차원 축소

차원의 저주 (Curse of dimension)

입력 데이터 수보다 데이터의 차원이 더 큰 경우 발생하는 문제이다. 데이터가 무수히 큰 벡터 공간 내에 마구 흩뿌려져 있으므로 분류 예측하는 모델의 복잡도가 높아지고, 성능은 낮아지는 문제가 발생한다.

따라서, 차원의 저주 문제를 해결하고자 하는 방법이 "차원 축소"이며 변수 선택, 변수 추출 등으로 구현할 수 있다

- 변수 선택 : 종속변수에 가장 영향을 미치는 독립변수를 선택해 차원 축소
- 변수 추출 : 압축(변수 조합)을 통해 새로운 변수를 만들어 차원 축소

관련 용어

- Sparsity, Sparse Matrix (분산행렬)
- Feature Selection : Multicollinearity (다중공선성), Forward Selection, Backward Elimination, Stepwise Selecction
- Feature Extraction : PCA, SVD, MF

1. 다중공선성 (Multicollinearity)

본래 서로 독립이어야 할 독립변수들 중 일부가 종속 관계(상호 상관관계)인 경우 발생한다. 독립변수의 일부가 다른 독립변수의 조합으로 표현될 수 있다.

- 독립변수가 늘어날수록 결과값의 Variance가 커지는 문제가 발생한다.
- 독립변수의 공분산행렬이 Full Rank여야 한다는 조건을 불만족한다.

결정계수 (R-Square) 관련 개념

1. 오차 (Error) : "모집단"에서 추정된 회귀선에서의 예측값과 실제 관측값의 차이

① 고정요소, ② 확률적 요소로 구분된다.

2. 잔차 (Residual) : "표본"에서 추정한 회귀선에서의 예측값과 실제 관측값의 차이

$$Residual(e) = y - \hat{y}$$

3. TSS (Total Sum of Square) : 종속변수 값의 변동 범위

$$TSS = \sum (y_i - \bar{y})^2$$

4. ESS (Explained Sum of Square) : 예측치의 변동 범위

$$ESS = \sum (\hat{y} - \bar{y})^2$$

혹은 다음과 같이 나타낼 수도 있다.

$$ESS = \sum (\hat{y} - \bar{\hat{y}})^2$$

회귀모형의 가정이 지켜지고 있다면

$$E(r_i) = 0$$
$$\bar{y} - \bar{\hat{y}} = 0$$

즉, 두 식이 결과적으로 같다. (종속변수의 평균과 모형 예측값의 평균이 같다.)

5. RSS (Residual Sum of Square) : 잔차의 변동 범위

$$RSS = \sum (y_i - \hat{y})^2$$

6. 결정계수 (R-Square)

$$R^2 = 1 - \frac{RSS}{TSS} = \frac{ESS}{TSS}$$

7. 수정결정계수 (Adjusted R-Square)

$$Adj.\ R^2 = 1 - \frac{n-1}{(n-p-1)(1-R^2)}$$

설명력이 지나치게 높은 상황이다.

독립변수 간의 높은 상호의존성으로 인하여 Overfitting(과적합)이 일어났음을 알 수 있다.

이를 방지하는 방법은 다음과 같다.

- 변수선택법
- PCA
- 정규화 (Regularization) : 독립변수 간 의존성이 높은 변수에 Penalty 부과

2. VIF (Variance Inflation Factor, 분산팽창계수)

다중공선성 측정 척도 중 하나이다.

$$VIF_i = \frac{\sigma^2}{(n-1)Var[X_i]} \cdot \frac{1}{1-R_i^2} = \frac{1}{1-R_i^2}$$

(R_i : X_i가 종속변수이고 나머지 독립변수들이 독립변수 역할을 하는 회귀식)

- $VIF > 10$: 다중공선성 존재
- 해당 i번째 독립변수가 제외되어도 나머지 변수들이 반응변수 y를 충분히(90% 이상) 설명하므로 모형에서 필수적인 독립변수가 아니라는 뜻이다.

모든 변수 다 VIF가 10보다 크지만, 상관계수와 VIF 두 가지를 모두 고려하여 독립변수를 선택한다.

→ GNP, ARMED, UNEMP 선택 시, 세 가지 변수만으로도 비슷한 수준의 성능이 나오는 것을 확인할 수 있다.

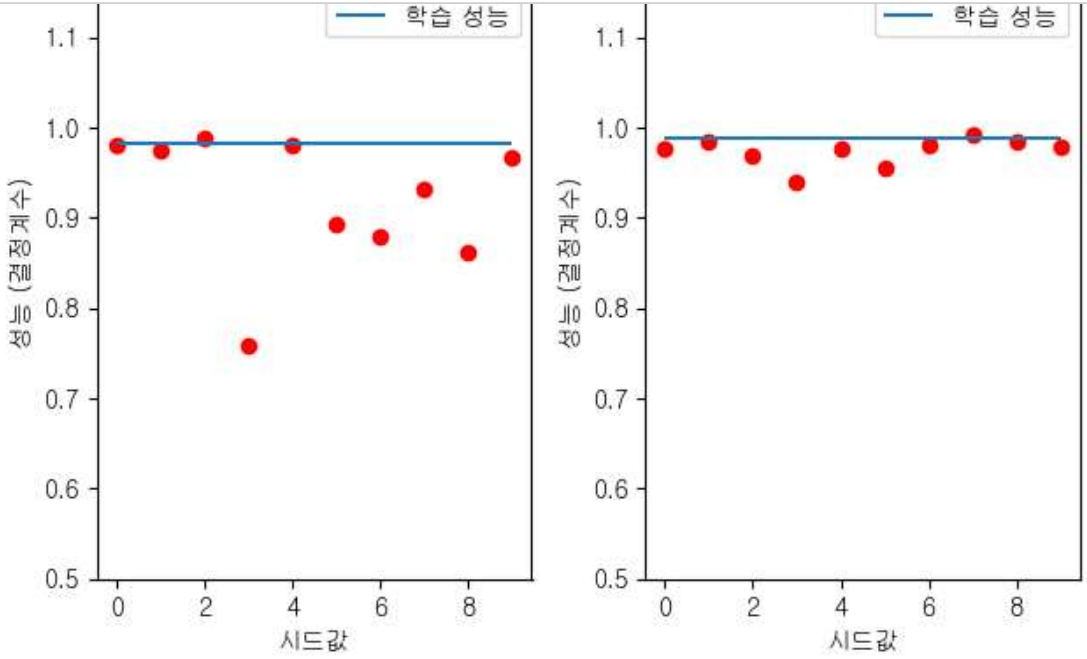
다중공선성 제거 후 Train set과 Test set 간 성능 차이가 줄어들었다. Overfitting가 발생하지 않는다.

In [36]:

```
plt.subplot(121)
plt.plot(test_1, 'ro', label = "검증 성능")
plt.hlines(result.rsquared, 0, 9, label = "학습 성능")
plt.legend()
plt.xlabel("시드값")
plt.ylabel("성능 (결정계수)")
plt.title("다중공선성 제거 전")
plt.ylim(0.5, 1.2)

plt.subplot(122)
plt.plot(test_2, 'ro', label = "검증 성능")
plt.hlines(result_2.rsquared, 0, 9, label = "학습 성능")
plt.legend()
plt.xlabel("시드값")
plt.ylabel("성능 (결정계수)")
plt.title("다중공선성 제거 후")
plt.ylim(0.5, 1.2)

plt.suptitle("다중공선성 제거 전과 제거 후의 성능 비교", y = 1.04)
plt.tight_layout()
plt.show()
```



3. Boston 집값 예측 문제에 적용해보기

조건수 (Condition Number) ?

Train Data의 성능을 희생하더라도 Test Data의 성능이 더 잘 나올 수 있도록 하는 방향으로 분석 진행.

y = f(x)의 조건수는 x의 작은 변화에 대한 함수의 민감도를 측정하는 지표이다.

조건수의 감소 목적

- 독립변수의 절대적 수치 크기나 서로 간의 의존도가 분석결과에 미치는 영향을 줄이고, 독립변수의 상대적인 비교효과를 반영한다.
- 공분산행렬의 변동성을 줄여 분석 결과의 변동을 줄인다.

입력값에 따른 예측값의 변동을 나타내므로, 조건수가 크면 x값의 미미한 변화에도 y 예측치의 변동이 매우 커지므로 값을 신뢰하지 못하게 된다. 반면 조건수가 작으면 x값에 미미한 변화가 있었을 경우 기존에 예측한 y값과 거의 차이가 없도록 산출된다.

즉, 조건수가 작을수록 Overfitting 확률이 낮다!

조건수를 낮추는 방법

- Scaling : 변수 단위 차이 제거
- 다중공선성 제거
- 독립변수 간 의존성이 높은 변수에 Penalty 부과

https://dsbook.tistory.com/297 (https://dsbook.tistory.com/297) https://pasus.tistory.com/103 (https://pasus.tistory.com/103) (Condition Number의 수학적 이해) https://pasus.tistory.com/34 (https://pasus.tistory.com/34) (Norm)

In [28]:

```
## 조건수의 이해
# 1. 조건수가 작을 때
mat_a = np.eye(4)
mat_a
```

Out[28]:

```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

In [3]:

```
mat_y = np.ones(4)
mat_y
```

Out[3]:

```
array([1., 1., 1., 1.])
```

In [5]:

```
np.linalg.solve(mat_a, mat_y)
    # np.linalg.solve : 연립방정식의 해
    # np.linalg.inv : 역행렬
    # np.dot : 행렬곱 (내적)
```

Out[5]:

```
array([1., 1., 1., 1.])
```

In [8]:

```
# x 데이터 오차 반영
mat_a_new = mat_a + 0.0001 * np.eye(4)
mat_a_new
```

Out[8]:

```
array([[1.0001, 0.    , 0.    , 0.    ],
       [0.    , 1.0001, 0.    , 0.    ],
       [0.    , 0.    , 1.0001, 0.    ],
       [0.    , 0.    , 0.    , 1.0001]])
```

In [9]:

```
np.linalg.solve(mat_a_new, mat_y)
```

Out[9]:

```
array([0.99990001, 0.99990001, 0.99990001, 0.99990001])
```

In [10]:

```
np.linalg.cond(mat_a)
    # np.linalg.cond : 조건수
```

Out[10]:

```
1.0
```

In [14]:

```
# 조건수가 클 때
from scipy.linalg import hilbert

mat_a_big = hilbert(4)
mat_a_big
```

Out[14]:

```
array([[1.        , 0.5       , 0.33333333, 0.25      ],
       [0.5       , 0.33333333, 0.25      , 0.2       ],
       [0.33333333, 0.25      , 0.2       , 0.16666667],
       [0.25      , 0.2       , 0.16666667, 0.14285714]])
```

In [15]:

```
np.linalg.solve(mat_a_big, mat_y)
```

Out[15]:

```
array([  -4.,   60., -180.,  140.])
```

In [16]:

```
mat_a_big_new = mat_a_big + 0.0001 * np.eye(4)
mat_a_big_new
```

Out[16]:

```
array([[1.0001    , 0.5       , 0.33333333, 0.25      ],
       [0.5       , 0.33343333, 0.25      , 0.2       ],
       [0.33333333, 0.25      , 0.2001    , 0.16666667],
       [0.25      , 0.2       , 0.16666667, 0.14295714]])
```

In [17]:

```
np.linalg.solve(mat_a_big_new, mat_y)
```

Out[17]:

```
array([ -0.58897672,  21.1225671 , -85.75912499,  78.45650825])
```

In [19]:

```
np.linalg.cond(mat_a_big)
```

Out[19]:

```
15513.738738929103
```