

특성 추출을 사용한 차원 축소

새로운 특성을 만드는 식으로 특성 행렬의 차원을 축소하는 방법은 이상적으로 훨씬 적은 차원으로 좋은 품질의 모델을 동일하게 훈련할 수 있습니다

특성 추출을 사용한 차원 축소

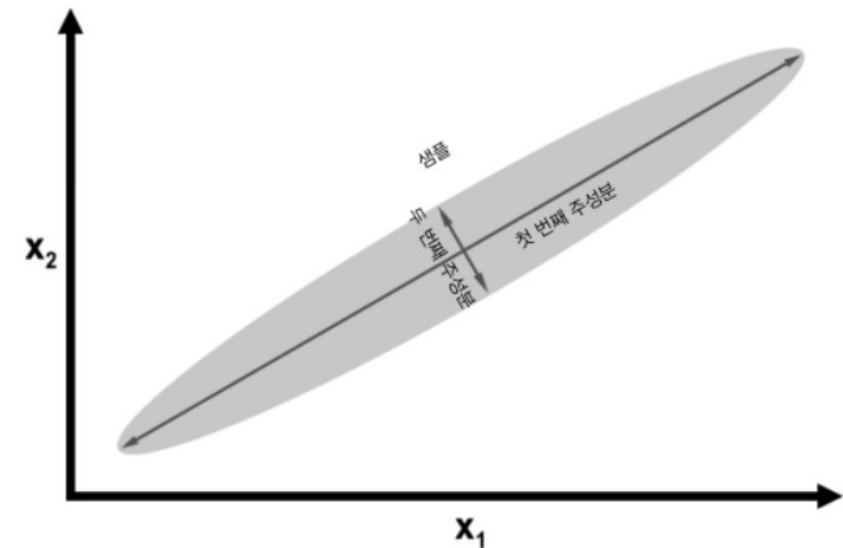
➤ 특성 추출을 사용한 차원 축소

- 학습 알고리즘은 충분한 데이터가 주어지지 않으면 올바르게 작동하지 않습니다.
- 차원 축소를 위한 특성 추출의 목적은 특성에 내재된 정보는 많이 유지하면서 특성 집합($P_{original}$)을 새로운 집합(P_{new})으로 변환하는 것입니다.
- 고품질 예측을 만들기 위한 데이터의 능력을 조금만 희생하고 특성의 수를 줄입니다.
- 특성 추출 기법의 단점은 새로운 특성을 사람이 이해하지 못한다는 것입니다.
- 모델을 훈련하기 위해 필요한 특성을 담고 있지만 사람의 눈에는 무작위한 숫자의 모음으로 보일 것입니다.

특성 추출을 사용한 차원 축소

➤ 주성분을 사용해 특성 줄이기

- 일련의 특성이 주어졌을 때 데이터의 분산을 유지하면서 특성의 수를 줄일 수 있습니다.
- PCA는 데이터의 분산을 유지하면서 특성의 수를 줄이는 선형 차원 축소 기법입니다
- PCA는 비지도 학습 기법입니다.
- PCA는 타킷 벡터의 정보를 사용하지 않고 특성 행렬만 이용합니다.
- 데이터는 두개의 특성 x_1 과 x_2 를 가집니다
- 그래프의 샘플들은 길이가 길고 높이는 낮은 타원 모양으로 퍼져 있는 '길이' 방향의 분산이 '높이'방향보다 훨씬 큽니다.
- 길이와 높이 대신 가장 분산이 많은 방향을 첫번째 주성분으로 부르고 두번째로 가장 많은 방향을 두번째 주성분이라고 부릅니다.
- 특성을 줄이는 한 가지 방법은 2D 공간의 모든 샘플을 1차원 주성분에 투영하는 것입니다.
- `n_components`의 입력 매개변수값이 1보다 크면 `n_components` 개수만큼의 특성이 반환
- `n_components`를 0과 1 사이로 지정하면 `pca`는 해당 비율의 분산을 유지할 수 있는 최소한의 특성 개수를 반환
- 원본 특성의 95%와 99%의 분산을 유지하는 0.95와 0.99가 사용됨
- `whiten=True`로 지정하면 각 주성분의 값을 평균이 0이고 분산이 1이 되도록 변환
- `solver="randomized"`는 아주 짧은 시간 안에 첫 번째 주성분을 찾아주는 확률적 알고리즘을 사용



특성 추출을 사용한 차원 축소

➤ 주성분을 사용해 특성 줄이기

- whitening은 주성분에 투영된 특성의 스케일을 맞추는 역할을 합니다.
- PCA는 평균을 0으로 맞추기 때문에 화이트닝 옵션 대신 나중에 투영된 특성을 표준화해도 됩니다

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn import datasets

digits = datasets.load_digits() # 8X8 크기의 손글씨 숫자 데이터 로드
features = StandardScaler().fit_transform(digits.data) # 특성 행렬을 표준화 처리

# 99%의 분산을 유지하도록 PCA 클래스 객체를 만듭니다.
pca = PCA(n_components=0.99, whiten=True)

features_pca = pca.fit_transform(features) # PCA를 수행

print("원본 특성 개수:", features.shape[1]) # 결과를 확인
print("줄어든 특성 개수:", features_pca.shape[1])
```

PCA는 데이터의 분산(variance)을 최대한 보존하면서 서로 직교하는 새 기저(축)를 찾아, 고차원 공간의 표본들을 선형 연관성이 없는 저차원 공간으로 변환하는 기법입니다.

특성 추출을 사용한 차원 축소

➤ 주성분을 사용해 특성 줄이기

- PCA 클래스의 `whiten` 매개변수의 기본값은 `False`로 화이트닝을 적용하지 않으면 평균은 0이지만 스케일은 맞춰지지 않습니다.
- PCA로 찾은 주성분은 `components_` 속성에 저장
- 각 주성분은 원본 특성 공간에서 어떤 방향을 나타내므로 이 벡터 크기는 64입니다.

```
# 주성분에 투영된 처음 두 개의 특성을 사용해 산점도 출력
import matplotlib.pyplot as plt

plt.scatter(features_pca[:, 0], features_pca[:, 1])
plt.show()

pca_nowhiten = PCA(n_components=0.99)
features_nowhiten = pca_nowhiten.fit_transform(features)
plt.scatter(features_nowhiten[:, 0], features_nowhiten[:, 1])
plt.show()

pca_nowhiten.components_.shape
```

특성 추출을 사용한 차원 축소

➤ 주성분을 사용해 특성 줄이기

- 특성 행렬을 주성분에 투영하려면 components_ 배열을 전치하여 행렬곱을 수행합니다.
- (넘파이 allclose())를 사용하여 features_nowhiten 배열과 동일한지 확인
- 적절한 분산 비율을 선택하기 위해 전체 주성분의 explained 분산에서 유지되는 분산의 양이 크게 늘어나지 않는 지점을 찾을 수 있습니다.
- n_components 매개변수를 지정하지 않으면 특성 개수만큼 주성분이 만들어집니다.
- explained 분산은 explained_variance_ratio_ 속성에 저장되어 있습니다.

```
import numpy as np
```

```
np.allclose(features_nowhiten, np.dot(features, pca_nowhiten.components_.T))
```

```
pca = PCA(whiten=True).fit(features)
```

```
plt.plot(np.cumsum(pca.explained_variance_ratio_))
```

```
plt.show()
```

```
#넘파이 cumsum()를 사용하여 분산을 누적하여 그래프 출력
```

```
#대략 30개의 주성분으로도 80% 이상의 분산을 유지
```

```
# 표준화하지 않은 원본 데이터를 사용합니다.
```

```
pca.fit(digits.data)
```

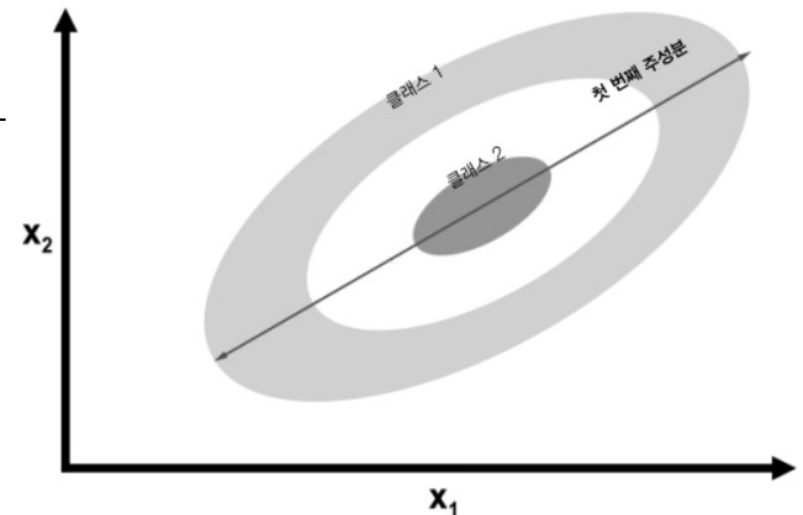
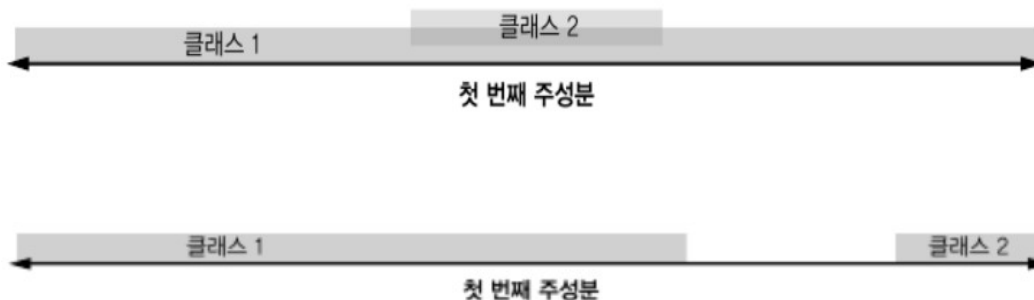
```
plt.plot(np.cumsum(pca.explained_variance_ratio_))
```

```
plt.show()
```

특성 추출을 사용한 차원 축소

➤ 선형적으로 구분되지 않는 데이터의 차원 축소

- 커널 트릭을 사용하여 주성분 분석의 확장을 사용하여 비선형 차원 축소를 수행
- 표준 PCA는 샘플이 다른 클래스 사이에 직선이나 초평면을 그릴 수 있다면 선형적으로 투영하여 특성을 축소합니다.
- 데이터가 구부러진 결정 경계를 사용해서만 클래스를 나눌 수 있다면 선형 변환이 맞지 않기 때문에 사이킷런의 `make_circles()`를 사용해 두개의 클래스를 가진 타깃 벡터와 두 개의 특성을 가진 모의 데이터셋을 만듭니다.
- `make_circles()`는 선형적으로 구분되지 않는 데이터를 만듭니다.
- 하나의 클래스가 다른 클래스 안에 둘러싸여 있습니다.
- 선형 PCA를 사용하여 데이터의 차원을 축소시킨다면 두 클래스가 첫 번째 주성분에 선형적으로 투영되기 때문에 서로 섞이게 됩니다.



특성 추출을 사용한 차원 축소

➤ 선형적으로 구분되지 않는 데이터의 차원 축소

- 커널 함수는 선형적으로 구분되지 않는 데이터를 선형적으로 구분되는 고차원으로 투영시켜줍니다. (커널 트릭)
- 사이킷런의 KernelPCA의 kernel 매개변수의 값 - rbf(가우시안 방사 기저 함수 커널), ploy(다항식 커널), sigmoid(시그모이드 커널), 선형 투영(linear)로 지정
- 여러 가지 커널과 매개변수 조합으로 머신러닝 모델을 여러 번 훈련시켜서 가장 높은 예측 성능을 만드는 값의 조합을 찾아야 합니다.
- 커널 트릭은 실제 고차원으로 데이터를 변환하지 않으면서 고차원 데이터를 다루는 듯한 효과를 냅니다.

```
from sklearn.decomposition import KernelPCA
from sklearn.datasets import make_circles

# 선형적으로 구분되지 않는 데이터를 만듭니다.
features, _ = make_circles(n_samples=1000, random_state=1, noise=0.1, factor=0.1)

# 방사 기저 함수(radius basis function, RBF)를 사용하여 커널 PCA를 적용합니다.
kpca = KernelPCA(kernel="rbf", gamma=15, n_components=1)
features_kpca = kpca.fit_transform(features)

print("원본 특성 개수:", features.shape[1])
print("줄어든 특성 개수:", features_kpca.shape[1])
```


특성 추출을 사용한 차원 축소

➤ 선형적으로 구분되지 않는 데이터의 차원 축소

- 실제 고차원 공간으로 변환하는 것은 아니기 때문에 PCA처럼 주성분을 얻을 수는 없습니다.
- kernel 매개변수의 기본값은 linear입니다.
- gamma 매개변수는 rbf, poly, sigmoid 커널에서 사용하는 계수이고 기본값은 특성 개수의 역수입니다.
- degree 매개변수는 poly 커널에 사용하는 거듭제곱 수이고 기본값은 3입니다.
- coef0 매개변수는 poly와 sigmoid 커널에 사용되는 상수항으로 기본값은 1입니다.

특성 추출을 사용한 차원 축소

➤ 클래스 분리를 최대화함으로써 특성 줄이기

- 선형 판별 분석(LDA)를 사용하여 클래스를 최대한 분리하는 성분 축으로 특성을 투영합니다.
- LDA는 분류 알고리즘이지만 차원 축소에도 자주 사용되는 기법
- LDA는 특성 공간을 저차원 공간으로 투영합니다
- PCA가 데이터에서 분산이 최대인 성분 축에만 관심이 있는 반면 LDA는 클래스 간의 차이를 최대화하는 추가적인 목적을 가집니다

```
from sklearn import datasets
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

iris = datasets.load_iris() # 붓꽃 데이터셋을 로드합니다.
features = iris.data
target = iris.target

# LDA 객체를 만들고 실행하여 특성을 변환합니다.
lda = LinearDiscriminantAnalysis(n_components=1)
features_lda = lda.fit(features, target).transform(features)

print("원본 특성 개수:", features.shape[1]) # 특성 개수를 출력합니다
print("줄어든 특성 개수:", features_lda.shape[1])

#explained_variance_ratio_를 사용하여 각 성분이 설명하는 분산의 양을 확인
lda.explained_variance_ratio_
```

특성 추출을 사용한 차원 축소

➤ 클래스 분리를 최대화함으로써 특성 줄이기

- LDA는 사이킷런의 LinearDiscriminantAnalysis 클래스로 구현
- n_components 매개변수에 원하는 특성의 개수를 지정합니다.
- 필요한 n_components의 값을 알기 위해서는 만들어진 각 특성이 설명하는 분산을 크기순으로 정렬한 explained_variance_ratio_를 참고할 수 있습니다.
- n_components를 None으로 지정하면 모든 성분 특성에 의해 설명된 분산의 비율을 반환합니다.
- LDA는 PCA와 달리 타깃 벡터를 사용합니다

```
lda = LinearDiscriminantAnalysis(n_components=None) # LDA를 만들고 실행
features_lda = lda.fit(features, target)
# 설명된 분산의 비율이 담긴 배열을 저장합니다.
lda_var_ratios = lda.explained_variance_ratio_

def select_n_components(var_ratio, goal_var: float) -> int:
    total_variance = 0.0 # 설명된 분산의 초기값을 지정
    n_components = 0 # 특성 개수의 초기값을 지정
    for explained_variance in var_ratio: # 각 특성의 설명된 분산을 순회
        total_variance += explained_variance # 설명된 분산 값을 누적
        n_components += 1 # 성분 개수를 카운트
        if total_variance >= goal_var: # 설명된 분산이 목표치에 도달하면
            break # 반복을 종료
    return n_components # 성분 개수를 반환
select_n_components(lda_var_ratios, 0.95) # 함수를 실행
```

특성 추출을 사용한 차원 축소

➤ 행렬 분해를 사용하여 특성 줄이기

- 음수가 아닌 특성 행렬이 있을 때 차원을 축소하려면 비음수 행렬 분해(non-negative matrix factorization NMF)를 사용합니다.
- 샘플과 특성 사이에 잠재되어 있는 관계를 표현하는 행렬로 특성 행렬을 분해합니다.
- 행렬 곱셈에서 행렬은 결과 행렬보다 훨씬 적은 차원을 가지기 때문에 NMF가 차원을 축소할 수 있습니다.

```
from sklearn.decomposition import NMF
from sklearn import datasets

digits = datasets.load_digits()
features = digits.data

nmf = NMF(n_components=10, random_state=1)
features_nmf = nmf.fit_transform(features)

print("원본 특성 개수:", features.shape[1])
print("줄어든 특성 개수:", features_nmf.shape[1])
```

데이터 로드
특성 행렬을 로드

NMF 생성
학습

특성 추출을 사용한 차원 축소

➤ 행렬 분해를 사용하여 특성 줄이기

- 원하는 특성 개수 r 이 주어지면 NMF는 다음과 같이 특성 행렬을 분해합니다.

$$V \approx WH$$

- V 는 $n \times d$ 크기의 특성 행렬입니다.(즉, n 개의 샘플, d 개의 특성)
- W 는 $n \times r$ 크기이고 H 는 $r \times d$ 크기 행렬입니다.
- r 값을 조절하여 필요한 차원 축소의 양을 정할 수 있습니다.
- **특성 행렬이 음수를 포함 할 수 없습니다.**
- H 행렬은 `components_` 속성에 저장
- W 행렬이 변환된 데이터 `features_nmf`
- 원본 데이터를 복원하려면 변환된 행렬 W 와 성분 행렬 H 을 곱합니다.
- NMF 클래스의 `solver` 매개변수의 기본값은 `cd`로 좌표 하강법을 사용합니다. (v0.19에서 곱셈 업데이트 알고리즘인 `mu` 옵션이 추가됨)

```
nmf.components.shape
np.all(nmf.components_ >= 0)
np.mean(features - np.dot(features_nmf, nmf.components_))

nmf_mu = NMF(n_components = 10, solver='mu', random_state=1)
features_nmf_mu = nmf_mu.fit_transform(features)
np.mean(features - np.dot(features_nmf_mu, nmf_mu.components_))
```

특성 추출을 사용한 차원 축소

➤ 희소한 데이터의 특성 줄이기

- TSVD (truncated singular value decomposition) 는 PCA와 비슷합니다.
- PCA의 단계 중 하나에서 종종 기본 SVD 방식을 사용합니다.
- 기본 SVD에서 d 개의 특성이 주어지면 SVD는 $d \times d$ 크기의 분해 행렬을 만듭니다.
- TSVD는 사전에 매개변수에서 지정한 n 으로 값 $n \times n$ 크기의 행렬을 만듭니다.
- TSVD는 PCA와 달리 희소 특성 행렬에 사용할 수 있습니다.
- TSVD는 난수 생성기를 사용하기 때문에 출력 부호가 훈련하는 사이에 뒤집힐 수 있기 때문에 전처리 파이프라인마다 한 번만 `fit()` 호출하고 그 다음 여러 번 `transform()`를 사용합니다. 선형 판별 분석처럼 `n_components`를 사용하여 필요한 특성(성분)의 개수를 지정해야 합니다.
- `n_components`를 하이퍼파라미터로 모델 선택 과정에서 최적화하는 것입니다.
- TSVD가 성분마다 원본 특성 행렬의 설명된 분산 비율을 제공하기 때문에 필요한 분산의 양을 설명할 수 있는 성분 개수를 선택할 수 있습니다.

특성 추출을 사용한 차원 축소

➤ 희소한 데이터의 특성 줄이기

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import TruncatedSVD
from scipy.sparse import csr_matrix
from sklearn import datasets
import numpy as np

digits = datasets.load_digits()
features = StandardScaler().fit_transform(digits.data)

features_sparse = csr_matrix(features)
tsvd = TruncatedSVD(n_components=10)

# 희소 행렬에 TSVD를 적용합니다.
features_sparse_tsvd = tsvd.fit(features_sparse).transform(features_sparse)

print("원본 특성 개수:", features_sparse.shape[1]) # 결과 출력
print("줄어든 특성 개수:", features_sparse_tsvd.shape[1])

# 처음 세 개의 성분이 설명하는 분산의 비율 합
tsvd.explained_variance_ratio_[0:3].sum()
```

데이터 로드
특성 행렬을 표준화 처리
희소 행렬 생성
TSVD 객체 생성

특성 추출을 사용한 차원 축소

➤ 희소한 데이터의 특성 줄이기

- 원본 특성 개수보다 하나 작게 `n_components`를 지정하고 TSVD를 실행하여 원하는 원본 데이터의 분산에서 설명된 양에 맞는 성분 개수를 계산하는 함수를 만들어 자동화 할 수 있습니다.

```
# 특성 개수보다 하나 작은 TSVD를 만들고 실행합니다.
tsvd = TruncatedSVD(n_components=features_sparse.shape[1]-1)
features_tsvd = tsvd.fit(features)

.
tsvd_var_ratios = tsvd.explained_variance_ratio_ # 설명된 분산을 리스트에 저장

def select_n_components(var_ratio, goal_var):
    total_variance = 0.0 # 설명된 분산을 초기화
    n_components = 0 # 특성 개수를 초기화
    for explained_variance in var_ratio: # 특성의 설명된 분산을 순환
        total_variance += explained_variance # 설명된 분산을 누적
        n_components += 1 # 성분 개수를 카운트
        # 설명된 분산의 목표에 도달하면 반복을 마칩니다.
    if total_variance >= goal_var:
        break
    return n_components # 성분 개수를 반환

select_n_components(tsvd_var_ratios, 0.95) # 함수 실행
```


특성 추출을 사용한 차원 축소

➤ 희소한 데이터의 특성 줄이기

- PCA는 최대 분산의 방향을 찾기 위해 원점에서 맞춘 특성 행렬의 공분산 행렬에서 고유 벡터를 찾습니다. (특성 행렬을 특이값 분해(SVD)하여 얻은 특이 벡터와 같습니다.)
- 특성 행렬을 원점에 맞추고 TSVD를 적요하면 PCA와 거의 같은 결과가 만들어집니다.
- PCA 클래스의 `svd_solver` 매개변수가 기본값 `auto`이면 샘플의 개수가 500개 이하일 때 SVD 분해를 사용하는 `full`이 됩니다.
- 500개보다 크면 랜덤 SVD를 사용하는 `randomized`입니다.
- `TruncatedSVD`의 `algorithm` 매개변수의 기본값은 랜덤 SVD를 의미하는 `randomized`입니다.

```
features = digits.data - np.mean(digits.data, axis=0)

pca = PCA(n_components=40, random_state=1)
features_pca = pca.fit_transform(features)

svd = TruncatedSVD(n_components=40, random_state=1)
features_tsvd = tsvd.fit_transform(features)

np.max(np.abs(features_pca - features_tsvd))
```