핸즈온 머신러닝 CH8

배경 : 차원의 저주 (특성이 너무 많은 문제) → 훈련 속도를 느리게 하고 좋은 솔루션 찾기 어려워

효과 : 훈련 속도 빠르게 하고 데이터 시각화에 유용 ex. 군집화의 시각화

해결 방법 : **PCA, 커널 PCA, LLE**

▼ 차원 축소를 위한 접근 방법 : 투영 / 매니폴드 학습

1. 투영
   모든 훈련 샘플이 고차원 공간 안의 저차원 부분 공간에 놓여 있는 점 고려하여,

   고차원 공간→ 저차원 공간으로 투영

2. 매니폴드
   실제 고차원 데이터셋이 더 낮은 저차원 매니폴드에 가깝게 놓여 있다는 매니폴드 가정,
   스위스 롤 : 2D 매니폴드

→ 훈련 세트의 차원 감소시키면 훈련 속도는 빨라지지만,
  항상 더 낫거나 간단한 솔루션이 되는 것은 X
→ 전적으로 데이터 셋에 달려있어

▼ PCA (주성분 분석)

: **데이터에 가장 가까운 초평면을 정의 → 데이터를 이 평면에 투영**

분산이 최대로 보존되는 축 선택
→ 정보가 가장 적게 손실되어 합리적
→ 원본 데이터셋과 투영된 것 사이의 평균 제곱 거리를 최소화 하는 축

첫 번째 축 : 분산이 최대인 축 → **1번째 주성분(PC)**
두 번째 축 : 첫 번째 축에 직교하고 남은 분산을 최대한 보존하는 축 → **2번째 주성분**
세 번째, 네 번째.....: ,,,

특잇값 분해(SVD)

:특잇값 분해로 훈련 세트의 주성분 찾아

```
X_centered=X-X.mean(axis=0) #PCA는 데이터 셋의 평균을 0이라고 가정하므로
U,s,Vt=np.linalg.svd(X_centered)
c1=Vt.T[:,0]  #첫 번째 주성분
c2=Vt.T[:,1]  #두 번째 주성분


#첫 두개의 주성분으로 정의된 평면에 훈련 세트 투영 (PCA 변환)
W2=Vt.T[:,:2]
X2D=X_centered.dot(W2)
```

```
#사이킷런 사용
from sklearn.decomposition import PCA

pca=PCA(n_components = 2)
X2D=pca.fit_transform(X)

# components_ 속성에 행렬의 전치가 담겨 있어
# 첫 번째 주성분 정의 단위 벡터는 pca.components_.T[:,0]
```

설명된 분산의 비율

```
pca.explained_variance_ratio_
```

적절한 차원의 수 선택하기

```
pca=PCA()
pca.fit(X_train)
cumsum=np.cumsum(pca.explained_variance_ratio_)
d=np.argmax(cumsum >=0.95)+1
# 충분한 분산(예를 들면 95%)이 될 때까지 더해야 할 차원의 수 선택

pca=PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train) #차원 축소한 후 적용
```

압축을 위한 PCA
: 차원 축소 후 훈련 세트의 크기 줄어들어
: 압축 된 데이터 셋을 복원하면 원본과 비슷하지만 일정량의 정보를 잃어버려 조금 손실
재구성 오차 : 원본 데이터와 재구성한(압축 후 원복한 것) 사이의 평균 제곱 거리

```
pca=PCA(n_components=154) # 데이터 셋을 154차원으로 압축
X_reduced=pca.fit_transform(X_train)
x_recovered=pca.inverse_transform(X_reduced)
```

랜덤 PCA
svd_solver="randomized" 로 지정
→ 확률적 알고리즘을 이용해 처음 d개의 주성분 근삿값 빠르게 찾아

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

점진적 PCA

PCA 의 문제: SVD 알고리즘 실행 위해 전체 훈련 세트를 메모리에 올려야
해결 : 점진적 PCA 알고리즘
→ 훈련 세트를 미니배치로 나눈 후 IPCA 알고리즘에 한 번에 하나씩 주입

```
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):
  inc_pca.partial_fit(X_batch)

X_reduced=inc_pca.transform(X_train)
```

▼ 커널 PCA

```
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.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)
```

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

```
# 재구성 원상의 오차를 최소화하는 커널과 하이퍼파라미터를 선택
rbf_pca = KernelPCA(n_components=2, kernel="rbf", gamma=0.0433,
                    fit_inverse_transform=True)
X_reduced = rbf_pca.fit_transform(X)
X_preimage = rbf_pca.inverse_transform(X_reduced)
```

```
from sklearn.metrics import mean_squared_error

mean_squared_error(X, X_preimage)
```

▼ LLE (지역 선형 임베딩)

```
# 지역 선형 임베딩
from sklearn.manifold import LocallyLinearEmbedding

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

▼ 다른 차원 축소 기법

- 랜덤 투영
- 다차원 스케일링
- Isomap
- t-SNE
- LDA