ROKEY BOOT CAMP

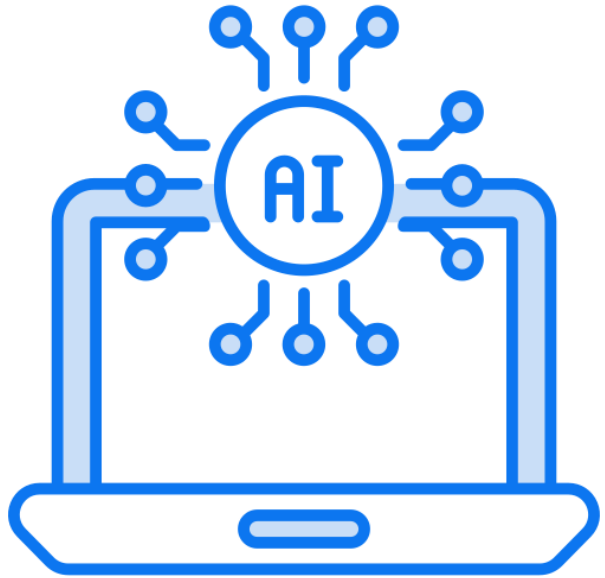
〈심화보충〉
AI(Computer Vision)개론

[4 차시] 분류 알고리즘(1)

김 균 창 강사

학습 내용

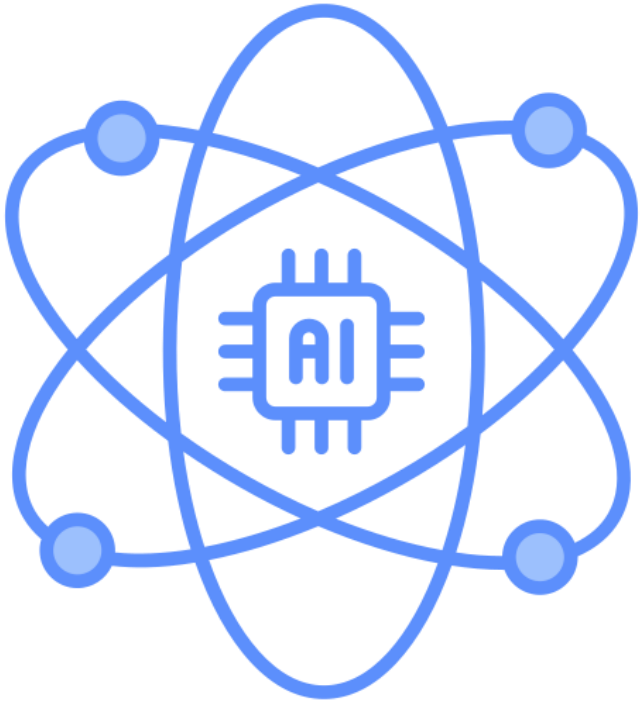
1. 데이터를 지식으로 바꾸는 지능적인 시스템 구축
2. 머신 러닝의 세 가지 종류
3. 기본 용어와 표기법 소개
4. 머신 러닝 시스템 구축 로드맵
5. 머신 러닝을 위한 파이썬



4.1 인공뉴런
: 초기 머신 러닝의 간단한 역사

적응형 선형 뉴런과 학습의 수렴

- 아달린 알고리즘은 진짜 클래스 레이블과 선형 활성화 함수의 실수 출력 값을 비교하여 모델의 오차를 계산하고 가중치를 업데이트
- 반대로 퍼셉트론은 진짜 클래스 레이블과 예측 클래스 레이블을 비교



4.3 적응형 선형 뉴런과 학습의 수렴

경사 하강법으로 비용 함수 최소화

- 지도 학습 알고리즘의 핵심 구성 요소 중 하나는 학습 과정 동안 최적화하기 위해 정의한 **목적 함수**(object function)
- 종종 최소화하려는 비용 함수가 목적 함수가 됨
- 아달린은 계산된 출력과 진짜 클래스 레이블 사이의 **제곱 오차합** (Sum of Squared Errors, SSE)으로 가중치를 학습하기 위한 비용 함수 J를 정의

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

경사 하강법으로 비용 함수 최소화

- $\frac{1}{2}$ 항은 다음 문단에서 설명할 가중치 파라미터에 대한 비용 함수 또는 손실 함수의 그레이디언트(gradient)를 간소하게 만들려고 편의상 추가한 것
- 단위 계단 함수 대신 연속적인 선형 활성화 함수를 사용하는 장점은 비용 함수가 미분 가능해진다는 것이 비용 함수의 또 다른 장점은 볼록 함수라는 것
- 간단하지만 강력한 최적화 알고리즘인 **경사 하강법**(gradient descent)을 적용하여 붓꽃 데이터셋의 샘플을 분류하도록 비용 함수를 최소화하는 가중치를 찾을 수 있음

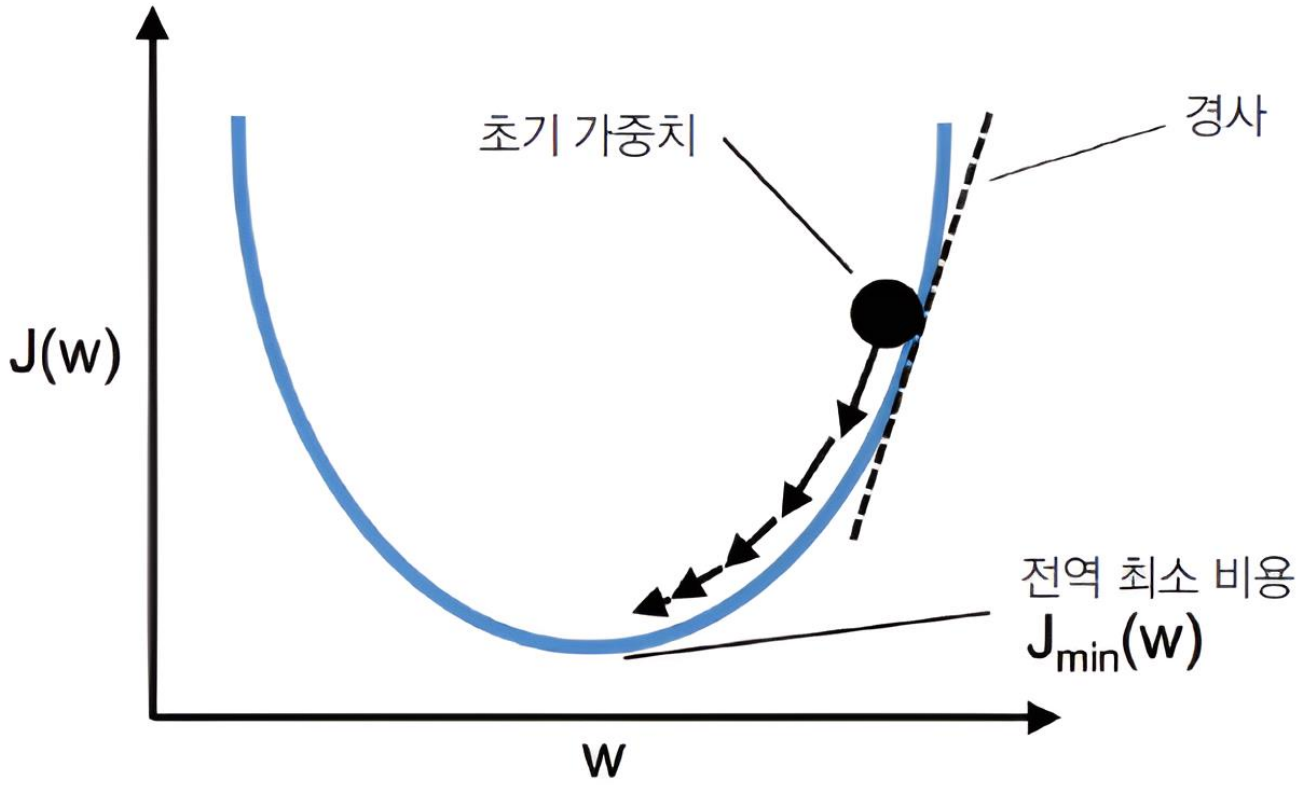
4.3 적응형 선형 뉴런과 학습의 수렴

경사 하강법으로 비용 함수 최소화

- 그림 2-10에서는 경사 하강법 이면에 있는 핵심 아이디어를 지역 또는 전역 최소값에 도달할 때까지 언덕을 내려옴
- 각 반복에서 경사의 반대 방향으로 진행
- 진행 크기는 경사의 기울기와 학습률로 결정

4.3 적응형 선형 뉴런과 학습의 수렴

▼ 그림 2-10 경사 하강법 알고리즘



4.3 적응형 선형 뉴런과 학습의 수렴

경사 하강법으로 비용 함수 최소화

- 경사 하강법을 사용하면 비용 함수 J(w)의 그레이디언트 ∇J(w) 반대 방향으로 조금씩 가중치를 업데이트할 수 있음

$$w := w + \Delta w$$

$$\Delta w = -\eta \nabla J(w)$$

- 가중치 변화량 Δw는 음수의 그레이디언트에 학습률 η를 곱한 것으로 정의

4.3 적응형 선형 뉴런과 학습의 수렴

경사 하강법으로 비용 함수 최소화

· 비용 함수의 그레이디언트를 계산하려면 각 가중치 w_j에 대한 편도 함수를 계산해야 함

$$\frac{\partial J}{\partial w_j} = -\sum_i \left(y^{(i)} - \phi\left(z^{(i)} \right) \right) x_j^{(i)}$$

· 가중치 w_j의 업데이트 공식을 다음과 같이 쓸 수 있음

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)} \right) \right) x_j^{(i)}$$

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}$$

· 모든 가중치가 동시에 업데이트되기 때문에 아달린 학습 규칙은 다음과 같음

4.3 적응형 선형 뉴런과 학습의 수렴

경사 하강법으로 비용 함수 최소화

- 아달린 학습 규칙이 퍼셉트론 규칙과 동일하게 보이지만 $z^{(i)} = \mathbf{w}^T\mathbf{x}^{(i)}$인

  $\phi\left(z^{(i)}\right)$는 정수 클래스 레이블이 아니고 실수임

- 또한, 훈련 데이터셋에 있는 모든 샘플을 기반으로 가중치 업데이트를 계산

  (각 샘플마다 가중치를 업데이트하지는 않음)

- 이 방식을 **배치 경사 하강법**(batch gradient descent)이라고도 함

파이썬으로 아달린 구현

- 퍼셉트론 규칙과 아달린이 매우 비슷하기 때문에 앞서 정의한 퍼셉트론 구현에서 fit 메서드를 바꾸어 경사 하강법으로 비용 함수가 최소화되도록 가중치를 업데이트

```
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
```

파이썬으로 아달린 구현

```
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
```

파이썬으로 아달린 구현

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

반환값
-------
self : object
```

파이썬으로 아달린 구현

```
        """
        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)
            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
```

파이썬으로 아달린 구현

```
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)
```

4.3 적응형 선형 뉴런과 학습의 수렴

파이썬으로 아달린 구현

- 퍼셉트론처럼 개별 훈련 샘플마다 평가한 후 가중치를 업데이트하지 않고

  전체 훈련 데이터셋을 기반으로 그레이디언트를 계산

- 절편(0번째 가중치)은 self.eta * errors.sum()이고 가중치 1에서 m까지는

  self.eta * X.T.dot(errors)

- 여기서 X.T.dot(errors)는 특성 행렬과 오차 벡터 간의 행렬-벡터 곱셈

4.3 적응형 선형 뉴런과 학습의 수렴

파이썬으로 아달린 구현

- 이 코드의 activation 메서드는 단순한 항등 함수(identity function)이기 때문에 아무런 영향을 미치지 않음
- 단일층 신경망을 통해 정보가 어떻게 흘러가는지 일반적인 개념을 표시하려고 (activation 메서드에서 계산되는) 활성화 함수를 추가
- 입력 데이터의 특성에서 최종 입력, 활성화, 출력 순으로 진행

4.3 적응형 선형 뉴런과 학습의 수렴

파이썬으로 아달린 구현

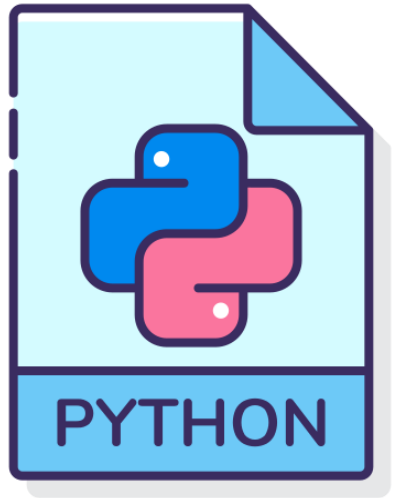
- 다음 장에서는 항등 함수가 아니고 비선형 활성화 함수를 사용하는 로지스틱 회귀 분류기
- 로지스틱 회귀 모델은 활성화 함수와 비용 함수만 다르고 아달린과 매우 비슷함
- 이전 퍼셉트론 구현과 마찬가지로 비용을 self.cost_ 리스트에 모아서 훈련이 끝난 후 알고리즘이 수렴하는지 확인

4.3 적응형 선형 뉴런과 학습의 수렴

파이썬으로 아달린 구현

- 실전에서는 최적으로 수렴하는 좋은 학습률 η를 찾기 위해 여러 번 실험을 해야 함
- 두 개의 학습률 $\eta = 0.1$과 $\eta = 0.0001$을 선택해 보자
- 에포크 횟수 대비 비용 함수의 값을 그래프로 나타내면 아달린 구현이

  훈련 데이터에서 얼마나 잘 학습하는지 볼 수 있음



파이썬으로 아달린 구현

- 두 학습률에서 에포크 횟수 대비 비용 그래프를 그려 보자

```
>>> 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)
```

파이썬으로 아달린 구현

```
>>> 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()
```

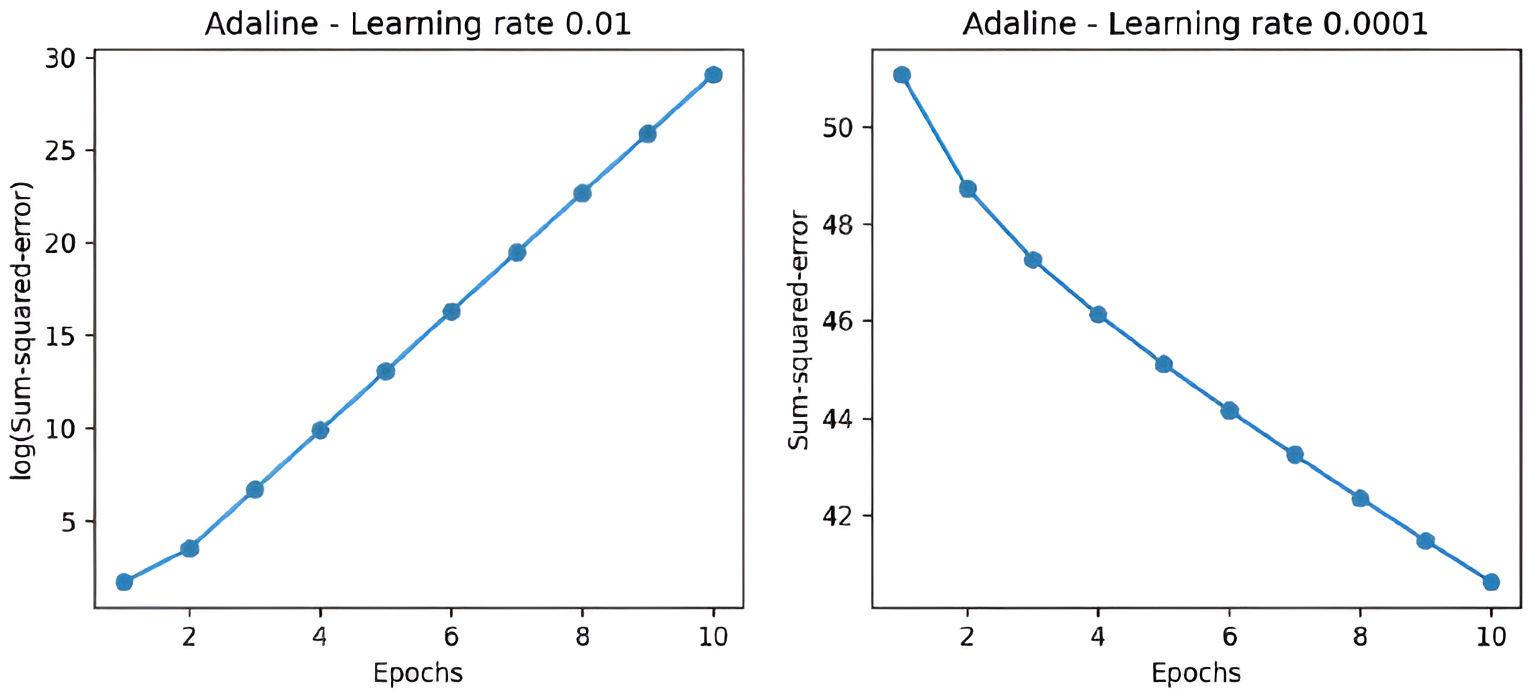
4.3 적응형 선형 뉴런과 학습의 수렴

파이썬으로 아달린 구현

- 출력된 비용 함수 그래프에서 볼 수 있듯이 두 개의 다른 문제가 발생
- 그림 2-11의 왼쪽 그래프는 학습률이 너무 클 때 발생
- 비용 함수를 최소화하지 못하고 오차는 에포크마다 점점 더 커짐
- 전역 최솟값을 지나쳤기 때문임
- 반면 오른쪽 그래프에서는 비용이 감소하지만 학습률 $\eta = 0.0001$은
- 너무 작기 때문에 알고리즘이 전역 최솟값에 수렴하려면 아주 많은 에포크가 필요함

4.3 적응형 선형 뉴런과 학습의 수렴

▼ 그림 2-11 학습률에 따른 아달린 알고리즘의 수렴



4.3 적응형 선형 뉴런과 학습의 수렴

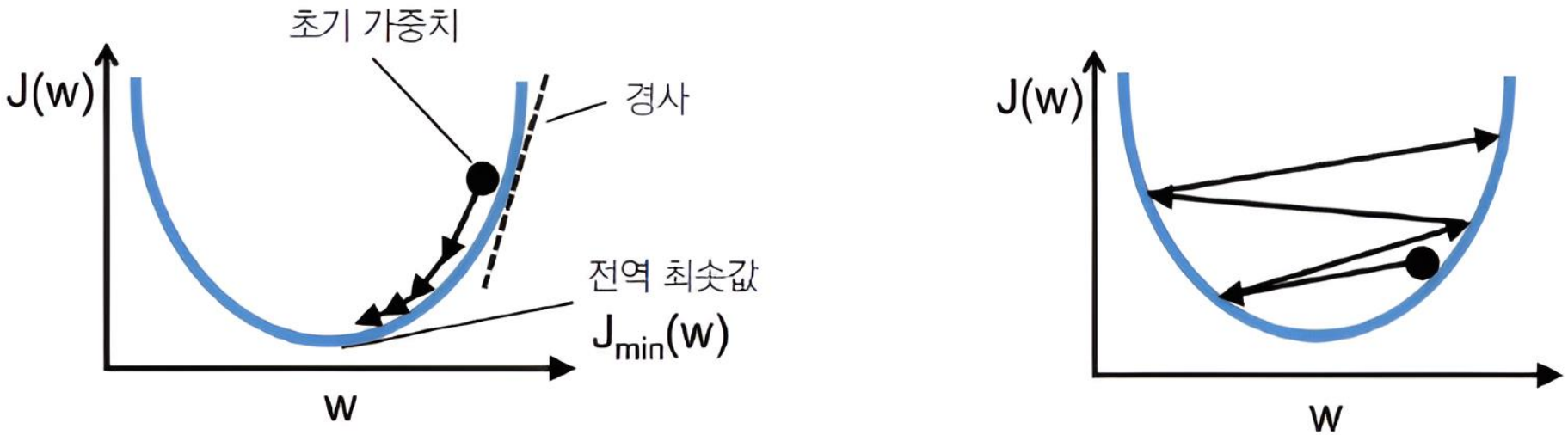
파이썬으로 아달린 구현

- 그림 2-12는 비용 함수 J를 최소화하려고 특정 가중치 값을 바꾸었을 때 어떤 일이 일어나는지 보여줌
- 왼쪽 그림은 적절하게 선택한 학습률의 경우
- 비용이 점차 감소하여 전역 최솟값의 방향으로 이동

파이썬으로 아달린 구현

- 오른쪽 그림은 너무 큰 학습률을 선택하여 전역 최솟값을 지나쳤음

▼ 그림 2-12 경사 하강법에서 학습률의 영향



4.3 적응형 선형 뉴런과 학습의 수렴

특성 스케일을 조정하여 경사 하강법 결과 향상

- 책에서 살펴볼 머신 러닝 알고리즘들은 최적의 성능을 위해 어떤 식으로든지 특성 스케일을 조정하는 것이 필요함
- 경사 하강법은 특성 스케일을 조정하여 혜택을 볼 수 있는 많은 알고리즘 중 하나임
- 이 절에서는 **표준화**(standardization)라고 하는 특성 스케일 방법을 사용
- 이 정규화 과정은 데이터에 평균이 0이고 단위 분산을 갖는 표준 정규 분포의 성질을 부여하여 경사 하강법 학습이 좀 더 빠르게 수렴되도록 도움
- 원본 데이터셋을 정규 분포로 만드는 것은 아님
- 표준화는 각 특성의 평균을 0에 맞추고 특성의 표준 편차를 1(단위 분산)로 만듦

특성 스케일을 조정하여 경사 하강법 결과 향상

- 예를 들어 j번째 특성을 표준화하려면 모든 샘플에서 평균 μ_j를 빼고 표준 편차 σ_j로 나누면 됨

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

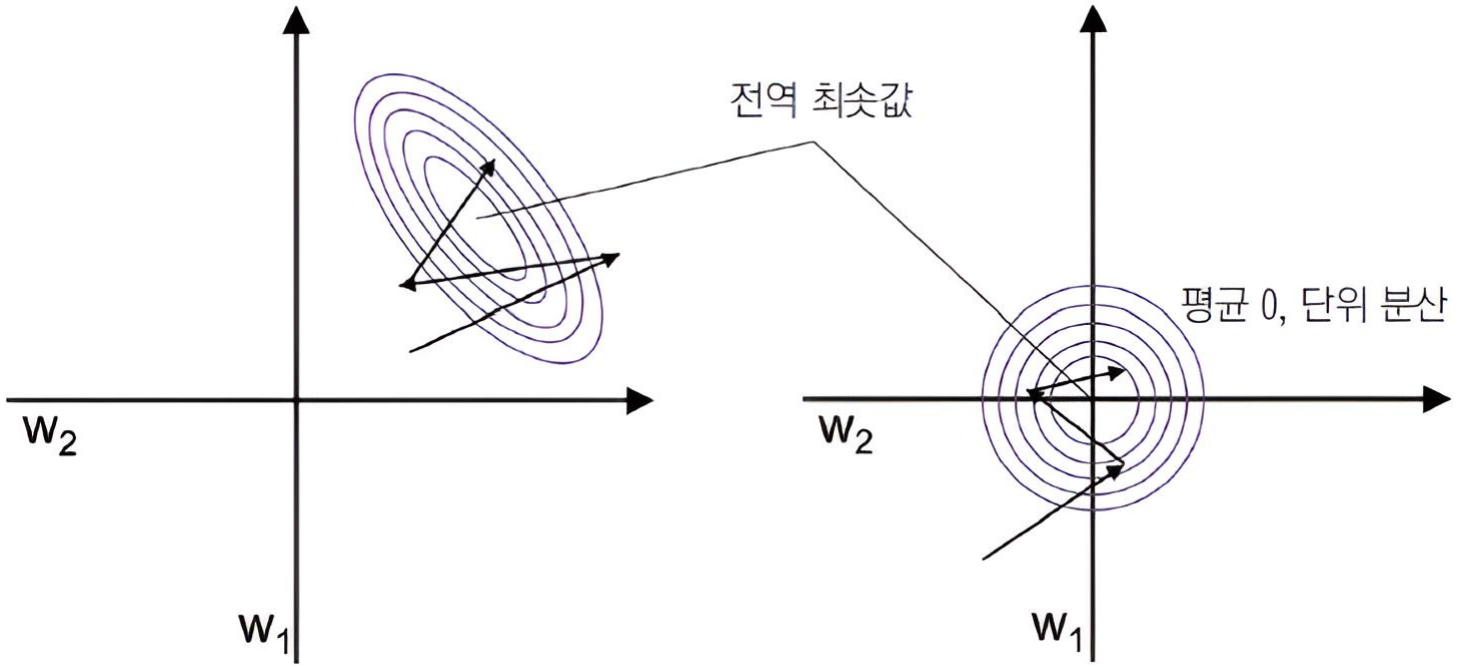
4.3 적응형 선형 뉴런과 학습의 수렴

특성 스케일을 조정하여 경사 하강법 결과 향상

- 여기서 x_j는 n개의 모든 훈련 샘플에서 j번째 특성 값을 포함한 벡터
- 표준화 기법을 데이터셋의 각 특성 j에 적용
- 표준화가 경사 하강법 학습에 도움이 되는 이유 중 하나는 그림 2-13에 나온 것처럼 더 적은 단계를 거쳐 최적 혹은 좋은 솔루션을 찾기 때문임
- 그림 2-13은 2차원 분류 문제에서 모델의 가중치에 따른 비용 함수의 등고선을 보여 줌

4.3 적응형 선형 뉴런과 학습의 수렴

▼ 그림 2-13 표준화가 비용 함수에 미치는 영향



특성 스케일을 조정하여 경사 하강법 결과 향상

- 표준화는 넘파이 내장 함수 mean과 std로 간단하게 처리할 수 있음

```
>>> 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()
```

특성 스케일을 조정하여 경사 하강법 결과 향상

- 표준화한 후 다시 아달린 모델을 훈련하고 학습률 $\eta = 0.01$에서 몇 번의 에포크 테스트

```
>>> 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()
```

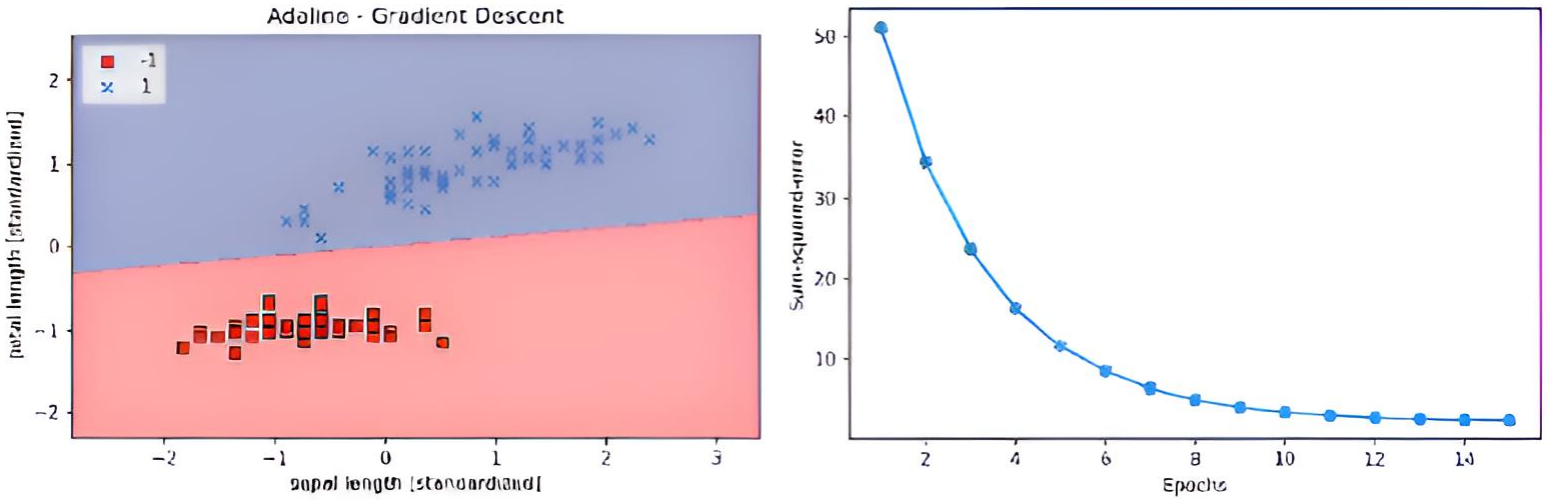
특성 스케일을 조정하여 경사 하강법 결과 향상

```
>>> 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()
```

특성 스케일을 조정하여 경사 하강법 결과 향상

- 이 코드를 실행하면 그림 2-14와 같은 결정 경계 그래프와 비용이 감소되는 그래프를 볼 수 있음

▼ 그림 2-14 표준화를 적용한 아달린의 결정 경계와 학습 곡선



4.3 적응형 선형 뉴런과 학습의 수렴

특성 스케일을 조정하여 경사 하강법 결과 향상

- 이 그래프에서 볼 수 있듯이 학습률 $\eta = 0.01$을 사용하고 표준화된 특성에서 훈련하니 아달린 모델이 수렴
- 모든 샘플이 완벽하게 분류되더라도 SSE가 0이 되지는 않음

4.3 적응형 선형 뉴런과 학습의 수렴

대규모 머신 러닝과 확률적 경사 하강법

- 이전 절에서 전체 훈련 데이터셋에서 계산한 그레이디언트의 반대 방향으로 한 걸음씩 진행하여 비용 함수를 최소화하는 방법을 배웠음
- 이 방식을 이따금 **배치 경사 하강법**이라고도 부름
- 수백만 개의 데이터 포인트가 있는 매우 큰 데이터셋을 생각해 보자
- 많은 머신 러닝 애플리케이션에서 이런 데이터셋은 드문 일이 아님
- 이때 배치 경사 하강법을 실행하면 계산 비용이 매우 많이 듦
- 전역 최솟값으로 나아가는 단계마다 매번 전체 훈련 데이터셋을 다시 평가해야 하기 때문임

대규모 머신 러닝과 확률적 경사 하강법

- **확률적 경사 하강법**(stochastic gradient descent)은 배치 경사 하강법의 다른 대안으로 인기가 높음
- 이따금 반복 또는 온라인 경사 하강법이라고도 부름
- 다음 첫 번째 수식처럼 모든 샘플 $\mathbf{x}^{(i)}$에 대하여 누적된 오차의 합을 기반으로 가중치를 업데이트하는

  대신 두 번째 수식처럼 각 훈련 샘플에 대해 조금씩 가중치를 업데이트

$$\Delta \boldsymbol{w} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) \boldsymbol{x}^{(i)}$$

$$\Delta \boldsymbol{w} = \eta \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) \boldsymbol{x}^{(i)}$$

대규모 머신 러닝과 확률적 경사 하강법

- 확률적 경사 하강법을 경사 하강법의 근사로 생각할 수 있지만 가중치가 더 자주 업데이트되기 때문에 수렴 속도가 훨씬 빠름
- 그레이디언트가 하나의 훈련 샘플을 기반으로 계산되므로 오차의 궤적은 배치 경사 하강법보다 훨씬 어지러움
- 비선형 비용 함수를 다룰 때 얕은 지역 최솟값을 더 쉽게 탈출할 수 있어 장점이 되기도 함
- 확률적 경사 하강법에서 만족스러운 결과를 얻으려면 훈련 샘플 순서를 무작위하게 주입하는 것이 중요함
- 또 순환되지 않도록 에포크마다 훈련 데이터셋을 섞는 것이 좋음

대규모 머신 러닝과 확률적 경사 하강법

- 확률적 경사 하강법의 또 다른 장점은 **온라인 학습**(online learning)으로 사용할 수 있다는 것
- 온라인 학습에서 모델은 새로운 훈련 데이터가 도착하는 대로 훈련됨
- 많은 양의 훈련 데이터가 있을 때도 유용함
- 예를 들어 고객 데이터를 처리하는 웹 애플리케이션
- 온라인 학습을 사용해서 시스템은 변화에 즉시 적응
- 저장 공간에 제약이 있다면 모델을 업데이트 한 후 훈련 데이터를 버릴 수 있음

대규모 머신 러닝과 확률적 경사 하강법

- 경사 하강법으로 아달린 학습 규칙을 구현했기 때문에 학습 알고리즘을 조금만 수정하면 확률적 경사 하강법으로 가중치를 업데이트할 수 있음
- fit 메서드 안에서 각 훈련 샘플에 대해 가중치를 업데이트할 것
- 추가로 partial_fit 메서드도 구현
- 이 메서드는 가중치를 다시 초기화하지 않아 온라인 학습에서 사용할 수 있음
- 훈련 후에는 알고리즘이 수렴하는지 확인하려고 에포크마다 훈련 샘플의 평균 비용을 계산
- 비용 함수를 최적화할 때 반복적인 순환이 일어나지 않도록 매 에포크가 일어나기 전에 훈련 샘플을 섞는 옵션을 추가

대규모 머신 러닝과 확률적 경사 하강법

- random_state 매개변수로는 재현 가능하도록 랜덤 시드를 지정할 수 있음

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

    매개변수
    ------------
    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

    속성
    -----------
```

대규모 머신 러닝과 확률적 경사 하강법

```
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
```

대규모 머신 러닝과 확률적 경사 하강법

```
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]
      타깃 벡터

    반환값
    -------
    self : object

    """
```

대규모 머신 러닝과 확률적 경사 하강법

```
        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"""
```

대규모 머신 러닝과 확률적 경사 하강법

```
        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]
```

대규모 머신 러닝과 확률적 경사 하강법

```
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
```

대규모 머신 러닝과 확률적 경사 하강법

```
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)
```

대규모 머신 러닝과 확률적 경사 하강법

- AdalineSGD 분류기에서 사용하는 _shuffle 메서드는 다음과 같이 작동
- np.random 모듈의 permutation 함수로 0에서 100까지 중복되지 않은 랜덤한 숫자 시퀀스(sequence)를 생성
- 이 숫자 시퀀스를 특성 행렬과 클래스 레이블 벡터를 섞는 인덱스로 사용

대규모 머신 러닝과 확률적 경사 하강법

- 그 다음 fit 메서드로 AdalineSGD 분류기를 훈련하고, plot_decision_regions로는 훈련 결과를 그래프로 그림

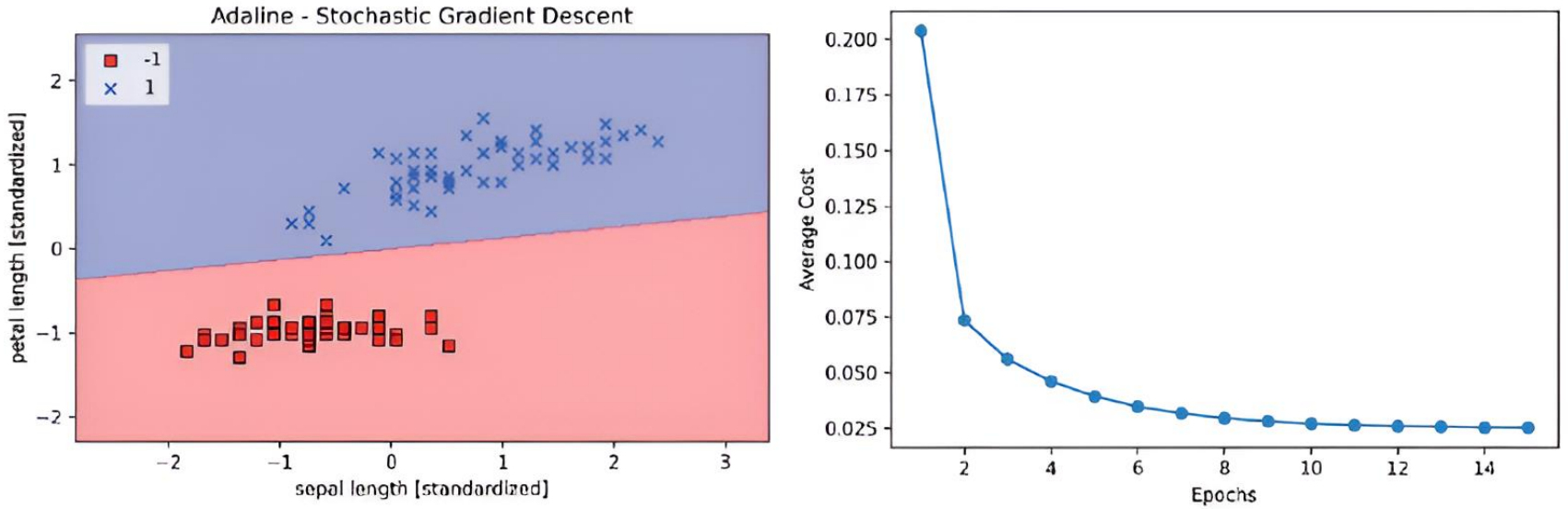
```
>>> ada = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
>>> ada.fit(X_std, y)

>>> 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.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()
```

대규모 머신 러닝과 확률적 경사 하강법

- 앞 코드를 실행하여 출력되는 두 그래프는 그림 2-15와 같음

▼ 그림 2-15 확률적 경사 하강법을 사용한 아달린의 결정 경계와 학습 곡선



대규모 머신 러닝과 확률적 경사 하강법

- 여기서 보듯이 평균 비용이 상당히 빠르게 감소
- 15번째 에포크 이후 최종 결정 경계는 배치 경사 하강법과 거의 비슷해 보임
- 스트리밍 데이터를 사용하는 온라인 학습 방식으로 모델을 훈련하려면 개개의 샘플마다 partial_fit 메서드를 호출하면 됨
- 예를 들어 ada.partial_fit(X_std[0, :], y[0])과 같음

요약

- 이 장에서 지도 학습의 기초적인 선형 분류기 개념 이해
- 퍼셉트론을 구현한 후 벡터화된 경사 하강법 방식으로 적응형 선형 뉴런을 어떻게 효율적으로 훈련의 이해
- 또 확률적 경사 하강법을 사용하여 온라인 학습으로 훈련하는 방법 이해
- 이제 파이썬으로 간단한 분류기를 구현하는 방법
- 파이썬의 사이킷런 머신 러닝 라이브러리에 있는 강력한 고급 머신 러닝 분류기를 사용
- 이런 모델들은 학계와 산업계에서 널리 사용

요약

- 퍼셉트론과 아달린 알고리즘 구현에 사용한 객체 지향 방식이 사이킷런 API를 이해하는 데 도움이 될 것
- 이 장에서 사용한 핵심 구조인 fit, predict 메서드와 동일한 바탕으로 구현되었기 때문임
- 이런 핵심 개념을 기초로 클래스 확률 모델링을 위한 로지스틱 회귀와 비선형 결정 경계를 위한 서포트 벡터 머신을 배우겠음
- 이외에도 다른 종류의 지도 학습 알고리즘으로 앙상블 분류기에 널리 사용되는 트리 기반 알고리즘을 소개

ROKEY BOOT CAMP

수고하셨습니다.