘플이 영향력을 행사하는 거리를 결정
- 영향력이 작으면, 거리가 짧으면 제일 바깥(결정경계 가까이에 있는) 데이터의 영향을 많이 받기 때문에 점점 구불구불해짐.

군집화용 데이터 생성기

- make_blobs : 개별 군집의 중심점과 표준 편차 제어 기능이 추가되어 있음
  - n_samples : 생성할 총 데이터의 개수
  - n_features : 데이터 피처 개수
  - centers : int 값이면 알아서 각 군집 당 센터를 만듦, ndarray로 할 경우 개별 군집 중심점의 좌표가 됨
  - cluster_std : 군집 데이터의 표준편차, 그냥 float형으로 하면 모든 군집이 같은 표준편차, ndarray로 할 경우, 군집 당 다른 표준편차
- make_classification : 노이즈를 포함한 데이터를 만드는데 유용
- 이 외
  - make_circle(), make_moon() : 중심 기반의 군집화로 해결하기 어려운 데이터셋을 제공

In [3]:

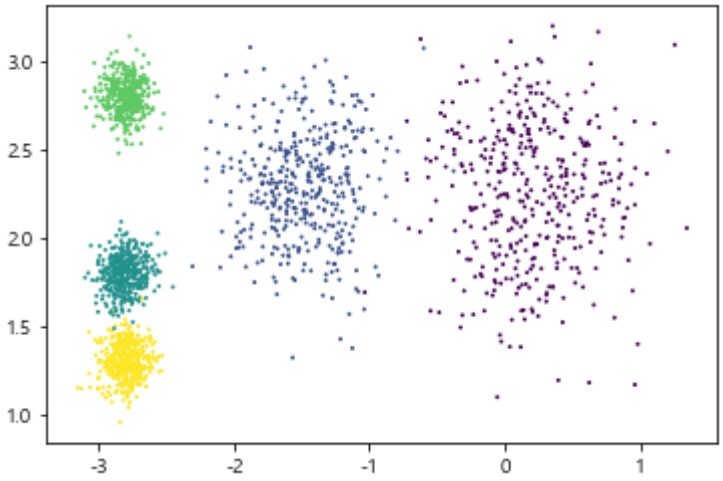
```
from sklearn.datasets import make_blobs # 데이터셋을 만드는 함수
blob_centers = np.array( # 클러스터의 기준이 될 중심점을 생성, 이 중심점에 따라 y label이 달라짐
    [[ 0.2,  2.3],
     [-1.5 ,  2.3],
     [-2.8,  1.8],
     [-2.8,  2.8],
     [-2.8,  1.3]])

blob_std = np.array([0.4, 0.3, 0.1, 0.1, 0.1]) # 각 중심점에 대한 분산을 정함

X, y = make_blobs(n_samples=2000, centers=blob_centers,
                  cluster_std=blob_std, random_state=7)
```

In [24]:

```
plt.scatter(x=X[:,0], y=X[:,1], c=y, s=1)
```

Out[24]:

```
<matplotlib.collections.PathCollection at 0x195a9be2ac8>
```



In [27]:

```
from sklearn.cluster import KMeans
k = 5
kmeans = KMeans(n_clusters = k) # 찾을 클러스터의 개수 지정
y_pred = kmeans.fit_predict(X) # 비지도학습이기 때문에 y label이 필요하지 않음

# kmeans.labels_ : 훈련된 샘플의 레이블을 가지고 있음
print(y_pred is kmeans.labels_)
# 센트로이드 또한 확인 가능
print(kmeans.cluster_centers_)
```

```
True
[[-2.79290307  2.79641063]
 [ 0.20876306  2.25551336]
 [-2.80037642  1.30082566]
 [-1.46679593  2.28585348]
 [-2.80389616  1.80117999]]
```

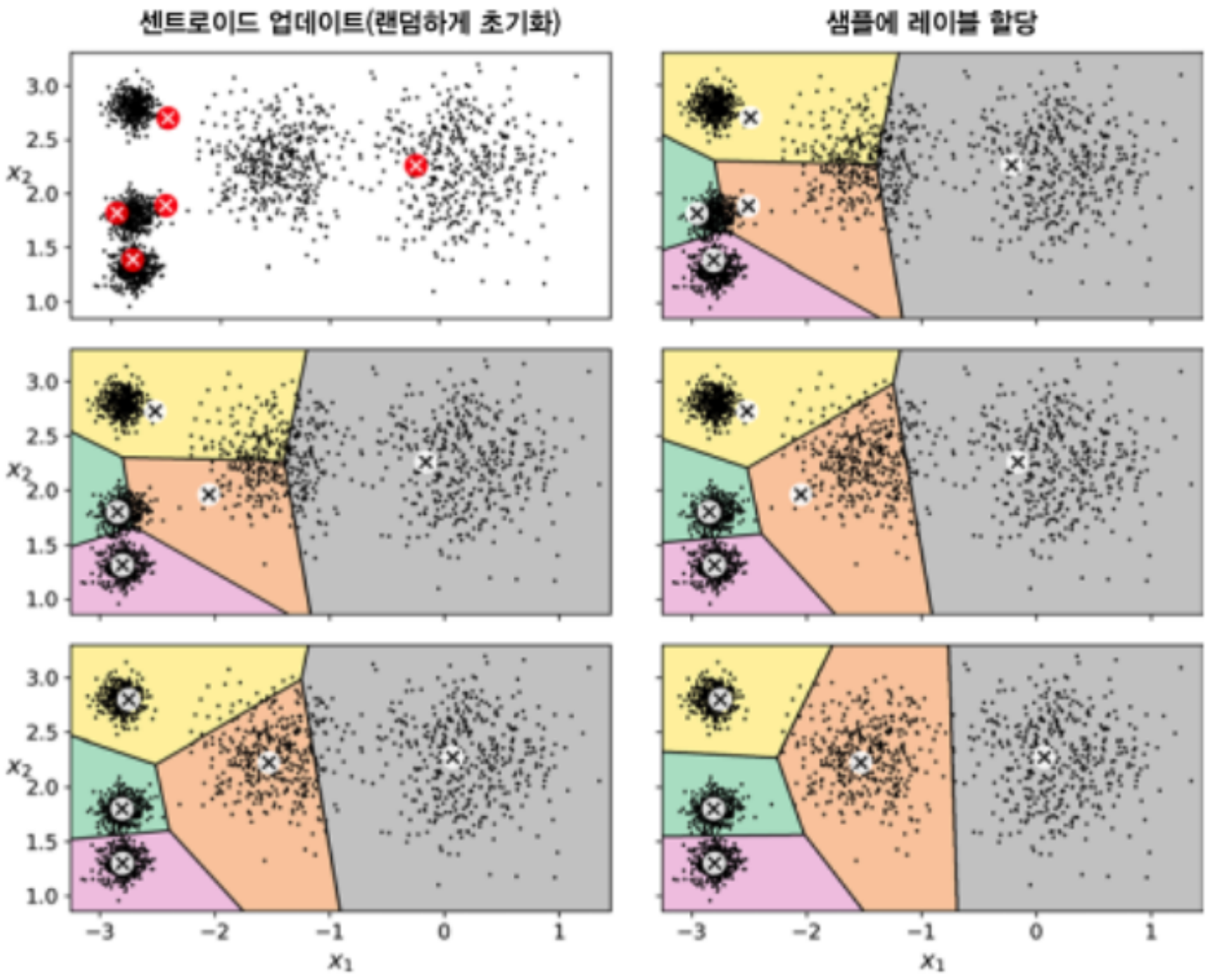
In [29]:

```
X_new = np.array([[0,2],[3,2],[-3,3],[-3,2.5]])
print(kmeans.predict(X_new)) # 예측 label
print(kmeans.transform(X_new)) # 각 클러스터의 센트로이드까지의 거리 -> feature로 사용할 수 있음
# 고차원의 데이터에 대한 차원축소의 기능
```

```
[1 1 0 0]
[[2.9042344  0.32995317 2.88633901 1.49439034 2.81093633]
 [5.84739223 2.80290755 5.84236351 4.4759332  5.80730058]
 [0.29040966 3.29399768 1.71086031 1.69136631 1.21475352]
 [0.36159148 3.21806371 1.21567622 1.54808703 0.72581411]]
```

1.1.1) k-평균 알고리즘

- 센트로이드를 랜덤하게 선정
- 샘플에 레이블을 할당하고 센트로이드를 업데이트 하는 과정을 반복
- 센트로이드에 변화가 없을 때까지 계속
  - 샘플과 가장 가까운 센트로이드 사이의 평균 제곱 거리가 매 단계마다 작아지기 때문에 제한된 횟수 안에 수렴하는 것을 보장(만약 데이터가 군집할 수 있는 구조)
- 시간복잡도 : 일반적으로 샘플 개수 m, 클러스터 개수, k, 차원 개수 m에 선형적
  - 가장 빠른 군집 알고리즘 중 하나
    - 초기화가 잘 되어있고, 데이터가 군집할 수 있는 구조라면



1.1.2 센트로이드 초기화 방법

- 센트로이드의 초기화가 어떻게 되느냐에 따라 적절한 솔루션이 될 수도, 적절하지 못한 솔루션이 될 수도 있음
- 초기화를 적절하게 해주는 것이 중요
- n_init : 랜덤 초기화 횟수, default : 10
  - 초기화 랜덤 값이 10개, 이 중에서 최선의 솔루션을 고름
  - 이 때 사용하는 성능 지표 : **이니셔**
  - *이니셔 : 각 샘플과 가장 가까운 센트로이드 사이의 평균 제곱 거리*
    - 이 값이 작으면 작을 수록 각 군집 데이터의 중앙에 위치했다고 생각하기 때문에 작으면 작을 수록 선호
    - 사이킷런은 score 값이 크면 클 수록 좋다고 인식하기 때문에 score 변수의 값이 음수

In [35]:

```
# n_init : 랜덤 초기화 횟수, default : 10
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)
kmeans.fit(X)

print(kmeans.inertia_)
print(kmeans.score(X))
```

211.5985372581684
-211.59853725816856

1.1.2.1 K-평균++ 알고리즘

- initial points로 비슷한 점들이 여러 개 선택된 경우 k-means는 불안정
- 따라서 initial points를 넓게 퍼지게 만들어줌
- initial points 선정의 default 값

1. 데이터셋에서 무작위로 균등하게 하나의 센트로이드 e(1)을 선택
2. 이후의 initial point는 이전에 선택한 C_{t-1}과의 거리인 $d(C_{t-1}, c_t)$가 큰점이 다음 initial points로 뽑힐 확률을 높여주기 위해, 확률 분포를 $(d(C_{t-1}, c_t)/\Sigma d(C_{t-1}, c_t)$ 으로 조절함.
3. 이 분포에 따라 하나의 점을 다음 initial point로 선택
4. k개의 initial points를 선택할 때까지 2~3을 반복

문제점

- sparse matrix에서는 다 거리가 비슷비슷하기 때문에 딱히 의미가 없음.

1.1.3 K-means 평균 속도 개선과 미니배치 K-means

- K-Means 알고리즘 속도 개선
  - 삼각 부등식 사용(두 점 사이의 직선은 항상 가장 짧은 거리)하여 불필요한 거리 계산을 많이 피함
  - 삼각 부등식
    - AC <= AB + BC

    - 샘플과 센트로이드 사이의 거리를 위한 하한선과 상한선을 유지
- 미니배치를 이용한 K-Means : 각 반복마다 미니배치를 사용해 센트로이드를 조금씩 이동
    - 알고리즘의 속도를 3~4배 정도 높임
    - 대량의 데이터셋에도 적용 가능
    - 초기화를 여러 번 수행, 가장 좋은 결과를 직접 골라야 하는 번거로움 존재
    - 배치 k-means 평균 알고리즘이 mini 배치 k-means 알고리즘 보다 더 좋은 이니셔
        - 하지만, 훈련 시간은 k-means가 훨씬 빠르고, k가 증가할 수록 더 뚜렷

미니배치를 사용한 K-Means

In [36]:

```
# fit을 이용하여 데이터 전체를 바로 넣는 방법
from sklearn.cluster import MiniBatchKMeans

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

Out[36]:

```
MiniBatchKMeans(batch_size=100, compute_labels=True, init='k-means++',
                init_size=None, max_iter=100, max_no_improvement=10,
                n_clusters=5, n_init=3, random_state=None,
                reassignment_ratio=0.01, tol=0.0, verbose=0)
```

In [45]:

```
minibatch_kmeans.score(X)
```

Out[45]:

```
-211.69292530888583
```

메모리 한계로 데이터 전체를 바로 넣을 수 없는 경우

- memmap 사용
- for문 이용

In [41]:

```
def load_next_batch(batch_size):
    return X[np.random.choice(len(X), batch_size, replace = False)] # 랜덤으로 idx중에서 batch_siz
```

In [42]:

```
k = 5
n_init = 10 # 초기화 값을 랜덤으로 10번 넣어보겠다
n_iterations = 100 # 100번 반복
batch_size = 100 # batch size
init_size = 500  # K-Means++ 초기화를 위해 충분한 데이터 전달 -> 초기화가 알고리즘 성능에 큰 영향을
evaluate_on_last_n_iters = 10

best_kmeans = None

for init in range(n_init):
    minibatch_kmeans = MiniBatchKMeans(n_clusters=k, init_size=init_size)
    X_init = load_next_batch(init_size)
    minibatch_kmeans.partial_fit(X_init)

    minibatch_kmeans.sum_inertia_ = 0
    for iteration in range(n_iterations):
        X_batch = load_next_batch(batch_size)
        minibatch_kmeans.partial_fit(X_batch)
        if iteration >= n_iterations - evaluate_on_last_n_iters:
            minibatch_kmeans.sum_inertia_ += minibatch_kmeans.inertia_

    if (best_kmeans is None or
        minibatch_kmeans.sum_inertia_ < best_kmeans.sum_inertia_):
        best_kmeans = minibatch_kmeans
```

In [43]:

```
best_kmeans.score(X)
```

Out[43]:

```
-211.69292530888583
```

1.1.4 최적의 클러스터 개수 찾기

- k가 증가함에 따라 각 샘플은 가까운 센트로이드에 더 가깝게 됨으로써 점점 이니셔가 작아짐.
- 이니셔의 작은 변화는 어쩌면 완벽한 클러스터를 아무 이유없이 반으로 나눈걸 수도 있음

1.1.4.1 엘보우 방식

In [49]:

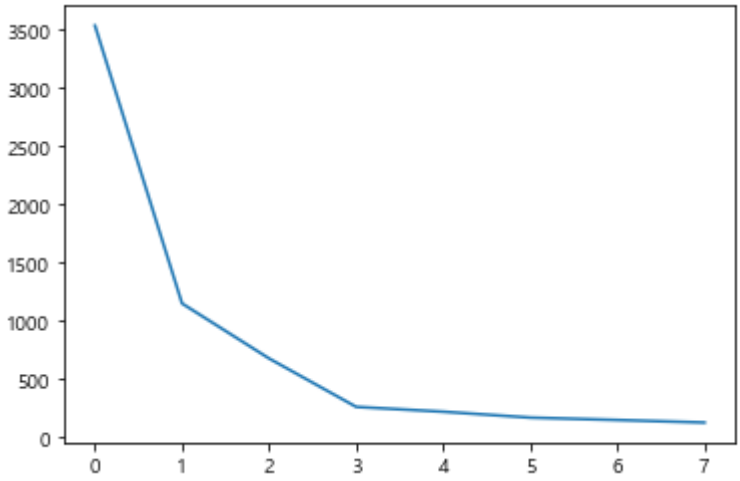
```
from sklearn.cluster import MiniBatchKMeans

scores = []

for i in range(1,9):
    minibatch_kmeans = MiniBatchKMeans(n_clusters = i)
    minibatch_kmeans.fit(X)
    scores.append(minibatch_kmeans.score(X)*-1) # 이니셔가 음수로 나오기 때문에 -1곱해줌
```

In [50]:

```
plt.plot(scores)

# 3지점까지 크게 감소하기 때문에 3~4 정도를 선택
```

Out[50]:

```
[<matplotlib.lines.Line2D at 0x195a817fa08>]
```



1.1.4.2 실루엣 점수

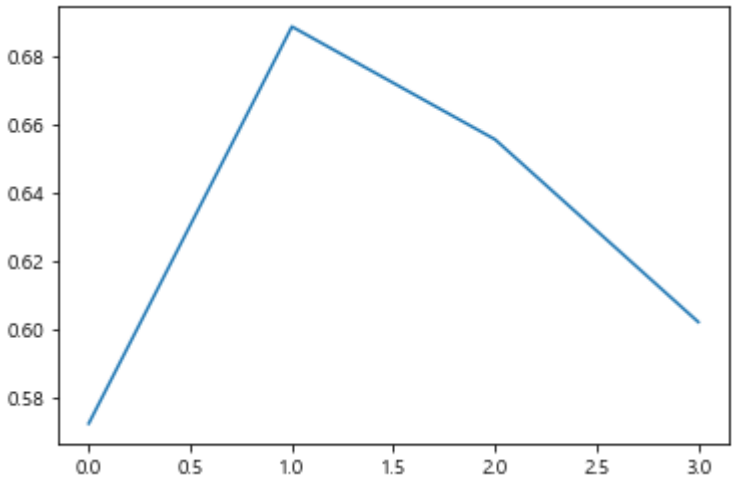
- 실루엣 점수 : 모든 샘플에 대한 실루엣 계수의 평균
- 실루엣 계수
  - 공식
    $(b-a)/max(a,b)$
  - a : 동일한 클러스터에 있는 다른 샘플까지 평균 거리
    - 클러스터 내부의 평균 거리
  - b : 가장 가까운 클러스터까지 평균 거리
    - 가장 가까운 클러스터의 샘플까지 평균 거리
- 실루엣 계수의 범위
  - -1~1
  - +1에 가까우면 자신의 클러스터 안에 잘 속해있고, 다른 클러스터와는 멀리 떨어져 있음
    - a가 작고 b가 커야지 1에 가깝기 때문
  - -1에 가까우면 잘못된 클러스터에 할당됨
    - b보다 a가 더 큰 것이기 때문
  - 0에 가까우면 경계에 할당
    - b와 a가 비슷한 것이기 때문

In [59]:

```
# 데이터셋과 알고리즘으로 할당한 label을 같이 넣어주어야 실루엣 점수를 계산할 수 있음
from sklearn.metrics import silhouette_score
scores = []

for k in range(3,7): # k의 개수(군집 개수)
    kmeans = KMeans(n_clusters = k, n_init=3)
    kmeans.fit(X)
    scores.append(silhouette_score(X, kmeans.labels_))

plt.plot(scores)
```

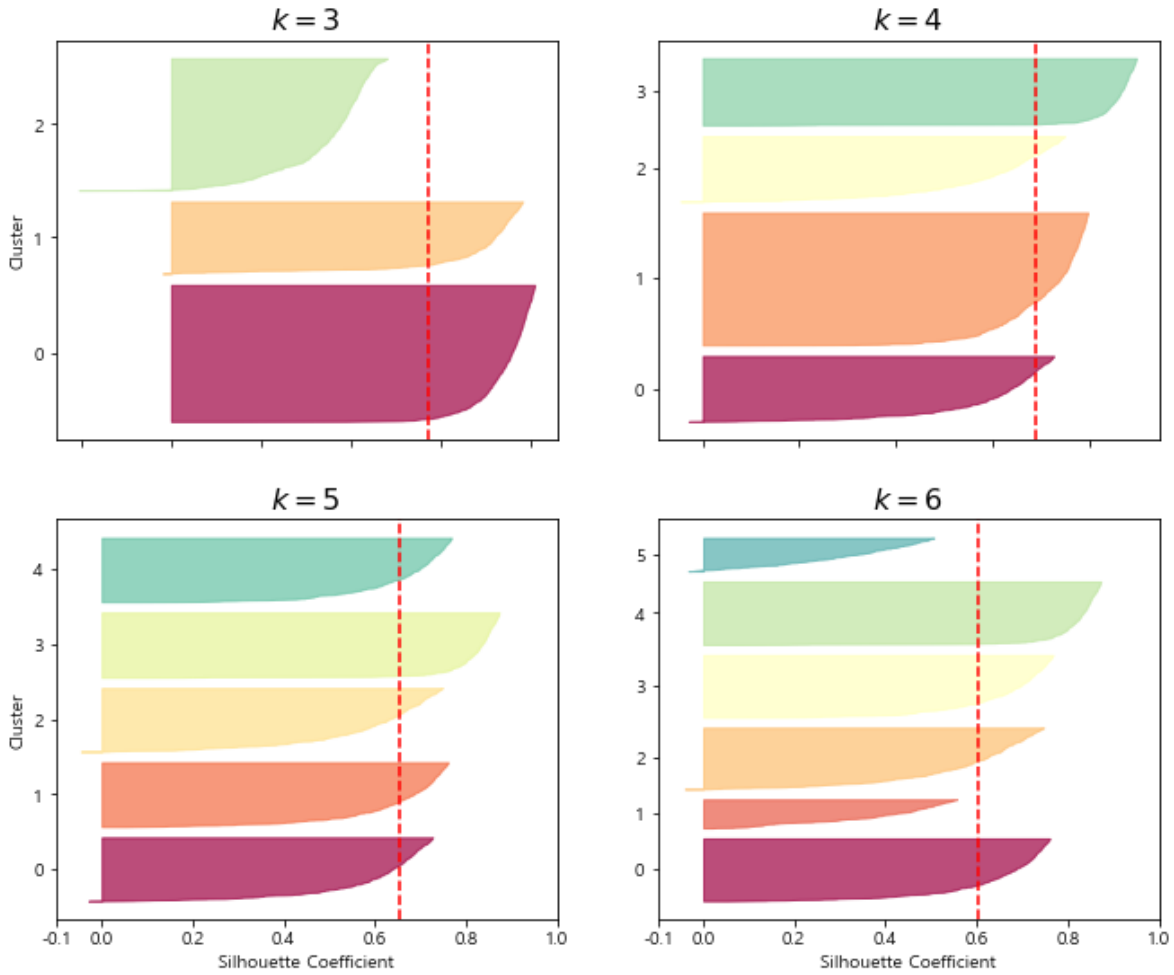
Out[59]:

```
[<matplotlib.lines.Line2D at 0x195aad90388>]
```



In [61]:

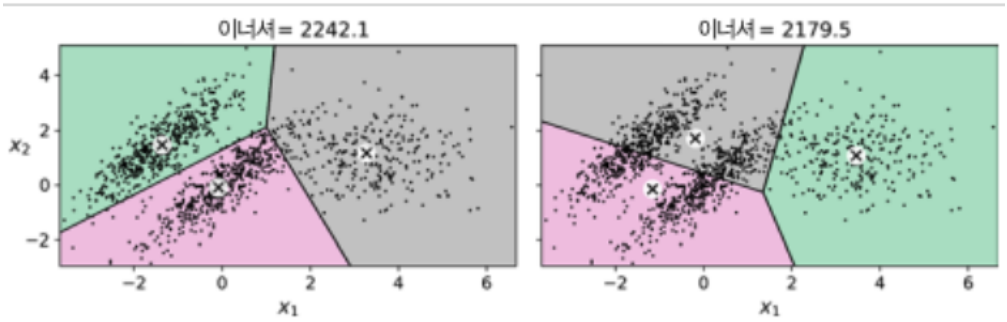
```
# 실루엣 다이어그램 -> 각 클러스터마다, 각 클러스터 안의 샘플 마다 실루엣 계수 그림
kmeans_per_k = [KMeans(n_clusters=k, random_state=42).fit(X)
                for k in range(1, 10)]
inertias = [model.inertia_ for model in kmeans_per_k]
silhouette_scores = [silhouette_score(X, model.labels_)
                     for model in kmeans_per_k[1:]]

from sklearn.metrics import silhouette_samples # 각 군집마다의 silouette_samples를 반환
from matplotlib.ticker import FixedLocator, FixedFormatter

plt.figure(figsize=(11, 9))

for k in (3, 4, 5, 6):
    plt.subplot(2, 2, k - 2)

    y_pred = kmeans_per_k[k - 1].labels_   # 각 k에 해당하는 index
    silhouette_coefficients = silhouette_samples(X, y_pred)

    padding = len(X) // 30
    pos = padding
    ticks = []
    for i in range(k):
        coeffs = silhouette_coefficients[y_pred == i]
        coeffs.sort()

        color = mpl.cm.Spectral(i / k)
        plt.fill_betweenx(np.arange(pos, pos + len(coeffs)), 0, coeffs,
                          facecolor=color, edgecolor=color, alpha=0.7)
        ticks.append(pos + len(coeffs) // 2)
        pos += len(coeffs) + padding

    plt.gca().yaxis.set_major_locator(FixedLocator(ticks))
    plt.gca().yaxis.set_major_formatter(FixedFormatter(range(k)))
    if k in (3, 5):
        plt.ylabel("Cluster")

    if k in (5, 6):
        plt.gca().set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])
        plt.xlabel("Silhouette Coefficient")
    else:
        plt.tick_params(labelbottom=False)

    plt.axvline(x=silhouette_scores[k - 2], color="red", linestyle="--")
    plt.title("$k={}$".format(k), fontsize=16)

plt.show()
```



- 점선은 실루엣 점수
- k=4일 때랑 k=5 일 때 대부분의 샘플이 점선을 넘는 등 비슷하지만, k=4 일 때보다 k=5일 때 샘플이 각 클러스터에 고르게 분포하고 있기 때문에, k=5의 선택지가 더 좋음

1.1.5 k-means 한계

- 장점
  - 속도가 빠르고 확장이 용이
- 단점
  - initial points 가 중요하기 때문에 알고리즘을 여러 번 실행해야 함
  - 클러스터 개수를 알맞게 설정해야 함
  - 클러스터의 크기나 밀집도가 서로 다른 경우 잘 작동하지 않음 -> 비슷한 크기, 밀집의 클러스터를 만들려는 경향이 있기 때문
  - 원형이 아닌 경우 잘 작동하지 않음 -> GMM(가우시안 혼합 모델)이 더 잘 작동
    - k-means를 실행하기 전에 normalize를 통해 스케일을 맞춰줌으로써, 최대한 원형의 형태에 가까워질 수 있도록 해주는 것이 좋음



1.1.6 군집을 사용한 이미지 분할

- 이미지 분할 : 이미지를 세그먼트 여러 개로 분할하는 작업
  - 시맨틱 분할 : 동일한 종류의 물체에 속한 모든 픽셀은 같은 세그먼트에 할당
    - ex) 자율주행자동차의 비전 시스템에서 보행자 이미지를 구성하는 모든 픽셀은 '보행자' 세그먼트에 할당
  - 색상 분할 : 동일한 색상을 가진 픽셀을 같은 세그먼트에 할당
    - ex) 인공위성 사진을 분석하여 전체 산림 면적이 얼마나 되는지 측정

- 이미지 채널
  - RGB : 가시광선에 의해 보이는 색에 대한 빨강, 초록, 파랑의 강도를 담음
  - 흑백 : 채널이 하나
  - 그 외 : 투명도를 위한 알파채널을 가진 이미지 / 여러 전자기파에 대한 채널을 포함하는 위성사진

In [70]:

```
from matplotlib.image import imread # 이미지를 0~1 사이로 로드
# imageio.imread()를 사용하면 0~255
import os

image = imread(os.path.join("flower.png"))
# 4차원으로 불러와졌기 때문에 3차원으로 축소
image = image[:,:,:-1]
image_shape = image.shape
```

Out[70]:

```
(450, 670, 3)
```

In [72]:

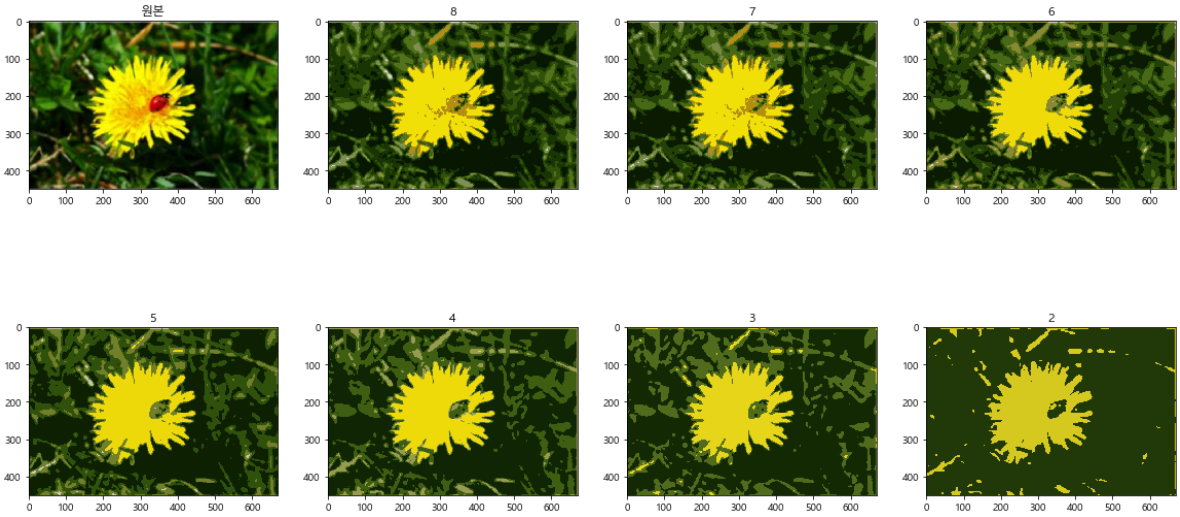
```
X = image.reshape(-1,3) # 각 픽셀의 RGB를 담은 행렬
kmeans = KMeans(n_clusters = 8).fit(X)
```

In [82]:

```
segmented_img = kmeans.cluster_centers_[kmeans.labels_] # 각 픽셀이 할당된 center의 행을 불러옴
segmented_img = segmented_img.reshape(image.shape)
```

In [89]:

```
plt.figure(figsize = (20,10))
plt.subplot(2,4,1)
plt.imshow(image)
plt.title("원본")
for idx, k in enumerate(range(8,1,-1)):
    plt.subplot(2,4,idx+2)
    kmeans = KMeans(n_clusters = k).fit(X)
    segmented_img = kmeans.cluster_centers_[kmeans.labels_].reshape(image.shape) # 군집화된 이미지
    plt.imshow(segmented_img)
    plt.title(k)

    # cluster의 개수가 점점 작아지면서 무당벌레의 빨간색 부분이 없어짐
    #-> kmeans의 경우 비슷한 크기, 밀집도의 클러스터를 만들려는 성향이 있기 때문에 소수의 부분은 없
```



1.1.7 군집을 사용한 전처리

- 차원 축소에 효과적인 방법이기 때문에 지도 학습 알고리즘을 적용하기 전에 전처리 단계로 사용 가능 =

In [94]:

```
from sklearn.datasets import load_digits

X_digits, y_digits = load_digits(return_X_y = True) # X,y를 각각 불러옴
# return_X_y = True 이면 dictionary 의 형태로 각각의 값을 저장한 상태
```

In [99]:

```
from sklearn.model_selection import train_test_split

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

In [106]:

```
# raw 데이터를 통한 학습 및 예측
from sklearn.linear_model import LogisticRegression

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

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

```
0.9577777777777777

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:940: Co
nvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-lear
n.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (h
ttps://scikit-learn.org/stable/modules/linear_model.html#logistic-regression)
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

In [108]:

```
# 군집화를 통한 전처리 데이터로의 학습 및 예측
# 데이터를 클러스터까지의 거리로 바꿈
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ("kmeans", KMeans(n_clusters=100)),
    ("log_reg", LogisticRegression())
])
pipeline.fit(X_train, y_train)

pipeline.score(X_test, y_test)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:940: Co
nvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-lear
n.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (h
ttps://scikit-learn.org/stable/modules/linear_model.html#logistic-regression)
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

Out[108]:

```
0.96
```

In [114]:

```
# 전처리를 위한 군집화의 적절한 k 개수 grid search
from sklearn.model_selection import GridSearchCV

# pipeline 안에서의 모델에 접근하려면 "모델닉네임__파라미터"
param_grid = dict(kmeans__n_clusters=range(0,100)) # n_clusters:[0~100]의 dictionary 생성
grid_clf = GridSearchCV(pipeline, param_grid, cv=3, verbose=2)
grid_clf.fit(X_train,y_train)
```

[CV] kmeans__n_clusters=12 ....

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model_logistic.py:940:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-learn.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression)
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)

[CV] ..... kmeans__n_clusters=12, total=   0.4s
[CV] kmeans__n_clusters=13 ....

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model_logistic.py:940:

In [115]:

```
print(grid_clf.best_params_)
print(grid_clf.score(X_test,y_test))
```

```
{'kmeans__n_clusters': 54}
0.96
```

1.1.8 군집을 사용한 준지도학습

- 레이블이 있는 데이터는 적고, 레이블이 없는 데이터가 많을 때 사용.

In [117]:

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

```
C:₩ProgramData₩Anaconda3₩lib₩site-packages₩sklearn₩linear_model₩_logistic.py:940: Co
nvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-lear
n.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (h
ttps://scikit-learn.org/stable/modules/linear_model.html#logistic-regression)
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

Out[117]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

In [118]:

```
log_reg.score(X_test, y_test) # 내가 예측하고 싶은 것이 있는데,
# train data에 라벨링되지 않은 데이터가 많아서 훈련시에 라벨된 데이터만 사용했을 때의 test 정확도
```

Out[118]:

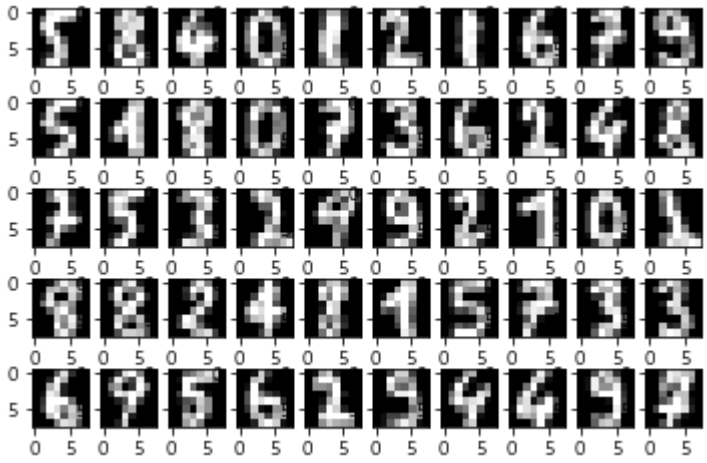
```
0.8022222222222222
```

In [123]:

```
k = 50
kmeans = KMeans(n_clusters = k)
X_digits_dist = kmeans.fit_transform(X_train) # 센트로이드와의 거리로 데이터를 변환
representative_digit_idx = np.argmin(X_digits_dist, axis=0) # 각 센트로이드와 가장 가까운 데이터
X_representative_digits = X_train[representative_digit_idx] # 각 군집의 대표 데이터
```

In [129]:

```
for idx, i in enumerate(X_representative_digits):
    plt.subplot(5,10,idx+1)
    plt.imshow(i.reshape(8,8), cmap="gray")
```



In [133]:

```
y_representative_digit = y_train[representative_digit_idx]
```

In [134]:

```
log_reg = LogisticRegression()
log_reg.fit(X_representative_digits, y_representative_digit)
log_reg.score(X_test,y_test)
```

```
C:WProgramDataWAnaconda3WlibWsite-packagesWsklearnWlinear_modelW_logistic.py:940: Co
nvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-lear
n.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (h
ttps://scikit-learn.org/stable/modules/linear_model.html#logistic-regression)
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

Out[134]:

```
0.8711111111111111
```

무작위 샘플 대신 어느정도 대표성이 있는 데이터(kmeans는 크기와 밀도를 비슷하게 해서 데이터를 구성하기 때문에 데이터가 몰려있는 곳에 센트로에드를 만들 것임)로 학습하는 것이 좋음 -> 어느 정도 대표성이 있는 데이터의 라벨을 찾는 것에 노력을 기울이는 것이 좋음

레이블 전파

- 동일한 클러스터에 있는 데이터에 레이블을 전파하는 것

클러스터에 속한 모든 샘플에 레이블을 전파한 경우

In [136]:

```
y_train_propagated = np.empty(len(X_train), dtype=np.int32)
for i in range(k):
    # 해당 클러스터로 분류된 데이터에 레이블된 데이터의 라벨을 전파
    y_train_propagated[kmeans.labels_==i] = y_representative_digit[i]
```

In [137]:

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

```
C:₩ProgramData₩Anaconda3₩lib₩site-packages₩sklearn₩linear_model₩_logistic.py:940: Co
nvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-lear
n.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (h
ttps://scikit-learn.org/stable/modules/linear_model.html#logistic-regression)
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

Out[137]:

```
0.8911111111111111
```

클러스터의 센트로이드에 가까운 샘플의 일부에만 레이블을 전파한 경우

- 클러스터의 센트로이드에서 멀수록 해당 데이터의 라벨이 센트로이드의 라벨이 아닐 확률이 매우 높기 때문

In [159]:

```
percentile_closet = 20 # 거리가 가까운 20%의 샘플에만 라벨을 전파
# 자신의 군집의 센트로이드와의 거리만을 추출
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] # 해당 군집의 데이터의 센트로이드와의 거리만을 추출
    # 젤 작은 것부터 줄 세웠을 때 20%의 값(하위 20%의 값)
    cutoff_distance = np.percentile(cluster_dist, percentile_closet)
    above_cutoff = (X_cluster_dist) > cutoff_distance
    X_cluster_dist[in_cluster & above_cutoff] = -1 # 상위 80%의 데이터 제거

partially_propagated = (X_cluster_dist != -1)
X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]
# 일단 모든 데이터에 전파한 후에 각 군집의 하위 20%의 데이터(각 센트로이드와 가까운 20%의 데이터만을
```

In [160]:

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

```
C:₩ProgramData₩Anaconda3₩lib₩site-packages₩sklearn₩linear_model₩_logistic.py:940: Co
nvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html (https://scikit-lear
n.org/stable/modules/preprocessing.html)
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression (h
ttps://scikit-learn.org/stable/modules/linear_model.html#logistic-regression)
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

Out[160]:

```
0.9044444444444445
```

In [161]:

```
np.mean(y_train_partially_propagated == y_train[partially_propagated])
```

Out[161]:

```
1.0
```

능동학습

- 전문가가 학습 알고리즘과 상호작용하여 알고리즘이 요청할 때 특정 레이블의 샘플을 제공하는 방법

<불확실성 샘플링>

1. 지금까지 수집한 레이블된 데이터에서 훈련, 레이블되지 않은 데이터에 대해 예측
2. 모델이 가장 불확실하게 예측한 샘플(추정 확률이 낮은 샘플)을 전문가에게 보내 레이블을 붙임
3. 레이블을 부여하는 노력만큼의 성능이 향상되지 않을 때까지 이를 반복

<그 외>

- 모델을 가장 크게 바꾸는 데이터에 대해 요청
- 모델의 검증 점수를 가장 크게 떨어트리는 샘플
- 여러 개의 모델이 동일한 예측을 내지 않는 샘플에 대해 레이블을 요청