

교차검증

교차검증 Hold-out

학습(training) 데이터 일부를 검증(validation) 데이터로 사용하는 방법을 홀드아웃(Hold-out) 교차 검증이라고 부른다. 검증 데이터는 모델 학습에 사용되지 않은 데이터이므로 모델의 일반화 성능을 평가하는데 사용한다. 결과적으로 테스트 데이터에 대한 예측력을 높일 수 있다.



사이킷런의 `train_test_split` 메소드를 사용하여 기존 학습 데이터 중에서 30%를 검증 데이터(X_{val} , y_{val})로 분리하고, 나머지 70%를 훈련용 데이터(X_{tr} , y_{tr})로 분할한다.

교차검증 Hold-out

```
[40] # 검증용 데이터셋 분리
```

```
X_tr, X_val, y_tr, y_val = train_test_split(X_train, y_train,  
                                             test_size=0.3,  
                                             shuffle=True,  
                                             random_state=20)  
  
print(X_tr.shape, y_tr.shape)  
print(X_val.shape, y_val.shape)
```

```
➡ (83, 4) (83,)  
   (36, 4) (36,)
```

Tip shuffle 옵션을 True로 설정하면 데이터를 랜덤하게 섞은 다음 분리 추출하게 된다. 예측 모델이 데이터 순서와 무관하게 일반화된 성능을 갖는지 확인할 수 있다.

랜덤 포레스트 모델에 훈련 데이터(X_{tr} , y_{tr})를 입력하여 학습하고, 검증 데이터(X_{val} , y_{val})를 사용하여 모델의 성능을 평가한다. 검증 정확도가 훈련 정확도보다 상당히 낮기 때문에 훈련 데이터에 과대적합이 발생되었다고 볼 수 있다. 즉, 새로운 데이터에 대한 성능이 떨어진다.

교차검증 Hold-out

```
[41] # 학습
rfc = RandomForestClassifier(max_depth=3, random_state=20)
rfc.fit(X_tr, y_tr)

# 예측
y_tr_pred = rfc.predict(X_tr)
y_val_pred = rfc.predict(X_val)

# 검증
tr_acc = accuracy_score(y_tr, y_tr_pred)
val_acc = accuracy_score(y_val, y_val_pred)

print('Train Accuracy:%.4f' % tr_acc)
print('Validation Accuracy:%.4f' % val_acc)
```

```
➡ Train Accuracy:0.9880
Validation Accuracy:0.9167
```

교차검증 Hold-out

```
[42] # 테스트 데이터 예측 및 평가
```

```
    y_test_pred = rfc.predict(X_test)
```

```
    test_acc = accuracy_score(y_test, y_test_pred)
```

```
    print("Test Accuracy: %.4f" % test_acc)
```

```
➡ Test Accuracy:0.9000
```

테스트 데이터(X_test)를 predict 메소드에 입력하여 예측한다. 테스트 정확도가 0.9000으로 검증 정확도보다 낮은 수준이다. 훈련 데이터에 과대적합하여 새로운 데이터에 대한 예측력 또한 낮은 편이다.

교차 검증 K-fold

홀드아웃 검증에서는 학습 데이터를 훈련용과 검증용으로 한번 분할하여 모델 성능을 검증한다. K-fold 교차 검증은 홀드아웃 방법을 여러 번 반복하는 방법이다.



랜덤 포레스트 모델을 K-fold 교차 검증으로 평가한다.

교차 검증 K-fold

```
[44] # 훈련용 데이터와 검증용 데이터의 행 인덱스를 각 fold별로 구분하여 생성
val_scores = []
num_fold = 1
for tr_idx, val_idx in kfold.split(X_train, y_train):
    # 훈련용 데이터와 검증용 데이터를 행 인덱스 기준으로 추출
    X_tr, X_val = X_train.iloc[tr_idx, :] X_train.iloc[val_idx, :]
    y_tr, y_val = y_train.iloc[tr_idx], y_train.iloc[val_idx]
    # 학습
    rfc = RandomForestClassifier(max_depth=5, random_state=20)
    rfc.fit(X_tr, y_tr)
```

교차 검증 K-fold

```
# 검증
y_val_pred = rfc.predict(X_val)
val_acc = accuracy_score(y_val, y_val_pred)
print('%d Fold Accuracy:%.4f' % (num_fold, val_acc))
val_scores.append(val_acc)
num_fold += 1
```

```
➡ 1 Fold Accuracy:0.8750
   2 Fold Accuracy:1.0000
   3 Fold Accuracy:0.9167
   4 Fold Accuracy:0.9583
   5 Fold Accuracy:0.9565
```

5개 폴드의 검증 정확도를 평균한다.

교차 검증 K-fold

```
[45] # 평균 Accuracy 계산
```

```
import numpy as np
```

```
mean_score = np.mean(val_scores)
```

```
print("평균 검증 Accuracy:", np.round(mean_score, 4))
```

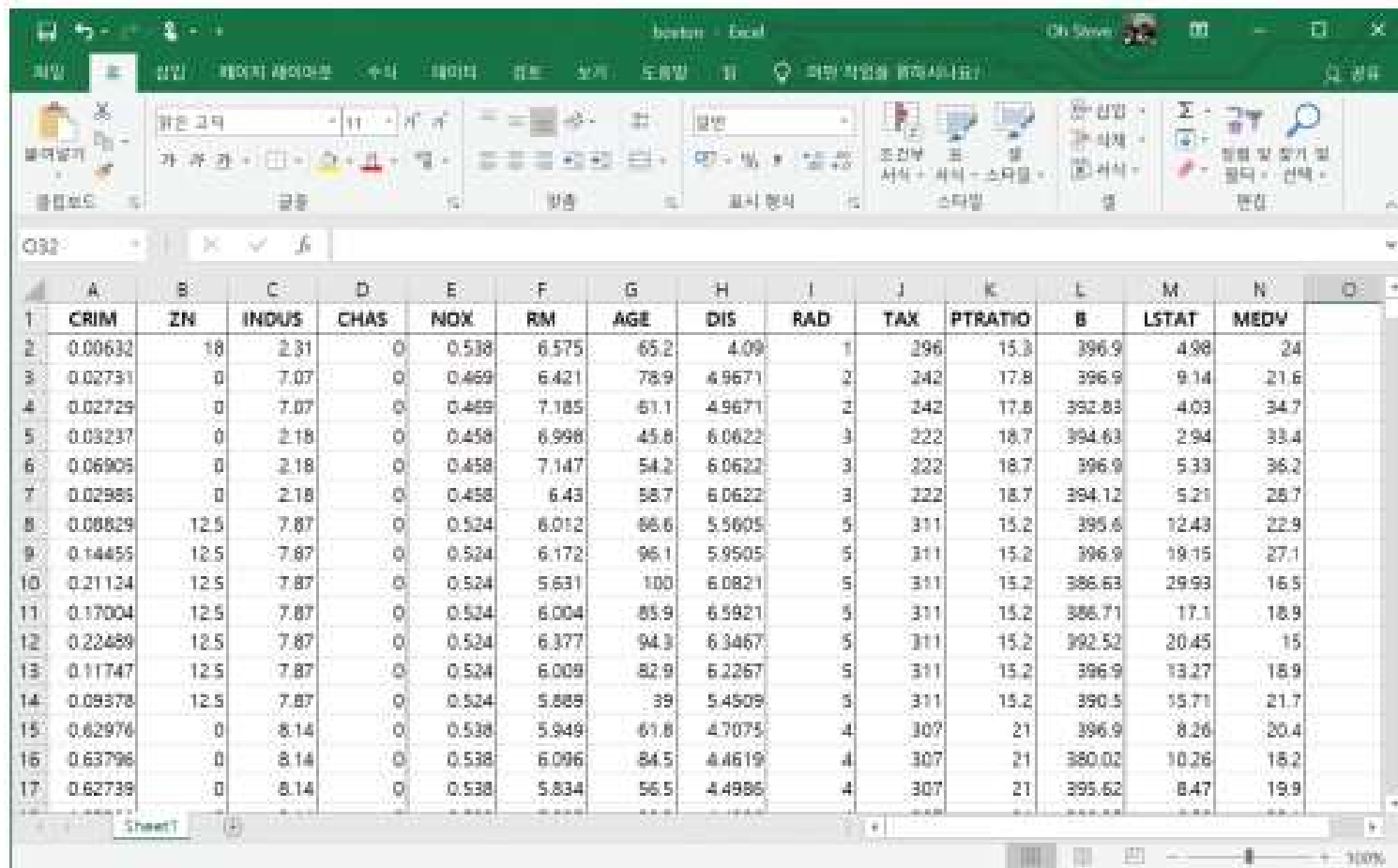
```
➡ 평균 검증 Accuracy:0.9413
```

회귀(Regression)

보스턴 주택 가격 예측

데이터 로딩

14개의 변수(열:column)로 구성된다. 각 열은 주택의 속성을 나타내는 피처(feature)를 말하고, 각 행은 개별 주택에 대한 데이터의 집합(레코드)을 나타낸다.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV	
2	0.00632	18	2.31	0	0.538	6.575	65.2	4.09	1	296	15.3	396.9	4.98	24	
3	0.02731	0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.9	9.14	21.6	
4	0.02729	0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7	
5	0.03237	0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4	
6	0.06905	0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.9	5.33	36.2	
7	0.02986	0	2.18	0	0.458	6.43	58.7	6.0622	3	222	18.7	394.12	5.21	28.7	
8	0.08829	12.5	7.87	0	0.524	6.012	68.6	5.9505	5	311	15.2	395.6	12.43	22.9	
9	0.14455	12.5	7.87	0	0.524	6.172	96.1	5.9505	5	311	15.2	396.9	18.15	27.1	
10	0.21124	12.5	7.87	0	0.524	5.631	100	6.0821	5	311	15.2	386.63	29.93	16.5	
11	0.17004	12.5	7.87	0	0.524	6.004	85.9	6.5921	5	311	15.2	386.71	17.1	18.9	
12	0.22489	12.5	7.87	0	0.524	6.377	94.3	6.3467	5	311	15.2	392.52	20.45	15	
13	0.11747	12.5	7.87	0	0.524	6.009	82.9	6.2267	5	311	15.2	396.9	13.27	18.9	
14	0.09378	12.5	7.87	0	0.524	5.889	39	5.4509	5	311	15.2	390.5	15.71	21.7	
15	0.62976	0	8.14	0	0.538	5.949	61.8	4.7075	4	307	21	396.9	8.26	20.4	
16	0.63796	0	8.14	0	0.538	6.096	84.5	4.4619	4	307	21	380.02	10.26	18.2	
17	0.62739	0	8.14	0	0.538	5.834	56.5	4.4986	4	307	21	395.62	8.47	19.9	

데이터 로딩

마지막 MEDV 속성이 주택 가격을 나타내며 회귀 모델을 설계할 때 최종 예측 대상인 목표 변수가 된다.

변수	설명
CRIM	해당 지역의 1인당 범죄 발생률
ZN	면적 25,000평방피트를 넘는 주택용 토지의 비율
INDUS	해당 지역의 비소매 상업 지역 비율
CHAS	해당 부지의 찰스강 인접 여부(인접한 경우 1, 그렇지 않은 경우 0)
NOX	일산화질소 농도
RM	(거주 목적의)방의 개수
AGE	1940년 이전에 건축된 자가 주택의 비율
DIS	보스턴의 5대 고용 지역까지의 거리
RAD	고속도로 접근성
TAX	재산세
PTRATIO	교사-학생 비율
B	흑인 거주비율
LSTAT	저소득층 비율
MEDV	소유주 거주 주택의 주택 가격(중간값)

데이터 로딩

(소스) 3_4_boston_regression.ipynb

[1] # 기본 라이브러리

```
import pandas as pd
```

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
import seaborn as sns
```

```
# sklearn 데이터셋에서 보스턴 주택 데이터셋 로딩
```

```
from sklearn import datasets
```

```
housing = datasets.load_boston()
```

```
# 딕셔너리 형태로 key 값 확인
```

```
housing.keys()
```

```
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'filename'])
```

데이터 로딩

housing 데이터의 'data' 키를 이용하여 피쳐(설명 변수) 데이터를 가져와 데이터프레임으로 변환 한다.

```
[2] # 판다스 데이터프레임으로 변환
```

```
data = pd.DataFrame(housing['data'] columns=housing['feature_names'])
```

```
target = pd.DataFrame(housing['target'] columns=['Target'])
```

```
# 데이터셋 크기
```

```
print(data.shape)
```

```
print(target.shape)
```

```
➡ (506, 13)
```

```
(506, 1)
```

13개 설명 변수가 담긴 data 데이터프레임과 목표 변수 데이터를 갖는 target 데이터프레임을 결 합한다.

데이터 로딩

[3] # 데이터프레임 결합

```
df = pd.concat([data, target], axis=1)
```

```
df.head(2)
```

➡ data 속성의 배열 크기: (506, 13)

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTATIO	B	LSTAT	Target
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.9	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.9	9.14	21.6

[그림 3-40] 데이터프레임 내용 확인

데이터 탐색 - 기본정보

info 함수를 사용하여 데이터프레임의 기본 정보를 확인한다.

[4] # 데이터프레임의 기본 정보

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>  
RangeIndex:506 entries, 0 to 505  
Data columns (total 14 columns):  
#   Column      Non-Null Count  Dtype  
---  ---  
0   CRIM        506 non-null    float64  
1   ZN          506 non-null    float64  
2   INDUS       506 non-null    float64  
3   CHAS        506 non-null    float64  
4   NOX         506 non-null    float64  
5   RM          506 non-null    float64  
6   AGE         506 non-null    float64  
7   DIS         506 non-null    float64  
8   RAD         506 non-null    float64  
9   TAX         506 non-null    float64  
10  PTRATIO     506 non-null    float64  
11  B           506 non-null    float64  
12  LSTAT       506 non-null    float64  
13  Target      506 non-null    float64  
dtypes:float64(14)  
memory usage:55.5 KB
```


데이터 탐색 - 결측값 확인

데이터프레임 각 열의 결측값(missing value) 개수를 확인한다.

```
[5] # 결측값 확인
```

```
df.isnull().sum()
```

CRIM	0
ZN	0
INDUS	0

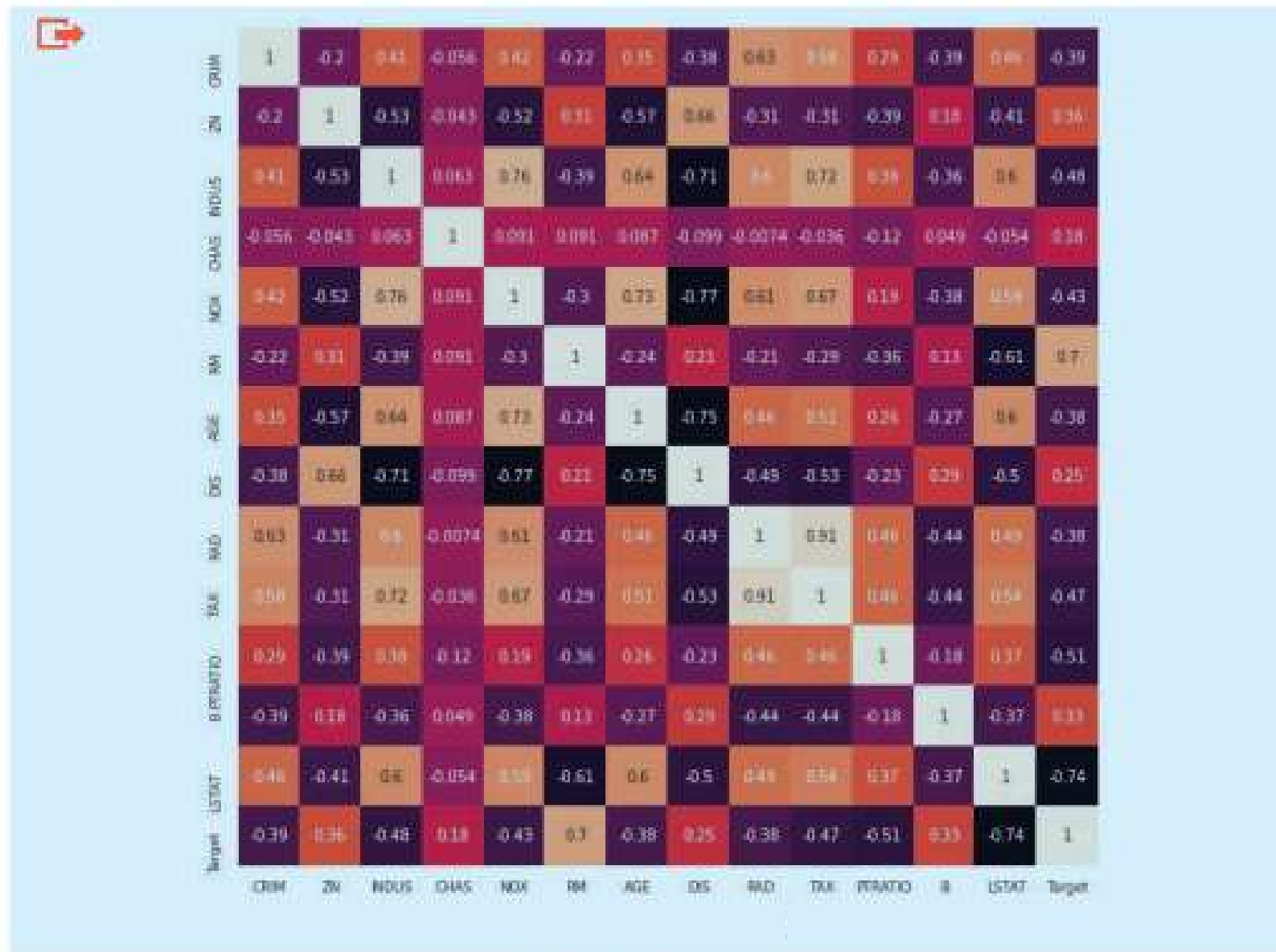
데이터 탐색 – 상관 관계 분석

변수 간의 상관 계수를 구하고, 시각화 분석한다.

```
[6] # 상관 관계 행렬
    df_corr = df.corr()

    # 히트맵 그리기
    plt.figure(figsize=(10, 10))
    sns.set(font_scale=0.8)
    sns.heatmap(df_corr, annot=True, cbar=False);
    plt.show()
```

데이터 탐색 - 상관 관계 분석




Tip cbar 옵션을 True로 바꾸고 실행해 보면 색상표가 오른쪽에 표시된다.

데이터 탐색 – 상관 관계 분석

목표 변수인 Target 열과 상관 계수가 높은 순서대로 열 이름과 상관 계수를 출력한다.

[7] # 변수 간의 상관 관계 분석 – Target 변수와 상관 관계가 높은 순서대로 정리

```
corr_order = df.corr().loc[:, 'Target'].abs().sort_values(ascending=False)  
corr_order
```



LSTAT	0.737663
RM	0.695360
PTRATIO	0.507787
INDUS	0.483725
TAX	0.468536
NOX	0.427321
CRIM	0.388305
RAD	0.381626
AGE	0.376955
ZN	0.360445

데이터 탐색 - 상관 관계 분석

데이터 분포를 파악하는 좋은 방법은 그래프를 그려보는 것이다. Target 변수와 함께 상관 계수가 높은 순서대로 4개 피처(LSTAT, RM, PTRATIO, INDUS)를 추출한다.

```
[8] # 시각화로 분석할 피처 선택 추출 - Target 변수와 상관 관계가 높은 4개 변수
plot_cols = ['Target', 'LSTAT', 'RM', 'PTRATIO', 'INDUS']
plot_df = df.loc[:, plot_cols]
plot_df.head()
```



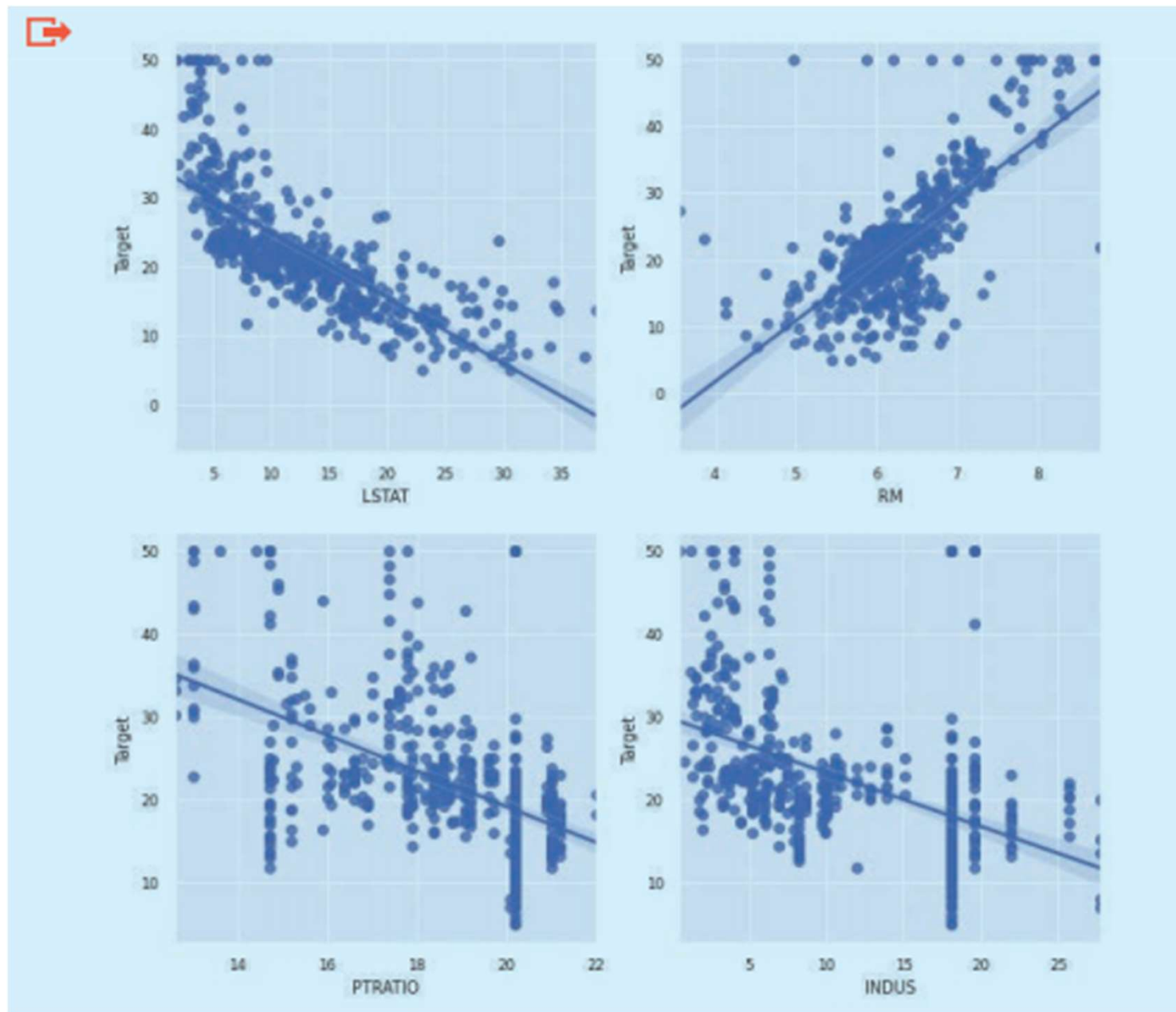
	Target	LSTAT	RM	PTRATIO	INDUS
0	240	4.98	6.575	15.3	2.31
1	216	9.14	6.421	17.8	7.07
2	347	4.03	7.185	17.8	7.07
3	334	2.94	6.996	18.7	2.18
4	362	5.33	7.147	18.7	2.18

데이터 탐색 – 상관 관계 분석

시본 regplot 함수로 선형 회귀선을 산점도에 표시한다.

```
[9] # regplot으로 선형회귀선 표시
plt.figure(figsize=(10,10))
for idx, col in enumerate(plot_cols[1:]):
    ax1 = plt.subplot(2, 2, idx+1)
    sns.regplot(x=col, y=plot_cols[0], data=plot_df, ax=ax1)
plt.show()
```

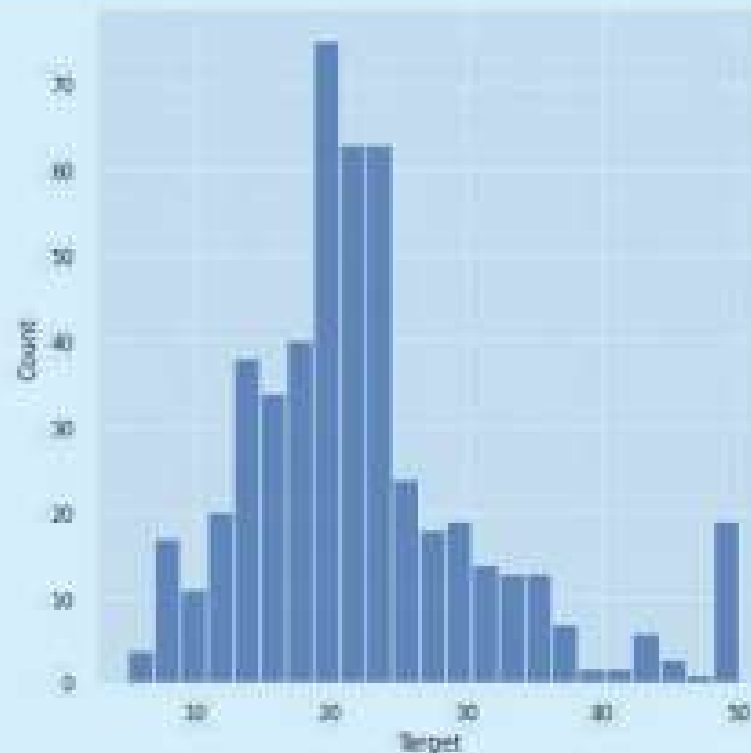
데이터 탐색 - 상관 관계 분석



데이터 탐색 - 목표 변수 (Target 열) - 주택 가격

Target 열의 주택 가격 데이터 분포를 `displot` 함수로 그린다. `kind` 속성을 'hist'로 지정하면 히스토그램을 그린다. 한편, 'kde'로 지정하면 KDE 밀도함수 그래프를 그린다.

```
[10] # Target 데이터 분포  
sns.displot(x='Target', kind='hist', data=df)  
plt.show()
```



데이터 전처리 – 피쳐 스케일링

각 피쳐(열)의 데이터 크기에 따른 상대적 영향력의 차이를 제거하기 위하여 피쳐의 크기를 비슷 한 수준으로 맞춰주는 작업이 필요하다. 이 과정을 피쳐 스케일링(Feature Scaling)이라고 부른다.

사이킷런 MinMaxScaler를 활용한다.

데이터 전처리 - 피쳐 스케일링

```
[11] # 사이킷런 MinMaxScaler 적용

from sklearn.preprocessing import MinMaxScaler
scaler=MinMaxScaler()

df_scaled = df.iloc[:, :-1]
scaler.fit(df_scaled)
df_scaled = scaler.transform(df_scaled)

# 스케일링 변환된 값을 데이터프레임에 반영
df.iloc[:, :-1] = df_scaled[:, :]
df.head()
```



	CRIM	ZN	INDUS	CHAS	NOX	RM	ASE	DIS	RAD	TAX	PTBATIO	B	LSTAT	Target
0	0.000000	0.18	0.067815	0.0	0.314815	0.577505	0.641607	0.261200	0.000000	0.208015	0.207254	1.000000	0.089680	24.0
1	0.000236	0.00	0.242302	0.0	0.172640	0.547998	0.782698	0.348962	0.043478	0.104962	0.553191	1.000000	0.204470	21.6
2	0.000236	0.00	0.242302	0.0	0.172640	0.694388	0.599362	0.348962	0.043478	0.104962	0.553191	0.989737	0.063466	34.7
3	0.000293	0.00	0.063050	0.0	0.150206	0.658555	0.441813	0.448545	0.066957	0.066794	0.648936	0.994276	0.033389	13.4
4	0.000705	0.00	0.063050	0.0	0.150206	0.487105	0.528321	0.448545	0.088957	0.066794	0.648936	1.000000	0.099338	36.2

Tip loc 인덱서는 인덱스 이름을 사용하지만, iloc 인덱서는 원소의 순서를 나타내는 정수 인덱스(0, 1, 2, ...)를 사용한다. 즉, iloc 인덱서는 행과 열의 위치 순서만 고려한다. loc 인덱서는 범위의 끝을 포함하지만, iloc 인덱서는 포함하지 않는다. 따라서 df.iloc[:, :-1]은 가장 마지막 열인 Target을 제외한 나머지 13개 열을 추출한다.

베이스라인 모델 - 선형 회귀

LinearRegression 클래스 객체를 생성 하고, fit 메소드에 학습 데이터(X_train, y_train)를 입력한다.

```
[13] # 선형 회귀 모델
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
lr.fit(X_train, y_train)

print ("회귀계수(기울기):", np.round(lr.coef_, 1))
print ("상수항(절편):", np.round(lr.intercept_, 1))
```

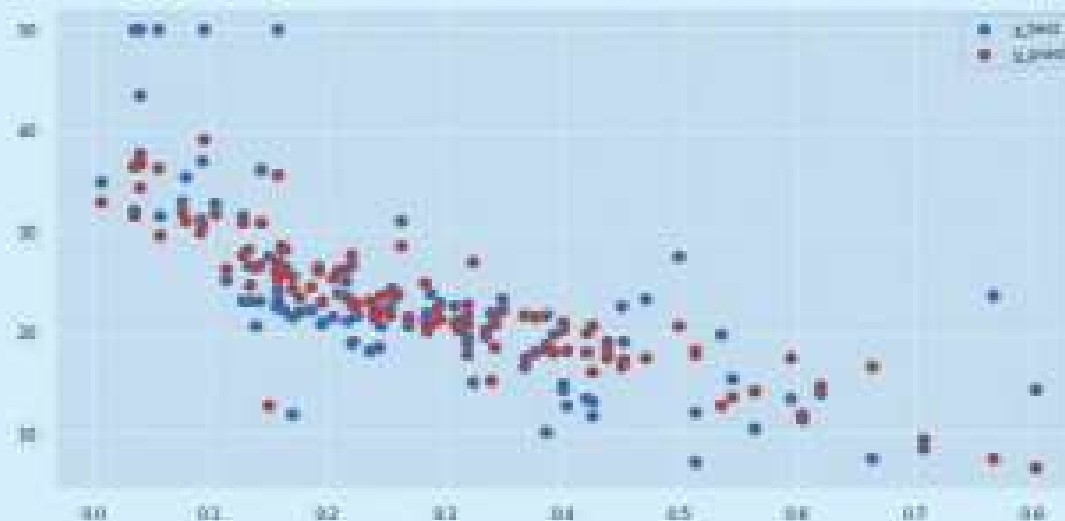


```
회귀계수(기울기):[-23.2 25.4]
상수항(절편):16.3
```

predict 함수에 테스트 데이터(X_test)를 입력하면 목표 변수(Target)에 대한 예측값을 얻는다. 예측값을 y_test_pred에 저장하고, 실제값인 y_test와 함께 산점도로 그려 비교한다. 맷플롯립의 scatter 함수를 이용한다.

베이스라인 모델 - 선형 회귀

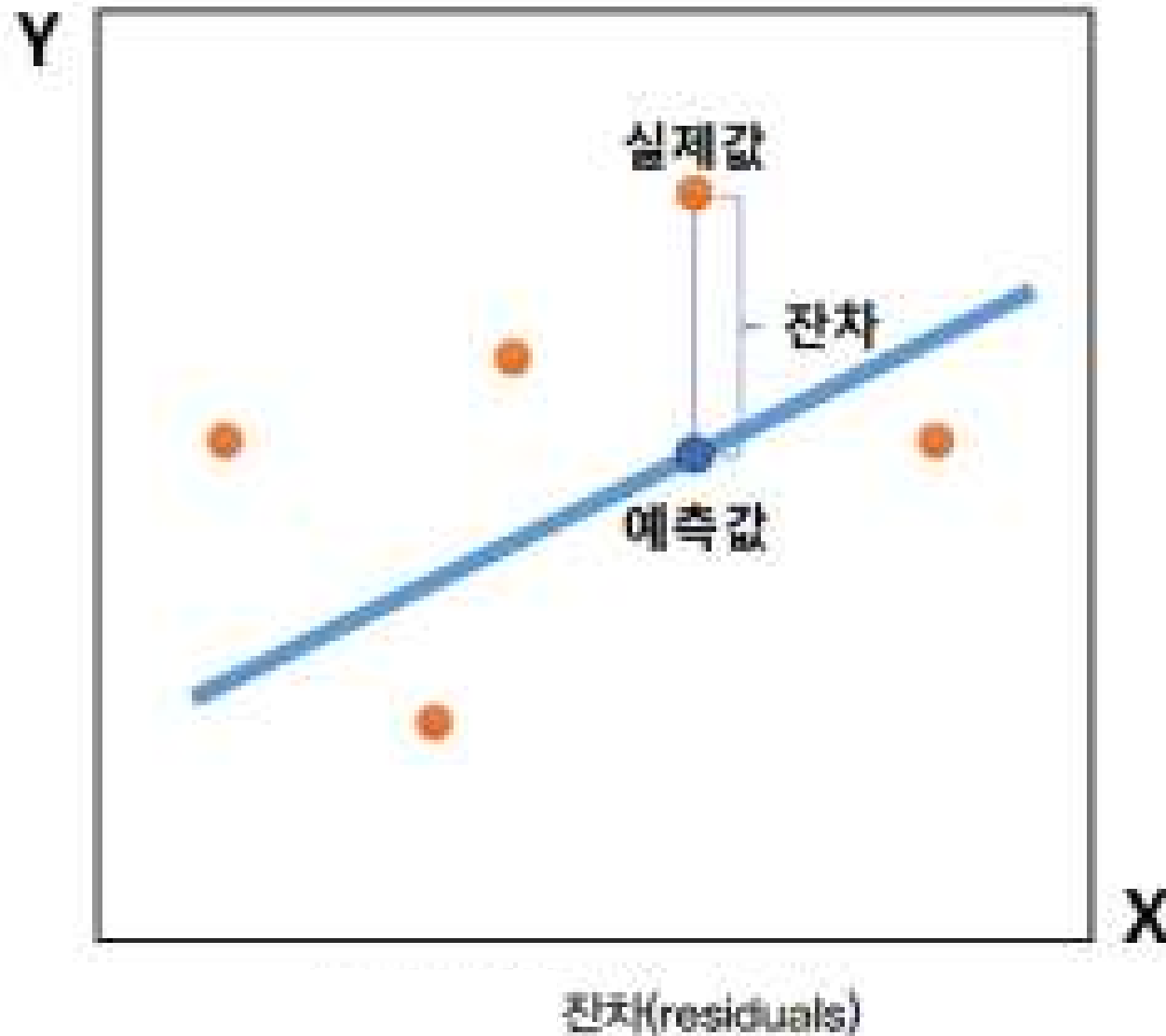
```
[14] # 예측  
y_test_pred = lr.predict(X_test)  
  
# 예측값, 실제값의 분포  
plt.figure(figsize=(10, 5))  
plt.scatter(X_test['LSTAT'], y_test, label='y_test')  
plt.scatter(X_test['LSTAT'], y_test_pred, c='r', label='y_pred')  
plt.legend(loc='best')  
plt.show()
```



X_test 검증 데이터에 대한 예측값(y_pred)과 실제값(y_test) 분포

모델 성능 평가

실제값(정답)과 예측값의 차이를 잔차(residuals)라고 한다.



모델 성능 평가

회귀모델의 성능을 평가하는 지표는 RMSE 등이 있다.

구분	수식	설명
MAE(Mean Absolute Error)	$\frac{1}{n} \sum_{i=1}^n Y_i - \hat{Y}_i $	실제값(Y_i)과 예측값(\hat{Y}_i)의 차이, 즉 잔차의 절대값을 평균한 값
MSE(Mean Squared Error)	$\frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$	실제값(Y_i)과 예측값(\hat{Y}_i)의 차이, 즉 잔차의 제곱을 평균한 값
RMSE(Root Mean Squared Error)	$\sqrt{\frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2}$	MSE의 제곱근

회귀 모형의 성능 평가 지표

mean_squared_error 함수로 계산한다.

모델 성능 평가

```
[15] # 평가  
  
from sklearn.metrics import mean_squared_error  
y_train_pred = lr.predict(X_train)  
  
train_mse = mean_squared_error(y_train, y_train_pred)  
print("Train MSE:%.4f" % train_mse)  
  
test_mse = mean_squared_error(y_test, y_test_pred)  
print("Test MSE:%.4f" % test_mse)
```

```
➡ Train MSE:30.8042  
Test MSE:29.5065
```

cross_val_score 함수를 이용하여 K-Fold 교차 검증을 간단하게 수행할 수 있다.

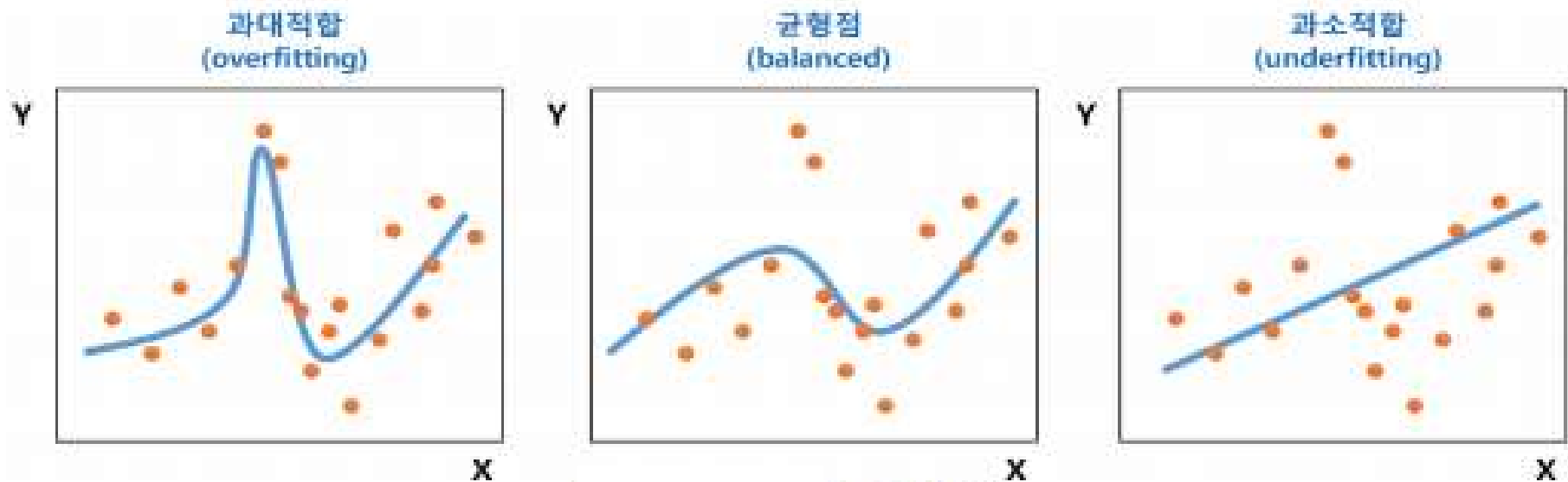
모델 성능 평가

```
[16] # cross_val_score 함수  
from sklearn.model_selection import cross_val_score  
lr = LinearRegression()  
mse_scores = -1*cross_val_score(lr, X_train, y_train, cv=5,  
                                scoring='neg_mean_squared_error')  
print("개별 Fold의 MSE:", np.round(mse_scores, 4))  
print("평균 MSE: %.4f" % np.mean(mse_scores))
```

```
➡ 개별 Fold의 MSE:[31.465  34.668  28.9147 29.3535 34.6627]  
평균 MSE:31.8128
```


과대적합 회피 – 과대적합 vs 과소적합

과대적합(overfitting)은 모델이 학습에 사용한 데이터와 비슷한 데이터는 잘 예측하지만, 새로운 특성을 갖는 데이터에 대해서는 예측력이 떨어지는 현상을 말한다. 한편, 과소적합(underfitting)은 훈련 데이터의 특성을 파악하기 충분하지 않을 정도로 모델의 구성이 단순하거나 데이터 개수가 부족할 때 발생한다.



단항식이 아닌 다항식으로 선형 회귀식을 만들면 복잡한 구조를 갖게 되어 모델의 예측력을 높일 수 있다.

과대적합 회피 – 과대적합 vs 과소적합

[17] # 2차 다항식 변환

```
from sklearn.preprocessing import PolynomialFeatures  
pf = PolynomialFeatures(degree=2)  
X_train_poly = pf.fit_transform(X_train)  
print("원본 학습 데이터셋:", X_train.shape)  
print("2차 다항식 변환 데이터셋:", X_train_poly.shape)
```

➡ 원본 학습 데이터셋:(404, 2)

2차 다항식 변환 데이터셋:(404, 6)

2차식으로 변환된 데이터셋을 선형 회귀 모델에 입력하여 학습한다.

1차항 모델보다 성능이 좋은 편이다.

과대적합 회피 – 과대적합 vs 과소적합

```
[18] # 2차 다항식 변환 데이터셋으로 선형 회귀 모형 학습

lr = LinearRegression()
lr.fit(X_train_poly, y_train)

# 테스트 데이터에 대한 예측 및 평가
y_train_pred = lr.predict(X_train_poly)
train_mse = mean_squared_error(y_train, y_train_pred)
print("Train MSE:%.4f" % train_mse)

X_test_poly = pf.fit_transform(X_test)
y_test_pred = lr.predict(X_test_poly)
test_mse = mean_squared_error(y_test, y_test_pred)
print("Test MSE:%.4f" % test_mse)
```

➡ Train MSE:21.5463

Test MSE:16.7954

15차 다항식으로 변환해 보자. Test MSE는 급격하게 증가한다. 과대적합이 발생한 것으로 볼 수 있다.

과대적합 회피 – 과대적합 vs 과소적합

```
[19] # 15차 다항식 변환 데이터셋으로 선형 회귀 모형 학습

pf = PolynomialFeatures(degree=15)
X_train_poly = pf.fit_transform(X_train)

lr = LinearRegression()
lr.fit(X_train_poly, y_train)

# 테스트 데이터에 대한 예측 및 평가
y_train_pred = lr.predict(X_train_poly)
train_mse = mean_squared_error(y_train, y_train_pred)
print("Train MSE:%.4f" % train_mse)

X_test_poly = pf.fit_transform(X_test)
y_test_pred = lr.predict(X_test_poly)
test_mse = mean_squared_error(y_test, y_test_pred)
print("Test MSE:%.4f" % test_mse)
```



Train MSE:11.1589

Test MSE:108504063264728.0625

다항식 차수에 따른 차이를 비교한다.

과대적합 회피 – 과대적합 vs 과소적합

```
[20] # 다항식 차수에 따른 모델 적합도 변화
plt.figure(figsize=(15,5))
for n, deg in enumerate([1, 2, 15]):
    ax1 = plt.subplot(1, 3, n+1)
    # plt.axis('off')
    # degree별 다항 회귀 모형 적용
    pf = PolynomialFeatures(degree=deg)
    X_train_poly = pf.fit_transform(X_train.loc[:, ['LSTAT']])
    X_test_poly = pf.fit_transform(X_test.loc[:, ['LSTAT']])
    lr = LinearRegression()
    lr.fit(X_train_poly, y_train)
    y_test_pred = lr.predict(X_test_poly)
```

과대적합 회피 – 과대적합 vs 과소적합

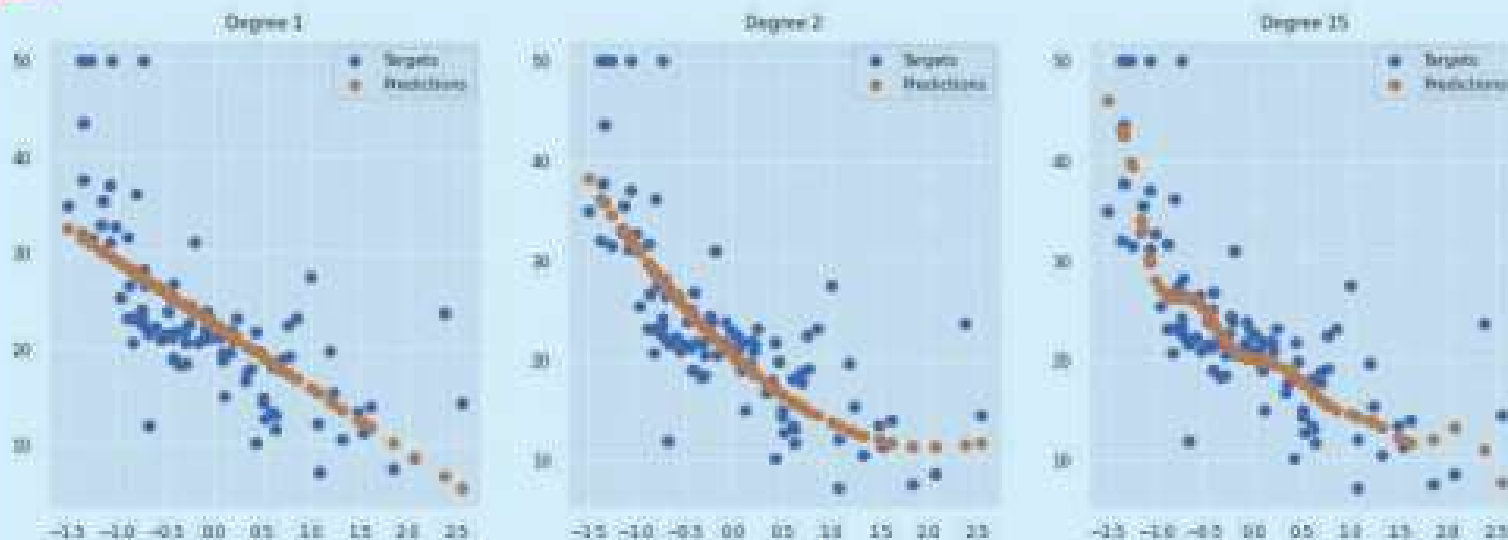
```
# 실제값 분포
plt.scatter(X_test.loc[:, 'LSTAT'], y_test, label='Targets')

# 예측값 분포
plt.scatter(X_test.loc[:, 'LSTAT'], y_test_pred, label='Predictions')

# 제목 표시
plt.title("Degree %d" % deg)

# 범례 표시
plt.legend()

plt.show()
```



모델 복잡도에 따른 과소/과대적합

과대적합 회피 – L2/L1 규제

모델의 복잡도를 낮추면 과대적합을 억제할 수 있다. 모델을 설명하는 각 피처가 모델의 예측 결과에 미치는 영향력을 가중치(회귀계수)로 표현하는데, 이런 가중치들이 커지면 페널티를 부과 하여 가중치를 낮은 수준으로 유지한다. 이처럼 모델의 구조가 복잡해지는 것을 억제하는 방법을 규제(Regularization)라고 부른다.

모델의 구조를 간결하게 하는 방법 중 L2/L1 규제가 있다. L2 규제는 모델의 '가중치의 제곱합'에 페널티를 부과하고, L1 규제는 '가중치 절대값의 합'에 페널티를 부과하여 모델의 복잡도를 낮출 수 있다.

Ridge 모델은 선형회귀 모형에 L2 규제를 구현한 알고리즘이다. 알파(alpha) 값으로 L2 규제 강도를 조정한다. 알파 값을 증가시키면 규제 강도가 커지고 모델의 가중치를 감소시킨다.

과대적합 회피 - L2/L1 규제

```
[21] # Ridge(L2 규제)

from sklearn.linear_model import Ridge
rdg = Ridge(alpha=2.5)
rdg.fit(X_train_poly, y_train)

y_train_pred = rdg.predict(X_train_poly)
train_mse = mean_squared_error(y_train, y_train_pred)
print("Train MSE:%.4f" % train_mse)

y_test_pred = rdg.predict(X_test_poly)
test_mse = mean_squared_error(y_test, y_test_pred)
print("Test MSE:%.4f" % test_mse)
```

```
➡ Train MSE:35.8300
   Test MSE:41.8125
```

L1 규제를 적용한 Lasso 모델을 학습한다.

과대적합 회피 - L2/L1 규제

```
[22] # Lasso(L1 규제)

from sklearn.linear_model import Lasso
las = Lasso(alpha=0.05)
las.fit(X_train_poly, y_train)

y_train_pred = las.predict(X_train_poly)
train_mse = mean_squared_error(y_train, y_train_pred)
print("Train MSE:%.4f" % train_mse)

y_test_pred = las.predict(X_test_poly)
test_mse = mean_squared_error(y_test, y_test_pred)
print("Test MSE:%.4f" % test_mse)
```

```
➡ Train MSE:32.3204
   Test MSE:37.7103
```

과대적합 회피 - L2/L1 규제

ElasticNet 알고리즘은 L2 규제와 L1 규제를 모두 적용한 선형 회귀 모델이다.

ElasticNet의 알파 (alpha) 값은 L2 규제 강도와 L1 규제 강도의 합이다.

l1_ratio 옵션은 L1 규제 강도의 상대적 비율을 조정한다.

```
ela = ElasticNet(alpha=0.01, l1_ratio=0.7)
ela.fit(X_train_poly, y_train)

y_train_pred = ela.predict(X_train_poly)
train_mse = mean_squared_error(y_train, y_train_pred)
print("Train MSE:%4f" % train_mse)

y_test_pred = ela.predict(X_test_poly)
test_mse = mean_squared_error(y_test, y_test_pred)
print("Test MSE:%4f" % test_mse)
```

```
➡ Train MSE:33.7390
   Test MSE:39.4738
```

트리 기반 모델 - 비선형 회귀 : 의사결정나무

DecisionTreeRegressor는 의사결정나무 알고리즘으로 회귀 모형을 구현한 것이다.

```
[35] # 의사결정나무
```

```
from sklearn.tree import DecisionTreeRegressor  
dtr = DecisionTreeRegressor(max_depth=3, random_state=12)  
dtr.fit(X_train, y_train)  
  
y_train_pred = dtr.predict(X_train)  
train_mse = mean_squared_error(y_train, y_train_pred)  
print("Train MSE: %.4f" % train_mse)  
  
y_test_pred = dtr.predict(X_test)  
test_mse = mean_squared_error(y_test, y_test_pred)  
print("Test MSE: %.4f" % test_mse)
```

```
➡ Train MSE:18.8029
```

```
Test MSE:17.9065
```

트리 기반 모델 – 비선형 회귀 : 랜덤 포레스트

하나의 트리를 사용하는 의사결정나무에 비하여, 여러 개의 트리 모델이 예측한 값을 종합하기 때문에 전체 예측력을 높일 수 있다.

```
[36] # 랜덤 포레스트
```

```
from sklearn.ensemble import RandomForestRegressor  
rfr = RandomForestRegressor(max_depth=3, random_state=12)  
rfr.fit(X_train, y_train)
```

트리 기반 모델 - 비선형 회귀 : 랜덤 포레스트

```
[36] # 랜덤 포레스트
```

```
from sklearn.ensemble import RandomForestRegressor  
rfr = RandomForestRegressor(max_depth=3, random_state=12)  
rfr.fit(X_train, y_train)  
  
y_train_pred = rfr.predict(X_train)  
train_mse = mean_squared_error(y_train, y_train_pred)  
print("Train MSE:%.4f" % train_mse)  
  
y_test_pred = rfr.predict(X_test)  
test_mse = mean_squared_error(y_test, y_test_pred)  
print("Test MSE:%.4f" % test_mse)
```



```
Train MSE:16.0201
```

```
Test MSE:17.7751
```

트리 기반 모델 – 비선형 회귀 : XGBoost

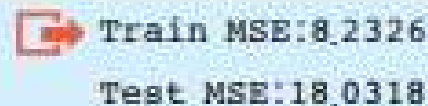
부스팅 알고리즘인 XGBRegressor를 적용한다.

```
[37] # XGBoost
from xgboost import XGBRegressor

xgbr = XGBRegressor(objective='reg:squarederror', max_depth=3, random_state=12)
xgbr.fit(X_train, y_train)

y_train_pred = xgbr.predict(X_train)
train_mse = mean_squared_error(y_train, y_train_pred)
print("Train MSE:%.4f" % train_mse)

y_test_pred = xgbr.predict(X_test)
test_mse = mean_squared_error(y_test, y_test_pred)
print("Test MSE:%.4f" % test_mse)
```



```
Train MSE:8.2326
Test MSE:18.0318
```

Tip 데이터의 개수가 작기 때문에 XGBoost와 같이 복잡도가 높은 알고리즘이 쉽게 과대적합될 위험성이 있다. XGBoost 알고리즘은 데이터의 개수가 비교적 많고, 모델 예측의 난이도가 높은 경우 탁월한 성능을 보인다.