1. 선형 모델

1 선형 회귀 (1) 모델 구조 및 특성

모델 구조 및 특성

선형 회귀는 특징 벡터와 가중치 벡터의 가중합에 편향을 더 하는 방식으로 라벨을 예측합니다.

$$f(\boldsymbol{x}) = w_1 x_1 + w_2 x_2 + \cdots + w_m x_m + b$$

- $\boldsymbol{w} = (w_1, w_2, \cdots, w_m)$: 가중치 벡터(weight vector)
- b: 편향(bias)

- 선형 회귀의 모델 구조는 선형적 (linear)이므로, 특징과 라벨이 선형 관계에 있을 때만 유의함
- 가중합 형태로 구성돼 있기에 특징의 스케일 차이에 크게 영향을 받음
- 모든 특징이 연속형이고 스케일이 유사한 데이터에 적합함

손실 함수: 오차 제곱합

다중 선형 회귀 모델 학습은 오차 제곱합을 최소화하는 파라미터를 추정하는 것입니다.

$$\boldsymbol{w}^*, b^* = \underset{\boldsymbol{w},b}{\operatorname{argmin}} \mathcal{L}(\boldsymbol{w}, b)$$

$$\mathcal{L}(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- n: 학습 데이터 크기(샘플 수)
- y_i: i번째 샘플의 라벨
- $\hat{y}_i = f(\boldsymbol{x}_i)$는 i번째 샘플의 라벨을 예측한 값
- 양의 오차와 음의 오차를 합해서 상쇄되는 것을 막기 위해, 오차 합 대신 오차 제곱합을 사용함

- 오차 제곱합을 손실 함수로 사용하면 학습 데이터에 대한 오차를 최대한 줄이려다 모델이 과적합 될 우려가 있음
- 과적합 된 선형 회귀 모델은 각 학습 샘플을 정밀하게 예측하는 방식으로 학습돼, 계수의 절댓값이 큰 경향이 있음

손실 함수: 계수 패널티 추가

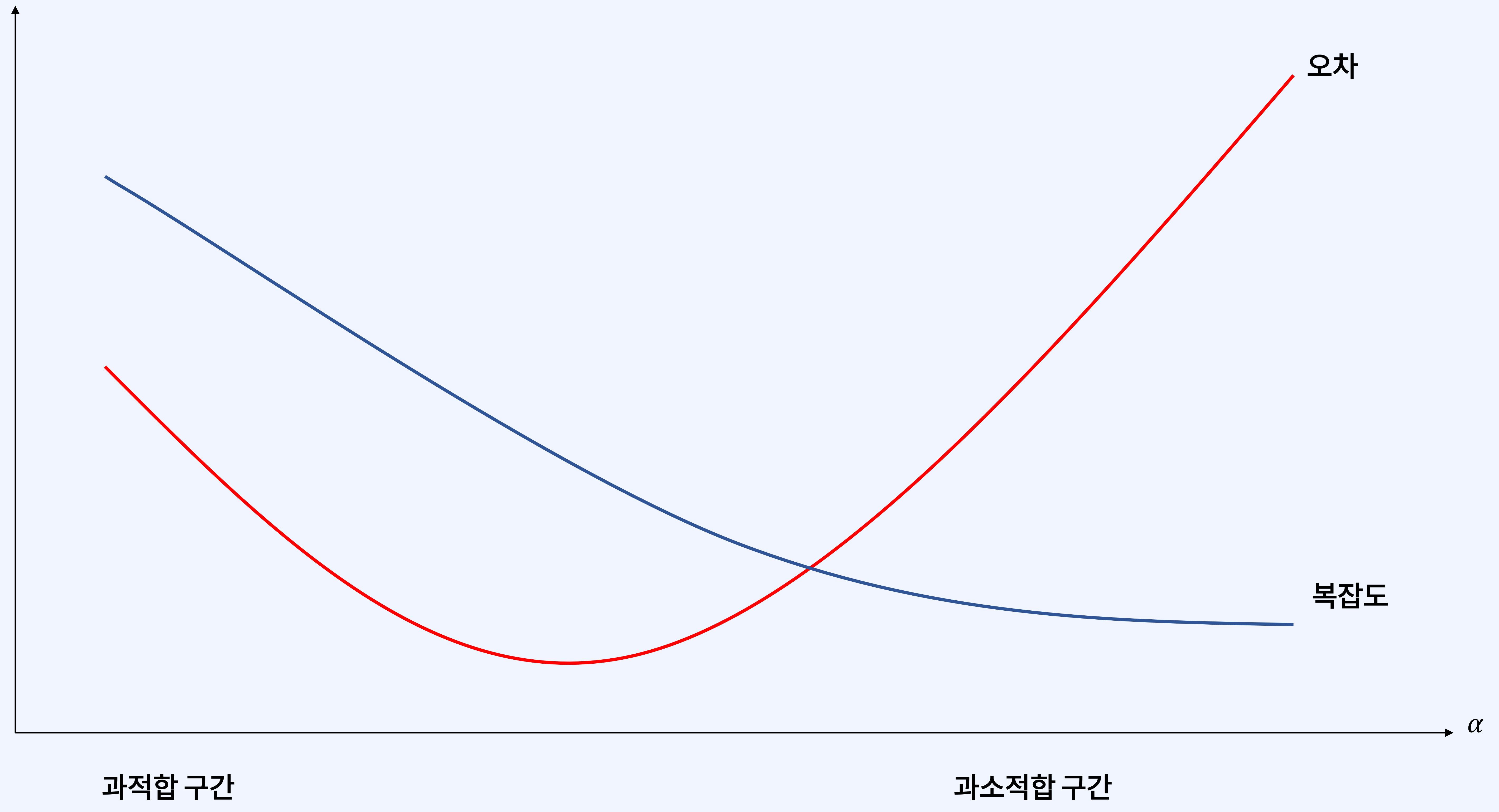
과적합을 방지하고자 손실 함수에 계수에 대한 페널티를 부여하며, 부여한 페널티에 따라 모델이 다릅니다

라쏘 (Lasso)	$\mathcal{L}(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \alpha \sum_{j=1}^{m}\lvert w_j\rvert$	L1 페널티 추가
릿지 (Ridge)	$\mathcal{L}(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \alpha \sum_{j=1}^{m} w_j^2$	L2 페널티 추가
엘라스틱 넷 (Elastic Net)	$\mathcal{L}(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \alpha_1 \sum_{j=1}^{m}\lvert w_j\rvert + \alpha_2 \sum_{j=1}^{m} w_j^2$	L1 페널티 & L2 페널티 추가

- L1 페널티는 불필요한 계수를 0으로 만들고 L2 페널티는 불필요한 계수를 0에 가깝게 만듦
- 그러나 어디까지나 이론적이고 해석적인 이야기이며, 새로운 데이터를 얼마나 잘 예측하는지에 대한 관점에서는 크게 차이가 없으므로 하나를 임의로 선택해도 무방함
- 엘라스틱 넷은 두 종류의 페널티를 모두 고려한 더 발전된 모델처럼 보이나, 하이퍼 파라미터 튜닝만 어려움

복잡도 하이퍼 파라미터

$\alpha, \alpha_1, \alpha_2$는 계수 페널티의 가중치를 나타내는 하이퍼 파라미터로 그 값이 클수록 모델이 단순해짐



1. 선형 모델

2 선형 회귀 (2) 사이킷런 실습

예제 데이터 불러오기

선형 모델 학습에 사용할 예제 데이터를 불러옵니다.

예제 데이터 불러오기

```python
import pandas as pd
from sklearn.model_selection import train_test_split
df = pd.read_csv("../../data/regression/compactiv.csv")
X = df.drop('y', axis = 1)
y = df['y']
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 2022)
```

모델 학습

선형 회귀 모델은 사이킷런의 linear_model 모듈로 구현할 수 있으며, 이 모듈에는 LinearRegression, Ridge, Lasso, ElasticNet이라는 클래스가 있습니다.

선형 회귀 모델 학습 예제

```python
from sklearn.linear_model import *
LR = LinearRegression().fit(X_train, y_train)
ridge = Ridge().fit(X_train, y_train)
lasso = Lasso().fit(X_train, y_train)
EN = ElasticNet().fit(X_train, y_train)
```

모델 평가

회귀 모델을 같은 방법으로 반복해서 평가해야 하므로 regression_model_test라는 함수를 만들어 모델을 평가하겠습니다.

선형 회귀 모델 평가 예제

```
from sklearn.metrics import mean_absolute_error as MAE
def regression_model_test(model, X_test, y_test):
    y_pred = model.predict(X_test)
    mae = MAE(y_test, y_pred)
    return mae
```

```
LR_mae = regression_model_test(LR, X_test, y_test)
ridge_mae = regression_model_test(ridge, X_test, y_test)
lasso_mae = regression_model_test(lasso, X_test, y_test)
EN_mae = regression_model_test(EN, X_test, y_test)
print(LR_mae, ridge_mae, lasso_mae, EN_mae)
```

```
6.0468771931780605  6.04686638162807  6.099519755350322  6.074987094435656
```

- 모델 간 성능 차이가 그리 크지는 않음

복잡도 하이퍼 파라미터 튜닝

Ridge와 Lasso 클래스는 모두 alpha라는 인자가 있는데, 이 인자는 손실 함수에서 계수 패널티에 대한 가중치를 나타냅니다.

alpha에 따른 성능 측정 예시

```
1   Lasso1 = Lasso(alpha = 0.1, random_state = 2022).fit(X_train, y_train)
2   Lasso2 = Lasso(alpha = 1, random_state = 2022).fit(X_train, y_train)
3   Lasso3 = Lasso(alpha = 10, random_state = 2022).fit(X_train, y_train)
4
5   Lasso1_mae = regression_model_test(Lasso1, X_test, y_test)
6   Lasso2_mae = regression_model_test(Lasso2, X_test, y_test)
7   Lasso3_mae = regression_model_test(Lasso3, X_test, y_test)
8   print(Lasso1_mae, Lasso2_mae, Lasso3_mae)
```

- **라인 1 – 3:** 우연에 의해 결과가 뒤바뀌지 않도록 alpha뿐만 아니라, random_state도 설정했습니다.

```
6.044036480564693    6.099519755350322    6.2038739785793755
```

- Lasso1_mae < Lasso2_mae < Lasso3_mae임을 알 수 있음.
  즉, alpha가 작을수록 더 좋은 성능이 나옴

복잡도 하이퍼 파라미터 튜닝 (계속)

alpha를 0.05와 5로 각각 설정해 추가로 평가해보겠습니다.

alpha에 따른 성능 측정 예시

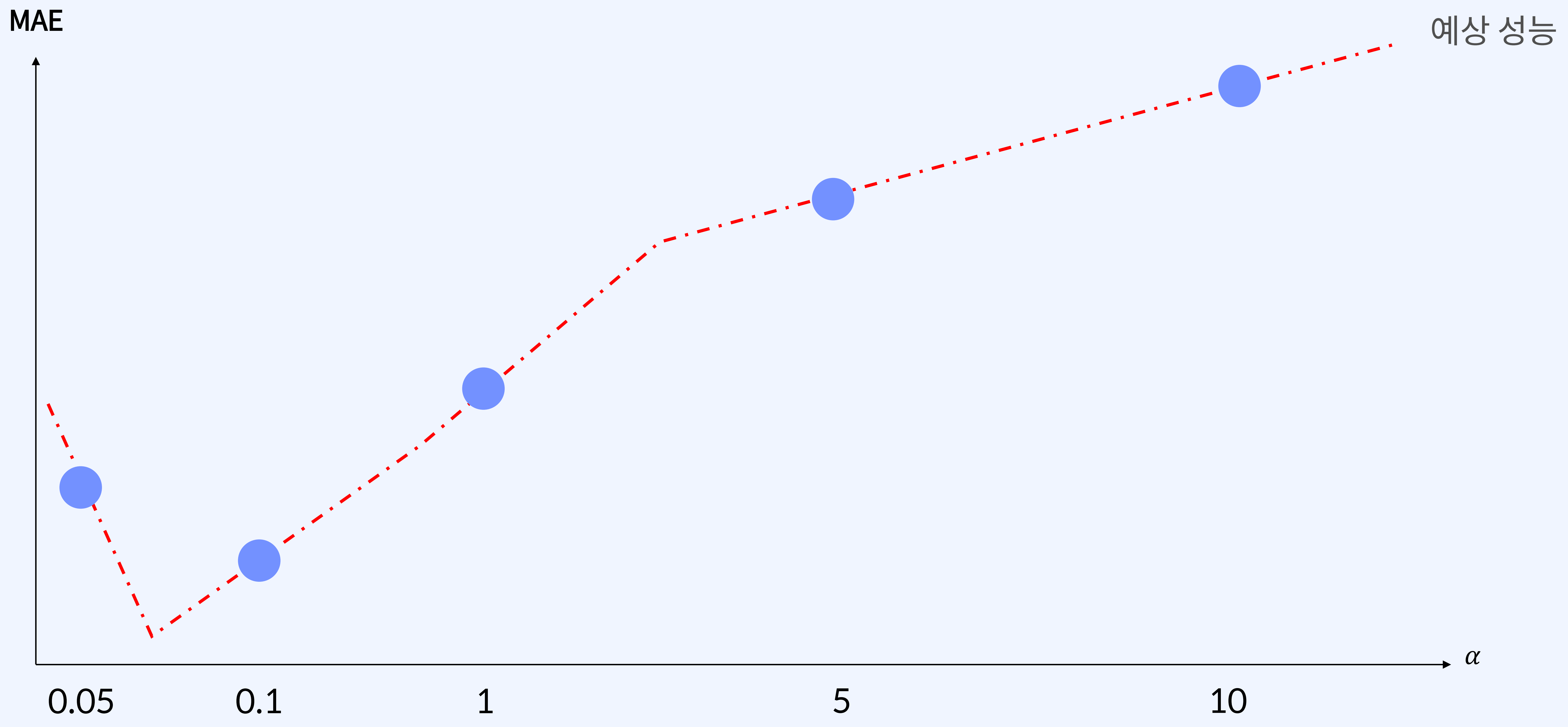
```
Lasso4 = Lasso(alpha = 0.05, random_state = 2022).fit(X_train, y_train)
Lasso5 = Lasso(alpha = 5, random_state = 2022).fit(X_train, y_train)

Lasso4_mae = regression_model_test(Lasso4, X_test, y_test)
Lasso5_mae = regression_model_test(Lasso5, X_test, y_test)
print(Lasso4_mae, Lasso5_mae)
```

```
6.045073780285435    6.1814548400964116
```

복잡도 하이퍼 파라미터 튜닝 (계속)

alpha가 0.05일 때보다 0.1일 때의 성능이 더 좋으며, 1일 때보다 5일 때 성능이 더 좋지 않습니다. 5개의 alpha를 평가한 결과를 통해, 시드가 2022로 고정됐을 때 한해 최적의 alpha는 0.05와 1 사이에 있다고 할 수 있습니다.



1. 선형 모델

3 로지스틱 회귀

모델 구조 및 특성

로지스틱 회귀는 특징이 주어졌을 때, 라벨이 1(긍정 클래스)일 확률을 다음과 같이 계산합니다.

$$\Pr(y = 1|\boldsymbol{x}) = \frac{1}{1 + \exp(-w_1 x_1 - w_2 x_2 - \cdots - w_m x_m - b)}$$

- $\boldsymbol{w} = (w_1, w_2, \cdots, w_m)$: 가중치 벡터(weight vector)
- b: 편향(bias)

확률과 임계치 θ를 바탕으로 다음과 같이 분류합니다.

$$\hat{y} = \begin{cases} 1, & \Pr(y = 1|\boldsymbol{x}) \geq \theta \\ 0, & \text{otherwise} \end{cases}$$

예제 데이터 불러오기

분류 모델을 학습할 데이터를 불러오고 분리하겠습니다.

예제 데이터 불러오기

```
df = pd.read_csv("../../data/classification/ecoli1.csv")
X = df.drop('y', axis = 1)
y = df['y']
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 2022)
```


로지스틱 회귀 모델 학습

로지스틱 회귀는 linear_model의 LogisticRegression 클래스로 구현할 수 있습니다.

주요 인자

인자	설명	기본 값
penalty	페널티 종류: {"l1", "l2", "elasticnet", "none"}	"l2"
C	페널티 계수에 대한 가중치로 부여되는 페널티는 C에 반비례함	1.0

로지스틱 회귀 모델 학습 및 평가 예제

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression().fit(X_train, y_train)
from sklearn.metrics import f1_score
y_pred = model.predict(X_test)
f1 = f1_score(y_test, y_pred)
print(f1)
```

```
0.7096774193548387
```

임계치에 따른 정밀도와 재현율 계산

클래스에 속할 확률 y_prob을 계산하고 모델의 클래스 정보 출력

```
1  y_prob = model.predict_proba(X_test)
2  print(model.classes_)
```

```
[0  1]
```

- **라인 2:** 분류 모델의 classes_ 속성은 분류 모델이 학습할 때 사용했던 클래스 목록을 반환합니다. 이 결과를 출력한 이유는 predict_proba 메서드가 반환한 배열의 i행 j열의 값이 샘플 i가 model.classes_[j]에 속할 확률이기 때문입니다.

임계치에 따른 예측값 계산 예제

```
1  from sklearn.metrics import precision_score, recall_score
2  def precision_and_recall_accto_threshold(y_prob, y_test, threshold):
3      y_prob_pred = (y_prob[:, 1] > threshold).astype(int)
4      precision = precision_score(y_test, y_prob_pred)
5      recall = recall_score(y_test, y_prob_pred)
6      return precision, recall
```

- **라인 3:** y_prob의 1번째 열을 threshold와 비교한 결과의 각 요소를 int 형으로 바꾼 배열을 y_prob_pred에 저장합니다. 즉, threshold보다 큰 값은 1이 되며, 그렇지 않은 값은 0이 됩니다.
- **라인 4 – 5:** y_prob_pred와 y_test를 바탕으로 재현율과 정밀도를 계산합니다.

임계치에 따른 정밀도와 재현율 계산 (계속)

임계치에 따른 예측값 계산 예제

```
1  import numpy as np
2  precision_list = []
3  recall_list = []
4  threshold_list = np.arange(0, 1, 0.01)
5  for threshold in threshold_list:
6      precision, recall = precision_and_recall_accto_threshold(y_prob, y_test, threshold)
7      precision_list.append(precision)
8      recall_list.append(recall)
```

- **라인 2 – 3:** 정밀도와 재현율을 담을 빈 리스트를 정의합니다.
- **라인 4:** 0부터 0.01씩 1까지 늘린 값으로 구성된 배열을 threshold_list에 저장합니다.
- **라인 5 – 7:** threshold를 0부터 0.01씩 늘려가면서 precision_and_recall_accto_threshold를 적용한 결과를 각각 precision_list와 recall_list에 추가합니다.

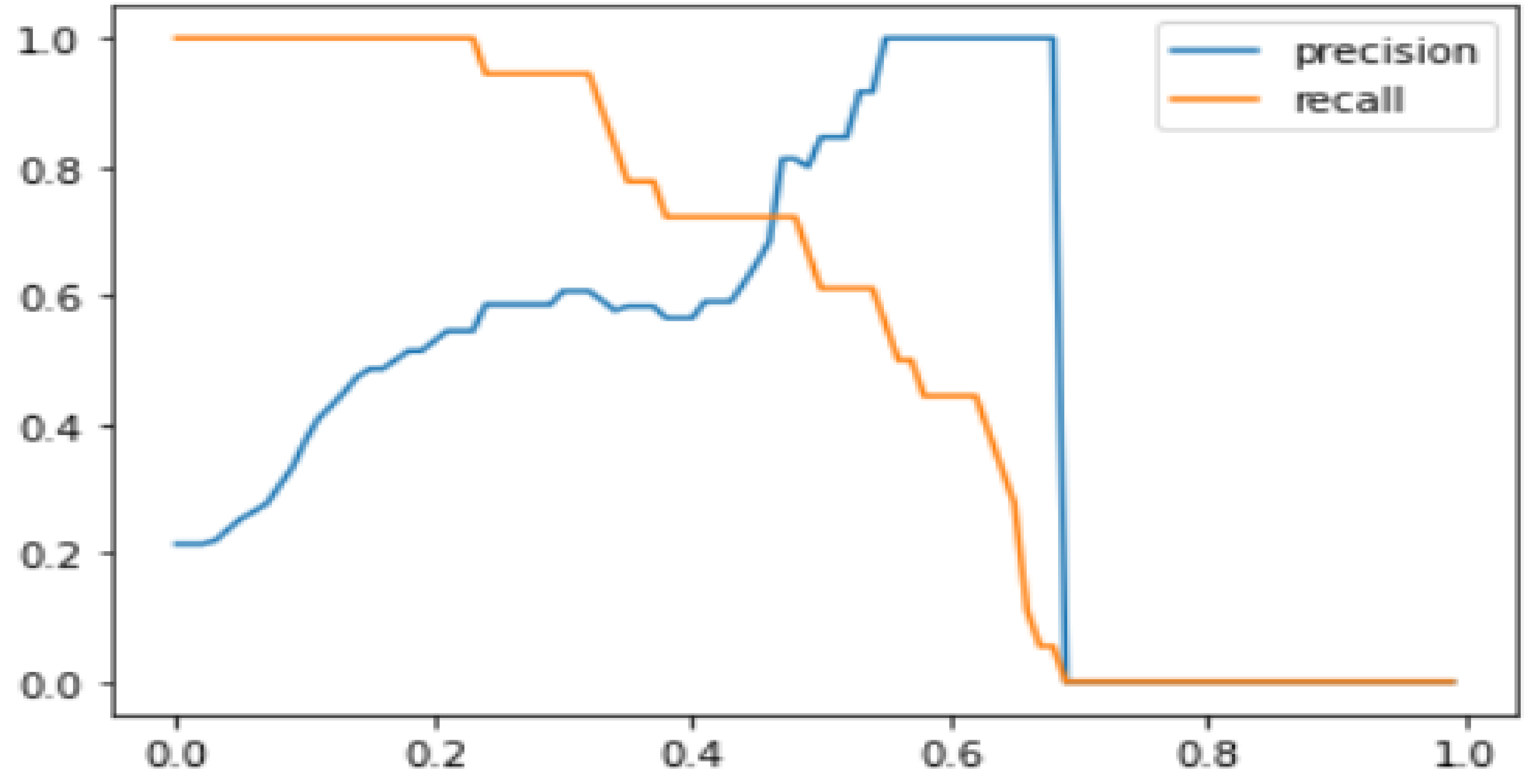
```
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 due to no predicted samples.
Use `zero_division` parameter to control this behavior.
_warn_prf(average, modifier, msg_start, len(result))
```

- 정밀도의 분모가 0이어서 정상적으로 계산하지 못하고 정밀도를 0으로 설정했다는 내용의 경고

임계치에 따른 정밀도와 재현율 계산 (계속)

임계치에 따른 예측값 계산 결과 시각화

```
from matplotlib import pyplot as plt
plt.plot(threshold_list, precision_list, label = "precision")
plt.plot(threshold_list, recall_list, label = "recall")
plt.legend()
plt.show()
```



- 임계치와 정밀도는 비례하고 임계치와 재현율은 반비례함
- 임계치가 0.7 정도 되는 시점에서 긍정이라 분류하는 샘플이 하나도 없어 정밀도가 0이 됐음
- 임계치에 따라 정밀도가 계속 증가하지 않고 소폭 감소하기도 함

1. 선형 모델

4 선형성을 고려한 특징 공학

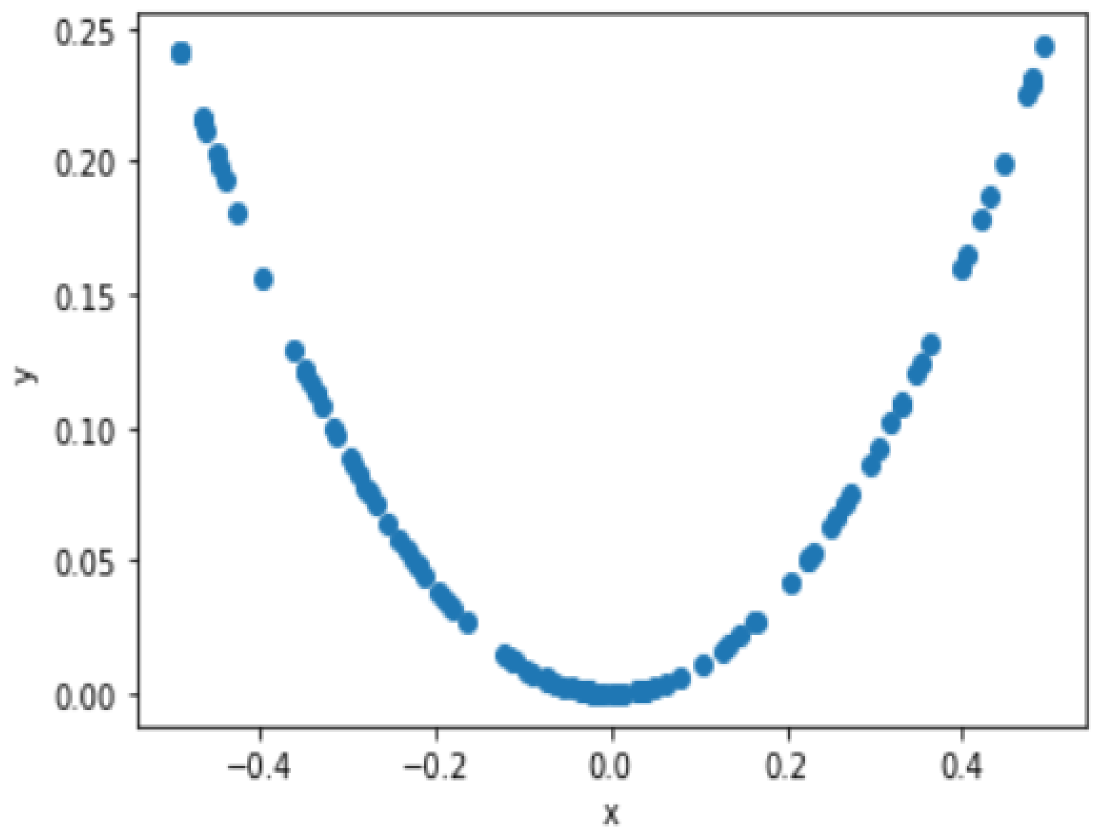
선형 모델의 한계

선형 모델은 선형적인 관계만 적절히 모델링할 수 있는 한계가 있습니다.

선형 모델의 한계 확인 예제

```
1  x = np.random.random(100) - 0.5
2  y = x ** 2
3  plt.scatter(x, y)
4  plt.xlabel("x")
5  plt.ylabel("y")
```

- **라인 1:** np.random.random 함수는 0과 1 사이의 난수를 생성하므로 0.5를 빼서 -0.5와 0.5 사이로 x의 범위를 수정했습니다.



- 두 변수 간에는 $y = x^2$이라는 자명한 관계가 있으므로 x로 y를 예측하기 매우 쉬워 보임

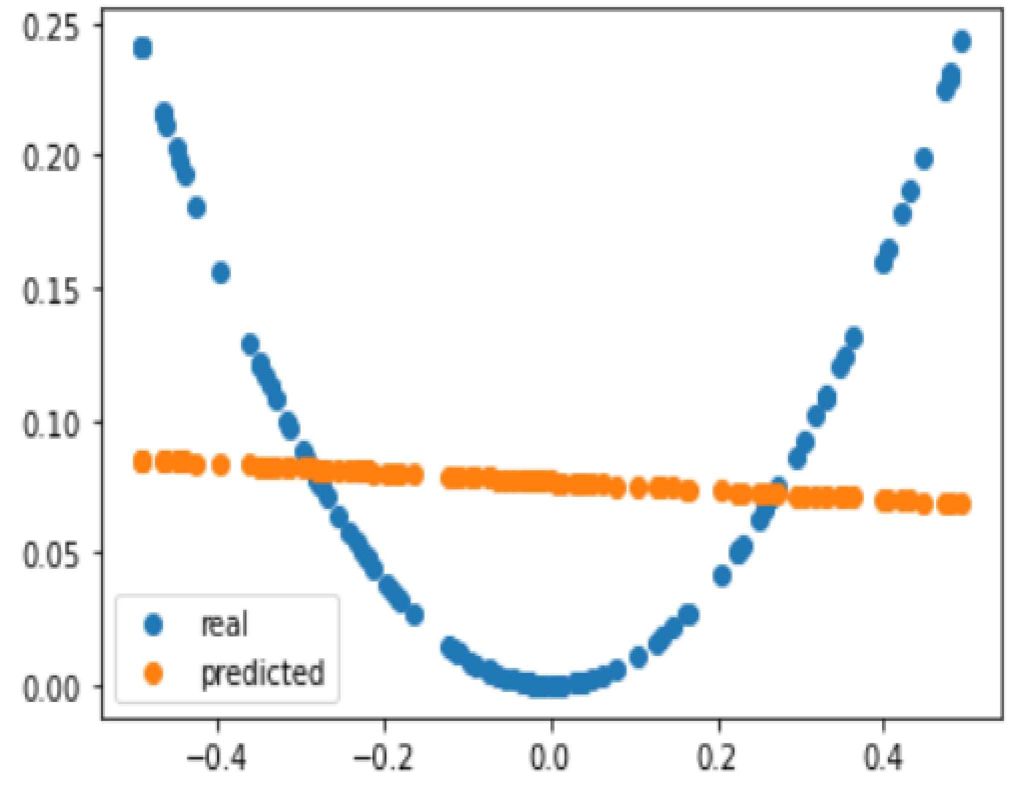
선형 모델의 한계 (계속)

선형 모델은 선형적인 관계만 적절히 모델링할 수 있는 한계가 있습니다.

선형 모델의 한계 확인 예제

```
1  model = LinearRegression().fit(x.reshape(-1, 1), y)
2  y_pred = model.predict(x.reshape(-1, 1))
3  plt.scatter(x, y, label = "real")
4  plt.scatter(x, y_pred, label = "predicted")
5  plt.legend()
6  plt.show()
```

- 라인 1: x와 y를 사용해 다중 선형 회귀 모델을 학습했습니다. 이때, x가 1차원인데 fit 메서드는 2차원 배열 형태의 특징 벡터를 입력받으므로 reshape 메서드를 이용해 모양을 바꿨습니다.



- 실제 라벨의 분포인 파란색과 예측된 결과의 분포인 주황색이 크게 다름
- 이러한 결과가 나온 이유는 선형 회귀는 $wx + b$ 형태의 구조로 y를 예측하는데, 이 구조로는 x^2을 표현할 수 없기 때문

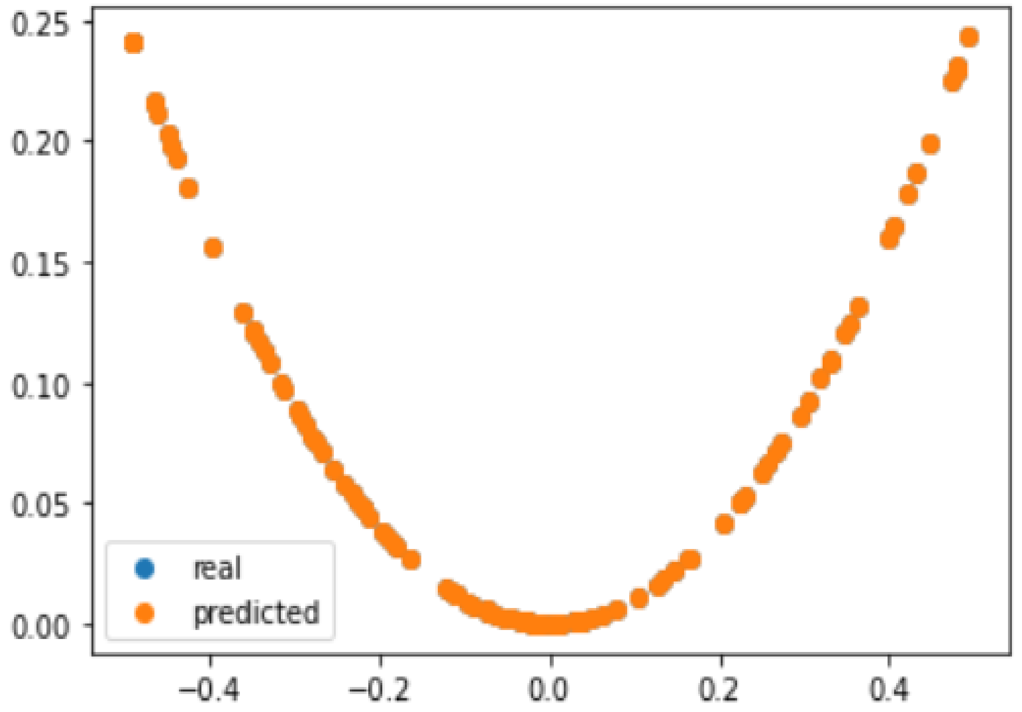
특징 변환 및 생성

선형 회귀 모델로 특징과 라벨 간 비선형 관계를 표현하는 방법은 새로운 특징을 추가하거나 기존 특징을 변환하는 것입니다.

비선형 특징 예제

```
1 new_X = pd.DataFrame({"x":x, "x_squared":x**2})
2 model = LinearRegression().fit(new_X, y)
3 y_pred = model.predict(new_X)
4 plt.scatter(x, y, label = "real")
5 plt.scatter(x, y_pred, label = "predicted")
6 plt.legend()
7 plt.show()
```

- **라인 1:** x와 x**2으로 구성된 새로운 데이터프레임 new_X를 생성했습니다.



- 완벽하게 예측되어 주황색 점에 의해 파란색 점이 모두 가려졌음

특징 변환 및 생성 (계속)

현실적으로는 특징과 라벨 간 관계를 알 수 없으므로 미리 새로운 특징을 생성하는 함수를 통해 여러 개의 특징을 생성한 뒤 특징 선택을 통해 다시 차원을 줄여야 합니다.

특징과 라벨 간 관계를 정확히 알고 있다는 것 자체가 매우 비현실적임

각 특징을 라벨과 함께 시각화하여 그 관계를 파악하고, 그 관계에 맞는 새로운 특징을 추가하는 것 역시 비현실적임

미리 새로운 특징을 생성하는 함수(예: 제곱, 루트, 지수 등)를 정의하고, 그 함수를 이용해 여러 개의 특징을 생성한 뒤 특징 선택을 통해 다시 차원을 줄이는 방법이 있음