선형회귀의 심화

- **01** 경사하강법의 종류

  02 과대적합과 정규화

  03 사이킷런을 이용한 선형회귀

학습목표

1. 전체-배치 경사하강법, 확률적 경사하강법 (SGD),
   미니-배치 경사하강법에 대해 알아본다.
2. SGD를 파이썬 코드로 작성하는 방법을 실습한다.
3. 과대적합을 극복하는 방법에 대해 알아본다.
4. L2 정규화인 리지 회귀와 L1 정규화인 라쏘 회귀에 대해 학습한다.
5. 사이킷런을 이용하여 선형회귀를 구현한다.

01
경사하강법의 종류

01 경사하강법의 종류

1. 전체-배치 경사하강법

- 전체-배치 경사하강법(full-batch gradient descent) : 모든 데이터를 한 번에 입력하는 경사하강법
  - 배치(batch) : 하나의 데이터셋
- 이전 장에서 배운 경사하강법은 하나의 값에 대한 경사도를 구한 다음 값들을 업데이트

$$x_{new} = x_{old} - \alpha \times (2x_{old})$$

- 실제로는 각 데이터의 경사도를 모두 더해 하나의 값으로 가중치를 업데이트

01 경사하강법의 종류

- 점 한 개씩 사용하여 가중치를 업데이트하지 않는 이유
  - 시간이 오래 걸림
  - 시작점에 따라 지역 최적화(local optimum)에 빠짐 : 그래프 전체에서 최솟점을 찾지 못하고 부분최솟점에 최적점이 위치

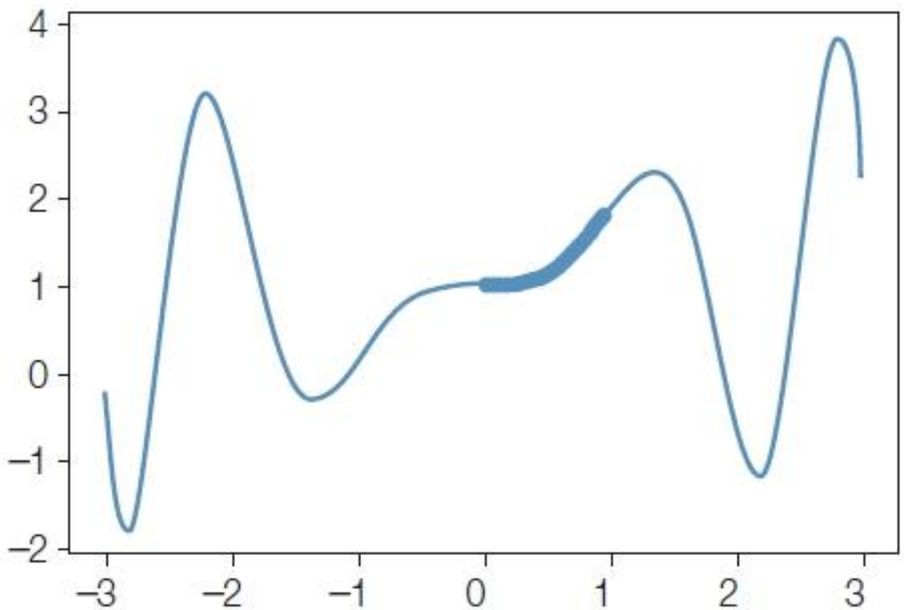


그림 8-1 점을 한 개씩 사용하여 가중치를 업데이트

01 경사하강법의 종류

- 전체-배치 경사하강법의 특징
  - 업데이트 횟수 감소 : 가중치 업데이트 횟수가 줄어 계산상 효율성 상승
  - 안정적인 비용함수 수렴 : 모든 값의 평균을 구하기 때문에 일반적으로 경사하강법이 갖는 지역 최적화 문제를 만날 가능성도 있음
  - 업데이트 속도 증가 : 대규모 데이터셋을 한 번에 처리하면 모델의 매개변수 업데이트 속도에 문제 발생이 적어짐
    - 데이터가 백만 단위 이상을 넘어가면 하나의 머신에서는 처리가 불가능해져서 메모리 문제가 발생

2. 확률적 경사하강법

- 확률적 경사하강법(Stochastic Gradient Decent, SGD) : 학습용 데이터에서 샘플들을 랜덤하게 뽑아서 사용
- 대상 데이터를 섞은(shuffle) 후, 일반적인 경사하강법처럼 데이터를 한 개씩 추출하여 가중치 업데이트

```
Procedure SGD
    shuffle(X)
    for i in number of X do
        $\theta_j = \theta_j - \alpha(\hat{y}^{(i)} - y^{(i)})x_j^{(i)}$
    end for
end procedure
```

그림 8-2 확률적 경사하강법(SGD) 알고리즘의 의사코드

01 경사하강법의 종류

- SGD의 장점
  - 빈번한 업데이트를 하기 때문에 데이터 분석가가 모델의 성능 변화를 빠르게 확인
  - 데이터의 특성에 따라 훨씬 더 빠르게 결과값을 냄
  - 지역 최적화를 회피
- SGD의 단점
  - 대용량 데이터를 사용하는 경우 시간이 매우 오래 걸림
  - 결과의 마지막 값을 확인하기 어려움
    - 흔히 '튀는 현상'이라고 불리는데 비용함수의 값이 줄어들지 않고 계속 변화할 때 정확히 언제 루프(loop)가 종료되는지 알 수 없어 판단이 어렵다

3. 미니-배치 경사하강법

- 미니-배치 경사하강법(mini-batch gradient descent) 또는 미니-배치 SGD(mini-batch SGD) : 데이터의 랜덤한 일부분만 입력해서 경사도 평균을 구해 가중치 업데이트
- 에포크(epoch) : 데이터를 한 번에 모두 학습시키는 횟수
  - 전체-배치 SDG를 한 번 학습하는 루프가 실행될 때 1에포크의 데이터가 학습된다고 말함
- 배치 사이즈(batch-size) : 한 번에 학습되는 데이터의 개수
  - 총 데이터가 5012개 있고 배치 사이즈가 512라면 10번의 루프가 돌면서 1에포크를 학습했다고 말함

01 경사하강법의 종류

- 에포크와 배치 사이즈는 하이퍼 매개변수이므로 데이터 분석가가 직접 선정함

```
Procedure MINI-BATCH SGD
    shuffle(X)
    BS ← BATCH SIZE
    NB ← Number of Batches
    NB ← len(X)//BS
    for i in NB do
```

$$\theta_j = \theta_j - \alpha \sum_{k=i\times BS}^{(i+1)\times BS} (\hat{y}^{(k)} - y^{(k)})x_j^{(k)}$$

그림 8-3 미니-배치 경사하강법 알고리즘의 의사코드

4. SGD의 파이썬 코드 작성하기

- 에포크, 셔플 여부, 배치 사이즈, 인터셉트 추가 여부를 코드에 반영

In [1]:
```
class LinearRegressionGD(object):
    def __init__(self, fit_intercept=True,
                 copy_X=True, eta0=0.001,
                 epochs=1000, batch_size = 1,
                 weight_decay=0.9,
                 shuffle = True):
        self.fit_intercept = fit_intercept
        self.copy_X = copy_X
        self._eta0 = eta0
        self._epochs = epochs

        self._cost_history = []
```

01 경사하강법의 종류

```
        self._coef = None
        self._intercept = None
        self._new_X = None
        self._w_history = None
        self._weight_decay = weight_decay
        self._batch_size = batch_size
        self._is_SGD = shuffle

    def gradient(self, X, y, theta):
        return X.T.dot(self.hypothesis_function(X, theta)-y) / len(X)

    def fit(self, X, y):
        self._new_X = np.array(X) # X 데이터 할당
        y = y.reshape(-1, 1)

        if self.fit_intercept: # intercept 추가 여부
            # 1로만 구성된 상수항을 모든 데이터에 추가
            intercept_vector = np.ones([len(self._new_X), 1])
            self._new_X = np.concatenate(
                    (intercept_vector, self._new_X), axis=1)
```

01 경사하강법의 종류


```
        theta_init = np.random.normal(0, 1, self._new_X.shape[1])
        # weight값 초기화

        self._w_history = [theta_init]
        self._cost_history = [self.cost(
           self.hypothesis_function(self._new_X, theta_init), y)]

        theta = theta_init

        for epoch in range(self._epochs): #지정된 epoch의 값만큼 학습 실행
            X_copy = np.copy(self._new_X)

            if self._is_SGD: # stochastic 적용 여부
                np.random.shuffle(X_copy)

            batch = len(X_copy) // self._batch_size
            # 배치 사이즈를 기준으로 전체데이터를 나눔
```

01 경사하강법의 종류

```
            for batch_count in range(batch):
                X_batch = np.copy( # 배치 사이즈를 기준으로 데이터를 slice
                         X_copy[batch_count * self._batch_size :
(batch_count+1) & self._batch_size])

                gradient = self.gradient(X_batch, y,
theta).flatten( )
                theta = theta - self._eta0 * gradient

            if epoch % 100 == 0:
                self._w_history.append(theta)
                cost = self.cost(
                    self.hypothesis_function(self._new_X, theta), y)
                self._cost_history.append(cost)
            self._eta0 = self._eta0 * self._weight_decay

        if self.fit_intercept:
            self._intercept = theta[0]
            self._coef = theta[1:]
        else:
            self._coef = theta
```

01 경사하강법의 종류


```
    def cost(self, h, y):
        return 1/(2*len(y)) * np.sum((h-y).flatten() ** 2)

    def hypothesis_function(self, X, theta):
        return X.dot(theta).reshape(-1, 1)

    def gradient(self, X, y, theta):
        return X.T.dot(self.hypothesis_function(X, theta)-y) / len(X)

    def fit(self, X, y):
        self._new_X = np.array(X)

    y = y.reshape(-1, 1)

    if self.fit_intercept:
        intercept_vector = np.ones([len(self._new_X), 1])
        self._new_X = np.concatenate(
                (intercept_vector, self._new_X), axis=1)
```

01 경사하강법의 종류


```
theta_init = np.random.normal(0, 1, self._new_X.shape[1])
self._w_history = [theta_init]
self._cost_history = [self.cost(
           self.hypothesis_function(self._new_X, theta_init), y)]

theta = theta_init

for epoch in range(self._epochs):
    gradient = self.gradient(self._new_X, y, theta).flatten( )
    theta = theta - self._eta0 * gradient

    if epoch % 100 == 0:
        self._w_history.append(theta)
        cost = self.cost(
            self.hypothesis_function(self._new_X, theta), y)
        self._cost_history.append(cost)
    self._eta0 = self._eta0 * self._weight_decay
```

01 경사하강법의 종류

```
    if self.fit_intercept:
        self._intercept = theta[0]
        self._coef = theta[1:]
    else:
        self._coef = theta

    def predict(self, X):
        test_X = np.array(X)

    if self.fit_intercept:
        intercept_vector = np.ones([len(test_X), 1])
        test_X = np.concatenate(
                (intercept_vector, test_X), axis=1)
        weights = np.concatenate(([self._intercept], self._coef),
axis=0)
    else:
        weights = self._coef
        return test_X.dot(weights)
```

01 경사하강법의 종류


```
    @property
    def coef(self):
        return self._coef

    @property
    def intercept(self):
        return self._intercept

    @property
    def weights_history(self):
        return np.array(self._w_history)

    @property
    def cost_history(self):
        return self._cost_history
```

01 경사하강법의 종류

- 생성된 경사하강법 모델을 사용하여 학습 수행

In [2]:
```
import pandas as pd
import numpy as np

df = pd.read_csv("c:/source/ch08/train.csv")

X = df["x"].values.reshape(-1,1)
y = df["y"].values

gd_lr = LinearRegressionGD(eta0=0.001, epochs=10000, 
batch_size=1, shuffle=False)
bgd_lr = LinearRegressionGD(eta0=0.001, epochs=10000, 
batch_size=len(X), shuffle=False)
sgd_lr = LinearRegressionGD(eta0=0.001, epochs=10000, 
batch_size=1, shuffle=True)
msgd_lr = LinearRegressionGD(eta0=0.001, epochs=10000, 
batch_size=100, shuffle=True)
```

01 경사하강법의 종류

- 각 학습 결과 cost 값의 변화
  - 학습 알고리즘에 따라 cost 값이 변함

In [3]:
```
gd_lr.fit(X, y)
bgd_lr.fit(X, y)
sgd_lr.fit(X,y)
msgd_lr.fit(X,y)
```

  - 50에포크까지 SGD_lr과 msgd_lr의 cost 값이 매우 진폭이 큼
    - 데이터 일부를 셔플해서 넣기 때문에 cost 값이 계속 변화하며 수렴
  - 복잡한 알고리즘일수록 SGD가 효과적

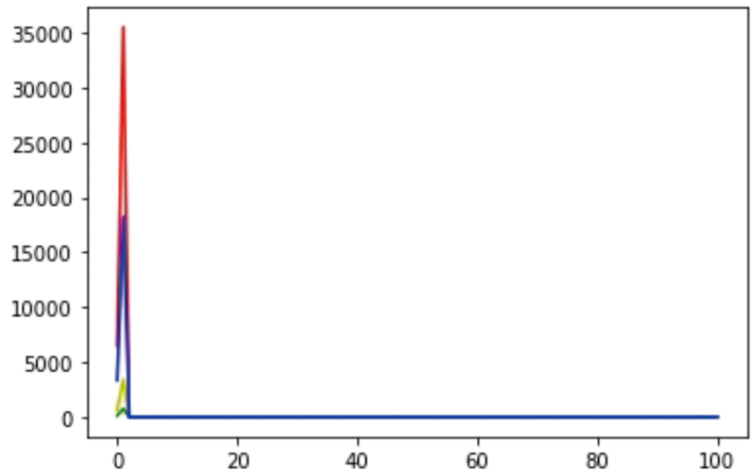
01 경사하강법의 종류

In [4]:

```
import matplotlib.pyplot as plt

plt.plot(range(len(gd_lr.cost_history)),
gd_lr.cost_history, c="r")
plt.plot(range(len(bgd_lr.cost_history)),
bgd_lr.cost_history, c="y")
plt.plot(range(len(sgd_lr.cost_history)),
sgd_lr.cost_history, c="g")
plt.plot(range(len(msgd_lr.cost_history)),
msgd_lr.cost_history, c="b")
```

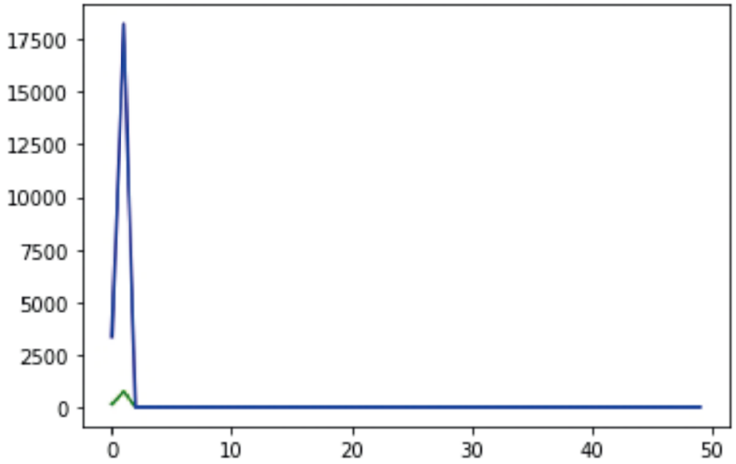
Out [4]:



01 경사하강법의 종류

In [5]:
```
plt.plot(range(50), sgd_lr.cost_history[:50], c="g")
plt.plot(range(50), msgd_lr.cost_history[:50], c="b")
```

Out [5]:



[TIP] Out [4]와 Out [5] 결과값이 랜덤하게 출력된다.

02
과대적합과 정규화

1. 과대적합 극복하기

- 편향(bias) : 학습된 모델이 학습 데이터에 대해 만들어 낸 예측값과 실제값과의 차이
  - 모델의 결과가 얼마나 한쪽으로 쏠려 있는지 나타냄
  - 편향이 크면 학습이 잘 진행되기는 했지만 해당 데이터에만 잘 맞음
- 분산(variance) : 학습된 모델이 테스팅 데이터에 대해 만들어 낸 예측값과 실제값과의 차이
  - 모델의 결과가 얼마나 퍼져 있는지 나타냄
- 편향-분산 트레이드오프(bias-variance trade-off) : 편향과 분산의 상충관계

02 과대적합과 정규화

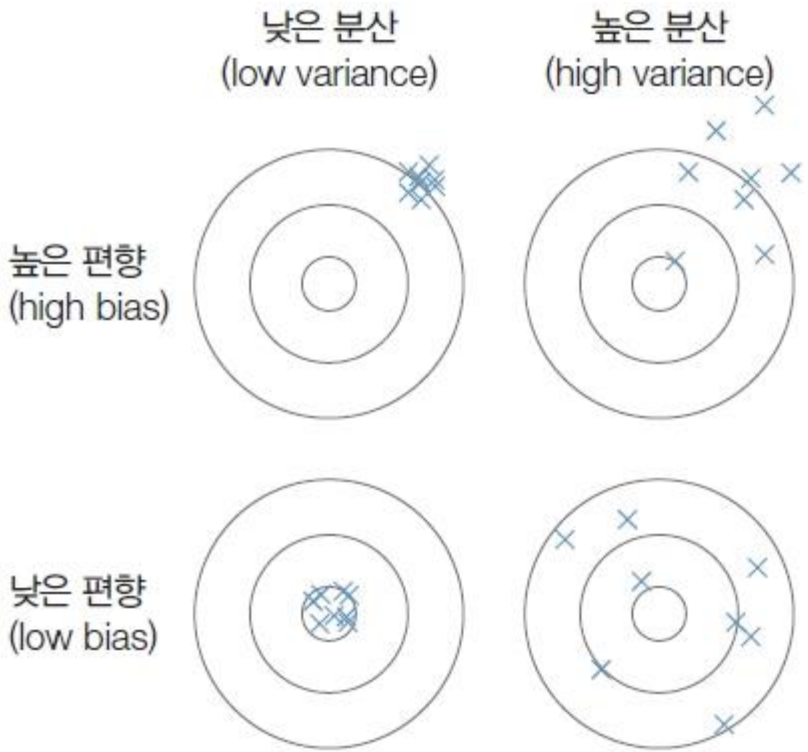


그림 8-4 편향-분산 트레이드오프

[TIP] 과대적합(overfitting) : 높은 분산 낮은 편향 상태로 함수가 훈련 데이터셋에만 맞음. 피쳐의 개수를 줄이거나 정규화하여 해결

[TIP] 과소적합(underfitting) : 낮은 분산 높은 편향 상태로 함수가 훈련 데이터셋과 테스트 데이터셋에 모두 맞지 않음. 피쳐를 추가하여 해결

02 과대적합과 정규화

- 과대적합이 발생할 때 경사하강법 루프가 진행될수록 학습 데이터셋에 대한 비용함수의 값은 줄어들지만 테스트 데이터셋의 비용함수 값은 증가

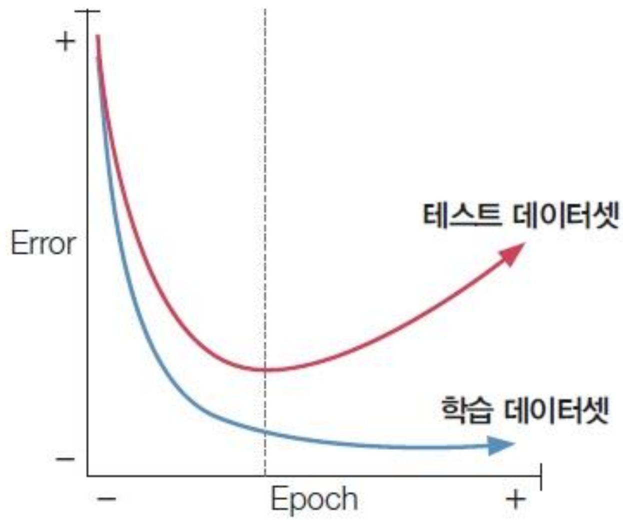


그림 8-5 과대적합이 발생할 때의 경사하강법

- 선형회귀 외에도 결정트리(decision tree)나 딥러닝처럼 연산에 루프가 필요한 모든 알고리즘에서 똑같이 발생

02 과대적합과 정규화

- 오컴의 면도날 원리 : '보다 적은 수의 논리로 설명이 가능한 경우, 많은 수의 논리를 세우지 않는다'
  - '경제성의 원리' 또는 '단순성의 원리'
  - 머신러닝에서는 너무 많은 피쳐를 사용하지 않는 것
- 선형회귀에서 과대적합 해결책
  - 더 많은 데이터 활용하기 : 오류가 없고, 분포가 다양한 데이터를 많이 확보
  - 피쳐의 개수 줄이기 : 필요한 피쳐만 잘 찾아 사용
  - 적절한 매개변수 선정하기 : SGD의 학습률이나 루프의 횟수처럼 적절한 하이퍼 매개변수를 선정
  - 정규화 적용하기 : 데이터 편향성에 따라 필요 이상으로 증가한 피쳐의 가중치 값을 적절히 줄이는 규제 수식을 추가

2. L2 정규화 : 리지 회귀

- 리지 회귀(ridge regression) : L2 정규화(L2 regularization)라고 부름
- 놈(norm) : 좌표평면의 원점에서 점까지의 거리를 나타내어 벡터의 크기를 측정하는 기법

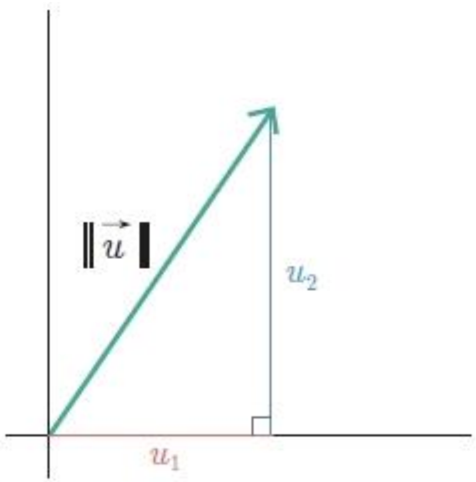


그림 8-6 놈(norm)의 이해

$$L_{2\,\mathrm{norm}} = \| x \| 2 = \sqrt{\left(\sum_i^n |x|^2\right)} = \left(\sum_i^n |x_i|^2\right)^{\frac{1}{2}}$$

$$\text{if } x = (x_1, x_2, \cdots x_n)$$

02 과대적합과 정규화

- X는 하나의 벡터
- L2 놈(L2 norm) : 벡터 각 원소들의 제곱합에 제곱근을 취함

- 리지 회귀는 L2 놈을 선형회귀의 비용함수 수식에 적용

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n}\theta_j^2$$

- 뒷부분에 새로 붙인 수식은 페널티텀(penalty term)으로, 모델의 가중치 값들의 제곱의 합
  - 가중치 값이 조금이라도 커질 때 비용함수에 매우 큰 영향
  - λ가 클수록 전체 페널티텀의 값이 커져 θ 값이 조절됨
  - λ는 사람이 직접 값을 입력하는 하이퍼 매개변수

02 과대적합과 정규화

- 리지 회귀 수식을 미분하면 j의 값이 1 이상일 때 페널티가 적용됨

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)}$$

$$\theta_j := \theta_j - \alpha \left[\left(\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \right] j \in \{1, 2, \cdots, n\}$$

$$\theta_j := \theta_j \left(1 - \alpha \frac{\lambda}{m} \right) - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

02 과대적합과 정규화

2. L1 정규화 : 라쏘 회귀

- 라쏘 회귀(lasso regression) L1 정규화(L1 regularization) 라고 부름
- 가중치에 페널티텀을 추가하는데, 기존 수식에다 L1 놈 페널티를 추가하여 계산

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2}\sum_{j=1}^{n}|\theta_j|$$

- L1 놈(L1 norm) : 절대값을 사용하여 거리를 측정

$$\|x\|_1 := \sum_{i=1}^{n}|x_i|$$

02 과대적합과 정규화

- L1 정규화와 L2 정규화가 실제 적용되는 과정

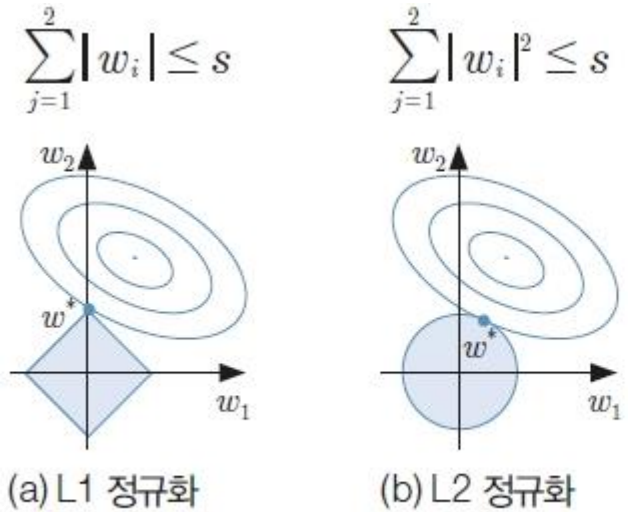


그림 8-7 L1 정규화와 L2 정규화의 실제 적용 과정

- 타원은 두 개의 가중치 값의 최적점과 그 가중치 값으로 생기는 비용함수의 공통 범위
- 아래 마름모나 원은 w가 가질 수 있는 범위이고 타원과 만나는 점이 바로 사용가능한 가중치 값

02 과대적합과 정규화

- L1정규화는 직선과 타원 만나는 점이 양쪽 끝에 생성됨
  - 극단적인 값이 생성되어 다른 가중치 값이 선택되지 않는 현상이 발생할 수 있음
  - 사용해야 하는 피쳐와 사용하지 않아도 되는 피쳐를 선택하여 사용하도록 지원
- L2 정규화는 원과 타원이 만나는 점이 많아져서 비교적 쉽게 연산되어 계산 효율(computational efficiency) 확보
  - 한 점에서 만나기 때문에 하나의 해답만 제공

03
사이킷런을 이용한 선형회귀

1. 사이킷런과 선형회귀 관련 함수

- 사이킷런(scikit-learn) : 대표적인 머신러닝 라이브러리

표 8-1 사이킷런의 선형회귀 관련 함수

함수명	설명	알고리즘
LinearRegression	가장 기본적인 선형회귀 알고리즘을 사용하며, SGD가 아닌 최소자승법으로 계산한다.	최소자승법
Lasso	L1 손실을 활용한 라쏘 알고리즘을 사용한다.	최소자승법
Ridge	L2 손실을 활용한 리지 알고리즘을 사용한다.	최소자승법
SGDRegressor	확률적 경사 하강법을 사용한 회귀 모델을 만든다. SGD에서 비용함수만을 변경하여 모든 함수를 지원하고 있어 필요한 하이퍼 매개변수를 설정해야 한다.	SGD

- 최소자승법과 SGD 기반 알고리즘 클래스 제공

2. 사이킷런을 활용하여 선형회귀 구현하기

- ‘boston housing prices(보스턴 집값)’ 데이터셋

구분	변수	설명
x 변수 13개	[01] CRIM	자치시(town)별 1인당 범죄율
	[02] ZN	25,000 평방피트를 초과하는 거주지역의 비율
	[03] INDUS	비소매상업지역이 점유하고 있는 토지의 비율
	[04] CHAS	찰스강에 대한 더미변수(강의 경계에 위치한 경우는 1, 아니면 0)
	[05] NOX	10ppm 당 농축 일산화질소
	[06] RM	주택 1가구당 평균 방의 개수
	[07] AGE	1940년 이전에 건축된 소유 주택의 비율
	[08] DIS	5개의 보스턴 직업센터까지의 접근성 지수
	[09] RAD	방사형 도로까지의 접근성 지수
	[10] TAX	10,000달러 당 재산세율
	[11] PTRATIO	자치시(town)별 학생/교사 비율
	[12] B	1000(Bk−0.63)^2, 여기서 Bk는 자치시별 흑인의 비율을 말함
	[13] LSTAT	모집단의 하위 계층의 비율(%)
y 변수	[14] MEDV	본인 소유의 주택 가격(중앙값) (단위 : $1,000)

그림 8-8 boston housing prices(보스턴 집값) 데이터셋

03 사이킷런을 이용한 선형회귀

2.1 데이터 확보하기

- sklearn.datasets 라이브러리 load_boston 모듈을 사용하여 데이터를 추출
  - 딕셔너리 타입의 객체를 반환

In [1]:
```
from sklearn.datasets import load_boston
import matplotlib.pyplot as plt
import numpy as np

boston = load_boston()
boston.keys()
```

Out [1]:
```
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'filename'])
```

03 사이킷런을 이용한 선형회귀

- data 키 값 추출

In [2]:
```
boston["data"]
```

Out [2]:
```
array([[6.3200e-03, 1.8000e+01, 2.3100e+00, ...,
        1.5300e+01, 3.9690e+02, 4.9800e+00],
       [2.7310e-02, 0.0000e+00, 7.0700e+00, ...,
        1.7800e+01, 3.9690e+02, 9.1400e+00],
       [2.7290e-02, 0.0000e+00, 7.0700e+00, ...,
        1.7800e+01, 3.9283e+02, 4.0300e+00],
       ...,
       [6.0760e-02, 0.0000e+00, 1.1930e+01, ...,
        2.1000e+01, 3.9690e+02, 5.6400e+00],
       [1.0959e-01, 0.0000e+00, 1.1930e+01, ...,
        2.1000e+01, 3.9345e+02, 6.4800e+00],
       [4.7410e-02, 0.0000e+00, 1.1930e+01, ...,
        2.1000e+01, 3.9690e+02, 7.8800e+00]])
```

- x와 y 각 데이터셋을 추출
  - y_data는 n×1의 형태로 변환

```
In [3]:  x_data = boston.data
         y_data = boston.target.reshape(boston.target.size,1)
         y_data.shape

Out [3]: (506, 1)
```

03 사이킷런을 이용한 선형회귀

2.2 데이터 전처리하기

- 피쳐 스케일링 적용

In [4]:
```
from sklearn import preprocessing

minmax_scale = preprocessing.MinMaxScaler(feature_
range=(0,5)).fit(x_data) # (1)
x_scaled_data = minmax_scale.transform(x_data) # (2)

x_scaled_data[:3]
```

Out [4]:
```
array([[0.00000000e+00, 9.00000000e-01, 3.39076246e-01,
        0.00000000e+00, 1.57407407e+00, 2.88752635e+00,
        3.20803296e+00, 1.34601570e+00, 0.00000000e+00,
        1.04007634e+00, 1.43617021e+00, 5.00000000e+00,
        4.48399558e-01],
```

		[1.17961270e-03, 0.00000000e+00, 1.21151026e+00, 0.00000000e+00, 8.64197531e-01, 2.73998850e+00, 3.91349125e+00, 1.74480990e+00, 2.17391304e-01, 5.24809160e-01, 2.76595745e+00, 5.00000000e+00, 1.02235099e+00], [1.17848872e-03, 0.00000000e+00, 1.21151026e+00, 0.00000000e+00, 8.64197531e-01, 3.47192949e+00, 2.99691040e+00, 1.74480990e+00, 2.17391304e-01, 5.24809160e-01, 2.76595745e+00, 4.94868627e+00, 3.17328918e-01]])

03 사이킷런을 이용한 선형회귀

2.3 데이터 분류하기

- 데이터를 훈련과 테스트 형태로 분류

In [5]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test =
train_test_split(x_scaled_data, y_data, test_size=0.33)
# X 데이터의 학습 데이터셋, X 데이터의 테스트 데이터셋
# Y 데이터의 학습 데이터셋, Y 데이터의 테스트 데이터셋

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

Out [5]:
```
((339, 13), (167, 13), (339, 1), (167, 1))
```

2.4 데이터 학습하기

- 학습에 사용할 알고리즘 해당하는 모델의 클래스 호출
  - 각 클래스의 매개변수를 이해해야 함
- 공통적으로 사용하는 매개변수
  - fit_intercept : 절편을 사용할지 말지를 선택
  - normalize : 학습할 때 값들을 정규화할지 말지
  - copy_X : 학습 시 데이터를 복사한 후 학습을 할지 결정
  - n_jobs : 연산을 위해 몇 개의 CPU를 사용할지 결정
  - alpha : 라쏘 회귀, 리지 회귀, SGD에 있음. 페널티 값을 지정

03 사이킷런을 이용한 선형회귀

- SGD의 매개변수
  - 직접 penalty 함수를 지정할 수 있는데, λ 값을 alpha에 입력
  - max_iter : 최대 반복 횟수를 지정
  - tol : 더 이상 비용이 줄어들지 않을 때 반복이 멈추는 최솟값
  - eta0 : 한 번에 실행되는 학습률

03 사이킷런을 이용한 선형회귀

In [6]:
```
from sklearn import linear_model
regr = linear_model.LinearRegression(
         fit_intercept=True,normalize=False,
         copy_X=True, n_jobs=8)
lasso_regr = linear_model.Lasso(
         alpha=0.01, fit_intercept=True,
         normalize=False,copy_X=True)
ridge_regr = linear_model.Ridge(
         alpha=0.01, fit_intercept=True,
         normalize=False,copy_X=True)
SGD__regr = Linear_model.SGDRegressor(penalty="l2",
                            alpha=0.01, max_iter=1000,
                            tol=0.001, eta0=0.01)
```

03 사이킷런을 이용한 선형회귀

- 사이킷런은 '적합-예측(fit-predict)' 또는 '적합-변형(fit-transform)'의 구조
  - 모델을 생성한 후 예측을 하거나 전처리 모델의 규칙을 세운 후 데이터 전처리를 적용하는 구조

```
In [7]:  regr.fit(X_train, y_train)
Out [7]: LinearRegression(n_jobs=8)
In [8]:  print('Coefficients: ', regr.coef_)
         print('intercept: ', regr.intercept_)
Out [8]: Coefficients: [[-2.86129759 0.27632862 0.10322333
           0.33532791 -1.89104482 3.55479622
           -0.04952964 -3.01015804 1.22330686 -1.00916771
           -1.98466467 0.62386235 -3.9996908 ]]
         intercept: [29.42877381]
```

2.5 예측하기와 결과 분석하기

- 만들어진 함수로 실제 예측을 한다

In [9]:	regr.predict(x_data[:5])
Out [9]:	array([[-58.72562452], [-32.88598964], [-11.38914021], [10.01207448], [1.87764423]])

- regr 대신 수식을 그대로 재현해도 같은 결과가 출력됨

In [10]:	x_data[:5].dot(regr.coef_.T) + regr.intercept_
Out [10]:	array([[-58.72562452], [-32.88598964], [-11.38914021], [10.01207448], [1.87764423]])

03 사이킷런을 이용한 선형회귀

- 사이킷런에서 지표들(metrics)을 호출하여 성능을 비교

In [11]:
```
from sklearn.metrics import r2_score
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error

y_true = y_test.copy()
y_hat = regr.predict(X_test)

r2_score(y_true, y_hat), mean_absolute_error(y_true,
y_hat), mean_squared_error(y_true, y_hat)
```

Out [11]:
```
(0.7012192205071575, 3.6874625281998266,
28.869826251555843)
```

03 사이킷런을 이용한 선형회귀

- 필요에 따라 시각화 도구로 예측값과 실제값 비교

In [12]:

```
plt.scatter(y_true, y_hat, s=10)
plt.xlabel("Prices: $Y_i$")
plt.ylabel("Predicted prices: $\hat{Y}_i$")
plt.title("Prices vs Predicted prices: $Y_i$ vs $\hat{Y}_i$")
```

Out [12]: