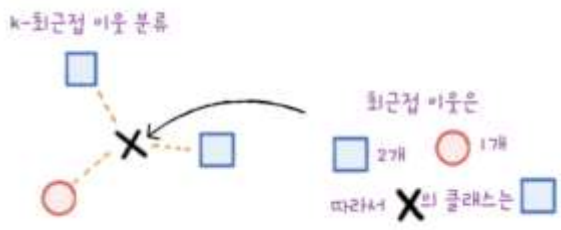
[K-최근접 이웃 회귀]

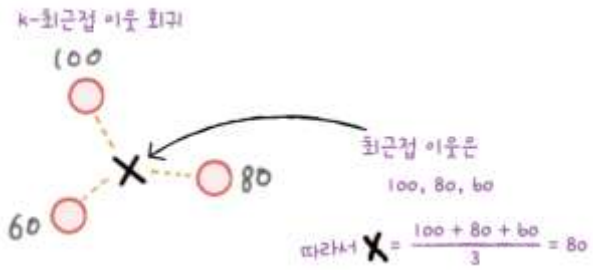
- 지도 학습의 한 종류인 회귀 문제를 이해하고 K-최근접 이웃 알고리즘을 사용해 농어의 무게를 예측하는 회귀 문제를 풀어 본다.
- 도미와 빙어에 이어 농어를 추가하려고 한다.
- 농어를 무게 단위로 판매하려 한다. 농어를 마리당 가격으로 판매했을 때 기대보다볼품없는 농어를 받은 고객이 항의하는일이 발생했기 때문
- 그런데 농어의 무게가 잘못 측정된 상황이어서 농어 무게를 재측정해야 한다.
- 농어의 길이, 높이, 두께를 측정한 데이터가 있어서 이걸로 농어의 무게를 예측하려고 한다. 정확하게 무게를 측정한 농어 샘플 56 개가 있는 상황이다.
- 지도학습에는 크게 분류와 **회귀(regression)**로 나뉜다. 분류는 샘플을 몇 개의 클래스 중 하나로 분류하는 문제이다. **회귀는 클래스 중 하나로 분류하는 것이 아니라 임의의 어떤 숫자를 예측하는 것**을 말한다.
- 예) 내년도 경제 성장률 예측, 배달이 도착할 시간 예측하는 것이 회귀 문제임
- 회귀는 정해진 클래스가 없고 임의의 수치를 출력함
- 혹은 두 변수의 상관관계를 분석하는 방법을 회귀라고 한다.
- k-최근접 이웃 알고리즘이 회귀에도 작동한다
- k-최근접 이웃 분류 알고리즘 - 예측하려는 샘플에 가장 가까운 샘플 k 개를 선택. 그 다음 이 샘플들의 클래스를 확인하여 다수 클래스를 새로운 샘플의 클래스로 예측.
- 예) k=3(샘플이 3 개)이라 가정하면 사각형이 2 개로 다수이기 때문에 새로운 샘플 X 의 클래스는 사각형이 됨



k-최근접 이웃 회귀 - 분류와 똑같이 예측하려는 샘플에 가장 가까운 샘플 k 개를 선택.회귀이기 때문에 이웃한 샘플의 타깃은 어떤 클래스가 아니라 임의의 수치임.

이웃 샘플들의 수치를 사용해 새로운 샘플 X 의 타깃을 예측하는 간단한 방법에는 이 수치들의 평균을 구하는 방법이 있다.

이웃한 샘플의 타깃값이 각각 100,80,60 이고 이를 평균하면 샘플 X 의 예측 타깃값은 80



1. 데이터 준비

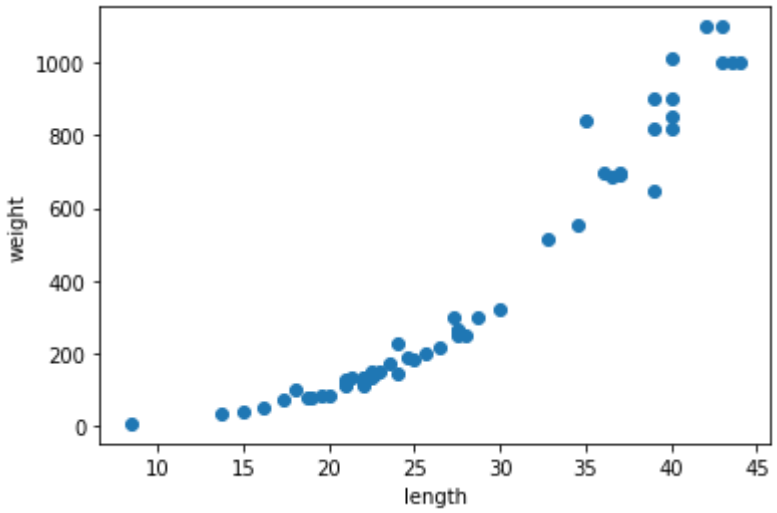
- 농어 데이터를 준비하고 사이킷런을 사용해 회귀모델을 훈련하자
- 훈련 데이터를 준비하자. 농어의 길이로 무게를 예측하고자 한다. 농어의 길이가 특성이고 무게가 타깃이 된다.

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_absolute_error

perch_length=np.array([8.4,13.7,15.0,16.2,17.4,18.0,18.7,19.0,19.6,20.0,21.0,
21.0,21.0,21.3,22.0,22.0,22.0,22.0,22.0,22.5,22.5,22.7,
23.0,23.5,24.0,24.0,24.6,25.0,25.6,26.5,27.3,27.5,27.5,
27.5,28.0,28.7,30.0,32.8,34.5,35.0,36.5,36.0,37.0,37.0,
39.0,39.0,39.0,40.0,40.0,40.0,40.0,42.0,43.0,43.0,43.5,
44.0])
perch_weight=np.array([5.9,32.0,40.0,51.5,70.0,100.0,78.0,80.0,85.0,85.0,110.0,
115.0,125.0,130.0,120.0,120.0,130.0,135.0,110.0,130.0,
```

```
150.0,145.0,150.0,170.0,225.0,145.0,188.0,180.0,197.0,
218.0,300.0,260.0,265.0,250.0,250.0,300.0,320.0,514.0,
556.0,840.0,685.0,700.0,700.0,690.0,900.0,650.0,820.0,
850.0,900.0,1015.0,820.0,1100.0,1000.0,1100.0,1000.0,
1000.0])
```

- 넘파이 배열로 만든 뒤, 이 데이터가 어떤 형태를 띠고 있는지 산점도를 그려 보자. 하나의 특성을 사용하기 때문에, 특성 데이터(농어의 길이)를 x 축에 놓고 타깃 데이터(농어의 무게)를 y 축에 놓는다.
- scatter() 함수를 이용해 산점도를 그린다.

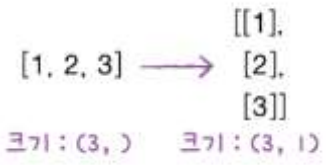
```
plt.scatter(perch_length,perch_weight)
plt.xlabel('length')
plt.ylabel('weight')
plt.show()
```



- 농어의 길이가 커짐에 따라 무게도 늘어난다
- 이전에 했듯이 사이킷런의 train_test_split() 함수를 사용해, 농어 데이터를 훈련 세트와 테스트 세트로 나누자.

```
train_input,test_input,train_target,test_target=train_test_split(perch_length,perch_weigh
t,random_state=42)
```

- 사이킷런에 사용할 훈련 세트는 2 차원 배열이어야 함. perch_length 가 1 차원 배열이기 때문에 이를 나눈 train_input 과 test_input 도 1 차원 배열임. 이런 1 차원 배열을 1 개의 열이 있는 2 차원 배열로 바꿔준다.



- 이 예제는 특성을 1 개만 사용하므로 수동으로 2 차원 배열을 만들어야 함. 넘파이 배열은 크기를 바꿀 수 있는 reshape()메서드를 제공함
- 예) (4,) 배열을 (2,2) 크기로 바꿔보자

```
test_array=np.array([1,2,3,4])
print(test_array.shape)
(4,)
test_array=test_array.reshape(2,2)
print(test_array.shape)
(2, 2)
```

- 이처럼 reshape() 메서드에는 바꾸려는 배열의 크기를 지정할 수 있다. 이 메서드를 이용해 train_input 과 test_input 을 2 차원 배열로 바꾸자.
- train_input 의 크기는 (42,) 임. 이를 2 차원 배열인 (42,1)로 바꾸려면 train_input.reshape(42,1)과 같이 사용함
- 넘파이는 배열의 크기를 자동으로 지정하는 기능도 제공함. 크기에 -1 을 지정하면 나머지 원소 개수로 모두 채우라는 의미.
- 예) 첫번째 크기를 나머지 원소 개수로 채우고, 두 번째 크기를 1 로 하려면 train_input.reshape(-1,1) 처럼 사용함

```
train_input=train_input.reshape(-1,1)
test_input=test_input.reshape(-1,1)
print(train_input.shape,test_input.shape)
(42, 1) (14, 1)
```

✓ **결정계수(R^2)**

사이킷런에서 k-최근접 이웃 회귀 알고리즘을 구현한 클래스는 KNeighborsRegressor 이다. 이 클래스의 사용법은 KNeighborsClassifier 와 매우 비슷하다. 객체를 생성하고 fit() 메서드로 회귀 모델을 훈련하겠다.

```
knr=KNeighborsRegressor()
knr.fit(train_input,train_target)
```

테스트 세트의 점수를 확인해보자

```
print(knr.score(test_input,test_target))
0.9928094061010639
```

분류의 경우 점수는 테스트 세트의 샘플을 정확하게 분류한 개수의 비율, 정확도이다. (- 정답을 맞힌 개수의 비율). 회귀에서는 정확한 숫자를 맞힌다는 것은 거의 불가능. 예측하는 값이나 타깃 모두 임의의 수치이기 때문. 회귀의 경우에는조금 다른 값으로 평가하는데 이 점수를 결정계수(coefficient of determination) 라고 부른다. 또는 간단히 R^2 라고도 부른다.

이 값은 다음과 같은 식으로 계산됨

$$R^2 = 1 - \frac{(\text{타깃} - \text{예측})^2\text{의 합}}{(\text{타깃} - \text{평균})^2\text{의 합}}$$

각 샘플의 타깃과 예측한 값의 차이를 제곱하여 더한 다음, 타깃과 타깃 평균의 차이를 제곱하여 더한 값으로 나눈다. 만약 타깃의 평균 정도를 예측하는 수준이라면 (즉 분자와 분모가비슷해져) R^2 는 0 에 가까워지고, 타깃이 예측에 아주 가까워지면(분자가 0 에 가까워지기 때문에) 1 에 가까운 값이 된다.

0.99 면 아주 좋은 값이다. 하지만 정확도처럼 R^2 가 직감적으로 얼마나 좋은지 이해하기는 어렵다. 대신 타깃과 예측한 값 사이의 차이를 구해보면 어느정도 예측이 벗어났는지 가늠하기 좋다.

사이킷런은 sklearn.metrics 패키지 아래 여러 측정 도구를 제공하는데, 이 중 mean_absolute_error 는 타깃와 예측의 절댓값 오차를 평균하여 반환한다.

```
#테스트세트에 대한 예측을 만든다
test_prediction=knr.predict(test_input)

#테스트세트에 대한 평균 절대값오차를 계산
```

```
mae=mean_absolute_error(test_target,test_prediction)
print(mae)
19.157142857142862
```

결과에서 예측이 평균적으로 19g 정도 타깃값과 다르다는 것을 알 수 있다.

지금까지는 훈련 세트를 사용해 모델을 훈련하고 테스트세트로 모델을 평가했다. 하지만 훈련세트를 사용해 평가하면 어떻게 될까? 즉 score() 메서드에 훈련 세트를 전달하여 점수를 출력해 보자. 이 값은 테스트 세트의 점수와 다를 것이다.

✓ **과대적합 vs 과소적합**

앞에서 훈련한 모델을 사용해 훈련 세트의 R^2 점수를 확인하자.

```
print(knr.score(train_input,train_target))
0.9698823289099255
```

앞에서 테스트 세트를 사용한 점수와 비교해보자. 어떤 값이 더 높은가?

모델을 훈련 세트에 훈련하면 훈련세트에 잘 맞는 모델이 만들어진다. 이 모델을 훈련 세트와 테스트 세트에서 평가하면, 보통 훈련 세트의 점수가 조금 더 높게 나온다. 훈련 세트에서 모델을 훈련했으므로 훈련 세트에서 더 좋은 점수가 나오는게 당연하다.

만약훈련 세트에서 점수가 굉장히 좋았는데 테스트 세트에서는 점수가 굉장히 나쁘다면 모델이 훈련 세트에 과대적합(overfitting) 되었다고 말한다. 즉 훈련 세트에만 잘 맞는 모델이라 테스트 세트와 나중에 실전에 투입하여 새로운 샘플에 대한 예측을 만들 때 잘 동작하지 않을 것이다.

반대로훈련 세트보다 테스트 세트의 점수가 높거나 두 점수가 모두 너무 낮은 경우를 모델이 훈련 세트에 과소적합(underfitting) 되었다고 말한다. 즉 모델이 너무 단순하여 훈련 세트에 적절히 훈련되지 않은 경우이다. 훈련세트가 전체 데이터를 대표한다고 가정하기 때문에 훈련세트를 잘 학습하는 것이 중요하다

과소적합이 일어나는 또 다른 원인은 훈련 세트와 테스트 세트의 크기가 매우 작기 때문. 데이터가 작으면 테스트 세트가 훈련세트의 특징을 따르지 못할 수 있다.

앞서 k-최근접 이웃 회귀로 평가한 결과, 훈련 세트보다 테스트 세트의 점수가 높아 과소적합이다.

이 문제를 해결하기 위해, 모델을 조금 더 복잡하게 만들 수 있다. 즉 훈련세트에 더 잘 맞게 만들면 테스트 세트의 점수는 조금 낮아질 것이다. k-최근접 이웃 알고리즘으로 모델을 더 복잡하게 만드는 방법은이웃의 개수 k 를 줄이는 것이다.이웃의 개수를 줄이면 훈련 세트에 있는 국지적인 패턴에 민감해지고, 이웃의 개수를 늘리면 데이터 전반에 있는 일반적인 패턴을 따를것이다. 사이킷런의 k-최근접 이웃 알고리즘의 기본 k 값은 5 이다. 이를 3 으로 낮추었더니 (n_neighbors 속성값을 바꾸면 됨), 훈련 세트의 R^2 점수가 높아졌다.

```
knr.n_neighbors=3

#모델을 다시 훈련한다
knr.fit(train_input,train_target)
print(knr.score(train_input,train_target))
0.9804899950518966
```

테스트 세트의 점수는 훈련 세트보다 낮아졌으므로 과소적합 문제를 해결한 것 같다. 또한 두 점수의 차이가 크지 않으므로 과대적합 된 것 같지도 않다. 이 모델이 테스트 세트와 추가될 농어 데이터에도 일반화를 잘하리라 예상할 수 있다.

```
print(knr.score(test_input,test_target))
0.974645996398761
```

회귀 문제 다루기 문제해결 과정

농어의 높이, 길이 등의 수치로 무게를 예측해 달라고 요청.이 문제는 분류가 아니라 회귀 문제이다. 회귀는 임의의 수치를 예측하는 문제이다.농어의 길이를 사용해 무게를 예측하는 k-최근접 이웃 회귀 모델을 만들었다.

k-최근접 이웃 회귀 모델은 분류와 동일하게 가장 먼저 가까운 k 개의 이웃을 찾는다.그 다음 이웃 샘플의 타깃값을 평균하여 이 샘플의 예측값으로 사용함.

사이킷런은 회귀 모델의 점수로 R2, 즉 결정계수 값을 반환함. 이 값은 1 에 가까울수록좋다. 정량적인 평가를 하고 싶다면 사이킷런에서 제공하는 다른 평가 도구를 사용할수있다. 대표적으로 절대값 오차가 있다

모델을 훈련하고 나서 훈련세트와 테스트 세트에 대해 모두 평가 점수를 구할 수 있다.훈련 세트의 점수와 테스트 세트의 점수 차이가 크면 좋지 않다. 일반적으로 훈련세트의점수가 테스트 세트보다 조금 더 높다. 만약 테스트 세트의 점수가 너무 낮다면 모델이훈련세트에 과도하게 맞춰진 것이다. 이를 과대적합이라고 함. 반대로 테스트세트 점수가너무 높거나 두 점수가 모두 낮으면 과소적합이다

과대적합일 경우 모델을 덜 복잡하게 만들어야 함. k-최근접 이웃의 경우 k 값을 늘린다.과소적합일 경우 모델을 더 복잡하게 만들어야 함. k-최근접 이웃의 경우 k 값을 줄이는 것이다.

[선형 회귀]

길이가 50cm 인 농어의 무게 예측. 실제 무게와 너무 차이가 나는 결과가 나옴!

- ✓ **k-최근접 이웃의 한계**

앞서 사용한 데이터와 모델을 준비한다.

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

perch_length=np.array([8.4,13.7,15.0,16.2,17.4,18.0,18.7,19.0,19.6,20.0,21
.0,
21.0,21.0,21.3,22.0,22.0,22.0,22.0,22.0,22.5,22.5,22.7,
23.0,23.5,24.0,24.0,24.6,25.0,25.6,26.5,27.3,27.5,27.5,
27.5,28.0,28.7,30.0,32.8,34.5,35.0,36.5,36.0,37.0,37.0,
39.0,39.0,39.0,40.0,40.0,40.0,40.0,42.0,43.0,43.0,43.5,
44.0])

perch_weight=np.array([5.9,32.0,40.0,51.5,70.0,100.0,78.0,80.0,85.0,85.0,1
10.0,
115.0,125.0,130.0,120.0,120.0,130.0,135.0,110.0,130.0,
150.0,145.0,150.0,170.0,225.0,145.0,188.0,180.0,197.0,
218.0,300.0,260.0,265.0,250.0,250.0,300.0,320.0,514.0,
556.0,840.0,685.0,700.0,700.0,690.0,900.0,650.0,820.0,
850.0,900.0,1015.0,820.0,1100.0,1000.0,1100.0,1000.0,
1000.0])
```

이번에도 데이터를 훈련 세트와 테스트 세트로 나눈다. 특성 데이터는 2 차원 배열로 변환한다.

```
# 훈련세트와테스트세트로나눈다
train_input,test_input,train_target,test_target=train_test_split(perch_length,perch_weigh
t,random_state=42)

#훈련세트와테스트세트를 2 차원배열로바꾼다
train_input=train_input.reshape(-1,1)
test_input=test_input.reshape(-1,1)
```

최근접 이웃 개수를 3 으로 하는 모델을 훈련한다. 앞에서 했던 내용 그대로이다.

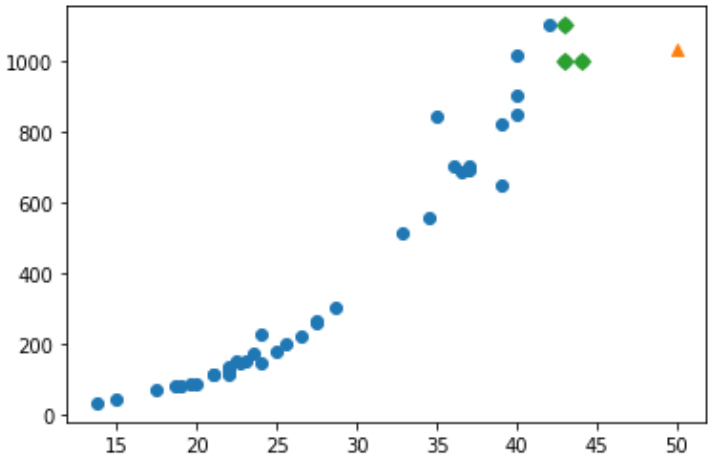
```
knr=KNeighborsRegressor(n_neighbors=3)

#k-최근접이웃회귀모델을훈련한다.
knr.fit(train_input,train_target)

#길이가 50cm 인농어의무게를예측한다.
print(knr.predict([[50]]))

[1033.33333333]
```

k-최근접 이웃 모델의 훈련 결과, 50cm 인 농어의 무게를 1,033g 정도로 예측했다.
하지만 실제 이 농어의 무게는 훨씬 더 많이 나간다.

훈련세트와 50cm 농어, 이 농어의 최근접 이웃을 산점도에 표시해보자. 사이킷런의 k-최근접 이웃 모델의 kneighbors() 메서드를 사용하면 가장 가까운 이웃까지의 거리와 이웃 샘플의 인덱스를 얻을 수 있다.

```
#50cm 농어의이웃을구한다
distance,indexes=knr.kneighbors([[50]])

# 훈련세트의산점도를그린다
plt.scatter(train_input,train_target)

# 훈련세트중에서이웃샘플만다시그린다
plt.scatter(train_input[indexes],train_target[indexes],marker='D')

# 50cm 농어데이터
```

```
plt.scatter(50,1033,marker='^')
plt.xlabel('length')
plt.ylabel('weight')
plt.show()
```



이 산점도를 보면 길이가 커질수록 농어의 무게가 증가하는 경향이 있다. 하지만 50cm 농어에서 가장 가까운 것은 45cm 근방이기 때문에 k-최근접 이웃 알고리즘은 이 샘플들의 무게를 평균한다. 이웃 샘플의 타깃의 평균을 구해보자

```
print(np.mean(train_target[indexes]))
1033.33333333333333
```

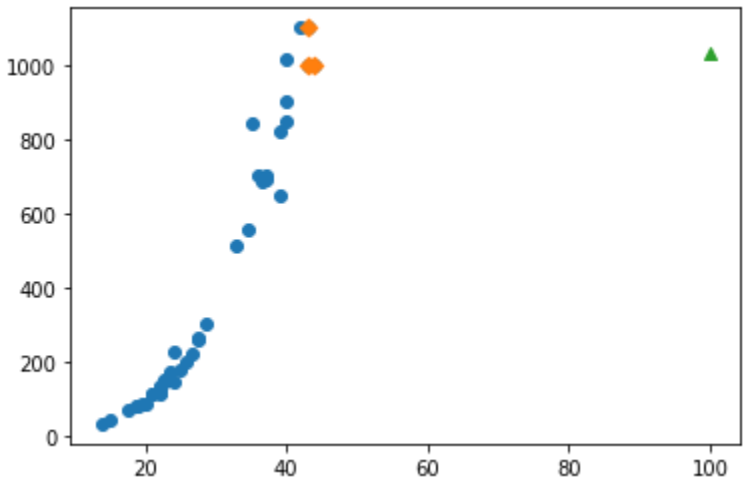
이웃 샘플의 타깃의 평균을 출력하면 모델이 예측했던 값과 일치한다.k-최근접 이웃 회귀는 가장 가까운 샘플을 찾아 타깃을 평균하기 때문에, 새로운 샘플이 훈련 세트의 범위를 벗어나면 엉뚱한 값을 예측할 수 있다. 예를 들어 길이가 100cm 인 농어도 여전히 1,033g 으로 예측함

```
print(knr.predict([[100]]))
[1033.33333333]
```

그래프를 그려 확인해보자

```
distance,indexes=knr.kneighbors([[100]])
plt.scatter(train_input,train_target)
plt.scatter(train_input[indexes],train_target[indexes],marker='D')
plt.scatter(100,1033,marker='^')
plt.xlabel('length')
```

```
plt.ylabel('weight')
plt.show()
```



k-최근접 이웃을 사용해 이 문제를 해결하려면 가장 큰 농어가 포함되도록 훈련 세트를 다시 만들어야 한다.

k-최근접 이웃 말고 다른 알고리즘을 찾아보자.위와 같은 문제가 계속되면 농어가 아무리 커져도 무게가 더 늘어나지는 않는다. 이를 해결하기 위해서는 선형회귀를 사용해야 한다.

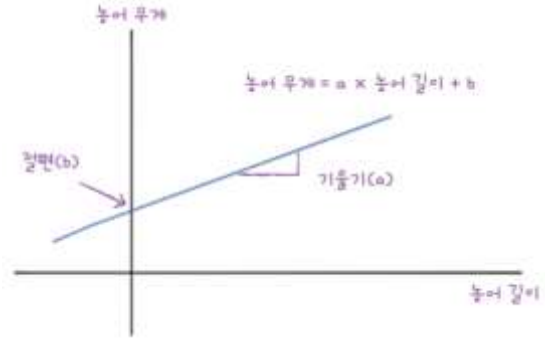
✓ **선형 회귀**

- **선형 회귀**는 널리 사용되는 대표적인 회귀알고리즘이다. 비교적 간단하고 성능이 뛰어나기 때문이다.
- 선형이라는 말에서 짐작할 수 있듯이 특성이 하나인 경우 어떤 직선을 학습하는 알고리즘을 말한다.
- 사이킷런은 sklearn.liner_model 패키지 아래에 LinearRegression 클래스로 선형 회귀 알고리즘을 구현해 놓았다.
- 사이킷런의 모델 클래스들은 훈련, 평가, 예측하는 메서드 이름이 모두 동일하다. 즉 LinearRegression 클래스에도 fit(), score(), predict()메서드가 있다.

```
lr = LinearRegression()

# 선형회귀 모델을 훈련한다
lr.fit(train_input, train_target)
#50 농어에 대해 예측한다
```

```
print(lr.predict([[50]]))
[1241.83860323]
```

- 50cm 의 농어 무게를 이전에 K-최근접 이웃회귀를 사용했을 때 보다 높게 예측되었다. 이 선형 회귀가 학습한 직선을 그려보고 어떻게 이런 값이 나왔는지 알아보자
- 하나의 직선을 그리려면 기울기와 절편이 있어야 한다. y= a * x+b 에서 x 는 농어의 길이 y 는 농어의 무게로 볼 수 있다.



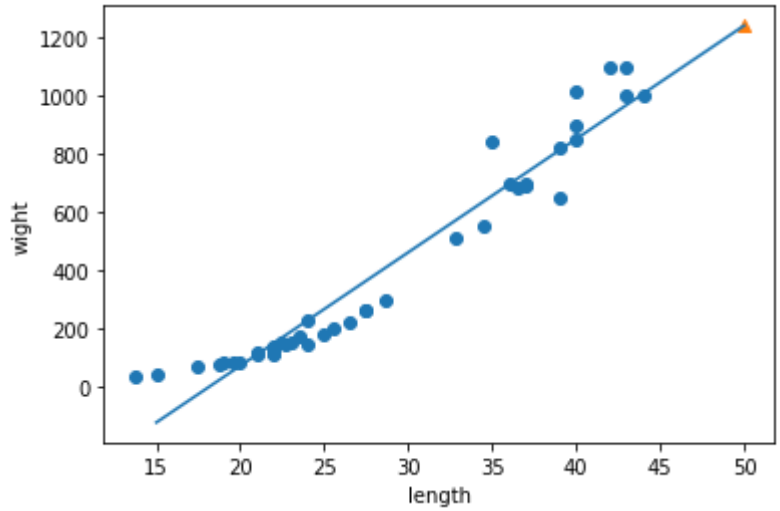
- 여기서 LinearRegression 은 적절한 a, b 를 찾는다. 이는 lr 객체에 coef_와 intercept_속성에 저장되어 있다.

```
print(lr.coef_, lr.intercept_)
[39.01714496] -709.0186449535477
```

- coef_와 intercept_를 머신러닝 알고리즘이 찾은 값이라는 의미로 **모델 파라미터**라고 부른다. 이 모델 파라미터를 가지는 모델들은 모델 기반 학습이라 한다.
- 모델파라미터가 없는 k-최근접 이웃 같은 모델은 사례기반 학습이라고 한다.
- 농어의 길이 15 에서 50 까지 직선으로 그려보자. 이 직선을 그리려면 앞에서 구한 기울기와 절편을 사용하여 (15, 15*39-709) 와 (50, 50*39-709) 두 점을 이으면 된다. 훈련세트의 산점도와 함께 그려보자

```
#훈련세트의 산점도를 그린다
plt.scatter(train_input, train_target)

#15 에서 50 까지 1 차 방정식 그래프를 그린다
plt.plot([15, 50], [15*lr.coef_+lr.intercept_, 50*lr.coef_+lr.intercept_])
```

```
# 50cm 농어 데이터
plt.scatter(50, 1241.8, marker="^")
plt.xlabel('length')
plt.ylabel('weight')
plt.show()
```



- 이 직선이 선형 회귀 알고리즘이 해당 데이터셋에서 찾은 최적의 직선이다. 길이가 50cm 인 농어에 대한 예측은 이 직선의 연장선에 있다. 이제 훈련 세트 범위를 벗어난 농어의 무게도 예측할 수 있다. 그럼 훈련세트와 테스트세트에 대한 결정계수값을 찾아보자.

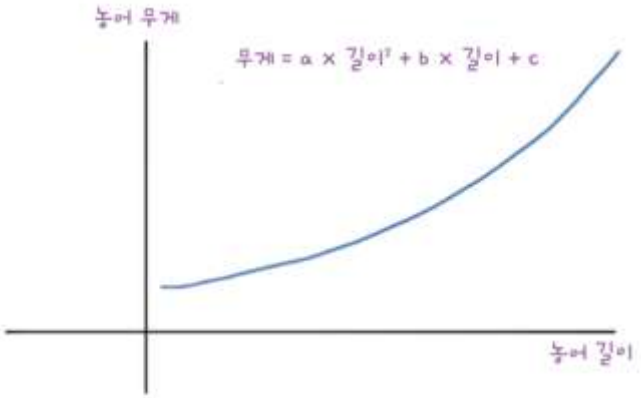
```
print(lr.score(train_input, train_target))
print(lr.score(test_input, test_target))
0.9398463339976041
0.824750312331356
```

- 훈련세트와 테스트 세트는 점수차가 난다. 이는 훈련 세트 또한 점수가 낮기 때문에 과소적합의 문제가 있어 보인다. 과소적합 뿐 아니라 다른 문제도 있다.

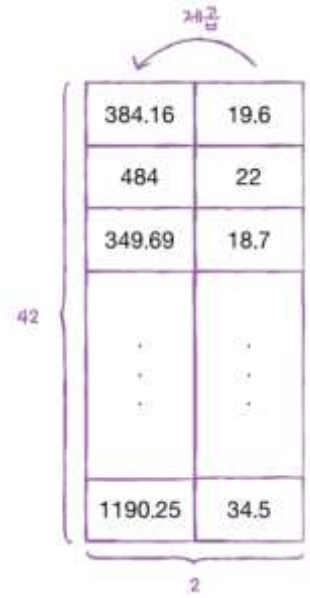
✓ 다항회귀

- 이 그래프에서 선형회귀가 만든 직선이 왼쪽 아래로 쭉 뻗어 있다. 이 직선대로 예측하면 농어의 무게가 0g 이하로 내려갈 텐데 현실에서는 있을 수 없는 일이다.
- 농어 길이와 무게에 대한 산점도를 보면 일직선이라기보다는 왼쪽 위로 조금 구부러진 곡선에 가깝다.

- 최적의 직선보다는 최적의 곡선을 찾는 것이 좀 더 바람직해 보인다.



- 이런 2 차 방정식의 그래프를 그리려면 길이를 제곱한 항이 훈련세트에 추가되어야 한다. 넘파이를 이용해 추가해주자. (농어의 길이를 제곱해서 원래 데이터 앞에 붙여보자)
- column_stack() 함수를 사용하면 아주 간단하다. train_input 을 제곱한 것과 train_input 두 배열을 나란히 붙이면 된다.




```
train_poly = np.column_stack((train_input **2, train_input))
test_poly = np.column_stack((test_input **2, test_input))
print(train_poly[:5])

[[ 384.16     19.6 ]
 [ 484.       22.  ]
```

```
 [ 349.69      18.7 ]
 [ 302.76      17.4 ]
 [1296.        36.  ]]

# 새롭게 만든 데이터셋의 크기 확인
print(train_poly.shape)
print(test_poly.shape)

(42, 2)
(14, 2)
```

- 원래 특성인 길이를 제곱해 왼쪽 열에 추가했기 때문에 훈련 세트와 테스트 세트 모두 열이 2 개로 늘어난 것을 확인할 수 있다.(넘파이 브로드캐스팅이 적용되어 모든 원소를 제곱해 넣어주었다.)
- train_poly 를 사용해 선형회귀 모델을 다시 훈련하자. 이 모델이 2 차 방정식의 a,b,c 를 잘 찾을 것이다. 훈련세트에 제곱항을 추가했지만, 타깃값은 그대로 사용한다. 목표하는 값은 어떤 그래프를 훈련하든지 바꿀 필요가 없다. 이 훈련세트로 선형회귀 모델을 훈련한 다음 50cm 농어에 대해 무게를 예측해보자. 훈련세트에서 했던 것처럼 테스트할 때는 이 모델에 농어길이의 제곱과 원래 길이를 함께 넣어주어야 한다.

```
lr=LinearRegression()
lr.fit(train_poly, train_target)
print(lr.predict([[50**2, 50]]))

[1573.98423528]
```

- 앞에서의 결과보다 더 큰 무게로 예측했다. 이 모델이 훈련한 계수와 절편을 출력하면 아래와 같다.

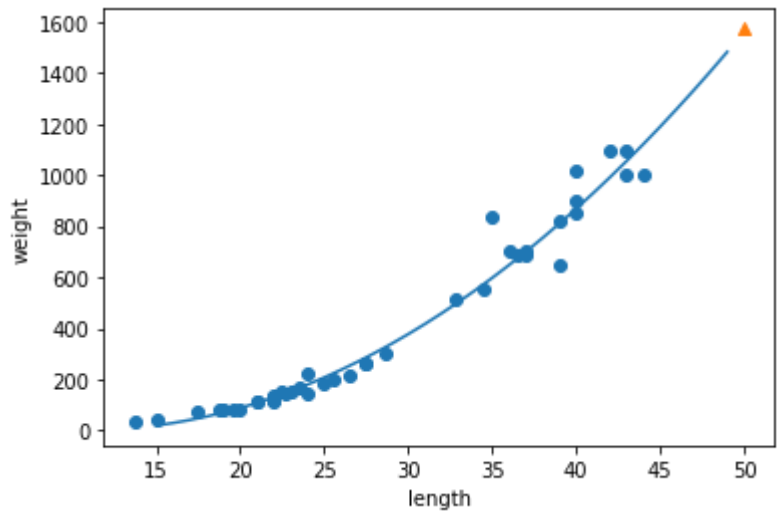
```
print(lr.coef_, lr.intercept_)

[   1.01433211 -21.55792498] 116.0502107827827
```

- 이 모델은 다음과 같은 그래프를 학습했다.

$$무게 = 1.01 \times 길이^2 - 21.6 \times 길이 + 116.05$$

- 위의 모델은 2 차 방정식이라 비선형으로 볼 수도 있다. 제곱의 길이를 다른 변수로 치환해 사용하면 선형이라고 볼 수 있다. 이런 방정식을 **다항식**이라 부르며, 다항식을 사용한 선형 회귀를 **다항회귀**라고 부른다.
- 이를 이용해 다시 산점도를 그려보자. 짧은 직선을 이어서 그리면 마치 곡선처럼 표현할 수 있다. 여기에서는 1 씩 짧게 끊어서 그려보겠다.

```
# 구간별 직선을 그리기 위해 15 에서 49 까지 정수 배열을 만든다
point = np.arange(15, 50)

#훈련세트의 산점도를 그린다
plt.scatter(train_input, train_target)

#15 에서 49 까지 2 차 방정식 그래프를 그린다
plt.plot(point, 1.01*point**2-21.6*point +116.05)

#50cm 농어 데이터
plt.scatter(50, 1574, marker='^')
plt.xlabel('length')
plt.ylabel('weight')
plt.show()
```



단순 선형 회귀 모델보다 훨씬 나은 그래프가 그려졌다. 훈련세트의 경향을 잘 따르고 있고 무게가 음수로 나오는 일도 없을 것 같다. 그럼 훈련세트와 테스트 세트의 R^2 점수를 평가해보자.

```
print(lr.score(train_poly, train_target))
print(lr.score(test_poly, test_target))

0.9706807451768623
0.9775935108325122
```

- 테스트세트와 훈련세트의 점수가 이전보다 상승했다. 하지만 여전히 테스트 세트의 점수가 더 높은 것으로 볼 때, **과소적합**문제가 아직 남아있다는 것을 알 수 있다. 그럼 조금 더 복잡한 모델이 필요할 것 같다.

선형 회귀로 훈련세트 범위 밖의 샘플 예측 문제 해결 과정

k-최근접 이웃 회귀를 사용해서 농어의 무게를 예측했을 때 발생하는 큰 문제는 훈련세트 범위 밖의 샘플을 예측할 수 없다는 점이다. k-최근접 이웃 회귀는 아무리 멀리 떨어져 있더라도 무조건 가장 가까운 샘플의 타깃을 평균하여 예측한다.

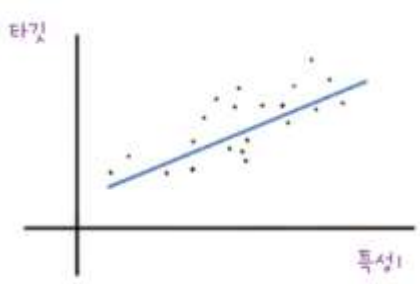
이 문제를 해결하기 위해 선형회귀를 사용했다. 선형 회귀는 훈련세트에 잘 맞는 직선의 방정식을 찾는 것이다. 사이킷런의 LinearRegression 클래스를 사용하면 k-최근접 이웃 알고리즘을 사용했을 때와 동일한 방식으로 모델을 훈련하고 예측에 사용할 수 있다.

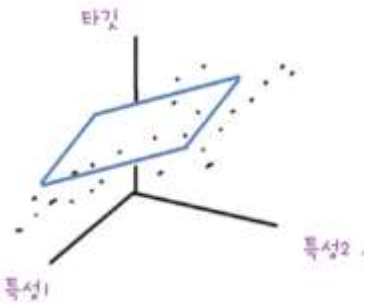
가장 잘 맞는 직선의 방정식을 찾는다는 것은 최적의 기울기와 절편을 구한다는 의미이다. 이 값들은 선형회귀 모델의 coef_와 intercept_ 속성에 저장되어 있다.
선형회귀 모델은 k-최근접 이웃 회귀와 다르게 훈련 세트를 벗어난 범위의 데이터도 잘 예측했다. 하지만 모델이 단순하여 농어의 무게가 음수일 수도 있다.

이를 해결하기 위해 다항회귀를 사용했다. 간단히 농어의 길이를 제곱하여 훈련세트에 추가한 다음 선형회귀 모델을 다시 훈련했다. 이 모델은 2 차 방정식의 그래프 형태를 학습하였고 훈련세트가 분포된 형태를 잘 표현했다. 또 훈련세트와 테스트세트의 성능이 단순한 선형회귀보다 훨씬 높아졌다. 하지만 훈련세트 성능보다 테스트 세트 성능이 조금 높은 것으로 보아 과소적합된 경향이 아직 남았다. 조금 더 복잡한 모델을 만들어 이 문제를 해결해야 한다. 또한 너무 복잡한 모델일 경우, 즉 과대적합된 모델을 반대로 억제하는 방법도 있다.

[특성 공학과 규제]

- 이전에 다항회귀로 농어의 무게를 어느 정도 예측이 가능했지만, 여전히 훈련 세트보다 테스트 세트의 점수가 더 높은 문제가 있었다.
- 선형회귀는 특성이 많을수록 효과가 좋다. 기존에는 1 개의 특성(길이)을 사용하는 직선형 모델이였다. 2 개의 특성을 사용하게 되면(높이와 두께를 다항회귀에 함께 적용)선형회귀는 평면을 학습한다. 이처럼 특성이 많은 고차원에서의 선형회귀는 매우 복잡한 모델을 표현할 수 있다.





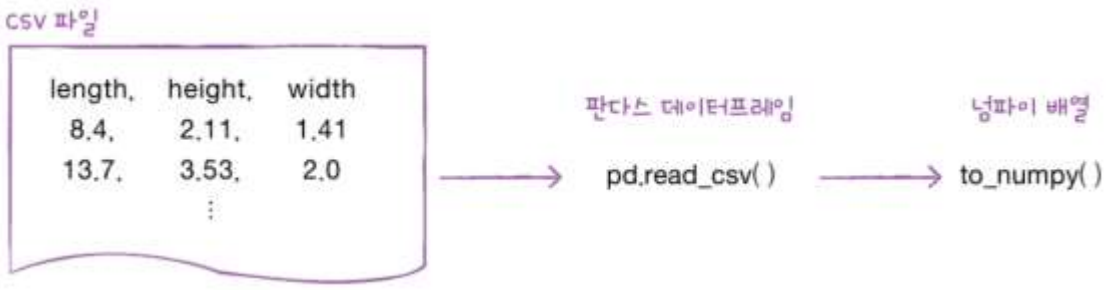
- 특성이 2 개면 타깃값과 함께 3 차원 공간을 형성하고 선형회귀 방정식 '타깃=a*특성 1 + b*특성 2 + 절편' 은 평면이 된다.
- 이전에는 하나의 특성을 사용하여 선형 회귀 모델을 훈련시켰다. 여러 개의 특성을 사용한 선형회귀를 다중회귀(multiple regression)라고 부른다.
- 이번에는 농어의 길이뿐만 아니라 농어의 높이와 두께도 함께 사용한다. 이전처럼 3 개의 특성을 각각 제곱하여 추가한다. 거기다가 각 특성을 서로 곱해서 또 다른 특성을 만들겠다. 즉 농어길이*농어높이 를 새로운 특성으로 만든다.
- 이렇게 기존의 특성을 사용해 새로운 특성을 뽑아내는 작업을 **특성공학**(feature engineering)이라고 부른다.
- 사이킷런에서 제공하는 도구를 사용해서 새로운 특성을 추가할 것이다.

✓ **데이터 준비**

이전과 달리 농어의 특성이 3 개로 늘어났기 때문에 데이터를 복사해 붙여 넣는 것이 번거롭다. 판다스(pandas) 를 사용하면 간단히 할 수 있다. 판다스는 유명한 데이터 분석 라이브러리이다. **데이터프레임(dataframe)** 은 판다스의 핵심 데이터 구조이다. 넘파이 배열과 비슷하게 다차원 배열을 다룰 수 있지만 훨씬 더 많은 기능을 제공한다. 또 데이터프레임은 넘파이 배열로 쉽게 바꿀 수도 있다.

판다스를 사용해 농어 데이터를 인터넷에서 내려 받아 데이터프레임에 저장한 다음 넘파이 배열로 변환하여 선형 회귀모델을 훈련하겠다. 판다스 데이터프레임을 만들기 위해 많이 사용하는 파일은 CSV 파일이다.

판다스의 read_csv() 함수로 데이터프레임을 만든 다음 to_numpy() 메서드를 사용해 넘파이 배열로 바꾼다.



그 다음 perch_full 과 perch_weight 를 훈련 세트와 테스트 세트로 나눈다.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
import matplotlib.pyplot as plt
root='https://bit.ly/perch_csv_data'
df=pd.read_csv(root)
perch_full=df.to_numpy()
print(perch_full[:5])

[[ 8.4   2.11  1.41]
 [13.7   3.53  2.  ]
 [15.    3.82  2.43]
 [16.2   4.59  2.63]
 [17.4   4.59  2.94]]

perch_weight=np.array([5.9,32.0,40.0,51.5,70.0,100.0,78.0,80.0,85.0,85.0,110.0,
```

```
115.0,125.0,130.0,120.0,120.0,130.0,135.0,110.0,130.0,
150.0,145.0,150.0,170.0,225.0,145.0,188.0,180.0,197.0,
218.0,300.0,260.0,265.0,250.0,250.0,300.0,320.0,514.0,
556.0,840.0,685.0,700.0,700.0,690.0,900.0,650.0,820.0,
850.0,900.0,1015.0,820.0,1100.0,1000.0,1100.0,1000.0,
1000.0])
train_input,test_input,train_target,test_target=train_test_split(perch_full,perch_weight,rando
m_state=42)
```

- 이 데이터를 사용해 새로운 특성을 만들겠다.

✓ **사이킷런의 변환기**

- 사이킷런은 특성을 만들거나 전처리하기 위한 다양한 클래스를 제공한다. 이런 클래스를 **변환기**라고 부른다. 사이킷런의 모델 클래스에 일관된 fit(), score(), predict()메서드가 있는 것처럼 변환기 클래스는 fit(), transform()메서드를 제공한다.
- 우리가 사용할 변환기는 PolynomialFeatures 클래스이다. 이 클래스는 sklearn.preprocessing 패키지에 포함되어 있다.
- 2 개의 특성, 2 와 3 으로 이루어진 샘플 하나를 적용하겠다.
- 이 클래스의 객체를 만든 다음 fit(), transform()메서드를 차례대로 호출한다.

```
poly=PolynomialFeatures()
poly.fit([[2,3]])
print(poly.transform([[2,3]]))

[[1. 2. 3. 4. 6. 9.]]
```

- fit() 메서드는 새롭게 만들 특성 조합을 찾고 transform() 메서드는 실제로 데이터를 변환한다. 변환기는 입력 데이터를 변환하는 데 타깃 데이터가 필요하지 않다. 따라서 모델 클래스와는 다르게 fit()메서드에 입력 데이터만 전달했다. 즉 여기에서는 2 개의 특성(원소)을 가진 샘플[2,3]이 6 개의 특성을 가진 샘플 [1. 2. 3. 4. 6. 9.]로 바뀌었다.
- PolynomialFeatures 클래스는 기본적으로 각 특성을 제곱한 항을 추가하고 특성끼리 서로 곱한 항을 추가한다.
- 1 이 추가된 이유

- 무게 = a*길이+b*높이 + c*두께 + d*1
- 선형 방정식의 절편을 항상 값이 1 인 특성과 곱해지는 계수라고 볼 수 있다. 특성은 (길이, 높이, 두께, 1)이 된다. 하지만 사이킷런의 선형모델은 자동으로 절편을 추가하므로 굳이 이렇게 특성을 만들 필요가 없다.
- 특성이 (길이, 높이, 두께, 1)이 되는 것을 방지하기 위해, include_bias=False 로 지정하여 다시 특성을 변환해준다. 절편을 위한 항이 제거되고 특성의 제곱과 특성끼리 곱한 항만 추가된다.

```
poly=PolynomialFeatures(include_bias=False)
poly.fit([[2,3]])
print(poly.transform([[2,3]]))

[[2. 3. 4. 6. 9.]]
```

- 이제 이 방식으로 훈련 데이터에 적용해보겠다.

```
poly=PolynomialFeatures(include_bias=False)
poly.fit(train_input)
train_poly=poly.transform(train_input)
print(train_poly.shape)

(42, 9)
```

- 9 개의 특성을 가지는 훈련셋을 만들었다. 해당 특성이 어떻게 만들어 졌는지 알아보고 싶을 경우 get_feature_names() 메서드를 호출하면 9 개의 특성이 각각 어떤 입력의 조합으로 만들어졌는지 알 수 있다.

```
poly.get_feature_names_out()

['x0', 'x1', 'x2', 'x0^2', 'x0 x1', 'x0 x2', 'x1^2', 'x1 x2', 'x2^2']
```

- 'x0'는 첫번째 특성을 의미하고 'x0^2'는 첫 번째 특성의 제곱, 'x0 x1'은 첫 번째 특성과 두 번째 특성의 곱을 나타내는 형식으로 만들어졌다.
- 테스트 세트도 변환해 준다. 이어서 변환된 특성을 사용하여 다중회귀 모델을 훈련한다.

```
test_poly=poly.transform(test_input)
lr=LinearRegression()
```

```
lr.fit(train_poly,train_target)
print(lr.score(train_poly,train_target))
print(lr.score(test_poly,test_target))

0.9903183436982124
0.9714559911594134
```

- 특성이 늘어나면서 점수가 아주 높아졌다. 농어의 길이 뿐 아니라 높이와 두께 모두 사용했고, 각 특성들을 제곱하거나 곱해서 다항 특성을 더 추가했다.
- 이로써 과소적합의 문제를 해결했다.
- 이를 통해 특성이 늘어나면 선형 회귀 능력은 매우 강하다는 것을 알 수 있다.
- 만약 특성을 더 많이 추가하면 어떻게 될까? 3 제곱, 4 제곱항을 넣는거다. PolynomialFeatures 클래스의 degree 매개변수를 사용하여 필요한 고차항의 최대 차수를 지정할 수 있다. 5 제곱까지 특성을 만들어 출력해 보겠다.

```
poly=PolynomialFeatures(degree=5,include_bias=False)
poly.fit(train_input)
train_poly=poly.transform(train_input)
test_poly=poly.transform(test_input)
print(train_poly.shape)

(42, 55)
```

- 이를 통해 만들어진 특성이 무려 55 개나 된다. train_poly 배열의 열의 개수가 특성의 개수이다. 이를 이용해 다시 선형회귀모델을 훈련시켜보도록 한다.

```
lr.fit(train_poly,train_target)
print(lr.score(train_poly,train_target))
print(lr.score(test_poly,test_target))

0.9999999999991097
-144.40579242684848
```

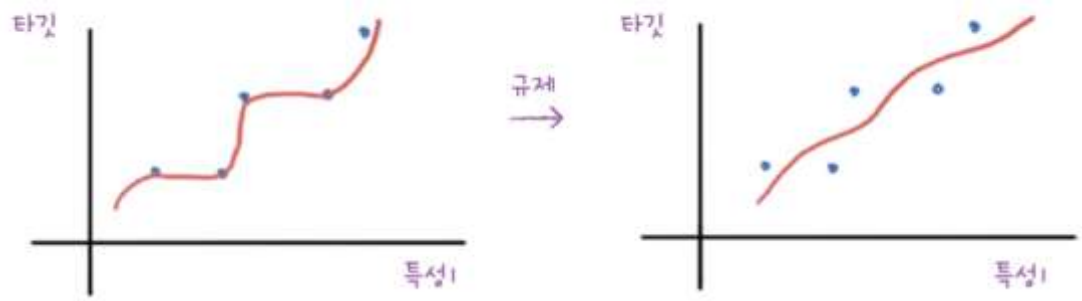
- 결과를 살펴보면 훈련세트에 대한 점수는 아주 좋다. 하지만 반대로 테스트세트의 점수는 아주 큰 음수가 나온다.
- 이를 통해 알 수 있는 것들은 다음과 같다.

1. 특성의 개수를 늘리면 선형 모델은 아주 강력해 진다.
2. 훈련세트에 대해 거의 완벽하게 학습할 수 있다.
3. 하지만 훈련세트에 너무 과대적합되므로 테스트 세트에서는 점수가 잘 나오지 않는다.

- 문제를 해결하기 위해서는 특성을 다시 줄이는 방법이 있다. 하지만 이외에도 또 다른 방법이 존재한다.

3. 규제

- **규제**는 머신러닝 모델이 훈련 세트를 너무 과도하게 학습하지 못하도록 훼방하는 것을 말한다.즉 모델이 훈련 세트에 과대적합되지 않도록 만드는 것이다.
- 선형 회귀 모델에서는 특성에 곱해지는 계수(or 기울기)의 크기를 작게 만드는 것을 말한다.



왼쪽은 훈련세트를 과도하게 학습했고, 오른쪽은 기울기를 줄여 보다 보편적인 패턴을 학습하고 있다.

- 앞서 55 개의 특성으로 훈련한 선형 회귀 모델의 계수를 규제하여 훈련 세트의 점수를 낮추고 대신 테스트 세트의 점수를 높여보도록 하겠다.

- 여기서 일단 특성의 스케일을 고려해 보아야 한다. 특성의 스케일이 정규화 되지 않으면 여기에 곱해지는 계수 값도 차이가 나기 때문이다. 선형회귀 모델에 규제를 적용할 때 계수 값의 크기가 서로 많이 다르면 공정하게 제어되기 힘들기 때문이다.

- 규제를 적용하기 전에 먼저 정규화를 해야 한다. 이전에는 평균과 표준편차를 직접 구해 특성을 표준점수로 바꾸었다. 이번에는 사이킷런에서 제공하는 StandardScaler 클래스를 사용하여 정규화를 수행하겠다.

```
ss=StandardScaler()
ss.fit(train_poly)
train_scaled=ss.transform(train_poly)
test_scaled=ss.transform(test_poly)
```

- 먼저 StandardScaler 클래스의 객체 ss 를 초기화한 후 PolynomialFeatures 클래스로 만든 train_poly 를 사용해 이 객체를 훈련한다.(테스트 세트도 같이 해줘야 한다)
- 이제 표준점수로 변환한 train_scaled 와 test_scaled 가 준비되었다.
- 선형 회귀 모델에 규제를 추가한 모델을 **릿지**, **라쏘**라고 부른다.
- 릿지는 계수를 제곱한 값을 기준으로 규제를 적용한다.
- 라쏘는 계수의 절댓값을 기준으로 규제를 적용한다.
- 일반적으로는 릿지를 조금 더 선호하는 경향이 있다. 두 알고리즘 모두 계수의 크기를 줄이지만 라쏘는 아예 0 으로 만들 수도 있다.

✓ 릿지 회귀

릿지와 라쏘 모두 sklearn.linear_model 패키지 안에 있다. 모델 객체를 만들고 fit() 메서드에서 훈련한 다음 score() 메서드로 평가한다.
앞서 준비한 train_scaled 데이터로 릿지 모델을 훈련해 보겠다.

```
ridge=Ridge()
ridge.fit(train_scaled,train_target)
print(ridge.score(train_scaled,train_target))
print(ridge.score(test_scaled,test_target))

0.9896101671037343
0.9790693977615386
```

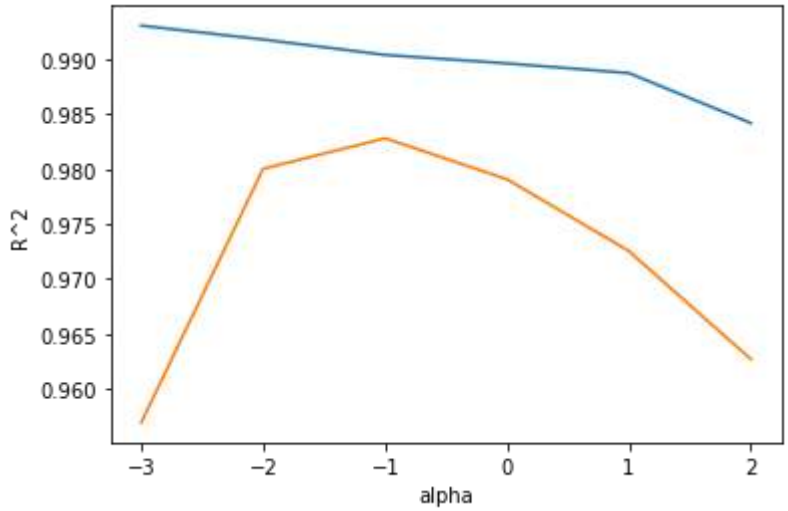
- 선형 회귀에서 거의 완벽에 가까웠던 점수가 조금 낮아졌다. 테스트 세트 점수도 정상으로 돌아왔다.

- 무조건 많은 특성을 사용한다 해서 좋은 성능을 내는 것은 아니다.
- 릿지와 라쏘 모델을 사용할 때 규제의 양을 임의로 조절할 수 있다. 모델 객체를 만들 때 alpha 매개변수로 규제의 강도를 조절한다. alpha 값이 크면 규제 강도가 세져 계수 값을 더 줄이고, 조금 더 과소적합되도록 유도한다. 반대로 alpha 값이 작으면 계수를 줄이는 역할이 줄고, 선형회귀 모델과 유사해지므로 과대적합될 가능성이 크다.
- alpha 값은 릿지 모델이 학습하는 값이 아니라 사전에 우리가 지정해야 하는 값이다. 이렇게 머신러닝 모델이 학습할 수 없고 사람이 알려줘야 하는 파라미터를 하이퍼파라미터라고 부른다.
- 적절한 alpha 값을 찾는 한 가지 방법은 alpha 값에 대한 결정계수(R^2)값의 그래프를 그려보는 것이다.
- 훈련세트와 테스트세트의 점수가 가장 가까운 지점이 최적의 alpha 값이 된다.
- 먼저 맷플롯립을 임포트하고 alpha 값을 바꿀때마다 score()메서드의 결과를 저장할 리스트를 만든다

```
train_score=[]
test_score=[]
```

- alpha 값을 0.001 에서 100 까지 10 배씩 늘려가며 릿지 회귀 모델을 훈련한 다음 훈련 세트와 테스트 세트의 점수를 파이썬 리스트에 저장한다.

```
alpha_list=[0.001,0.01,0.1,1,10,100]
for alpha in alpha_list:
    # 릿지 모델을 만든다
    ridge=Ridge(alpha=alpha)
    # 릿지 모델을 훈련한다
    ridge.fit(train_scaled,train_target)
    #훈련 점수와 테스트 점수를 저장한다
    train_score.append(ridge.score(train_scaled,train_target))
    test_score.append(ridge.score(test_scaled,test_target))

plt.plot(np.log10(alpha_list),train_score)
plt.plot(np.log10(alpha_list),test_score)
plt.xlabel('alpha')
plt.ylabel('R^2')
plt.show()
```



- alpha 값을 0.001 부터 10 배씩 늘렸기 때문에 이대로 그래프를 그리면 그래프 왼쪽이 너무 촘촘해진다. alpha_list 에 있는 6 개의 값을 동일한 간격으로 나타내기 위해 로그함수로 바꾸어 지수로 표현하겠다. 즉 0.001 은 -3, 0.01 은 -2 가 되는 식이다.
- 위는 훈련세트 그래프이며, 아래는 테스트세트 그래프이다. 적절한 alpha 값은 두 그래프가 가장 가깝고 테스트 세트의 점수가 가장 높은 -1, 즉 0.1 이다.
- alpha 값을 0.1 로 해서 최종 모델을 훈련하자.

```
ridge=Ridge(alpha=0.1)
ridge.fit(train_scaled,train_target)
print(ridge.score(train_scaled,train_target))
print(ridge.score(test_scaled,test_target))

0.9903815817570366
0.9827976465386922
```

- 이 모델은 훈련세트와 테스트세트의 점수가 비슷하게 모두 높고 과대적합과 과소적합 사이에서 균형을 맞추고 있다.

✓ 라쏘 회귀

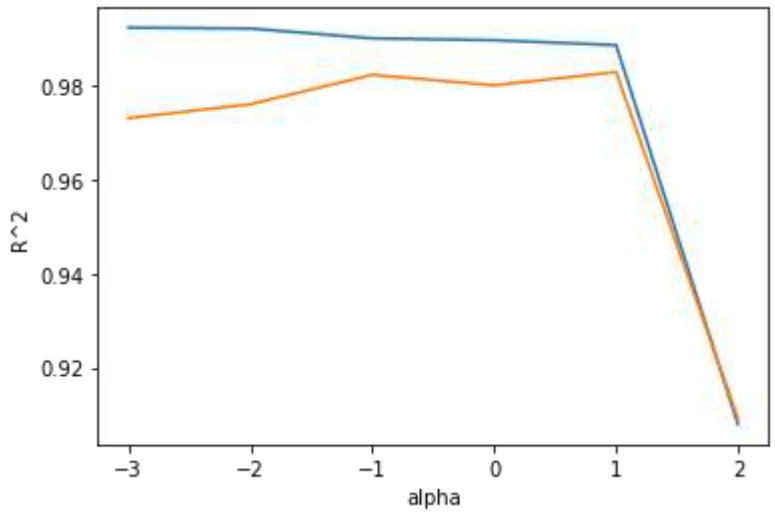
라쏘 모델을 훈련하는 것은 릿지와 매우 비슷하다. Ridge 클래스를 Lasso 클래스로 바꾸는 것이 전부이다.

```
lasso=Lasso()
lasso.fit(train_scaled,train_target)
```

```
print(lasso.score(train_scaled,train_target))
print(lasso.score(test_scaled,test_target))

0.989789897208096
0.9800593698421883
```

- 라쏘도 과대적합을 잘 억제한 결과를 보여준다.
- 테스트 세트의 점수도 릿지만큼 좋은 점수를 보여준다. 라쏘 모델도 alpha 매개변수로 규제의 강도를 조절할 수 있다. 앞에서와 같이 alpha 값을 바꾸어 가며 훈련 세트와 테스트 세트에 대한 점수를 계산하겠다.
- 그 다음 train_score 와 test_score 리스트를 사용해 그래프를 그린다. 이 그래프도 x 축은 로그 스케일로 바꿔 그린다.

```
train_score=[]
test_score=[]
alpha_list=[0.001,0.01,0.1,1,10,100]
for alpha in alpha_list:
    # 라쏘 모델을 만든다
    lasso=Lasso(alpha=alpha,max_iter=10000)
    # 라쏘 모델을 훈련한다
    lasso.fit(train_scaled,train_target)
    # 훈련점수와 테스트점수를 저장한다+
    train_score.append(lasso.score(train_scaled,train_target))
    test_score.append(lasso.score(test_scaled,test_target))

ConvergenceWarning: Objective did not converge. You might want to increase the
number of iterations. Duality gap: 18778.697957792876, tolerance: 518.2793833333334
  model = cd_fast.enet_coordinate_descent...

plt.plot(np.log10(alpha_list),train_score)
plt.plot(np.log10(alpha_list),test_score)
plt.xlabel('alpha')
plt.ylabel('R^2')
plt.show()
```



- 라쏘 모델을 훈련할 때 ConvergenceWarning 이란 경고가 발생할 수 있다. 사이킷런의 라쏘 모델은 최적의 계수를 찾기 위해 반복적인 계산을 수행하는데, 지정한 반복횟수가 부족할 때 이런 경고가 발생함. 이 반복 횟수를 충분히 늘리기 위해 max_iter 매개변수의 값을 10000 으로 지정했다
- 이 그래프도 왼쪽은 과대적합을 보여주고 있고, 오른쪽으로 갈수록 훈련 세트와 테스트 세트의 점수가 좁혀지고 있다. 최적의 alpha 값은 1, 즉 10 이다. 이 값으로 최종 모델을 훈련하겠다.

```
lasso=Lasso(alpha=10)
lasso.fit(train_scaled,train_target)
print(lasso.score(train_scaled,train_target))
print(lasso.score(test_scaled,test_target))

0.9888067471131867
0.9824470598706695
```

- 모델 훈련이 아주 적절하다. 특성을 많이 사용했지만 릿지와 마찬 가지로 라쏘 모델이 과대적합을 잘 억제하고 테스트 세트의 성능을 높여주었다.
- 라쏘 모델은 계수값을 아예 0 으로 만들 수 있다. coef_ 속성에 저장된 계수 중 0 인 것을 확인하겠다.

```
print(np.sum(lasso.coef_==0))

40
```

np.sum() 함수는 배열을 모두 더한 값을 반환함. 넘파이 배열에 비교 연산자를 사용했을 때 각 원소는 True 또는 False 가 됨. np.sum() 함수는 True 를 1 로, False 를 0 으로 인식하여 덧셈을 할 수 있기 때문에 마치 비교연산자에 맞는 원소 개수를 헤아리는 효과를 냄

- 55 개의 특성을 모델에 주입했지만 라쏘 모델이 사용한 특성은 15 개밖에 되지 않는다. 이런 특징 때문에 라쏘 모델을 유용한 특성을 골라내는 용도로도 사용할 수 있다.

모델의 과대적합을 제어하기 위한 문제해결 과정

선형회귀 알고리즘을 사용해 농어의 무게를 예측하는 모델을 훈련시켰지만 훈련 세트에 과소적합되는 문제가 발생했다. 이를 위해 농어의 길이뿐만 아니라 높이와 두께도 사용하여 다중회귀 모델을 훈련시켰다.

또한 다항특성을 많이 추가하여 훈련세트에서 거의 완벽에 가까운 점수를 얻는 모델을 훈련했다. 특성을 많이 추가하면 선형회귀는 매우 강력한 성능을 낸다. 하지만 특성이 너무 많으면 선형 회귀모델을 제약하기 위한 도구가 필요하다.

이를 위해 릿지 회귀와 라쏘 회귀에 대해 알아보았다. 사이킷런을 사용해 다중회귀모델과 릿지, 라쏘 모델을 훈련시켰다. 또 릿지와 라쏘 모델의 규제 양을 조절하기 위한 최적의 alpha 값을 찾는 방법을 알아보았다.