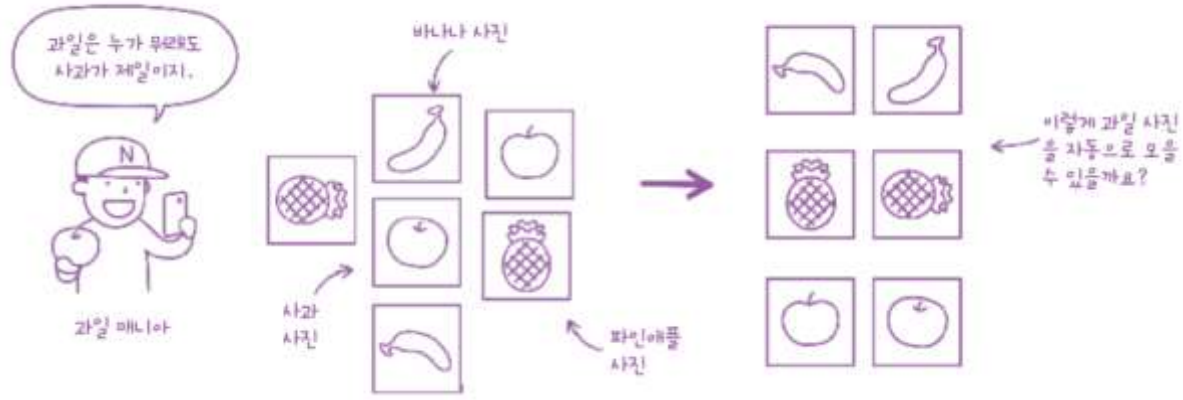
[군집 알고리즘]

- 이전시간까지 다양한 머신러닝 알고리즘을 사용해 물고기 데이터와, 와인분류 데이터를 활용해 실습을 진행했다.
- 고객이 마켓에서 사고 싶은 과일 사진을 보내면 그 중 가장 많이 요청하는 과일을 판매 품목으로 선정하려 한다. 또 1 위로 선정된 과일 사진을 보낸 고객 중 몇 명을 뽑아 이벤트 당첨자로 선정할 것이다.



- 고객이 올린 사진을 사람이 하나씩 분류하기는 어렵다. 그렇다고 생선처럼 미리 과일 분류기를 훈련하기에는 고객들이 어떤 과일 사진을 보낼지 알 수 없으니 곤란하다. 사진에 대한 정답(타깃)을 알지 못하는데 어떻게 이 사진을 종류대로 모을 수 있을까
- 타깃을 모르는 사진을 종류별로 분류하려 한다.
- 타깃을 알지 못하는 데이터를 활용해 분류작업을 진행해야 한다. 이때 **비지도 학습**을 이용해야 한다.
- **비지도 학습**은 사람이 가르쳐 주지 않아도 데이터에 있는 무언가를 스스로 학습하는 것을 말한다.

- 이번에는 캐글에 공개된 과일 흑백 사진 데이터(사과, 바나나, 파인애플)를 활용할 것이다. 이 데이터는 넘파이 기본 저장 포맷인 npy 파일로 저장되어 있다. 이 파일을 불러오도록 한다.

이 과일 데이터는 캐글에 공개된 데이터셋입니다.

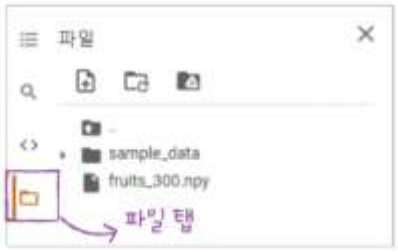
- https://www.kaggle.com/moltean/fruits

손코딩

```
!wget https://bit.ly/fruits_300_data -O fruits_300.npy
```

✚ 여기서 잠깐 !는 뭔가요?

코랩의 코드 셀에서 '!' 문자로 시작하면 코랩은 이후 명령을 파이썬 코드가 아니라 리눅스 셸(shell) 명령으로 이해합니다. wget 명령은 원격 주소에서 데이터를 다운로드하여 저장합니다. -O 옵션에서 저장할 파일 이름을 지정할 수 있습니다.
이 명령을 실행하고 나서 코랩의 왼쪽 파일 탭을 열면 다음 그림처럼 fruits_300.npy가 저장된 것을 볼 수 있습니다.



!wget https://bit.ly/fruits_300_data -O fruits_300.npy

넘파이에서 npy 파일을 로드하려면 load() 메서드에 파일이름을 전달하면 된다.

```
import numpy as np
import matplotlib.pyplot as plt

fruits = np.load('fruits_300.npy')

#fruits 배열의 크기 확인
print(fruits.shape)

(300, 100, 100)
```

- fruits 는 넘파이 배열이고 fruits_300.npy 파일에 들어 있는 모든 데이터를 담고 있다.
- 이 배열의 첫 번째 차원(300)은 샘플의 개수를 나타내고, 두 번째 차원(100)은 이미지 높이, 세 번째 차원(100)은 이미지의 너비를 나타낸다. 이미지 크기는

100x100 이며, 각 픽셀은 넘파이 배열의 원소 하나에 대응하기 때문에 배열의 크기 또한 100x100 이다.

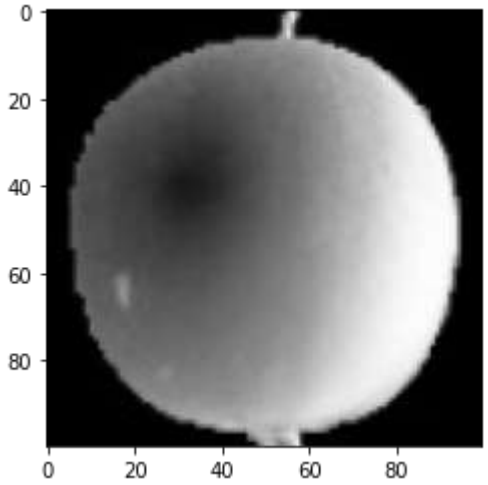
- 먼저 첫 번째 행을 출력해 보자. 3 차원 배열이기 때문에 처음 2 개의 인덱스를 0 으로 지정하고 마지막 인덱스는 지정하지 않거나 슬라이싱 연산자를 써서 첫 번째 이미지의 첫 번째 행을 모두 선택 할 수 있다.

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

```
[  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]
```

- 첫 번째 행에 있는 픽셀 100 개에 들어 있는 값을 출력했다. 이 넘파이 배열은 흑백사진을 담고 있으므로 0~255 의 정숫값을 가진다. 첫 번째 이미지를 그림으로 그려서 이 숫자와 비교하자.
- 맷플롯립의 imshow()함수를 이용하면 넘파이 배열로 저장된 이미지를 쉽게 그릴 수 있다.

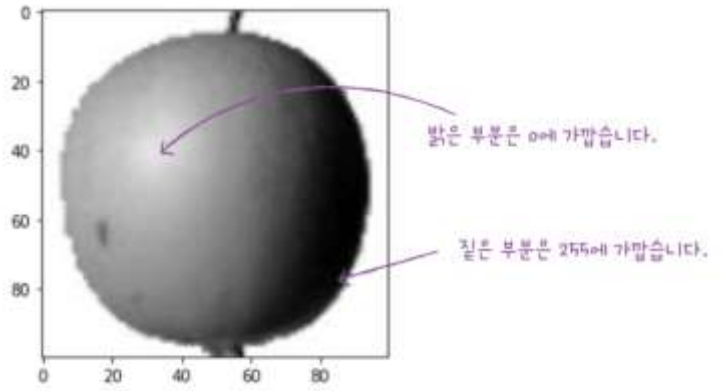
```
plt.imshow(fruits[0], cmap='gray')
plt.show()
```



- 첫 번째 이미지는 사과처럼 보인다. 그림에서 알 수 있는 것 처럼 첫 번째 행이 위에서 출력한 배열 값에 해당한다. 0 에 가까울 수록 검게 나타나고 높은 값은 밝게 표시된다.

- 그림이 우리가 아는 흑백의 사진과는 조금 다른 형태를 띄는 것을 확인할 수 있다. 자연스럽게 만드려면 'gray_r'을 이용하면 된다. 그렇게 하면 값이 낮을수록 밝아지고 높을수록 짙어진다. (바탕이 흰색이 된다)

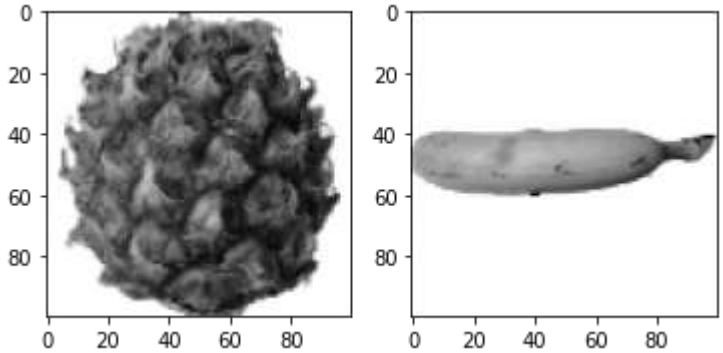
```
plt.imshow(fruits[0], cmap='gray_r')
plt.show()
```



밝은 부분이 0 에 가깝고 짙은 부분이 255 에 가까운 값이다.

- 해당 데이터에는 사과, 바나나, 파인애플이 각각 100 개씩 포함되어있다. 바나나와 파인애플도 출력해보아야 한다.

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

<matplotlib.image.AxesImage at 0x1258e48b0>
```

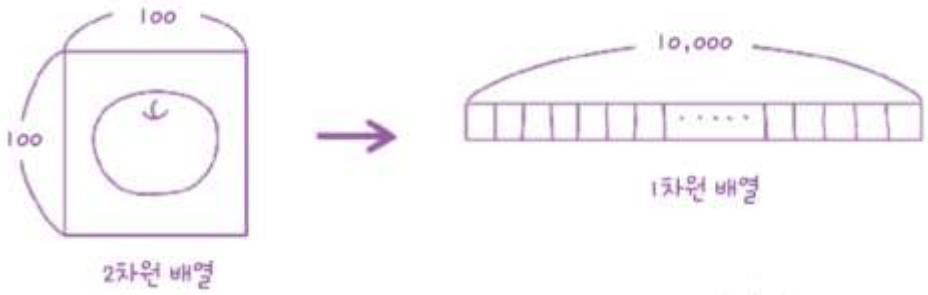


- 맷플롯립의 subplots()함수를 사용하면 여러 개의 그래프를 배열처럼 쌓을 수 있도록 도와준다. subplots() 함수의 두 매개변수는 그래프를 쌓을 행과 열을 지정한다. 여기에서는 subplots(1,2) 처럼 하나의 행과 2 개의 열을 지정했다.

- 반환된 axs 는 2 개의 서브 그래프를 담고 있는 배열이다. axs[0]에 파인애플 이미지를 axs[1]에 바나나 이미지를 그렸다.
- 데이터 준비가 모두 끝났다. 이 데이터의 처음 100 개는 사과, 그 다음 100 개는 파인애플, 마지막 100 개는 바나나이다. 각 과일 사진의 평균을 내서 차이를 살펴보도록 하자.

2. 픽셀값 분석하기

- 사용하기 쉽게 fruits 데이터를 사과, 파인애플, 바나나로 각각 나누어보자.
- 넘파이 배열을 나눌 때 100 x 100 이미지를 펼쳐서 길이가 10,000 인 1 차원 배열로 만든다. 이렇게 펼치면 이미지로 출력하기 어렵지만 배열을 계산할 때 편리하다.



- fruits 배열에서 순서대로 100 개씩 선택하기 위해 슬라이싱 연산자를 사용한다. reshape() 메서드를 사용해 두 번째 차원(100)과 세 번째 차원(100)을 10,000 으로 합친다. 첫 번째 차원을 -1 로 지정하면 자동으로 남은 차원을 할당한다. 여기에서는 첫 번째 차원이 샘플 개수를 의미한다.

```
apple = fruits[0:100].reshape(-1, 100*100)
pineapple = fruits[100:200].reshape(-1, 100*100)
banana = fruits[200:300].reshape(-1, 100*100)
```

- 각각의 과일의 배열크기는 (100, 10000)이다. 아래에서 확인해볼 수 있다.

```
print(apple.shape)

(100, 10000)
```

- 각각의 배열에 들어있는 샘플의 픽셀 평균값을 구해보도록 하자. 넘파이 mean() 메서드를 이용해 평균을 구할 수 있다. 샘플마다 픽셀의 평균값을 계산해야 하기

때문에 axis=0 으로 하면 첫 번째 축인 행을 따라 계산한다. axis=1 로 지정하면 두 번째 축인 열을 따라 계산한다.

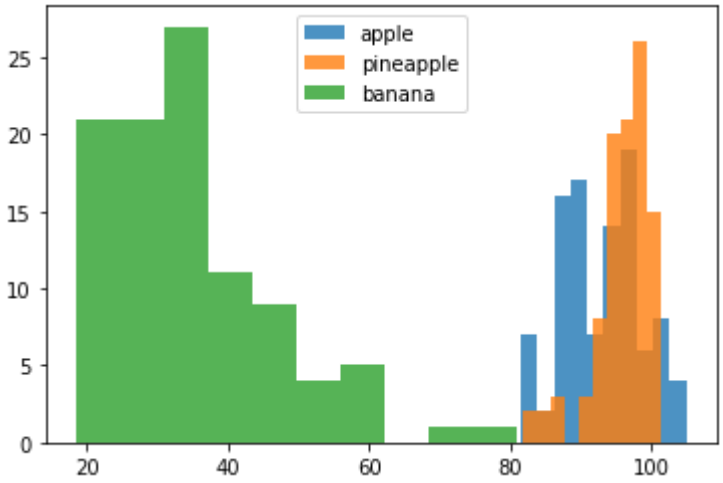
- 우리는 샘플을 가로로 나열했기 때문에 열을 기준으로 계산을 해야 한다. 따라서 axis=1 을 사용한다.

```
print(apple.mean(axis=1))
```

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

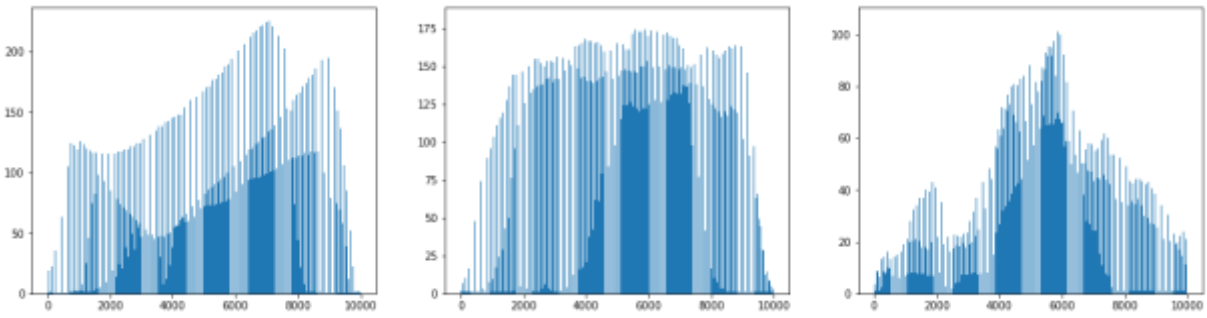
- 사과 샘플 100 개에 대한 픽셀 평균값을 계산했다. 이를 히스토그램을 통해 어떻게 평균값이 분포되어 있는지 알 수 있다.
- 맷플롯립의 hist()함수를 사용하면 히스토그램을 그릴수 있다.
- alpha 매개변수를 1 보다 작게 하면 투명도를 줄 수 있다.

```
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()
```



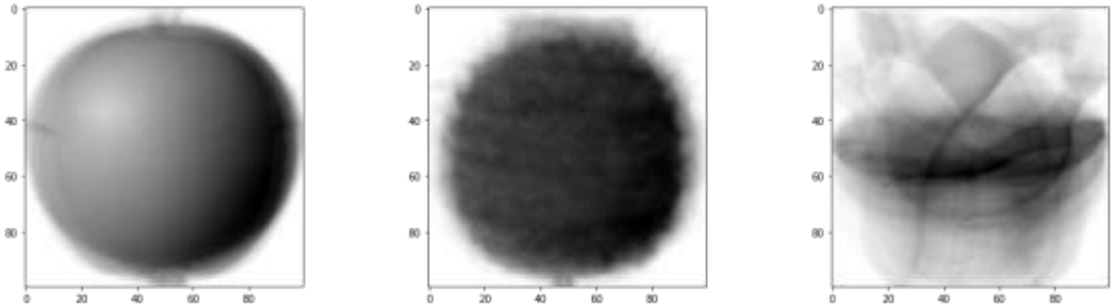
- 히스토그램을 살펴보니 바나나 사진의 평균값은 40 아래에 집중되어있는 모습을 보인다. 사과와 파인애플은 90~100 사이에 가장 많은 수가 모여있는 것을 확인할 수 있다. 바나나는 픽셀 평균값만으로 사과나 파인애플과 확실히 구분된다. 바나나는 사진에서 차지하는 영역이 작기 때문에 평균값이 작다.
- 반면 사과와 파인애플은 많이 겹쳐있어서 픽셀값만으로는 구분하기 쉽지 않다. 사과나 파인애플은 대체로 형태가 동그랗고 사진에서 차지하는 크기도 비슷하기 때문이다.
- 샘플의 평균값이 아니라 픽셀별 평균값을 비교해보면 어떨까. 전체 샘플에 대해 각 픽셀의 평균을 계산하는 것이다. 세 과일은 모양이 다르므로 픽셀값이 높은 위치가 조금 다를 것 같다.
- 픽셀의 평균을 계산하려면 axis=0 으로 지정하면 된다. 맷플롯립의 bar() 함수를 이용해 픽셀 10,000 개에 대한 평균값을 막대그래프로 그리자. subplots() 함수로 3 개의 서브 그래프를 만들어 사과, 파인애플, 바나나에 대한 막대그래프를 그려보자

```
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))

<BarContainer object of 10000 artists>
```



- 순서대로 사과, 파인애플, 바나나 그래프이다. 각 과일마다 값이 높은 구간이 다른 형태를 보인다. 사과는 사진 아래쪽으로 갈수록 값이 높아지고, 파인애플은 비교적 고르면서 높다. 바나나는 확실히 중앙의 픽셀값이 높다
- 픽셀 평균값을 100 x 100 크기로 바꿔서 이미지처럼 출력하여 위 그래프와 비교하면 더 좋다. 픽셀을 평균 낸 이미지를 모든 사진을 합쳐 놓은 대표 이미지로 생각할 수 있다.

```
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')

<matplotlib.image.AxesImage at 0x13504e460>
```



- 세 과일은 픽셀 위치에 따라 값의 크기가 차이가 난다. 따라서 이 대표 이미지와 가까운 사진을 골라낼 수 있으면 사과, 파인애플, 바나나를 구분 할 수 있을 것이다.

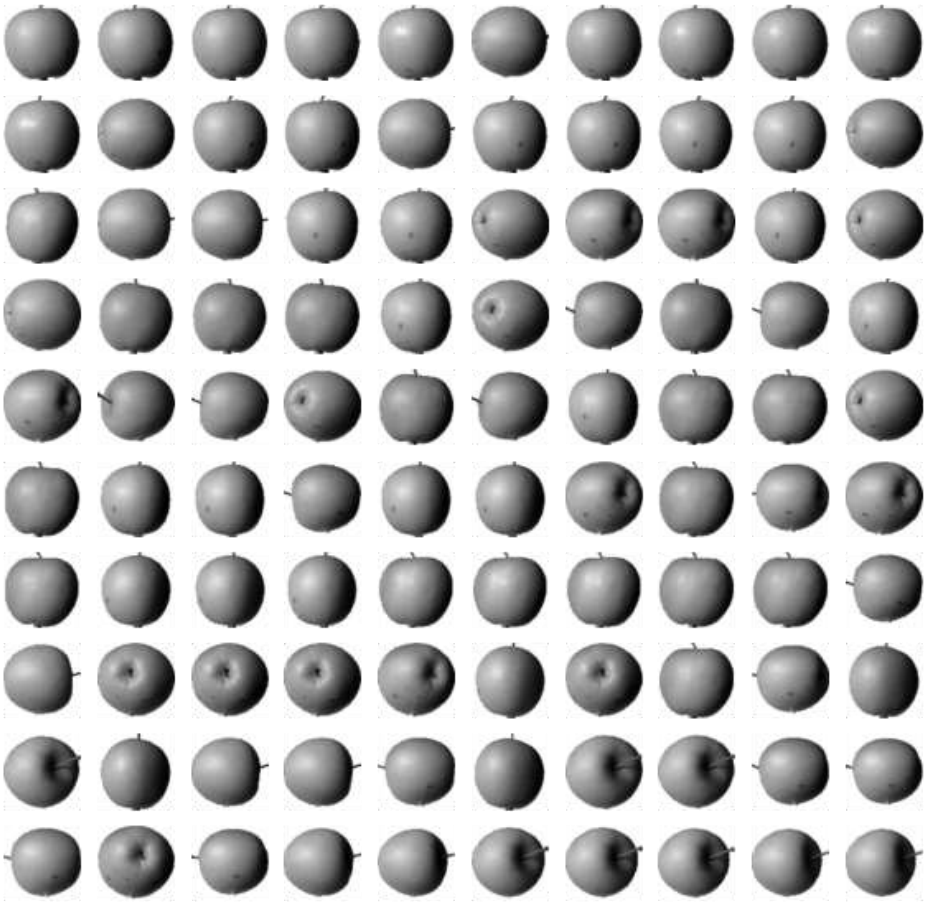
3. 평균값과 가까운 사진 고르기

- 사과 사진의 평균값인 apple_mean 과 가장 가까운 사진을 골라보자. 절댓값 오차를 사용해 구해보자. fruits 배열에 있는 모든 샘플에서 apple_mean 을 뺀 절댓값의 평균을 계산하면 된다.
- abs()함수를 이용해 절댓값 계산이 가능하다. 배열을 입력하면 모든 원소의 절대값을 계산하여 입력과 동일한 크기의 배열을 반환한다. 다음 코드에서 abs_diff 는 (300, 100, 100)크기의 배열이다. 따라서 각 샘플에 대한 평균을 구하기 위해 axis 에 두 번째, 세 번째 차원을 모두 지정했다. 이렇게 계산한 abs_mean 은 각 샘플의 오차 평균이기 때문에 크기가 (300,)인 1 차원 배열이다.

```
abs_diff = np.abs(fruits - apple_mean)
abs_mean = np.mean(abs_diff, axis=(1,2))
print(abs_mean.shape)

(300,)
```

- 이 값이 가장 작은 순서대로 100 개를 골라보자. 즉 apple_mean 과 오차가 가장 작은 샘플 100 개를 고르는 것이다. np.argsort() 함수는 작은 것에서 큰 순서대로 나열한 abs_mean 배열의 인덱스를 반환한다. 이 인덱스 중에서 처음 100 개를 선택해 10x10 격자로 이루어진 그래프를 그려보자.

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



apple_mean 과 가장 가까운 사진 100 개를 골랐더니 모두 사과이다.

먼저 subplots() 함수로 10 X 10, 총 100 개의 서브 그래프를 만든다. 그래프가 많기 때문에 전체 그래프의 크기를 figsize=(10,10)으로 조금 크게 지정했다.

그 다음 2 중 for 반복문을 순회하면서 10 개의 행과 열에 이미지를 출력한다. axs 는 (10,10)크기의 2 차원 배열이므로 i, j 두 첨자를 사용하여 서브 그래프 위치를 지정한다. 또 깔끔하게 이미지만 그리기 위해 axis('off')를 사용하여 좌표축을 그리지 않았다.

- 흑백 사진에 있는 픽셀값을 사용해 과일 사진을 모으는 작업을 진행해 보았다. 이렇게 비슷한 샘플끼리 그룹으로 모으는 작업을 **군집**이라고 한다. 군집은 대표적인 비지도 학습 작업 중 하나이다. 군집 알고리즘에서 만든 그룹을 **클러스터**라고 부른다.
- 이번에는 어떤 타깃이 존재하는지 알고 있었다. (사과, 파인애플, 바나나가 있다는 것을 알고 있었다.) 타깃값을 알고 있었기 때문에 사과, 파인애플, 바나나의 사진

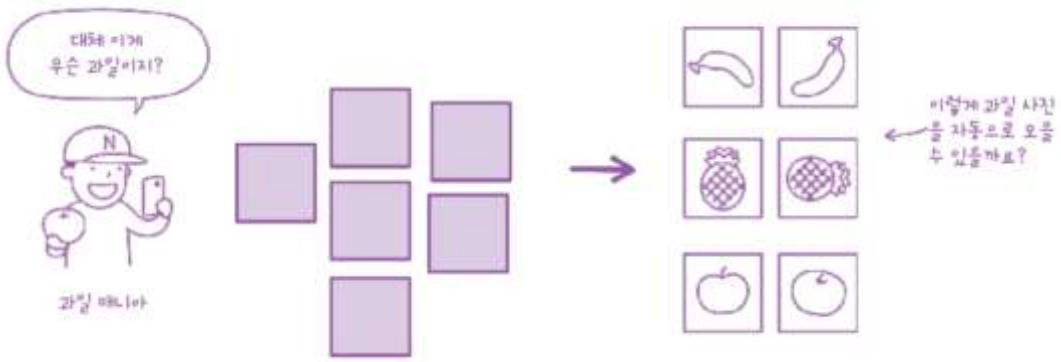
평균값을 계산해서 가장 가까운 과일을 찾을 수 있었다. 하지만 실제로 비지도 학습에서는 타깃값을 모르기 때문에 이처럼 샘플의 평균값을 미리 구할 수 없다. 타깃값을 모르면서 어떻게 세 과일의 평균값을 찾을 수 있을까에 대한 해답은 k-평균 알고리즘에 있다.

비슷한 샘플끼리 모으기 문제해결 과정

- 새로운 이벤트를 위해 고객들이 올린 과일 사진을 자동으로 모아야 한다. 어떤 과일 사진을 올릴지 미리 예상할 수 없기 때문에 타깃값을 준비하여 분류모델을 훈련하기 어렵다.
- 타깃값이 없을 때 데이터에 있는 패턴을 찾거나 데이터 구조를 파악하는 머신러닝 방식을 비지도 학습이라고 한다. 타깃이 없기 때문에 알고리즘을 직접적으로 가르칠 수가 없다. 대신 알고리즘은 스스로 데이터가 어떻게 구성되어 있는지 분석한다.
- 대표적인 비지도 학습 문제는 군집이다. 군집은 비슷한 샘플끼리 그룹으로 모으는 작업이다. 이번에는 사진의 픽셀을 사용해 군집과 비슷한 작업을 수행해보았다. 하지만 샘플이 어떤 과일인지 미리 알고 있었기 때문에 사과 사진의 평균값을 알 수 있었다.
- 실제 비지도 학습에서는 타깃이 없는 사진을 사용해야 한다.

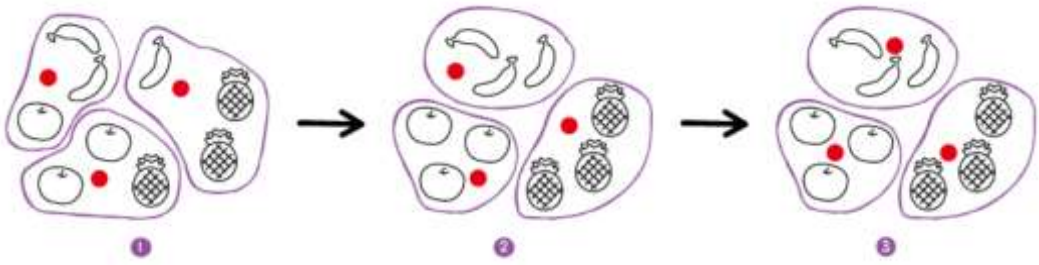
[k-평균]

- 바로 직전에는 사과, 파인애플, 바나나에 있는 각 픽셀의 평균값을 구해서 가장 가까운 사진을 고르는 작업을 수행했다. 이 경우에는 사과, 파인애플, 바나나 사진임을 미리 알고 있었기 때문에 각 과일의 평균을 구할 수 있었다.
- 하지만 실제로 비지도 학습에서는 사진에 어떤 과일이 들어 있는지 알지 못한다.
- 이런 경우 **k-평균 군집 알고리즘**을 사용한다. **k-평균 군집 알고리즘**이 평균값을 자동으로 찾아준다. 이 평균값이 클러스터 중심에 위치하기 때문에 **클러스터 중심** 또는 **센트로이드**라고 부른다.



1. k-평균 알고리즘

- k-평균 알고리즘 작동방식
  - 무작위로 k 개의 클러스터 중심을 정한다.
  - 각 샘플에서 가장 가까운 클러스터 중심을 찾아 해당 클러스터의 샘플로 지정한다.
  - 클러스터에 속한 샘플의 평균값으로 클러스터 중심을 변경한다.
  - 클러스터 중심에 변화가 없을 때까지 2 번으로 돌아가 반복한다.



- 먼저 3 개의 클러스터 중심(빨간 점)을 랜덤하게 지정한다. (1) 그리고 클러스터 중심에서 가장 가까운 샘플을 하나의 클러스터로 묶는다. 왼쪽 위부터 시계 방향으로 바나나 2 개와 사과 1 개 클러스터, 바나나 1 개와 파인애플 2 개 클러스터, 사과 2 개와 파인애플 1 개 클러스터가 만들어졌다, 클러스터에는 순서나 번호는 의미가 없다.
- 그 다음 클러스터의 중심을 다시 계산하여 이동시킨다. 맨 아래 클러스터는 사과쪽으로 중심이 조금 더 이동하고 왼쪽 위의 클러스터는 바나나 쪽으로 중심이 더 이동하는 식이다.

- 클러스터 중심을 다시 계산한 다음 가장 가까운 샘플을 다시 클러스터로 묶는다. (2) 이제 3 개의 클러스터에는 바나나와 파인애플, 사과가 3 개씩 올바르게 묶여 있다. 다시 한번 클러스터 중심을 계산한다. 그 다음 빨간 점을 클러스터의 가운데 부분으로 이동시킨다.
- 이동된 클러스터 중심에서 다시 한번 가장 가까운 샘플을 클러스터로 묶는다. (3) 중심에서 가장 가까운 샘플은 이전 클러스터와 동일하다. 따라서 만들어진 클러스터에 변동이 없으므로 k-평균 알고리즘을 종료한다.
- k-평균 알고리즘은 처음에는 랜덤하게 클러스터 중심을 선택하고 점차 가장 가까운 샘플의 중심으로 이동하는 알고리즘이다. 이를 사이킷런을 이용해 구현해보자

KMeans 클래스

- 데이터는 이전에 사용한 데이터 셋을 활용해 구현해 보도록 한다. np.load()함수를 이용해 npy 파일을 읽어 넘파이 배열을 준비하고 k-평균 모델 훈련을 위해 3 차원배열(샘플개수, 너비, 높이)을 2 차원(샘플개수, 너비 X 높이) 크기의 배열로 변경하는 작업을 수행한다.

```
import pandas as pd
import numpy as np
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

raw = 'fruits_300.npy'
fruits = np.load(raw)
fruits_2d = fruits.reshape(-1, 100*100)

km = KMeans(n_clusters=3, random_state=42)
km.fit(fruits_2d)

#결과
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
        n_clusters=3, n_init=10, n_jobs=None, precompute_distances='auto',
        random_state=42, tol=0.0001, verbose=0)
```

- 사이킷런의 k-평균 알고리즘은 sklearn.cluster 모듈 아래 KMeans 클래스에 구현되어 있다. 클러스터의 개수를 지정하기 위해 n_clusters 매개변수를 3 으로 지정했다.
- 비지도 학습이기 때문에 fit() 메서드에서 타깃데이터를 이용하지 않았다.
- 군집된 결과는 KMeans 클래스 객체의 labels_ 속성에 저장된다. labels 배열의 길이는 샘플 개수와 같다. 이 배열은 각 샘플이 어떤 레이블에 해당되는지 나타낸다. 여기서 클러스터의 개수를 3 으로 지정했기 때문에 labels_ 배열의 값은 0,1,2 중 하나로 나타나게 된다.

```
print(km.labels_)

[0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 2 0 2 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 2 2 0 0 0 0 0 0 0 0 2 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 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 1 1 1 1 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
 1 1 1 1 1 1 1 1 1 1 1 1 1 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]

print(np.unique(km.labels_, return_counts=True))

(array([0, 1, 2], dtype=int32), array([ 91,  98, 111]))
```

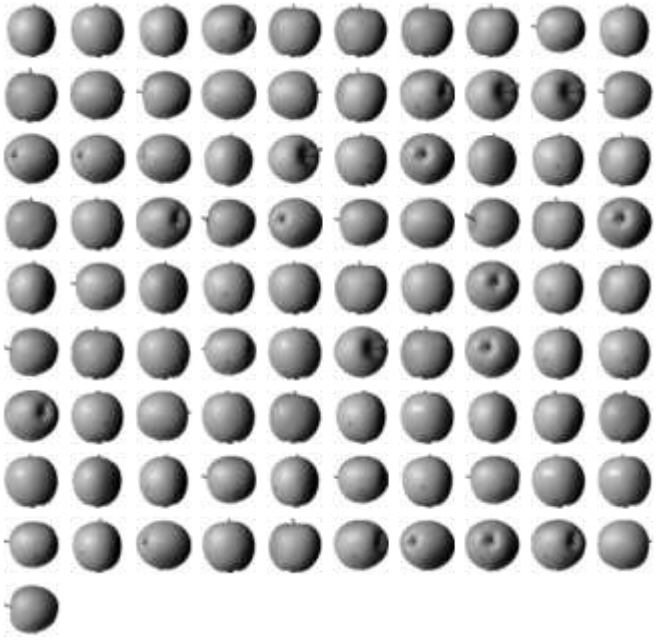
- 레이블값 0,1,2 와 레이블 순서에는 어떤 의미도 없다. 실제 레이블 0,1,2 가 어떤 과일 사진을 주로 모았는지 알아보려면 직접 이미지를 출력하는 것이 최선이다. 그 전에 레이블 0,1,2 로 모은 샘플의 개수를 확인하였다
- 첫 번째 클러스터(레이블 0)는 91 개의 샘플을 모았고, 두 번째 클러스터(레이블 1)는 98 개 그리고 세 번째 클러스터(레이블 2)는 111 개의 샘플을 모았다. 각 클러스터가 어떤 이미지를 나타냈는지 그림으로 출력하기 위해 간단한 유틸리티 함수 draw_fruits()를 만들어보자.

```
def draw_fruits(arr, ratio=1):
    n = len(arr)    # n 은 샘플 개수
```

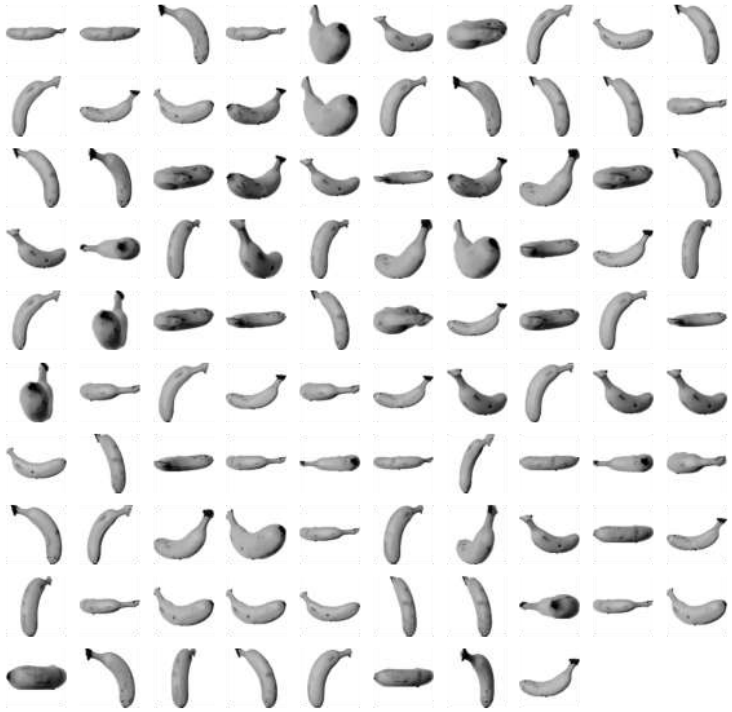
```
    # 한 줄에 10 개씩 이미지를 그린다. 샘플 개수를 10 으로 나누어 전체 행 개수를
계산한다.
    rows = int(np.ceil(n/10))
    # 행이 1 개 이면 열 개수는 샘플 개수이다. 그렇지 않으면 10 개이다.
    cols = n if rows < 2 else 10
    fig, axs = plt.subplots(rows, cols,
                                figsize=(cols*ratio, rows*ratio), squeeze=False)
    for i in range(rows):
        for j in range(cols):
            if i*10 + j < n:      # n 개까지만 그린다.
                axs[i, j].imshow(arr[i*10 + j], cmap='gray_r')
            axs[i, j].axis('off')
    plt.show()
```

- draw_fruits() 함수는 (샘플개수, 너비, 높이)의 3 차원 배열을 입력 받아 가로로 10 개씩 이미지를 출력한다.
- 샘플 개수에 따라 행과 열의 개수를 계산하고 figsize 를 지정한다. figsize 는 ratio 매개변수에 비례해 커진다.
- 그리고 이중 for 문을 사용해 먼저 첫 번째 행을 따라 이미지를 그린다. 이어서 두 번째를 그리는 식으로 진행한다.
- 해당 함수를 가지고 레이블이 0 인 과일 사진을 모두 그려보자. km.labels_==0 과 같이 쓰면 0 인 값을 참으로 반환해준다. 즉 km.labels_ 배열에서 값이 0 인 위치는 True, 그 외는 모두 False 가 된다. 넘파이는 이런 불리언 배열을 사용해 원소를 선택할 수 있다. 이를 **불리언 인덱싱**이라고 한다. 넘파이 배열에 불리언 인덱싱을 적용하면 True 인 위치의 원소만 모두 추출한다.

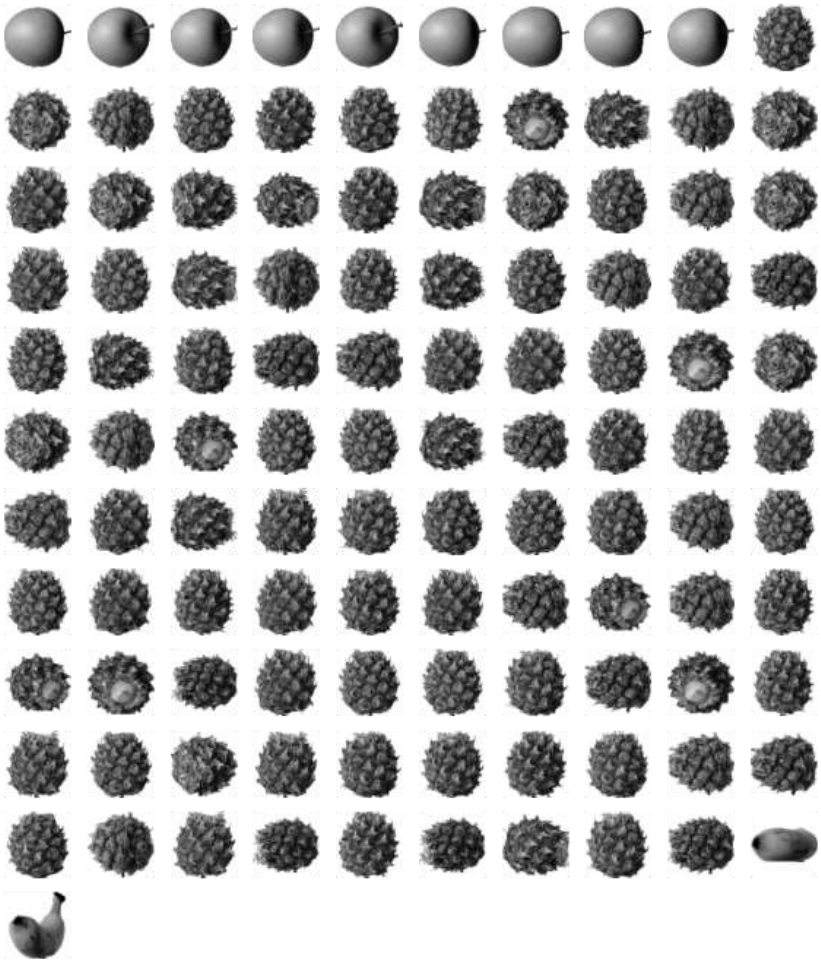
```
draw_fruits(fruits[km.labels_==0])
```



```
draw_fruits(fruits[km.labels_==1])
```



```
draw_fruits(fruits[km.labels_==2])
```



- 레이블 0 으로 클러스터링된 91 개의 이미지를 모두 출력하니 이 클러스터는 모두 사과가 올바르게 모였다. 레이블이 1 인 클러스터는 바나나로만 이루어져 있다. 하지만 레이블이 2 인 클러스터는 파인애플에 사과 9 개와 바나나 2 개가 섞여 있다. k-평균 알고리즘이 이 샘플들을 완벽하게 구별해내지는 못했다.
- 하지만 훈련 데이터에 타깃 레이블을 전혀 제공하지 않았음에도 스스로 비슷한 샘플들을 아주 잘 모은 것 같다.

클러스터 중심

- KMeans 클래스가 최종적으로 찾은 클러스터 중심은 cluster_centers_속성에 저장되어 있다. 이 배열은 fruits_2d 샘플의 클러스터 중심이기 때문에 이미지로 출력하려면 100x100 의 2 차원 배열로 변경해야 한다.

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

- 이전에 사과, 바나나, 파인애플의 픽셀 평균값을 출력했던 것과 매우 비슷하다.
- KMeans 클래스는 훈련 데이터 샘플에서 클러스터 중심까지 거리로 변환해주는 transform()메서드를 가지고 있다. transform()메서드가 있다는 것은 표준화를 진행해주는 StandardScaler 클래스처럼 특성값을 변환하는 도구로 사용할 수 있다는 것을 의미한다.
- 인덱스가 100 인 샘플에 transform()메서드를 적용해보자. fit() 메서드와 마찬가지로 2 차원 배열을 기대한다. fruits_2d[100] 처럼 쓰면 (10000,) 크기의 배열이 되므로 에러가 발생한다. 슬라이싱 연산자를 사용해서 (1,10000) 크기의 배열을 전달하겠다.

```
print(km.transform(fruits_2d[100:101]))

[[5267.70439881 8837.37750892 3393.8136117 ]]
```

- 하나의 샘플을 전달했기 때문에 반환된 배열은 크기가 (1, 클러스터개수)인 2 차원 배열이다. 첫 번째 클러스터(레이블 0), 두 번째 클러스터 (레이블 1)가 각각 첫 번째 원소, 두 번째 원소의 값이다. 세 번째 클러스터까지의 거리가 3393.8 로 가장 작다. 이 샘플은 레이블 2 에 속한 것 같다.
- (인덱스가 100 인 샘플의 각 클러스터까지의 거리를 알 수 있다)
- KMeans 클래스는 가장 가까운 클러스터 중심을 예측 클래스로 출력하는 predict() 메서드를 제공한다.

```
print(km.predict(fruits_2d[100:101]))

[2]
```

- transform()의 결과에서 짐작할 수 있듯이 레이블 2 로 예측했다. 클러스터 중심을 그려보았을 때 레이블 2 는 파인애플이었으므로 이 샘플은 파인애플일 것이다.
- 확인해보자

```
draw_fruits(fruits[100:101])
```

- k-평균 알고리즘은 반복적으로 클러스터 중심을 옮기면서 최적의 클러스터를 찾는다. 알고리즘이 반복한 횟수는 KMeans 클래스의 n_iter_속성에 저장된다.

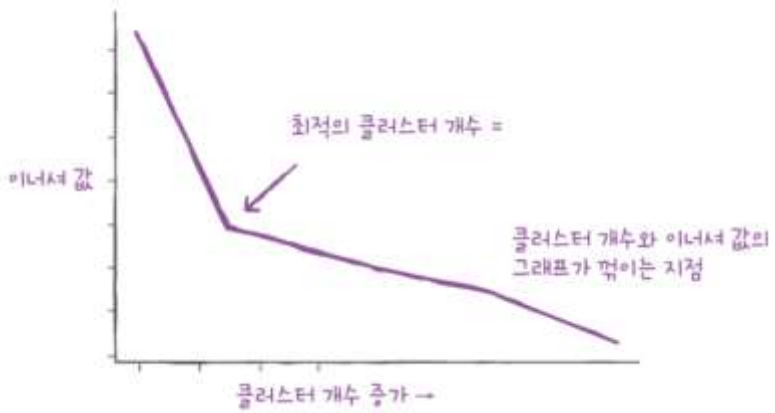
```
print(km.n_iter_)

3
```

- 클러스터 중심을 특성 공학처럼 사용해 데이터셋을 저차원(이 경우에는 10000 에서 3 으로 줄인다.)으로 변환할 수 있다. 또는 가장 가까운 거리에 있는 클러스터 중심을 샘플의 예측 값으로 사용할 수 있다.
- 우리는 타깃값을 사용하지 않았지만, 약간의 편법을 사용했다. n_cluster 를 3 으로 지정한 것은 타깃에 대한 정보를 활용한 셈이다. 하지만 실제로는 이런 클러스터 개수 조차도 알기 힘들다.
- n_cluster 를 어떻게 지정해야 할까? 최적의 클러스터 개수는 얼마일까?

2. 최적의 k 찾기

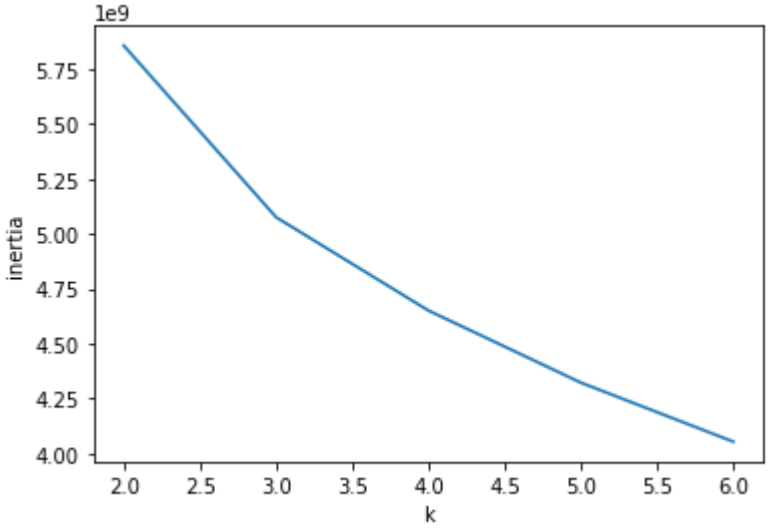
- k-평균 알고리즘은 클러스터 개수를 사전에 지정해야 한다. 실전에서는 클러스터의 개수가 몇 개인지 알 수 없다. 실제로 군집 알고리즘에서 적절한 k 값을 찾기 위한 완벽한 방법은 존재하지 않는다. 몇 가지 도구가 있지만 저마다 장단점이 있다. 적절한 클러스터 개수를 찾기 위한 대표적인 방법인 **엘보우 방법**에 대해 알아보겠다.
- k-평균 알고리즘은 클러스터 중심과 클러스터에 속한 샘플 사이의 거리를 잴 수 있다. 이 거리의 제곱 합을 **이너셔**라고 부른다. 이너셔는 클러스터에 속한 샘플이 얼마나 가깝게 모여 있는지를 나타내는 값이다. 일반적으로 클러스터 개수가 늘어나면 클러스터 개개의 크기는 줄어들기 때문에 이너셔도 줄어든다. 엘보우 방법은 클러스터 개수를 늘려가면서 이너셔의 변화를 관찰하여 최적의 클러스터 개수를 찾는 방법이다.
- 클러스터 개수를 증가시키면서 이너셔를 그래프로 그리면 감소하는 속도가 꺾이는 지점이 있다. 이 지점부터는 클러스터 개수를 증가시켜도 클러스터에 잘

밀집된 정도가 크게 개선되지 않는다. 즉 이너셔가 크게 줄어들지 않는다. 이 지점이 마치 팔꿈치 모양과 닮아서 이를 엘보우 방법이라 한다.



- 과일 데이터셋을 사용해 이너셔를 계산해보자. KMeans 클래스는 자동으로 이너셔를 계산해서 inertia_ 속성으로 제공함. 다음 코드에서 클러스터 개수 k 를 2~6 까지 바꿔가며 KMeans 클래스를 5 번 훈련한다. fit() 메서드로 모델을 훈련한 후 inertia_ 속성에 저장된 이너셔값을 inertia 리스트에 추가한다. 마지막으로 inertia 리스트에 저장된 값을 그래프로 출력한다.

```
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.xlabel('k')
plt.ylabel('inertia')
plt.show()
```



이 그래프에서는 꺾이는 지점이 뚜렷하지는 않지만, k=3 에서 그래프의 기울기가 조금 바뀐 것을 볼 수 있다. 엘보우 지점보다 클러스터 개수가 많아지면 이너셔의 변화가 줄어들면서 군집 효과도 줄어든다. 하지만 이 그래프에서는 이런 지점이 명확하지는 않다.

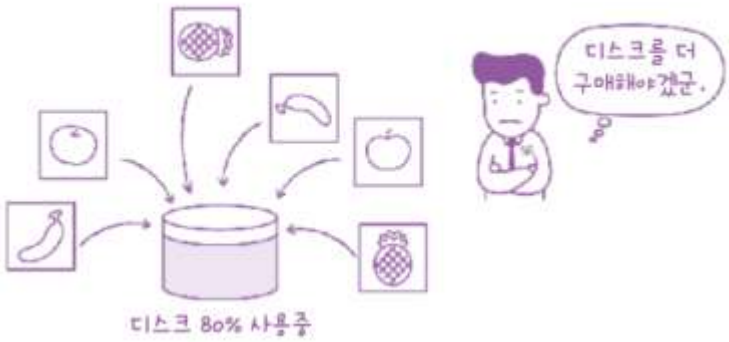
과일을 자동으로 분류하기 문제해결 과정

- 이전에는 과일 종류별로 픽셀 평균값을 계산했다. 하지만 실전에서는 어떤 과일 사진이 들어올지 모른다. 따라서 타깃값을 모르는 척하고 자동으로 사진을 클러스터로 모을 수 있는 군집 알고리즘이 필요하다.
- 대표적인 군집 알고리즘인 k-평균 알고리즘을 사용했다. k-평균은 비교적 간단하고 속도가 빠르며 이해하기도 쉽다. k-평균 알고리즘을 구현한 사이킷런의 KMeans 클래스는 각 샘플이 어떤 클러스터에 소속되어 있는지 labels_ 속성에 저장한다.
- 각 샘플에서 각 클러스터까지의 거리를 하나의 특성으로 활용할 수도 있다. 이를 위해 KMeans 클래스는 transform() 메서드를 제공한다. 또한 predict() 메서드에서 새로운 샘플에 대해 가장 가까운 클러스터를 예측값으로 출력한다.
- k-평균 알고리즘은 사전에 클러스터 개수를 미리 지정해야 한다. 사실 데이터를 직접 확인하지 않고서는 몇 개의 클러스터가 만들어질지 알기 어렵다. 최적의 클러스터 개수 k 를 알아내는 한 가지 방법은 클러스터가 얼마나 밀집되어 있는지 나타내는 이너셔를 사용하는 것이다. 이너셔가 더 이상 크게 줄어들지 않는다면 클러스터 개수를 더 늘리는 것은 효과가 없다. 이를 엘보우 방법이라고 부른다.

- 사이킷런의 KMeans 클래스는 자동으로 이너셔를 계산하여 inertia_ 속성으로 제공한다. 클러스터 개수를 늘리면서 반복하여 KMeans 알고리즘을 훈련하고 이너셔가 줄어드는 속도가 꺾이는 지점을 최적의 클러스터 개수로 결정한다.
- 이번에는 k-평균 알고리즘의 클러스터 중심까지 거리를 특성으로 사용할 수도 있다는 점을 보았다. 이렇게 하면 훈련 데이터의 차원을 크게 줄일 수 있다. 데이터셋의 차원을 줄이면 지도학습 알고리즘의 속도를 크게 높일 수 있다.

[주성분 분석]

과일사진 이벤트도 잘 진행되고 있어서 매일 각양각색의 과일 사진이 업로드되고 있다. k-평균 알고리즘으로 업로드된 사진을 클러스터로 분류하여 폴더별로 저장했다. 그런데 이벤트가 진행되면서 문제가 생겼다. 너무 많은 사진이 등록되어 저장공간이 부족하다. 나중에 군집이나 분류에 영향을 끼치지 않으면서 업로드된 사진의 용량을 줄일 수 있을까?



1. 차원과 차원축소

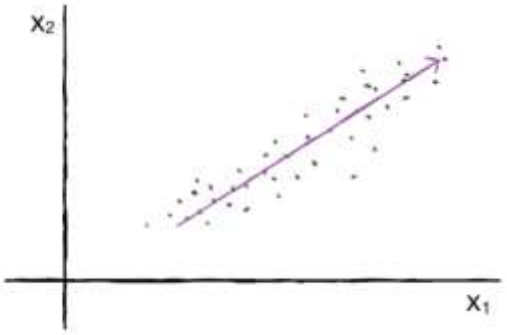
- 데이터가 가진 속성을 특성이라고 한다. 이전에 다룬 과일 사진의 경우 10000 개의 픽셀이 있기 때문에 10000 개의 특성을 가지고 있다고 볼 수 있다. 머신러닝에서는 이런 특성을 **차원**이라고도 부른다. 즉 10000 개의 특성은 10000 개의 차원이라고도 볼 수 있다.
- 이런 차원을 줄일 수 있다면 저장 공간을 크게 절약할 수 있다. 이를 위해 비지도 학습 작업 중 하나인 **차원축소**알고리즘을 이용할 수 있다.

- 이전에 특성이 많으면 선형 모델의 성능이 높아지고 훈련 데이터에 쉽게 과대적합된다는 것을 배웠다. 차원 축소는 데이터를 가장 잘 나타내는 일부 특성을 선택해 데이터 크기를 줄임과 동시에 지도 학습 모델의 성능을 향상시키는 방법이다.
- 또한 줄어든 차원에서 다시 원본 차원으로 손실을 최대한 줄이면서 복원도 가능하다. 대표적인 차원 축소 알고리즘인 **주성분 분석**을 알아보자. 주성분 분석은 **PCA** 라고도 한다.

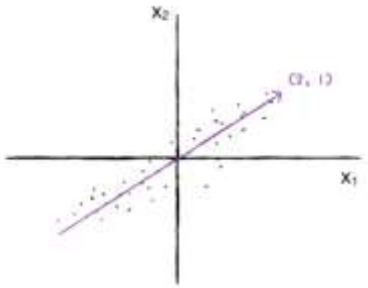
2. 주성분 분석

- 주성분 분석은 데이터에 있는 분산이 큰 방향을 찾는 것으로 이해할 수 있다.
- 분산은 데이터가 널리 퍼져있는 정도를 말한다. 분산이 큰 방향이란 데이터를 잘 표현하는 어떤 벡터라고 생각할 수 있다.
- 이해하기 쉽도록 다음과 같은 2 차원 데이터를 생각해보자

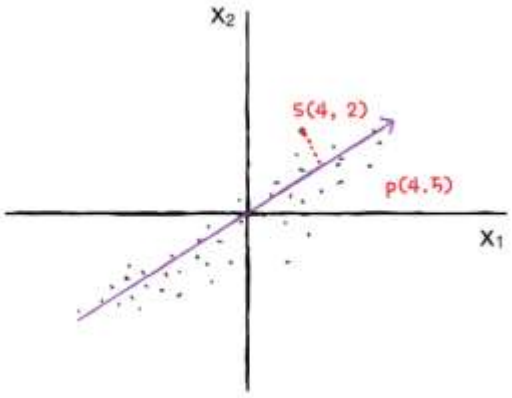
- 이 데이터는 x1, x2 2 개의 특성이 있다. 대각선 방향으로 길게 늘어진 형태를 가지고 있다. 이 데이터에서 가장 분산이 큰 방향(**가장 데이터의 분포를 잘 표현**하는 방향)을 찾아보자.



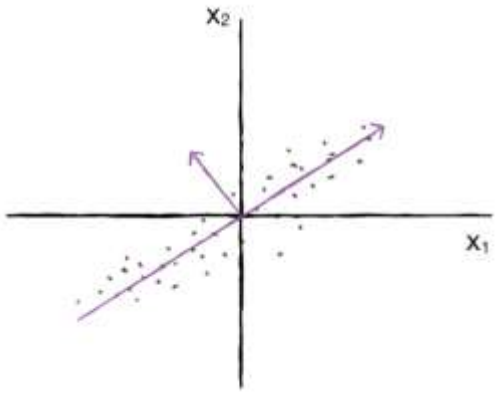
- 직관적으로 우리는 길게 늘어진 대각선 방향이 분산이 가장 크다고 알 수 있다. 이 그림에서 화살표 위치는 큰 의미가 없다. 오른쪽 위로 향하거나 왼쪽 아래로 향할 수도 있다. 중요한 것은 분산이 큰 방향을 찾는 것이다.
- 여기서 찾은 직선이 원점에서 출발한다면 두 원소로 이루어진 벡터로 쓸 수 있다. 예를 들어 다음 그림의 (2,1)처럼 나타낼 수 있다.



- 이 벡터를 주성분이라고 부른다. 이 주성분 벡터는 원본 데이터에 있는 어떤 방향이다. 따라서 주성분 벡터의 원소개수는 원본 데이터셋에 있는 특성 개수와 같다. 하지만 **원본 데이터는 주성분을 사용해 차원을 줄일 수 있다**. 예를 들면 다음과 같이 샘플 데이터 s(4,2)를 주성분에 직각으로 투영하면 1 차원 데이터 p(4.5)를 만들 수 있다.



- 주성분은 원본 차원과 같고 주성분으로 바꾼 데이터는 차원이 줄어든다!!
- 주성분이 가장 분산이 큰 방향이기 때문에 주성분에 투영하여 바꾼 데이터는 원본이 가지고 있는 특성을 가장 잘 나타내고 있을 것이다.
- **첫 번째 주성분을 찾은 다음 이 벡터에 수직이고 분산이 가장 큰 다음 방향을 찾는다. 이 벡터가 두 번째 주성분**이다. 여기서는 2 차원이기 때문에 두 번째 주성분의 방향은 다음처럼 하나뿐이다.



- 일반적으로 주성분은 원본 특성의 개수만큼 찾을 수 있다.
- 사이킷런으로 과일사진 데이터에서 주성분 분석을 수행해보자. 데이터는 이전에 이용했던 데이터를 사용한다. (과일사진 데이터를 다운로드하여 넘파이 배열로 적재한다)

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_validate
from sklearn.cluster import KMeans
fruits = np.load('fruits_300.npy')
fruits_2d = fruits.reshape(-1, 100*100)

pca = PCA(n_components=50)
pca.fit(fruits_2d)
```

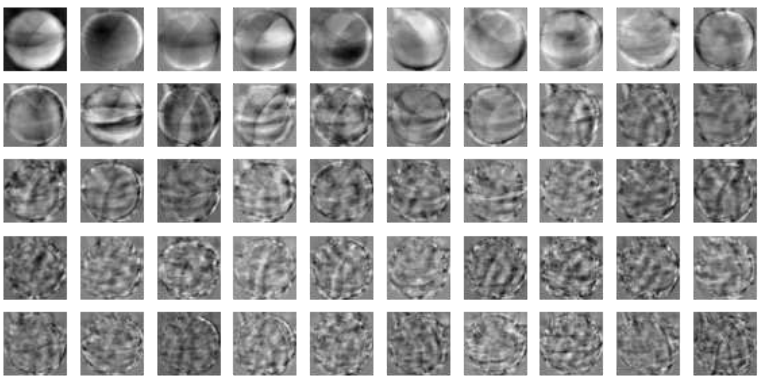
- 사이킷런은 sklearn.decompositon 모듈 아래 PCA 클래스로 주성분 분석 알고리즘을 제공한다. PCA 클래스의 객체를 만들 때 n_components 매개변수에 주성분의 개수를 지정해야 한다. k-평균과 마찬가지로 비지도학습이기 때문에 fit() 메서드에 타깃값을 제공하지 않는다.
- PCA 클래스가 찾은 주성분은 components_ 속성에 저장되어 있다. 이 배열의 크기를 확인하자

```
print(pca.components_.shape)

(50, 10000)
```

- 주성분의 개수(n_components)를 50 개로 지정했기 때문에 pca.components_ 배열의 첫 번째 차원이 50 이다. 즉 50 개의 주성분을 찾은 것이다. 두 번째 차원은 원본 데이터의 특성 개수와 같은 10,000 입니다.

- 원본 데이터와 차원이 같기 때문에 주성분을 100x100 크기의 이미지처럼 출력해볼 수 있다.
- 이전에 사용했던 draw_fruits()함수를 사용해서 이 주성분을 그림으로 그려보자.

```
def draw_fruits(arr, ratio=1):
    n = len(arr) # 샘플의 개수
    # 한 줄에 10 개씩 이미지를 그린다. 샘플개수를 10 으로 나누어 전체 행 개수를 계산한다.
    rows = int(np.ceil(n/10))
    # 행이 1 개이면 열의 개수는 샘플 개수이다. 그렇지 않다면 10 개이다.
    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:
                axs[i,j].imshow(arr[i*10 + j], cmap='gray_r')
            axs[i, j].axis('off')
    plt.show()

draw_fruits(pca.components_.reshape(-1, 100, 100))
```



- 이 주성분은 원본 데이터에서 가장 분산이 큰 방향을 순서대로 나타낸 것이다. 데이터셋에 있는 어떤 특징을 잡아낸 것처럼 생각할 수도 있다.

- 주성분을 찾았기 때문에 원본 데이터를 주성분에 투영해 특성의 개수를 10000 개에서 50 개로 줄일 수 있다. 이는 마치 원본 데이터를 각 주성분으로 분해하는 것으로 생각할 수 있다
- PCA 의 transform() 메서드를 이용해 원본 데이터의 차원을 50 으로 줄여보자

```
print(fruits_2d.shape)

(300, 10000)

fruits_pca = pca.transform(fruits_2d)
print(fruits_pca.shape)

(300, 50)
```

- fruits_2d 는 (300, 10000)크기의 배열이었다. 10000 개의 픽셀(특성)을 가지는 300 개의 이미지였다. 50 개의 주성분을 찾은 PCA 모델을 사용해 이를 (300, 50) 크기의 배열로 변환했다. 이제 fruits_pca 배열은 50 개의 특성을 가진 데이터이다.
- 데이터의 차원을 줄였으니 용량도 줄어들었다. 그렇다면 차원을 줄이는 것과 반대로 원상복구도 가능할까? 이를 직접 해보도록 하자.

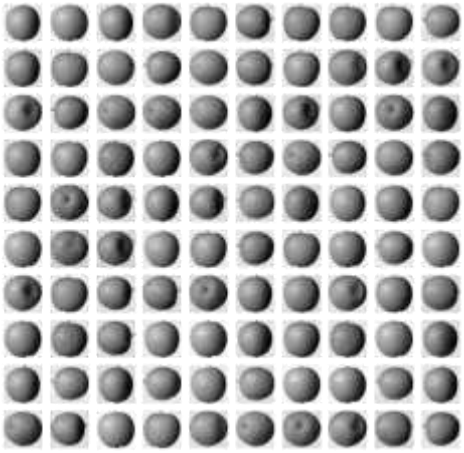
3. 원본 데이터의 재구성

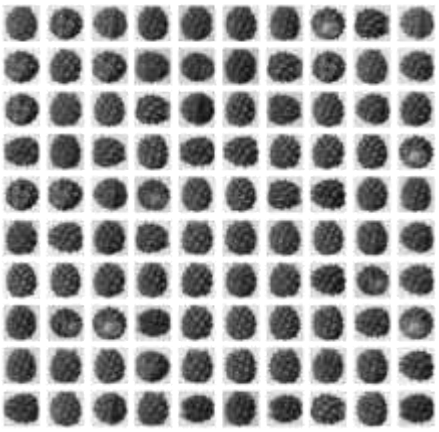
- 이전까지 10000 개의 특성을 50 개로 줄였다. 이로 인한 손실은 발생할 수밖에 없다. 하지만 최대한 분산이 큰 방향으로 데이터를 투영했기 때문에 원본 데이터를 상당부분 재구성할 수 있다.
- 이를 위해 PCA 는 inverse_transform() 메서드를 제공한다. 앞서 50 개의 차원으로 축소한 fruits_pca 데이터를 전달해 10000 개의 특성을 복원하겠다.

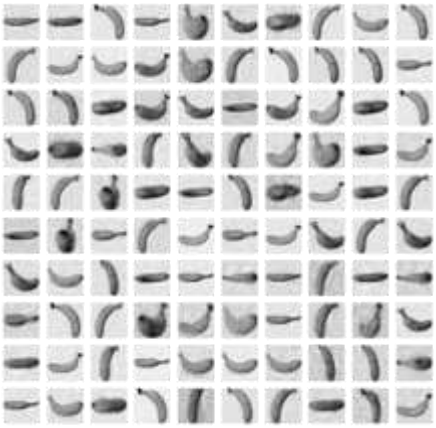
```
fruits_inverse = pca.inverse_transform(fruits_pca)
print(fruits_inverse.shape)

(300, 10000)
```

- 10000 개의 특성이 복원되었다. 이 데이터를 100x100 크기로 바꾸어 100 개씩 나누어 출력해보자. 이 데이터는 순서대로 사과, 파인애플, 바나나를 100 개씩 담고 있다.

```
fruits_reconstruct = fruits_inverse.reshape(-1, 100, 100)
for start in [0, 100, 200]:
    draw_fruits(fruits_reconstruct[start:start+100])
    print('₩n')
```







- 거의 모든 과일이 잘 복원되었다. 일부는 흐리고 번진 부분이 존재하지만 50 개의 특성을 10000 개로 늘린 것을 감안하면 놀라운 일이다. 이 50 개의 특성이 분산을 가장 잘 보존하도록 변환된 것이기 때문이다.
- 만약 주성분을 최대로 사용했다면 보다 완벽하게 재구성이 가능했을 것이다.
  그렇다면 50 개의 특성(주성분)은 얼마나 분산을 보존하고 있던 것인지 알아보자.

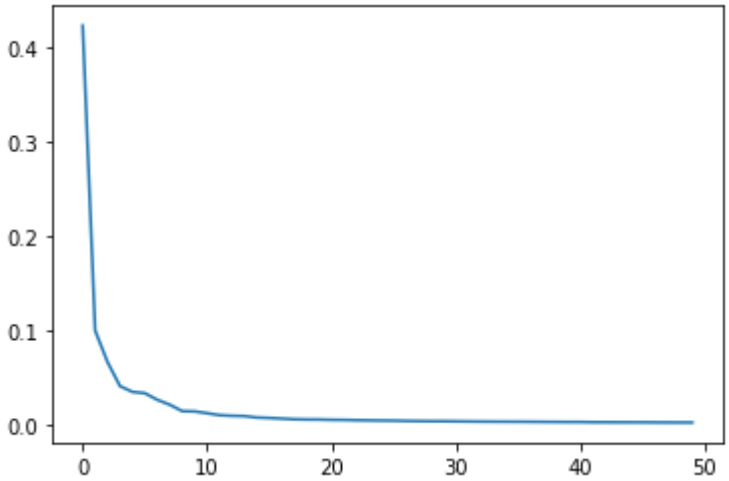
4. 설명된 분산

- 주성분이 원본 데이터의 분산을 얼마나 잘 나타내는지 기록한 값을 **설명된 분산**이라고 한다.
- PCA 클래스의 explained_variance_ratio_에 각 주성분의 설명된 분산 비율이 기록되어 있다. 첫 번째 주성분의 설명된 분산이 가장 크다. 이 분산비율을 모두 더하면 50 개의 주성분으로 표현하고 있는 총 분산 비율을 얻을 수 있다.

```
print(np.sum(pca.explained_variance_ratio_))

0.9215175552398827
```

- 92%가 넘는 분산을 유지하고 있다. 앞에서 50 개의 특성에서 원본 데이터를 복원했을 때 원본 이미지의 품질이 높았던 이유가 여기 있었다. 설명된 분산의 비율을 그래프로 그려보면 적절한 주성분의 개수를 찾는데 도움이 된다.
  맷플롯립의 plot()함수로 설명된 분산을 그래프로 출력해보자.

```
plt.plot(pca.explained_variance_ratio_)
plt.show()
```



- 그래프를 보면 처음 10 개의 주성분이 대부분의 분산을 표현하고 있다. 그 다음부터의 각 주성분이 설명하는 분산은 비교적 작다.
- 이번에는 PCA 로 차원 축소된 데이터를 사용해 지도학습 모델을 훈련해보도록 하자. 그리고 원본데이터를 사용했을 때와 비교를 해보겠다.

5. 다른 알고리즘과 함께 사용하기

```
lr = LogisticRegression()
```

- 과일 사진 원본 데이터와 PCA 로 축소한 데이터를 지도 학습에 적용해보고 어떤 차이가 있는지 알아보자. 3 개의 과일 사진을 분류해야 하므로 간단히 로지스틱 회귀 모델을 사용하자.
- 지도 학습 모델을 사용하기 위해서는 타깃값이 필요하다. 여기서는 사과를 0, 파인애플을 1, 바나나를 2 로 지정한다. 파이썬 리스트와 정수를 곱하면 리스트 안의 원소를 정수만큼 반복한다. 이를 이용하면 100 개의 0, 100 개의 1, 100 개의 2 로 이루어진 타깃 데이터를 만들 수 있다.

```
target = np.array([0]*100 + [1]*100 + [2]*100)
```

- 먼저 원본 데이터인 fruits_2d 를 사용해보자. 로지스틱 회귀 모델에서 성능을 가늠해 보기 위해 cross_validate()로 교차검증을 수행한다.

```
scores = cross_validate(lr, fruits_2d, target)
print(np.mean(scores['test_score']))
print(np.mean(scores['fit_time']))
```

```
0.9966666666666667
0.9422160625457764
```

- 교차 검증의 점수는 0.997 정도로 매우 높게 나타난다. 특성이 10000 개나 되기 때문에 300 개의 샘플에서는 금방 과대적합된 모델이 나오기 쉽다. cross_validate() 함수가 반환하는 딕셔너리에는 fit_time 항목에 각 교차 검증 폴드의 훈련시간이 기록되어있다. 0.94 초 걸렸다고 나와있다. 이 값을 PCA 로 축소한 fruits_pca 를 사용했을 때와 비교해보도록 하겠다.

```
scores = cross_validate(lr, fruits_pca, target)
print(np.mean(scores['test_score']))
print(np.mean(scores['fit_time']))
```

```
1.0
0.03256878852844238
```

- 50 개의 특성만 사용했는데도 정확도가 100%이며 훈련시간은 0.03 초로 20 배 이상 감소했다. **PCA 로 훈련 데이터의 차원을 축소하면 저장 공간 뿐 아니라 머신러닝 모델의 훈련속도도 높일 수 있**다.
- 앞에서 PCA 클래스를 사용할 때 n_components 매개변수에 주성분의 개수를 지정했다. 대신 원하는 설명된 분산의 비율을 입력할 수도 있다. PCA 클래스는 지정된 비율에 도달할 때까지 자동으로 주성분을 찾는다. 설명된 분산의 50%에 달하는 주성분을 찾도록 PCA 모델을 만들어보자.

```
pca = PCA(n_components=0.5)
pca.fit(fruits_2d)

# 몇 개의 주성분을 찾았는지 확인
print(pca.n_components_)

2
```

- 주성분 개수 대신 0~1 사이의 비율을 실수로 입력하면 된다.
- 2 개의 특성만으로도 원본 데이터에 있는 분산의 50%를 표현할 수 있다.
- 이 모델로 원본 데이터를 변환해보자. 주성분이 2 개 이므로 변환된 데이터의 크기는 (300, 2)가 될 것이다.

- 2 개의 특성만 사용하고도 교차검증의 결과가 좋을지도 살펴보겠다.

```
fruits_pca = pca.transform(fruits_2d)
print(fruits_pca.shape)
(300, 2)

scores = cross_validate(lr, fruits_pca, target)
print(np.mean(scores['test_score']))
print(np.mean(scores['fit_time']))
```

```
0.9933333333333334
0.04122166633605957
```

note 앞의 코드를 입력하면 로지스틱 회귀 모델이 완전히 수렴하지 못했으니 반복 횟수를 증가하라는 경고(Convergence Warning: lbfgs failed to converge)가 출력됩니다. 하지만 교차 검증의 결과가 충분히 좋기 때문에 무시해도 좋습니다.

- 결과를 보게 되면 단 2 개의 특성으로도 99%의 정확도를 보여준다.
- 이번에는 차원축소된 데이터를 이용해 k-평균 알고리즘으로 클러스터를 찾아보자.

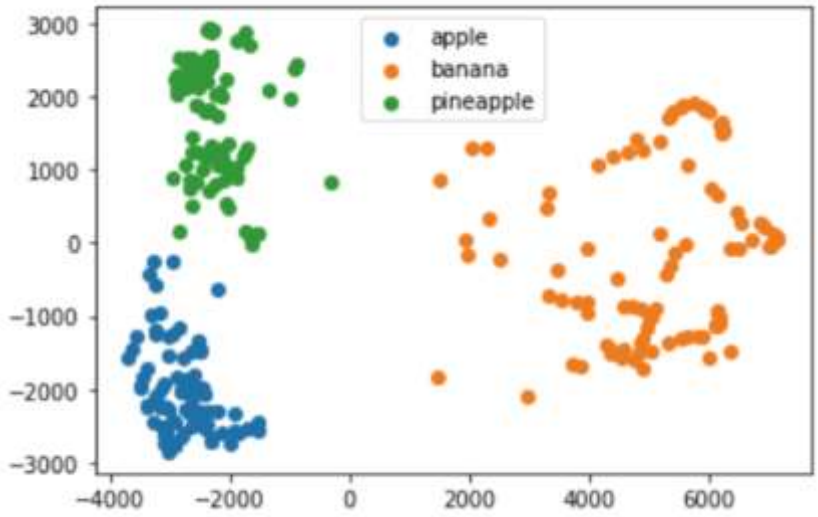
```
km = KMeans(n_clusters=3, random_state=42)
km.fit(fruits_pca)
print(np.unique(km.labels_, return_counts=True))

#결과
(array([0, 1, 2], dtype=int32), array([91, 99, 110]))
```

- fruits_pca 로 찾은 클러스터는 각각 91 개, 99 개, 110 개의 샘플을 포함하고 있다. 이는 이전에 원본 데이터를 사용했을 때와 거의 비슷한 결과이다.
- KMeans 가 찾은 레이블을 사용해 과일 이미지를 출력하겠다.

```
for label in range(0,3):
    draw_fruits(fruits[km.labels_ == label])
    print('\n')
```

- 이전에 찾은 클러스터와 비슷하게 파인애플은 사과와 조금 혼돈되는 면이 있다. 몇 개의 사과가 파인애플 클러스터에 섞여 들어가 있다.
- 훈련 데이터의 차원을 줄이면 시각화에 용이하다는 장점을 얻을 수 있다. 3 개 이하로 차원을 줄이면 화면에 출력하기 비교적 쉽다. fruits_pca 데이터는 2 개의 특성이 있기 때문에 2 차원으로 표현이 가능하다. 앞에서 찾은 km.labels_를 사용해 클러스터별로 나누어 산점도를 그려보자.

```
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()
```



- 각 클러스터의 산점도가 아주 잘 구분된다. 2 개의 특성만으로도 로지스틱 회귀 모델의 교차검증 점수가 99%에 달하는 결과라고 할 수 있다. 위의 그림을 보면 파인애플과 사과의 클러스터의 경계가 가깝게 붙어 있다. 그래서 둘이 혼동하는 결과가 생긴 것이 아닌가 추측할 수 있다.

- 데이터를 시각화하면 예상치 못한 통찰을 얻을 수 있다. 그런 면에서 차원 축소는 매우 유용한 도구 중 하나이다.

주성분 분석으로 차원 축소 문제해결 과정

대표적인 비지도 학습 문제 중 하나인 차원축소에 대해 알아보았다. 차원 축소를 사용하면 데이터셋의 크기를 줄일 수 있고 비교적 시각화하기 쉽다. 또 차원 축소된 데이터를 지도 학습 알고리즘이나 다른 비지도 학습 알고리즘에 재사용하여 성능을 높이거나 훈련 속도를 빠르게 만들 수 있다.

사이킷런의 PCA 클래스를 사용해 과일 사진 데이터의 특성을 50 개로 크게 줄였다. 특성 개수는 작지만 변환된 데이터는 원본 데이터에 있는 분산의 90% 이상을 표현한다. 이를 **설명된 분산**이라 부른다.

PCA 클래스는 자동으로 설명된 분산을 계산하여 제공해 준다. 또한 주성분의 개수를 명시적으로 지정하는 대신 설명된 분산의 비율을 설정하여 원하는 비율만큼 주성분을 찾을 수 있다.

PCA 클래스는 변환된 데이터에서 원본 데이터를 복원하는 메서드도 제공한다. 변환된 데이터가 원본 데이터의 분산을 모두 유지하고 있지 않다면 완벽하게 복원되지 않는다. 하지만 적은 특성으로도 상당부분의 디테일을 복원할 수 있다.