

Machine Learning with Python

Supervised Learning

Contacts

Haesun Park

Email : haesunpark@gmail.com

Meetup: <https://www.meetup.com/Hongdae-Machine-Learning-Study/>

Facebook : <https://facebook.com/haesunpark>

Blog : <https://tensorflow.blog>

Book

파이썬 라이브러리를 활용한 머신러닝, 박해선.

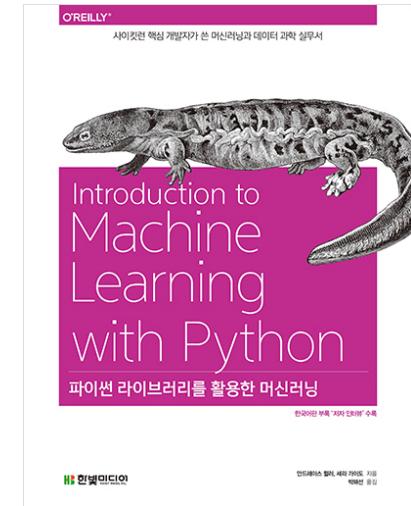
(Introduction to Machine Learning with Python, Andreas Muller & Sarah Guido의 번역서입니다.)

번역서의 1장과 2장은 블로그에서 무료로 읽을 수 있습니다.

원서에 대한 프리뷰를 온라인에서 볼 수 있습니다.

Github:

https://github.com/rickiepark/introduction_to_ml_with_python/



지도 학습

분류 classification와 회귀 regression

분류는 가능성 있는 클래스 레이블 중 하나를 예측

이진 분류 binary classification: 두 개의 클래스 분류(스팸메일, 0-음성클래스, 1-양성클래스)

다중 분류 multiclass classification: 셋 이상의 클래스 분류(붓꽃 품종)

언어의 종류를 분류(한국어와 프랑스어 사이에 다른 언어가 없음)

회귀는 연속적인 숫자(실수)를 예측

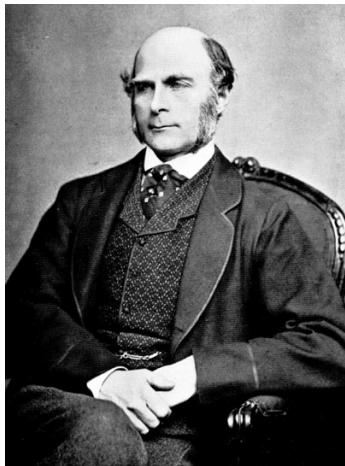
교육, 나이, 주거지를 바탕으로 연간 소득 예측

전년도 수확량, 날씨, 고용자수로 올해 수확량 예측

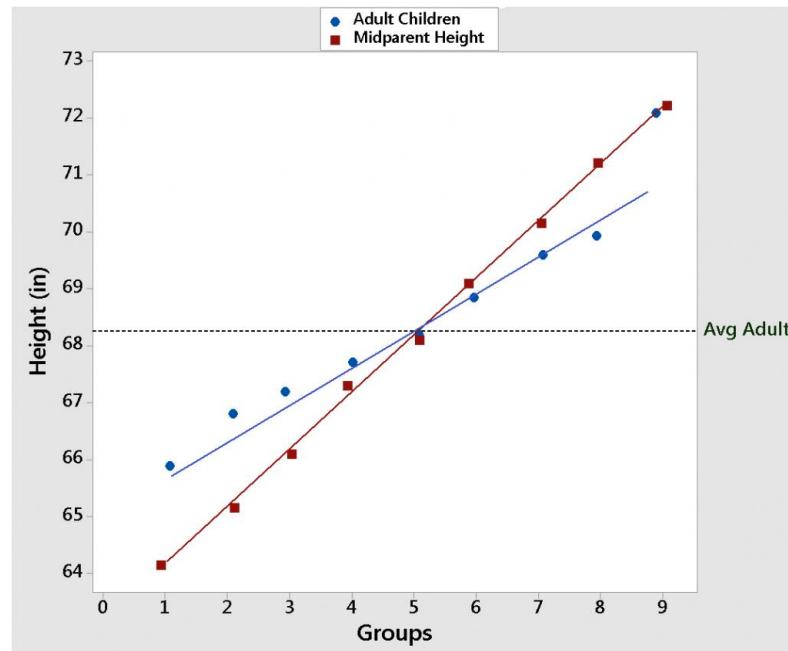
예측 값에 미묘한 차이가 크게 중요하지 않음

회귀 - Regression

“regression toward the mean”



프랜시스 골턴 Francis Galton



일반화, 과대적합, 과소적합

일반화 generalization

훈련 세트로 학습한 모델을 테스트 세트에 적용하는 것

과대적합 overfitting

훈련 세트에 너무 맞추어져 있어 테스트 세트의 성능 저하

과소적합 underfitting

훈련 세트를 충분히 반영하지 못해 훈련 세트, 테스트 세트에서 모두 성능 저하

요트 회사의 고객

45세 이상, 자녀 셋 미만,
이혼하지 않은 고객

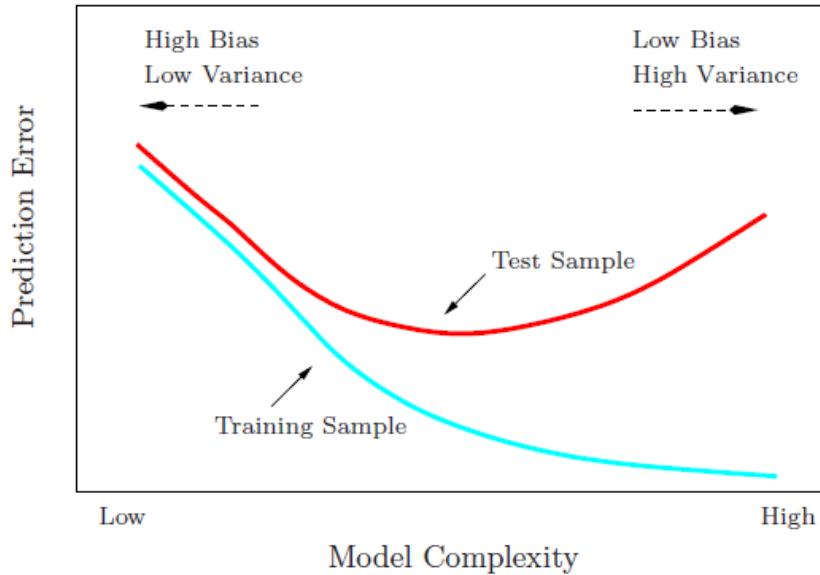
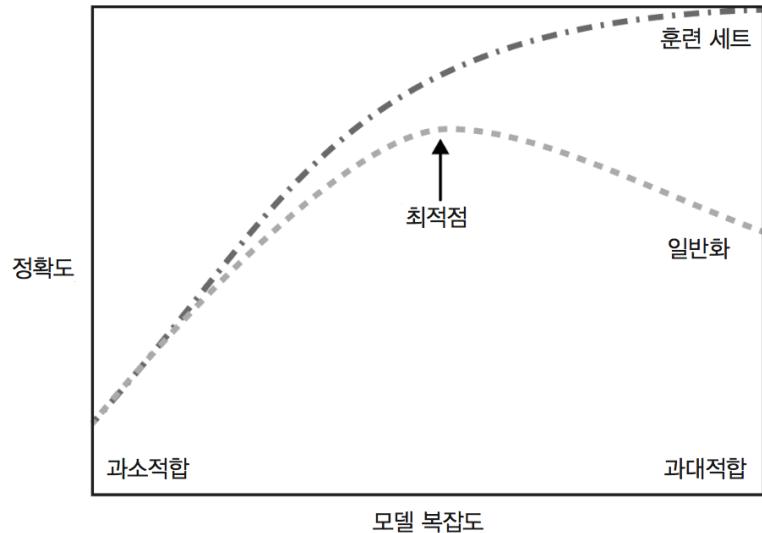
나이	보유차량수	주택보유	자녀수	혼인상태	애완견	보트구매
66	1	yes	2	사별	no	yes
52	2	yes	3	기혼	no	yes
22	0	no	0	기혼	yes	no
25	1	no	1	미혼	no	no
44	0	no	2	이혼	yes	no
39	1	yes	2	기혼	yes	no
26	1	no	2	미혼	no	no
40	3	yes	1	기혼	yes	no
53	2	yes	2	이혼	no	yes
64	2	yes	3	이혼	no	no
58	2	yes	2	기혼	yes	yes
33	1	no	1	미혼	no	no

요트 회사의 고객

집이 있는 고객

나이	보유차량수	주택보유	자녀수	혼인상태	애완견	보트구매
66	1	yes	2	사별	no	yes
52	2	yes	3	기혼	no	yes
22	0	no	0	기혼	yes	no
25	1	no	1	미혼	no	no
44	0	no	2	이혼	yes	no
39	1	yes	2	기혼	yes	no
26	1	no	2	미혼	no	no
40	3	yes	1	기혼	yes	no
53	2	yes	2	이혼	no	yes
64	2	yes	3	이혼	no	no
58	2	yes	2	기혼	yes	yes
33	1	no	1	미혼	no	no

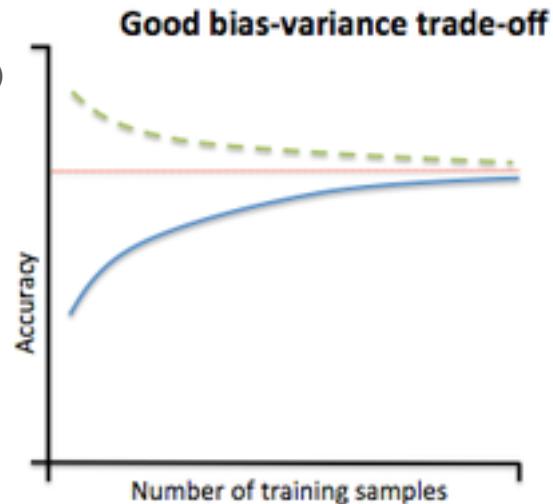
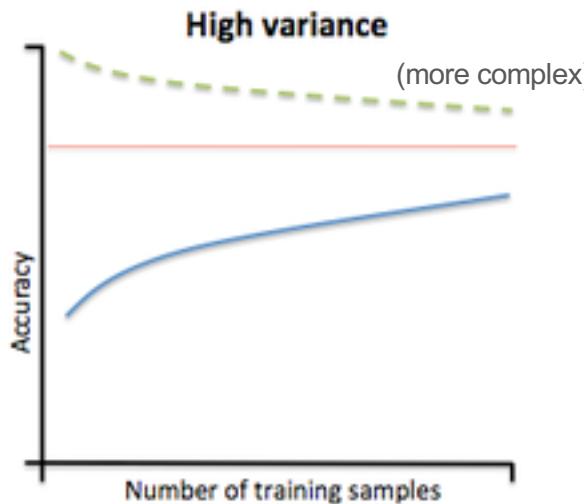
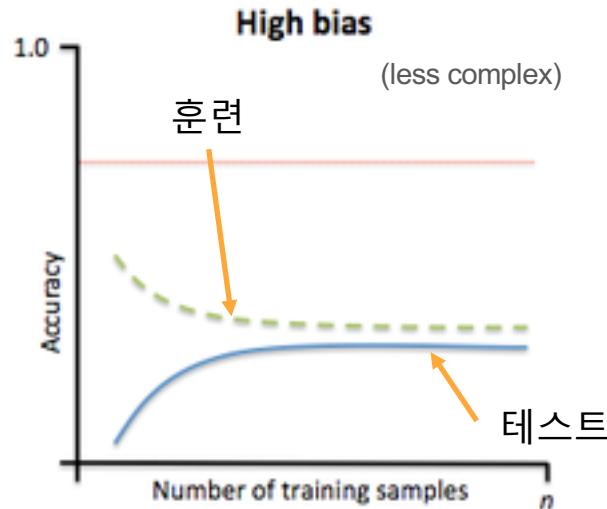
모델 복잡도 곡선



데이터셋과 복잡도 관계

데이터가 많으면 다양성이 커져 복잡한 모델을 만들 수 있음

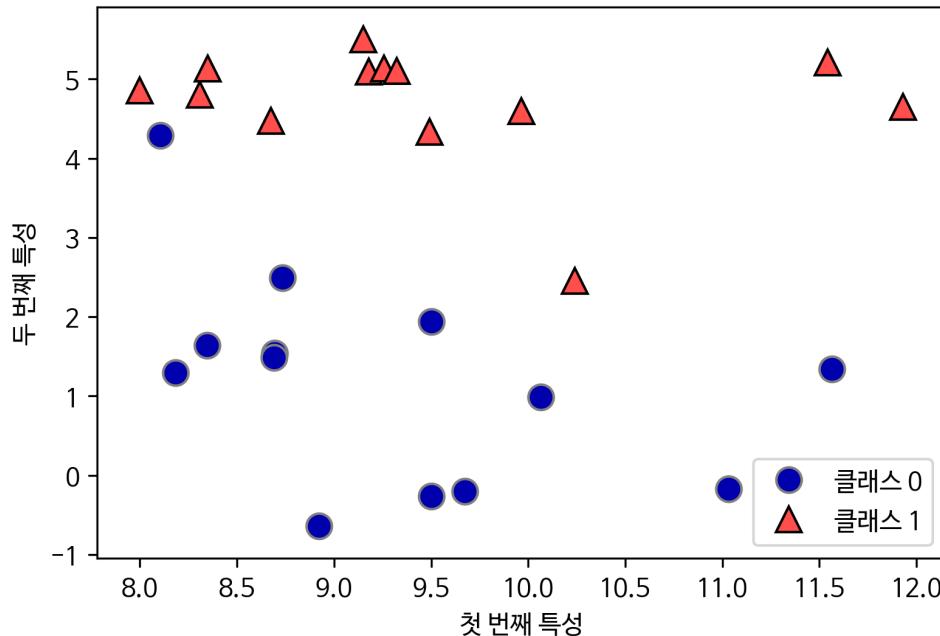
10,000명의 고객 데이터에서 45세 이상, 자녀 셋 미만, 이혼하지 않은 고객이
요트를 사려한다면 이전보다 훨씬 신뢰 높은 모델임



데이터셋

forge 데이터셋

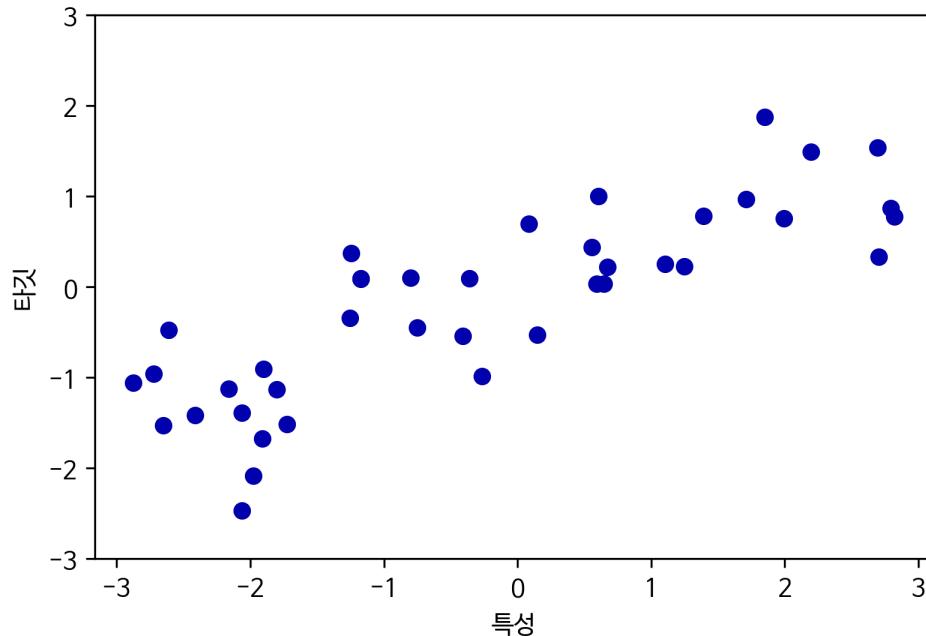
인위적으로 만든 이진 분류 데이터셋, 26개의 데이터 포인트, 2개의 속성



데이터셋

wave 데이터셋

인위적으로 만든 회귀 데이터셋, 40개의 데이터 포인트, 1개의 속성



데이터셋

유방암 데이터셋

위스콘신 유방암 데이터셋, 악성(1)/양성(0)의 이진 분류, `load_breast_cancer()`, 569개의 데이터 포인트, 30개의 특성

보스턴 주택가격 데이터셋

1970년대 보스턴 주변의 주택 평균가격 예측, 회귀 데이터셋, `load_boston()`, 506개의 데이터 포인트, 13개의 특성

확장 보스턴 주택가격 데이터셋

특성끼리 곱하여 새로운 특성을 만듦(특성 공학, 4장), `PolynomialFeatures`, `mglearn.datasets.load_extended_boston()`, 104개의 데이터 포인트

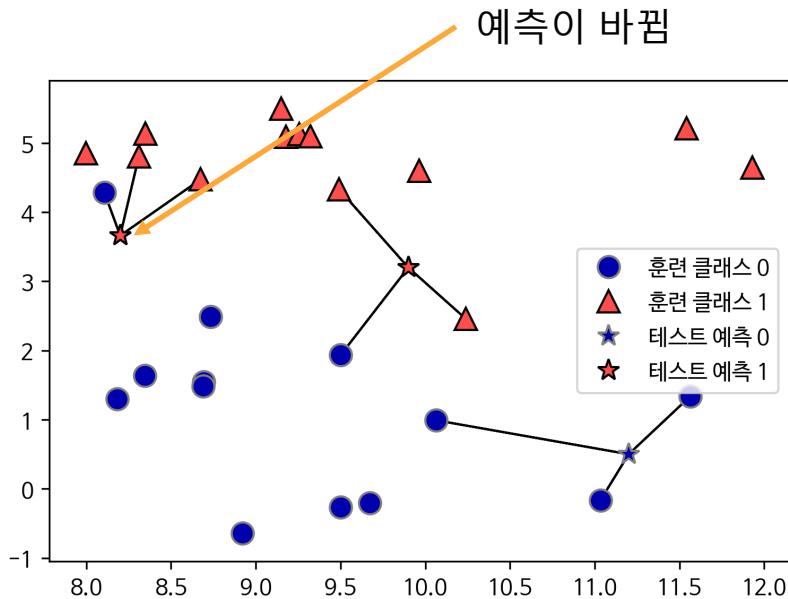
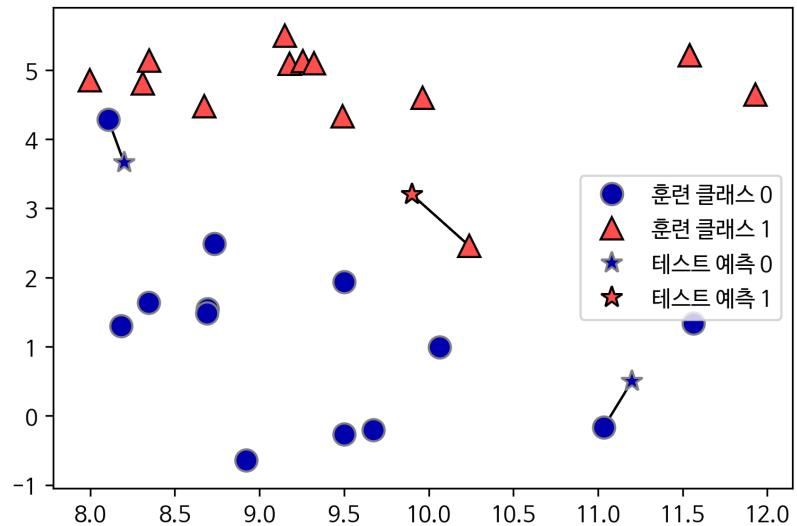
중복 조합 공식은 $\binom{n}{k} = \binom{n+k-1}{k}$ 이므로 $\binom{13}{2} = \binom{13+2-1}{2} = \frac{14!}{2!(14-2)!} = 91$

k-최근접 이웃 분류

k-최근접 이웃 분류

새로운 데이터 포인트에 가까운 이웃 중 다수 클래스(majority voting)가 예측이 됨

forge 데이터셋에 1-최근접 이웃, 3-최근접 이웃 적용



k-NN 분류기

훈련 세트와 테스트 세트로 분리

```
In [13]: from sklearn.model_selection import train_test_split  
X, y = mglearn.datasets.make_forge()  
  
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

```
In [14]: from sklearn.neighbors import KNeighborsClassifier  
clf = KNeighborsClassifier(n_neighbors=3) ← 모델 객체 생성
```

```
In [15]: clf.fit(X_train, y_train) ← 모델 학습
```

```
Out[15]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',  
metric_params=None, n_jobs=1, n_neighbors=3, p=2,  
weights='uniform')
```

```
In [16]: print("테스트 세트 예측: {}".format(clf.predict(X_test)))
```

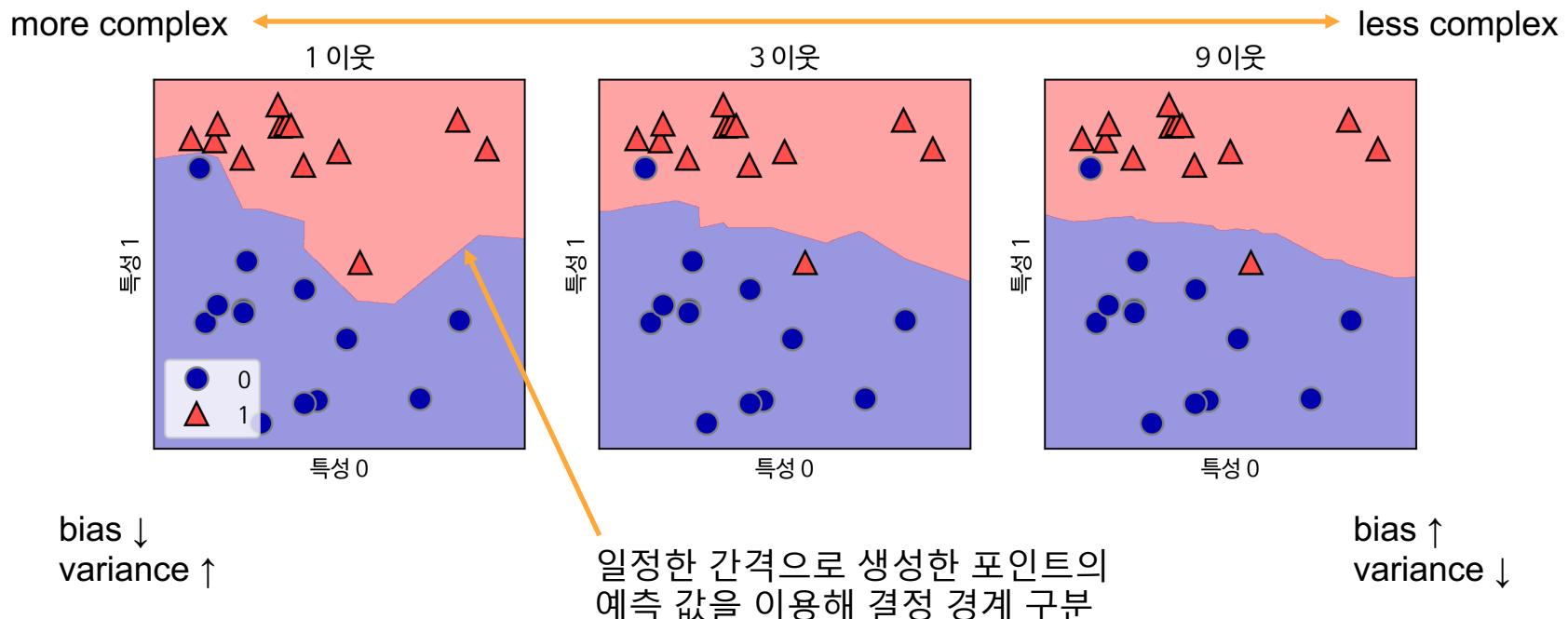
테스트 세트 예측: [1 0 1 0 1 0 0]

모델 평가

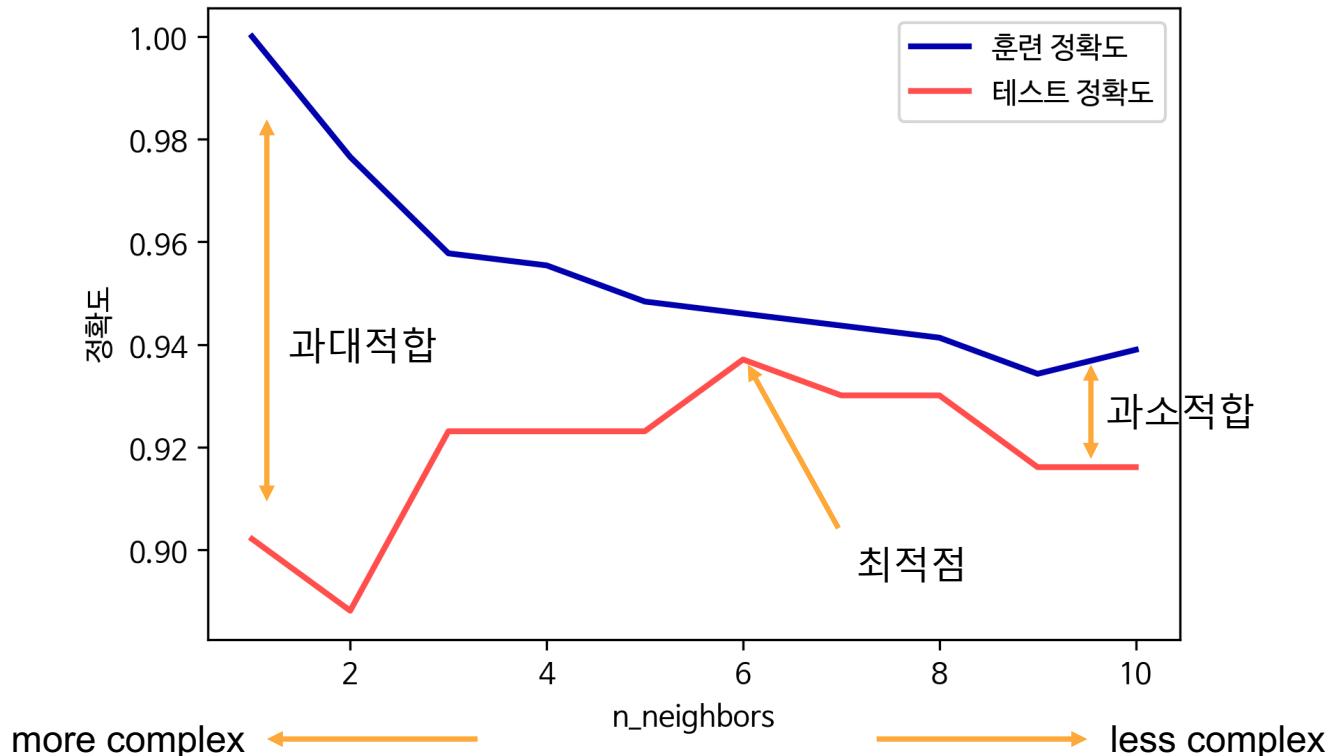
```
In [17]: print("테스트 세트 정확도: {:.2f}".format(clf.score(X_test, y_test)))
```

테스트 세트 정확도: 0.86

KNeighborsClassifier 분석



k-NN 분류기의 모델 복잡도

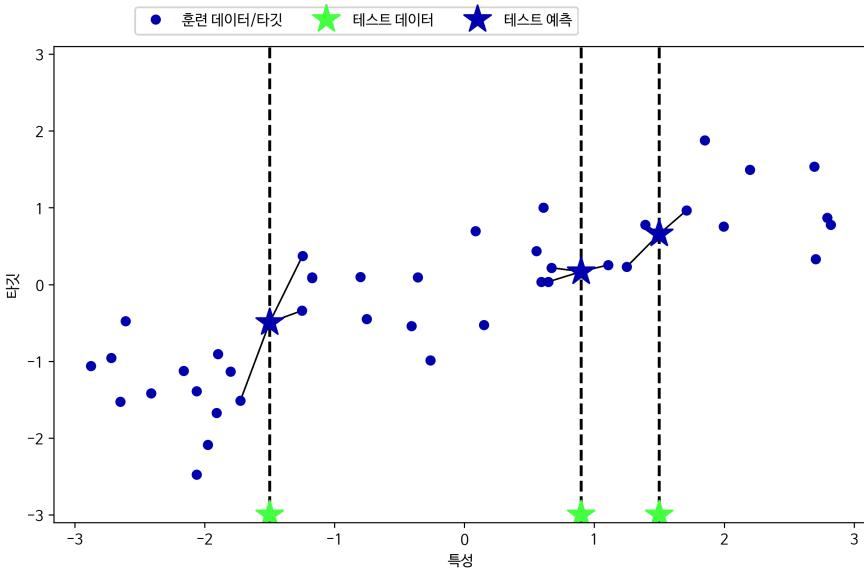
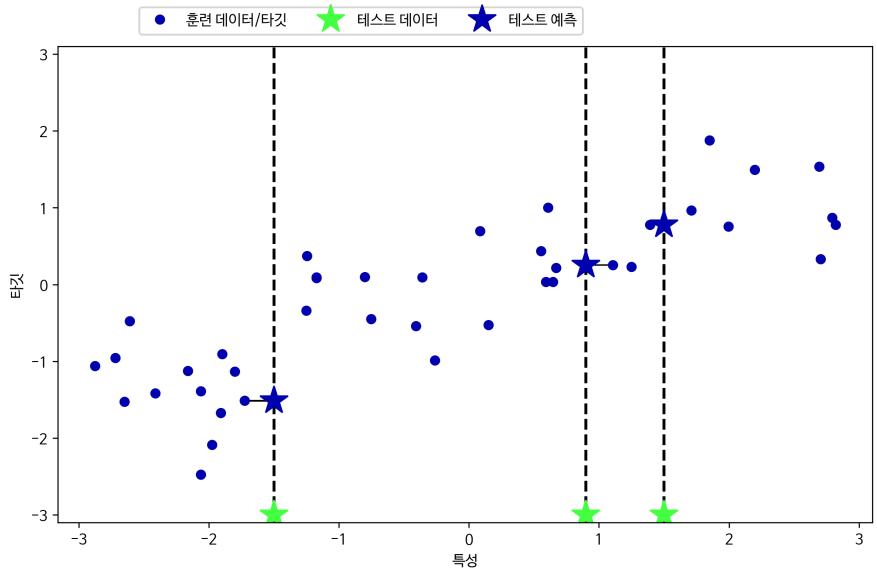


k-최근접 이웃 회귀

k-최근접 이웃 회귀

새로운 데이터 포인트에 가까운 이웃의 출력값 평균이 예측이 됨

wave 데이터셋(1차원)에 1-최근접 이웃, 3-최근접 이웃 적용



k-NN 추정기

```
In [22]: from sklearn.neighbors import KNeighborsRegressor  
X, y = mglearn.datasets.make_wave(n_samples=40)  
  
# wave 데이터셋을 훈련 세트와 테스트 세트로 나눕니다  
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)  
  
# 이웃의 수를 3으로 하여 모델의 객체를 만듭니다  
reg = KNeighborsRegressor(n_neighbors=3)  
# 후려 데이터와 타겟을 사용하여 모델을 학습시킵니다  
reg.fit(X_train, y_train)
```

훈련 세트와 테스트 세트로 분리

모델 객체 생성

모델 학습

```
Out[22]: KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',  
                             metric_params=None, n_jobs=1, n_neighbors=3, p=2,  
                             weights='uniform')
```

```
In [23]: print("테스트 세트 예측:\n{}".format(reg.predict(X_test)))
```

테스트 세트 예측:

[-0.054 0.357 1.137 -1.894 -1.139 -1.631 0.357 0.912 -0.447 -1.139]

모델 평가

```
In [24]: print("테스트 세트 R^2: {:.2f}".format(reg.score(X_test, y_test)))
```

테스트 세트 R^2: 0.83

회귀 모델의 평가

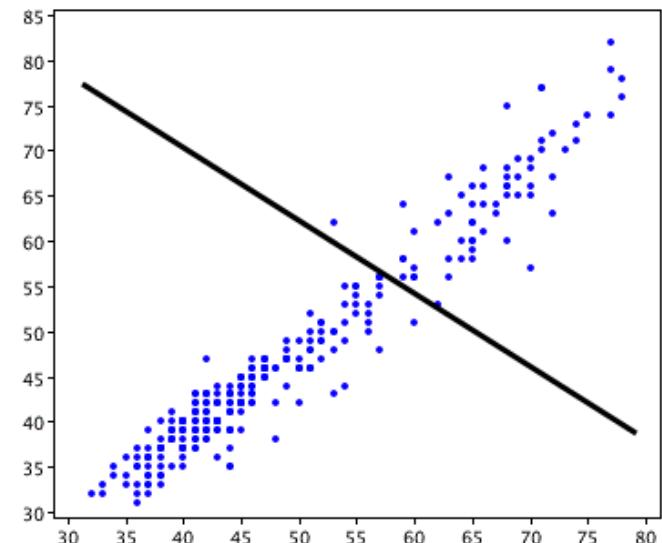
회귀 모델의 score() 함수는 결정 계수 R^2 를 반환

$$R^2 = 1 - \frac{\sum_{i=0}^n (y - \hat{y})^2}{\sum_{i=0}^n (y - \bar{y})^2} \quad y: \text{타깃값} \quad \bar{y}: \text{타깃값의 평균} \quad \hat{y}: \text{모델의 예측}$$

완벽 예측: 타깃값==예측 \rightarrow 문자==0, $R^2 = 1$

타깃값의 평균 정도 예측: 문자==분모, $R^2 = 0$ 이 됨

평균 보다 나쁘게 예측하면 음수가 될 수 있음

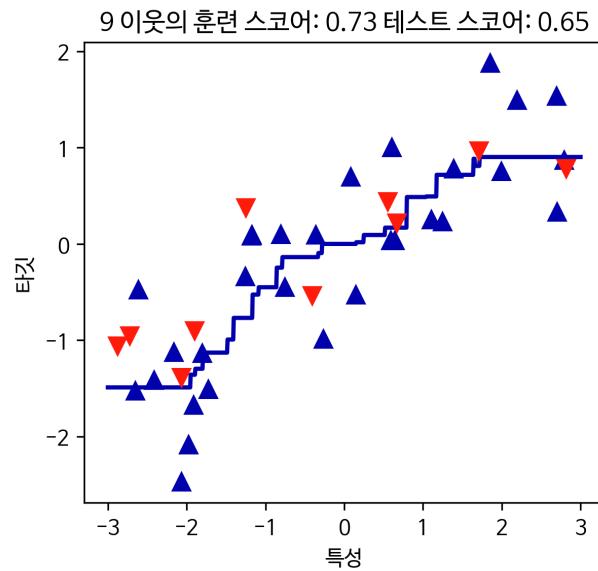
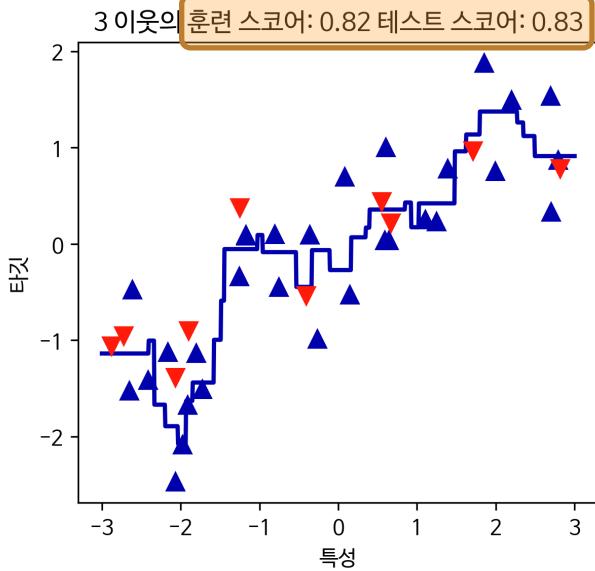
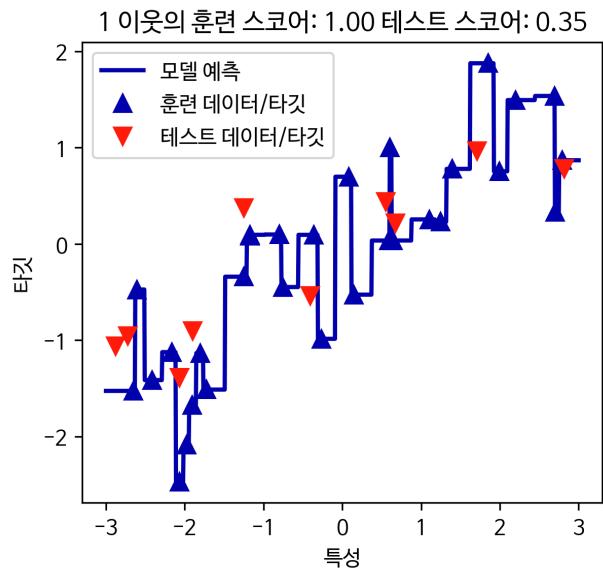


KNeighborsRegressor 분석

more complex



less complex



장단점과 매개변수

K-NN 분류기의 중요 매개변수

포인트 사이의 거리 측정 방법: metric 기본값 ‘minkowski’, p 기본값 2 →

$$\text{유클리디안 거리 } \sqrt{\sum_{i=0}^p (x_1^{(i)} - x_2^{(i)})^2}$$

이웃의 수: 3개나 5개가 보편적

장점 이해하기 쉬움, 특별한 조정 없이 잘 작동, 처음 시도하는 모델로 적합,
 비교적 모델을 빠르게 만들 수 있음

단점 특성 개수나 샘플 개수가 크면 예측이 느림, 데이터 전처리 중요,
 희소한 데이터셋에 잘 작동하지 않음

선형 모델 - 회귀

회귀의 선형 모델

선형 함수(linear model)를 사용하여 예측

$$\hat{y} = w[0] \times x[0] + w[1] \times x[1] + \cdots + w[p] \times x[p] + b$$

특성 : $x[0] \sim x[p]$

특성 개수: $(p + 1)$

모델 파라미터: $w[0] \sim w[p], b$

ω : 가중치 weight, 계수 coefficient, θ, β e.g. `model.coef_`

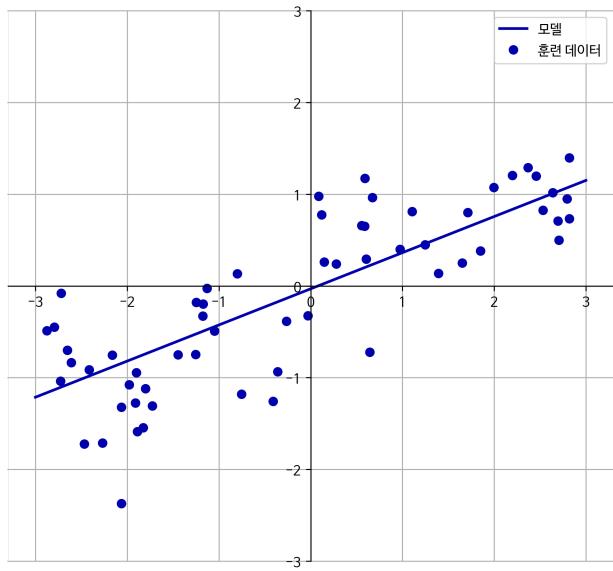
b : 절편 intercept, 편향 bias e.g. `model.intercept_`

하이퍼파라미터 hyperparameter: 학습되지 않고 직접 설정해 주어야 함(매개변수)

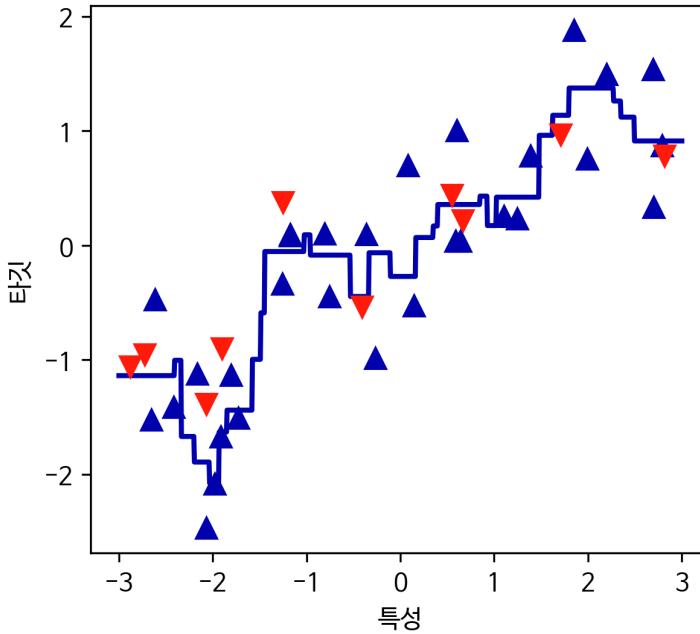
e.g. KNeighborsRegressor의 `n_neighbors`

wave 데이터셋으로 비교

선형 모델



k-최근접 이웃



최소제곱법 OLS, ordinary least squares

평균제곱오차 mean square error $\frac{1}{n} \sum_{i=0}^n (y_i - \hat{y}_i)^2$ 를 최소화

LinearRegression: 정규방정식 normal equation $\hat{\beta} = (X^T X)^{-1} X^T y$ 을 사용하여 w, b를 구함

```
from sklearn.linear_model import LinearRegression
X, y = mglearn.datasets.make_wave(n_samples=60)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

lr = LinearRegression().fit(X_train, y_train)           ← 모델 객체 생성 & 학습

print("lr.coef_: {}".format(lr.coef_))
print("lr.intercept_: {}".format(lr.intercept_))

lr.coef_: [ 0.394]
lr.intercept_: -0.031804343026759746

print("훈련 세트 점수: {:.2f}".format(lr.score(X_train, y_train)))
print("테스트 세트 점수: {:.2f}".format(lr.score(X_test, y_test)))           ← 모델 평가
```

훈련 세트와 테스트 세트로 분리

모델 객체 생성 & 학습

모델 평가

훈련 세트 점수: 0.67
테스트 세트 점수: 0.66

최소제곱법 OLS, ordinary least squares

보스턴 주택 가격 데이터셋, 506개 샘플, 104개 특성

```
In [30]: X, y = mglearn.datasets.load_extended_boston()  
  
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)  
lr = LinearRegression().fit(X_train, y_train)
```

```
In [31]: print("훈련 세트 점수: {:.2f}".format(lr.score(X_train, y_train)))  
print("테스트 세트 점수: {:.2f}".format(lr.score(X_test, y_test)))
```

훈련 세트 점수: 0.95
테스트 세트 점수: 0.61

특성이 많아 가중치가 풍부해져 과대적합 됨

릿지 ridge

선형 모델 + 가중치 최소화

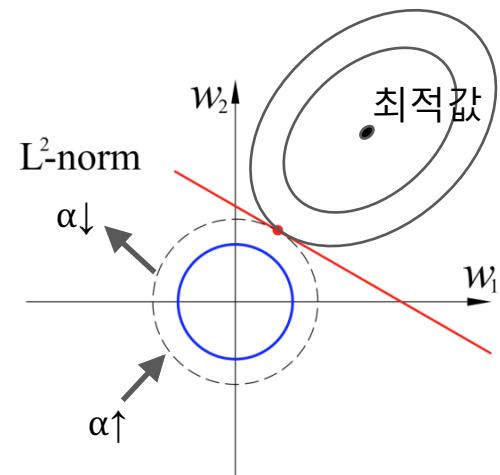
L2 규제 regularization : L2 노름^{norm}의 제곱 $\|w\|_2^2 = \sum_{j=1}^m w_j^2$

비용 함수 cost function : $MSE + \alpha \sum_{j=1}^m w_j^2$

α 가 크면 페널티가 커져 w_j 가 더 작아져야 함

w_j 가 0에 가깝게 되지만 0이 되지는 않음

페널티
penalty



Ridge

In [32]:

```
from sklearn.linear_model import Ridge
ridge = Ridge(alpha=1.0).fit(X_train, y_train)
print("훈련 세트 점수: {:.2f}".format(ridge.score(X_train, y_train)))
print("테스트 세트 점수: {:.2f}".format(ridge.score(X_test, y_test)))
```

훈련 세트 점수: 0.89
테스트 세트 점수: 0.75

alpha=1.0

(가중치가 규제되어) 과대적합이 줄고
테스트 세트 점수가 상승됨

In [33]:

```
ridge10 = Ridge(alpha=10).fit(X_train, y_train)
print("훈련 세트 점수: {:.2f}".format(ridge10.score(X_train, y_train)))
print("테스트 세트 점수: {:.2f}".format(ridge10.score(X_test, y_test)))
```

훈련 세트 점수: 0.79
테스트 세트 점수: 0.64

제약이 너무 커짐
과소적합

In [34]:

```
ridge01 = Ridge(alpha=0.1).fit(X_train, y_train)
print("훈련 세트 점수: {:.2f}".format(ridge01.score(X_train, y_train)))
print("테스트 세트 점수: {:.2f}".format(ridge01.score(X_test, y_test)))
```

훈련 세트 점수: 0.93
테스트 세트 점수: 0.77

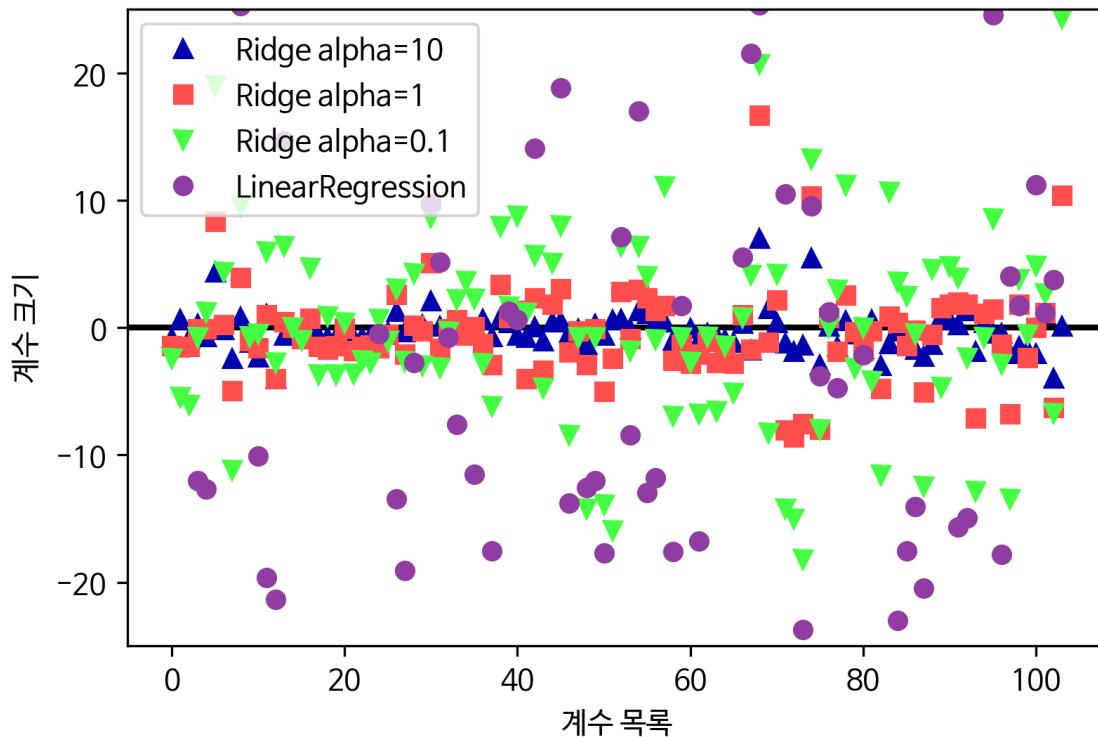
alpha=0.00001

X, y = mglearn.dataset
X_train, X_test, y_train
lr = LinearRegression()

print("훈련 세트 점수: {:.2f}")
print("테스트 세트 점수: {:.2f}")

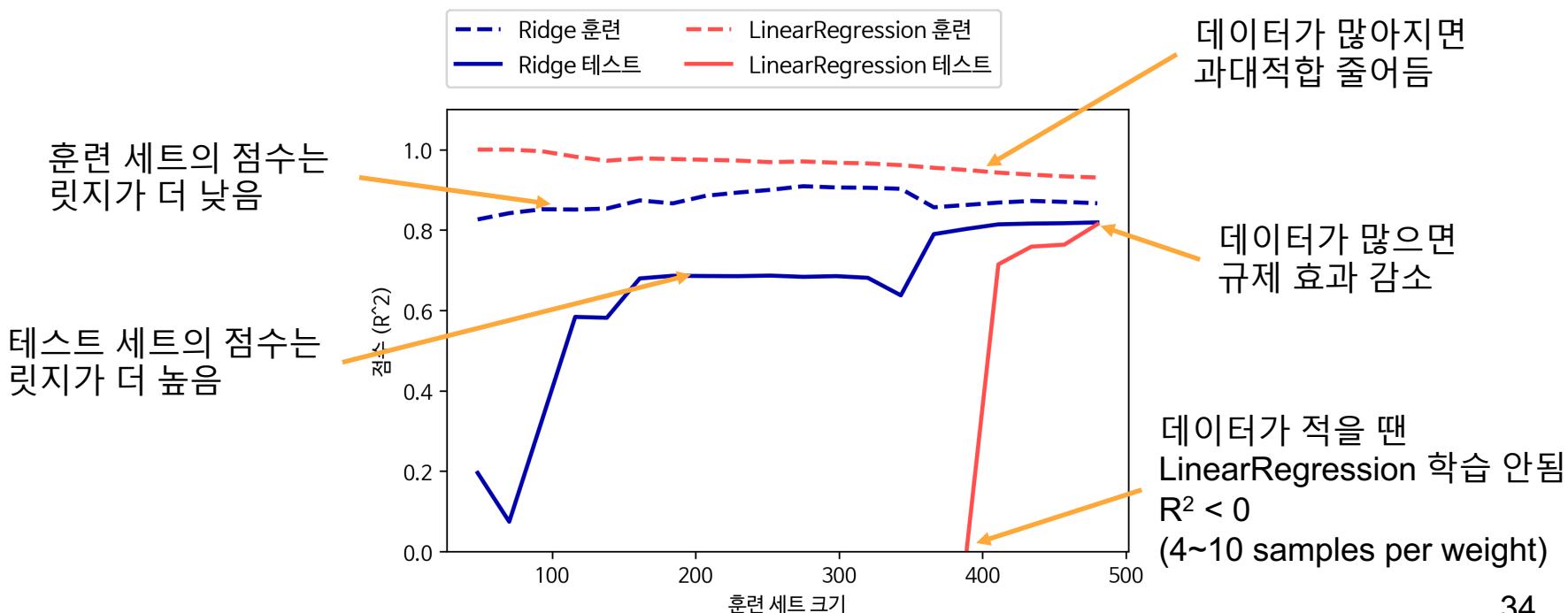
훈련 세트 점수: 0.95
테스트 세트 점수: 0.61

Ridge.coef_



규제와 훈련 데이터의 관계

학습곡선 : LinearRegression vs Ridge(alpha=1)



라쏘 Lasso

선형 모델 + 가중치 최소화

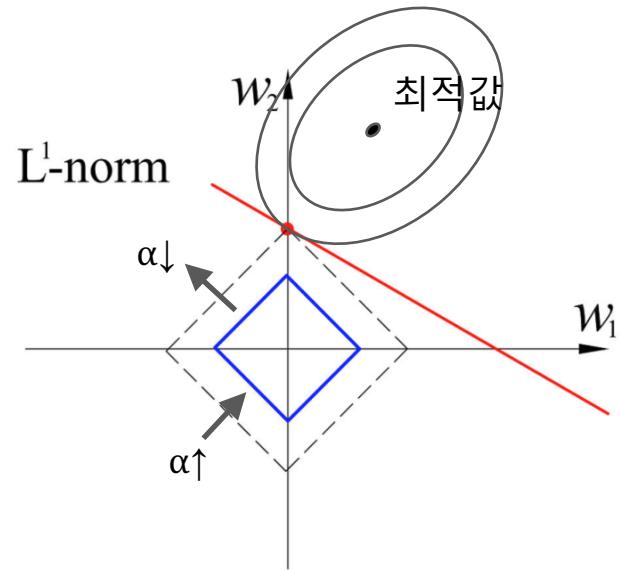
L1 규제 : L1 노름 $\|w\|_1 = \sum_{j=1}^m |w_j|$

비용 함수^{cost function} : $MSE + \alpha \sum_{j=1}^m |w_j|$

α 가 크면 페널티가 커져 w_j 가 더 작아져야 함

w_j 가 0이 될 수 있음

특성 선택의 효과

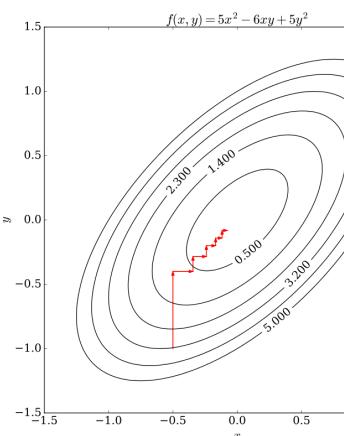


Lasso

alpha=1.0, max_iter=1000

```
In [37]: from sklearn.linear_model import Lasso  
  
lasso = Lasso(alpha=1.0).fit(X_train, y_train)  
print("훈련 세트 점수: {:.2f}".format(lasso.score(X_train, y_train)))  
print("테스트 세트 점수: {:.2f}".format(lasso.score(X_test, y_test)))  
print("사용한 특성의 개수: {}".format(np.sum(lasso.coef_ != 0)))
```

훈련 세트 점수: 0.29
테스트 세트 점수: 0.21 ← 과소적합(규제가 너무 큼)
사용한 특성의 개수: 4



```
In [38]: # "max_iter" 기본 값을 증가시키지 않으면 max_iter 값을 늘이라는 경고가 발생합니다  
lasso001 = Lasso(alpha=0.01, max_iter=100000).fit(X_train, y_train)  
print("훈련 세트 점수: {:.2f}".format(lasso001.score(X_train, y_train)))  
print("테스트 세트 점수: {:.2f}".format(lasso001.score(X_test, y_test)))  
print("사용한 특성의 개수: {}".format(np.sum(lasso001.coef_ != 0)))
```

훈련 세트 점수: 0.90
테스트 세트 점수: 0.77
사용한 특성의 개수: 33

```
x, y = mglearn.datasets.load_mammographic_data()  
  
X_train, X_test, y_train = train_test_split(x, y, test_size=0.2, random_state=42)
```

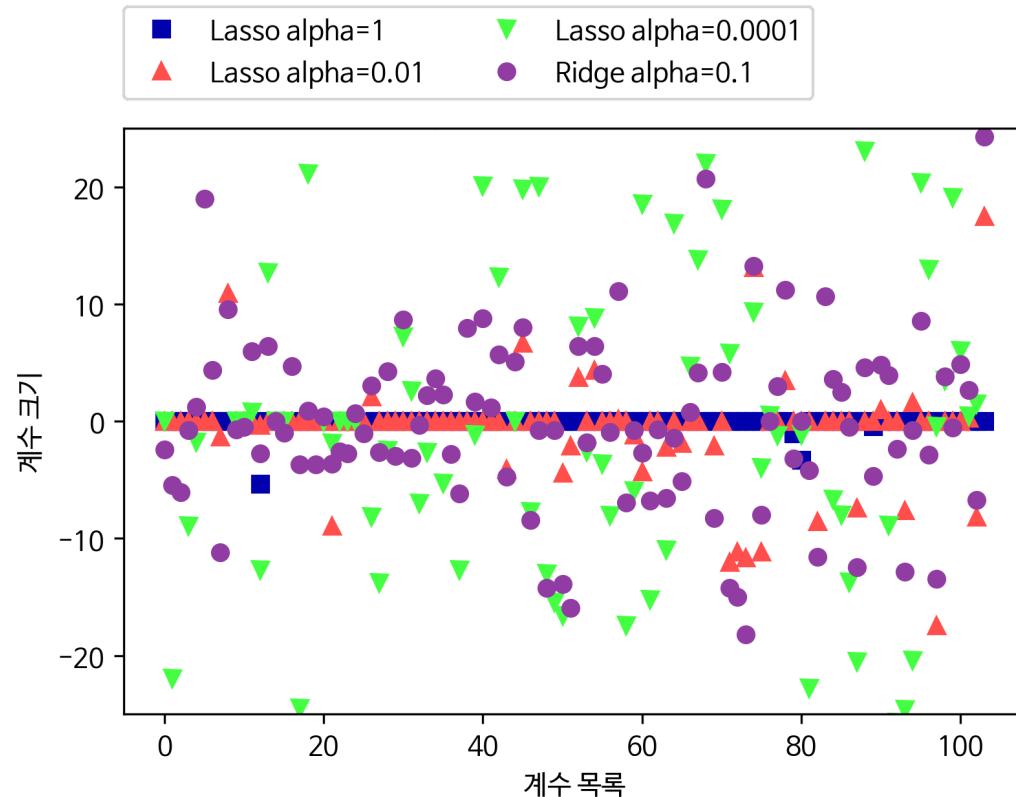
```
In [39]: lasso00001 = Lasso(alpha=0.0001, max_iter=100000).fit(X_train, y_train)  
print("훈련 세트 점수: {:.2f}".format(lasso00001.score(X_train, y_train)))  
print("테스트 세트 점수: {:.2f}".format(lasso00001.score(X_test, y_test)))  
print("사용한 특성의 개수: {}".format(np.sum(lasso00001.coef_ != 0)))
```

훈련 세트 점수: 0.95
테스트 세트 점수: 0.64 ← 규제를 너무 낮추면 LinearRegression과 비슷
사용한 특성의 개수: 94

```
print("훈련 세트 점수: {:.2f}")  
print("테스트 세트 점수: {:.2f}")  
  
훈련 세트 점수: 0.95  
테스트 세트 점수: 0.61
```

좌표하강법 coordinate descent

Lasso.coef_



Ridge vs Lasso

일반적으로 릿지가 라쏘보다 선호됨

L2 페널티가 L1 페널티보다 선호됨 (SGD에서 강한 수렴)

많은 특성 중 일부만 중요하다고 판단되면 라쏘

분석하고 이해하기 쉬운 모델을 원할 때는 라쏘

ElasticNet

릿지와 라쏘 페널티 결합 (R의 glmnet)

alpha, l1_ratio 매개변수로 L1 규제와 L2 규제의 양을 조절

$$\frac{MSE + \alpha \times l1_ratio \sum_{j=1}^m |x_j| + \frac{1}{2} \alpha \times (1 - l1_ratio) \sum_{j=1}^m w_j^2}{l_1 + l_2}$$
$$\alpha = l_1 + l_2 \quad l1_ratio = \frac{l_1}{l_1 + l_2}$$

사실 Lasso는 ElasticNet(l1_ratio=1.0)와 동일

선형 모델 - 분류

이진 분류 binary classification

회귀와 같은 선형 방정식을 사용

$$\hat{y} = w[0] \times x[0] + w[1] \times x[1] + \cdots + w[p] \times x[p] + b > 0$$

0보다 크면 양성 클래스(+1), 0보다 작으면 음성 클래스(-1)

\hat{y} 이 입력에 대한 결정 경계^{decision boundary}를 나타냄

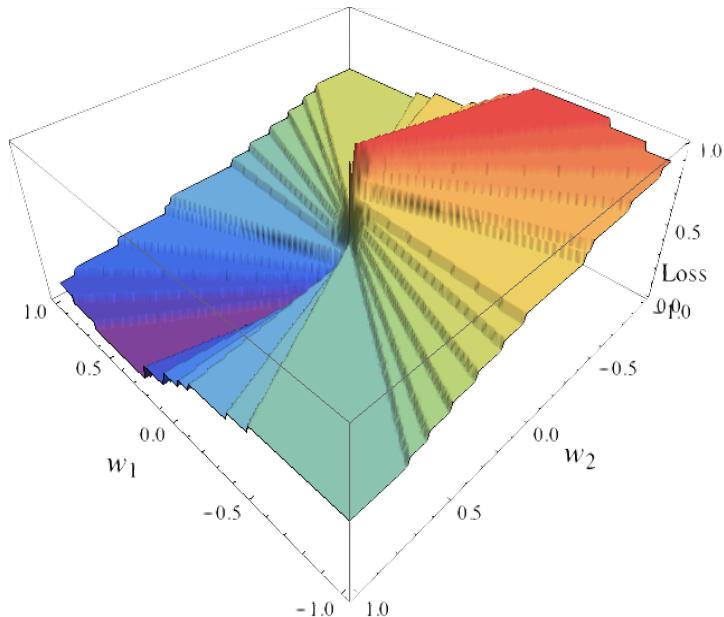
결정 경계는 직선(특성 1개), 면(특성 2개), 초평면(특성 3개)으로 표현됨

선형 모델의 알고리즘 구분

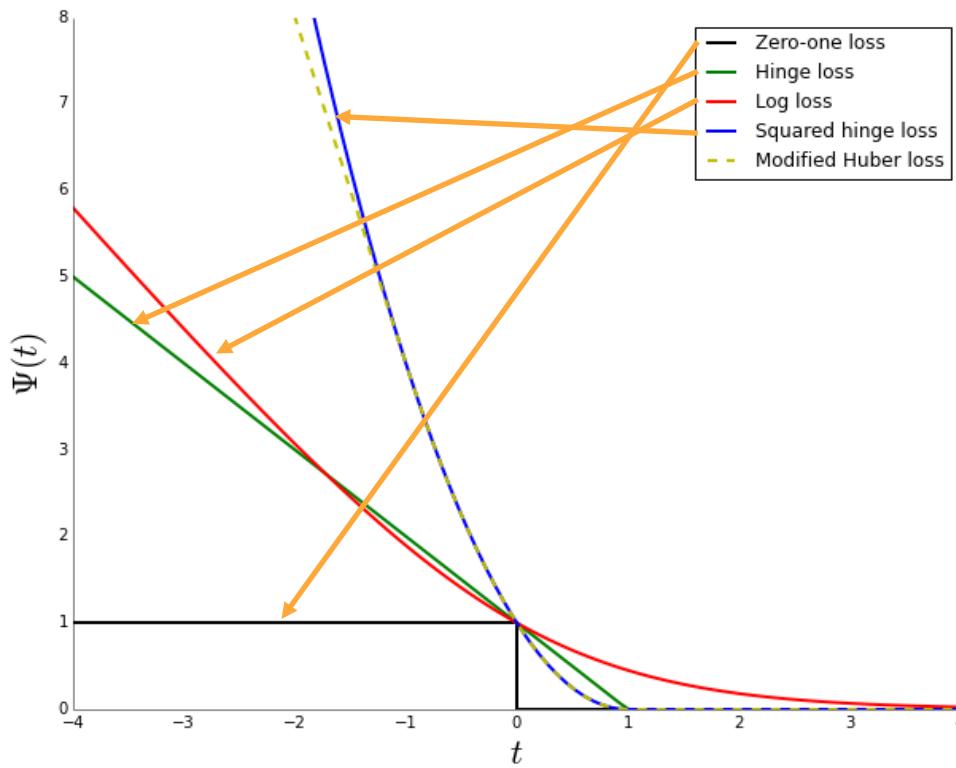
가중치와 절편이 훈련 데이터에 얼마나 잘 맞는지 → 비용 함수 또는 손실 함수
사용할 수 있는 규제의 방법

0-1 손실

계단 함수라 미분 불가능 \rightarrow 최적화에 사용하기 어려움



대리 손실 함수 surrogate loss function

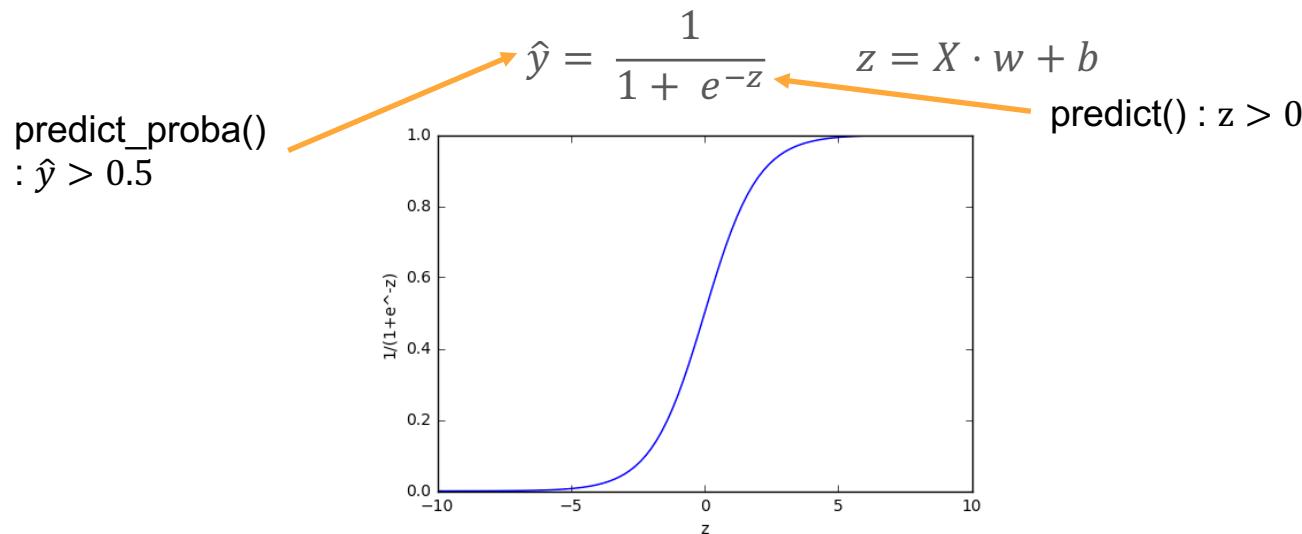


LogisticRegression, LinearSVC

대표적인 선형 분류 알고리즘

로지스틱 회귀 Logistic Regression, 선형 서포트 벡터 머신 Linear SVM

로지스틱(시그모이드 sigmoid) 함수



로지스틱 비용함수

multi_class=multinomial

(cross-entropy loss)

$$-\sum_{i=1}^n y \log(\hat{y}) \quad , \hat{y} = \frac{e^z}{\sum e^z}$$

소프트맥스 softmax 함수

multi_class=ovr

(logistic loss)

$$-\sum_{i=1}^n [y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})] \quad , \hat{y} = \frac{1}{1 + e^{-z}}, y = \{1, 0\}$$

$$-\sum_{i=1}^n \log(e^{-y(w \times x + b)} + 1) \quad , \underline{y = \{1, -1\}}$$

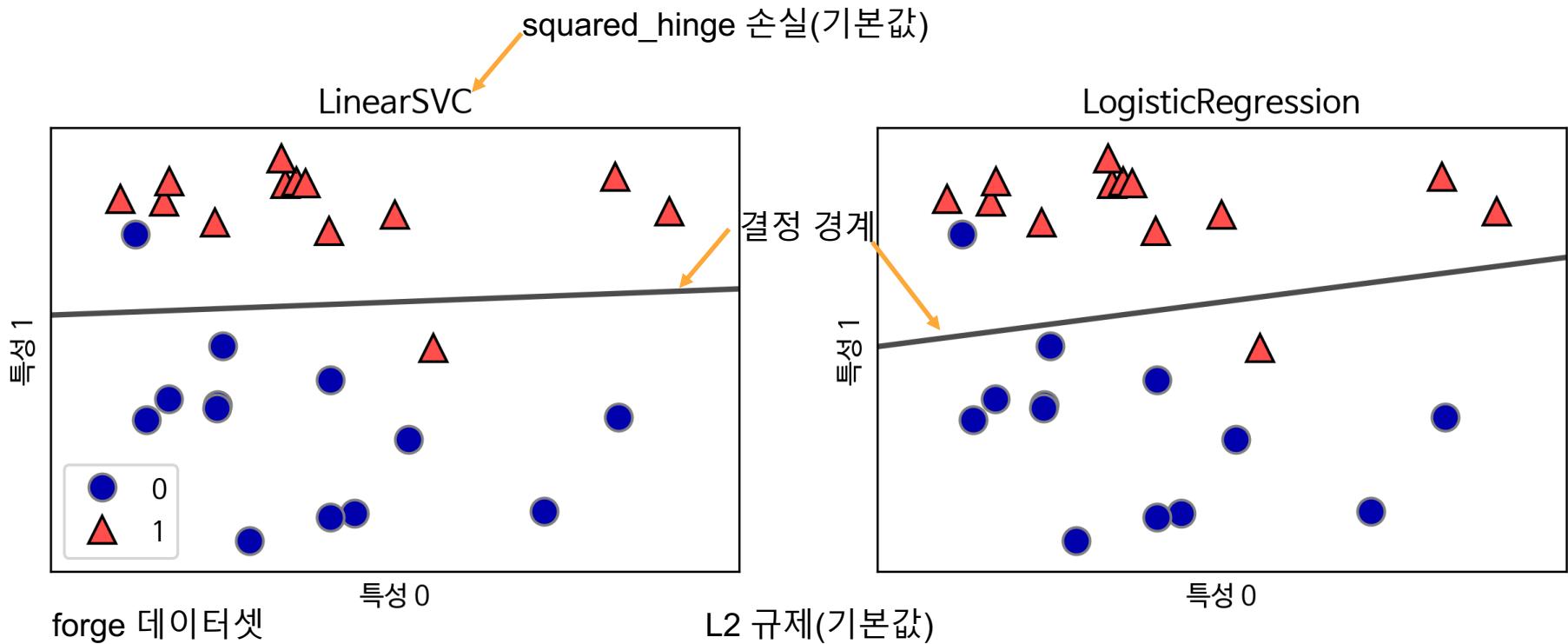
텍스트의 공식과
구현 방식에는
종종 차이가 있음

$$C \cdot L(w) + l2 \text{ or } l1, \quad C = \frac{1}{\alpha}$$

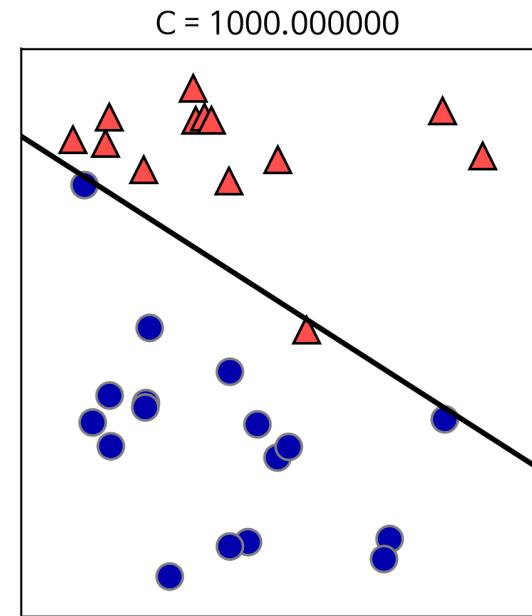
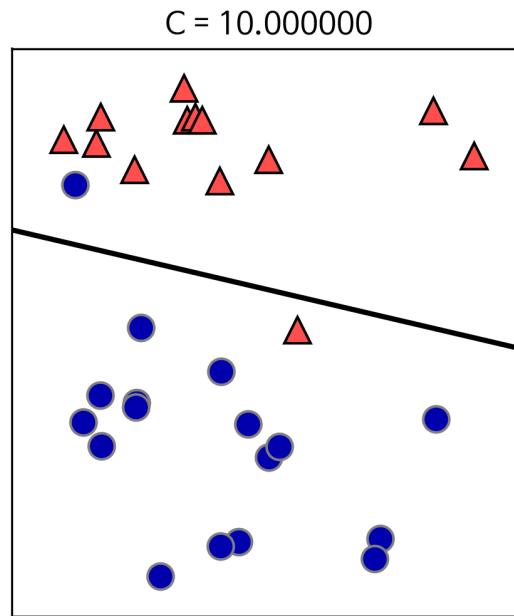
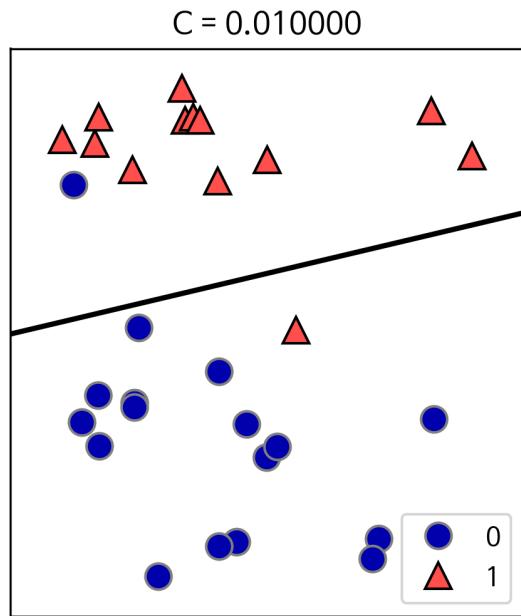
규제 강도

$$\begin{array}{lll} \alpha \uparrow & C \downarrow & \rightarrow w \downarrow \\ \alpha \downarrow & C \uparrow & \rightarrow w \uparrow \end{array}$$

LinearSVC vs LogisticRegression



LinearSVC's C param



과소적합

과대적합

LogisticRegression + cancer

```
In [43]: from sklearn.datasets import load_breast_cancer  
cancer = load_breast_cancer()  
X_train, X_test, y_train, y_test = train_test_split(  
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)  
logreg = LogisticRegression().fit(X_train, y_train)  
print("훈련 세트 점수: {:.3f}".format(logreg.score(X_train, y_train)))  
print("테스트 세트 점수: {:.3f}".format(logreg.score(X_test, y_test)))
```

훈련 세트 점수: 0.953
테스트 세트 점수: 0.958

과소적합(규제가 너무 큼)

```
In [44]: logreg100 = LogisticRegression(C=100).fit(X_train, y_train)  
print("훈련 세트 점수: {:.3f}".format(logreg100.score(X_train, y_train)))  
print("테스트 세트 점수: {:.3f}".format(logreg100.score(X_test, y_test)))
```

훈련 세트 점수: 0.972
테스트 세트 점수: 0.965

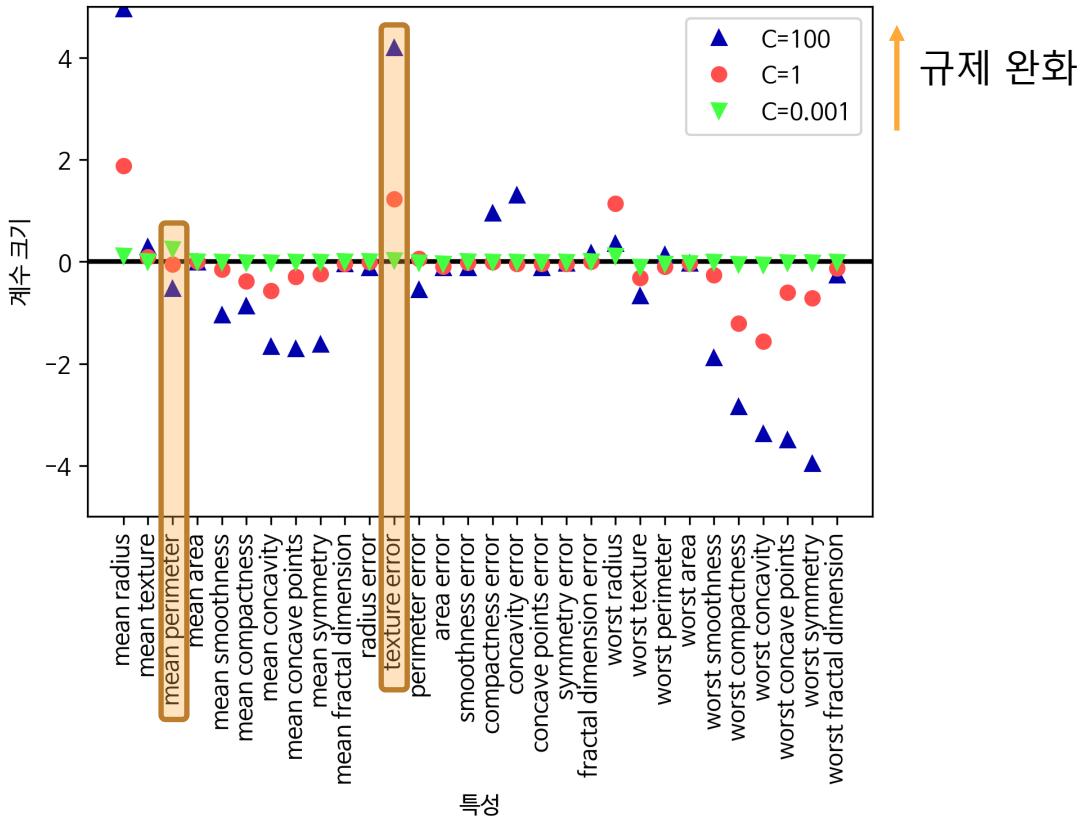
복잡도 증가, 성능향상

```
In [45]: logreg001 = LogisticRegression(C=0.01).fit(X_train, y_train)  
print("훈련 세트 점수: {:.3f}".format(logreg001.score(X_train, y_train)))  
print("테스트 세트 점수: {:.3f}".format(logreg001.score(X_test, y_test)))
```

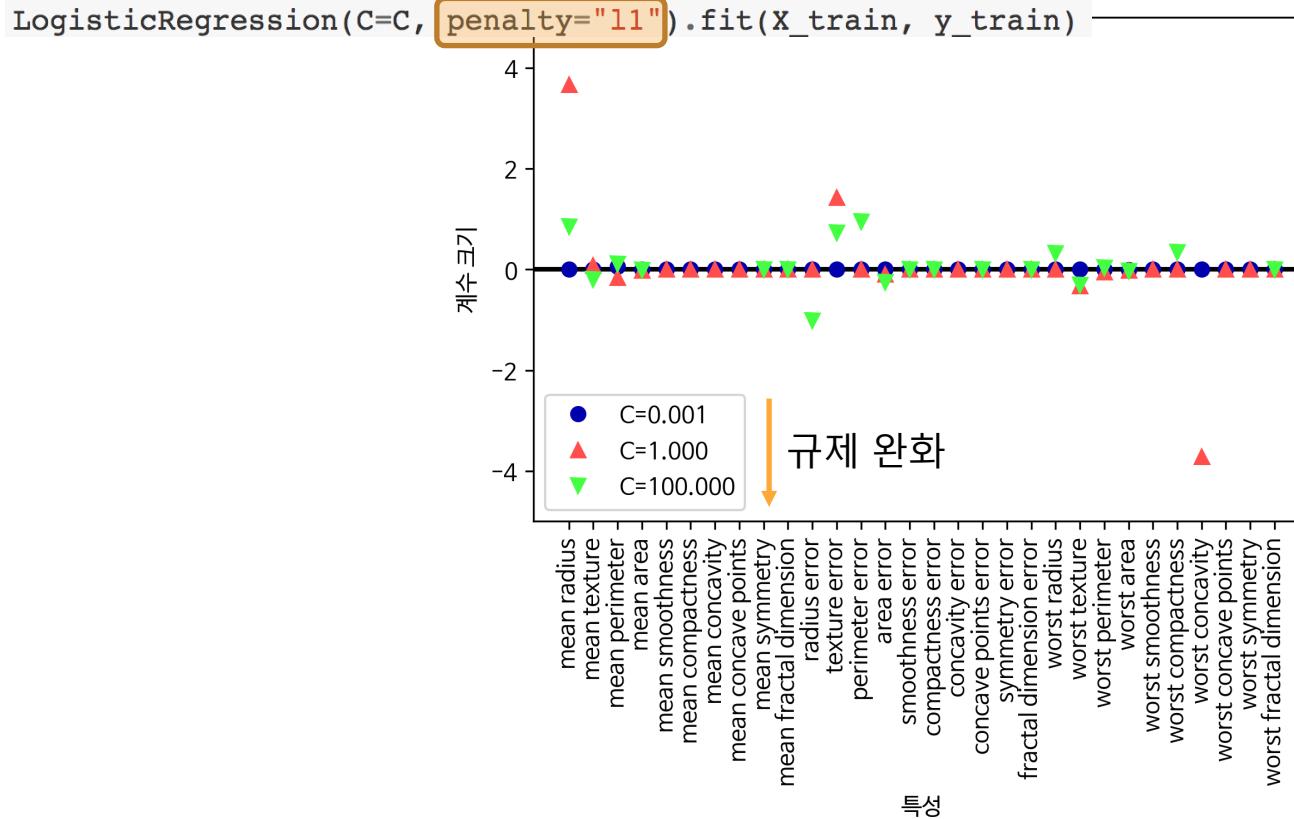
훈련 세트 점수: 0.934
테스트 세트 점수: 0.930

복잡도를 기본값 보다 더 낮추면

LogisticRegression.coef_ (L2)



LogisticRegression.coef_ (L1)



다중 클래스 분류

로지스틱 회귀를 제외하고 대부분 선형 분류 모델은 이진 분류만을 지원

로지스틱 회귀는 소프트맥스 함수를 사용하여 다중 분류 지원

LogisticRegression(multi_class=multinomial)

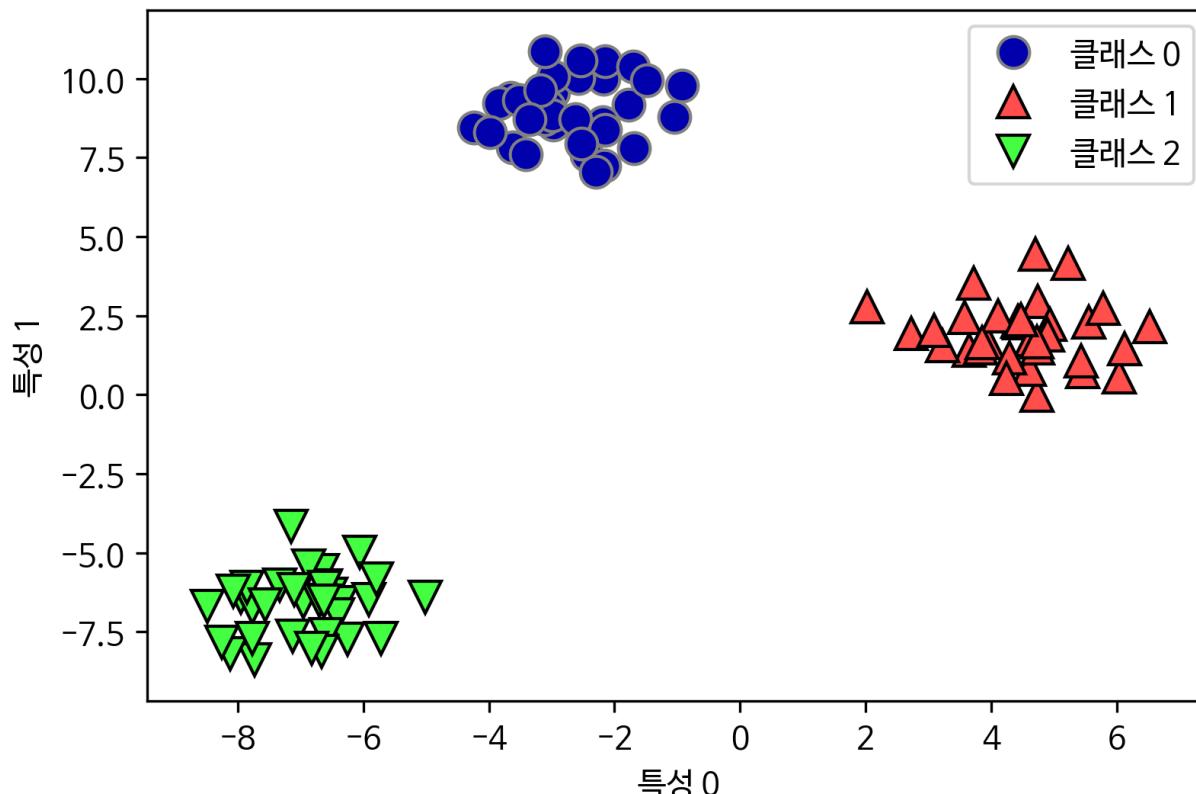
$$-\sum_{i=1}^n y \log(\hat{y}), \hat{y} = \frac{e^z}{\sum e^z}$$

클래스마다 배타적으로 이진 분류 모델을 만드는 일 때 one-vs-rest 방식 사용

이진 분류 모델 중에서 가장 높은 점수의 클래스가 선택

클래스마다 가중치와 절편이 만들어짐

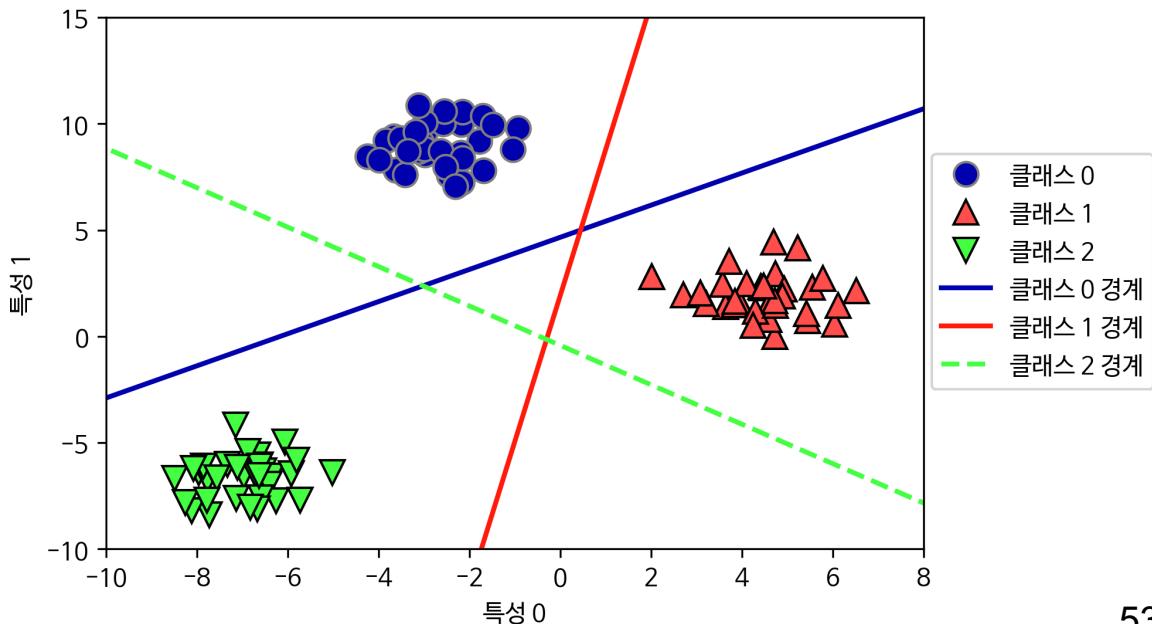
make_blobs dataset



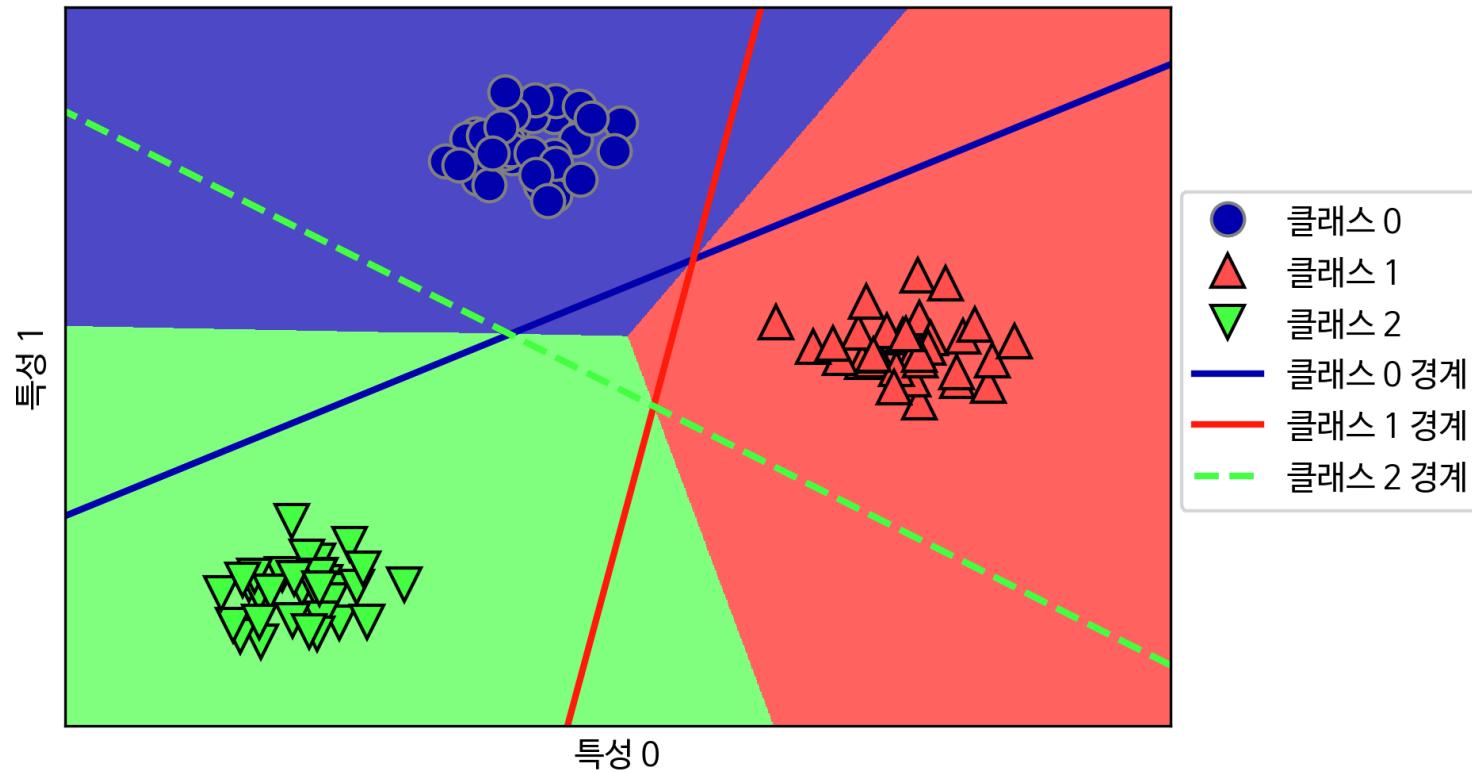
LinearSVC 다중 분류

```
In [49]: linear_svm = LinearSVC().fit(X, y)
print("계수 배열의 크기: ", linear_svm.coef_.shape)
print("절편 배열의 크기: ", linear_svm.intercept_.shape)
```

계수 배열의 크기: (3, 2)
절편 배열의 크기: (3,)



다중 분류 결정 경계



장단점과 매개변수

규제 매개변수: 회귀 모델은 alpha, 분류 모델은 C

보통 규제는 로그 스케일로 조절 (0.1, 1, 10, 100)

일부 특성이 중요하거나 해석을 쉽게하려면 L1 규제, 아니면 기본값 L2

선형 모델의 장점

학습속도와 예측속도 빠름

큰 데이터셋과 희박 데이터셋에 적합

샘플 수(n)보다 특성(m)이 많을 때. ex) m = 10,000, n = 1,000

특성이 부족할 때는 선형 모델 보다는 커널 SVM 등이 효과적

수십, 수백만개라면 LogisticRegression 에 solver='sag' 옵션을 사용(L2만 가능)

더 큰 데이터셋에서는 SGDRegressor, SGDClassifier

$$\begin{array}{lll} \alpha \uparrow & C \downarrow & \rightarrow w \downarrow \\ \alpha \downarrow & C \uparrow & \rightarrow w \uparrow \end{array}$$

메서드 연결

객체 반환 logreg = LogisticRegression()
 logreg = logreg.fit(X_train, y_train)

In [52]: # 한 줄에서 모델의 객체를 생성과 학습을 한번에 실행합니다

```
logreg = LogisticRegression().fit(X_train, y_train)
```

In [53]: logreg = LogisticRegression()

```
y_pred = logreg.fit(X_train, y_train).predict(X_test)
```

In [54]: y_pred = LogisticRegression().fit(X_train, y_train).predict(X_test)

로지스틱 회귀 모델을 다음 코드에서 사용할 수 없음

나이브 베이즈

나이브 베이즈 Naive Bayes 분류기

선형 분류기보다 훈련 속도가 빠르지만 일반화 성능이 조금 떨어짐

특성마다 클래스별 통계를 취합해 파라미터를 학습

GaussianNB : 연속적인 데이터

BernoulliNB : 이진 데이터, 텍스트 데이터

MultinomialNB : 정수 카운트 데이터, 텍스트 데이터

BernoulliNB

```
In [55]: x = np.array([[0, 1, 0, 1],  
[1, 0, 1, 1],  
[0, 0, 0, 1],  
[1, 0, 1, 0]])  
y = np.array([0, 1, 0, 1])
```

특성 카운트:

{0: array([0, 1, 0, 2]), 1: array([2, 0, 2, 1])}

MultinomialNB : 클래스별 특성의 평균

GaussianNB : 클래스별 특성의 표준편차와 평균

장단점과 매개변수

alpha 매개변수로 모델 복잡도 조절

가상의 양수 데이터를 alpha 개수만큼 추가하여 통계를 완만하게 만듬

alpha가 크면 모델의 복잡도가 낮아지지만 성능의 변화는 크지 않음

GaussianNB는 고차원 데이터셋에 BernoulliNB, MultinomialNB는 텍스트와 같은 희소 데이터를 카운트하는데 사용

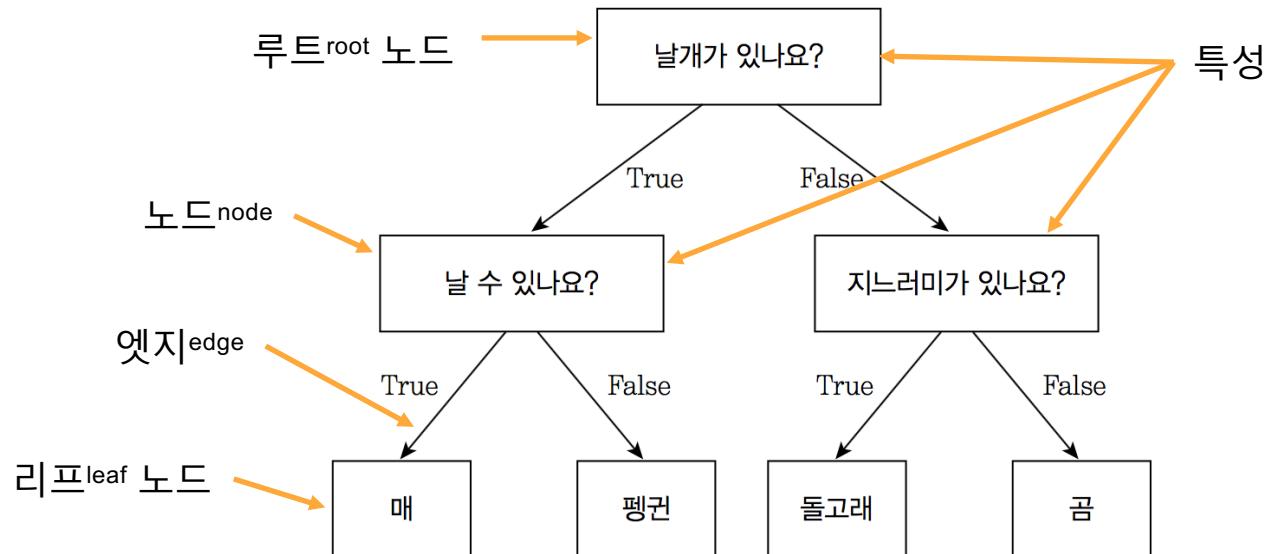
훈련과 예측 속도가 빠르고 과정을 이해하기 쉬움

희소한 고차원 데이터셋에 잘 작동하고 매개변수에 민감하지 않음

결정 트리

결정 트리 | decision tree

분류와 회귀에 사용 가능

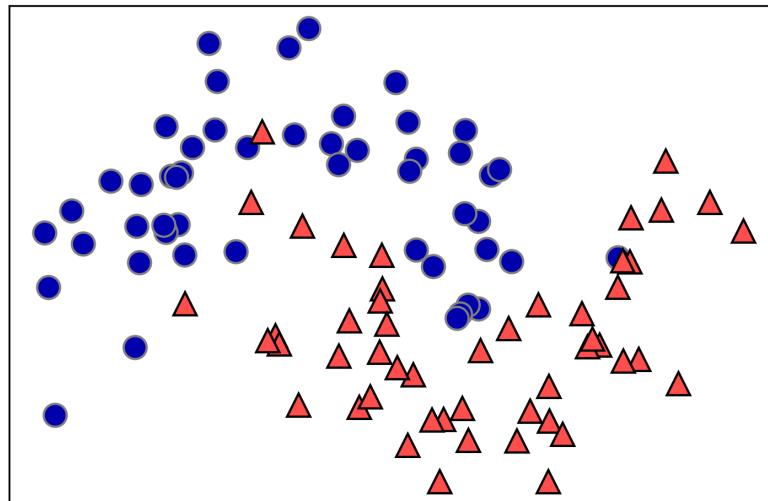


결정 트리 만들기

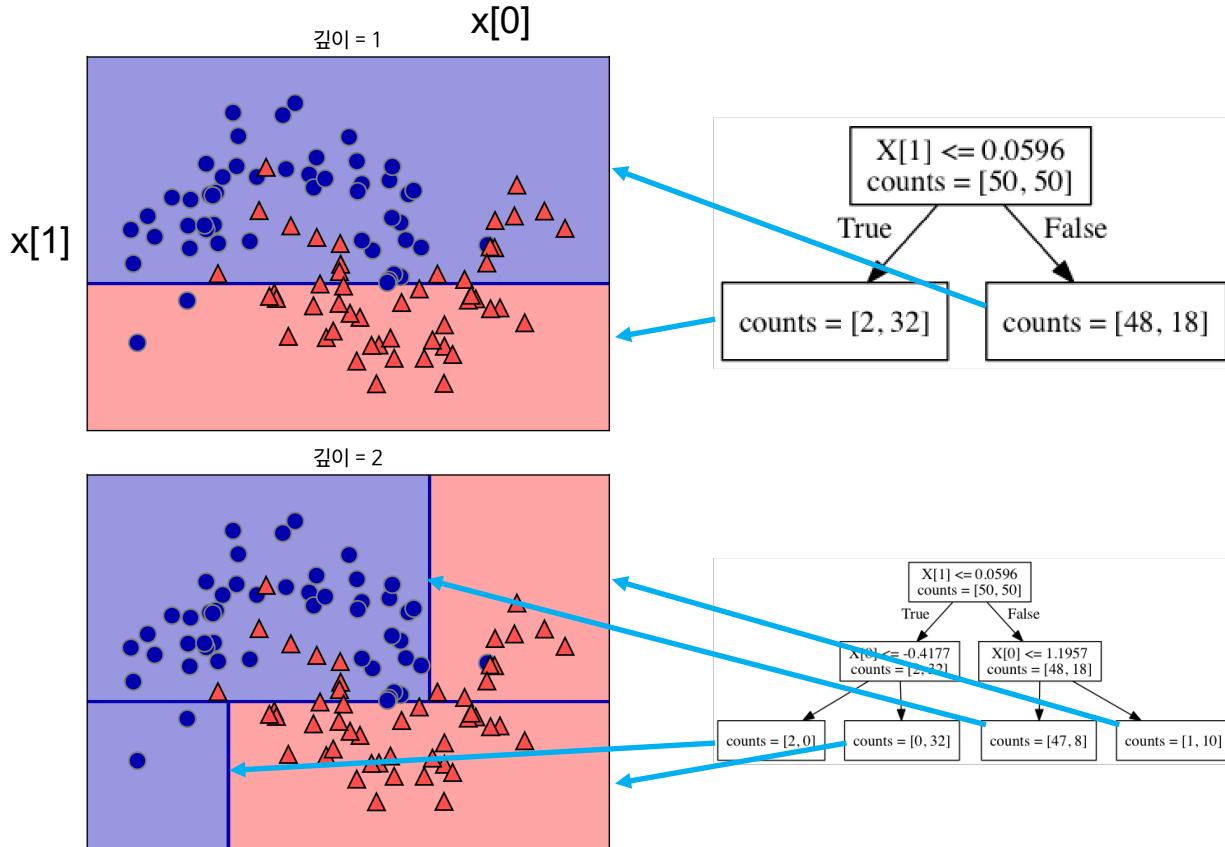
two_moons 데이터셋

노드의 yes/no 질문(테스트)을 학습

연속적인 데이터셋에서는 “특성 i 가 a 보다 큰가?”와 같은 형태

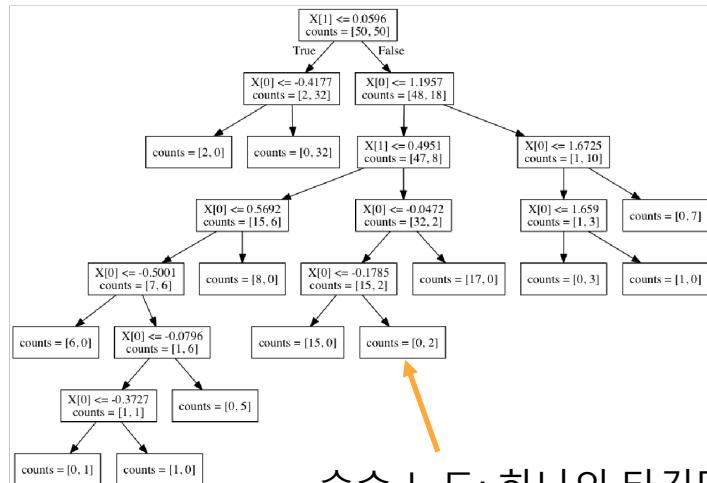
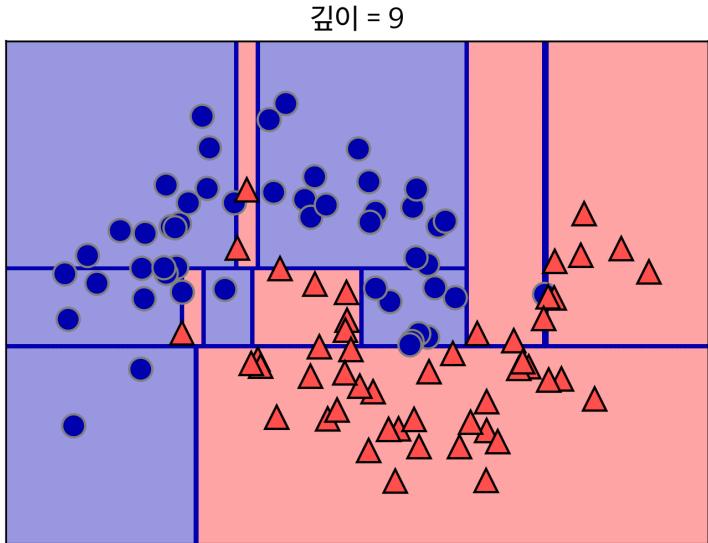


결정 트리 훈련



결정 트리 예측

테스트는 하나의 특성을 사용하므로 축에 평행하게 분리됨



순수 노드: 하나의 타깃만 가짐
회귀일 땐 리프 노드의 평균

복잡도 제어

모든 리프가 순수 노드가 될 때 까지 진행하여 복잡해지고 과대적합됨

순수 노드로만 이루어진 트리는 훈련 데이터를 100% 정확히 맞춤(일반화 낮음)

사전 가지치기|^{pre-prunning}: 트리 생성을 미리 중단

- 트리의 최대 깊이 제한, 리프의 최대 개수 제한
- 분할 가능한 포인트의 최소 개수 지정

사후 가지치기|^{post-prunning}: 트리를 만든 후 노드를 삭제하거나 병합

DecisionTreeRegressor, DecisionTreeClassifier는 사전 가지치기만 지원

복잡도 제어 효과

```
In [59]: from sklearn.tree import DecisionTreeClassifier  
  
cancer = load_breast_cancer()  
X_train, X_test, y_train, y_test = train_test_split(  
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)  
tree = DecisionTreeClassifier(random_state=0)  
tree.fit(X_train, y_train)  
print("훈련 세트 정확도: {:.3f}".format(tree.score(X_train, y_train)))  
print("테스트 세트 정확도: {:.3f}".format(tree.score(X_test, y_test)))
```

결정트리 생성 및 훈련

훈련 세트 정확도: 1.000
테스트 세트 정확도: 0.937

과대 적합

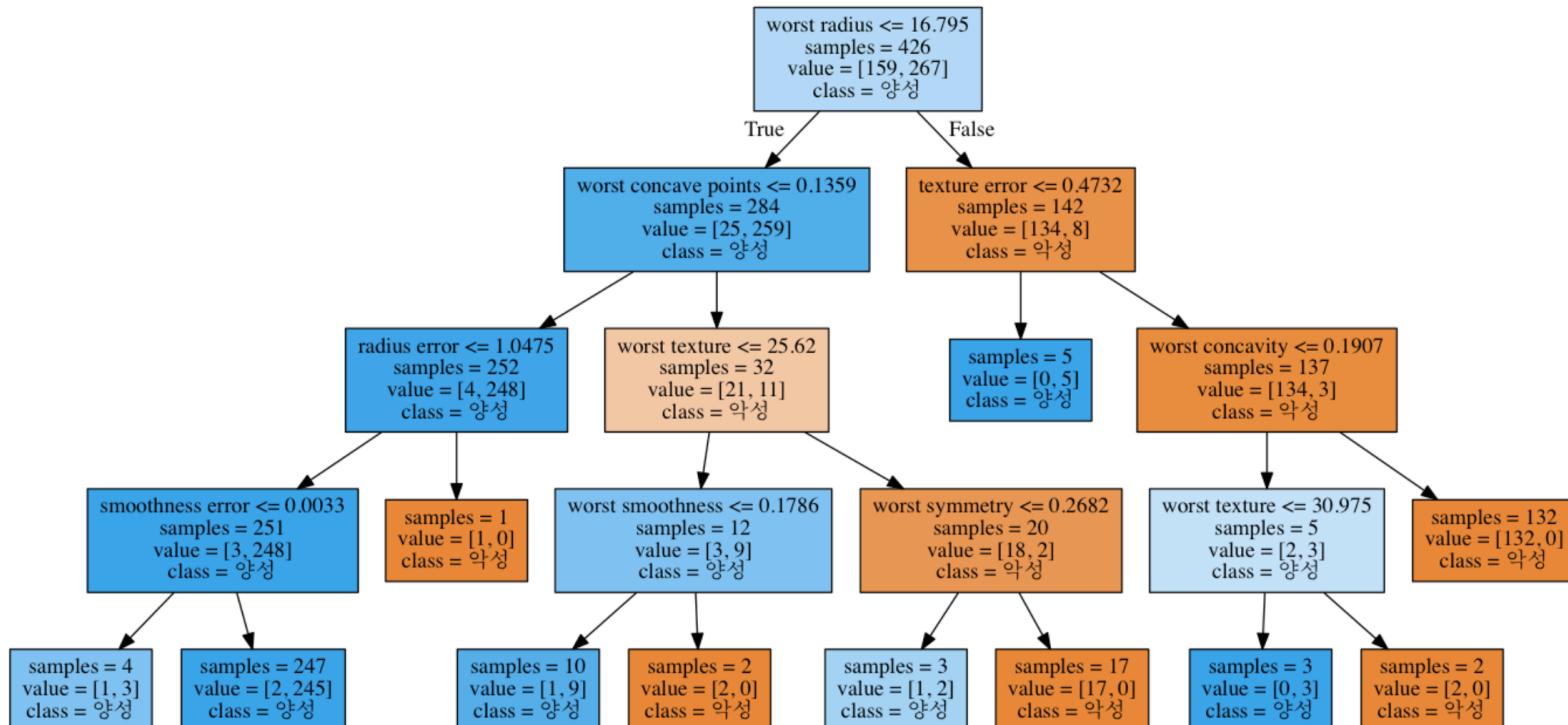
트리의 최대 깊이 4로 제한

```
In [60]: tree = DecisionTreeClassifier(max_depth=4, random_state=0)  
tree.fit(X_train, y_train)
```

```
print("훈련 세트 정확도: {:.3f}".format(tree.score(X_train, y_train)))  
print("테스트 세트 정확도: {:.3f}".format(tree.score(X_test, y_test)))
```

훈련 세트 정확도: 0.988
테스트 세트 정확도: 0.951

결정 트리 분석



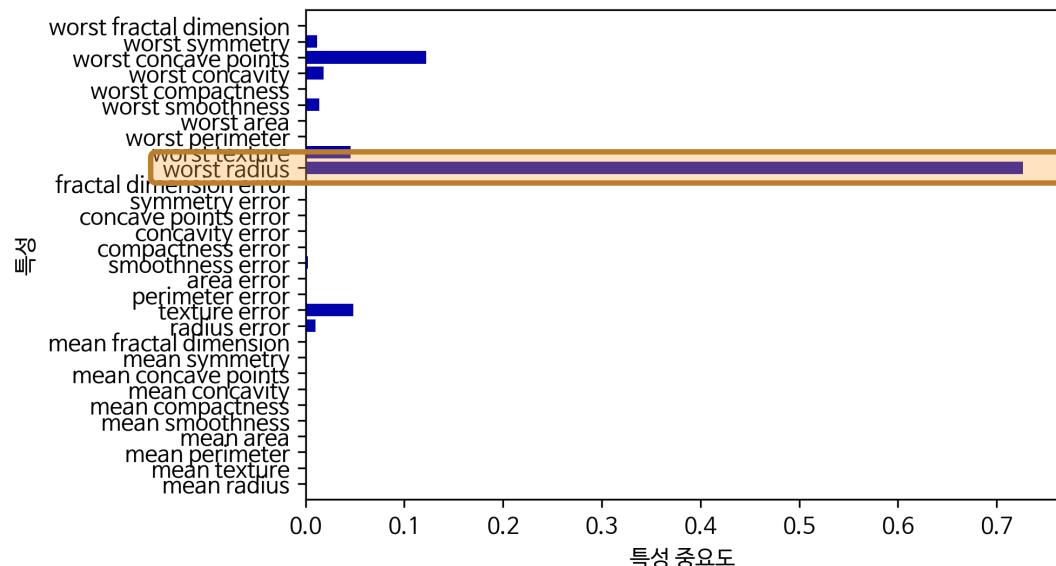
특성 중요도 feature importance

0(예측에 사용되지 않음)과 1(예측과 완벽하게 동일) 사이의 값, 전체 합은 1

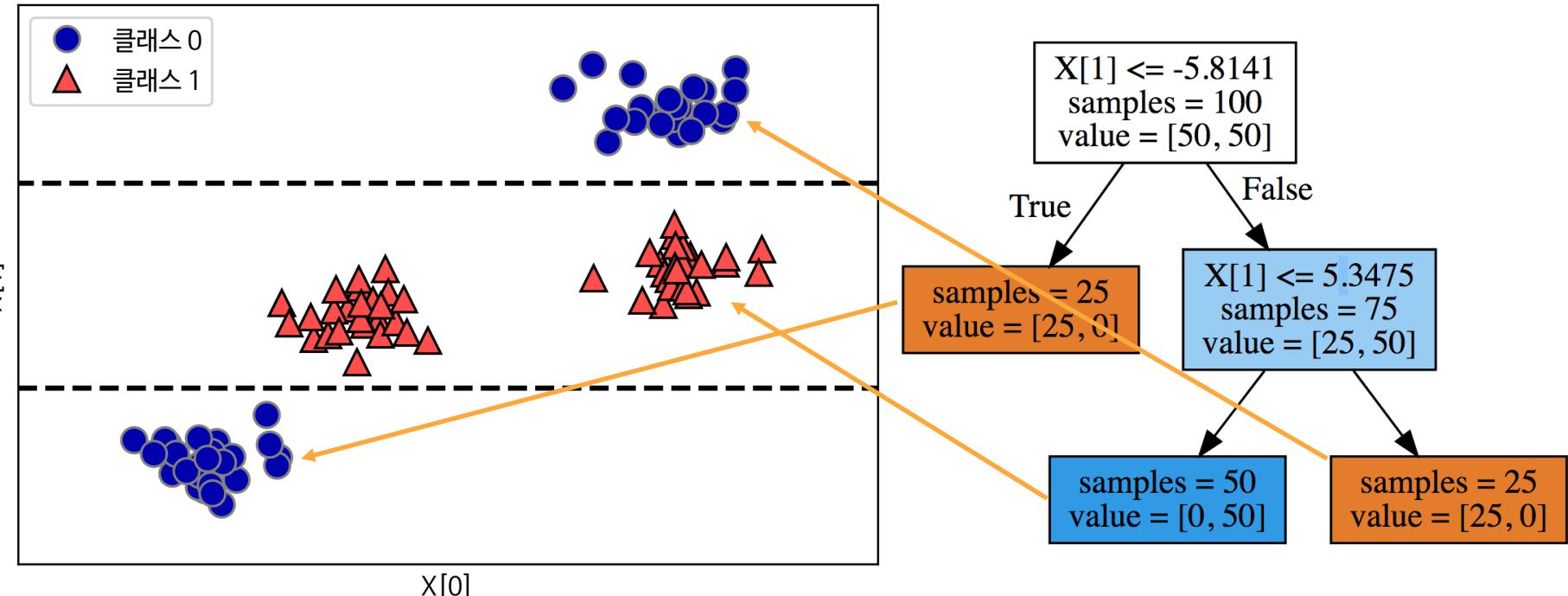
```
In [63]: print("특성 중요도:\n{}".format(tree.feature_importances_))
```

특성 중요도:

```
[ 0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.         0.01
  0.048      0.         0.         0.002     0.         0.         0.         0.         0.         0.         0.727
  0.046      0.         0.         0.014     0.         0.018     0.122     0.012     0.         ]
```



특성과 클래스 사이의 관계



결정 트리 - 회귀

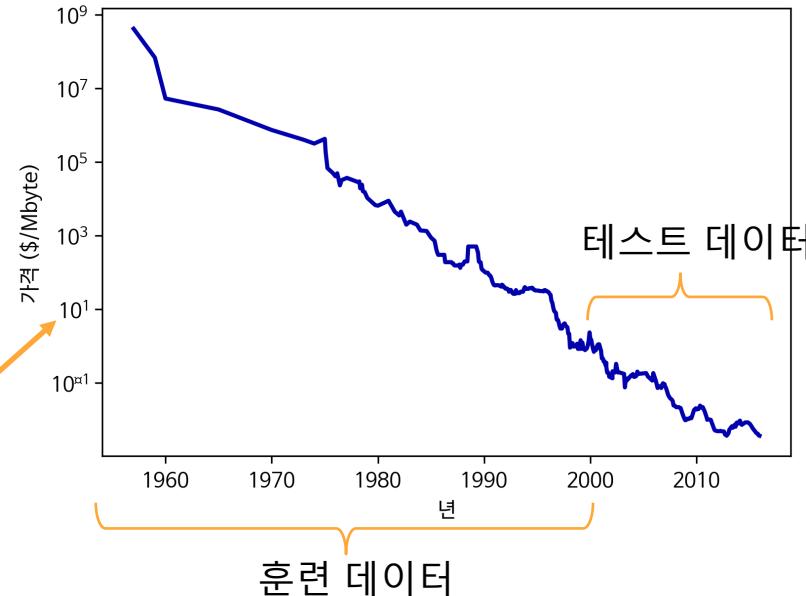
DecisionTreeRegressor

훈련 데이터 범위 밖을 예측하는 외삽^{extrapolation}이 불가능

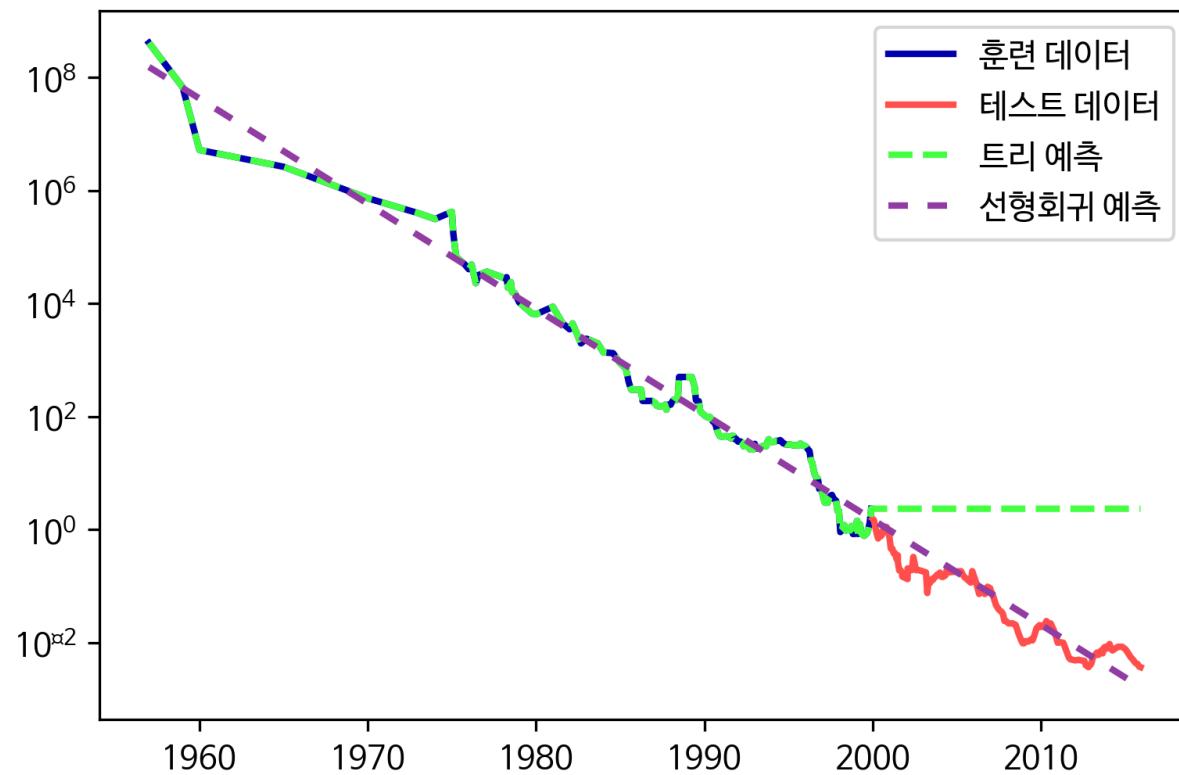
스케일에 영향 받지 않음

컴퓨터 메모리 가격 데이터셋

로그 스케일



LinearRegression vs DecisionTreeRegressor



장단점과 매개변수

모델 복잡도 제어(사전 가지치기) 매개변수

max_depth : 트리의 최대 깊이

max_leaf_nodes : 리프 노드의 최대 개수

min_samples_leaf : 리프 노드가 되기 위한 최소 샘플 개수

min_samples_split : 노드가 분할하기 위한 최소 샘플 개수

작은 트리일 때 시각화가 좋아 설명하기 쉬움

특성이 각각 처리되므로 데이터 스케일에 구애 받지 않음

정규화나 표준화 같은 전처리 과정이 필요 없음

바이너리 특성이나 연속적인 특성이 혼합되어 있어도 가능

단점: 과대적합 가능성 높음

랜덤 포레스트

랜덤 포레스트 random forest

여러 결정 트리를 묶어 과대적합을 회피할 수 있는 앙상블 방법 중 하나임

예측을 잘 하는(과대적합된) 트리를 평균내어 과대적합을 줄임

랜덤 포雷스트는 트리 생성시 무작위성을 주입함

트리 생성시 데이터 중 일부를 무작위로 선택
노드 분할 시 무작위로 후보 특성을 선택

RandomForestClassifier, RandomForestRegressor의 n_estimators 매개변수로
트리의 개수를 지정(기본값 10)

모든 트리의 예측을 만든 후, 회귀: 예측 값의 평균, 분류: 예측 확률의 평균

무작위성

부트스트랩 샘플 bootstrap sample

n 개의 훈련 데이터에서 무작위로 추출해 n 개의 데이터셋을 만듦

중복 추출이 가능하며 하나의 부트스트랩 샘플엔 대략 1/3 정도 샘플이 누락됨

['a', 'b', 'c', 'd'] → ['b', 'd', 'd', 'c'], ['d', 'a', 'd', 'a']

노드 분할에 사용할 후보 특성을 랜덤하게 선택(max_features 매개변수)

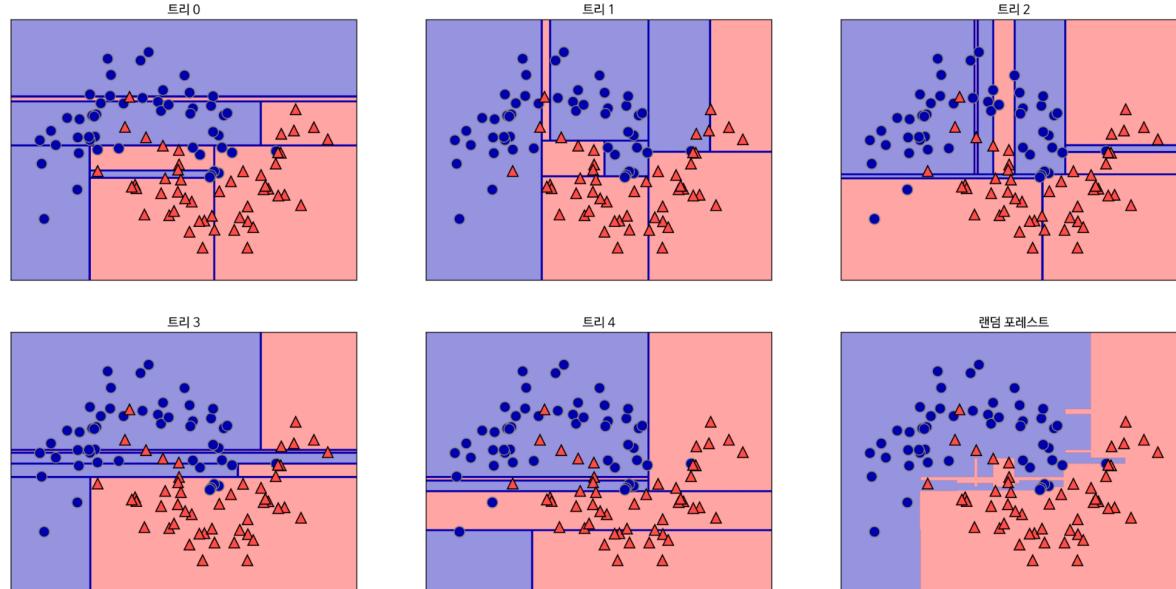
→ 랜덤 포레스트의 트리가 모두 다르게 생성됨

max_features == n_features : 모든 특성을 사용. 비슷한 트리들이 생성.

max_features == 1 : 노드 분할을 무작위로 수행. 많이 다르고 깊은 트리 생성.

랜덤 포레스트 분석

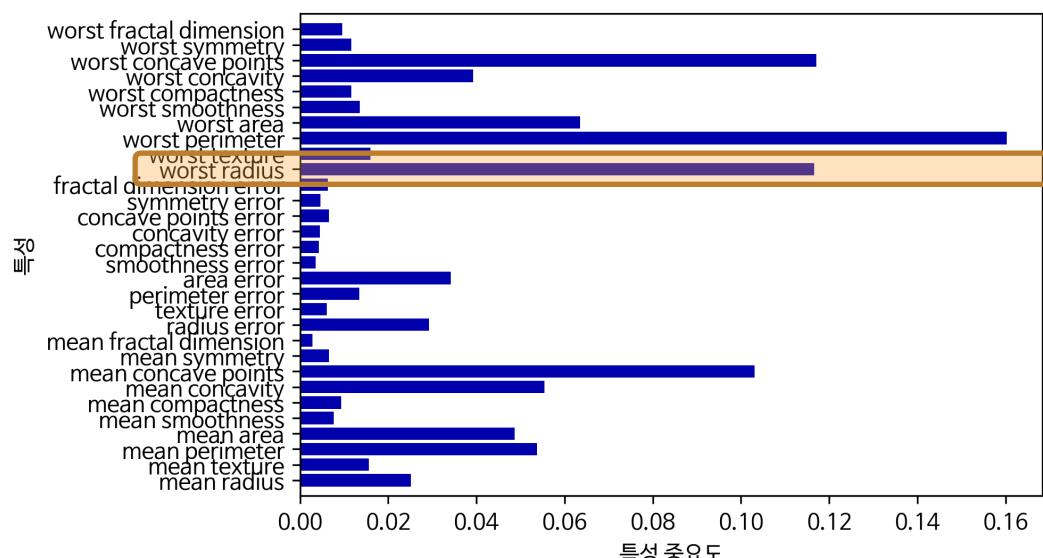
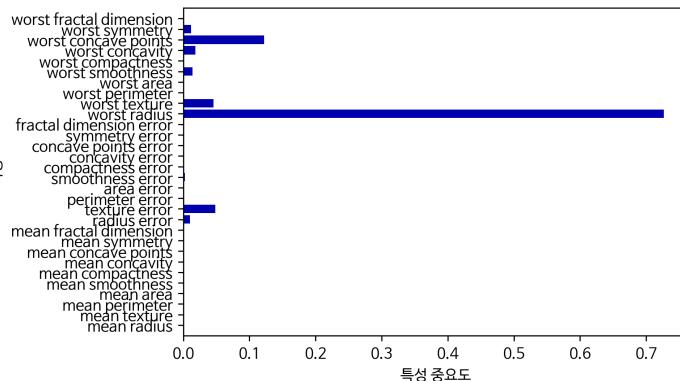
```
In [69]: from sklearn.ensemble import RandomForestClassifier  
from sklearn.datasets import make_moons  
  
X, y = make_moons(n_samples=100, noise=0.25, random_state=3)  
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, random_state=42)  
  
forest = RandomForestClassifier(n_estimators=5, random_state=2)  
forest.fit(X_train, y_train)
```



cancer 데이터셋에 적용

```
In [71]: X_train, X_test, y_train, y_test = train_test_split(  
    cancer.data, cancer.target, random_state=0)  
forest = RandomForestClassifier(n_estimators=100, random_state=0)  
forest.fit(X_train, y_train)  
  
print("훈련 세트 정확도: {:.3f}".format(forest.score(X_train, y_train)))  
print("테스트 세트 정확도: {:.3f}".format(forest.score(X_test, y_test)))
```

훈련 세트 정확도: 1.000
테스트 세트 정확도: 0.972



장단점과 매개변수

장점

회귀와 분류에서 가장 널리 사용되는 알고리즘

뛰어난 성능, 매개변수 튜닝 부담 적음, 데이터 스케일 불필요

큰 데이터셋 적용 가능, 여러 CPU 코어에 병렬화 가능(`n_jobs`: 기본값 1, 최대 -1)

단점

많은 트리가 생성되므로 자세한 분석이 어렵고 트리가 깊어지는 경향이 있음

차원이 크고 희소한 데이터에 성능 안좋음(예, 텍스트 데이터)

선형 모델 보다 메모리 사용량이 많고 훈련과 예측이 느림

매개변수

`n_estimators`, `max_features`

사전가지치기: `max_depth`, `max_leaf_nodes`, `min_samples_leaf`, `min_samples_split`

`max_features` 기본값, 회귀: `n_features`, 분류: `sqrt(n_features)`

그래디언트 부스팅

그래디언트 부스팅 Gradient Boosting 회귀

결정 트리(DecisionTreeRegressor)를 기반으로 하는 또 다른 앙상블 알고리즘

회귀와 분류에 모두 사용 가능

랜덤 포레스트와는 달리 무작위성 대신 사전 가지치기를 강하게 적용

다섯 이하의 얇은 트리를 사용하여 이전 트리의 오차를 보완하도록 다음 트리 생성

회귀 : 최소제곱오차 손실함수, 분류 : 로지스틱 손실함수

경사 하강법 gradient descent 사용(learning_rate 매개변수 중요, 기본값 0.1)

머신러닝 경연대회(캐글)에서 많이 사용됨

랜덤 포레스트 보다 매개변수에 조금 더 민감, 조금 더 높은 성능

GradientBoostingClassifier

```
In [73]: from sklearn.ensemble import GradientBoostingClassifier
```

```
x_train, x_test, y_train, y_test = train_test_split(  
    cancer.data, cancer.target, random_state=0)
```

```
gbrt = GradientBoostingClassifier(random_state=0)  
gbrt.fit(x_train, y_train)
```

```
print("훈련 세트 정확도: {:.3f}".format(gbrt.score(x_train, y_train)))  
print("테스트 세트 정확도: {:.3f}".format(gbrt.score(x_test, y_test)))
```

훈련 세트 정확도: 1.000

테스트 세트 정확도: 0.958

과대적합

n_estimators=100, max_depth=3

```
In [74]: gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)  
gbrt.fit(x_train, y_train)
```

```
print("훈련 세트 정확도: {:.3f}".format(gbrt.score(x_train, y_train)))  
print("테스트 세트 정확도: {:.3f}".format(gbrt.score(x_test, y_test)))
```

훈련 세트 정확도: 0.991

테스트 세트 정확도: 0.972

트리 깊이 제한

```
In [75]: gbrt = GradientBoostingClassifier(random_state=0, learning_rate=0.01)  
gbrt.fit(x_train, y_train)
```

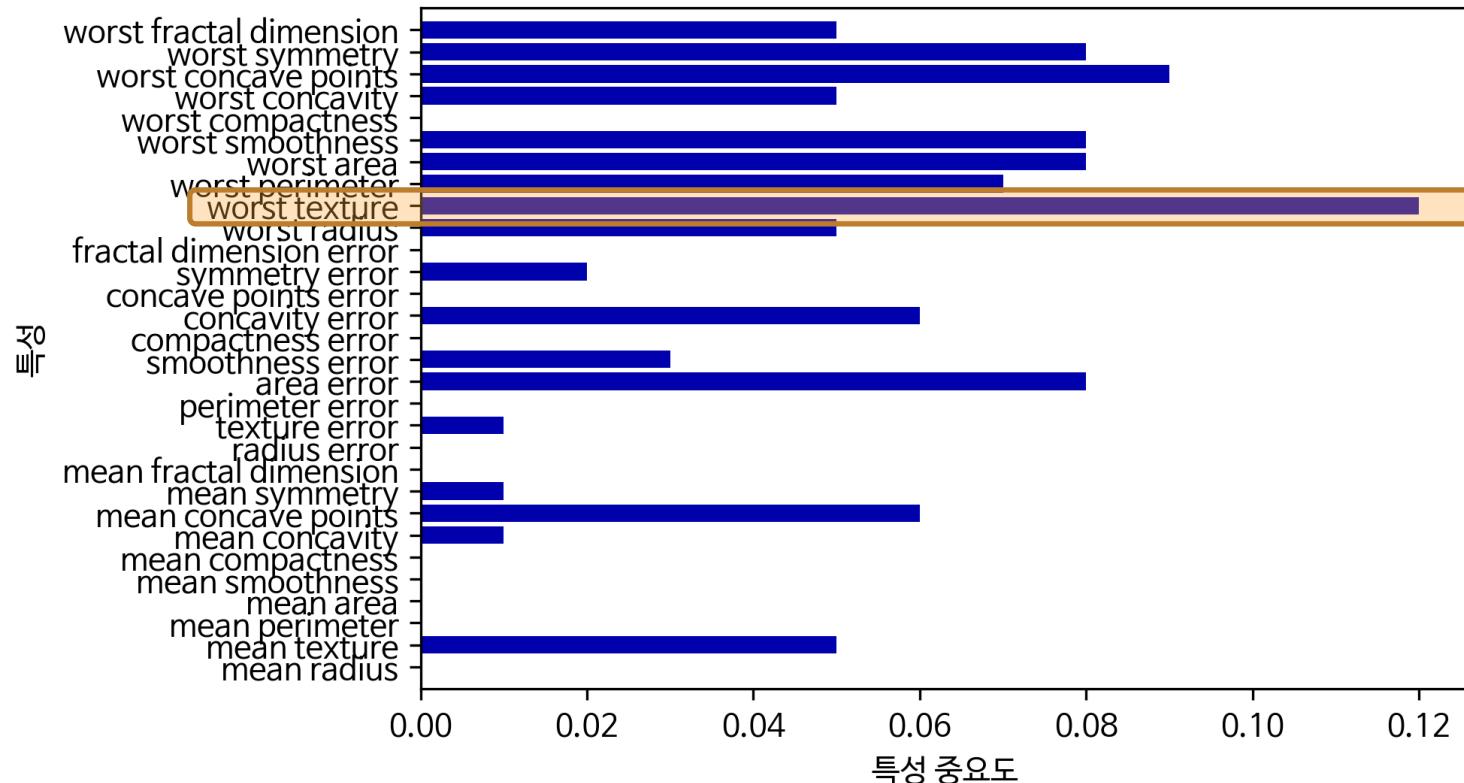
```
print("훈련 세트 정확도: {:.3f}".format(gbrt.score(x_train, y_train)))  
print("테스트 세트 정확도: {:.3f}".format(gbrt.score(x_test, y_test)))
```

훈련 세트 정확도: 0.988

테스트 세트 정확도: 0.965

학습률 감소

그래디언트 부스팅의 특성 중요도



장단점과 매개변수

장점

랜덤 포레스트에서 더 성능을 줘야 할 때 사용(보다 대규모일 경우 xgboost)
특성 스케일 조정 필요 없고 이진, 연속적인 특성에도 사용 가능함

단점

희소한 고차원 데이터에 잘 작동하지 않음
매개변수에 민감, 훈련 시간이 더 걸림

매개변수

n_estimators, learning_rate, max_depth(≤ 5)

learning_rate를 낮추면 복잡도가 낮아져 성능을 올리려면 더 많은 트리가 필요함

랜덤 포레스트와는 달리 n_estimators를 크게하면 과대적합 가능성 높아짐

가용 메모리와 시간안에서 n_estimators를 맞추고 learning_rate으로 조절

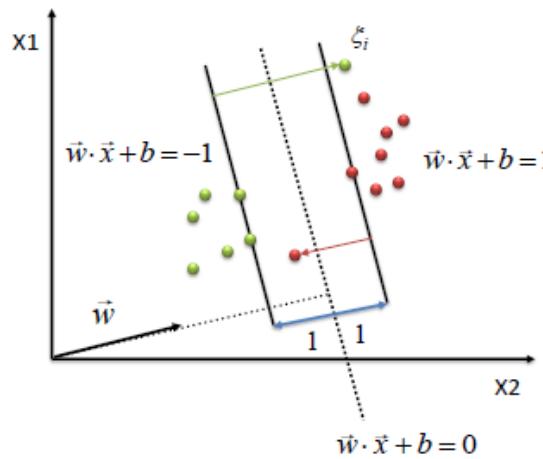
커널 서포트 벡터 머신

커널 kernel SVM

보통 그냥 SVM으로 부르는 경우가 많음

선형 SVM의 초평면으로 해결되지 않는 비선형 문제를 해결하기 위해 고안됨

분류 : SVC, 회귀 : SVR



Constraint becomes :

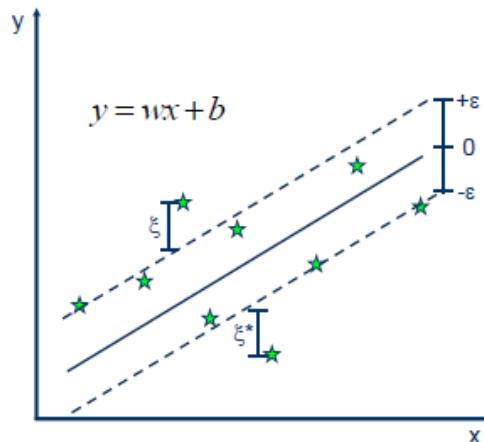
$$y_i(\vec{w} \cdot \vec{x}_i + b) \geq 1 - \xi_i, \forall x_i$$

$$\xi_i \geq 0$$

Objective function
penalizes for misclassified
instances and those within
the margin

$$\min \frac{1}{2} \|\vec{w}\|^2 + C \sum_i \xi_i$$

C trades-off margin width
and misclassifications



- Minimize:

$$\frac{1}{2} \|\vec{w}\|^2 + C \sum_{i=1}^N (\xi_i + \xi_i^*)$$

- Constraints:

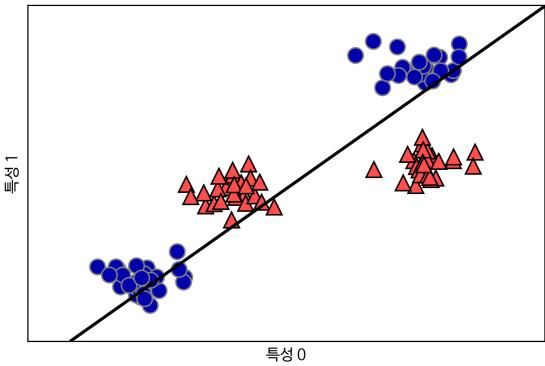
$$y_i - w\vec{x}_i - b \leq \epsilon + \xi_i$$

$$w\vec{x}_i + b - y_i \leq \epsilon + \xi_i^*$$

$$\xi_i, \xi_i^* \geq 0$$

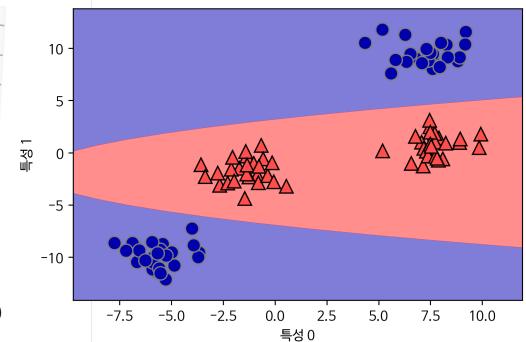
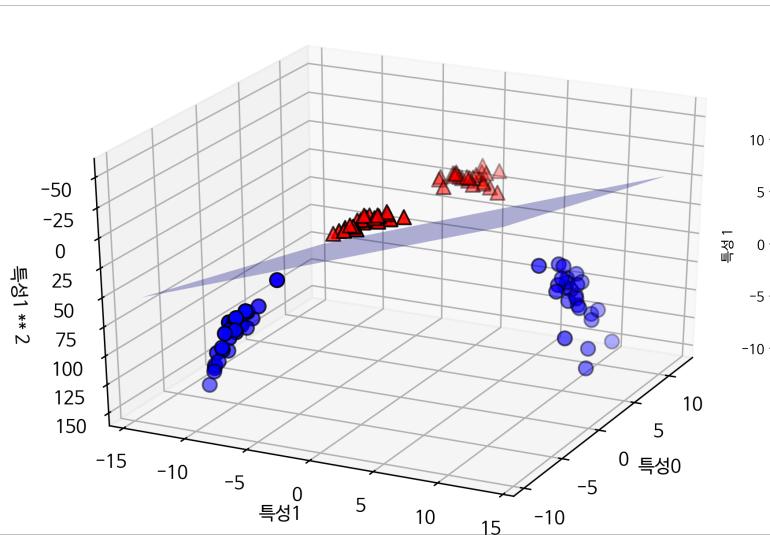
비선형 특성

```
In [78]: from sklearn.svm import LinearSVC  
linear_svm = LinearSVC().fit(X, y)
```



```
In [79]: # 두 번째 특성을 제곱하여 추가합니다  
X_new = np.hstack([X, X[:, 1:] ** 2])
```

```
In [80]: linear_svm_3d = LinearSVC().fit(X_new, y)
```



커널 기법 kernel trick

어떤 특성을 추가해야 할지 불분명하고 많은 특성을 추가하면 연산 비용 커짐

커널 함수라 부르는 특별한 함수를 사용하여 데이터 포인트 간의 거리를 계산하여 데이터 포인트의 특성이 고차원에 매핑된 것 같은 효과를 얻음

다항 커널

$$x_a = (a_1, a_2), x_b = (b_1, b_2) \text{ 일때 } (x_a \cdot x_b)^2 = (a_1 b_1 + a_2 b_2)^2 = \\ a_1^2 b_1^2 + 2a_1 b_1 a_2 b_2 + a_2^2 b_2^2 = (a_1^2, \sqrt{2}a_1 a_2, a_2^2) \cdot (b_1^2, \sqrt{2}b_1 b_2, b_2^2)$$

RBF radial basis function 커널

$$\exp(-\gamma \|x_a - x_b\|^2), e^x \text{의 테일러 급수 전개: } C \sum_{n=0}^{\infty} \frac{(x_a \cdot x_b)^n}{n!}$$

무한한 특성 공간으로 매핑하는 효과, 고차항일 수록 특성 중요성은 떨어짐

서포트 벡터

클래스 경계에 위치한 데이터 포인트를
서포트 벡터라 부름

RBF 커널을 가우시안 Gaussian 커널이라고도 부름

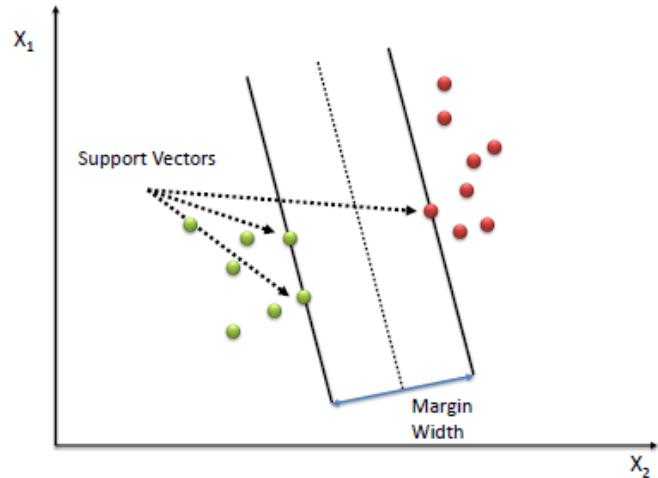
커널의 폭(gamma), $e^0 \sim e^{-\infty} = 1 \sim 0$

γ 가 작을 수록 샘플의 영향 범위 커짐

$\gamma = \frac{1}{2\sigma^2}$ 일 때 σ 를 커널의 폭이라 부름

$$\exp(-\gamma \|x_a - x_b\|^2)$$

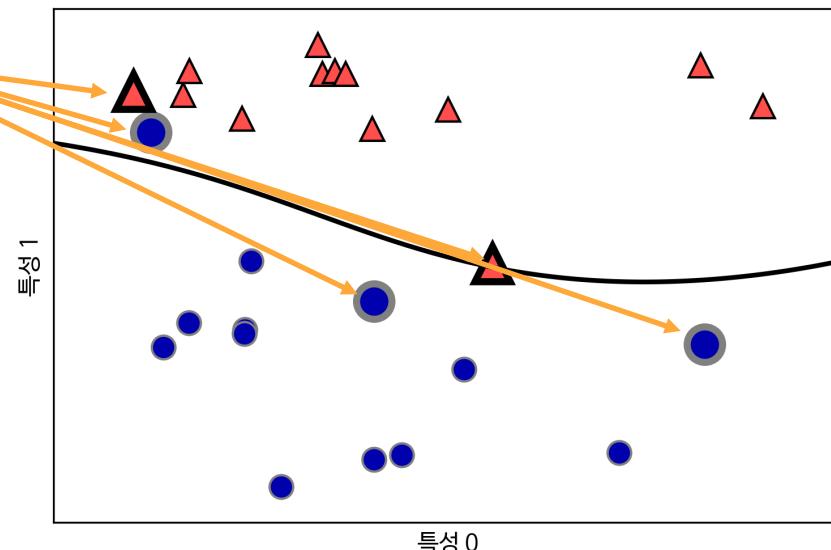
유클리디안 거리



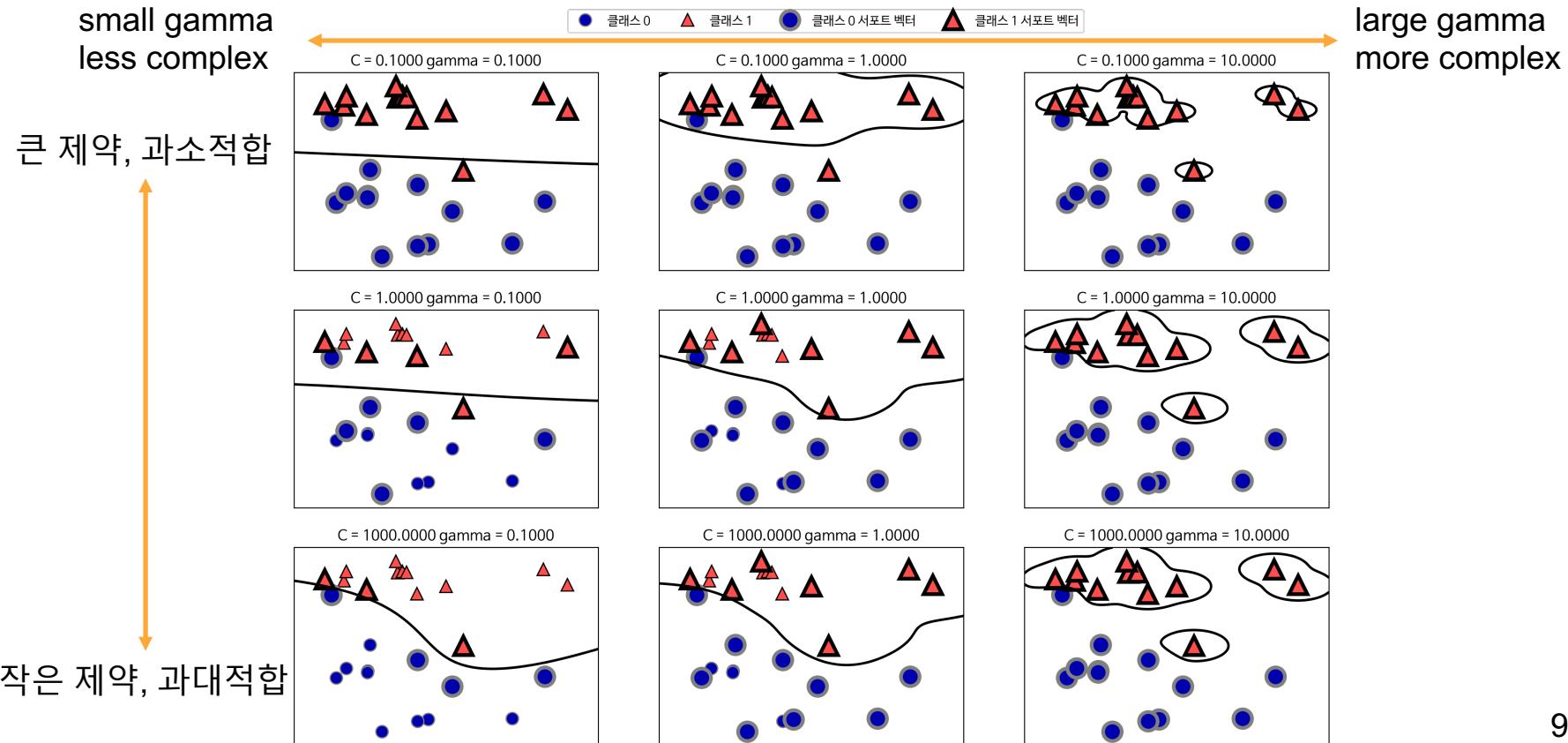
SVC + forge dataset

```
In [82]: from sklearn.svm import SVC  
  
X, y = mglearn.tools.make_handcrafted_dataset()  
svm = SVC(kernel='rbf', C=10, gamma=0.1).fit(X, y)  
# 서포트 벡터  
sv = svm.support_vectors_  
# dual_coef_의 부호에 의해 서포트 벡터의 클래스 레이블이 결정됩니다  
sv_labels = svm.dual_coef_.ravel() > 0
```

서포트 벡터



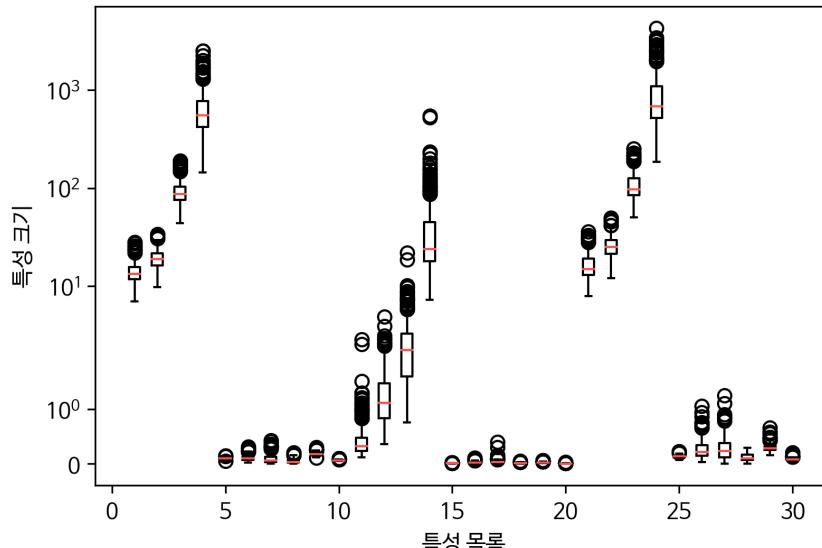
매개변수 튜닝



SVC + cancer dataset

```
In [84]:  
X_train, X_test, y_train, y_test = train_test_split(  
    cancer.data, cancer.target, random_state=0)  
  
svc = SVC() ← C=1, gamma=1/n_features  
svc.fit(X_train, y_train)  
  
print("훈련 세트 정확도: {:.2f}".format(svc.score(X_train, y_train)))  
print("테스트 세트 정확도: {:.2f}".format(svc.score(X_test, y_test)))  
  
훈련 세트 정확도: 1.00 ← 과대적합  
테스트 세트 정확도: 0.63
```

cancer 데이터셋의
데이터 스케일



SVM은 특성의 범위에
크게 민감함

데이터 전처리

MinMaxScaler : $\frac{X - \min(X)}{\max(X) - \min(X)}$, 0 ~ 1 사이로 조정

```
In [86]: # 훈련 세트에서 특성별 최솟값 계산  
min_on_training = X_train.min(axis=0)  
# 훈련 세트에서 특성별 (최댓값 - 최솟값) 범위 계산  
range_on_training = (X_train - min_on_training).max(axis=0)  
  
# 훈련 데이터에 최솟값을 빼고 범위로 나누면  
# 각 특성에 대해 최솟값은 0 최댓값은 1 임  
X_train_scaled = (X_train - min_on_training) / range_on_training  
print("특성별 최솟값\n{}".format(X_train_scaled.min(axis=0)))  
print("특성별 최댓값\n{}".format(X_train_scaled.max(axis=0)))
```

특성별 최소 값

```
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  
 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

특성별 최대 값

```
[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  
 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

```
In [87]: # 테스트 세트에도 같은 작업을 적용하지만  
# 훈련 세트에서 계산한 최솟값과 범위를 사용합니다(자세한 내용은 3장에 있습니다)  
X_test_scaled = (X_test - min_on_training) / range_on_training
```

SVC + 전처리 데이터

```
In [88]: svc = SVC()  
svc.fit(X_train_scaled, y_train)  
  
print("훈련 세트 정확도: {:.3f}".format(svc.score(X_train_scaled, y_train)))  
print("테스트 세트 정확도: {:.3f}".format(svc.score(X_test_scaled, y_test)))
```

훈련 세트 정확도: 0.948
테스트 세트 정확도: 0.951 ← 과소적합

```
In [89]: svc = SVC(C=1000) ← 제약완화  
svc.fit(X_train_scaled, y_train)  
  
print("훈련 세트 정확도: {:.3f}".format(svc.score(X_train_scaled, y_train)))  
print("테스트 세트 정확도: {:.3f}".format(svc.score(X_test_scaled, y_test)))
```

훈련 세트 정확도: 0.988
테스트 세트 정확도: 0.972

장단점과 매개변수

장점

강력하며 여러 종류의 데이터셋에 적용가능
특성이 적을 때에도 복잡한 결정 경계 만들(커널 트릭)

단점

샘플이 많을 경우 느림 ($>100,000$)
데이터 전처리와 매개변수에 민감
분석하기 어렵고 비전문가에게 설명하기 어려움

매개변수

C, gamma(RBF 커널), coef0(다항, 시그모이드), degree(다항)
 $linear: x_1 \cdot x_2$, $poly: (\gamma(x_1 \cdot x_2) + c)^d$, $sigmoid: \tanh(\gamma(x_1 \cdot x_2) + c)$

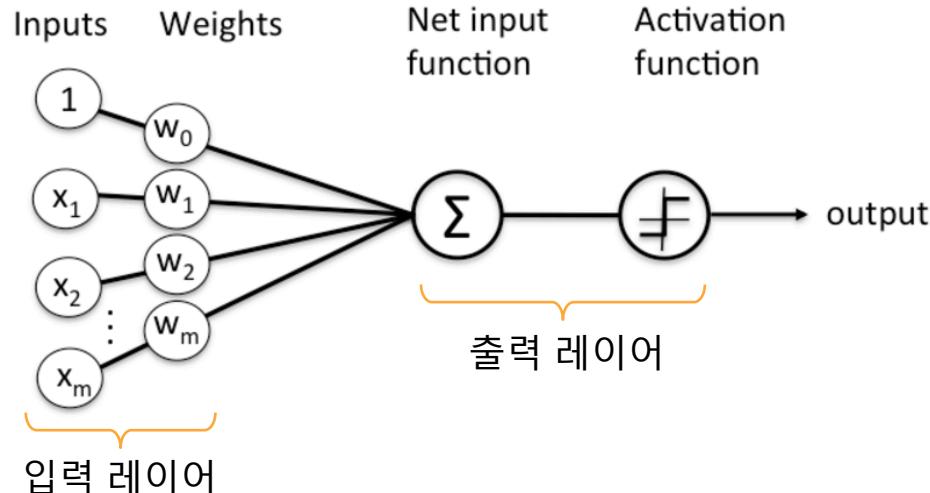
신경망 neural network

퍼셉트론 perceptron

1957년 프랑크 로젠틀라트가 발표

뉴럴 네트워크를 멀티레이어 퍼셉트론으로도 부름

사이킷런(v0.18)에 MLPClassifier, MLPRegressor 추가



Naming

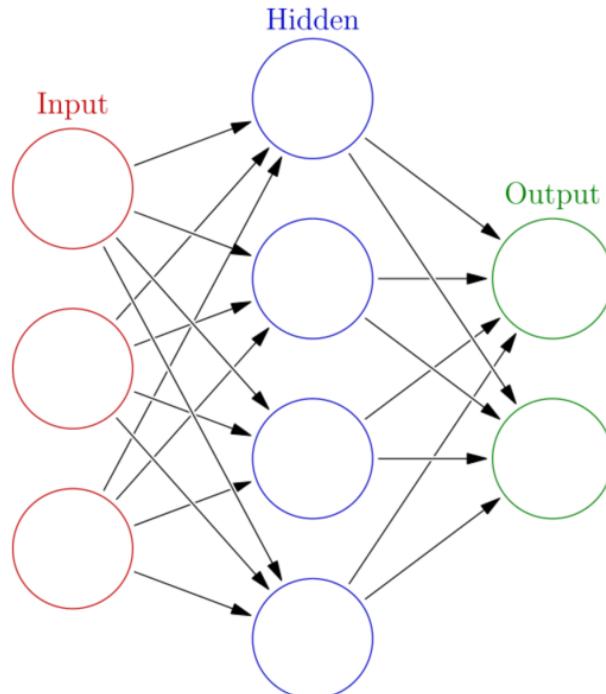
완전 연결 뉴럴 네트워크
(Fully Connected Neural Network)

덴스 네트워크
(Dense Network)

멀티 레이어 퍼셉트론
(Multi-Layer Perceptron)

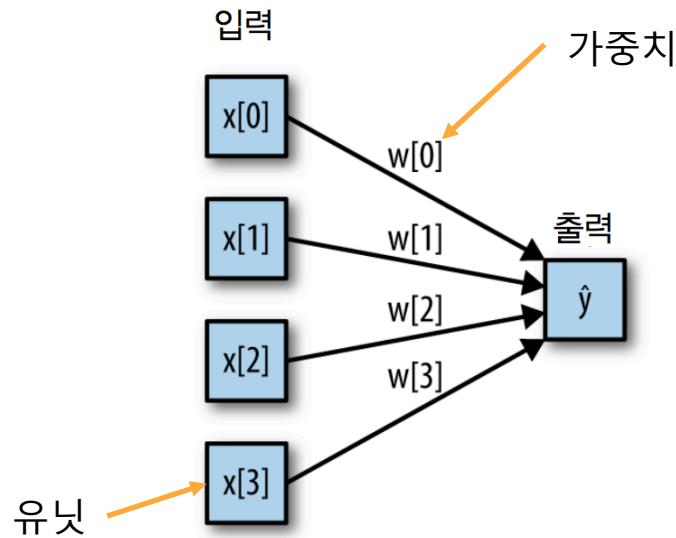
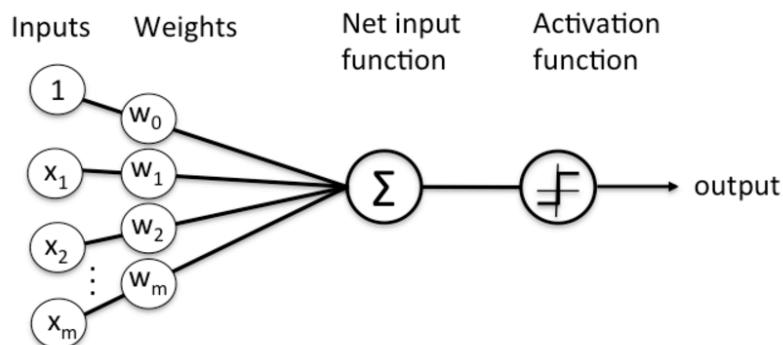
피드 포워드 뉴럴 네트워크
(Feed Forward Neural Network)

딥 러닝
(Deep Learning)



선형 모델의 일반화

$$\hat{y} = w[0] \times x[0] + w[1] \times x[1] + \cdots + w[p] \times x[p] + b$$



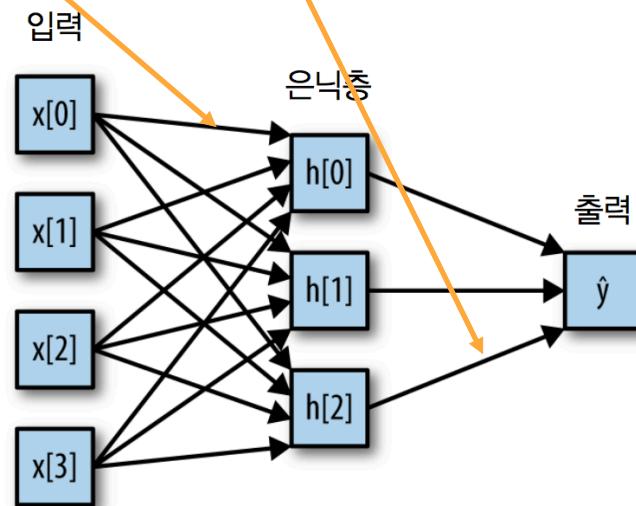
다층 레이어

$$h[0] = w[0,1] \times x[0] + w[1,0] \times x[1] + w[2,0] \times x[2] + w[3,0] \times x[3] + b[0]$$

$$h[0] = w[0,1] \times x[0] + w[1,1] \times x[1] + w[2,1] \times x[2] + w[3,1] \times x[3] + b[1]$$

$$h[0] = w[0,1] \times x[0] + w[1,2] \times x[1] + w[2,2] \times x[2] + w[3,2] \times x[3] + b[2]$$

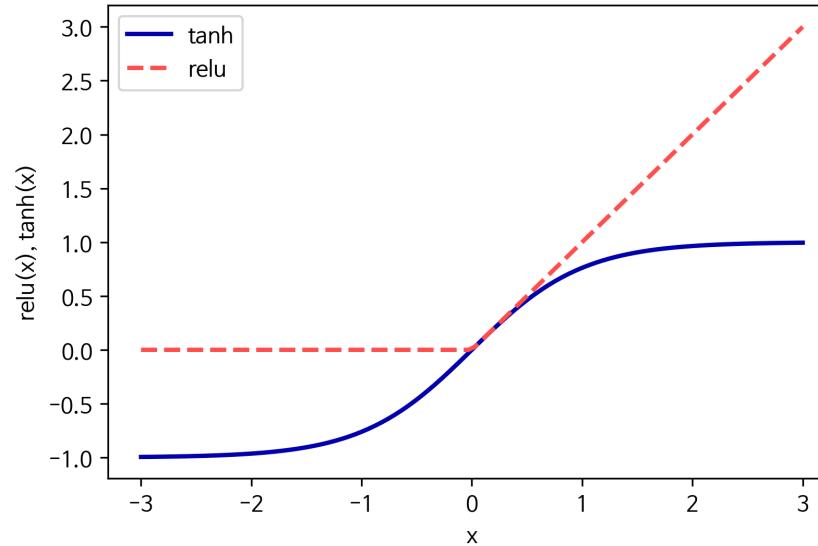
$$\hat{y} = v[0] \times h[0] + v[1] \times h[1] + v[2] \times h[2] + b$$



활성화 함수 activation function

은닉 유닛의 가중치 합의 결과에 비선형성을 주입

렐루 ReLU, 하이퍼볼릭 탄젠트 \tanh , 시그모이드 sigmoid



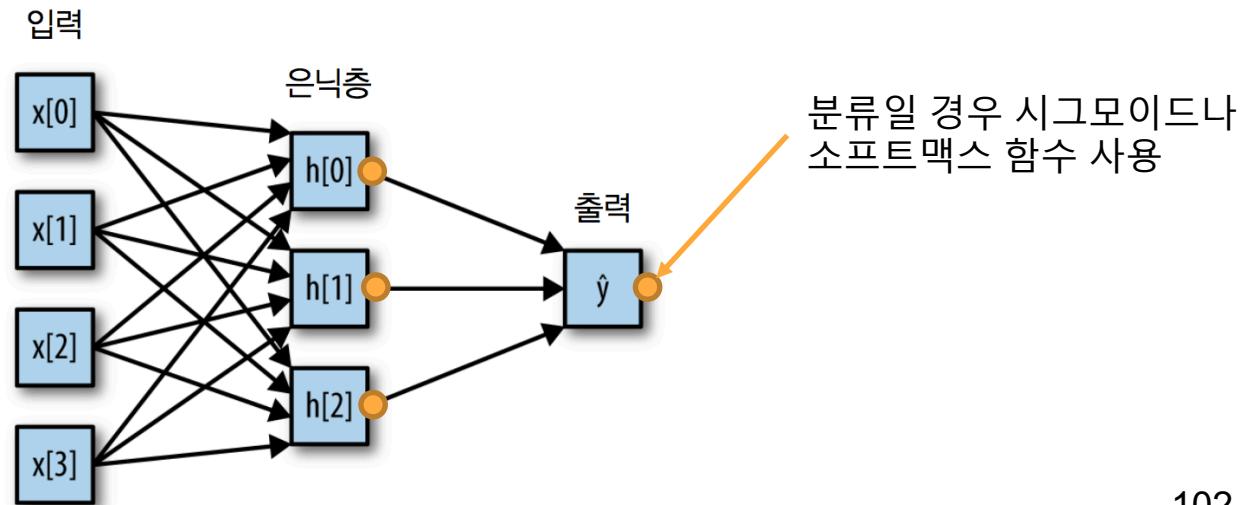
다층 퍼셉트론

$$h[0] = \tanh(w[0,1] \times x[0] + w[1,0] \times x[1] + w[2,0] \times x[2] + w[3,0] \times x[3] + b[0])$$

$$h[0] = \tanh(w[0,1] \times x[0] + w[1,1] \times x[1] + w[2,1] \times x[2] + w[3,1] \times x[3] + b[1])$$

$$h[0] = \tanh(w[0,1] \times x[0] + w[1,2] \times x[1] + w[2,2] \times x[2] + w[3,2] \times x[3] + b[2])$$

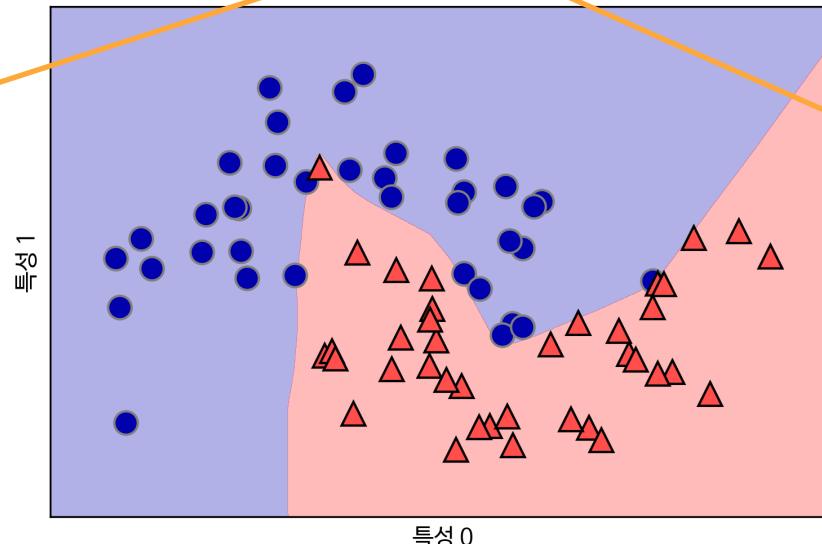
$$\hat{y} = v[0] \times h[0] + v[1] \times h[1] + v[2] \times h[2] + b$$



MLPClassifier + two_moons

```
In [94]: from sklearn.neural_network import MLPClassifier  
from sklearn.datasets import make_moons  
  
X, y = make_moons(n_samples=100, noise=0.25, random_state=3)  
  
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,  
random_state=42)  
  
mlp = MLPClassifier(solver='lbfgs', random_state=0).fit(X_train, y_train)
```

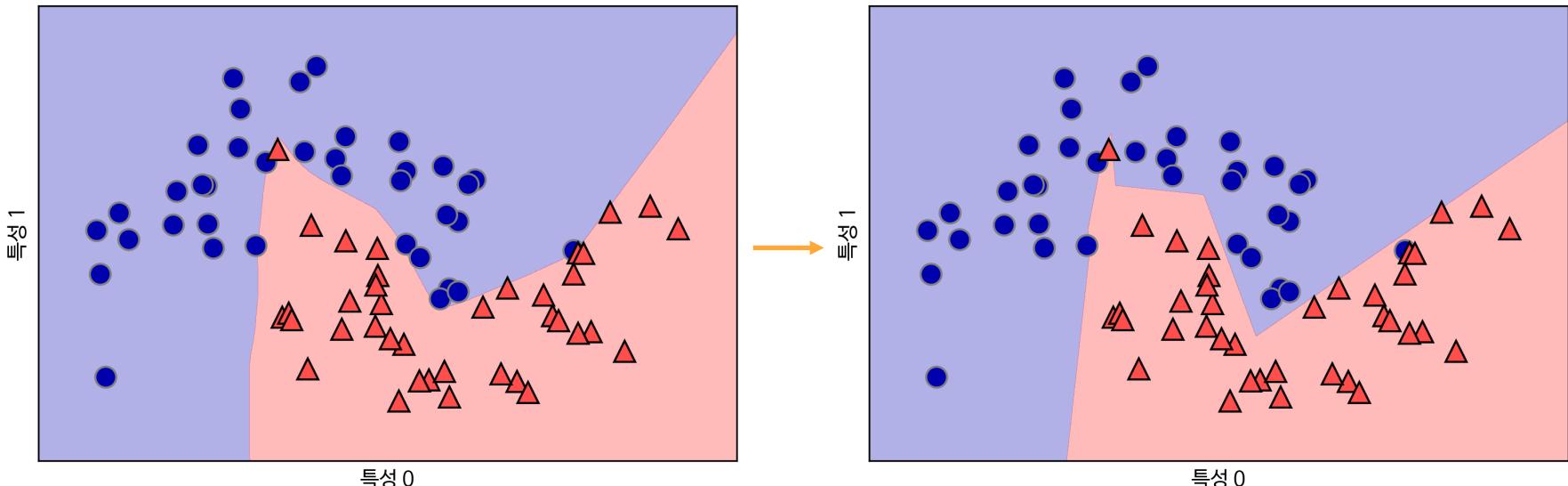
Limited BFGS
의사 뉴턴 방법



hidden_layer_sizes=[100],
activation='relu'

은닉 유닛 개수 = 10

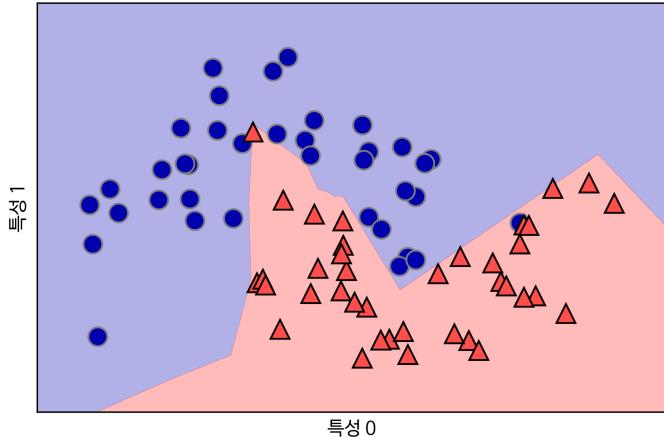
```
In [95]: mlp = MLPClassifier(solver='lbfgs', random_state=0, hidden_layer_sizes=[10])
mlp.fit(X_train, y_train)
```



레이어 추가, tanh 활성화 함수

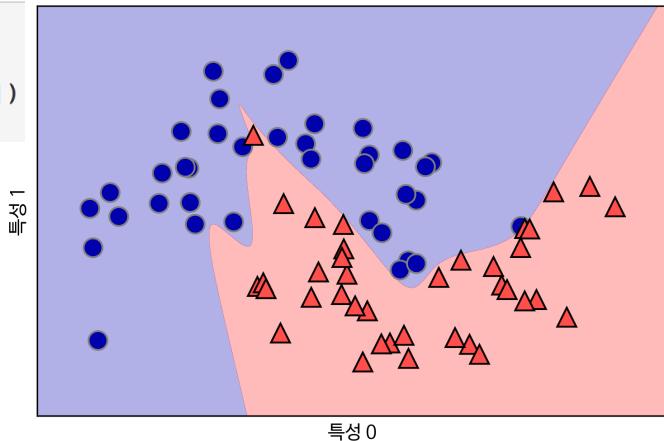
In [96]: # 10개의 유닛으로 된 두 개의 은닉층

```
mlp = MLPClassifier(solver='lbfgs', random_state=0,  
                     hidden_layer_sizes=[10, 10])  
mlp.fit(X_train, y_train)
```

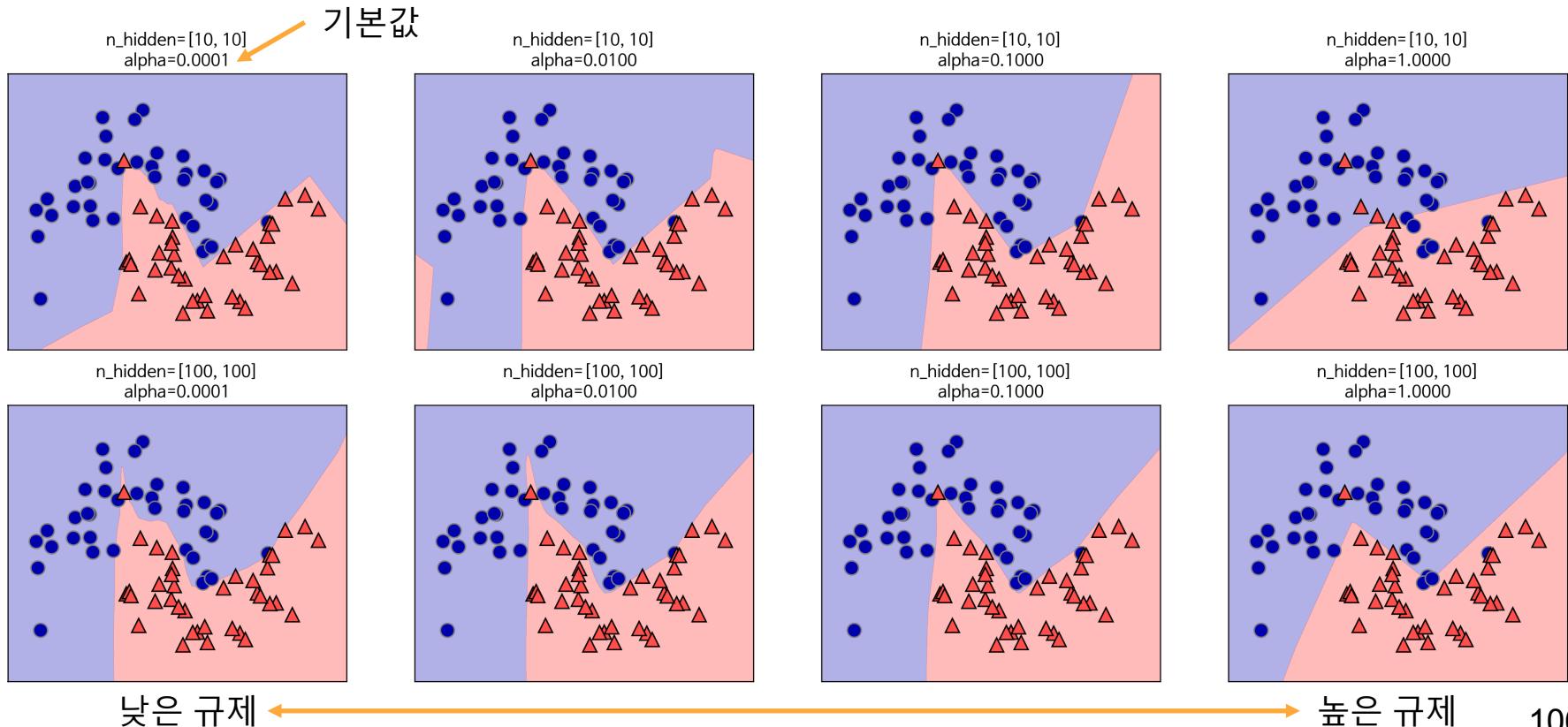


In [97]: # tanh 활성화 함수가 적용된 10개의 유닛으로 된 두 개의 은닉층

```
mlp = MLPClassifier(solver='lbfgs', activation='tanh',  
                     random_state=0, hidden_layer_sizes=[10, 10])  
mlp.fit(X_train, y_train)
```



L2 규제 매개변수 C



신경망의 복잡도

은닉층의 수가 많을 수록,

은닉층의 유닛 개수가 많을 수록,

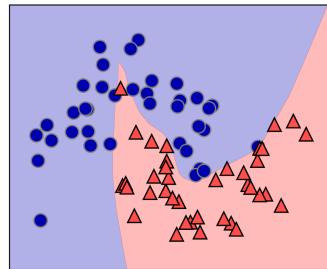
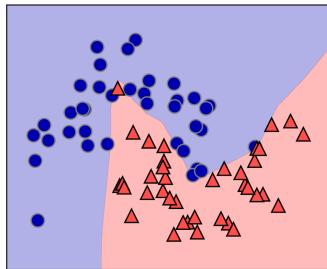
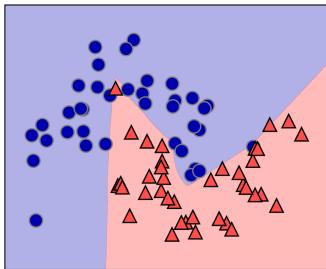
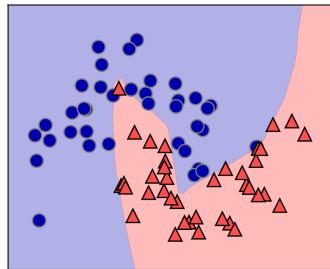
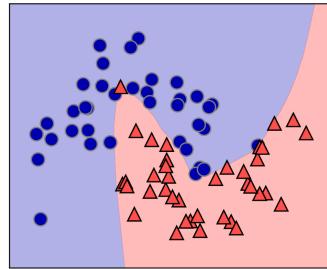
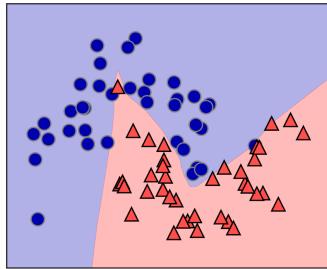
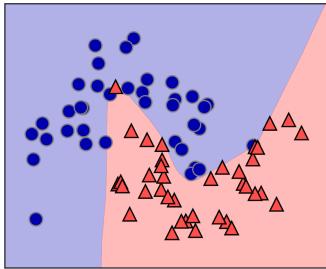
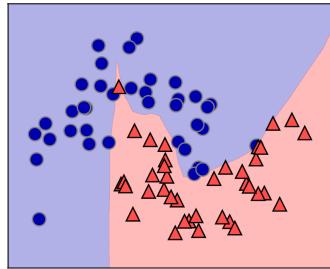
규제가 낮을 수록 복잡도가 증가됨

모델 훈련시 은닉층의 유닛의 일부를 랜덤하게 작동시키지 않아 마치 여러개의 신경망을 앙상블하는 것 같은 드롭아웃(dropout)이 신경망에서 과대적합 방지하는 대표적인 방법 → scikit-learn 0.19에서 추가될 예정

랜덤 초기화

모델을 훈련할 때 가중치를 무작위로 초기화

모델의 크기와 복잡도가 낮으면 영향을 미칠 수 있음



MLPClassifier + cancer dataset

신경망에서도 데이터의 범주에 민감함

```
In [101]: X_train, X_test, y_train, y_test = train_test_split(  
            cancer.data, cancer.target, random_state=0)  
  
mlp = MLPClassifier(random_state=42)  
mlp.fit(X_train, y_train)  
  
print("훈련 세트 정확도: {:.2f}".format(mlp.score(X_train, y_train)))  
print("테스트 세트 정확도: {:.2f}".format(mlp.score(X_test, y_test)))
```

훈련 세트 정확도: 0.91
테스트 세트 정확도: 0.88

```
In [102]: # 훈련 세트 각 특성의 평균을 계산합니다  
mean_on_train = X_train.mean(axis=0)  
# 훈련 세트 각 특성의 표준 편차를 계산합니다  
std_on_train = X_train.std(axis=0)  
  
# 데이터에서 평균을 빼고 표준 편차로 나누면  
# 평균 0, 표준 편차 1 인 데이터로 변환됩니다.  
X_train_scaled = (X_train - mean_on_train) / std_on_train  
# (훈련 데이터의 평균과 표준 편차를 이용해) 같은 변화를 테스트 세트에도 합니다  
X_test_scaled = (X_test - mean_on_train) / std_on_train
```

표준정규분포(평균:0, 분산:1)
StandardScaler

MLPClassifier + adam

최대 반복횟수 도달
(기본값 200)

```
mlp = MLPClassifier(random_state=0)
mlp.fit(X_train_scaled, y_train)

print("훈련 세트 정확도: {:.3f}".format(mlp.score(X_train_scaled, y_train)))
print("테스트 세트 정확도: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

훈련 세트 정확도: 0.991
테스트 세트 정확도: 0.965

```
/Users/rickypark/anaconda/envs/introduction_to_ml_with_python/lib/python3.!
% (), ConvergenceWarning)
```

In [103]:

```
mlp = MLPClassifier(max_iter=1000, random_state=0)
mlp.fit(X_train_scaled, y_train)

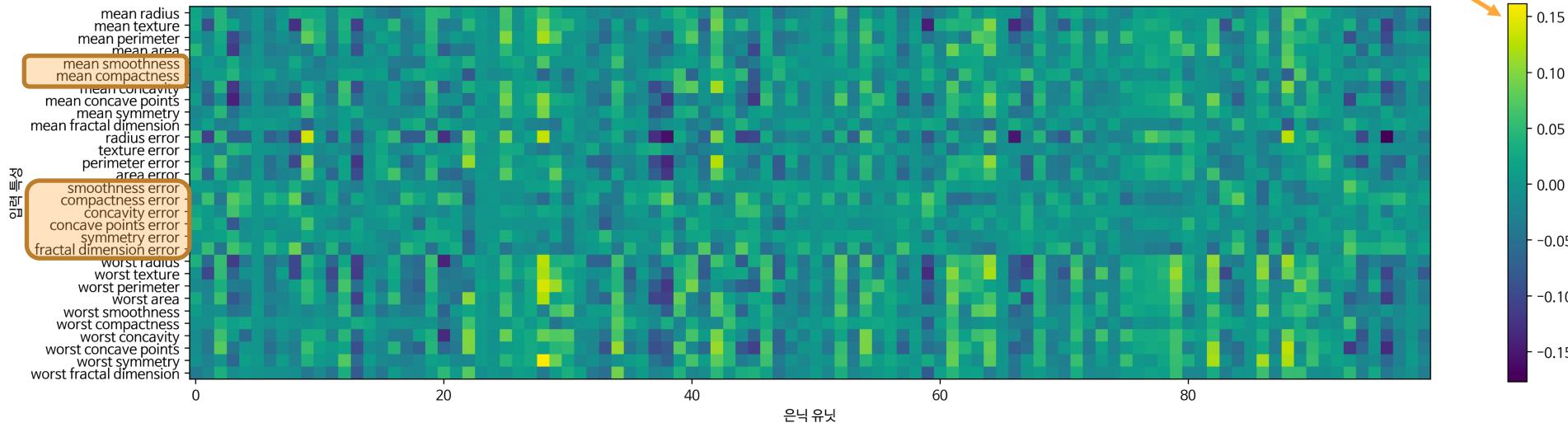
print("훈련 세트 정확도: {:.3f}".format(mlp.score(X_train_scaled, y_train)))
print("테스트 세트 정확도: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

훈련 세트 정확도: 0.993
테스트 세트 정확도: 0.972

신경망의 가중치 검사

입력층과 은닉층 사이의 가중치

밝을수록 큰 값



은닉층과 출력층 사이의 가중치는 해석하기 더 어려움

장단점과 매개변수

장점

충분한 시간과 데이터가 있으면 매우 복잡한 모델을 만들어 냄
다른 알고리즘들을 압도하는 성능을 발휘(음성, 영상, 번역 등)

단점

데이터 전처리에 민감, 이종의 데이터 타입일 경우 잘 안맞음
매개변수 튜닝이 매우 어려움
콘볼루션 convolution이나 리커런트 recurrent 신경망을 제공하지 않음

매개변수

```
solver=['adam', 'sgd', 'lbfgs'],
if sgd, momentum + nesterovs_momentum
alpha
```

분류의 불확실성 추정

분류의 불확실성

어떤 테스트 포인트에 대해 예측 클래스 뿐만 아니라 얼마나 그 클래스임을 확신하는지가 중요할 때가 있음

대부분 decision_function 과 predict_proba 메서드 둘 중 하나는 제공함

```
In [107]: from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import make_circles
X, y = make_circles(noise=0.25, factor=0.5, random_state=1)

# 예제를 위해 클래스의 이름을 "blue" 와 "red" 로 바꿉니다
y_named = np.array(["blue", "red"])[y]

# 여러개의 배열을 한꺼번에 train_test_split 에 넣을 수 있습니다
# 훈련 세트와 테스트 세트로 나누는 방식은 모두 같습니다.
X_train, X_test, y_train_named, y_test_named, y_train, y_test = \
    train_test_split(X, y_named, y, random_state=0)

# 그래디언트 부스팅 모델을 만듭니다
gbdt = GradientBoostingClassifier(random_state=0)
gbdt.fit(X_train, y_train_named)
```

결정 함수

이진 분류에서 decision_function() 반환값의 크기 (n_samples,)

```
In [108]: print("X_test.shape: {}".format(X_test.shape))
print("결정 함수 결과 형태: {}".format(gbrt.decision_function(X_test).shape))
```

```
X_test.shape: (25, 2)
결정 함수 결과 형태: (25,)
```

```
In [109]: # 결정 함수 결과 중 앞부분 일부를 확인합니다
print("결정 함수:\n{}".format(gbrt.decision_function(X_test)[:6]))
```

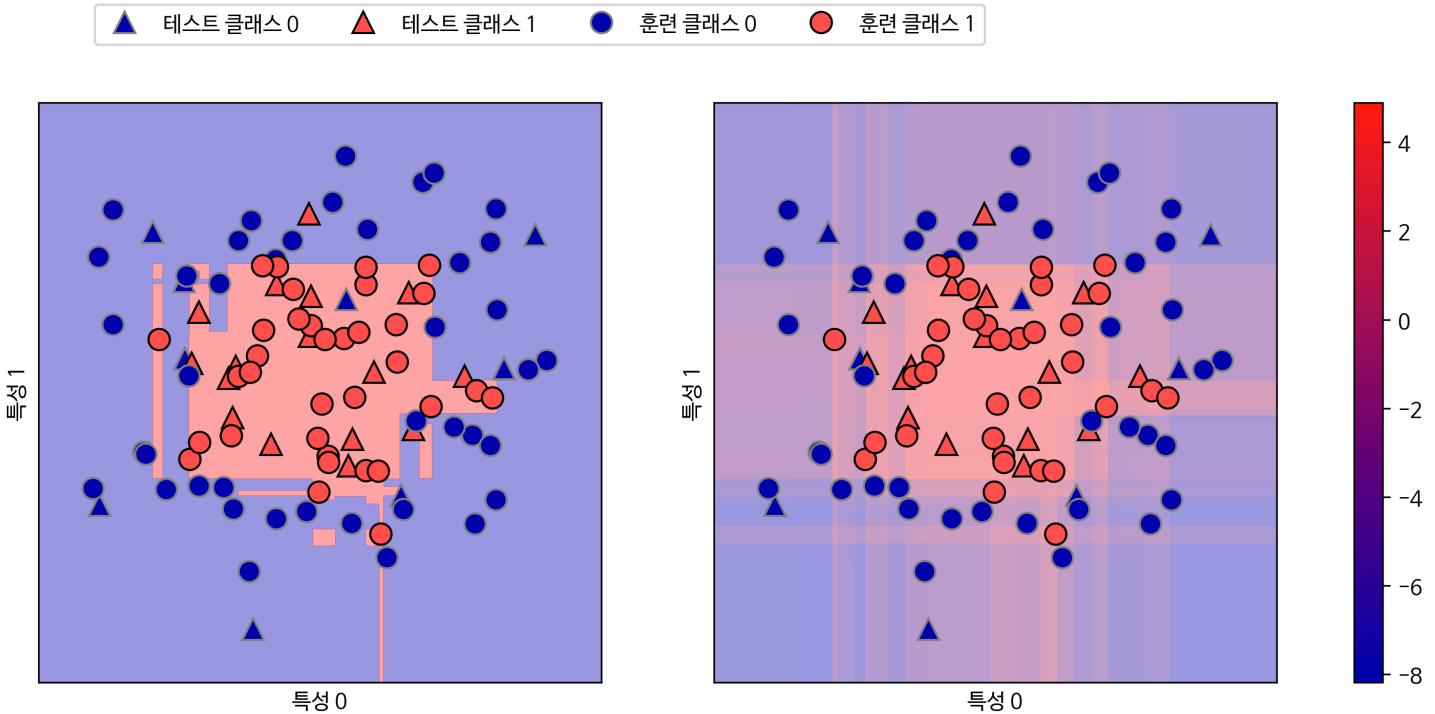
```
결정 함수:
[ 4.136 -1.702 -3.951 -3.626  4.29   3.662]
```

```
In [110]: print("임계치와 결정 함수 결과 비교:\n{}".format(
    gbrt.decision_function(X_test) > 0))
print("예측:\n{}".format(gbrt.predict(X_test)))
```

```
임계치와 결정 함수 결과 비교:
[ True False False False  True  True False  True  True False  True
  True False  True False False  True  True  True  True False
 False]
예측:
```

```
[ 'red' 'blue' 'blue' 'blue' 'red' 'red' 'blue' 'red' 'red' 'red' 'blue'
  'red' 'red' 'blue' 'red' 'blue' 'blue' 'blue' 'red' 'red' 'red' 'red'
  'red' 'blue' 'blue']
```

결정 경계 + decision_function



예측 확률

`predict_proba()` 반환값의 크기 (`n_samples, n_classes`)

가장 큰 확률 값을 가진 클래스를 예측 클래스로 선택함

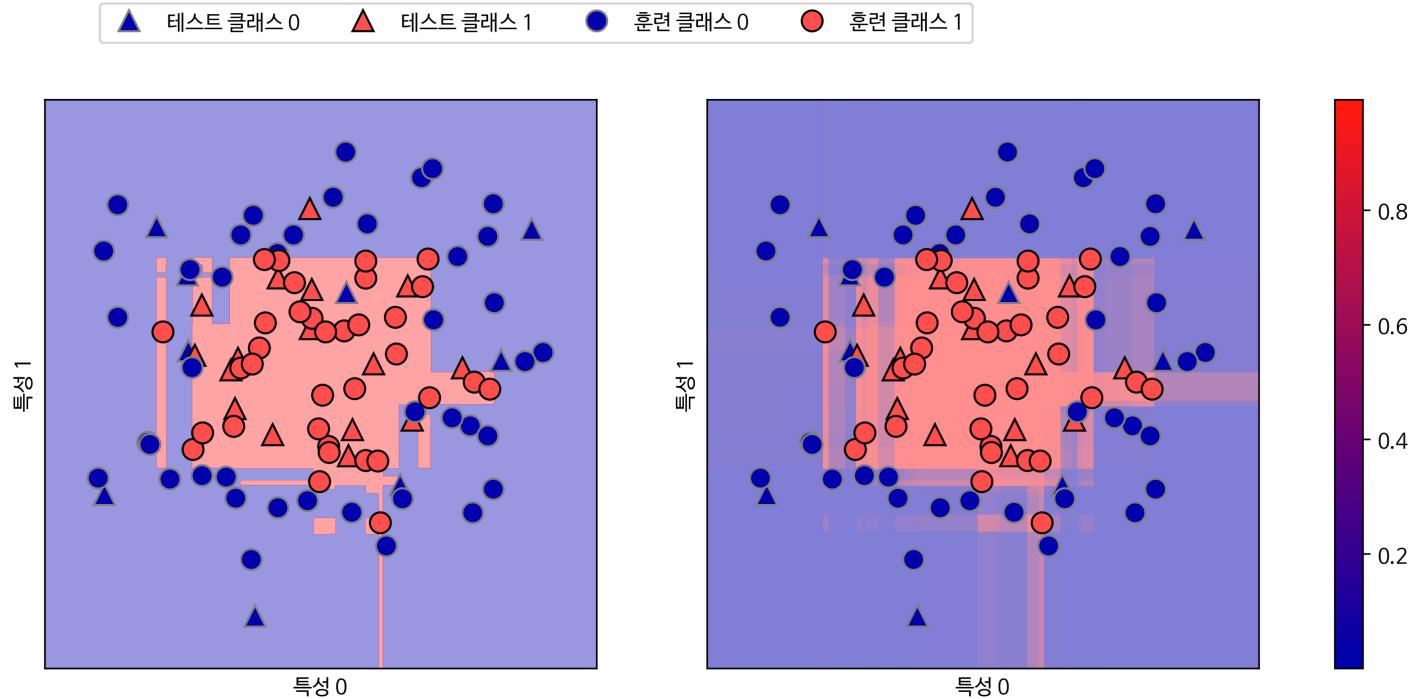
```
In [114]: print("확률 값의 형태: {}".format(gbrt.predict_proba(X_test).shape))
```

확률 값의 형태: (25, 2)

```
In [115]: # predict_proba 결과 중 앞부분 일부를 확인합니다
print("Predicted probabilities:\n{}".format(
    gbrt.predict_proba(X_test[:6])))
```

```
Predicted probabilities:
[[ 0.016  0.984]
 [ 0.846  0.154]
 [ 0.981  0.019]
 [ 0.974  0.026]
 [ 0.014  0.986]
 [ 0.025  0.975]]
```

결정 경계 + predict_proba



다중 분류 + decision_function

반환값의 크기는 (n_samples, n_classes)

가장 큰 값이 예측 클래스가 됨

```
In [117]: from sklearn.datasets import load_iris

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=42)

gbrt = GradientBoostingClassifier(learning_rate=0.01, random_state=0)
gbrt.fit(X_train, y_train)

In [118]: print("결정 함수의 결과 형태: {}".format(gbrt.decision_function(X_test).shape))
# plot the first few entries of the decision function
print("결정 함수 결과:\n{}".format(gbrt.decision_function(X_test)[:6, :]))
```

결정 함수의 결과 형태: (38, 3)

결정 함수 결과:

```
[[ -0.529  1.466 -0.504]
 [ 1.512 -0.496 -0.503]
 [-0.524 -0.468  1.52 ]
 [-0.529  1.466 -0.504]
 [-0.531  1.282  0.215]
 [ 1.512 -0.496 -0.503]]
```

다중 분류 + predict_proba

반환값의 크기는 (n_samples, n_classes)

```
In [120]: # predict_proba 결과 중 앞부분 일부를 확인합니다
print("예측 확률:\n{}".format(gbdt.predict_proba(X_test)[:6]))
# 행 방향으로 확률을 더하면 1 이 됩니다
print("합: {}".format(gbdt.predict_proba(X_test)[:6].sum(axis=1)))
```

```
예측 확률:
[[ 0.107  0.784  0.109]
 [ 0.789  0.106  0.105]
 [ 0.102  0.108  0.789]
 [ 0.107  0.784  0.109]
 [ 0.108  0.663  0.228]
 [ 0.789  0.106  0.105]]
합: [ 1.  1.  1.  1.  1.]
```

```
In [121]: print("가장 큰 예측 확률의 인덱스:\n{}".format(
    np.argmax(gbdt.predict_proba(X_test), axis=1)))
print("예측:\n{}".format(gbdt.predict(X_test)))
```

```
가장 큰 예측 확률의 인덱스:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1
 0]
예측:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1
 0]
```

요약 및 정리

알고리즘 정리

최근접 이웃

작은 데이터셋일 경우, 기본 모델로서 좋고 설명하기 쉬움.

선형 모델

첫 번째로 시도할 알고리즘. 대용량 데이터셋 가능. 고차원 데이터에 가능.

나이브 베이즈

분류만 가능. 선형 모델보다 훨씬 빠름. 대용량 데이터셋과 고차원 데이터에 가능.

선형 모델보다 덜 정확함.

결정 트리

매우 빠름. 데이터 스케일 조정이 필요 없음. 시각화하기 좋고 설명하기 쉬움.

랜덤 포레스트

결정 트리 하나보다 거의 항상 좋은 성능을 냄. 매우 안정적이고 강력함.
데이터 스케일 조정 필요 없음. 고차원 희박 데이터에는 잘 안 맞음.

그래디언트 부스팅

랜덤 포레스트보다 조금 더 성능이 좋음. 랜덤 포레스트보다 학습은 느리나 예측은
빠르고 메모리를 조금 사용. 랜덤 포레스트보다 매개변수 튜닝이 많이 필요함.

서포트 벡터 머신

비슷한 의미의 특성으로 이뤄진 중간 규모 데이터셋에 잘 맞음.
데이터 스케일 조정 필요. 매개변수에 민감.

신경망

특별히 대용량 데이터셋에서 매우 복잡한 모델을 만들 수 있음.
매개변수 선택과 데이터 스케일에 민감. 큰 모델은 학습이 오래 걸림.