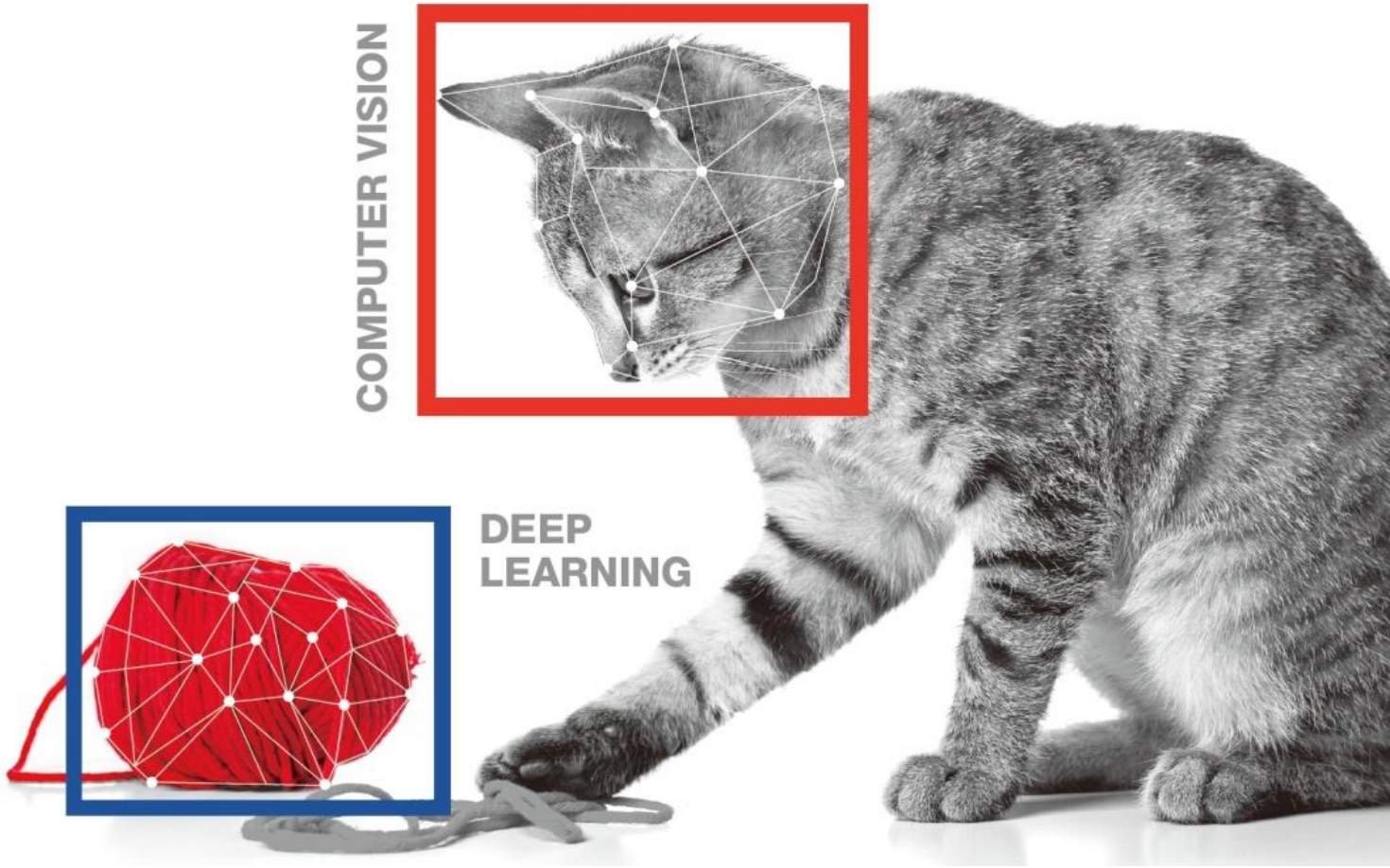
# 이미지딥러닝




04주차

## 군집 알고리즘

# 목차

# 비지도 학습

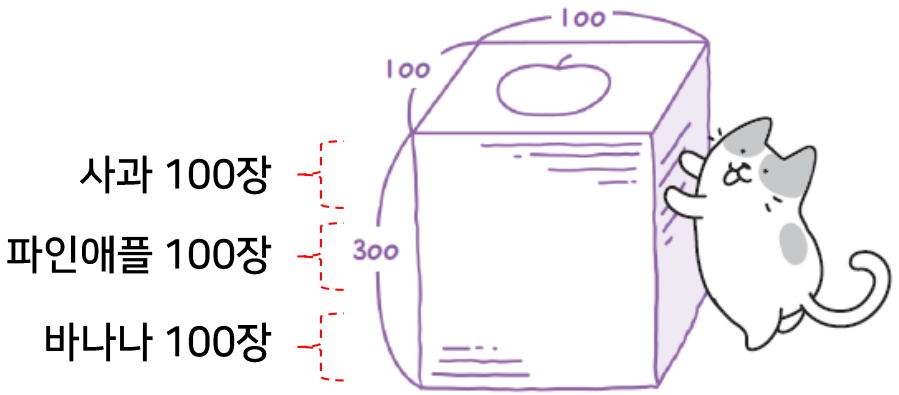
- 타깃이 없을 때 사용하는 머신러닝 알고리즘이 비지도 학습(Unsupervised Learning)
  - 레이블(정답) 없이 데이터를 스스로 이해하고 구조를 찾아내는 학습 방법
  - 사람의 개입 없이 데이터의 패턴, 군집, 분포 등을 자동으로 파악함
  - 주로 클러스터링(군집화), 차원 축소, 이상치 탐지 등에 활용

# 비지도 학습 실습: 데이터 준비

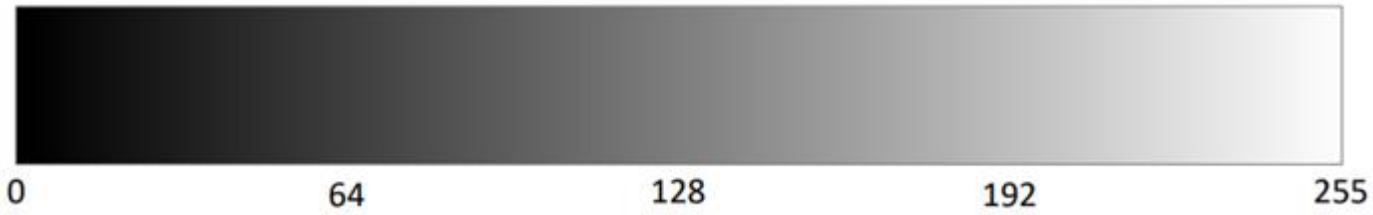
- 과일 사진 데이터 준비하기
  - 준비한 과일 데이터는 사과, 바나나, 파인애플을 담고 있는 흑백 사진
  - 이 데이터는 넘파이 배열의 기본 저장 포맷인 npy 파일로 저장되어 있음
  - 넘파이에서 load( ) 메서드에 파일 이름을 전달하여 npy 파일을 로드

| In | `fruits = np.load('C:/cv_workspace/data/fruits_300.npy')`<br>`print(fruits.shape)` |
|---|---|
| Out | `(300, 100, 100)` |

(샘플갯수, 이미지 높이, 이미지 너비)




- 컴퓨터는 이미지를 일종의 숫자로 변환하여 인식
  - 이미지를 일종의 점(dot)으로 생각하면 m×n만큼의 공간이 존재
  - 그 공간 안에서 색깔이 진할수록 높은 값 색깔이 옅을수록 낮은 값을 가짐 (0~255 값)



# 비지도 학습 실습

- 첫 번째 이미지의 첫 번째 행에 들어 있는 픽셀 100개 값을 출력

  흑백 사진을 담고 있으므로 0~255의 정숫값을 가짐

In
```
print(fruits[0, 0, :])
```

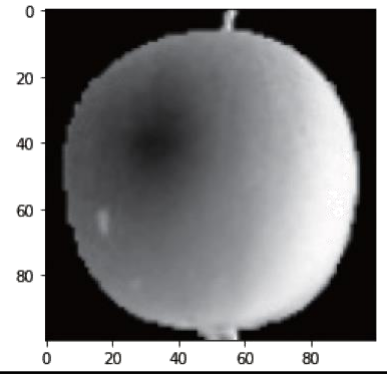
Out
```
[  1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   2   1
   2   2   2   2   2   2   1   1   1   1   1   1   1   1   2   3   2   1
   2   1   1   1   1   2   1   3   2   1   3   1   4   1   2   5   5   5
  19 148 192 117  28   1   1   2   1   4   1   1   3   1   1   1   1   1
   2   2   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1]
```

- 맷플롯립의 imshow( ) 함수를 사용하여 넘파이 배열로 저장된 이미지를 구현

  흑백 이미지이므로 cmap 매개변수를 'gray'로 지정

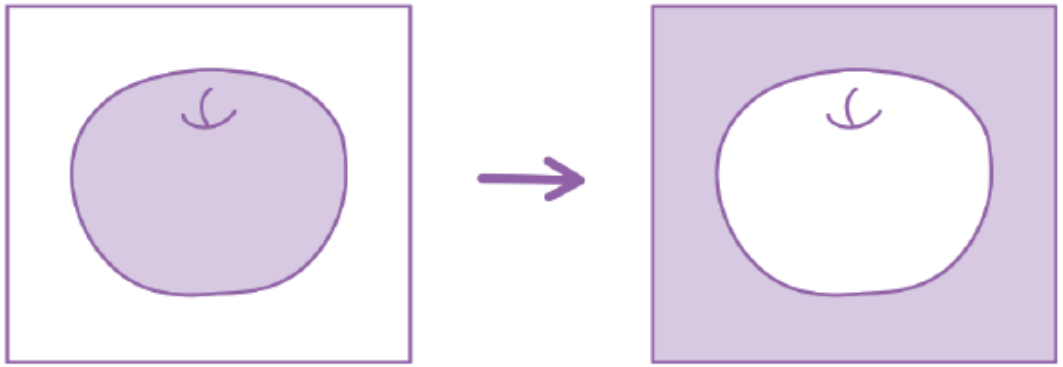
In
```
plt.imshow(fruits[0], cmap='gray')
plt.show()
```

Out



# 비지도 학습 실습: 데이터 준비

- 흑백 이미지는 사진으로 찍은 이미지를 넘파이 배열로 변환할 때 반전시킨 것
- 사진의 흰 바탕(높은 값)은 검은색(낮은 값)으로 만들고 실제 사과가 있어 짙은 부분(낮은 값)은 밝은색(높은 값)으로 변환
- 흰색 바탕은 우리에게 중요하지 않지만 컴퓨터는 255에 가까운 바탕에 집중. 따라서 바탕을 검게 만들고 사진에 짙게 나온 사과를 밝은색으로 만듦



컴퓨터는 왜 255에 가까운 바탕에 집중하나?

- 알고리즘이 어떤 출력을 만들기 위해 곱셈, 덧셈을 수행
- 픽셀값이 0이면 출력도 0이 되어 의미가 없음
- 픽셀값이 높으면 출력값도 커지기 때문에 의미를 부여하기 좋음

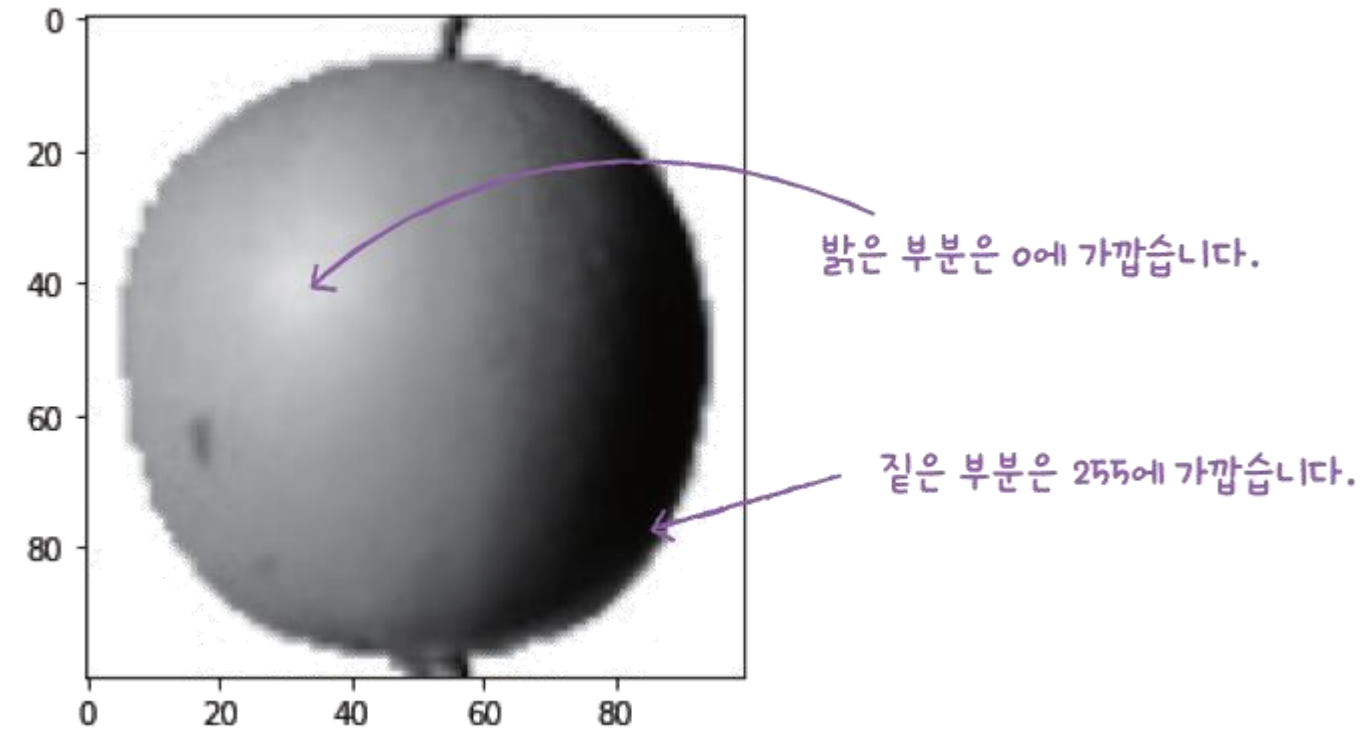
# 비지도 학습 실습: 데이터 준비

- cmap 매개변수를 'gray_r'로 지정하면 다시 반전하여 우리 눈에 보기 좋게 출력

In

```
plt.imshow(fruits[0], cmap='gray_r')
plt.show()
```

Out



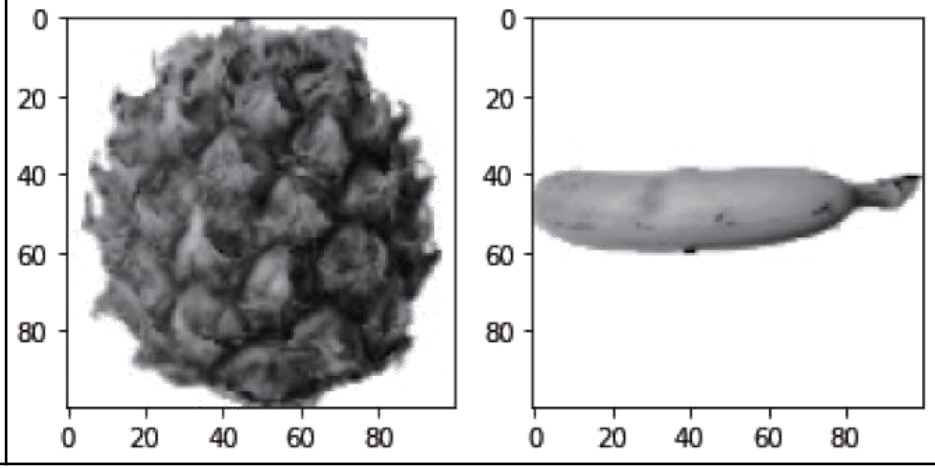
# 비지도 학습 실습: 데이터 준비

- 바나나와 파인애플 이미지도 출력

In

```
fig, axs = plt.subplots(1, 2)
axs[0].imshow(fruits[100], cmap='gray_r')
axs[1].imshow(fruits[200], cmap='gray_r')
plt.show()
```

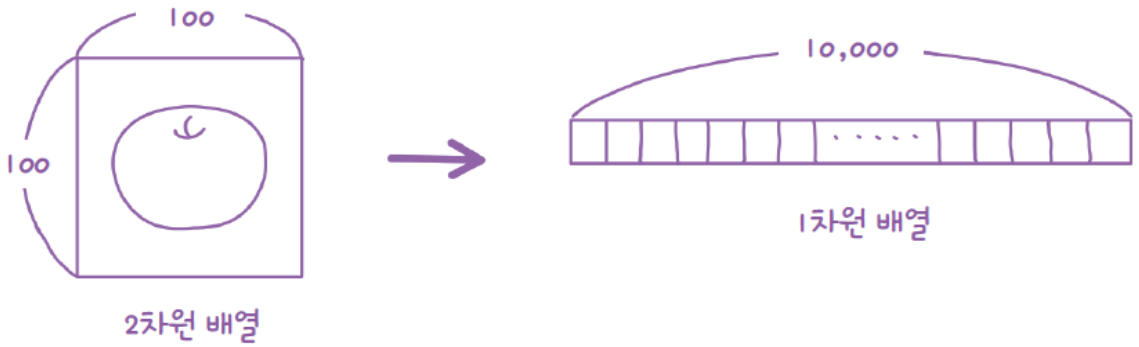
Out




- 맷플롯립의 subplots( ) 함수를 사용하면 여러 개의 그래프를 배열처럼 쌓을 수 있도록 도와줌
- subplots( ) 함수의 두 매개변수는 그래프를 쌓을 행과 열을 지정
- 위에서는 subplots(1, 2)처럼 하나의 행과 2개의 열을 지정

# 비지도 학습 실습: 데이터 픽셀 분석

- 사용하기 쉽게 fruits 데이터를 사과, 파인애플, 바나나로 각각 나누기
- 넘파이 배열을 나눌 때 100 × 100 이미지를 펼쳐서 길이가 10,000인 1차원 배열로 만들기
  - 이렇게 펼치면 이미지로 출력하긴 어렵지만 배열을 계산할 때 편리




- fruits 배열에서 순서대로 100개씩 선택하기 위해 슬라이싱 연산자를 사용. 그다음 reshape( ) 메서드를 사용해 두 번째 차원(100)과 세 번째 차원(100)을 10,000으로 합침

| | |
|---|---|
| In | apple = fruits[0:100].reshape(-1, 100*100)<br>pineapple = fruits[100:200].reshape(-1, 100*100)<br>banana = fruits[200:300].reshape(-1, 100*100) |

(1행 = 1개 이미지)

- 이제 apple, pineapple, banana 배열의 크기는 (100, 10000)이 됨

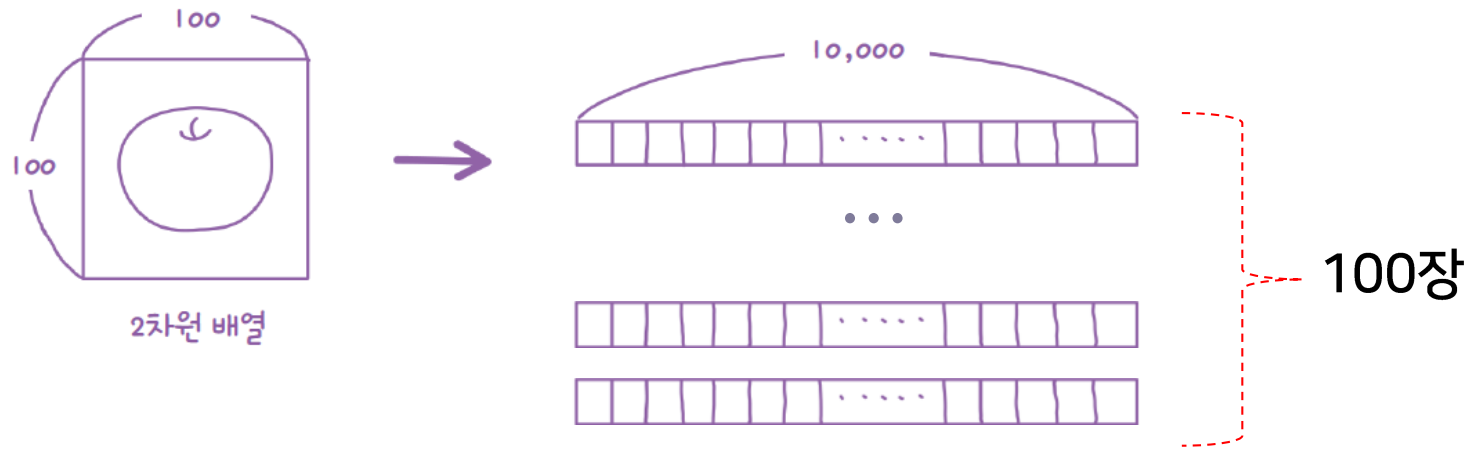
# 비지도 학습 실습: 데이터 픽셀 분석

- apple 배열의 mean( ) 메서드로 각 샘플의 픽셀 평균값을 계산

| In | print(apple.mean(axis=1)) |
|---|---|
| Out | [ 88.3346 97.9249 87.3709 98.3703 92.8705 82.6439 94.4244 95.5999<br>90.681 81.6226 87.0578 95.0745 93.8416 87.017 97.5078 87.2019<br>88.9827 100.9158 92.7823 100.9184 104.9854 88.674 99.5643 97.2495<br>94.1179 92.1935 95.1671 93.3322 102.8967 94.6695 90.5285 89.0744<br>97.7641 97.2938 100.7564 90.5236 100.2542 85.8452 96.4615 97.1492<br>90.711 102.3193 87.1629 89.8751 86.7327 86.3991 95.2865 89.1709<br>96.8163 91.6604 96.1065 99.6829 94.9718 87.4812 89.2596 89.5268<br>93.799 97.3983 87.151 97.825 103.22 94.4239 83.6657 83.5159<br>102.8453 87.0379 91.2742 100.4848 93.8388 90.8568 97.4616 97.5022<br>82.446 87.1789 96.9206 90.3135 90.565 97.6538 98.0919 93.6252<br>87.3867 84.7073 89.1135 86.7646 88.7301 86.643 96.7323 97.2604<br>81.9424 87.1687 97.2066 83.4712 95.9781 91.8096 98.4086 100.7823<br>101.556 100.7027 91.6098 88.8976] |

```
In [21]: apple.shape
Out[21]: (100, 10000)
```

axis 0 axis 1



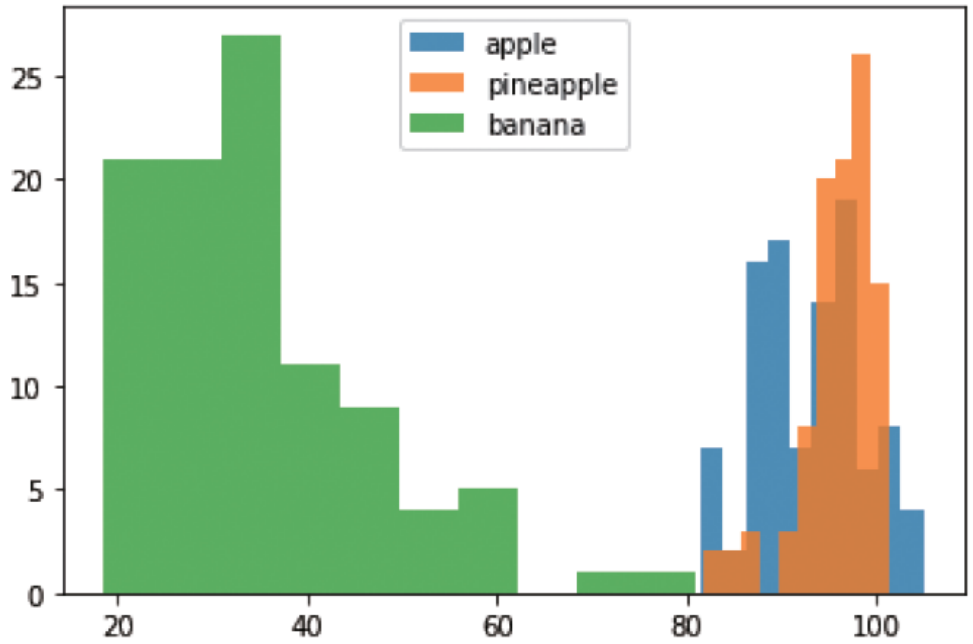
# 비지도 학습 실습: 데이터 픽셀 분석

- 맷플롯립의 hist( ) 함수로 평균값 분포 히스토그램을 구현

  - legend( ) 함수를 사용해 어떤 과일의 히스토그램인지 범례 추가

```
In  plt.hist(np.mean(apple, axis=1), alpha=0.8)
    plt.hist(np.mean(pineapple, axis=1), alpha=0.8)
    plt.hist(np.mean(banana, axis=1), alpha=0.8)
    plt.legend(['apple', 'pineapple', 'banana'])
    plt.show()
```

→




  - alpha는 시각화할 때 색상의 투명도를 조절하는 파라미터

- 바나나는 픽셀 평균값만으로 사과나 파인애플과 확실히 구분됨

  - 바나나는 사진에서 차지하는 영역이 작기 때문에  평균값이 작음

- 반면 사과와 파인애플은 많이 겹쳐있어서 픽셀값만으로는 구분하기 쉽지 않음

  - 사과나 파인애플은 대체로 형태가 동그랗고 사진에서 차지하는 크기도 비슷하기 때문

# 비지도 학습 실습: 데이터 픽셀 분석

- 샘플의 평균값이 아니라 픽셀별 평균값 비교

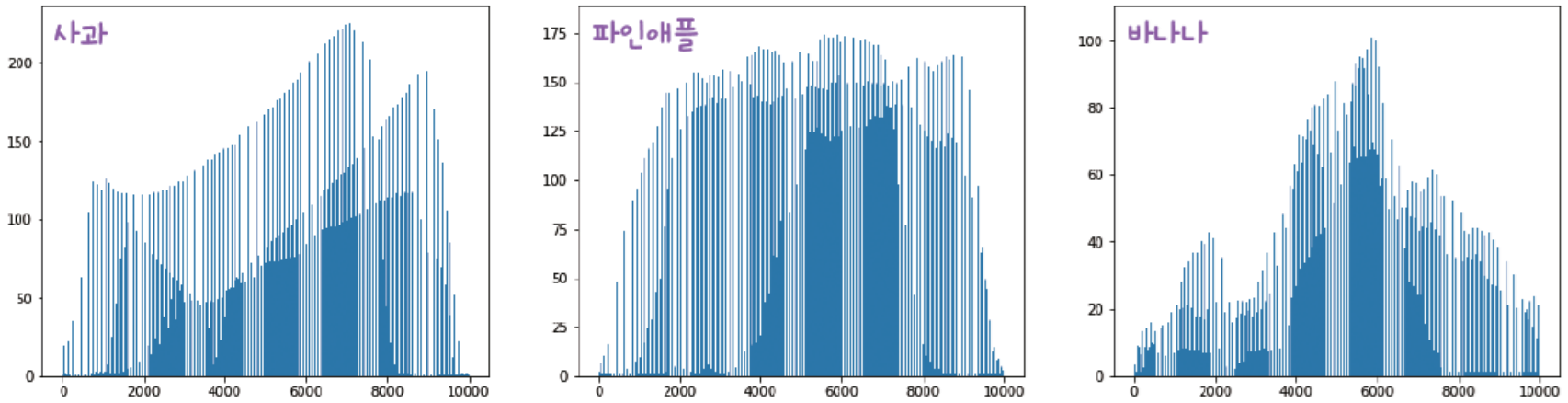
  - 픽셀의 평균을 계산하기 위해 axis=0으로 지정

- 맷플롯립의 bar( ) 함수를 사용해 픽셀 10,000개에 대한 평균값을 막대그래프로 구현

In

```
fig, axs = plt.subplots(1, 3, figsize=(20,5))
axs[0].bar(range(10000), np.mean(apple, axis=0))
axs[1].bar(range(10000), np.mean(pineapple, axis=0))
axs[2].bar(range(10000), np.mean(banana, axis=0))
plt.show()
```

Out



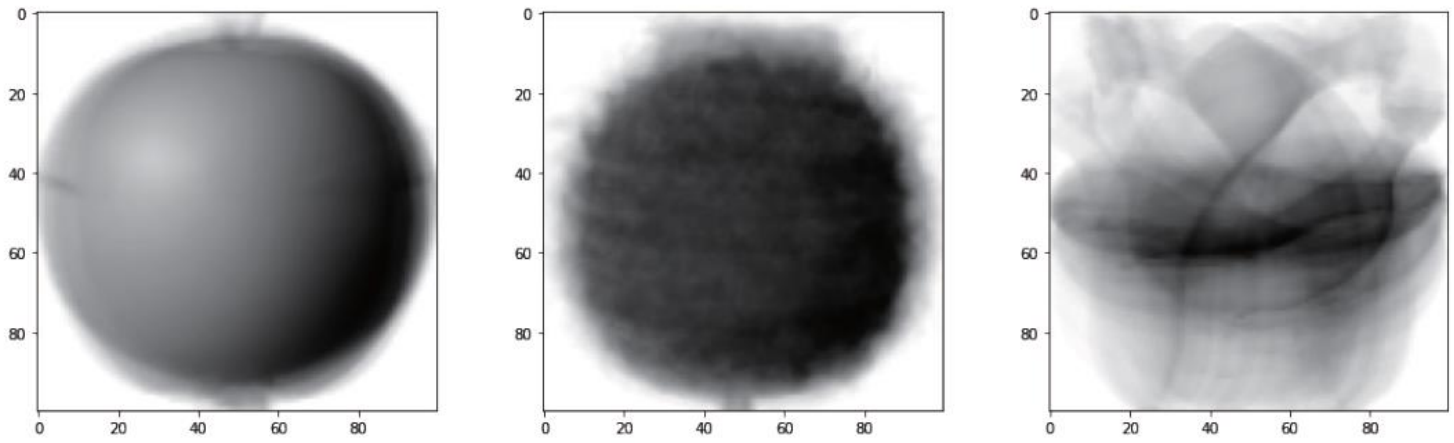
# 비지도 학습 실습: 데이터 픽셀 분석

- 픽셀 평균값을 100 × 100 크기로 바꿔서 이미지처럼 출력하여 앞의 그래프와 비교

In

```
apple_mean = np.mean(apple, axis=0).reshape(100, 100)
pineapple_mean = np.mean(pineapple, axis=0).reshape(100, 100)
banana_mean = np.mean(banana, axis=0).reshape(100, 100)
fig, axs = plt.subplots(1, 3, figsize=(20,5))
axs[0].imshow(apple_mean, cmap='gray_r')
axs[1].imshow(pineapple_mean, cmap='gray_r')
axs[2].imshow(banana_mean, cmap='gray_r')
plt.show()
```

Out



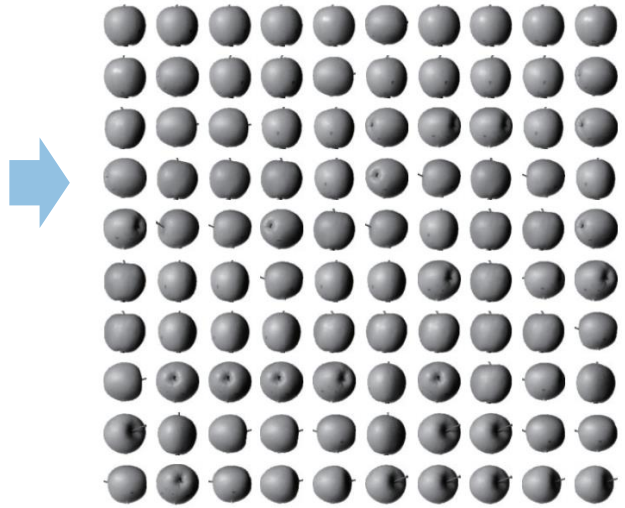
# 비지도 학습 실습: 데이터 픽셀 분석

- 평균값과 가까운 사진 고르기
  - 사과 사진의 평균값인 apple_mean과 가장 가까운 사진 고르기
    - 3장에서 학습한 절댓값 오차를 사용
  - fruits 배열에 있는 모든 샘플에서 apple_mean을 뺀 절댓값의 평균을 계산

| In | abs_diff = np.abs(fruits - apple_mean)<br>abs_mean = np.mean(abs_diff, axis=(1,2))<br>print(abs_mean.shape) |
|---|---|
| Out | (300,) |

  - apple_mean과 오차가 가장 작은 샘플 100개 고르기

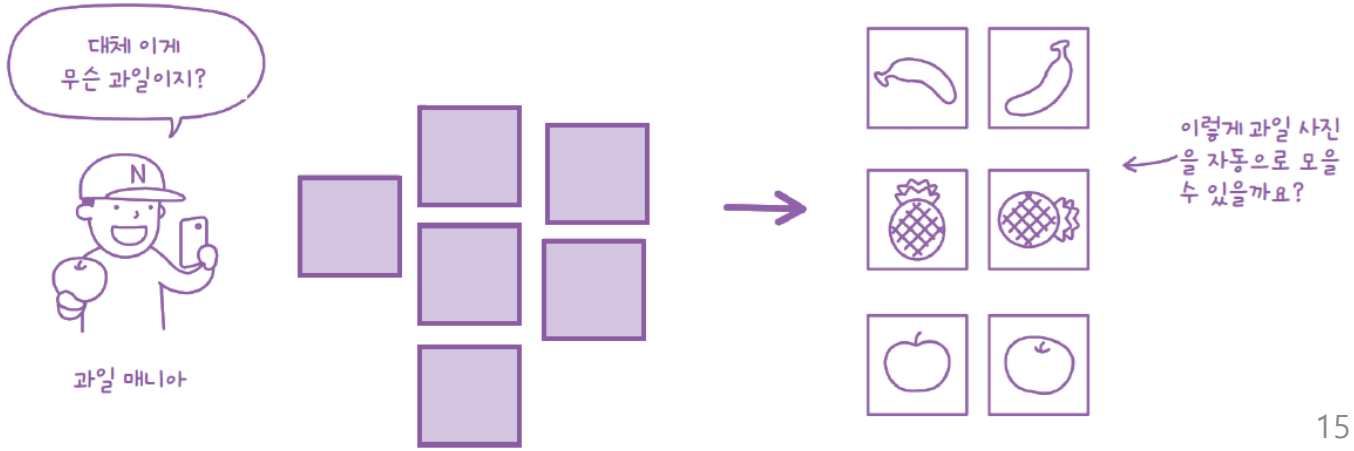
| In | apple_index = np.argsort(abs_mean)[:100]<br>fig, axs = plt.subplots(10, 10, figsize=(10,10))<br>for i in range(10):<br>  for j in range(10):<br>  axs[i, j].imshow(fruits[apple_index[i*10 + j]], cmap='gray_r')<br>  axs[i, j].axis('off')<br>plt.show() |
|---|---|



- 군집(clustering): 비슷한 샘플끼리 그룹으로 모으는 작업
- 클러스터(cluster): 군집 알고리즘에서 만든 그룹

# k-평균

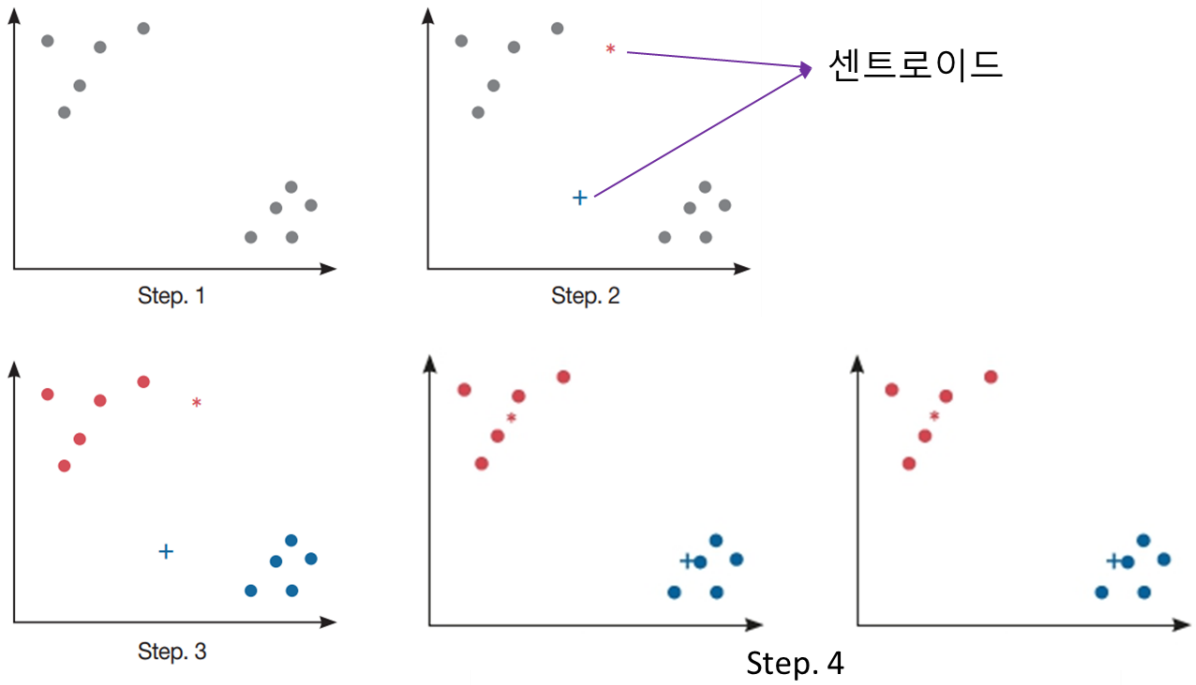
- k-평균(k-means) 군집 알고리즘이 평균값을 자동으로 찾아줌
- k-평균은 산업 현장은 물론 학계에서도 널리 사용
- 군집은 비슷한 객체로 이루어진 그룹을 찾는 기법
- 한 그룹 안의 객체들은 다른 그룹에 있는 객체보다 더 관련되어 있음
- 군집에 대한 상용 애플리케이션의 예로는 문서나 음악, 영화를 여러 주제의 그룹으로 모으는 경우를 들 수 있음
- 또는 추천 엔진에서 하듯이 구매 이력의 공통 부분을 기반으로 관심사가 비슷한 고객을 찾는 것
- k-평균 알고리즘은 구현하기 매우 쉽고 다른 군집 알고리즘에 비해 계산 효율성이 높기 때문에 인기가 많음
- k-평균 알고리즘이 원형 클러스터를 구분하는 데 뛰어나지만 이 알고리즘 단점은 사전에 클러스터 개수 k를 지정해야 하는 것
- 적절하지 않은 k를 고르면 군집 성능이 좋지 않음



# k-평균 알고리즘의 작동 방식

- 실제 군집 애플리케이션에서는 샘플에 대한 진짜 카테고리 정보(추론이 아니라 실증적인 정보)가 전혀 없음
  - 클래스 레이블이 있다면 이 작업은 지도 학습에 해당
- 클러스터링의 목표는 특성의 유사도에 기초하여 샘플을 그룹으로 모으는 것

1) 무작위로 k개의 클러스터 중심을 정함
2) 각 샘플에서 가장 가까운 클러스터 중심을 찾아 해당 클러스터의 샘플로 지정
3) 클러스터에 속한 샘플의 평균값으로 클러스터 중심을 변경
4) 클러스터 중심에 변화가 없을 때까지 2번으로 돌아가 반복



# KMeans 클래스

- 앞서 사용한 fruits 파일 사용
- np.load( ) 함수를 사용해 npy 파일을 읽어 넘파이 배열을 준비

  - k-평균 모델을 훈련하기 위해 (샘플 개수, 너비, 높이) 크기의 3차원 배열을 (샘플 개수, 너비×높이) 크기를 가진 2차원 배열로 변경

| In | import numpy as np<br>fruits = np.load('fruits_300.npy')<br>fruits_2d = fruits.reshape(-1, 100*100) |
|---|---|

- k-평균 알고리즘으로 3개 클러스터로 군집

| In | from sklearn.cluster import KMeans<br>km = KMeans(n_clusters=3, random_state=42)<br>km.fit(fruits_2d) |
|---|---|
| Out | KMeans(n_clusters=3, random_state=42) |

# KMeans 클래스

- k-평균의 레이블 출력

| In | print(km.labels_) |
|---|---|
| Out | [2 2 2 2 2 0 2 2 2 2 2 2 2 2 2 2 2 0 2 2 2 2 2 2 2 0 2 2 2 2 2 2 2 2 2<br>2 2 2 2 2 0 2 0 2 2 2 2 2 2 2 0 2 2 2 2 2 2 2 2 2 0 0 2 2 2 2 2 2 2 2 0 2<br>2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 0 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0<br>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0<br>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0<br>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1<br>1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1<br>1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1<br>1 1 1 1] |

- 레이블 0, 1, 2로 모은 샘플의 개수 확인

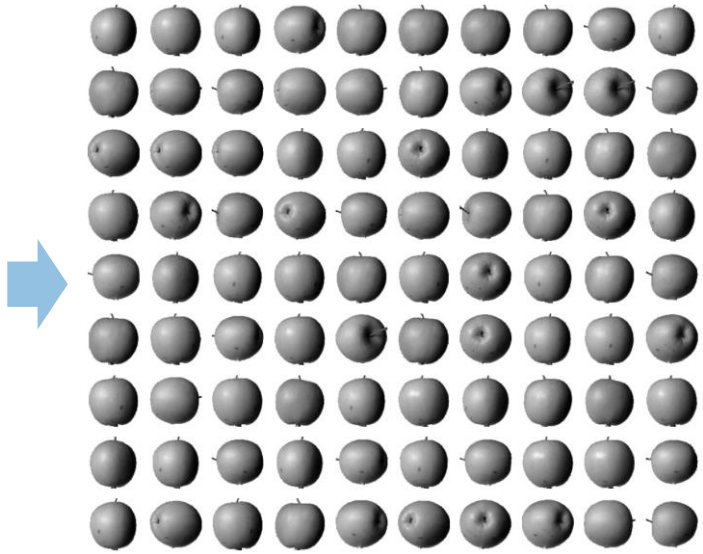
| In | print(np.unique(km.labels_, return_counts=True)) |
|---|---|
| Out | (array([0, 1, 2], dtype=int32), array([112,  98,  90])) |

# KMeans 클래스

- 각 클러스터가 어떤 이미지를 나타냈는지 그림으로 출력하기 위해 간단한 유틸리티 함수 draw_fruits( ) 사용

In

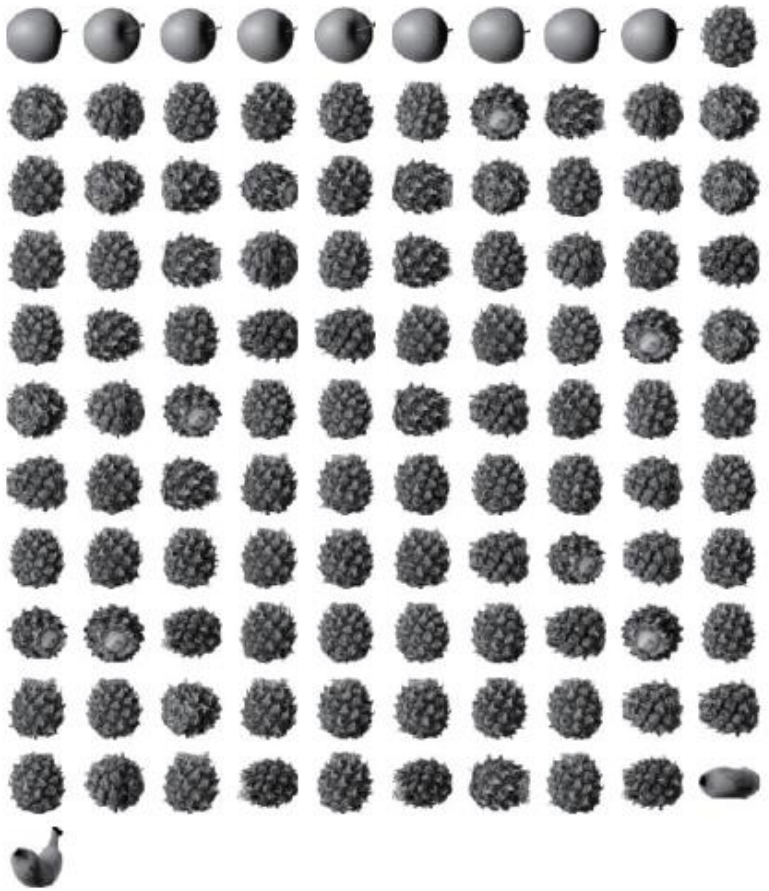
```
import matplotlib.pyplot as plt
def draw_fruits(arr, ratio=1):
      n = len(arr) # n은 샘플 개수
      # 한 줄에 10개씩 이미지, 샘플 개수를 10으로 나누어 전체 행 개수를 계산
      rows = int(np.ceil(n/10))
      # 행이 1개이면 열의 개수는 샘플 개수, 그렇지 않으면 10개
      cols = n if rows < 2 else 10
      fig, axs = plt.subplots(rows, cols, figsize=(cols*ratio, rows*ratio), squeeze=False)
      for i in range(rows):
            for j in range(cols):
                  if i*10 + j < n: # n 개까지만 그리기
                        axs[i, j].imshow(arr[i*10 + j], cmap='gray_r')
                  axs[i, j].axis('off')
      plt.show()
draw_fruits(fruits[km.labels_== 2])
```

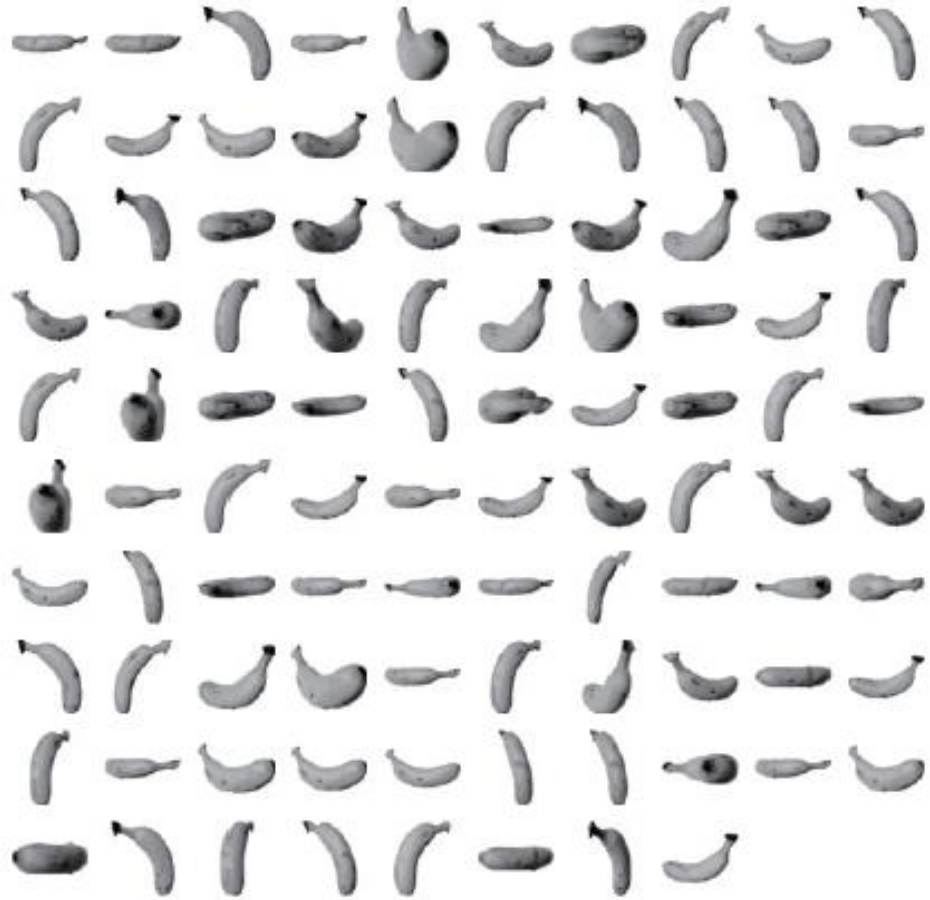


# KMeans 클래스

- 앞의 함수를 사용해 레이블이 0인 과일 사진을 모두 그리기

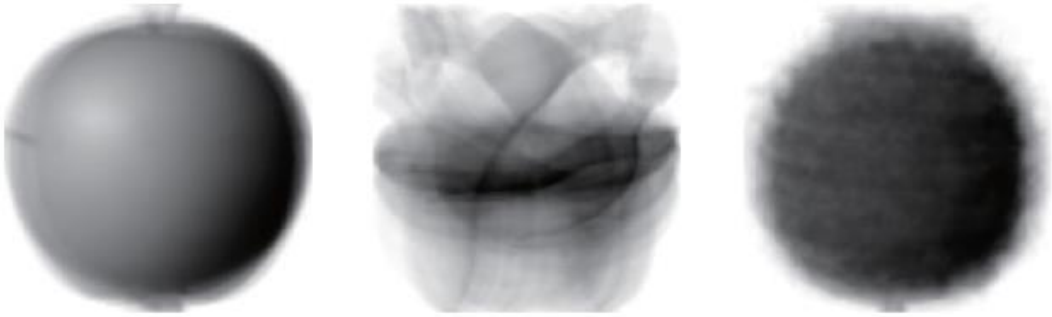
  - 다른 두 클러스터의 이미지 출력

| In | draw_fruits(fruits[km.labels_==0]) |
|---|---|
| Out |  |

| In | draw_fruits(fruits[km.labels_==1]) |
|---|---|
| Out |  |

# 클러스터 중심

- KMeans 클래스가 최종적으로 찾은 클러스터 중심은 cluster_centers_ 속성에 저장됨.
- 이 배열은 fruits_2d 샘플의 클러스터 중심이기 때문에 이미지로 출력하려면 100 × 100 크기의 2차원 배열로 바꿔야 함

| In | `draw_fruits(km.cluster_centers_.reshape(-1, 100, 100), ratio=3)` |
|---|---|
| Out |  |

- 훈련 데이터 샘플에서 클러스터 중심까지 거리로 변환해 주는 transform( ) 메서드
- 인덱스가 100인 샘플에 transform( ) 메서드를 적용

  - 슬라이싱 연산자를 사용해서 (1, 10000) 크기의 배열을 전달

| In | `print(km.transform(fruits_2d[100:101]))` |
|---|---|
| Out | `[[3400.24197319 8837.37750892 5279.33763699]]` |

# 클러스터 중심

- 가장 가까운 클러스터 중심을 예측 클래스로 출력하는 predict( ) 메서드

| In | print(km.transform(fruits_2d[100:101])) |
|---|---|
| Out | [0] |

- 클러스터 중심을 그려보았을 때 레이블 2는 파인애플. 이미지로 확인

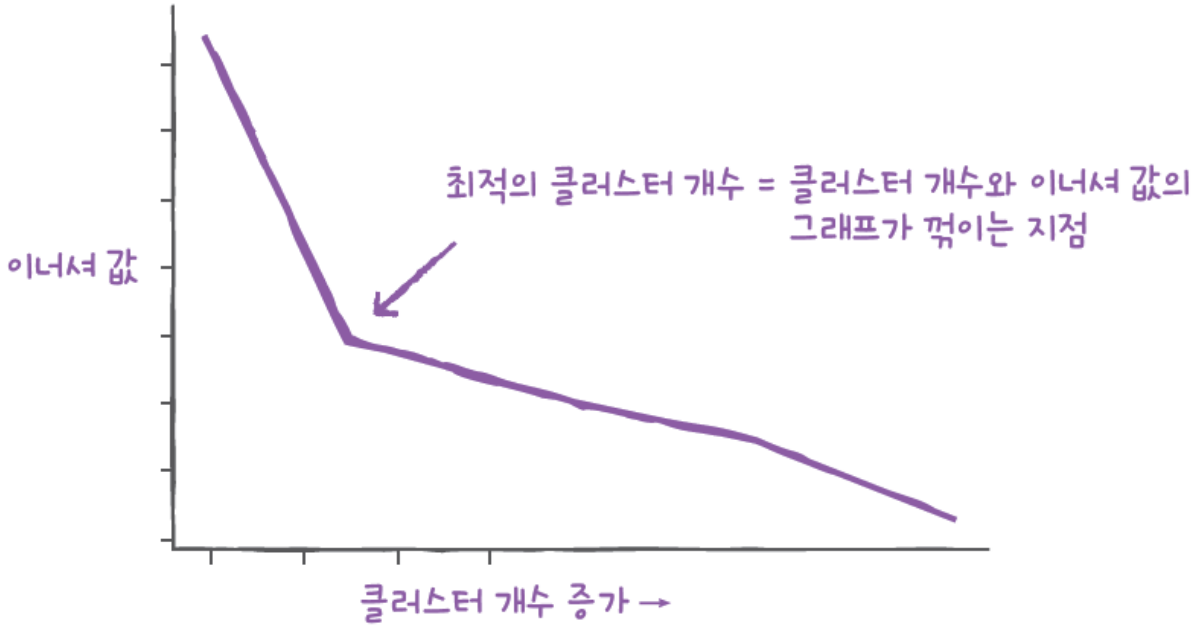
| In | draw_fruits(fruits[100:101]) |
|---|---|
| Out |  |

- 알고리즘이 반복한 횟수는 KMeans 클래스의 n_iter_ 속성에 저장됨

| In | print(km.n_iter_) |
|---|---|
| Out | 4 |

# 최적의 k 찾기

- k-평균 알고리즘의 단점 중 하나는 클러스터 개수를 사전에 지정해야 한다는 점
- 엘보우(elbow) 방법: 적절한 클러스터 개수를 찾기 위한 대표적인 방법
  - 이너셔(inertia): k-평균 알고리즘은 클러스터 중심과 클러스터에 속한 샘플 사이 거리의 제곱 합
  - 이너셔는 클러스터에 속한 샘플이 얼마나 가깝게 모여 있는지를 나타내는 값



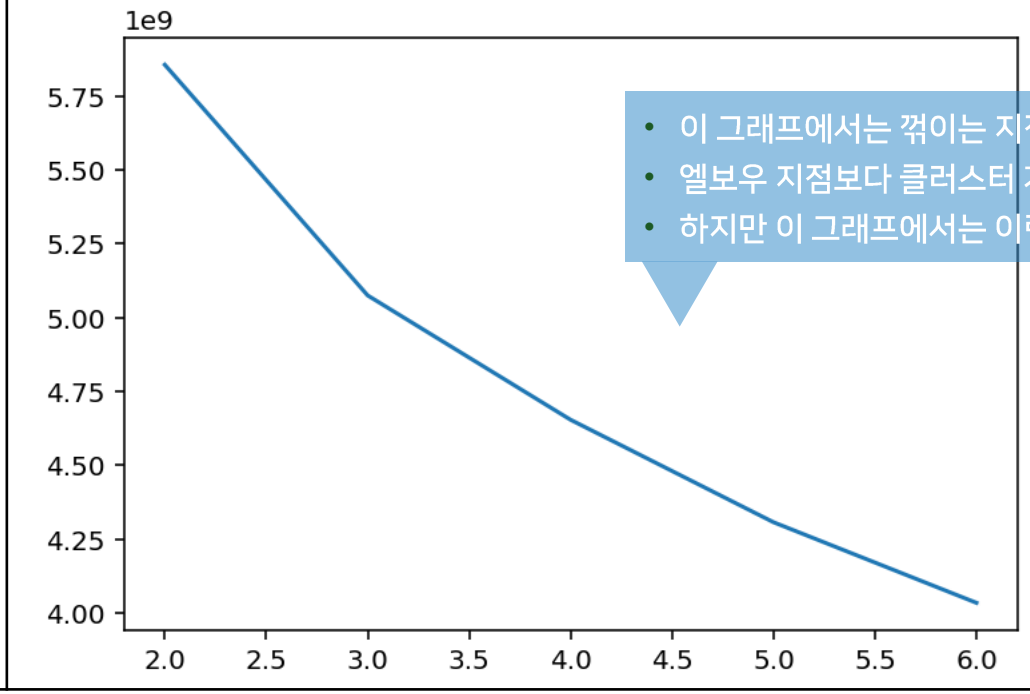
# 최적의 k 찾기

- 엘보우(elbow) 방법: 적절한 클러스터 개수를 찾기 위한 대표적인 방법
  - 과일 데이터셋을 사용해 이너셔 계산하기

In

```
inertia = []
for k in range(2, 7):
        km = KMeans(n_clusters=k, random_state=42)
        km.fit(fruits_2d)
        inertia.append(km.inertia_)
plt.plot(range(2, 7), inertia)
plt.show()
```

Out



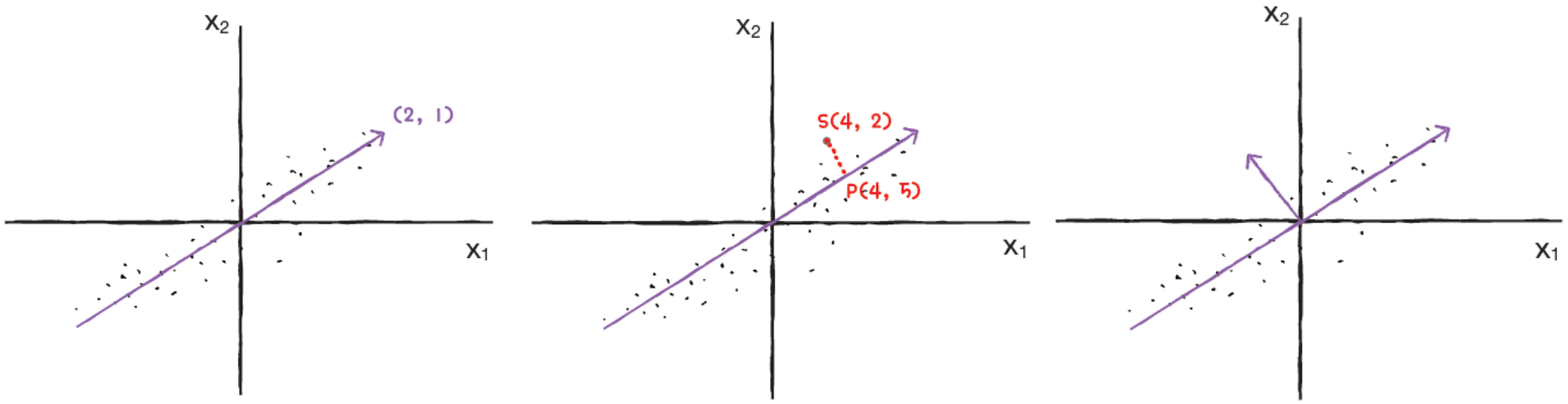

- 이 그래프에서는 꺾이는 지점이 뚜렷하지는 않지만 k = 3에서 그래프의 기울기가 조금 바뀐 것을 볼 수 있음
- 엘보우 지점보다 클러스터 개수가 많아지면 이너셔의 변화가 줄어들면서 군집 효과도 줄어듬
- 하지만 이 그래프에서는 이런 지점이 명확하지는 않음

# 주성분 분석

- 차원과 차원 축소
  - 특성은 데이터가 가진 속성. 과일 사진의 경우 10,000개의 픽셀은 10,000개의 특성이 있는 셈
  - 머신러닝에서는 이런 특성을 차원(dimension)이라고도 함
  - 10,000개의 특성은 결국 10,000개의 차원이라는 건데 이 차원을 줄일 수 있다면 저장 공간을 크게 절약
- 차원 축소(dimensionality reduction) 알고리즘
  - 차원 축소는 데이터를 가장 잘 나타내는 일부 특성을 선택하여 데이터 크기를 줄이고 지도 학습 모델의 성능을 향상시킬 수 있는 방법
  - 또한 줄어든 차원에서 다시 원본 차원(예를 들어 과일 사진의 경우 10,000개의 차원)으로 손실을 최대한 줄이면서 복원할 수도 있음
  - 주성분 분석(principal component analysis, PCA)은 대표적인 차원 축소 알고리즘

# 주성분 분석 소개

- 주성분 분석(PCA)은 데이터에 있는 분산이 큰 방향을 찾는 것
- 분산이 큰 방향을 찾은 직선이 원점에서 출발한다면 두 원소로 이루어진 벡터로 표현 가능
- 주성분 벡터의 원소 개수는 원본 데이터셋에 있는 특성 개수와 같음
- 원본 데이터는 주성분을 사용해 차원을 줄일 수 있음
- 주성분은 원본 차원과 같고 주성분으로 바꾼 데이터는 차원이 줄어듦




▲ 실제로 사이킷런의 PCA 모델을 훈련하면 자동으로 특성마다 평균값을 빼서 원점에 맞춰 줌.
▲ 따라서 우리가 수동으로 데이터를 원점에 맞출 필요가 없음

# PCA 클래스

- 이전과 마찬가지로 과일 사진 데이터를 다운로드하여 넘파이 배열로 적재
  - (샘플 개수, 너비, 높이) 크기의 3차원 배열을 (샘플 개수, 너비×높이) 크기를 가진 2차원 배열로 변경

| In | import numpy as np<br>fruits = np.load('fruits_300.npy')<br>fruits_2d = fruits.reshape(-1, 100*100) |
|---|---|

- PCA 클래스로 주성분 분석

| In | from sklearn.decomposition import PCA<br>pca = PCA(n_components=50)<br>pca.fit(fruits_2d) |
|---|---|
| Out | PCA(n_components=50) |

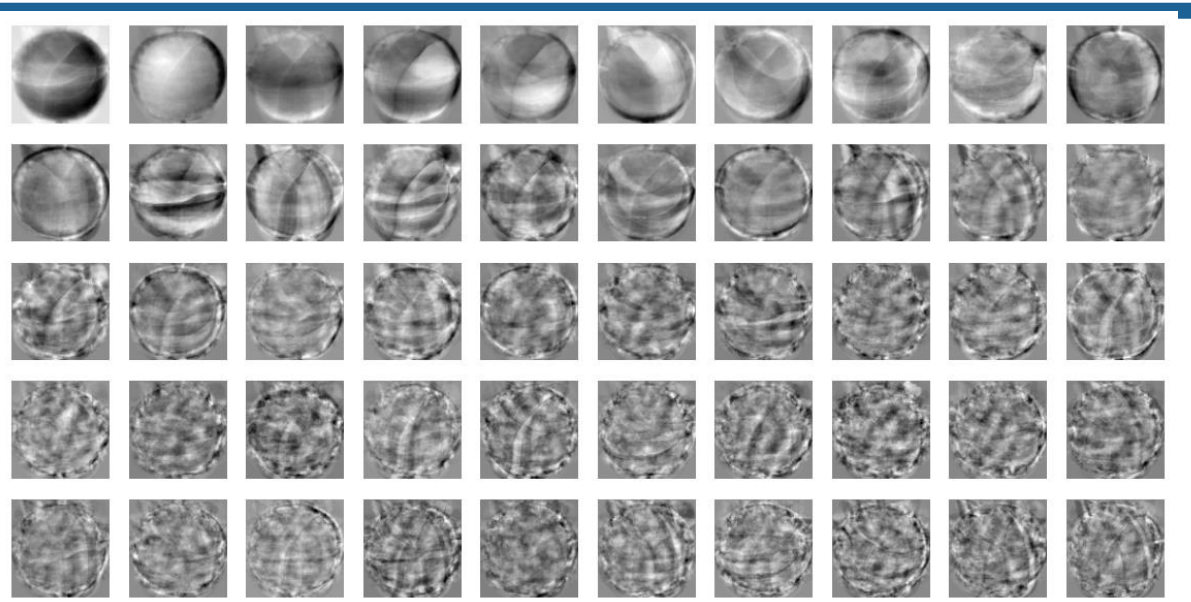
- 배열의 크기 확인

| In | print(pca.components_.shape) |
|---|---|
| Out | (50, 10000) |

# PCA 클래스

- draw_fruits( ) 함수를 사용해서 주성분 그리기

| | |
|---|---|
| In | draw_fruits(pca.components_.reshape(-1, 100, 100)) |
| Out | |



- 원본 데이터를 주성분에 투영하여 특성의 개수를 10,000개에서 50개로 줄일 수 있음

| | |
|---|---|
| In | print(fruits_2d.shape) |
| Out | (300, 10000) |

- PCA의 transform( ) 메서드를 사용해 원본 데이터의 차원을 50으로 축소

| | |
|---|---|
| In | fruits_pca = pca.transform(fruits_2d)<br>print(fruits_pca.shape) |
| Out | (300, 50) |

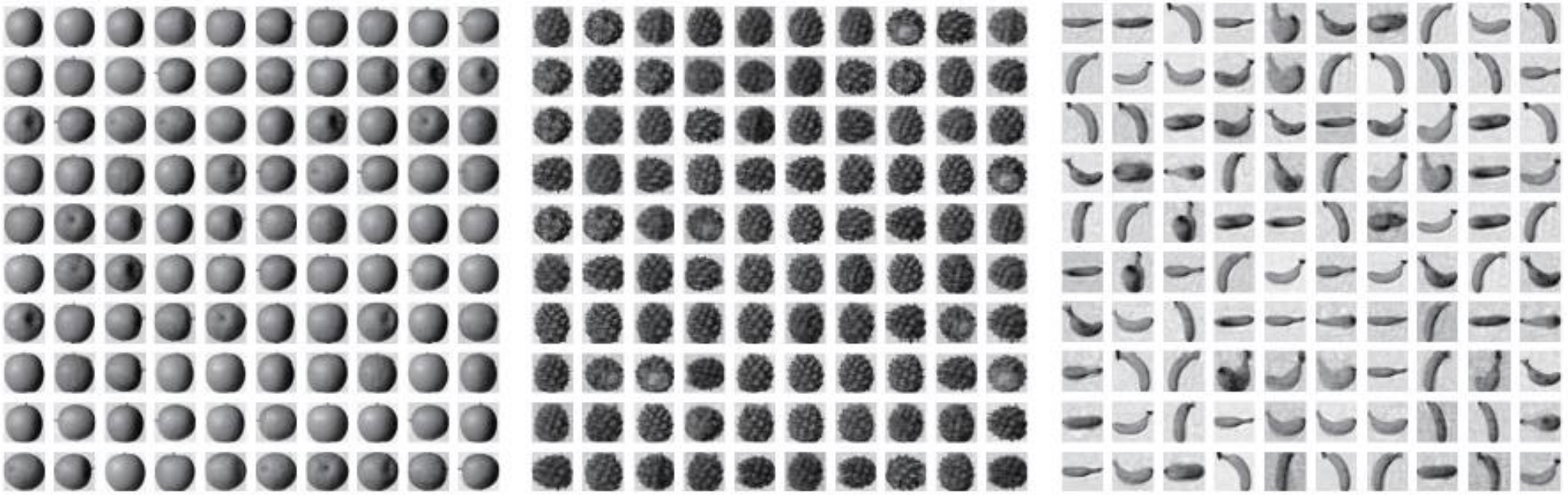
▲ 데이터가 1/200으로 축소

# SECTION 6-3 주성분 분석(5)

- inverse_transform( ) 메서드 사용해 앞서 50개의 차원으로 축소한 fruits_pca 데이터를 전달해 10,000개의 특성을 복원

| | |
|---|---|
| In | `fruits_inverse = pca.inverse_transform(fruits_pca)`<br>`print(fruits_inverse.shape)` |
| Out | `(300, 10000)` |

- 100 × 100 크기로 바꾸어 100개씩 나누어 출력

| | |
|---|---|
| In | `fruits_reconstruct = fruits_inverse.reshape(-1, 100, 100)`<br>`for start in [0, 100, 200]:`<br>`  draw_fruits(fruits_reconstruct[start:start+100])`<br>`  print("\n")` |
| Out |  |

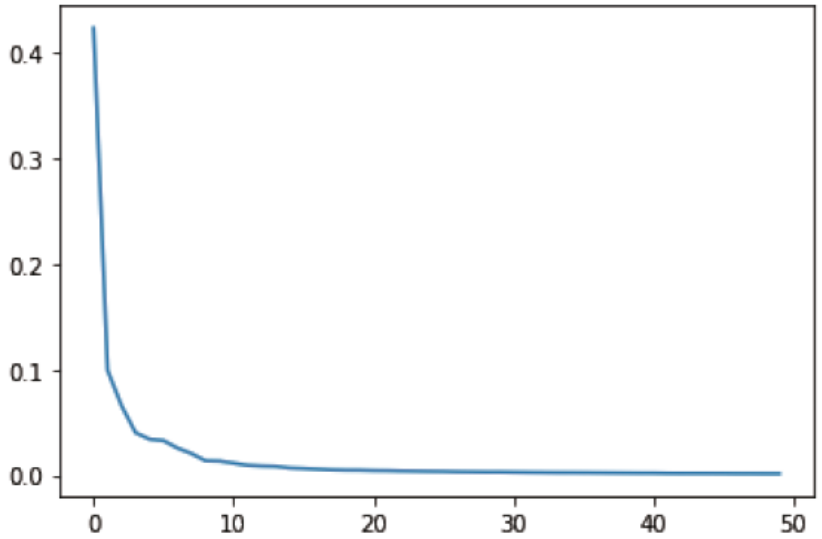
# 설명된 분산(explained variance)

- 주성분이 원본 데이터의 분산을 얼마나 잘 나타내는지 기록한 값
- PCA 클래스의 explained_variance_ratio_에 각 주성분의 설명된 분산 비율이 기록
  - 첫 번째 주성분의 설명된 분산이 가장 큼
  - 분산 비율을 모두 더하면 50개의 주성분으로 표현하고 있는 총 분산 비율을 얻을 수 있음

| | |
|---|---|
| In | `print(np.sum(pca.explained_variance_ratio_))` |
| Out | `0.9215387234611283` |

- 맷플롯립의 plot( ) 함수로 설명된 분산을 그래프로 출력

| | |
|---|---|
| In | `plt.plot(pca.explained_variance_ratio_)`<br>`plt.show()` |



# 다른 알고리즘과 함께 PCA 사용하기

- 과일 사진 원본 데이터와 PCA로 축소한 데이터를 지도 학습에 적용해 보고 차이점 관찰
- 3개의 과일 사진을 분류를 위해 사이킷런의 LogisticRegression 모델 만들기

| In | from sklearn.linear_model import LogisticRegression<br>lr = LogisticRegression() |
|---|---|

- 사과를 0, 파인애플을 1, 바나나를 2로 지정하여, 100개의 0, 100개의 1, 100개의 2로 이루어진 타깃 데이터 만들기

| In | target = np.array([0]*100 + [1]*100 + [2]*100) |
|---|---|

- cross_validate( )로 교차 검증 수행

| In | from sklearn.model_selection import cross_validate<br>scores = cross_validate(lr, fruits_2d, target)<br>print(np.mean(scores['test_score']))<br>print(np.mean(scores['fit_time'])) |
|---|---|
| Out | 0.9966666666666667<br>0.11803574562072754 |

- PCA로 축소한 fruits_pca를 사용했을 때와 비교

| In | scores = cross_validate(lr, fruits_pca, target)<br>print(np.mean(scores['test_score']))<br>print(np.mean(scores['fit_time'])) |
|---|---|
| Out | 0.9966666666666667<br>0.006903696060180664 |

# 다른 알고리즘과 함께 PCA 사용하기

- n_components 매개변수에 분산의 비율을 입력하여. 설명된 분산의 50%에 달하는 주성분을 찾도록 PCA 모델 만들기

| | |
|---|---|
| In | pca = PCA(n_components=0.5)<br>pca.fit(fruits_2d) |
| Out | PCA(n_components=0.5) |

- 탐색한 주성분 개수 확인

| | |
|---|---|
| In | print(pca.n_components_) |
| Out | 2 |

- 이 모델로 원본 데이터를 변환

| | |
|---|---|
| In | fruits_pca = pca.transform(fruits_2d)<br>print(fruits_pca.shape) |
| Out | (300, 2) |

- 교차 검증의 결과 확인

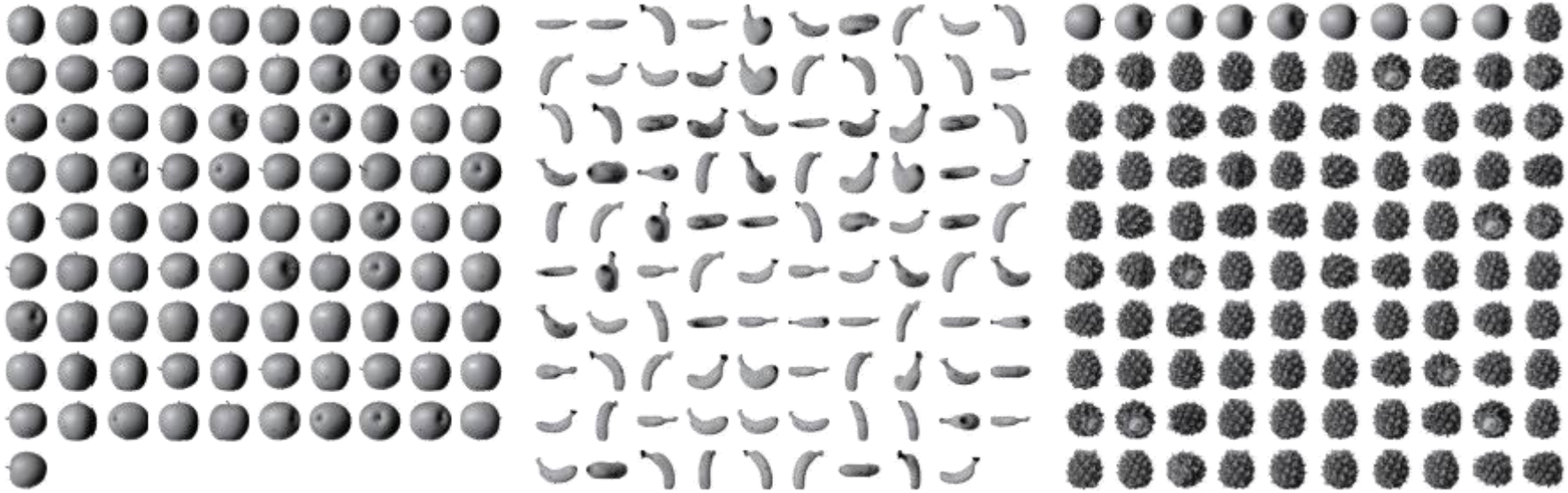
| | |
|---|---|
| In | scores = cross_validate(lr, fruits_pca, target)<br>print(np.mean(scores['test_score']))<br>print(np.mean(scores['fit_time'])) |
| Out | 0.9933333333333334<br>0.01223602294921875 |

# 다른 알고리즘과 함께 PCA 사용하기

- 차원 축소된 데이터를 사용해 k-평균 알고리즘으로 클러스터 찾기

| | |
|---|---|
| In | `from sklearn.cluster import KMeans`<br>`km = KMeans(n_clusters=3, random_state=42)`<br>`km.fit(fruits_pca)`<br>`print(np.unique(km.labels_, return_counts=True))` |
| Out | `(array([0, 1, 2], dtype=int32), array([110,  99,  91]))` |

- KMeans가 찾은 레이블을 사용해 과일 이미지 출력

| | |
|---|---|
| In | `for label in range(0, 3):`<br>`  draw_fruits(fruits[km.labels_ == label])`<br>`  print("\n")` |
| Out |  |

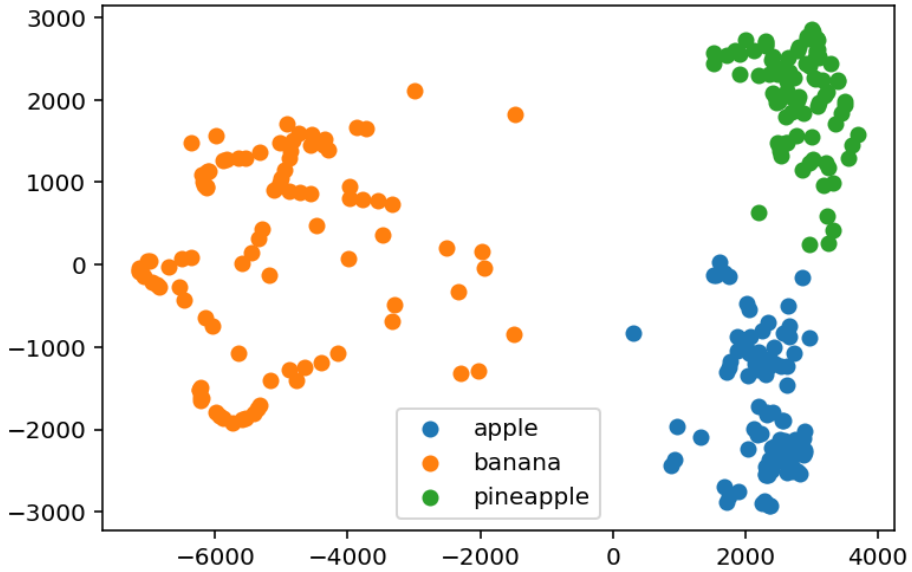
# 다른 알고리즘과 함께 PCA 사용하기

- 훈련 데이터의 차원을 줄이면 또 하나 얻을 수 있는 장점은 시각화
- 3개 이하로 차원을 줄이면 화면에 출력하기 비교적 쉬움
- fruits_pca 데이터는 2개의 특성이 있기 때문에 2차원으로 표현
- 앞에서 찾은 km.labels_를 사용해 클러스터별로 나누어 산점도 그리기

In

```
for label in range(0, 3):
      data = fruits_pca[km.labels_ == label]
      plt.scatter(data[:,0], data[:,1])
plt.legend(['apple', 'banana', 'pineapple'])
plt.show()
```

Out



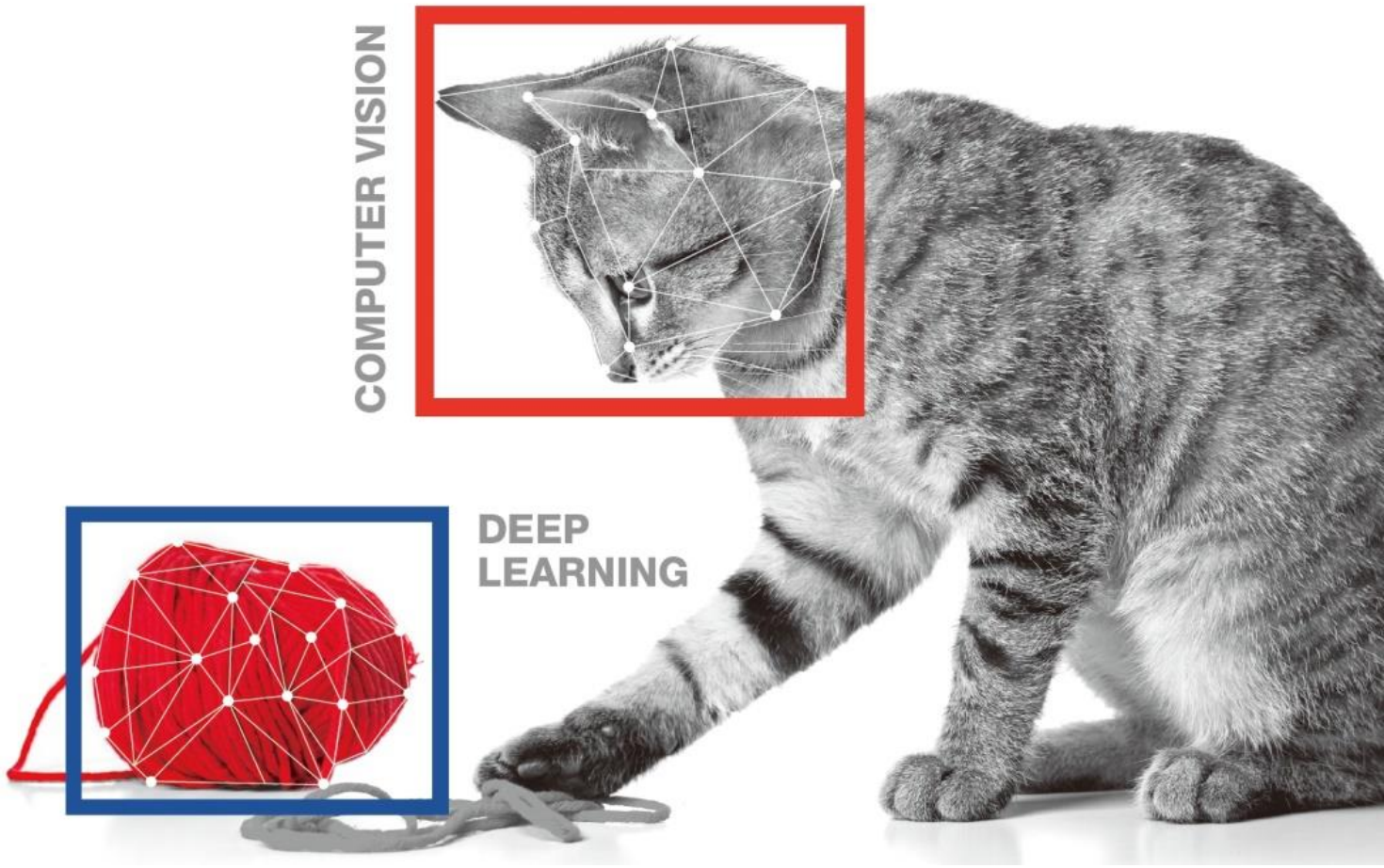
# 요약

- 비지도 학습
  - 머신러닝의 한 종류, 훈련 데이터에 타깃(정답)이 없음
  - 외부 도움 없이 데이터 속 패턴·구조를 스스로 학습
  - 주요 작업: 군집, 차원 축소
  - 히스토그램: 값의 구간별 발생 빈도 시각화
  - 군집(Cluster): 비슷한 샘플끼리 그룹화

- k-평균 알고리즘
  - 랜덤하게 클러스터 중심 설정 후 반복적으로 갱신
  - 클러스터 중심 = 속한 샘플 특성의 평균값(센트로이드)
  - 가장 가까운 중심을 기준으로 샘플 배정
  - 엘보우 방법: 최적의 클러스터 개수 결정
  - 이너셔: 중심과 샘플 간 거리 제곱합

# 요약

- 차원 축소 & PCA
  - 원본 데이터를 적은 수의 특성으로 변환
  - 저장 공간 절약·시각화 용이·성능 향상 가능
  - 주성분 분석(PCA): 가장 분산이 큰 방향(주성분) 탐색
  - 데이터를 주성분에 투영하여 새 특성 생성
  - 설명된 분산: 주성분이 원본 분산을 얼마나 잘 나타내는지




# Thank You:)

Any Question?