스터디 1주차(chap2)

2. 간단한 분류 알고리즘 훈련

2.1 인공 뉴런: 초기 머신 러닝의 간단한 역사

▼ 퍼셉트론

▼ 2.1.1 인공 뉴런의 수학적 정의

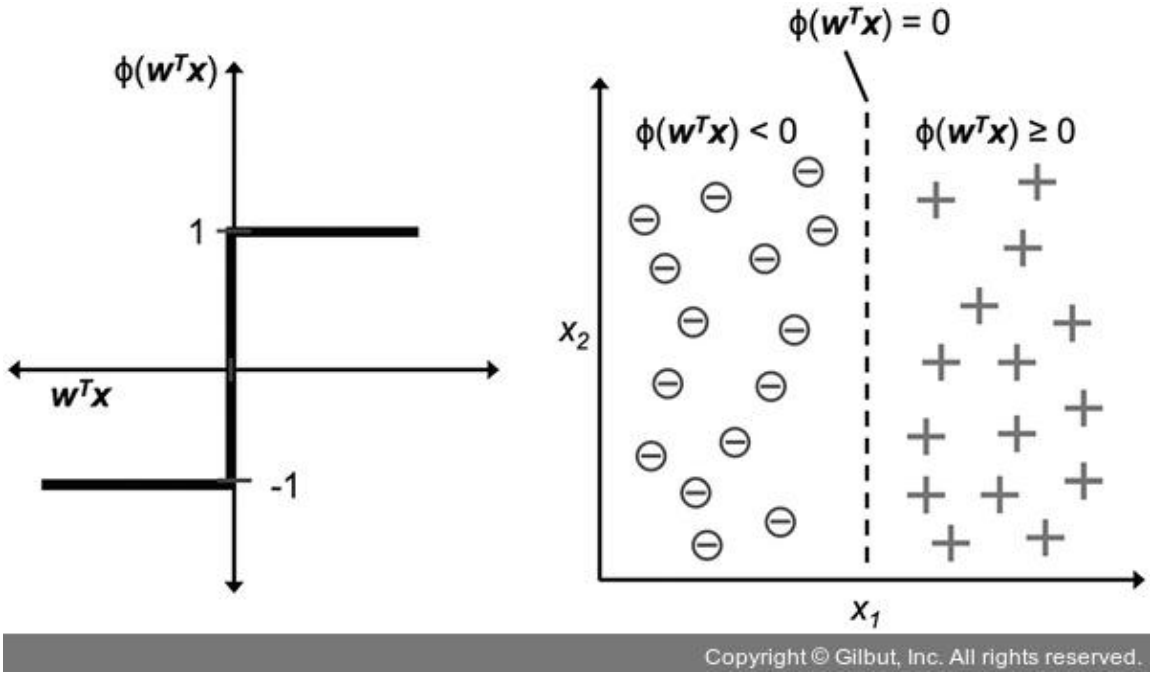
인공 뉴런(artifical neuron) 아이디어를 두 개의 클래스가 있는 이진 분류 작업으로 볼 수 있다. 두 클래스는 1(양성 클래스)과 -1(음성 클래스)로 나뉜다. 그 다음 입력 값 x와 이에 상응하는 가중치 벡터 w의 선형 조합으로 결정 함수(ϕ(z))를 정의한다. 최종입력(net input)인 z는 z=w1x1+ w2x2+...+wmxm이다. 특정 샘플 x(i)의 최종 입력이 사전에 정의된 임계 값 θ보다 크면 클래스 1로, 그렇지 않으면 클래스 -1로 예측한다.

임계 값 θ를 식의 왼쪽으로 옮겨 w0 - -θ고 xo=1인 0번째 가중치를 정의한다.
z = w0x0 + w1x1 + ... + wmxm = wTx

결정 함수는 다음과 같다.

$$y = \begin{cases} 0\,, x \leq 0 \\ -1\,, x > 0 \end{cases}$$

머신러닝 분야에서는 음수 임계값 또는 가중치 w0 - -θ를 절편이라고 한다.



왼) 퍼셉트론 결정 함수로 최종입력 z=wTx가 이진 출력(-1 or 1)으로 압축 되는 방법
오) 선형 분리가 가능한 두 개의 클래스 사이를 구별하는 방법

▼ 2.1.2 퍼셉트론 학습 규칙

1. 가중치를 0 또는 랜덤한 작은 값으로 초기화 한다.
2. 각 훈련 샘플 x(i)에서 다음 작업을 한다.
   a. 출력값 y를 계산한다.
   b. 가중치를 업데이트 한다.

**수학 어쩌고 정리할것

2.2 파이썬으로 퍼셉트론 학습 알고리즘 구현

▼ 2.2.1 객체 지향 퍼셉트론 API

Perceptron 객체를 초기화한 후 fit 메서드로 데이터에서 학습하고, 별도의 predict 매서드로 예측을 만든다.

```python
class Perceptron(object):
    """퍼셉트론 분류기

    매개변수
    ------------
    eta : float
      학습률 (0.0과 1.0 사이)
    n_iter : int
      훈련 데이터셋 반복 횟수
    random_state : int
      가중치 무작위 초기화를 위한 난수 생성기 시드

    속성
    -----------
    w_ : 1d-array
      학습된 가중치
    errors_ : list
      에포크마다 누적된 분류 오류

    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """훈련 데이터 학습

        매개변수
        ----------
        X : {array-like}, shape = [n_samples, n_features]
          n_samples개의 샘플과 n_features개의 특성으로 이루어진 훈련 데이터
        y : array-like, shape = [n_samples]
          타깃값

        반환값
        -------
        self : object

        """
```

```
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_[1:] += update * xi
                self.w_[0] += update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self

    def net_input(self, X):
        """최종 입력 계산"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        """단위 계단 함수를 사용하여 클래스 레이블을 반환합니다"""
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```

퍼셉트론 구현을 사용하여 학습률 eta와 에포크 횟수(훈련 데이터셋을 반복하는 횟수) n_iter로 새로운 Perceptron 객체를 초기화한다.

▶️fit 메서드

-self.w_ 가중치를 백터 R로 초기화한다.

-m: 데이터셋에 있는 차원(특성) 개수

-self.w_[0]: 절편

-이 벡터(self.w_ / 벡터 R)는 표준 편차가 0.01인 정규 분포에서 뽑은 랜덤한 작은 수를 담고 있음

-가중치를 0으로 초기화 하지 않는 이유: 학습률 파라미터 eta에 영향을 미침 / 가중치가 0이라면 eta는 가중치 벡터의 방향이 아니라 크기에만 영향을 미침

-가중치 초기화 후 모든 개개의 샘플을 반복순회, 가중치 업데이트

-self.errors 리스트에 잘못 분류된 횟수 기록: 퍼셉트론을 잘 수행했는지 분석 가능

▶️predict 메서드

-클래스 레이블에 대한 예측

-모델이 학습되고 난 후 새로운 데이터의 클래스 레이블을 예측할 때도 사용

▶️net_input 메서드: np.dot 함수는 벡터 점곱 wTx를 계산

▼ 2.2.2 붓꽃 데이터셋에서 퍼셉트론 훈련

50개의 setosa와 50개의 versicolor 꽃에 해당하는 처음 100개의 클래스 레이블 추출
꽃받침 길이와 꽃잎 길이 사용

In [24]:
```
df = pd.read_csv('https://archive.ics.uci.edu/ml/'
        'machine-learning-databases/iris/iris.data', header=None)
df.tail()
```

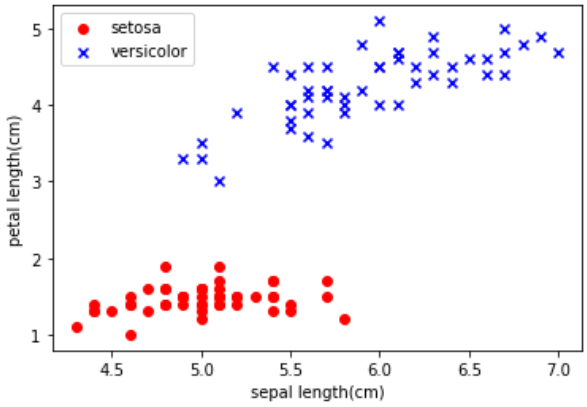
Out [24]:

	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

In [27]:
```
#Iris-setos(50개)와 Iris-versicolor(50개)에 해당하는 100개의 클래스 레이블 추출
y = df.iloc[0:100, 4].values
y = np.where(y=='Iris-setosa',-1,1)
```

In [28]:
```
#꽃받침 길이와 꽃잎 길이를 추출
X = df.iloc[0:100, [0,2]].values
```

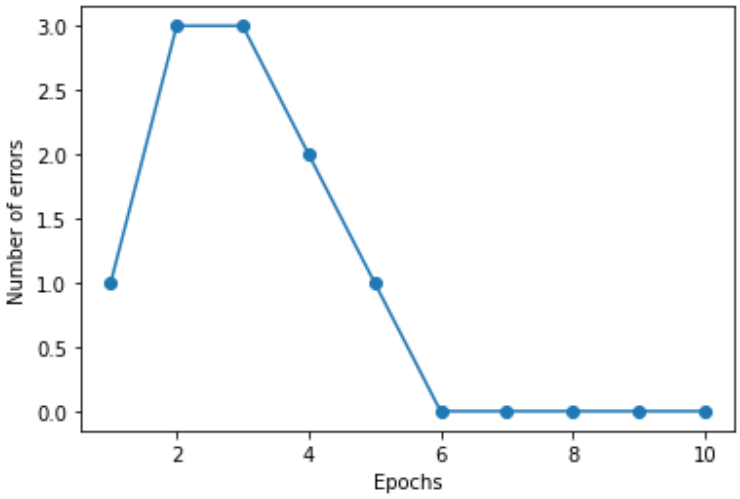
In [29]:
```
#산점도 그리기
plt.scatter(X[:50,0],X[:50,1],color='red',marker='o',label='setosa')
plt.scatter(X[50:100,0],X[50:100,1],color='blue',marker='x',label='versicolor')
plt.xlabel('sepal length(cm)')
plt.ylabel('petal length(cm)')
plt.legend(loc='upper left')
plt.show()
```



-꽃받침 길이와 꽃잎 길이 두 개의 특성 축을 따라 분포된 형태

In [30]:

```
ppn = Perceptron(eta=0.1, n_iter=10)

ppn.fit(X, y)

plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Number of errors')

plt.show()
```



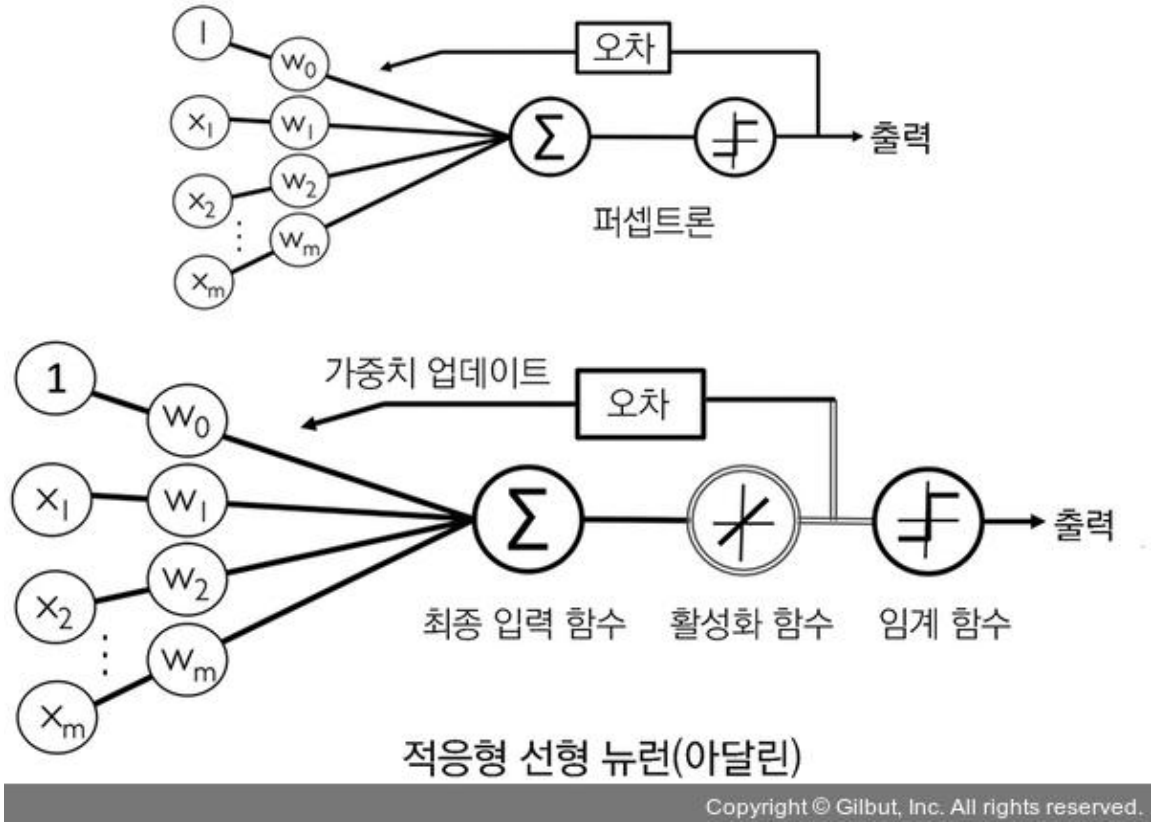
-에포크 대비 잘못 분류된 오차를 그래프로 그리기
-6번째 에포크 이후 수렴했음, 그 후 완벽하게 분류

**아이패드 필기 후 첨부 그리고 이해 못함

2.3 적응형 선형 뉴런과 학습의 수렴

▼ 아달린

-적응형 선형 뉴런

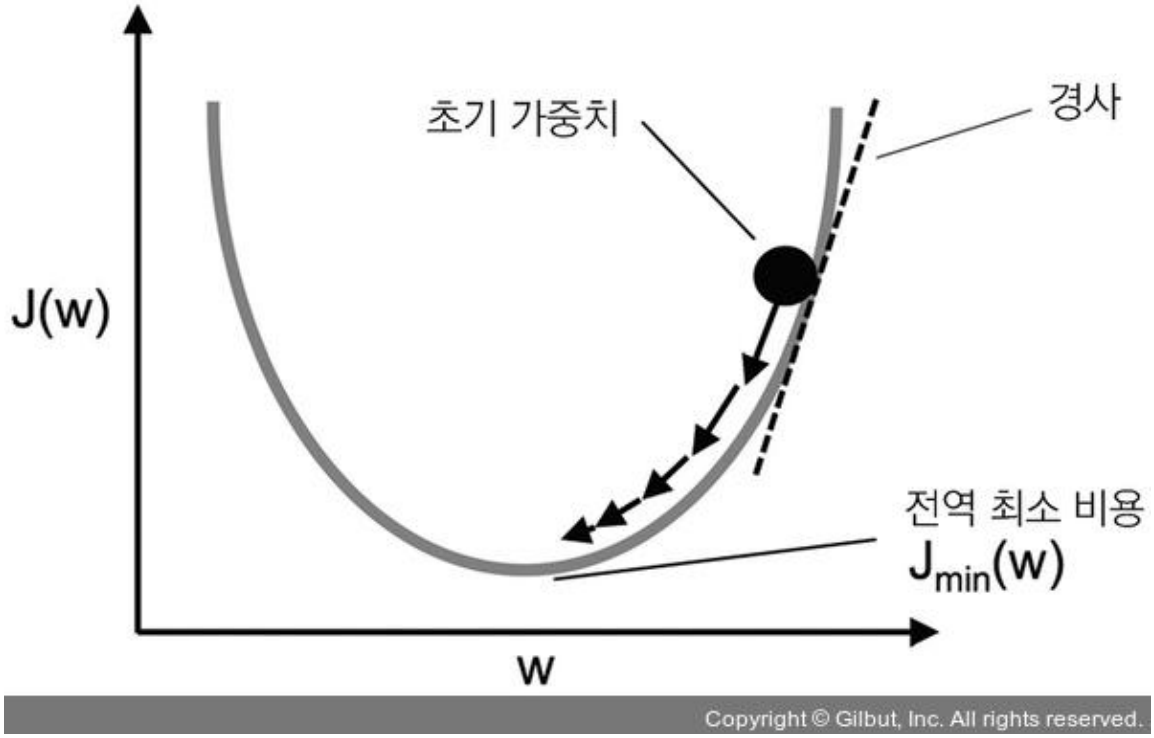


아달린은 연속 함수로 비용 함수를 정의, 최소화하는 핵심 개념
아달린과 퍼셉트론의 가장 큰 차이점: 가중치를 업데이트 할 때, 단위함수가 아닌 선형 활성화 함수 사용
-아달린: 진짜 클래스 레이블과 선형 활성화 함수의 실수 출력 값을 비교하여 모델의 오차 계산, 가증치 증가
-퍼셉트론: 진짜 클래스 레이블과 예측 클래스 레이블을 비교

▼ 2.3.1 경사 하강법으로 비용 함수 최소화

학습 과정 동안 최적화하기 위해 정의한 목적 함수이다. (최소화하려는 비용 함수가 목적 함수가될 수도 있다.) 아달린은 계산된 출력과 진짜 클래스 레이블 사이의 제곱 오차합으로 가중치를 학습하기 위한 비용 함수 J를 정의한다.

$$J(w) = \frac{1}{2}\sum_i (y^{(i)} - \emptyset(z^{(i)}))^2$$



경사 하강법 알고리즘

: 초기 가중치를 최솟값에 도달할 때까지 이동시킴
: 각 반복에서 경사의 반대 방향으로 진행됨
:경사의 기울기와 학습률로 결정됨

▼ 2.3.2 파이썬으로 아달린 구현

```python
class AdalineGD(object):
    """적응형 선형 뉴런 분류기

    매개변수
    ------------
    eta : float
      학습률 (0.0과 1.0 사이)
    n_iter : int
      훈련 데이터셋 반복 횟수
    random_state : int
      가중치 무작위 초기화를 위한 난수 생성기 시드

    속성
    -----------
    w_ : 1d-array
      학습된 가중치
    cost_ : list
      에포크마다 누적된 비용 함수의 제곱합

    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """훈련 데이터 학습

        매개변수
        ----------
        X : {array-like}, shape = [n_samples, n_features]
          n_samples 개의 샘플과 n_features 개의 특성으로 이루어진 훈련 데이터
        y : array-like, shape = [n_samples]
          타깃값
        반환값
        -------
        self : object

        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        self.cost_ = []

```

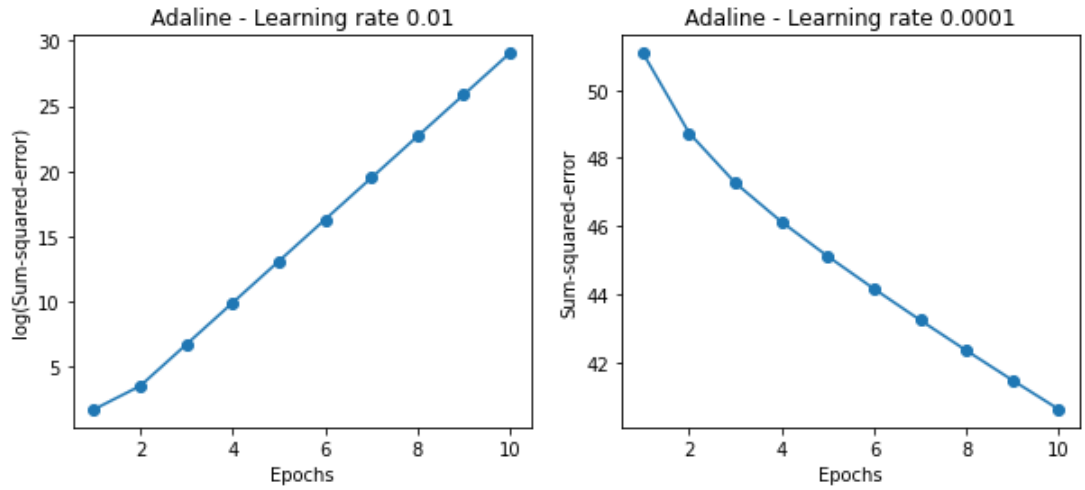
```
        for i in range(self.n_iter):
            net_input = self.net_input(X)
            # Please note that the "activation" method has no effect
            # in the code since it is simply an identity function. We
            # could write `output = self.net_input(X)` directly instead.
            # The purpose of the activation is more conceptual, i.e.,
            # in the case of logistic regression (as we will see later),
            # we could change it to
            # a sigmoid function to implement a logistic regression classifier.
            output = self.activation(net_input)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        """최종 입력 계산"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        """선형 활성화 계산"""
        return X

    def predict(self, X):
        """단위 계단 함수를 사용하여 클래스 레이블을 반환합니다"""
        return np.where(self.activation(self.net_input(X))>=0.0,1,-1)
```

:퍼셉트론처럼 개별 훈련 샘플마다 평가 후 가중치를 업데이트 하지 않고, 전체 훈련 데이터셋을 기반으로 그레이디언트를 계산한다.

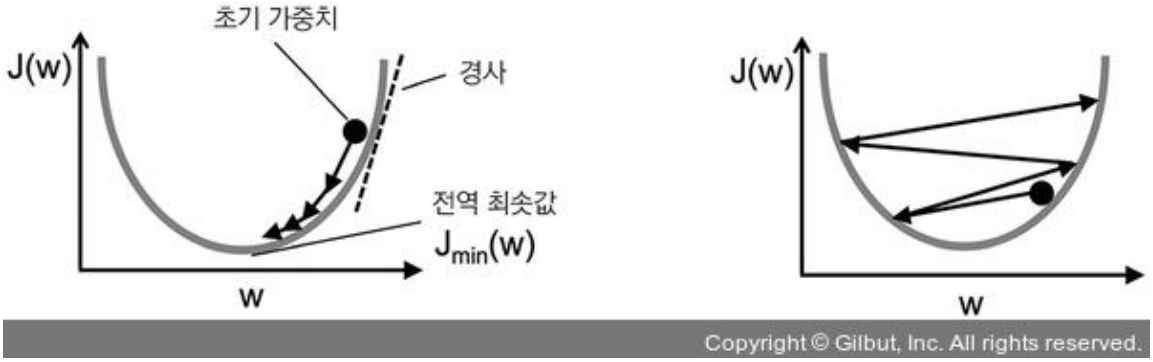
:self.w_[0]: 0번째 가중치(절편)

:self.w_[1:] 가중치 1~m까지

```python
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))

ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)
ax[0].plot(range(1, len(ada1.cost_) + 1), np.log10(ada1.cost_), marker='o')
ax[0].set_xlabel('Epochs')
ax[0].set_ylabel('log(Sum-squared-error)')
ax[0].set_title('Adaline - Learning rate 0.01')

ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
ax[1].plot(range(1, len(ada2.cost_) + 1), ada2.cost_, marker='o')
ax[1].set_xlabel('Epochs')
ax[1].set_ylabel('Sum-squared-error')
ax[1].set_title('Adaline - Learning rate 0.0001')

plt.show()
```



n=0.1일 때:
비용 함수의 최소화를 찾지 못하고 오차는 에포코마다 점점 커짐. 전역 최솟값을 지나쳤기 때문.
n=0.01일 때:
학습률이 너무 작기 때문에 전역 최솟값에 수렴하려면 아주 많은 에포코 필요함.



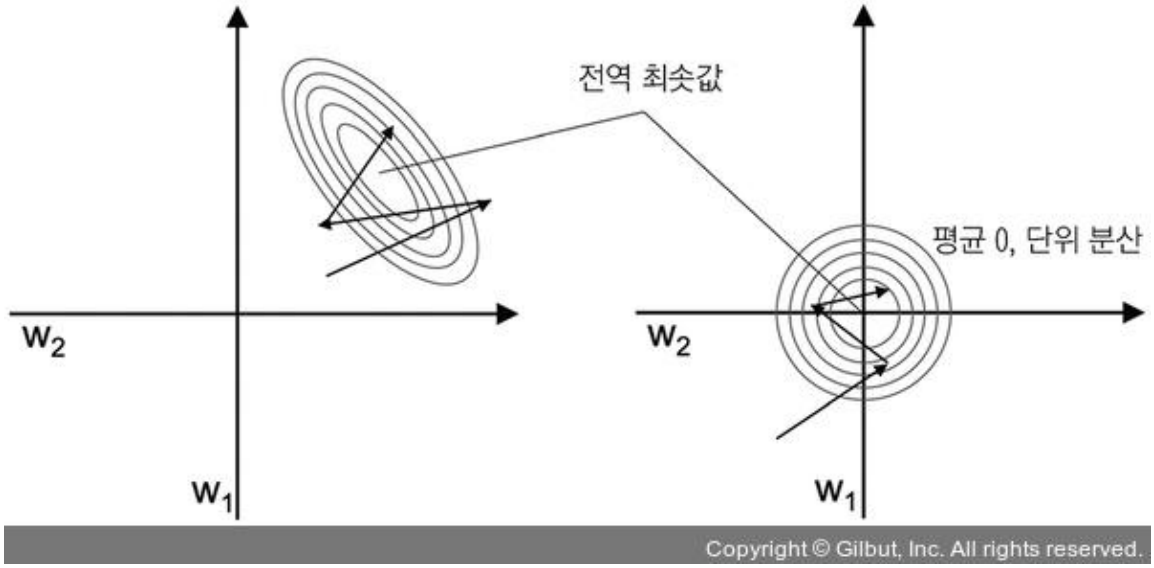
왼) 비용이 점차 감소하여 전역 최솟값의 방향으로 이동
오) 너무 큰 학습률은 선택하여 전역 최솟값을 지나침

▼ 2.3.3 특성 스케일을 조정하여 경사 하강법 결과 향상

경사하강법은 특정 스케일을 조정하여 혜택을 볼 수 있는 많은 알고리즘 중 하나이다. 이 절에서는 표준화라고 하는 특성 스케일 방법을 사용한다. 각 특성의 평균을 0에 맞추고 특성의 표준 편차를 1(단위 분산)으로 만든다.

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

모든 샘플에서 평균을 빼고 표준 편차로 나눈 값이다. xj는 n개의 모든 훈련 샘플에서 j번째 특성값을 포함한 벡터이다.

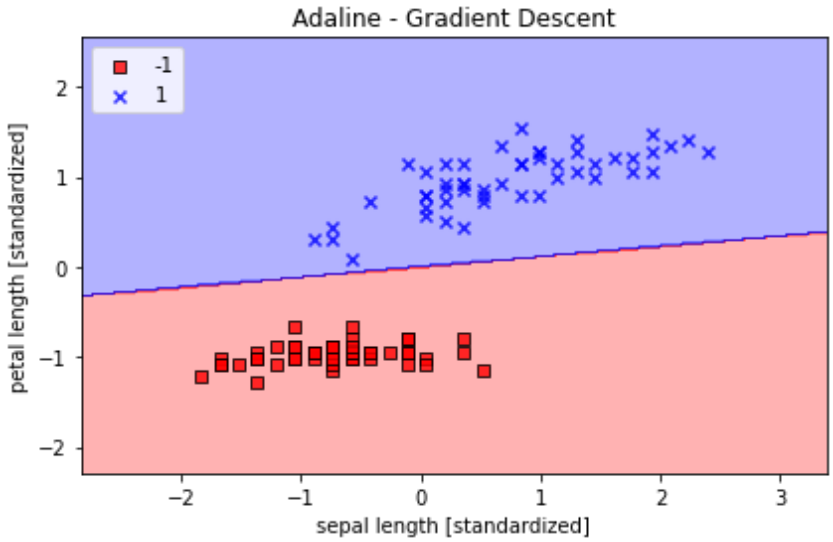


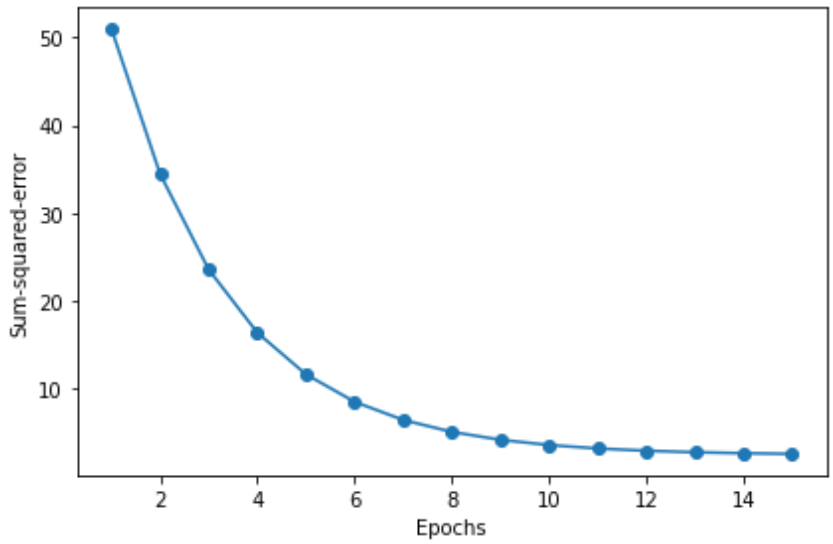
표준화가 비용 함수에 미치는 영향

2차원 분류 모델에서 모델의 가중치에 따른 비용함수의 등고선을 보여준다

```
# 특성을 표준화합니다.
X_std = np.copy(X)
X_std[:, 0] = (X[:, 0] - X[:, 0].mean()) / X[:, 0].std()
X_std[:, 1] = (X[:, 1] - X[:, 1].mean()) / X[:, 1].std()
```

```
ada = AdalineGD(n_iter=15, eta=0.01)
ada.fit(X_std, y)

plot_decision_regions(X_std, y, classifier=ada)
plt.title('Adaline - Gradient Descent')
plt.xlabel('sepal length [standardized]')
plt.ylabel('petal length [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()

plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Sum-squared-error')

plt.tight_layout()
plt.show()
```





학습률 n=0.01을 사용하고 표준화된 특성에서 훈련하니 아달린 모델이 수렴했다. 모든 샘플이 완벽하게 분류되더라도 SSE가 0이 되지는 않았다.

▼ 2.3.4 대규모 머신 러닝과 확률적 경사 하강법

배치 경사 하강법을 실행하면 계산 비용이 높다. 단계마다 매번 전체 훈련 데이터셋을 다시 평가 해야 하기 때문이다.

확률적 경사 하강법(반복 또는 온라인 경사 하강법)은 배치 경사 하강법의 다른 대안이다. 다음 첫 번째 수식처럼 모든 샘플에 대하여 누적된 오차의 합을 기반으로 가중치를 업데이트하는 대신 두 번째 수식처럼 각 훈련 샘플에 대해 조금씩 가중치를 업데이트한다.

$$\Delta w = n \sum_{i} (y^{(i)} - \phi(z^{(i)}))x_j^{(i)}$$

$$\Delta w = n(y^{(i)} - \phi(z^{(i)}))x_j^{(i)}$$

-가중치가 더 자주 업데이트 되기 때문에 수렴 속도가 훨씬 빠르다. (하지만 오차 궤적은 배치 경사 하강법보다 훨씬 복잡하다.)
-만족스러운 결과를 얻기 위해 훈련 샘플 순서를 무작위하게 주입하는 것이 중요하다.
-또 순환되지 않도록 에포크마다 훈련 데이터셋을 섞는 것이 좋다.

-온라인 학습으로 사용할 수 있다.(온라인 학습에서 모델은 새로운 훈련 데이터가 도착하는 대로 훈련, 많은 양의 데이터)

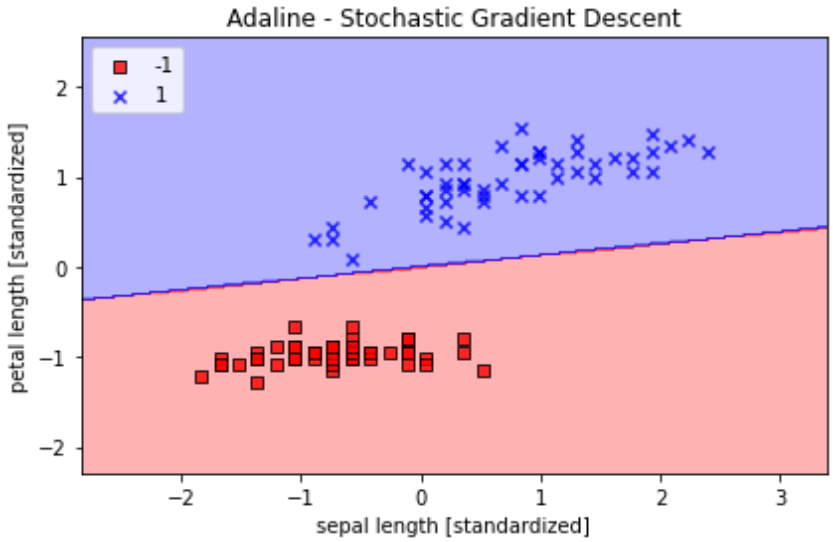
```
class AdalineSGD(object):
    """ADAptive LInear NEuron 분류기

    Parameters
    ------------
    eta : float
      학습률 (0.0과 1.0 사이)
    n_iter : int
      훈련 데이터셋 반복 횟수
    shuffle : bool (default: True)
      True로 설정하면 같은 반복이 되지 않도록 에포크마다 훈련 데이터를 섞습니다
    random_state : int
      가중치 무작위 초기화를 위한 난수 생성기 시드

    Attributes
    -----------
    w_ : 1d-array
      학습된 가중치
    cost_ : list
      모든 훈련 샘플에 대해 에포크마다 누적된 평균 비용 함수의 제곱합

    """
    def __init__(self, eta=0.01, n_iter=10, shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.shuffle = shuffle
        self.random_state = random_state

    def fit(self, X, y):
        """훈련 데이터 학습

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
          n_samples 개의 샘플과 n_features 개의 특성으로 이루어진 훈련 데이터
        y : array-like, shape = [n_samples]타깃 벡터

        반환값
        -------
        self : object

        """
```

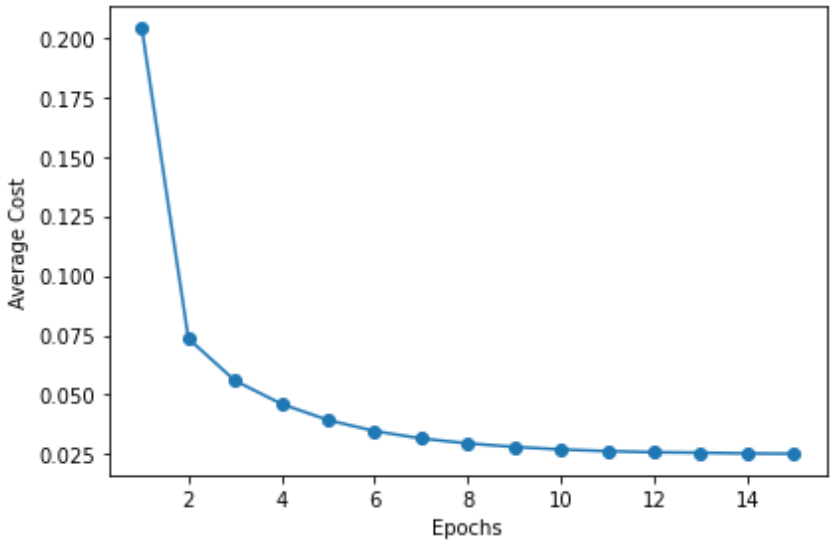
```
        self._initialize_weights(X.shape[1])
        self.cost_ = []
        for i in range(self.n_iter):
            if self.shuffle:
                X, y = self._shuffle(X, y)
            cost = []
            for xi, target in zip(X, y):
                cost.append(self._update_weights(xi, target))
            avg_cost = sum(cost) / len(y)
            self.cost_.append(avg_cost)
        return self

    def partial_fit(self, X, y):
        """가중치를 다시 초기화하지 않고 훈련 데이터를 학습합니다"""
        if not self.w_initialized:
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:
            self._update_weights(X, y)
        return self

    def _shuffle(self, X, y):
        """훈련 데이터를 섞습니다"""
        r = self.rgen.permutation(len(y))
        return X[r], y[r]

    def _initialize_weights(self, m):
        """랜덤한 작은 수로 가중치를 초기화합니다"""
        self.rgen = np.random.RandomState(self.random_state)
        self.w_ = self.rgen.normal(loc=0.0, scale=0.01, size=1 + m)
        self.w_initialized = True

    def _update_weights(self, xi, target):
        """아달린 학습 규칙을 적용하여 가중치를 업데이트합니다"""
        output = self.activation(self.net_input(xi))
        error = (target - output)
        self.w_[1:] += self.eta * xi.dot(error)
        self.w_[0] += self.eta * error
        cost = 0.5 * error**2
        return cost

    def net_input(self, X):
        """최종 입력 계산"""
        return np.dot(X, self.w_[1:]) + self.w_[0]
```

```
    def activation(self, X):
        """선형 활성화 계산"""
        return X

    def predict(self, X):
        """단위 계단 함수를 사용하여 클래스 레이블을 반환합니다"""
        return np.where(self.activation(self.net_input(X)) >= 0.0, 1, -1)
```

▶️fit 메서드
각 훈련 샘플에 대해 가중치를 업데이트 할 것
훈련 후 알고리즘이 수렴하는 확인하려고 에포크마다 훈련 샘플의 평균 비용을 계산함
비용 함수를 최적화 할 때 반복적인 순환이 이러나지 않도록 매 에포크에 훈련 샘플을 섞음
▶️partial_fit 메서드
가중치를 다시 초기화하지 않아 온라인 학습에서 사용할 수 있음
▶️_suffle 메서드
0부터 100까지 중복되지 않은 랜덤한 숫자 시퀀스 생성
이 숫자 시퀀스를 특성 행렬과 클래스 레이블 벡터를 섞는 인덱스로 사용

```
ada = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
ada.fit(X_std, y)

plot_decision_regions(X_std, y, classifier=ada)
plt.title('Adaline - Stochastic Gradient Descent')
plt.xlabel('sepal length [standardized]')
plt.ylabel('petal length [standardized]')
plt.legend(loc='upper left')

plt.tight_layout()
plt.show()

plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Average Cost')

plt.tight_layout()
plt.show()
```


Adaline - Stochastic Gradient Descent
-1
1
petal length [standardized]
sepal length [standardized]


Average Cost
Epochs

평균 비용 빠르게 감소