[7주차]
Ch8 차원축소

1기 이다현
1기 최하경

EURON

목차
(선택)

1. 차원의 저주

야외활동에 적합한 날씨인지 (good=1, bad=0) 분류하는 머신러닝 모델을 만들어 보자

가령 101개의 야외활동과 관련된 항목들과 그 수치가 나열된 데이터 테이블이 있다고 한다면

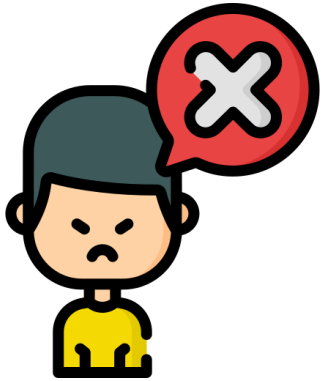
	온도	습도	강수량	미세먼지	풍속	태풍여부	...	교통량	유동인구
0801	32	15	66	3	111	0		15	40326127
0802									
...									
0831									

풍속, 온도, 습도, 미세먼지, 강수량… 많은 요인이 존재

위의 주어진 데이터를 그대로 학습 모델에 입력하면
총 101차원의 벡터가 됨

1. 차원의 저주

101개의 특성을 그냥 학습하게 되면……

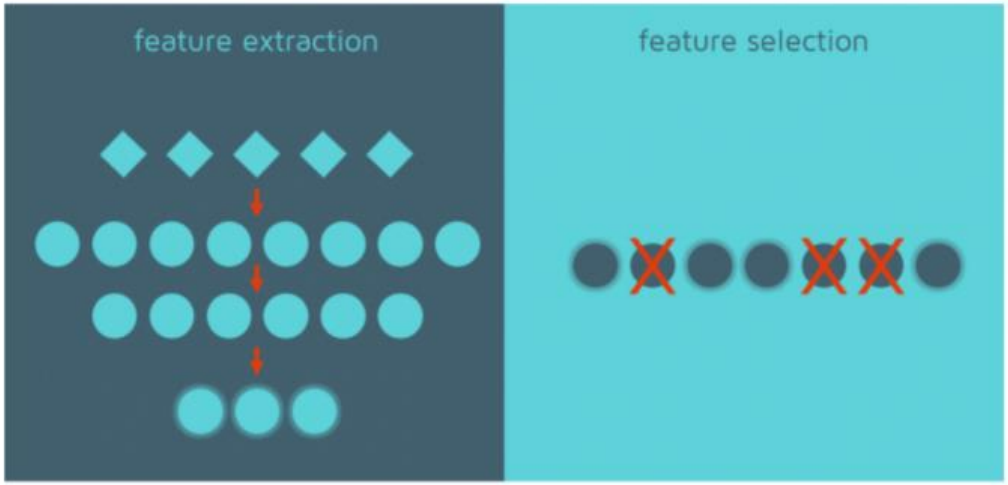


▷ 문제1) 훈련속도가 느려진다.
▷ 문제2) 시각화가 어렵다. 우리가 인식할 수 있는 범위는 3차원까지 뿐, 101차원은 데이터 패턴을 우리가 쉽게 인지하기 어렵다.
▷ 문제3) 쓸모없는 특성을 학습하면서 노이즈가 섞일 수도 있다.
▷ 문제4) 큰 차원을 커버 할 만한 매우 많은 데이터가 수집되어야 한다.
▷ 문제5) 몇몇 특성들끼리 강한 상관관계를 보이는 경우로 다중 공산성 문제가 발생할 수 있다. 서로 의존성이 높은 속성을 함께 학습하면 모델의 과적합이 발생해 학습성능이 저하된다.

매우 많은 양의 데이터를 모으는 건 어려우니까 어떤 속성이 모델의 성능향상에 도움이 될지 파악하고 속성을 선택/가공하는 [차원축소] 과정을 거치자!

1. 차원의 저주

1. 피처 선택 : 데이터의 특징을 잘 나타내는 주요 속성만 선택. 상관계수 값을 통해 판단한다.
2. **피처 추출** : 기존 속성을 저차원의 중요 속성으로 '압축 ' 해 추출하는 것(기존 속성과는 완전히 다른 값이 된다. PCA, LDA,SVD,NMF와 같은 차원축소 방법을 통해 구현

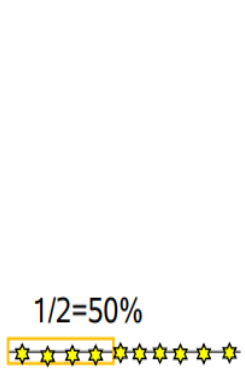


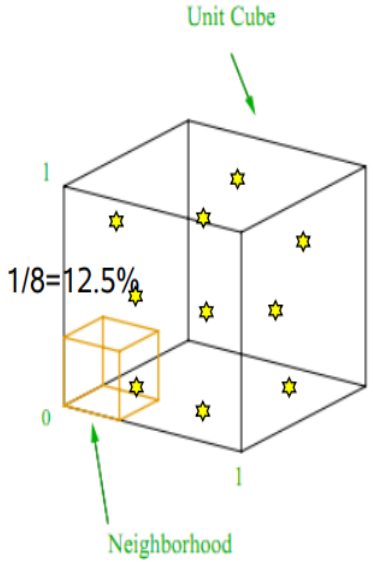
피처 선택과 추출의 차이

[피처추출 ex] 학생 평가 요소로 모의고사 성적, 내신 성적, 수능성적, 봉사, 대외활동을 학업 성취도와 같은 함축적 요약 특성으로 추출 가능

차원의 저주

차원이 늘어남에 따라(=변수 x 의 개수가 증가할 수록) 같은 영역의 자료를 가지고 있더라도, 전체 영역대비 우리가 설명할 수 있는 데이터의 패턴은 줄어들게 된다. 즉 대부분의 훈련 데이터가 서로 멀리 떨어져 있고 따라서 예측이 불안정해진다.



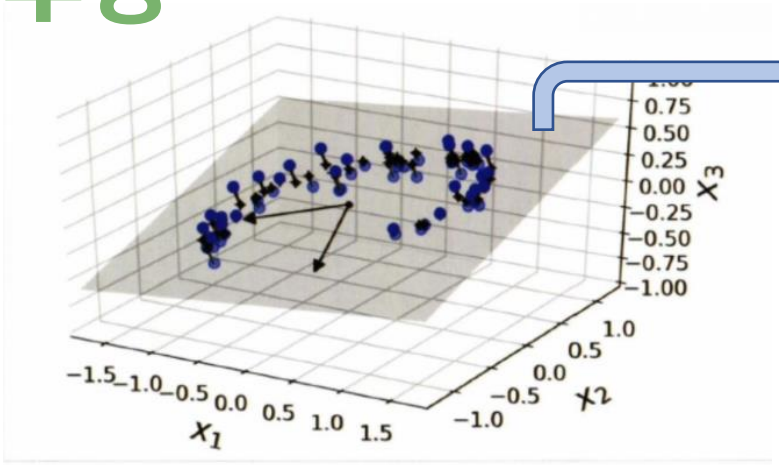


데이터에서 모델을 학습할 때 독립적인 샘플이 많을수록 학습이 잘 되는 반면, 차원이 커질수록 학습이 어려워지고 더 많은 데이터를 필요로 하는 현상을 말한다.

즉, **"관측치 수 < 변수의 수"**일 때 발생하는 현상이다.

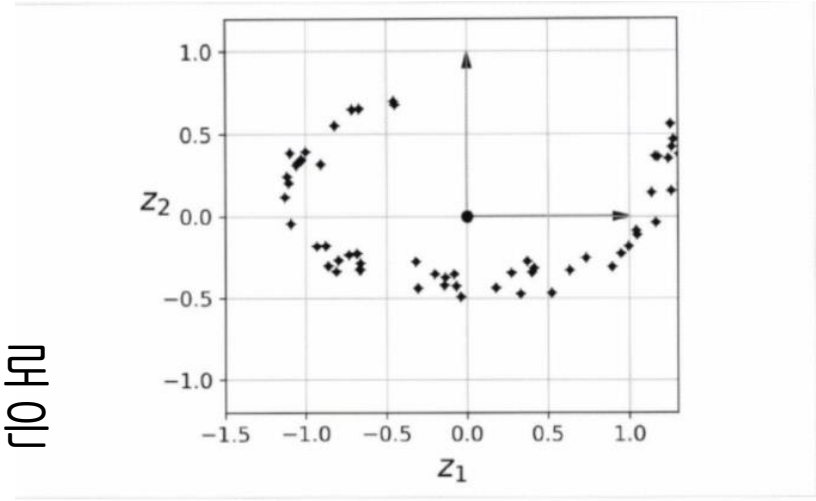
2. 차원축소 접근방법

투영



2차원에 가깝게 배치된 3차원 데이터 셋

3차원 공간이지만, 대부분의 훈련 샘플은 회색 평면 위에 위치한다.



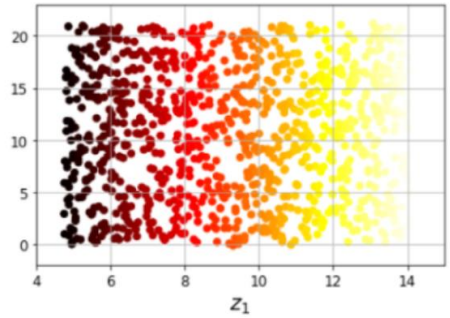
투영(projection)해서 만들어진 새로운 2차원 데이터 셋

따라서 3차원을 2차원으로 수직으로 정사영(투영) 시키면 오른쪽과 같은 2차원 데이터 셋을 얻을 수 있다.

*정사영 : 어떤 선이나 물체 위에서 빛이 비춰졌을 때 아래 평면에 그림자가 생기는 것

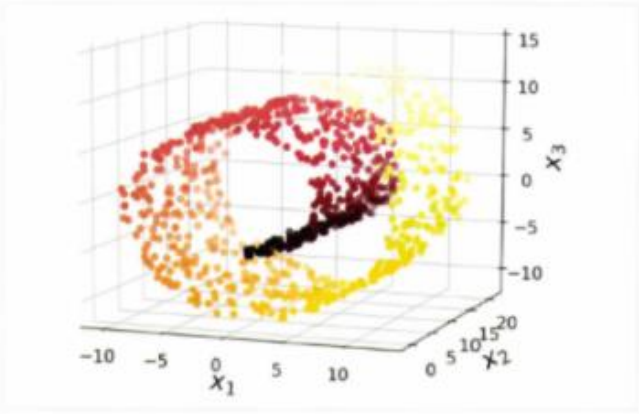
2. 차원축소 접근방법

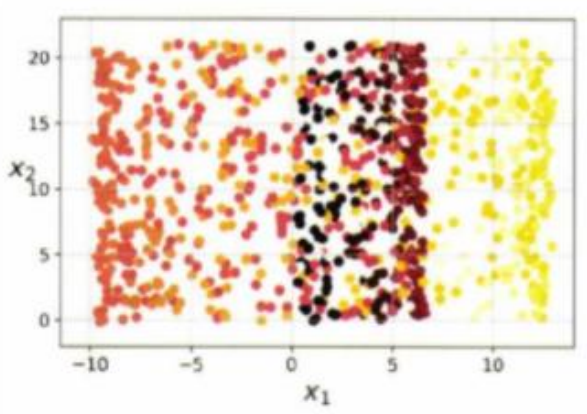
(얻고 싶었던 데이터 결과)



그러나 투영이 항상 최선은 아니다!

- 아래와 같은 **"스위스 롤" 형태의 데이터 셋**은 부분 공간이 뒤틀려있기 때문에, 투영(projection)을 통해 차원을 축소시키는 방법은 **좋지 못한 결과를 초래**한다.

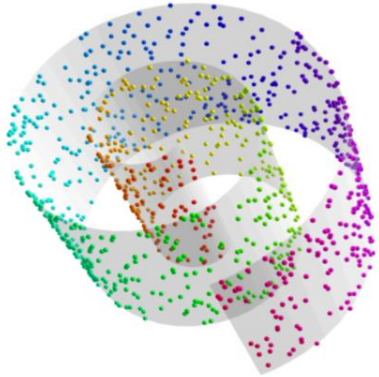




스위스 롤 형태의 데이터 셋(왼쪽)과 투영(projection) 이후의 뭉개진 2차원 데이터 셋(오른쪽)

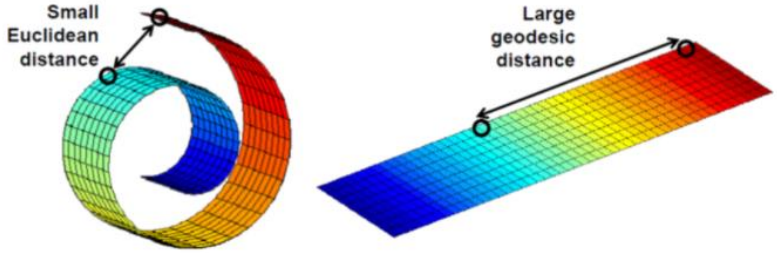
2. 차원축소 접근방법

매니폴드 학습



고차원 데이터를 데이터 공간에 뿌리면 샘플들을 잘 아우르는 subspace가 있을 것이라는 가정에서 시작

: 스위스 롤의 경우, 3차원 공간에 분포한 데이터를 아우르는 소용돌이 모양의 구부러진 평면을 2차원 매니폴드라고 한다. 이런 매니폴드를 찾아 2차원 평면에 데이터 포인트를 매핑하면 된다. 일반적으로 d차원 매니폴드는 전체 중 일부분이 d차원 초평면으로 보일 수 있는 n차원 공간의 일부이다. 많은 차원축소 알고리즘이 훈련샘플이 놓여있는 매니폴드를 모델링 하는 식으로 작동한다.

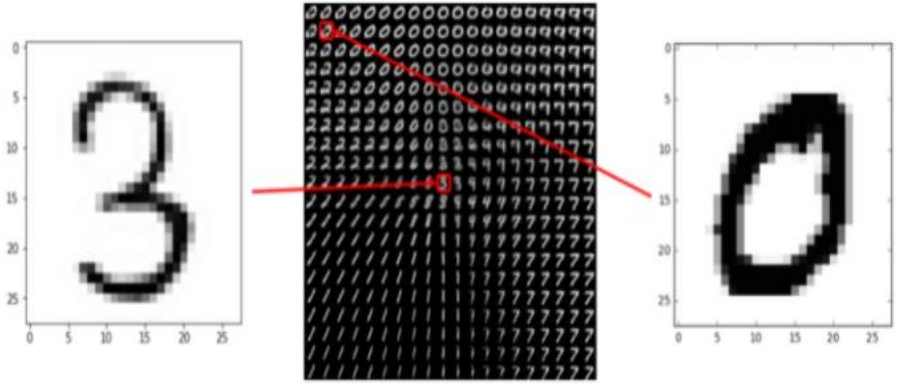


3차원 공간상의 2차원 매니폴드를 2차원 공간에서 표현했을 때 각 점 사이의 최단 경로. 공간에 따라 최단 경로가 바뀌는 것을 볼 수 있습니다.

고차원상에서 가까운 거리에 있던 데이터 포인트들일지라도, 매니폴드를 보다 저차원 공간으로 맵핑하면 오히려 거리가 멀어질 수 있다. 그리고 저차원의 공간상에서 가까운 점끼리는 실제로도 비슷한 특징feature을 갖는다. 즉, 저차원의 각 공간의 차원 축은 고차원에서 비선형적으로 표현될 것이며, 데이터의 특징을 각각 표현하게 된다.

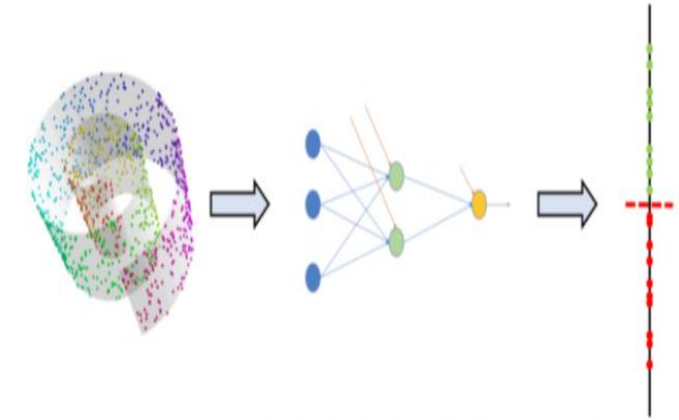
2. 차원축소 접근방법

예를 들어 다음 그림과 같이 MNIST 데이터를 2차원의 숨겨진 저차원low-dimensional latent space에 표현한다고 가정합니다. 빨간색으로 표시된 각 샘플은 2차원 공간에서는 사람이 인지하는 특징과 비슷한 특징을 갖는 위치와 관계에 있겠지만, 원래의 데이터 차원인 784차원의 고차원 공간에서는 전혀 다른 거리와 관계를 지닐 것입니다.



MNIST 데이터를 2차원 공간에 표현했을 때 두 샘플의 위치

매니폴드 학습은 딥러닝에서 훌륭한 성능을 내는데, 매니폴드는 비선형적인 방식으로 차원축소를 수행하기 때문에, 복잡한 데이터셋을 다루는 딥러닝은 매니폴드를 자연스럽게 찾아낼 수 있다.



3차원 데이터를 입력으로 받아 1차원의 이진 분류를 수행할 때

https://kh-kim.gitbook.io/natural-language-processing-with-pytorch/00-cover-5/02-dimension-reduction

3. PCA : 가장 대표적인 차원축소 알고리즘

ㅇ **초평면** : 평면에서 더 확장된 개념으로, 직선은 2차원의 초평면, 평면은 3차원의 초평면 이라고 한다. 즉 n차원에 그려진 초평면은 n-1 차원 공간과 같다. 즉 그려진 공간의 차원보다 한 차원 낮은 공간을 의미한다. 초평면은 주로 공간을 분할하는 역할에 쓰이며, (직선이 평면을 분할하는 것처럼) 이런 원리는 분류 머신러닝 모델을 만들 때 사용된다.

PCA : 데이터에 가장 가까운 **초평면**을 정의한 다음 데이터를 평면에 투영함

↓

투영시킬 **올바른 초평면** 선택하기

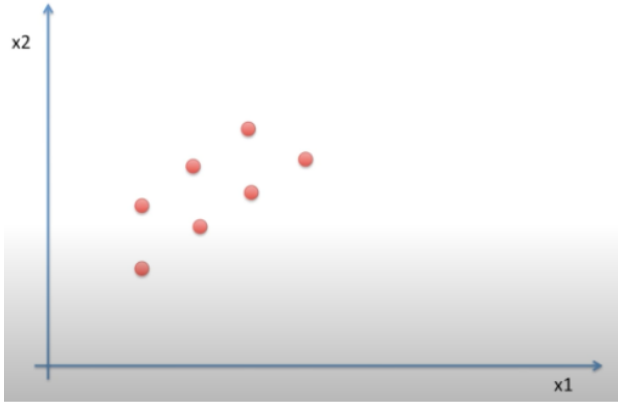
↓

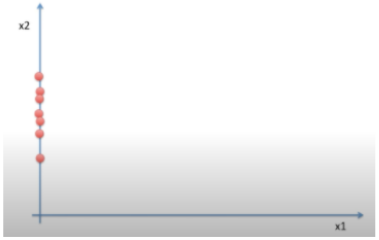
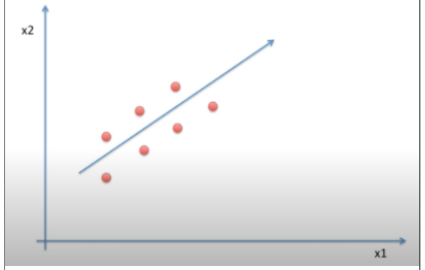
정보가 적게 손실되는(=분산이 최대로 보존되는) 축을 선택하기

3. PCA

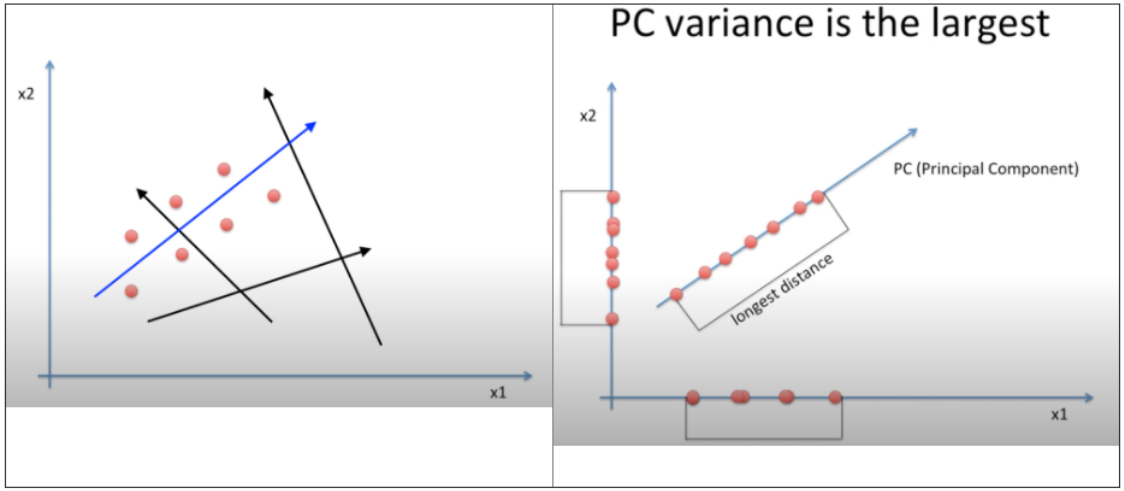
쉽게 이해해보자

ㅁ 목표 : 2차원 공간의 데이터를 1차원 공간의 데이터로 만들어주기



방법 1. 냅다 내려버리기	방법 2. 새로운 차원 찾기
how about to use x1 axis?  how about to use x2 axis?  → 내린 후 값이 겹치는 데이터들이 많고 아예 한 차원의 정보는 유실됨	How PCA reduce dimension?  → 새로운 차원인 화살표에 데이터를 내려주면 데이터들이 안 겹치고 새로운 축에 잘 반영됨

3. PCA



→ 무수히 다양한 화살표를 그릴 수 있는데, 그중 파란색 화살표처럼 데이터를 해당 화살표에 1차원으로 내렸을 때, 데이터들이 서로 겹치지 않게 하는 화살표를 찾아야 한다. 데이터가 최대한 안 겹치려면 데이터들끼리 멀리 퍼지게 해야 하고, 즉 차원을 내렸을 때, 데이터의 분산이 커야 하고 이는 화살표의 길이가 길어야 함을 의미한다.

→ 지금 예시는 2차원을 1차원으로 내리는 거지만, 3차원 이상의 경우를 저차원으로 내리는 경우엔, 만든 화살표에 직각이 되는 또 다른 축(화살표)를 찾아 나갈 수 있다.

→ IN 선형대수학

공분산 행렬에서 고유벡터, 고유값을 구하고 가장 분산이 큰 방향을 가진 고유벡터 e1 에 입력데이터를 선형변환하고, 그 다음으로 e1과 직교하며 e1 다음으로 분산이 큰 e2 고유벡터에 또 선형변환하고……

3. PCA

공분산 행렬

두 속성 간의 변동. 속성 쌍들의 변동(데이터의 분포형태) 이 얼마나 닮았는가(변수 간 상관정도)

대칭행렬

고유벡터를 직교행렬로
고유값을 정방행렬로
대각화 (행렬을 쪼갬)

$$A = V\Lambda V^T$$

* 행렬 x 벡터 -> 벡터 : 행렬은 하나의 벡터공간을 선형적으로 다른 벡터공간으로 매핑하는 기능을 가진다.

고유벡터 (화살표=새로운 축)

: 행렬X에 다른 행렬A을 곱했을 때, 행렬 X의 벡터들 중 벡터의 크기는 변하지만, 방향은 변하지 않는 벡터. 공분산 행렬의 고유벡터는 데이터가 어떤 방향으로 분산되었는지를 나타내 준다.

고유값 (화살표 길이)

: 해당 고유 벡터의 크기. 고유벡터 방향으로 얼마만큼의 크기로 벡터 공간이 늘려지는지 이야기 한다. 고유값이 큰 순서대로 (가장 길이가 긴 화살표) 고유벡터를 정렬하면 중요한 순서대로 주성분(새로운 축)을 구하는 것이 된다.

3. PCA

공분산 행렬 : 두 속성 간의 변동. 속성 쌍들의 변동이 얼마나 닮았는가(변수 간 상관정도)를 행렬형태로 보여줌

대칭행렬

고유벡터를 직교행렬로
고유값을 정방행렬로 대각화 할 수 있음
(-> SVD 특이값 분해=행렬을 대각화 하는 방법)

$$A = V\Lambda V^T$$

공분산행렬은 대칭행렬이기 때문에 고유값 분해를 했을 때, 고유백터의 직교 행렬(V) * 고유값의 정방행렬 ∧ * 고유 벡터의 직교 행렬의 전치행렬(VT) 로 표현된다. 이때 v1 은 가장 분산이 큰 방향을 가진 고유벡터이며, v2는 v1 에 직교하면서, 다음으로 가장 분산이 큰 방향을 가진 고유벡터가 된다. 즉 고유벡터들이 분산이 큰 화살표들인 것임!

$$A = V\Lambda V^T$$

$$= \begin{bmatrix} v_1 & v_2 & \cdots & v_N \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_N \end{bmatrix} \begin{bmatrix} v_1^T \\ v_2^T \\ \vdots \\ v_N^T \end{bmatrix}$$

$$= \begin{bmatrix} \lambda_1 v_1 & \lambda_2 v_2 & \cdots & \lambda_N v_N \end{bmatrix} \begin{bmatrix} v_1^T \\ v_2^T \\ \vdots \\ v_N^T \end{bmatrix}$$

3. PCA : 가장 대표적인 차원축소 알고리즘

[정리]

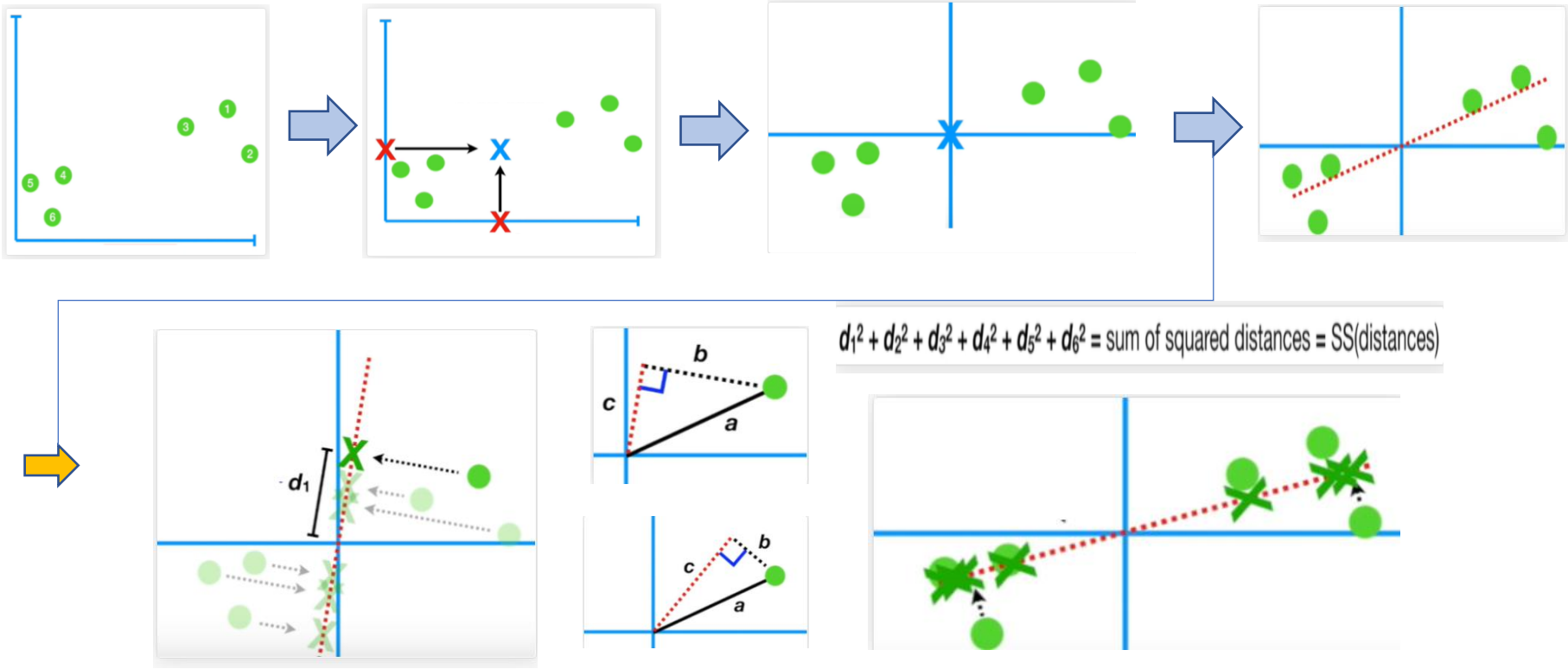
0. 평균이 0이 되도록 데이터들을 정규화 해준다. 현재 평균을 구해 각 데이터에 평균만큼 빼면 전체 평균이 0이 되도록 만들 수 있다.
1. 입력 데이터의 공분산 행렬을 구한다.
2. 공분산 행렬을 고유 값 분해 해서 고유 벡터와 고유 값을 구한다 (앞서 말한 최고의 화살표찾기)
3. 고유 값이 가장 큰 (화살표 길이가 가장 긴) k개의 고유 벡터를 추출
4. 고유 값이 가장 큰 순으로 추출된 고유벡터를 이용해 입력 데이터들을 선형 변환 (저차원으로 매핑)

PCA는 N개의 차원에서 N개의 주요 성분(새로운 축=PC)이 나오게 된다. 이 주요성분 중 데이터들을 잘 표현하고 있는 주요 성분만 선택하여 사용한다. 여기서 PC(새로운 축)는 기존의 속성을 가지고 새롭게 정의된 성분임을 잊지 말아야 한다.

★ 고유 값의 크기가 PC의 중요도! 고유값의 크기가 가장 큰 것이 고유벡터를 선택하는 기준이 됨

3. PCA : 가장 대표적인 차원축소 알고리즘

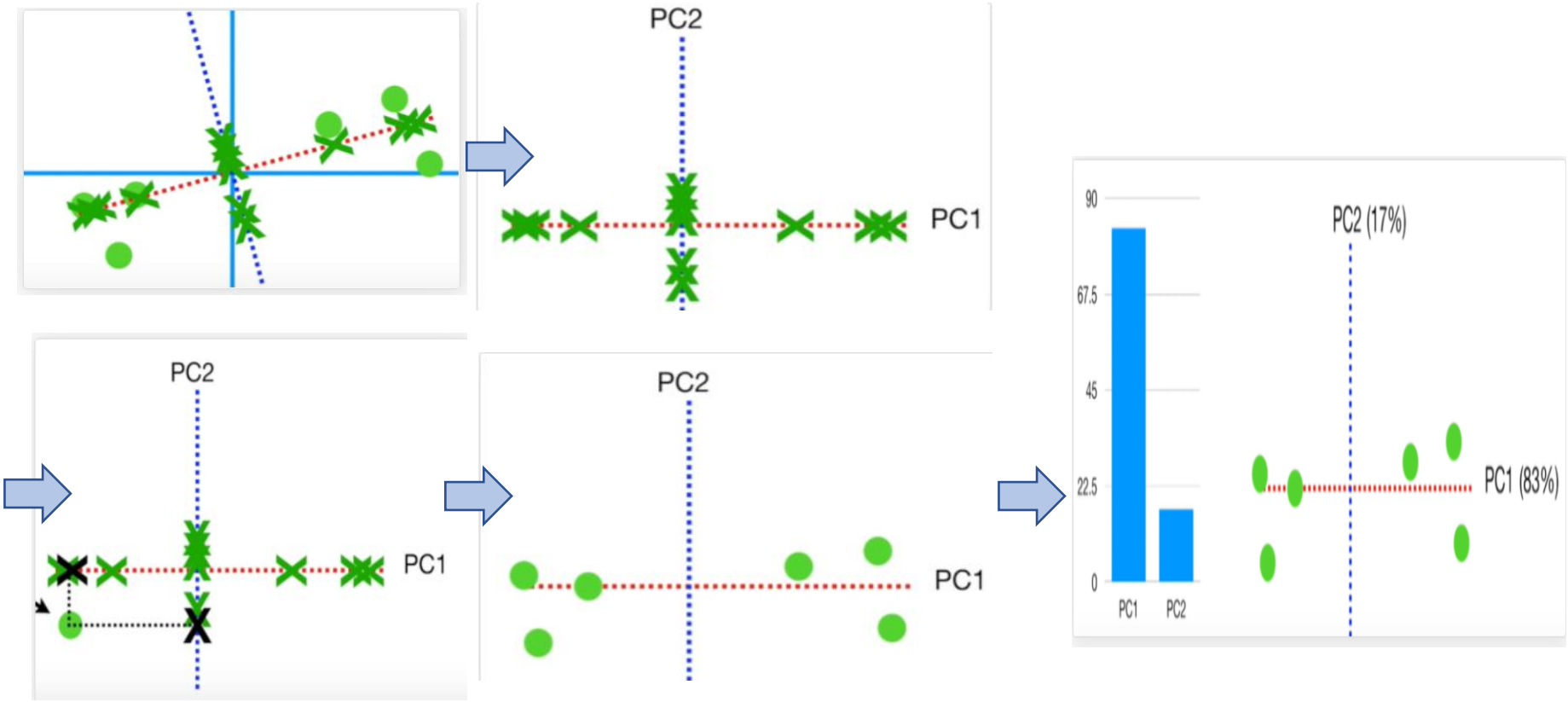
[그림으로 이해하기 : 왜 화살표 길이가 길어야 데이터의 분산이 큰거죠]



https://bkshin.tistory.com/entry/%EB%A8%B8%EC%8B%A0%EB%9F%AC%EB%8B%9D-9-PCA-Principal-Components-Analysis

3. PCA : 가장 대표적인 차원축소 알고리즘

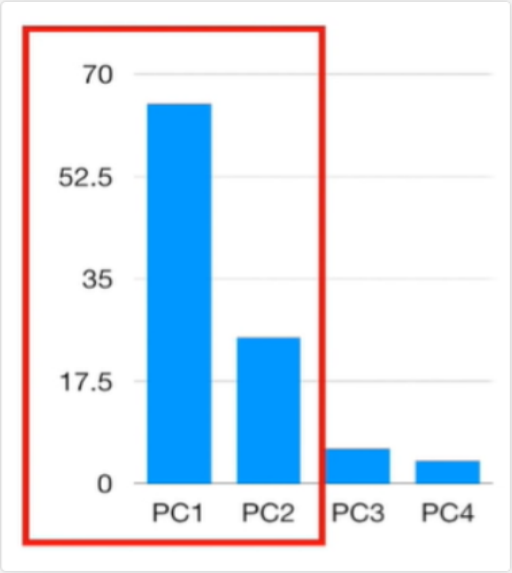
[그림으로 이해하기 : PC1,PC2]



3. PCA : 가장 대표적인 차원축소 알고리즘

3.6 적절한 차원 수 선택

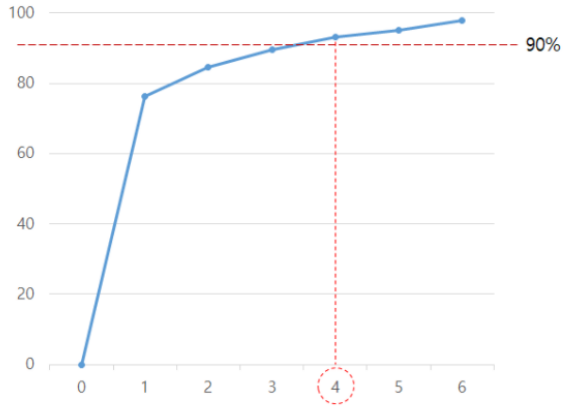
그렇다면 feature가 4개인 경우에 어떻게 되는지 보겠습니다. 앞서 말씀드린 것처럼 feature가 4개일 때 4차원 시각화는 할 수 없습니다. Scree Plot을 그렸을 때 아래와 같이 나온다고 합시다. PC1, PC2가 90% 이상을 차지합니다. 따라서 PC1과 PC2 만으로 2차원 시각화를 하더라도 전체의 90% 이상을 설명할 수 있습니다.



3. PCA : 가장 대표적인 차원축소 알고리즘

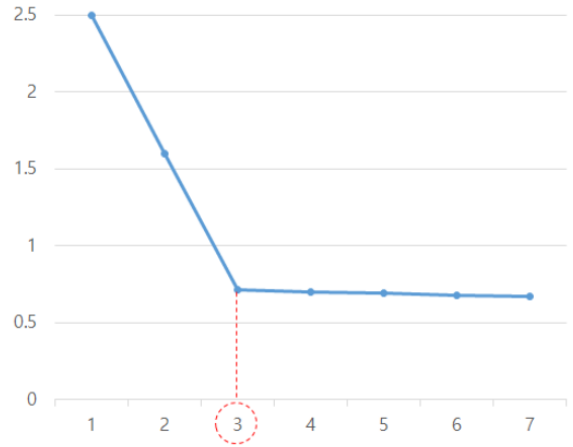
3.6 적절한 차원 수 선택

4-1 고유값을 퍼센트를 계산해 기준 값으로 주요 성분 선택하기



전체 고유 값을 %를 계산하여 선택 기준이 되는 threshold %를 정하고 기준을 넘는 부분까지 주요 성분을 선택합니다. 위 그래프는 90%를 기준으로 정하고 4개의 주요 성분을 선택하는 예제입니다.

4-2 고유값을 내림차순하여 주요 성분을 선택.



고유값 그대로 내림 차순으로 정렬하고 급격하게 완하 되는 지점으로 주요 성분의 개수를 선택합니다.

3. PCA

3.4 사이킷런으로 PCA 구하기

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
X2D = pca.fit_transform(X)
```

- **n_components 매개변수**
  - 주성분 개수를 지정

3.5 설명된 분산의 비율 : PC의 설명정도

- **explained_variance_ratio_ 변수**
  - 주성분의 설명된 분산의 비율
  - 즉, **전체 분산에서 차지하는 비율**(이 비율이 높을수록 차원 축소 과정에서 정보 손실이 적게 발생)
  - 주로 이 비율을 확인해서 주성분의 개수(즉, 적절한 차원 수)를 결정한다.

```
>>> pca.explained_variance_ratio_
array([0.84248607, 0.14631839])
```

ex) 2개의 주성분으로 전체 분산의 약 98%를 설명 가능

- **components_ 속성**
  - 주성분 벡터

```
print('singular vector :₩n', pca.components_.T)
```

```
singular vector :
[[-0.93636116 0.34027485 -0.08626012]
[-0.29854881 -0.90119108 -0.31420255]
[-0.18465208 -0.2684542   0.94542898]]
```

3. PCA

3.6 적절한 차원 수 선택(코드)

축소할 차원 수를 임의로 정하기 보단 충분한 분산이 될 때 까지 차원수를 줄여가는 방법을 사용하면 된다.
데이터 시각화를 위한 감소는 2차원이나 3차원이면 된다.

PCA를 계산한 뒤 훈련세트의 분산을 95%유지하는데 필요한 최소한의 차원수 계산

방법1. 차원을 축소하지 않고 PCA를 계산한 뒤 훈련 세트의 분산을 95%로 유지하는 데 필요한 최소한의 차원수를 계산

```
pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
```

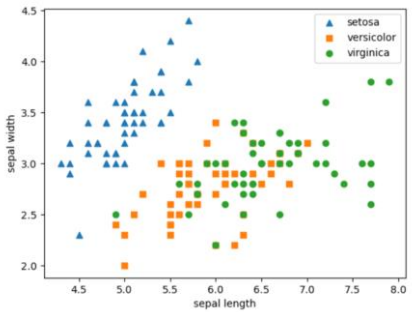
어떤 **함수**를 최소/최대로 만드는 정의역의 점들,

방법2. n_components를 설정해 PCA를 다시 실행하는 인자로 보존할 분산의 비율을 입력하기

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

3. PCA

사이킷런으로 PCA 구하기 _붓꽃 예제
4개의 속성을 PCA를 이용하여 2개의 차원으로 변환



주성분 분석은 기본적으로 수치형 변수를 이용하기 때문에 수치형 데이터(Sepal length, Sepal width, Petal length, Petal width)만 추출하여 x 변수에 저장하겠습니다.

```
#수치형 데이터만 추출
>>> features = ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal
>>> x = df.loc[:, features].values
```

3-2. Python로 PCA 실습하기 : 정규화 시키기

변수간의 스케일이 차이가 나면 스케일 큰 변수가 주성분에 영향을 많이 주기 때문에 주성분 분석 전에 변수를 표준화나 정규화시켜주는 것이 좋습니다.[4)]

Python에서는 사이킷런 패키지의 StandardScaler모듈을 통해 변수들을 정규화시킬 수 있습니다.

```
#수치형 변수 정규화
>>> from sklearn.preprocessing import StandardScaler
>>> x = StandardScaler().fit_transform(x)
```

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2],
```

정규화

```
array([[-9.00681170e-01,  1.01900435e+00, -1.34022653e+00,
        -1.31544430e+00],
       [-1.14301691e+00, -1.31979479e-01, -1.34022653e+00,
        -1.31544430e+00],
       [-1.38535265e+00,  3.28414053e-01, -1.39706395e+00,
        -1.31544430e+00],
       [-1.50652052e+00,  9.82172869e-02, -1.28338910e+00,
        -1.31544430e+00],
       [-1.02184904e+00,  1.24920112e+00, -1.34022653e+00,
        -1.31544430e+00],
```

변수를 정규화하면 평균 0, 분산이 1인 새로운 변수로 변환되게 됩니다. 이 정규화된 변수로 주성분 분석을 진행해보도록 하겠습니다.

3-3. Python로 PCA 실습하기 : 주성분 분석 실시하기

sklearn.decomposition.PCA() 함수를 통해 주성분 객체를 생성할 수 있으며 이 객체의 fit_transform() 함수를 이용해 데이터에 적합하여 주성분 점수(주성분 선형 변환에 생성된 값)을 반환받게 됩니다. 그럼 PCA 객체를 만들어 데이터에 적합하여 봅시다.

```
#주성분 분석 실시하기
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=2) #PCA 객체 생성 (주성분 갯수 2개 생성)
>>> principalComponents = pca.fit_transform(x)
>>> principalDf = pd.DataFrame(data = principalComponents
>>>              , columns = ['principal component 1', 'principal component 2'])
```

위의 코드 중에서 PCA() 함수의 n_components 인자는 주성분 갯수를 설정하는 인자이며 default는 None으로 원래 변수의 갯수만큼 생성하게 됩니다. 주성분 점수는 principalDf에 저장되었으면 확인하면 아래와 같습니다.

3. PCA

사이킷런으로 PCA 구하기 _붓꽃 예제

```
#주성분 점수 확인
>>> principalDf
```

Out[43]:

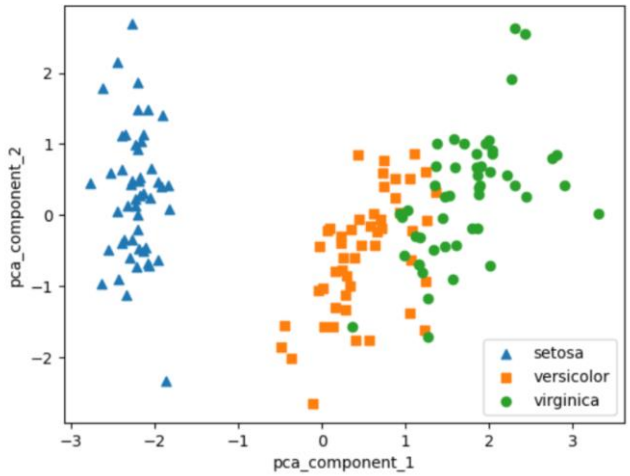
	principal component 1	principal component 2
0	-2.264703	0.480027
1	-2.080961	-0.674134
2	-2.364229	-0.341908
3	-2.299384	-0.597395
4	-2.389842	0.646835
...	...	...
145	1.870503	0.386966
146	1.564580	-0.896687
147	1.521170	0.269069
148	1.372788	1.011254
149	0.960656	-0.024332

150 rows × 2 columns

위에서 확인할 수 있듯이 행수는 기존 행수와 같은 150개, 열 수는 설정한 주성분 갯수 2개가 나타나는 것을 확인할 수 있습니다. 모든 주성분 점수들의 분산 합은 원래 변수의 분삽 합과 같다는 특징을 가지고 있으며 또한 주성분 축이 직교함으로 주성분 점수들 간의 상관계수는 0임을 알 수 있습니다.

```
#원래 변수 분산합 = 주성분 점수의 분산합
>>> df.loc[:, features].values.var()  #원 변수의 분산합
3.896056416666667

>>> principalDf.var() #주성분 점수의 분산
principal component 1    2.938085
principal component 2    0.920165
dtype: float64

#주성분 점수간의 상관계수 = 0
>>> principalDf.corr() #주성분 점수간의 상관계수
                      principal component 1     principal component 2
principal component 1          1.000000e+00     -4.053019e-17
principal component 2          -4.053019e-17     1.000000e+00
```

```
#setosa를 세모, versicolor를 네모, virginica를 동그라미로 표시
markers=['^','s','o']

#분포 확인
for i, marker in enumerate(markers):
    x_axis_data = irisDF_pca[irisDF_pca['target']==i]['pca_component_1']
    y_axis_data = irisDF_pca[irisDF_pca['target']==i]['pca_component_2']
    plt.scatter(x_axis_data, y_axis_data, marker=marker, label=iris.target_names[i])

plt.legend()
plt.xlabel('pca_component_1')
plt.ylabel('pca_component_2')
plt.show()
```



3. PCA

3.7 압축을 위한 PCA

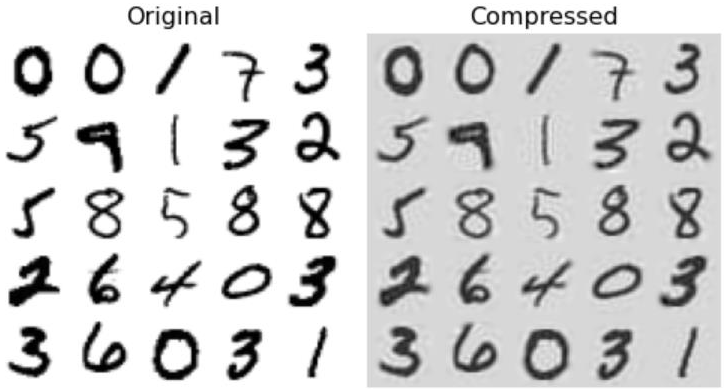
$$\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d\text{-proj}} \mathbf{W}_d^T$$

◆ 압축을 위한 PCA

- **차원 축소**를 하면 훈련 세트의 크기가 줄어들어, **분류 알고리즘의 속도를 크게 높일 수 있다.**
- 또한 반대로 **압축된 데이터 셋에 PCA 투영 반환을 적용시켜, 원본 데이터와 유사하게 만들 수 있다.**
  - inverse_transform( ) 메소드를 사용하면 된다.
  - 단, **원본 데이터와 똑같이 복구시킬 수는 없다!!**

```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

다음은 MNIST 데이터셋에 대하여 원본 데이터셋과 압축 후 복원된 결과를 비교한 그림입니다. 이미지의 품질이 손상되긴 했지만 숫자의 모양은 온전한 것을 확인할 수 있습니다.



3. PCA

3.8 랜덤 PCA

svd_solver = "randomized"로 지정하면 sklearn은 **랜덤 PCA**라는 확률적 알고리즘을 이용하여 **축소할 d차원에 대한 d개의 PC를 '근삿값'으로 빠르게 찾습니다.**

```
rnd_pca = PCA(n_components=154, svd_solver="randomized",
random_state=42)
X_reduced = rnd_pca.fit_transform(X_train)
```

svd_solver의 기본값은 **"auto"**인데, 원본 데이터의 크기나 차원 수가 500보다 크고, 축소할 차원이 이것들의 80%보다 작으면 sklearn은 자동으로 랜덤 PCA 알고리즘을 사용합니다. 만약 이것을 방지하고 싶다면 **"full"**을 사용하면 됩니다.

3. PCA

3.9 점진적 PCA

PCA 구현의 문제는 SVD 알고리즘 실행을 위해 전체 데이터셋을 메모리에 올려야 한다는 점입니다. **점진적 PCA(IPCA : incremental PCA)는 dataset을 mini-batch로 나눈 뒤 하나 씩 주입하여 적용하여 이를 보완합니다.**

다음 코드는 MNIST 데이터셋을 100개의 mini-batch로 나눠 차원을 축소하는 코드입니다.

```
from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    print(".", end="") # 책에는 없음
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)
```

4. 커널 PCA

커널 PCA

커널 트릭을 PCA에 적용해 차원 축소를 위한 복잡한 비선형 투형을 수행할 수 있음.
투영된 후에 샘플 군집을 유지하거나 꼬인 매니폴드에 가까운 데이터셋을 펼칠 때 유용

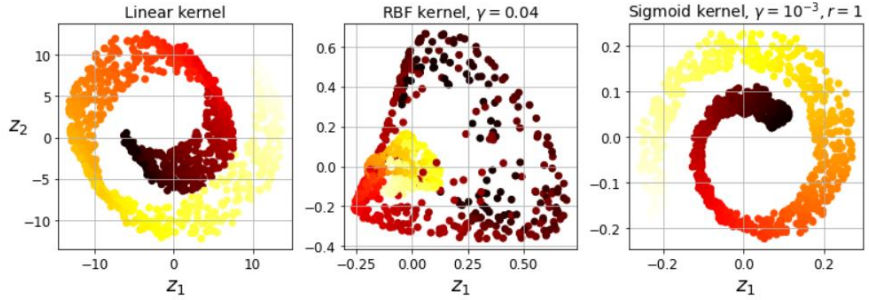
커널 트릭

샘플을 특성 공간으로 암묵적으로 매핑하여 서포트 벡터 머신의 비선형 분류와 회귀를 가능하게 하는 수학적 기법

```
X, t = make_swiss_roll(n_samples=1000, noise=0.2, random_state=42)
```

```
from sklearn.decomposition import KernelPCA

rbf_pca = KernelPCA(n_components=2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

```
from sklearn.decomposition import KernelPCA

lin_pca = KernelPCA(n_components=2, kernel="linear", fit_inverse_transform=True)
rbf_pca = KernelPCA(n_components=2, kernel="rbf", gamma=0.0433, fit_inverse_transform=True)
sig_pca = KernelPCA(n_components=2, kernel="sigmoid", gamma=0.001, coef0=1, fit_inverse_transform=True)

y = t > 6.9
```



4. 커널 PCA

커널 PCA

커널 트릭을 PCA에 적용해 차원 축소를 위한 복잡한 비선형 투형을 수행할 수 있음.
투영된 후에 샘플 군집을 유지하거나 꼬인 매니폴드에 가까운 데이터셋을 펼칠 때 유용

비지도 학습으로 좋은 커널과 하이퍼파라미터를 선택하기 위한 명확한 성능 기준이 없음 → 해결방법..??!!

1. 그리드 탐색 이용

1. kPCA를 사용해 차원을 2차원으로 축소 & 로지스틱 회귀를 이용해 분류
2. GridSearchCV를 사용해 kPCA의 가장 좋은 커널과 gamma 파라미터를 찾는다.

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

clf = Pipeline([
        ("kpca", KernelPCA(n_components=2)),
        ("log_reg", LogisticRegression(solver="lbfgs"))
    ])

param_grid = [{
        "kpca__gamma": np.linspace(0.03, 0.05, 10),
        "kpca__kernel": ["rbf", "sigmoid"]
    }]

grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
```

```
GridSearchCV(cv=3,
             estimator=Pipeline(steps=[('kpca', KernelPCA(n_components=2)),
                                       ('log_reg', LogisticRegression())]),
             param_grid=[{'kpca__gamma': array([0.03      , 0.03222222, 0.03444444, 0.03666667, 0.03888889,
       0.04111111, 0.04333333, 0.04555556, 0.04777778, 0.05      ]),
                          'kpca__kernel': ['rbf', 'sigmoid']}])
```

```
print(grid_search.best_params_)
```

```
{'kpca__gamma': 0.04333333333333335, 'kpca__kernel': 'rbf'}
```

best_params_ 변수를 이용해 커널과 gamma 파라미터를 찾는다.

4. 커널 PCA

커널 PCA

커널 트릭을 PCA에 적용해 차원 축소를 위한 복잡한 비선형 투형을 수행할 수 있음.
투영된 후에 샘플 군집을 유지하거나 꼬인 매니폴드에 가까운 데이터셋을 펼칠 때 유용

비지도 학습으로 좋은 커널과 하이퍼파라미터를 선택하기 위한 명확한 성능 기준이 없음 → 해결방법..??!!

2. 재구성 원상 이용

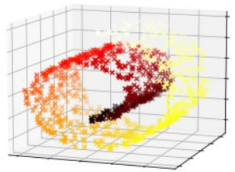
재구성 원상을 얻어 원본 샘플과의 제곱 거리를 측정해, 재구성 원상의 오차를 최소화하는 커널과 하이퍼 파라미터를 선택

재구성 원상

1. 축소된 공간에 있는 샘플에 대해 선형 PCA를 역전시키면 재구성된 데이터 포인트는 특성 공간에 놓인다.
2. 재구성된 포인트에 가깝게 매핑된 원본 공간의 포인트를 찾는다.

```
plt.figure(figsize=(6, 5))

X_inverse = rbf_pca.inverse_transform(X_reduced_rbf)

ax = plt.subplot(111, projection='3d')
ax.view_init(10, -70)
ax.scatter(X_inverse[:, 0], X_inverse[:, 1], X_inverse[:, 2], c=t, cmap=plt.cm.hot, marker="x")
ax.set_xlabel("")
ax.set_ylabel("")
ax.set_zlabel("")
ax.set_xticklabels([])
ax.set_yticklabels([])
ax.set_zticklabels([])

save_fig("preimage_plot", tight_layout=False)
plt.show()
```

그림 저장 preimage_plot



(커널 트릭 덕분)
특성 맵을 사용해 훈련세트를 무한 차원의 특성 공간에 매핑한 다음, 변환된 데이터셋을 선형 PCA를 사용해 2D로 투영시킨 데이터셋 == 원본 데이터에 RBF 커널의 kPCA를 적용한 2D 데이터셋

5. LLE

지역 선형 임베딩

하나의 비선형 차원 축소
투영에 의존하지 않는 매니폴드 학습

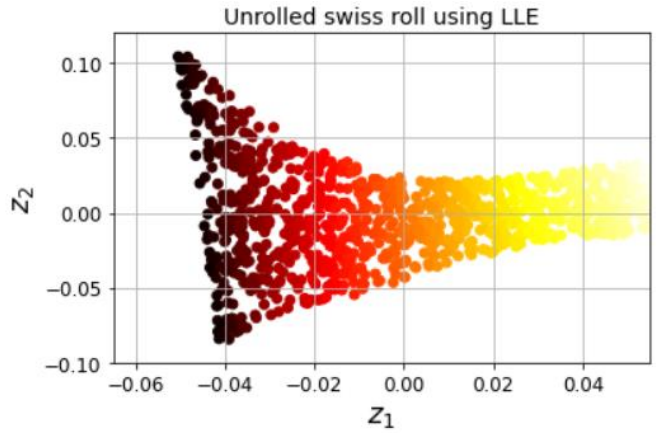
장점: 잡음이 너무 많지 않은 경우 꼬인 매니폴드를 펼치는 데 잘 작동함.
단점: 대용량 데이터셋에 적용하기 어려움.

과정

1. 각 훈련 샘플이 가장 가까운 이웃 (c.n.)에 얼마나 선형적으로 연관되어 있는지 측정한다.
2. 국부적인 관계가 가장 잘 보존되는 훈련 세트의 저차원 표현을 찾는다.

```
from sklearn.manifold import LocallyLinearEmbedding

lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10, random_state=42)
X_reduced = lle.fit_transform(X)
```

```
plt.title("Unrolled swiss roll using LLE", fontsize=14)
plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=t, cmap=plt.cm.hot)
plt.xlabel("$z_1$", fontsize=18)
plt.ylabel("$z_2$", fontsize=18)
plt.axis([-0.065, 0.055, -0.1, 0.12])
plt.grid(True)

save_fig("lle_unrolling_plot")
plt.show()
```



5. LLE

단계 1. 선형적인 지역 관계 모델링

1. 각 훈련 샘플에 대해서 $x^{(i)}$에 대해 가장 가까운 k개의 샘플을 찾습니다.
2. $x^{(i)}$와 $\sum_{j=1}^{m} w_{i,j}x^{(j)}$ 사이의 제곱거리가 최소가 되는 $w_{(i,j)}$를 찾습니다.
3. $x^{(j)}$가 $x^{(i)}$의 가장 가까운 k개의 샘플이 아닐 경우 $w_{i,j} = 0$
   식으로 표현하면 밑의 식이 됩니다.

$$\hat{W} = argmax_W \sum_{i=1}^{m} \|x^{(i)} - \sum_{j=1}^{m} w_{i,j}x^{(j)}\|^2$$

where

$$\begin{cases} w_{i,j} = 0, & \text{if } x^{(j)} \text{ is not one of the } k \text{ c.n. of } x^{(i)} \\ \sum_{j=1}^{m} w_{i,j} = 1, & \text{for i} = 1,2,\cdots,\text{m} \end{cases}$$

샘플을 고정하고 최적의 가중치를 찾는다.

→ 제한이 있는 최적화 문제

단계 2. 관계를 보존하는 차원 축소

1. $z^{(i)}$를 d차원 공간에서 $x^{(i)}$의 image라면 $z^{(i)}$와 $\sum_{j=1}^{m} \hat{w_{i,j}}z^{j}$사이의 거리를 최소화 하는 $z^{(i)}$를 찾습니다.
   이것을 식으로 나타내면 다음과 같이 됩니다.

$$\hat{Z} = argmax_Z \sum_{i=1}^{m} \|z^{(i)} - \sum_{j=1}^{m} \hat{w_{i,j}}z^{j}\|^2$$

가중치를 고정하고 저차원의 공간에서 샘플 이미지의 최적 위치를 찾는다.

6. 다른 차원 축소 기법

랜덤 투영 (random projection)	선형 투영을 사용해 데이터를 저차원 공간으로 투영 실제로 거리를 잘 보존하는 것으로 밝혀짐 차원 축소 품질은 샘플 수와 목표 차원수에 따라 다름
다차원 스케일링 (MDS)	샘플 간의 거리를 보존하면서 차원 축소
isomap	각 샘플을 가장 가까운 이웃과 연결하는 식으로 그래프를 만듦 샘플 간의 거리를 보존하면서 차원 축소
t-SNE	시각화에 많이 사용 고차원 공간에 있는 샘플의 군집을 시각화할 때 사용
선형 판별 분석 (LDA)	사실 분류 알고리즘 투영 이용 → 다른 분류 알고리즘을 적용하기 전에 차원을 축소시키는 데 좋음

7. 연습문제

9번. MNIST 데이터셋과 랜덤 포레스트 분류기를 이용해 raw data일 경우와 분산이 95%가 되도록 차원 축소시켰을 경우 중 어느 것이 훈련 속도가 더 빨라졌는지 비교하시오.

[raw data의 경우]

In [3]:
```
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', version=1, as_frame=False)
mnist.target = mnist.target.astype(np.uint8)
```

In [4]:
```
X_train = mnist['data'][:60000]
y_train = mnist['target'][:60000]

X_test = mnist['data'][60000:]
y_test = mnist['target'][60000:]
```

In [6]:
```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=100, random_state=42)

import time

t0 = time.time()
rnd_clf.fit(X_train, y_train)
t1 = time.time()

print("훈련 시간: {:.2f}s".format(t1 - t0))
```

훈련 시간: 53.66s ←

In [7]:
```
from sklearn.metrics import accuracy_score

y_pred = rnd_clf.predict(X_test)
accuracy_score(y_test, y_pred)
```

Out[7]: 0.9705

[차원 축소시켰을 경우]

In [9]:
```
from sklearn.decomposition import PCA

pca = PCA(n_components=0.95)
X_train_reduced = pca.fit_transform(X_train)

rnd_clf2 = RandomForestClassifier(n_estimators=100, random_state=42)
t0 = time.time()
rnd_clf2.fit(X_train_reduced, y_train)
t1 = time.time()

print("훈련 시간: {:.2f}s".format(t1 - t0))
```

훈련 시간: 115.49s ←

In [10]:
```
X_test_reduced = pca.transform(X_test)

y_pred = rnd_clf2.predict(X_test_reduced)
accuracy_score(y_test, y_pred)
```

Out[10]: 0.9481

훈련 시간이 늘어났다!!

→ 데이터셋, 모델, 훈련 알고리즘에 따라 차원 축소가 훈련 시간에 주는 영향은 다르다!

7. 연습문제

10번. t-SNE, PCA, LLE, MDS 등의 차원 축소 알고리즘을 이용해 MNIST 데이터셋을 2차원으로 축소시키고 시각화 결과를 비교해 보시오.

산점도와 색깔 있는 숫자를 쓰기 위해 plot_digits 함수를 만든다. → 군집을 더 정확히 파악 가능

In [17]:
```
from sklearn.preprocessing import MinMaxScaler
from matplotlib.offsetbox import AnnotationBbox, OffsetImage

def plot_digits(X, y, min_distance=0.05, images=None, figsize=(13, 10)):
    # 입력 특성의 스케일을 0에서 1 사이로 만듭니다.
    X_normalized = MinMaxScaler().fit_transform(X)
    # 그릴 숫자의 좌표 목록을 만듭니다.
    # 반복문 아래에서 'if' 문장을 쓰지 않기 위해 시작할 때 이미 그래프가 그려져 있다고 가정합니다.
    neighbors = np.array([[10., 10.]])
    # 나머지는 이해하기 쉽습니다.
    plt.figure(figsize=figsize)
    cmap = mpl.cm.get_cmap("jet")
    digits = np.unique(y)
    for digit in digits:
        plt.scatter(X_normalized[y == digit, 0], X_normalized[y == digit, 1], c=[cmap(digit / 9)])
    plt.axis("off")
    ax = plt.gcf().gca()  # 현재 그래프의 축을 가져옵니다.
    for index, image_coord in enumerate(X_normalized):
        closest_distance = np.linalg.norm(neighbors - image_coord, axis=1).min()
        if closest_distance > min_distance:
            neighbors = np.r_[neighbors, [image_coord]]
            if images is None:
                plt.text(image_coord[0], image_coord[1], str(int(y[index])),
                         color=cmap(y[index] / 9), fontdict={"weight": "bold", "size": 16})
            else:
                image = images[index].reshape(28, 28)
                imagebox = AnnotationBbox(OffsetImage(image, cmap="binary"), image_coord)
                ax.add_artist(imagebox)
```

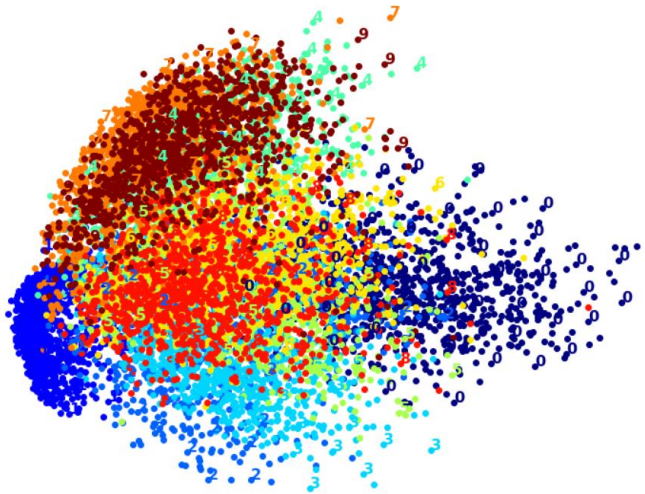
7. 연습문제

10번. t-SNE, PCA, LLE, MDS 등의 차원 축소 알고리즘을 이용해 MNIST 데이터셋을 2차원으로 축소시키고 시각화 결과를 비교해 보시오.

[PCA]

In [26]:
```
from sklearn.decomposition import PCA
import time

t0 = time.time()
X_pca_reduced = PCA(n_components=2, random_state=42).fit_transform(X)
t1 = time.time()
print("PCA 시간: {:.1f}s.".format(t1 - t0))
plot_digits(X_pca_reduced, y)
plt.show()
```

```
PCA 시간: 0.3s.
```



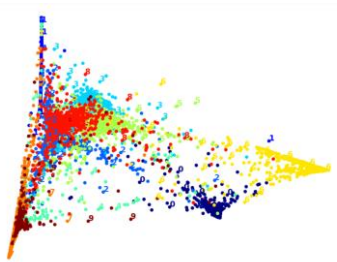
→ 빠르지만 군집이 겹쳐져 있어 확인이 어렵다.

7. 연습문제

10번. t-SNE, PCA, LLE, MDS 등의 차원 축소 알고리즘을 이용해 MNIST 데이터셋을 2차원으로 축소시키고 시각화 결과를 비교해보시오.

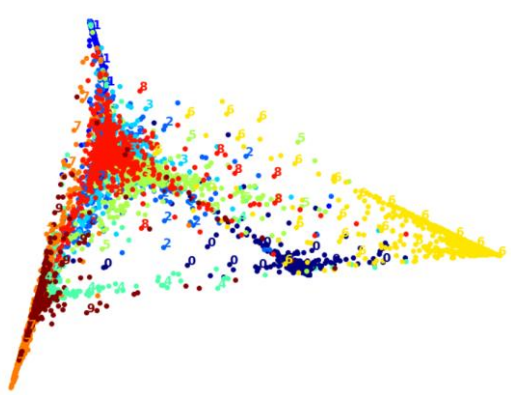
[LLE]

```
In [19]: ## LLE
         from sklearn.manifold import LocallyLinearEmbedding

         t0 = time.time()
         X_lle_reduced = LocallyLinearEmbedding(n_components=2, random_state=42).fit_transform(X)
         t1 = time.time()
         print("LLE 시간: {:.1f}s.".format(t1 - t0))
         plot_digits(X_lle_reduced, y)
         plt.show()

         LLE 시간: 187.2s.
```



[PCA+LLE]

```
In [20]: ## 먼저 분산 95%를 보존하는 PCA
         from sklearn.pipeline import Pipeline

         pca_lle = Pipeline([
             ("pca", PCA(n_components=0.95, random_state=42)),
             ("lle", LocallyLinearEmbedding(n_components=2, random_state=42)),
         ])
         t0 = time.time()
         X_pca_lle_reduced = pca_lle.fit_transform(X)
         t1 = time.time()
         print("PCA+LLE 시간: {:.1f}s.".format(t1 - t0))
         plot_digits(X_pca_lle_reduced, y)
         plt.show()

         PCA+LLE 시간: 50.9s.
```



→ 군집이 중복되어 있다. (좋지 않은 결과)
→ PCA와 같이 사용했을 때, 더 빠른 속도로 비슷한 결과를 얻는다.

7. 연습문제

10번. t-SNE, PCA, LLE, MDS 등의 차원 축소 알고리즘을 이용해 MNIST 데이터셋을 2차원으로 축소시키고 시각화 결과를 비교해보시오.

[MDS]

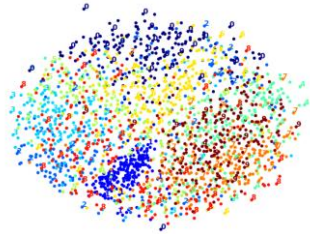
In [21]:
```
## MDS
from sklearn.manifold import MDS

m = 2000
t0 = time.time()
X_mds_reduced = MDS(n_components=2, random_state=42).fit_transform(X[:m])
t1 = time.time()
print("MDS 시간 {:.1f}s (on just 2,000 MNIST images instead of 10,000).".format(t1 - t0))
plot_digits(X_mds_reduced, y[:m])
plt.show()
```

```
MDS 시간 210.1s (on just 2,000 MNIST images instead of 10,000).
```

[PCA+MDS]

In [22]:
```
from sklearn.pipeline import Pipeline

pca_mds = Pipeline([
    ("pca", PCA(n_components=0.95, random_state=42)),
    ("mds", MDS(n_components=2, random_state=42)),
])
t0 = time.time()
X_pca_mds_reduced = pca_mds.fit_transform(X[:2000])
t1 = time.time()
print("PCA+MDS 시간 {:.1f}s (on 2,000 MNIST images).".format(t1 - t0))
plot_digits(X_pca_mds_reduced, y[:2000])
plt.show()
```

```
PCA+MDS 시간 240.0s (on 2,000 MNIST images).
```



→ 모든 군집이 중복되어 있다. (좋지 않은 결과)
→ PCA와 같이 사용했을 때, 비슷한 속도로 같은 결과를 얻는다. (도움 X)

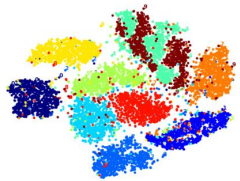
7. 연습문제

10번. t-SNE, PCA, LLE, MDS 등의 차원 축소 알고리즘을 이용해 MNIST 데이터셋을 2차원으로 축소시키고 시각화 결과를 비교해보시오.

[t-SNE]

In [24]:
```
## t-SNE
from sklearn.manifold import TSNE

t0 = time.time()
X_tsne_reduced = TSNE(n_components=2, random_state=42).fit_transform(X)
t1 = time.time()
print("t-SNE 시간 {:.1f}s.".format(t1 - t0))
plot_digits(X_tsne_reduced, y)
plt.show()
```

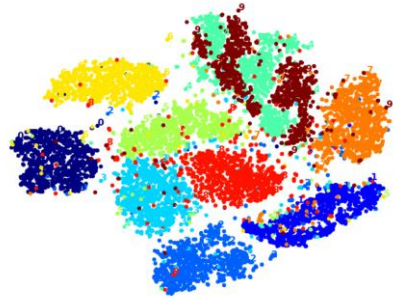
```
t-SNE 시간 264.3s.
```



[PCA+t-SNE]

In [25]:
```
pca_tsne = Pipeline([
    ("pca", PCA(n_components=0.95, random_state=42)),
    ("tsne", TSNE(n_components=2, random_state=42)),
])
t0 = time.time()
X_pca_tsne_reduced = pca_tsne.fit_transform(X)
t1 = time.time()
print("PCA+t-SNE 시간 {:.1f}s.".format(t1 - t0))
plot_digits(X_pca_tsne_reduced, y)
plt.show()
```

```
PCA+t-SNE 시간 131.0s.
```



→ 군집의 중복이 다른 알고리즘들에 비해 적다. (제일 좋은 결과)
→ PCA와 같이 사용했을 때, 더 빠른 속도로 같은 결과를 얻는다.

감사합니다

Q&A