Introduction to Machine Learning with Python

2. Supervised Learning(2)

Honedae Machine Learning Study Epoch #2

Contacts

Haesun Park

Email : haesunrpark@gmail.com

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

Facebook : https://facebook.com/haesunrpark

Blog : https://tensorflow.blog

Book

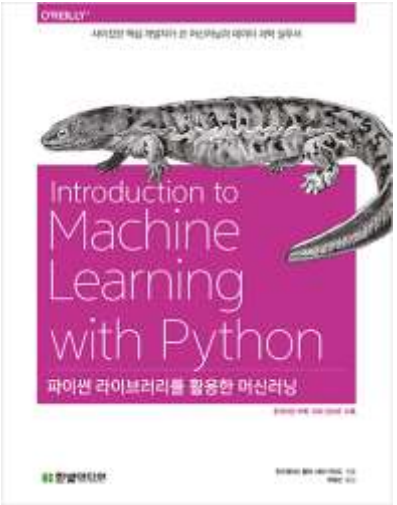
파이썬 라이브러리를 활용한 머신러닝, 박해선.

(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/



선형 모델 - 분류

이진 분류binary 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)

predict_proba()는 sigmoid 함수사용, predict()는 decision_function() 사용

$\hat{y}$이 입력에 대한 결정 경계decision boundary를 나타냄

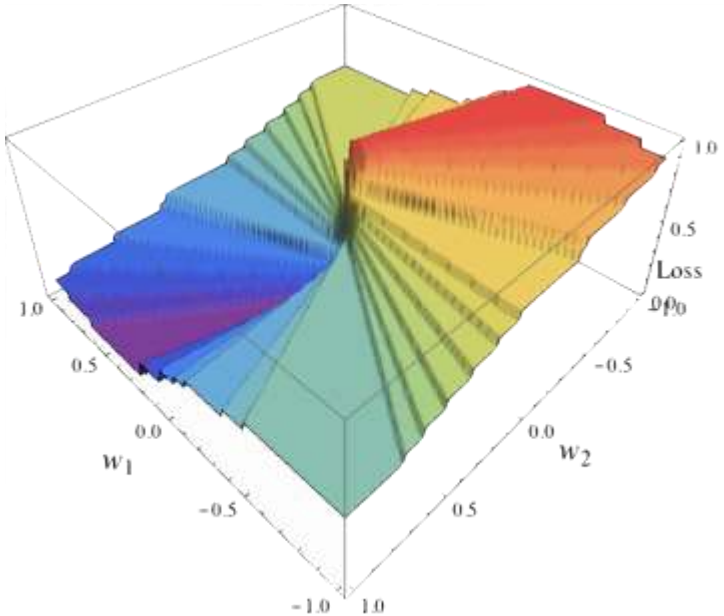
결정 경계는 직선(특성 1개), 면(특성 2개), 초평면(특성 3개)으로 표현됨

선형 모델의 알고리즘 구분

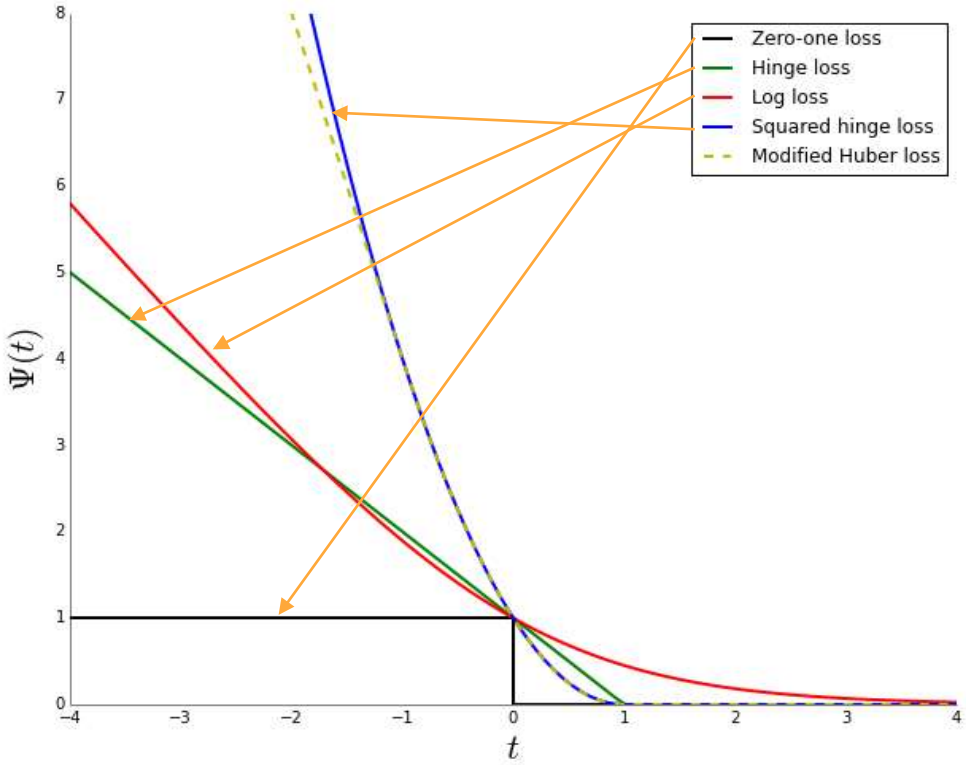
가중치와 절편이 훈련 데이터에 얼마나 잘 맞는지 → 비용 함수 또는 손실 함수 사용할 수 있는 규제의 방법

0-1 손실

계단 함수라 미분 불가능 → 최적화에 사용하기 어려움



대리 손실 함수surrogate loss function



LogisticRegression, LinearSVC

대표적인 선형 분류 알고리즘

로지스틱 회귀Logistic Regression(sklearn.linear_model.LogisticRegression)

선형 서포트 벡터 머신Linear SVM(sklearn.svm.LinearSVC)

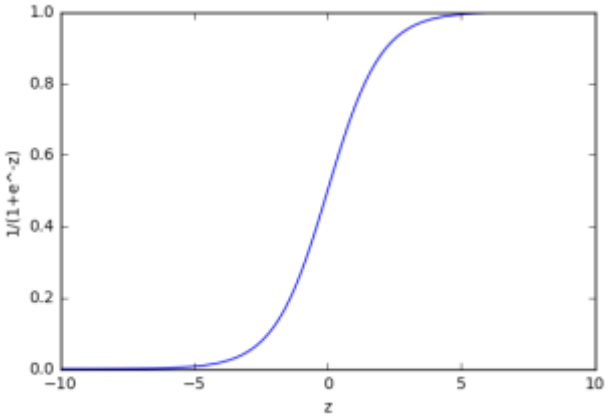
로지스틱(시그모이드sigmoid) 함수

$$\hat{y} = \frac{1}{1 + e^{-z}}$$

$$z = X \cdot w + b$$

predict_proba() : $\hat{y} > 0.5$

predict() : $z > 0$



로지스틱 회귀의 비용함수

multi_class=multinomial
(크로스 엔트로피cross-entropy 손실)

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

소프트맥스softmax 함수

multi_class=ovr
(로지스틱logistic 손실)

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

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

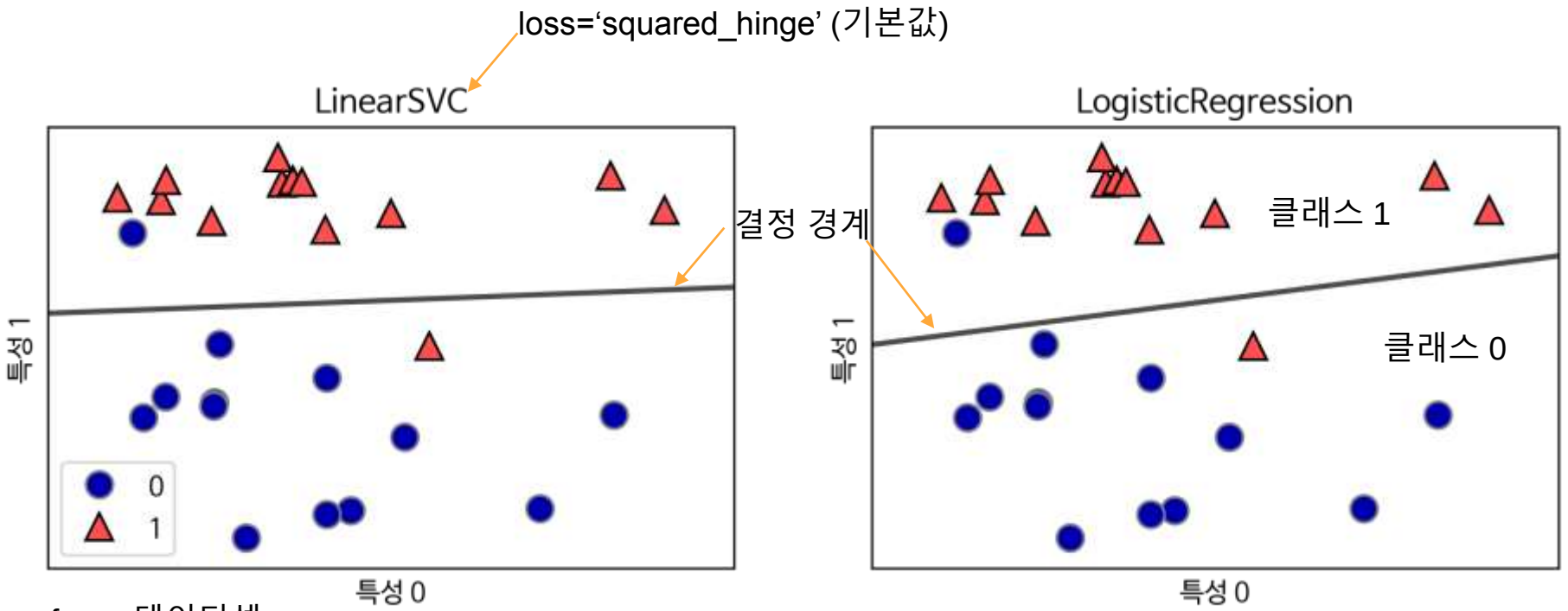
텍스트의 공식과
구현 방식에는
종종 차이가 있음

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

규제 강도

$$\alpha \uparrow \quad C \downarrow \rightarrow \text{규제} \uparrow \quad w \downarrow$$
$$\alpha \downarrow \quad C \uparrow \rightarrow \text{규제} \downarrow \quad w \uparrow$$

LinearSVC vs LogisticRegression

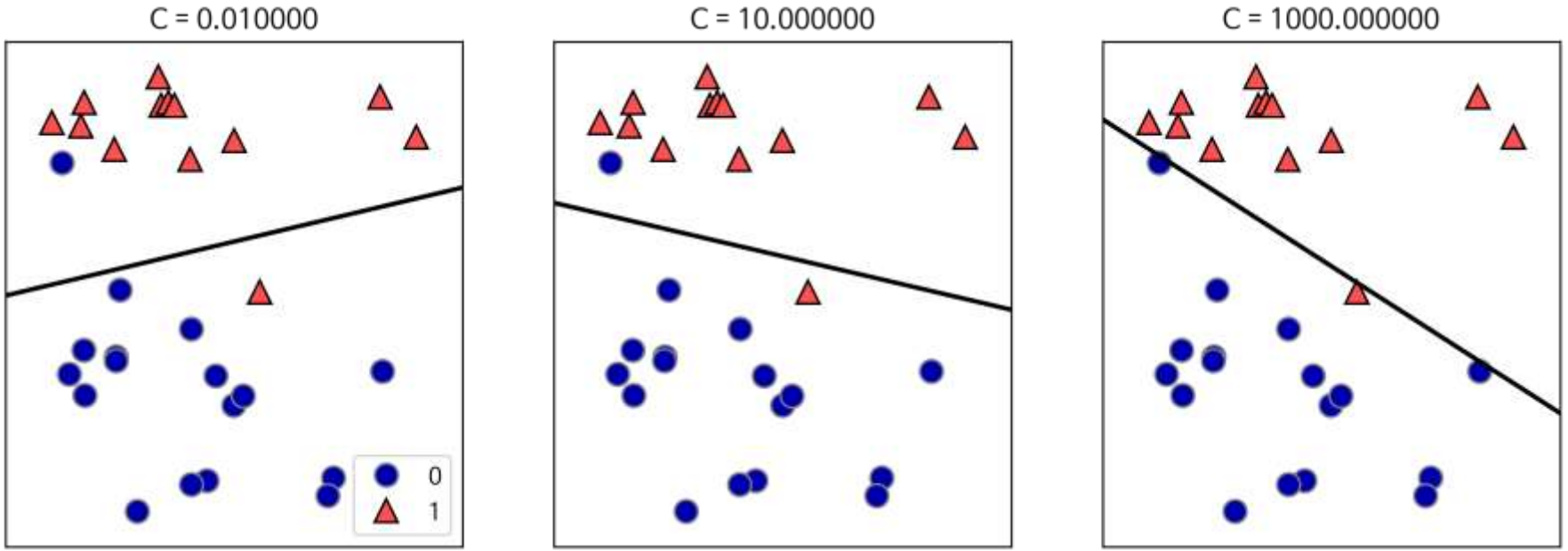


forge 데이터셋

L2 규제(penalty='l2' 기본값)

LinearSVC's C param

C=1.0 기본값



과소적합 ← → 과대적합

규제 강도

$C \downarrow \rightarrow$ 규제 $\uparrow$ $w \downarrow$ (다수의 포인트에 맞춤)

$C \uparrow \rightarrow$ 규제 $\downarrow$ $w \uparrow$ (개개의 포인트에 맞춤)

LogisticRegression + cancer

30개 특성

C=1.0

```
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

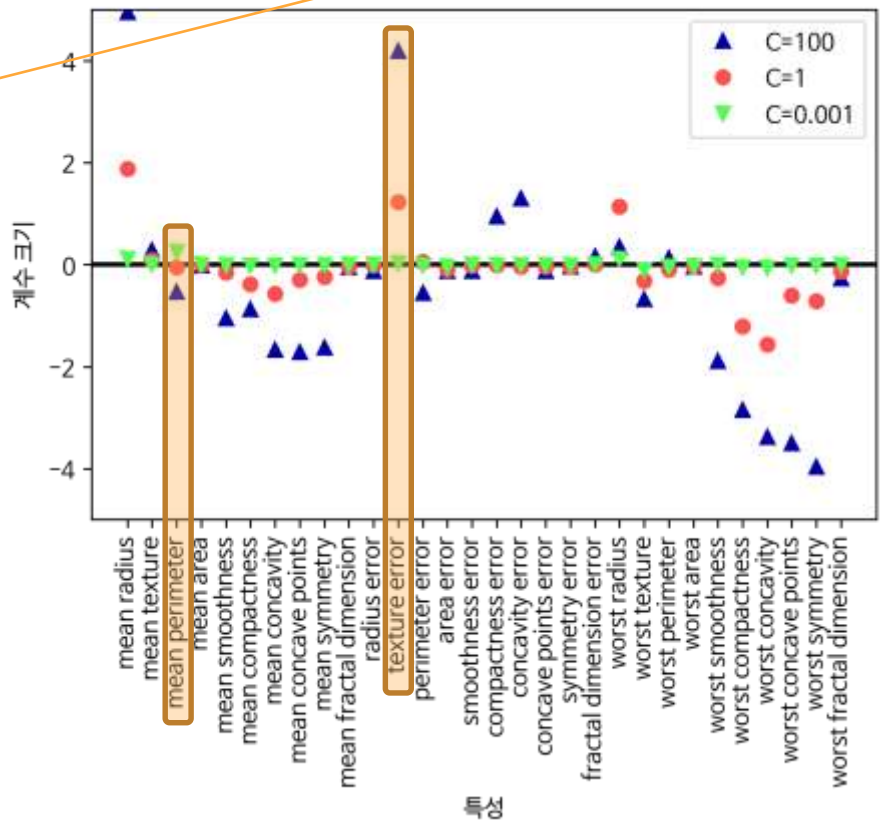
```
In [45]: logreg001 = LogisticRegression(C=0.01).fit(X_train, y_train)
print("훈련 세트 점수: {:.3f}".format(logreg001.score(X_train, y_train)))
print("테스트 세트 점수: {:.3f}".format(logreg001.score(X_test, y_test)))
```

```
훈련 세트 점수: 0.934
테스트 세트 점수: 0.930
```

복잡도를 기본값 보다 더 낮추면

LogisticRegression.coef_ (L2)

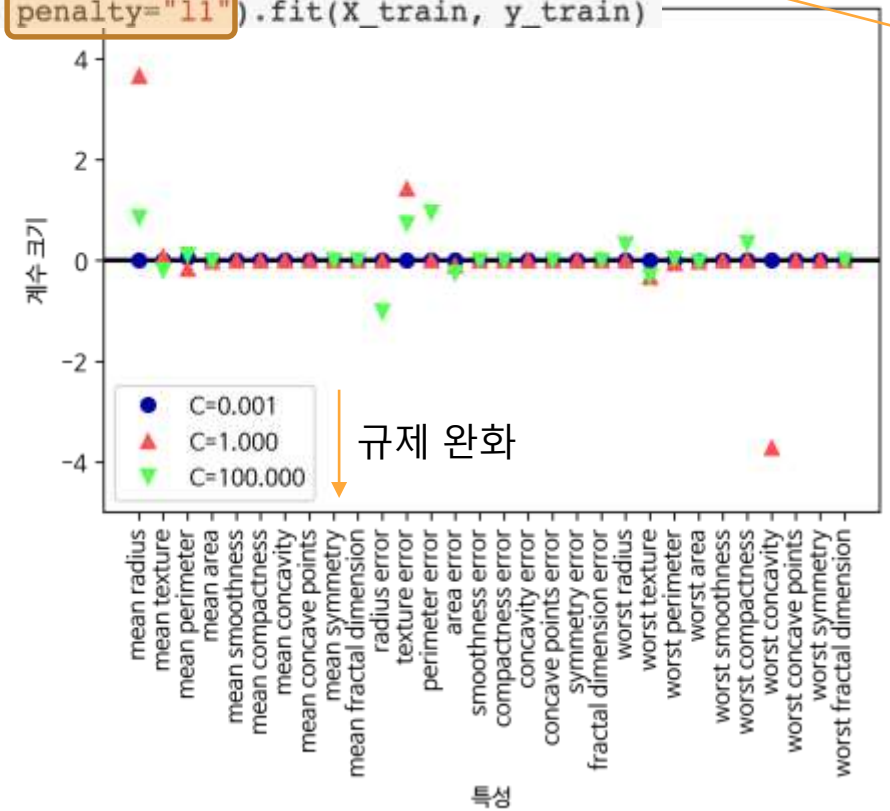
회귀의 Ridge와 비슷



규제 완화

LogisticRegression.coef_ (L1)

```
LogisticRegression(C=C, penalty="l1").fit(X_train, y_train)
```



회귀의 Lasso와 비슷

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)

$$-\sum_{i=1}^{n} y log(\hat{y}) \quad ,\hat{y}_c = \frac{e^{z_c}}{\sum_{k=1}^{K} e^{z_k}}$$

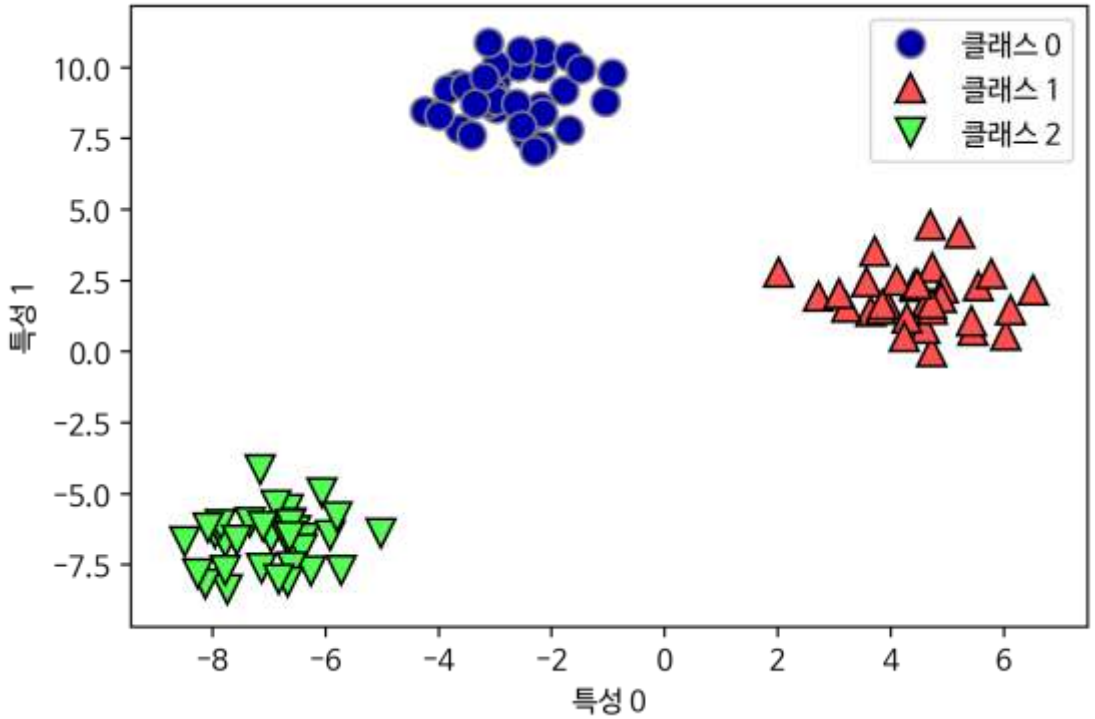
$$z_k = w_k[0] \times x[0] + w_k[1] \times x[1] + \cdots + w_k[p] \times x[p] + b_k$$

대부분 모델들은 클래스마다 따로 이진 분류 모델을 만드는 일대다one-vs-rest 방식(또는 one-vs-all)을 사용하여 이진 분류 모델 중에서 가장 높은 점수의 클래스가 선택

클래스마다 가중치와 절편이 만들어짐

make_blobs dataset

3개의 클래스를 가진 2차원 데이터셋



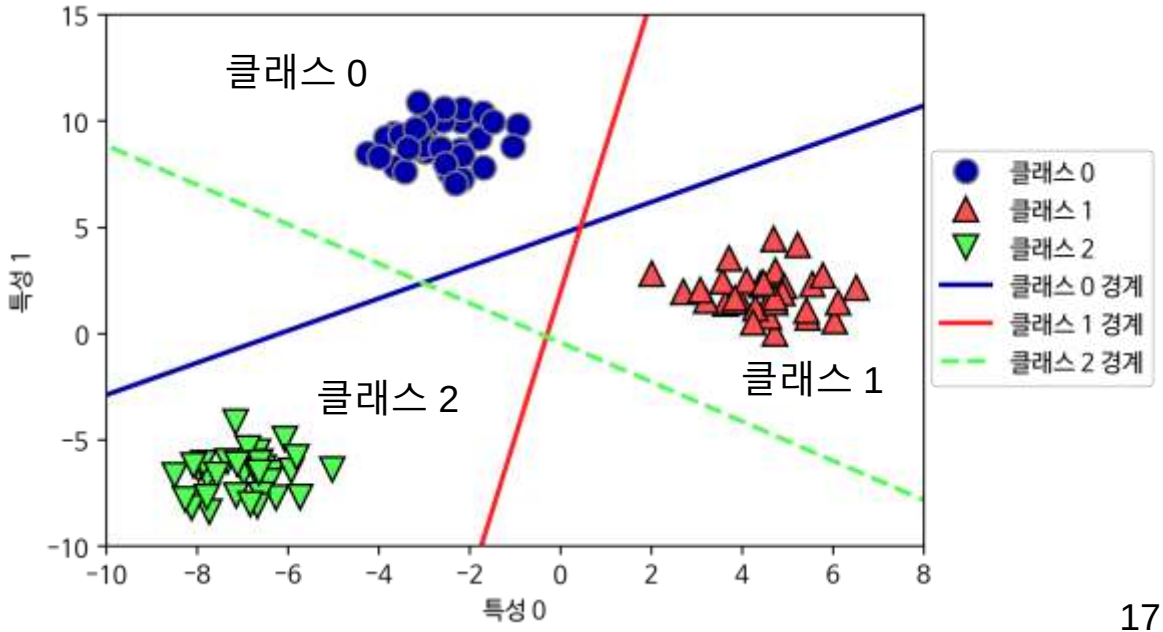
LinearSVC 다중 분류

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

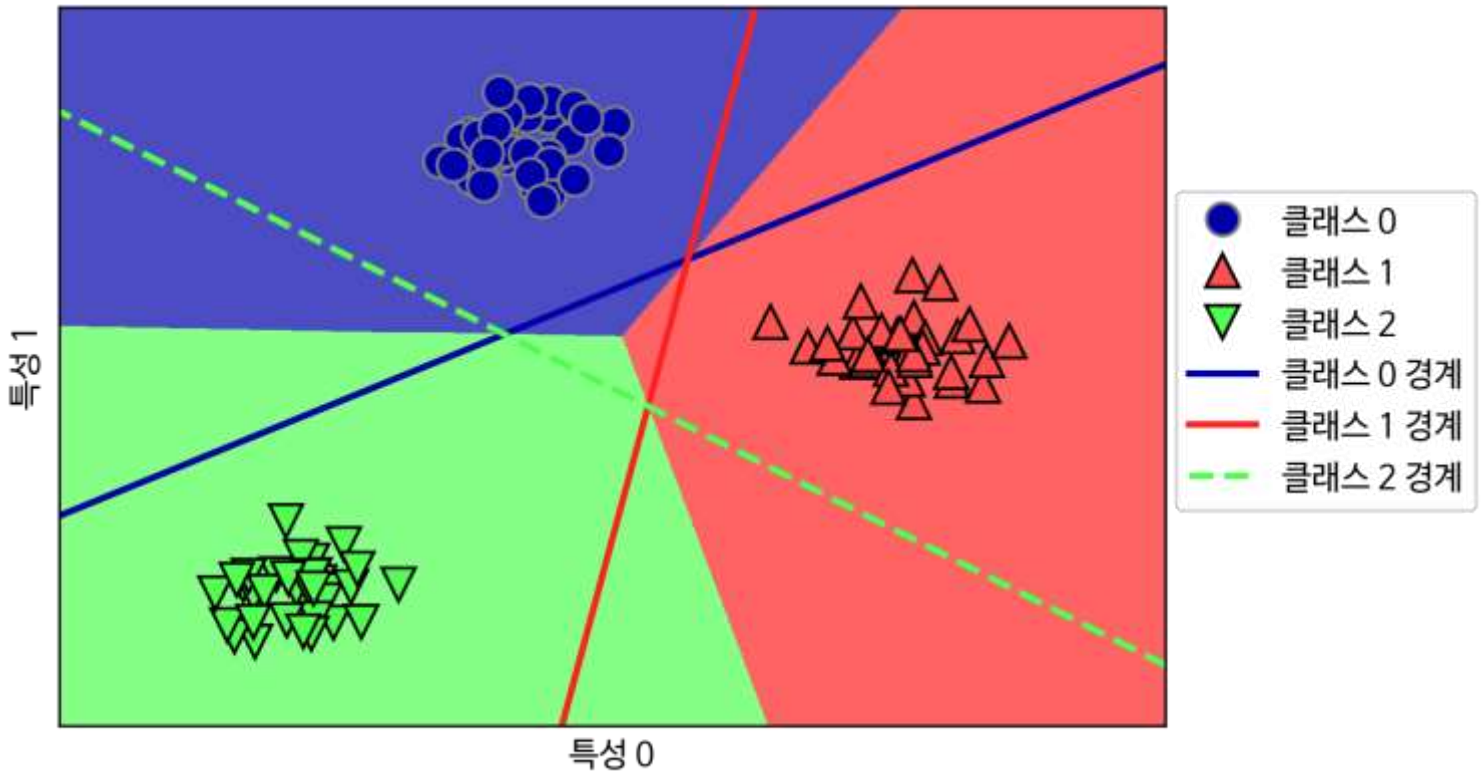
```
계수 배열의 크기:  (3, 2)
절편 배열의 크기:  (3,)
```

세 개의 클래스

두 개의 특성



다중 분류 결정 경계



장단점과 매개변수

규제 매개변수: 회귀 모델은 alpha, 분류 모델은 C

$\alpha\uparrow \quad C\downarrow \rightarrow$ 규제 $\uparrow \quad w\downarrow$
$\alpha\downarrow \quad C\uparrow \rightarrow$ 규제 $\downarrow \quad w\uparrow$

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(벡터 곱셈)
- 비교적 예측 과정을 이해하기 쉬움(계수 분석이 어려울 수 있음)
- 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

메서드 연결method chaining

객체(self) 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)
```

예측 결과 반환

학습한 로지스틱 회귀 모델을 다음 코드에서 사용할 수 없음

나이브 베이즈

나이브 베이즈Navie Bayes 분류기

선형 분류기보다 훈련 속도가 빠르지만 일반화 성능이 조금 떨어집니다.

특성마다 클래스별 통계를 취합해 파라미터를 학습합니다.

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 조절합니다.

가상의 양수 데이터를 alpha 개수만큼 추가하여 통계를 완만하게 만듦니다.

alpha가 크면 모델의 복잡도가 낮아지지만 성능의 변화는 크지 않습니다.

GaussianNB는 고차원 데이터셋에 BernoulliNB, MultinomialNB는 텍스트와 같은 희소 데이터를 카운트하는데 사용합니다.

MultinomialNB는 0이 아닌 특성이 많은 경우 BernoulliNB보다 성능이 좋습니다.

훈련과 예측 속도가 빠르고 과정을 이해하기 쉽습니다.

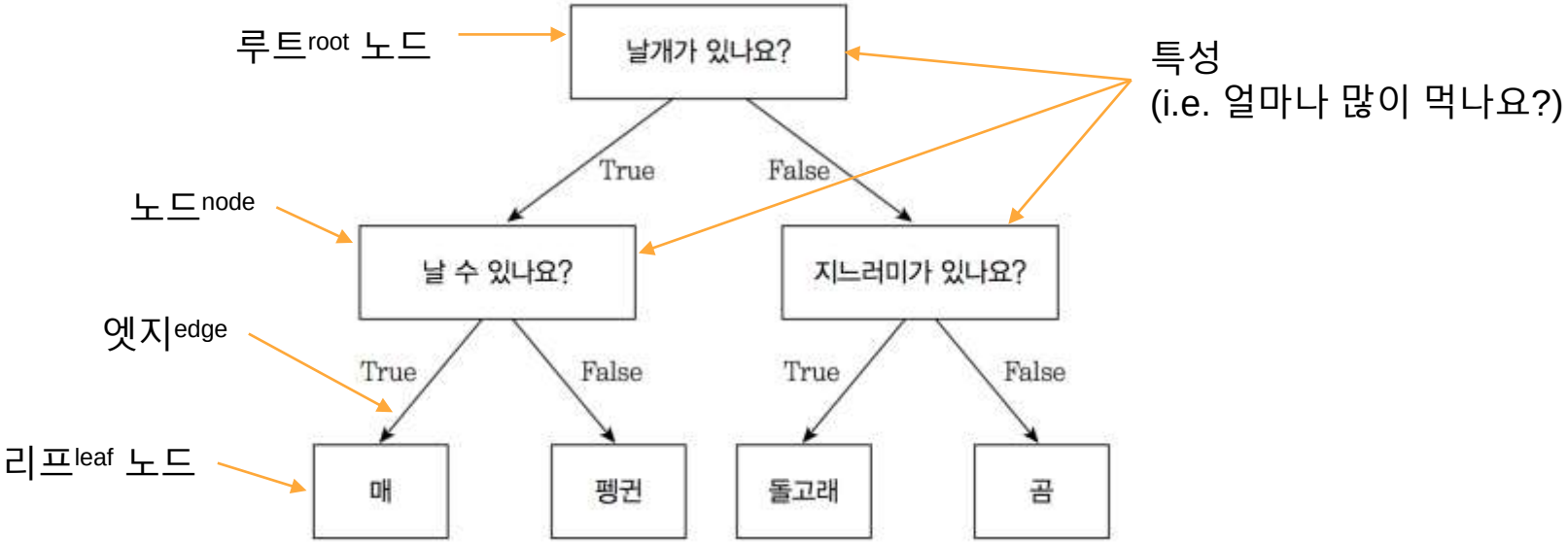
희소한 고차원 데이터셋에 잘 작동하고 매개변수에 민감하지 않습니다.

결정 트리

결정 트리decision tree

분류와 회귀에 널리 사용됩니다.

결정에 다다르기 위해 예/아니오 질문을 이어나가면서 학습하는 이진 트리입니다.

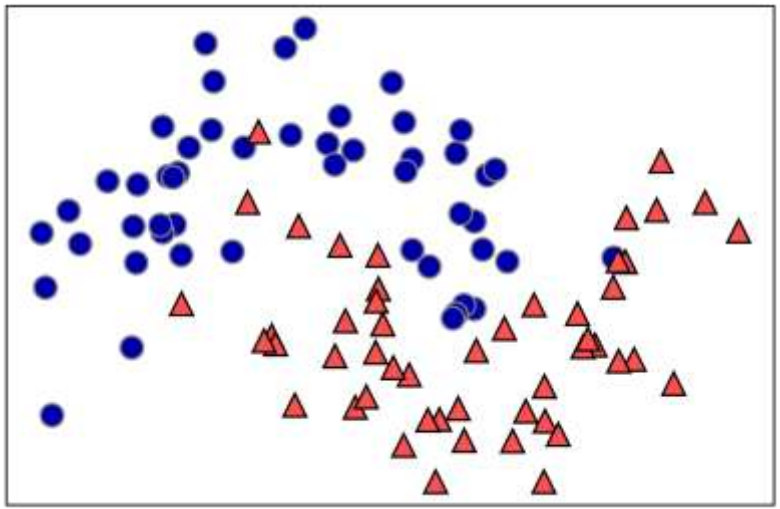


결정 트리 만들기

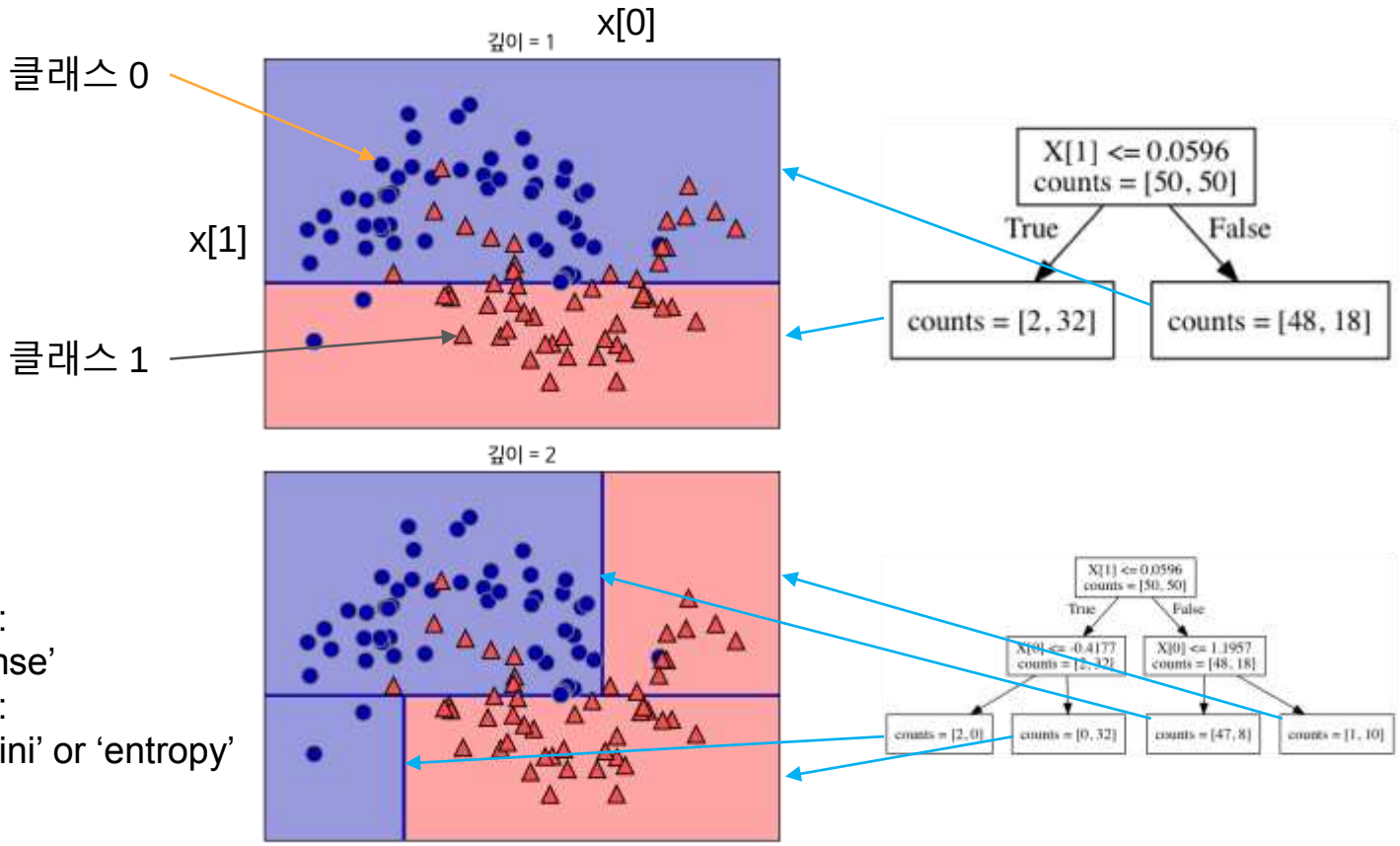
sklearn.datasets.make_moons(): two_moons 데이터셋

정답에 가장 빨리 도달하는 yes/no 질문(테스트)을 학습합니다.

연속적인 데이터셋에서는 “특성 *i* 가 *a* 보다 큰가?”와 같은 형태가 됩니다.



결정 트리 훈련

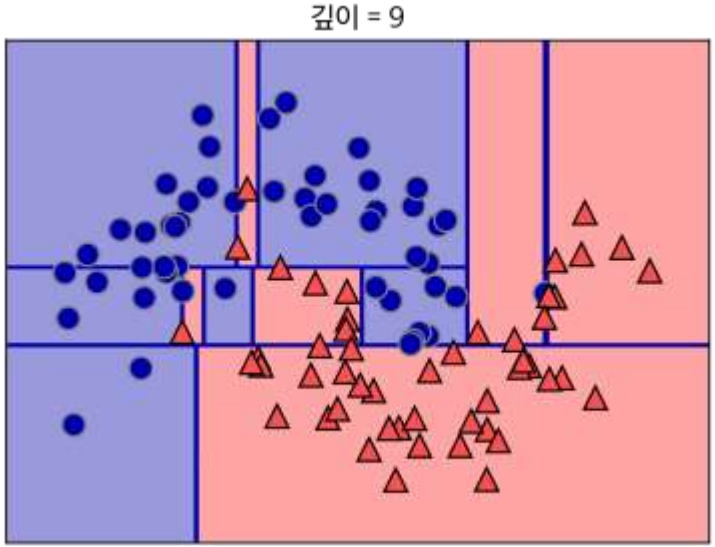


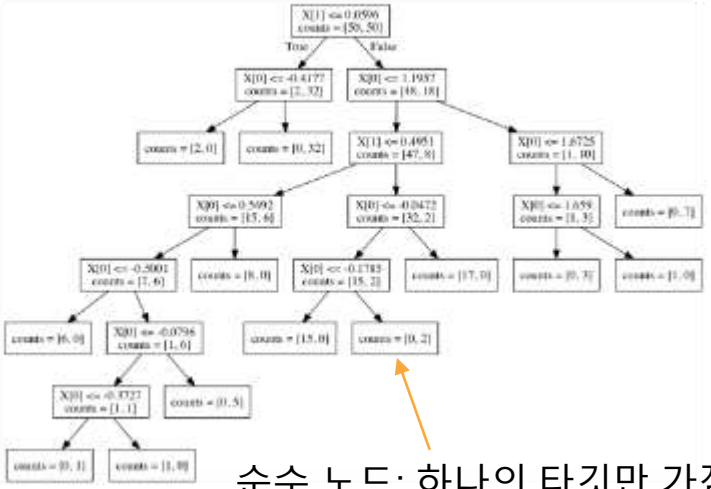
* 분할 기준
  - 회귀 문제:
  criterion='mse'
  - 분류 문제:
  criterion='gini' or 'entropy'

결정 트리 예측

하나의 특성을 사용하여 데이터를 둘로 나누므로 항상 축에 평행하게 분리됩니다.

학습은 순수노드만 남을 때까지 분할이 반복되고 새로운 데이터 포인트가 속한 분할 영역이 예측(다수의 타깃 혹은 노드의 평균값)이 됩니다.





순수 노드: 하나의 타깃만 가짐

복잡도 제어

모든 리프가 순수 노드가 될 때 까지 진행하면 복잡해지고 과대적합됩니다.

순수 노드로만 이루어진 트리는 훈련 데이터를 100% 정확히 맞추므로 일반화 성능이 낮습니다.

사전 가지치기pre-prunning: 트리 생성을 미리 중단

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

사후 가지치기post-prunning: 트리를 만든 후 노드를 삭제하거나 병합합니다.

DecisionTreeRegressor, DecisionTreeClassifier는 사전 가지치기만 지원합니다.

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

모든 리프노드가 순수노드

```
훈련 세트 정확도: 1.000
테스트 세트 정확도: 0.937
```

과대 적합

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)))
```

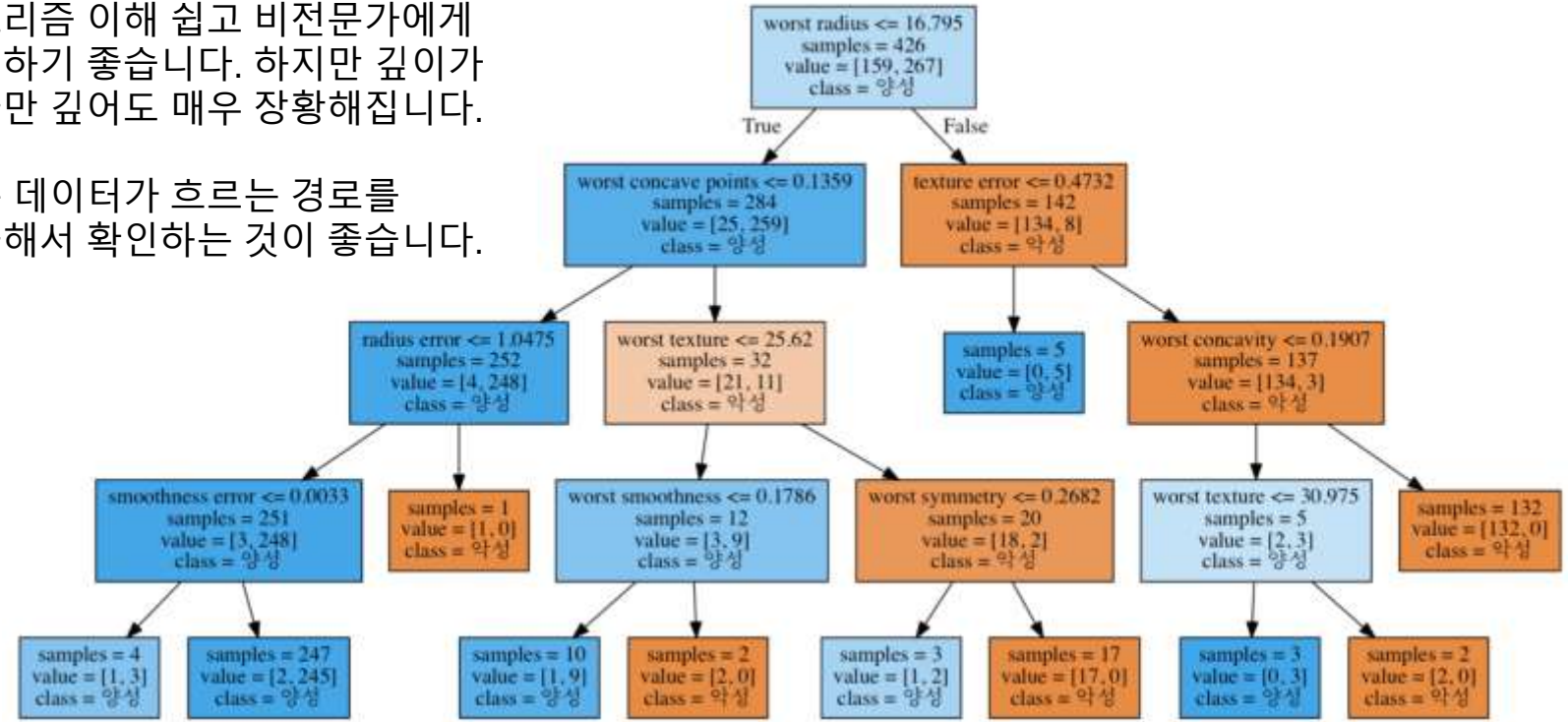
트리의 최대 깊이 4로 제한

```
훈련 세트 정확도: 0.988
테스트 세트 정확도: 0.951
```

결정 트리 분석

알고리즘 이해 쉽고 비전문가에게 설명하기 좋습니다. 하지만 깊이가 조금만 깊어도 매우 장황해집니다.

많은 데이터가 흐르는 경로를 집중해서 확인하는 것이 좋습니다.

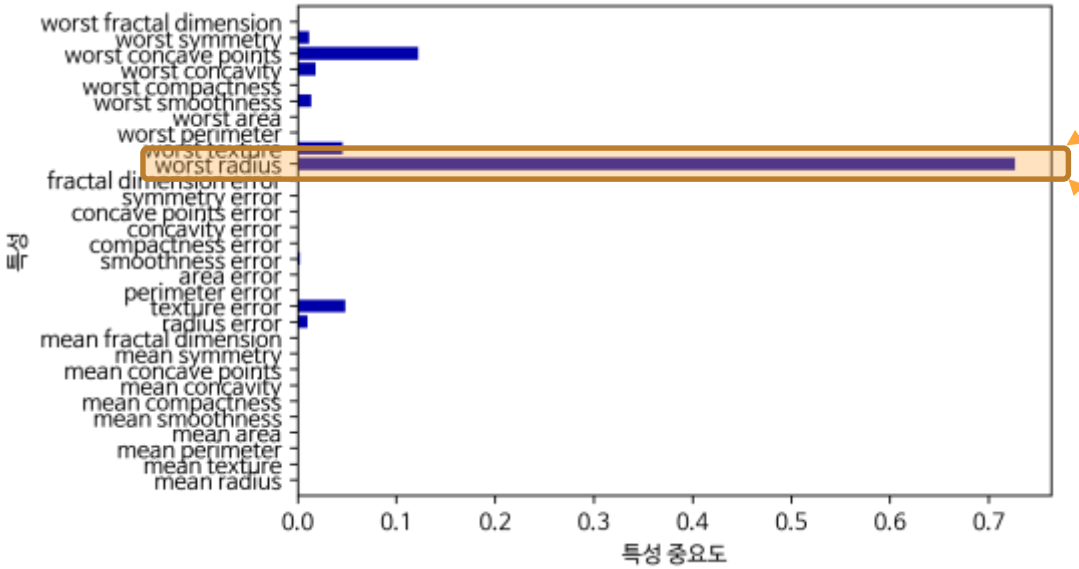


특성 중요도feature importance

0(예측에 사용되지 않음)과 1(완벽하게 예측) 사이의 값으로 전체 합은 1입니다.

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

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

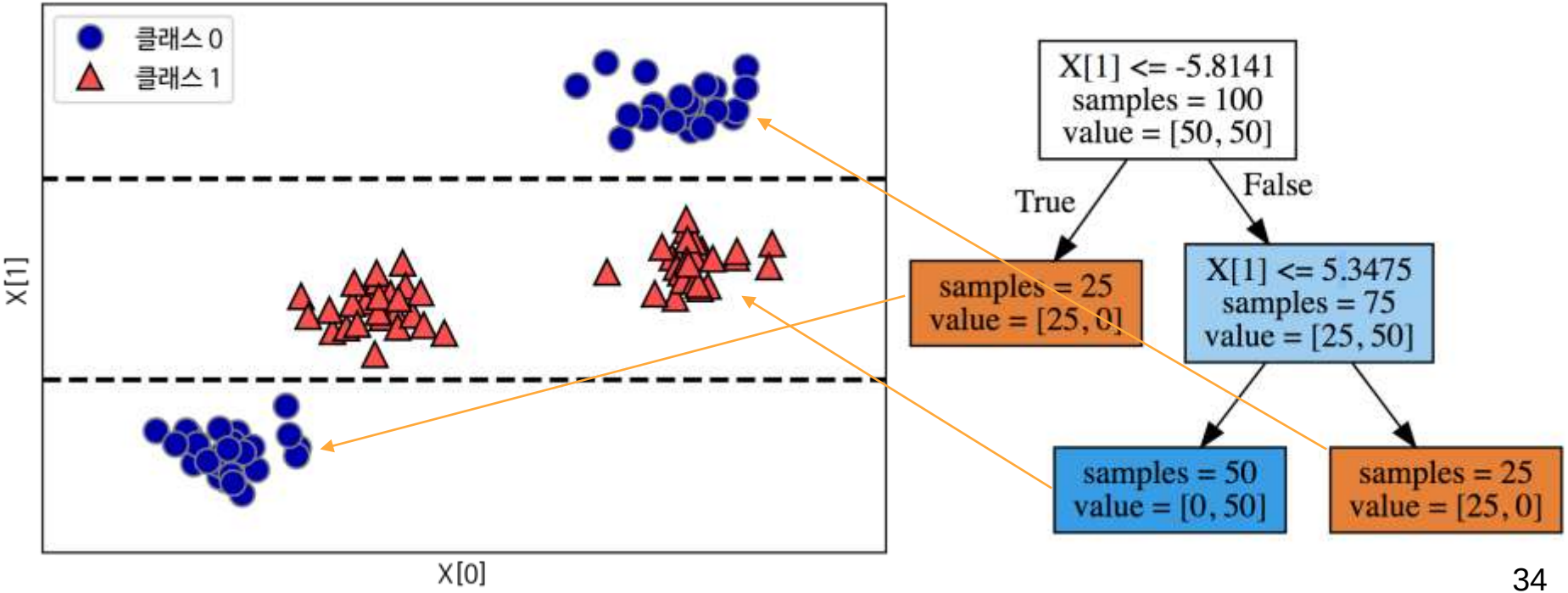


트리 그래프의 첫번째 노드로 활용 (악성 또는 양성의 의미인지 알 수 없음)

다른 특성과 동일한 정보를 가지고 있을 수 있음

특성과 클래스 사이의 관계

X[1]의 값과 출력과의 관계가 단순히 비례/반비례하지 않습니다.



결정 트리 - 회귀

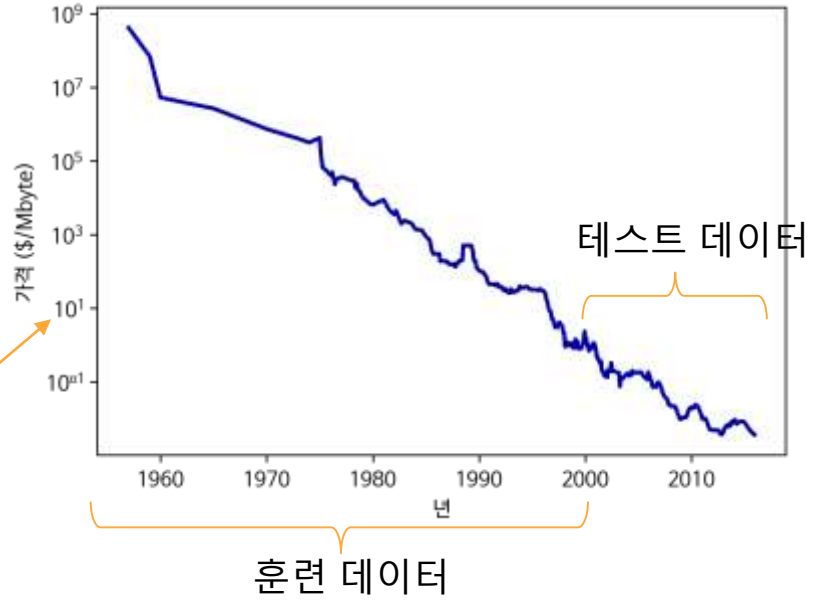
sklearn.tree.DecisionTreeRegressor

훈련 데이터 범위 밖을 예측하는 외삽extrapolation이 불가능합니다.

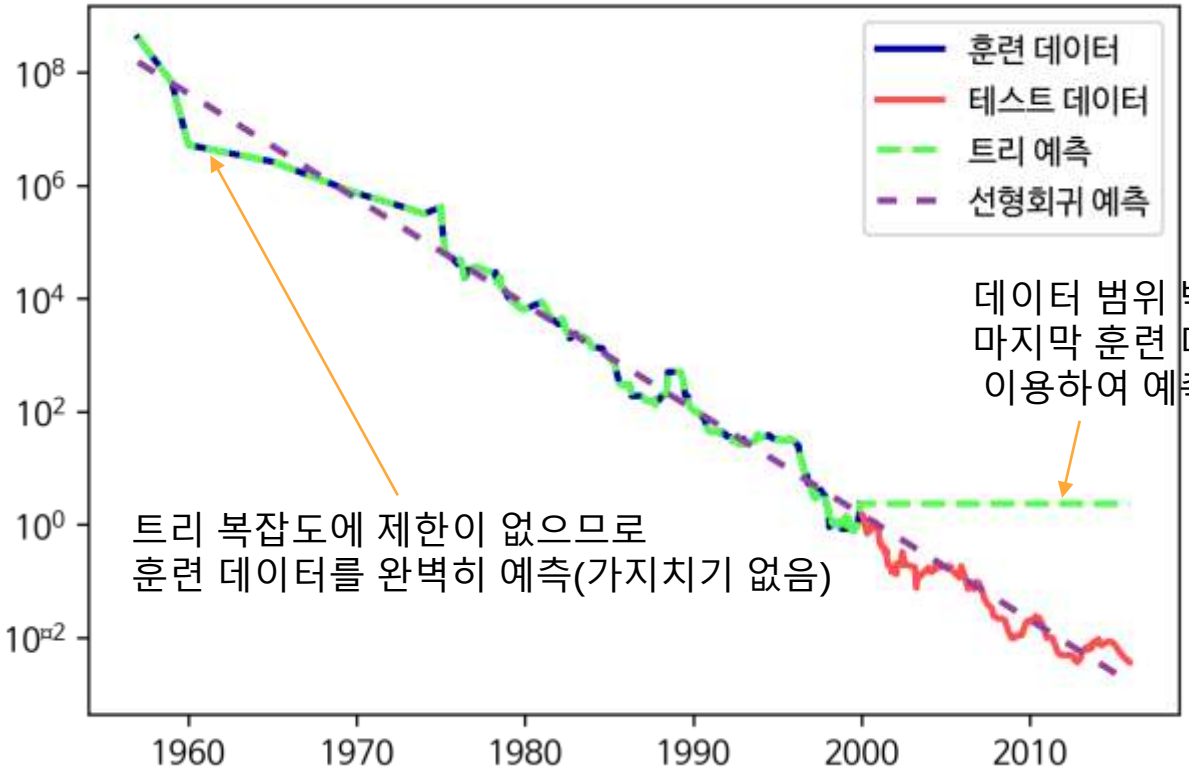
스케일에 영향 받지 않습니다.

e.g. 컴퓨터 메모리 가격 데이터셋

선형회귀와 비교하기 위해
로그 스케일로 바꾸었습니다.



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 쉽습니다.

특성이 각각 처리되므로 데이터 스케일에 구애 받지 않습니다.

- 정규화나 표준화 같은 전처리 과정이 필요 없습니다.
- 이진 특성이나 연속적인 특성이 혼합되어 있어도 가능합니다.

단점: 과대적합 가능성 높아 일반화 성능이 좋지 않습니다.

랜덤 포레스트

랜덤 포레스트random forest

여러 결정 트리를 묶어 과대적합을 회피할 수 있는 앙상블 방법 중 하나입니다.

예측을 잘 하는(과대적합된) 트리들를 평균내어 과대적합을 줄입니다.

랜덤 포레스트는 트리 생성시 무작위성을 주입합니다.

- 트리 생성시 데이터 중 일부를 무작위로 선택합니다.
- 노드 분할 시 무작위로 후보 특성을 선택합니다.

sklearn.ensemble.RandomForestClassifier, RandomForestRegressor의 n_estimators 매개변수로 트리의 개수를 지정(기본값 10)합니다.

모든 트리의 예측을 만든 후, 회귀는 각 예측 값의 평균, 분류는 예측 확률의 평균(약한 투표 전략)을 합니다.

무작위성

부트스트랩 샘플boostrap sample

n 개의 훈련 데이터에서 무작위로 추출해 n 개의 데이터셋을 만듦
중복 추출이 가능하며 하나의 부트스트랩 샘플엔 대략 1/3 정도 샘플이 누락됨

(100개 샘플 중 하나가 선택되지 않을 확률을 100번 반복 = $\left(\frac{99}{100}\right)^{100}$ = 0.366)

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

노드 분할에 사용할 **후보 특성을 랜덤하게 선택**(max_features 매개변수)합니다.

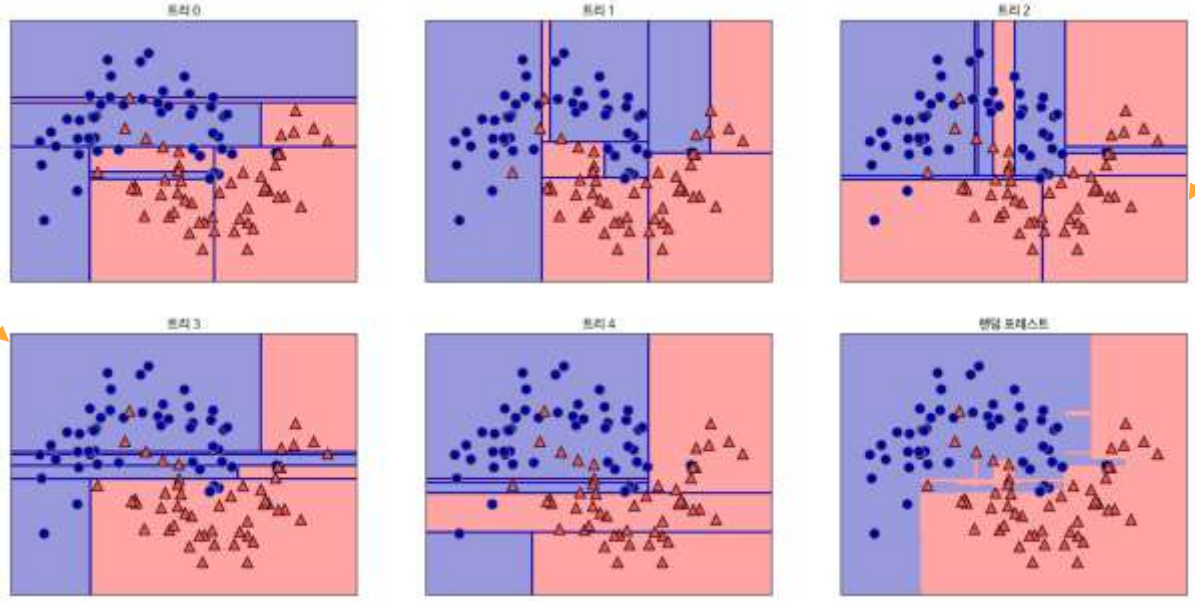
max_features → n_features : 모든 특성을 사용. 비슷한 트리들이 생성.
max_features → 1 : 노드 분할을 무작위로 수행. 서로 많이 다르고 깊은 트리 생성.

→ 랜덤 포레스트의 트리가 모두 다르게 생성됨

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)
```

각 결정 트리는
서로 많이 다릅니다



forest.estimators_

앙상블된 결정경계

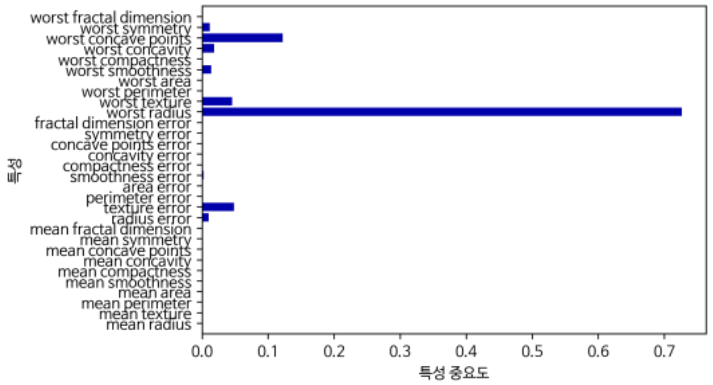
cancer 데이터셋에 적용

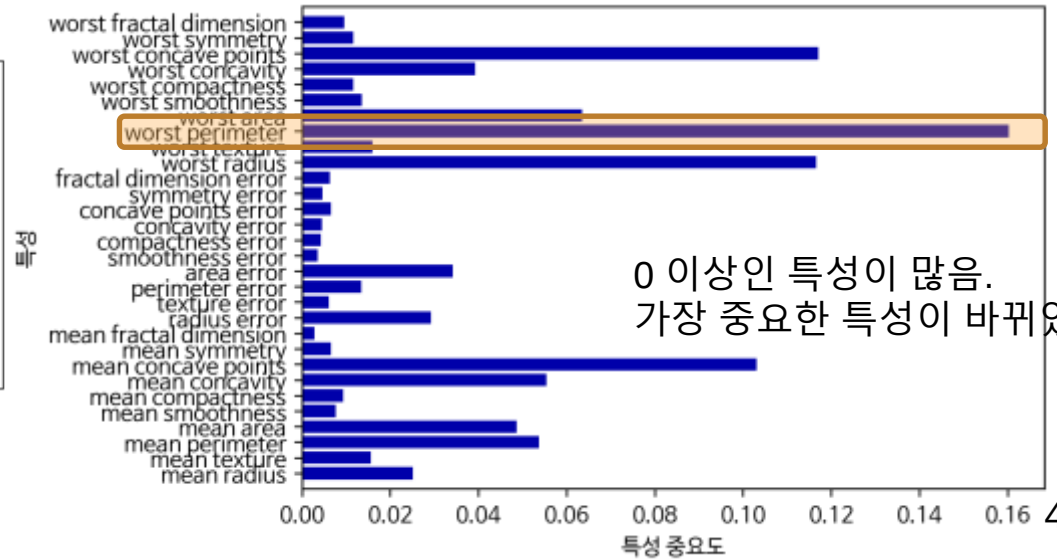
```
In [71]: X_train, X_test, y_train, y_test = train_test_split(
             cancer.data, cancer.target, random_state=0)
         forest = RandomForestClassifier(n_estimators=100, random_state=0)
         forest.fit(X_train, y_train)

         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
```

단일 결정트리
훈련 세트: 1.0
테스트 세트: 0.937

특별히 매개변수 튜닝을 하지 않아도 좋은 성능을 냅니다.





0 이상인 특성이 많음.
가장 중요한 특성이 바뀌었음

장단점과 매개변수

장점

회귀와 분류에서 가장 널리 사용되는 알고리즘입니다.(설명하기는 어렵습니다)
뛰어난 성능을 내며 매개변수 튜닝 부담이 적고 데이터 스케일 불필요합니다.
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 있습니다.
차원이 크고 희소한 데이터에 성능 안좋습니다(e.g. 텍스트 데이터) →선형모델
선형 모델 보다 메모리 사용량이 많고 훈련과 예측이 느립니다.

매개변수

n_estimators(트리 개수, 메모리와 훈련시간 고려), max_features(후보 특성의 개수)
max_features 기본값, 회귀일 때는 n_features, 분류일 때는 sqrt(n_features)
n_estimators가 클수록, max_features가 작을수록 과대적합을 줄여 줍니다.
사전가지치기: max_depth, max_leaf_nodes, min_samples_leaf, min_samples_split

그래디언트 부스팅

그래디언트 부스팅Gradient Boosting 회귀

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

회귀와 분류에 모두 사용 가능합니다.

랜덤 포레스트와는 달리 무작위성 대신 사전 가지치기를 강하게 적용합니다.

다섯 이하의 얕은 트리(약한 학습기)를 사용하여 이전 트리의 오차를 보완하도록 다음 트리 생성합니다.

회귀 : 최소제곱오차 손실함수, 분류 : 로지스틱 손실함수
경사 하강법gradient descent 사용(**learning_rate** 매개변수 중요, 기본값 0.1)

머신러닝 경연대회(e.g. 캐글Kaggle)에서 많이 사용됩니다.

랜덤 포레스트 보다 매개변수에 조금 더 민감하지만 조금 더 높은 성능을 냅니다.

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
```

n_estimators=100, max_depth=3
learning_rate=0.1

과대적합

트리 깊이 제한

```
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)))
```

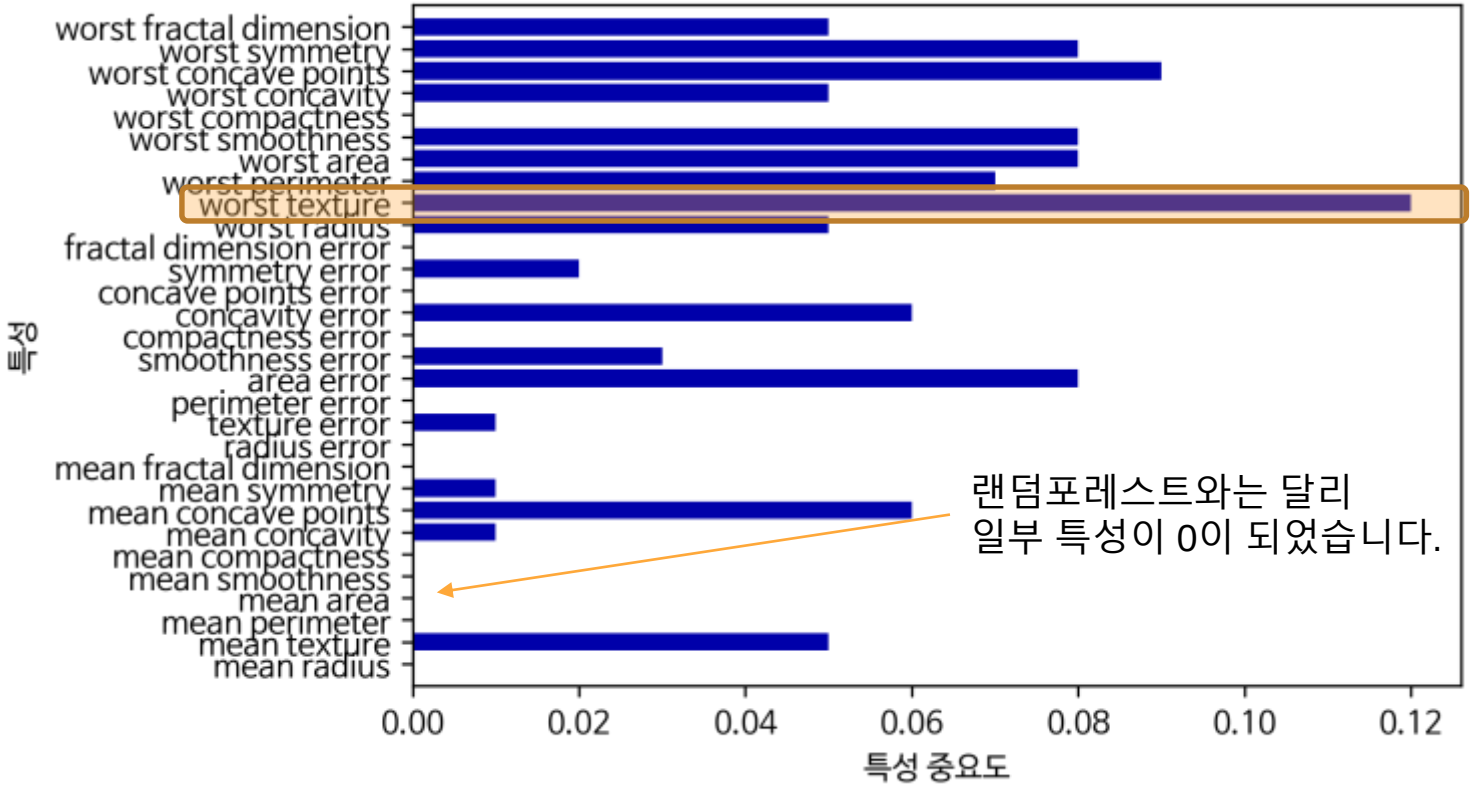
```
훈련 세트 정확도: 0.991
테스트 세트 정확도: 0.972
```

학습률 감소

```
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
```

그래디언트 부스팅의 특성 중요도



장단점과 매개변수

장점

랜덤 포레스트보다 예측 속도가 빨라야하거나 성능을 더 쥐어짜야 할 때 사용합니다.
특성 스케일 조정 필요 없고 이진, 연속적인 특성에도 사용 가능합니다.
아주 대규모일 경우 분산처리가 가능한 xgboost가 빠르고 튜닝하기도 쉽습니다.

단점

희소한 고차원 데이터에 잘 작동하지 않습니다.
매개변수에 민감, 훈련 시간이 더 걸립니다.

매개변수

n_estimators, learning_rate, max_depth(<=5)
learning_rate을 낮추면 복잡도가 낮아져 성능을 올리려면 더 많은 트리가 필요합니다.
랜덤 포레스트와는 달리 n_estimators를 크게하면 과대적합 가능성 높아집니다.
가용 메모리와 시간안에서 n_estimators를 맞추고 learning_rate으로 조절합니다.