



파이썬 머신러닝 완벽가이드 4장 Part 1

2팀 정지은, 강정인, 신유진

목차

#4.1 분류의 개요

#4.2 결정 트리

#4.3 앙상블 학습

#4.4 랜덤 포레스트



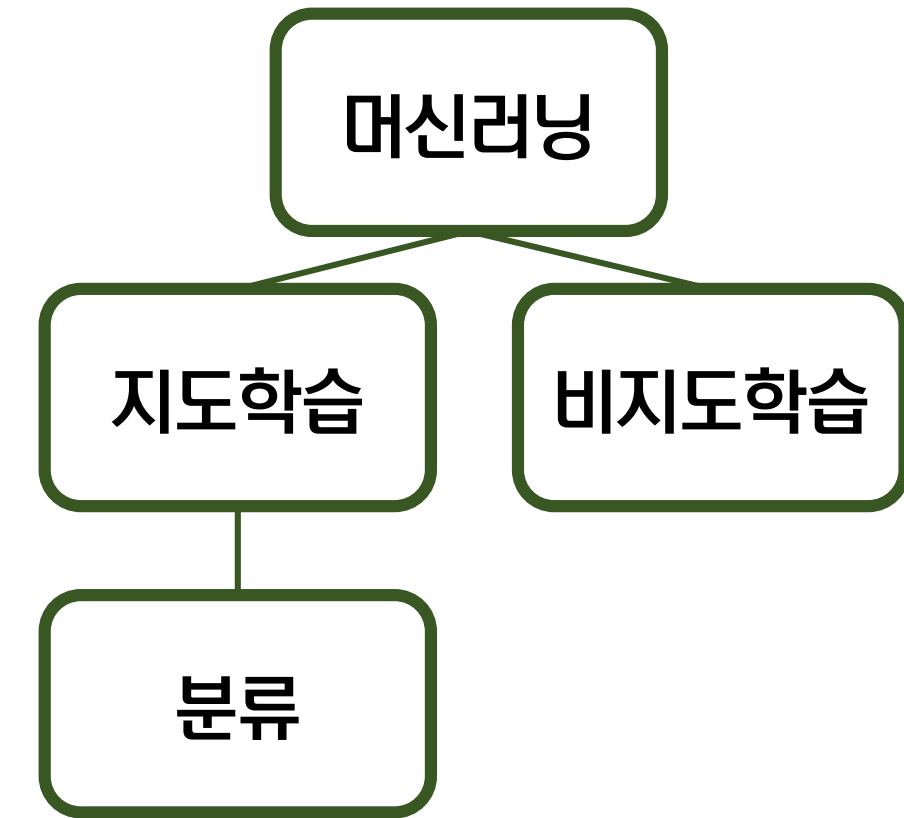
4.1 분류의 개요



#4.1 분류(Classification)의 개요

지도학습

- 명시적인 정답이 있는 데이터가 주어진 상태에서 학습하는 머신러닝 방식
- 대표적인 유형 : 분류



분류

- 학습 데이터로 주어진 데이터의 피처와 레이블값(결정 값, 클래스 값)을 머신러닝 알고리즘으로 학습해 모델 생성
 - 생성된 모델에 새로운 데이터 값이 주어졌을 때 미지의 레이블 값을 예측
- => 기존 데이터가 어떤 레이블에 속하는지 패턴을 알고리즘으로 인지한 뒤에 새롭게 관측된 데이터에 대한 레이블을 판별하는 것

#4.1 분류(Classification)의 개요

다양한 머신러닝 알고리즘으로 분류 구현

- 베이즈 통계와 생성 모델에 기반한 나이브 베이즈
- 독립변수와 종속변수의 선형 관계성에 기반한 로지스틱 회귀
- 데이터 균일도에 따른 규칙 기반의 결정 트리
- 개별 클래스 간의 최대 분류 마진을 효과적으로 찾아주는 서포트 벡터 머신
- 근접 거리를 기준으로 하는 최소 근접 알고리즘
- 심층 연결 기반의 신경망
- 서로 다른(또는 같은) 머신러닝 알고리즘을 결합한 앙상블

#4.1 분류(Classification)의 개요

앙상블 방법(Ensemble Method)

- 분류에서 가장 각광을 받는 방법 중 하나
- 정형 데이터의 예측 분석 영역에서 매우 높은 예측 성능으로 인해 많은 분석가와 데이터 과학자들에게 애용되고 있음
- 서로 다른/또는 같은 알고리즘을 단순히 결합한 형태도 있음 (대부분 동일한 알고리즘을 결합)
- 일반적으로는 배깅(Bagging)과 부스팅(Boosting) 방식으로 나뉨
- 근래의 앙상블 방법은 부스팅 방식으로 지속해서 발전



#4.1 분류(Classification)의 개요

배깅

- 랜덤 포레스트
 - 배깅 방식의 대표
 - 뛰어난 예측 성능, 상대적으로 빠른 수행 시간, 유연성 등으로 많은 분석가가 애용하는 알고리즘

부스팅

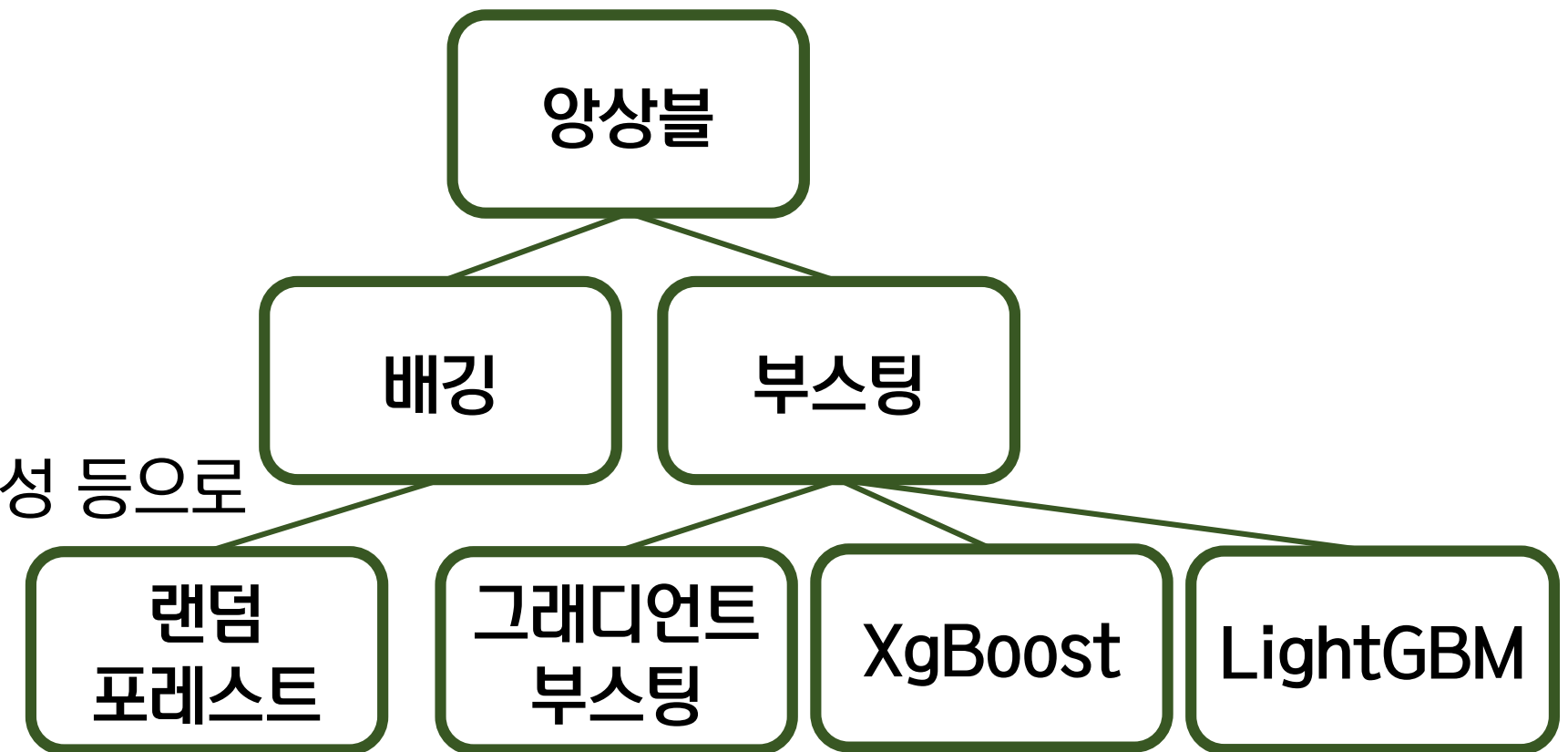
그래디언트 부스팅(Gradient Boosting)

- (+) 뛰어난 예측 성능 (-) 수행 시간이 너무 오래 걸림 -> 최적화 모델 튜닝 어려움

XgBoost(eXtra Gradient Boost)와 LightGBM

- 기존 그래디언트 부스팅의 예측 성능을 한 단계 발전 + 수행 시간 단축시킨 알고리즘

⇒ 정형 데이터 분류 영역에서 가장 활용도가 높은 알고리즘으로 자리 잡음



#4.1 분류(Classification)의 개요

이 장에서 배울 내용

- 앙상블 방법의 개요
- (전통적 앙상블 기법) 랜덤 포레스트, 그래디언트 부스팅
- (부스팅 계열 최신 기법) XGBoost, LightGBM
- 스택킹(Stacking) 기법
- 결정 트리

#4.1 분류(Classification)의 개요

결정 트리

- 앙상블의 기본 알고리즘
- **장점(+)**
 - 매우 쉽고 유연하게 적용될 수 있음
 - 데이터의 스케일링, 정규화 등 사전 가공 영향이 매우 적음
- **단점(-)**
 - 예측 성능 향상을 위해 복잡한 규칙 구조를 가짐 -> **과적합(overfitting)** 발생 -> 예측 성능 저하
 - 위 단점은 **앙상블 기법에서 오히려 장점으로 작용**
 - 앙상블 : 매우 많은 여러 개의 약한 학습기(예측 성능이 상대적으로 떨어지는 학습 알고리즘)를 결합 -> 확률적 보완과 오류가 발생한 부분에 대한 가중치를 계속 업데이트하며 예측 성능 향상
 - 결정 트리가 좋은 약한 학습기가 됨

4.2 결정 트리 (개념)



#4.2 결정 트리

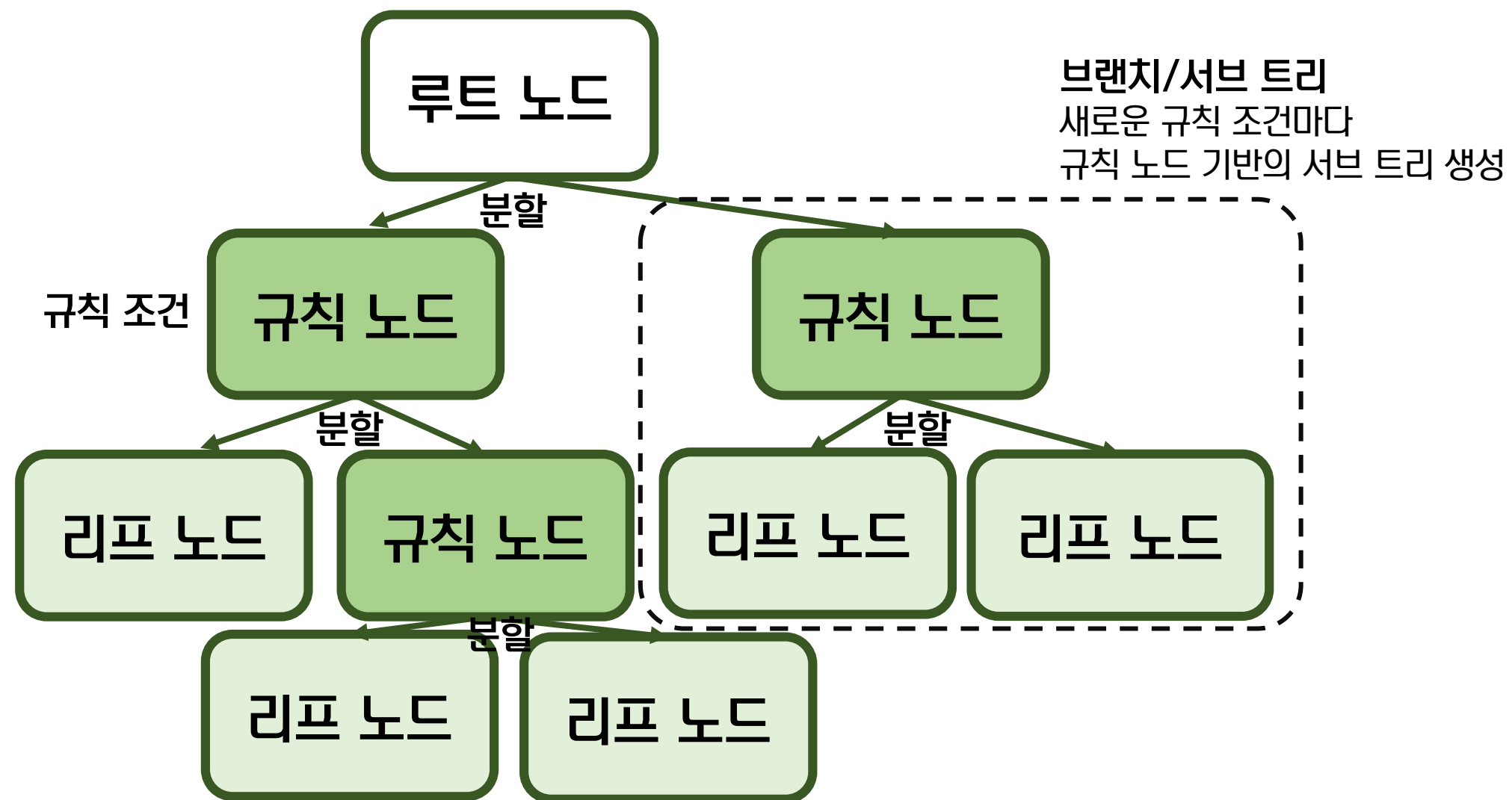
결정 트리(Decision Tree)

- ML 알고리즘 중 직관적으로 이해하기 쉬운 알고리즘
- 학습 -> 데이터에 있는 규칙 자동으로 찾아냄 -> 트리 기반의 분류 규칙 만듦
- 규칙을 가장 쉽게 표현하는 방법 : if/else 기반으로 나타내는 것
- 데이터의 어떤 기준을 바탕으로 규칙을 만들어야 가장 효율적인 분류가 될 것인가? -> 알고리즘 성능 크게 좌우함

#4.2 결정 트리

결정 트리의 구조

- 규칙 노드(Decision Node) : 규칙 조건이 됨
- 리프 노드(Leaf Node) : 결정된 클래스 값
- 서브 트리(Sub Tree) : 새로운 규칙 조건마다 생성됨
- 데이터 세트 피처가 결합해 규칙 조건을 만들 때마다 규칙 노드가 만들어짐



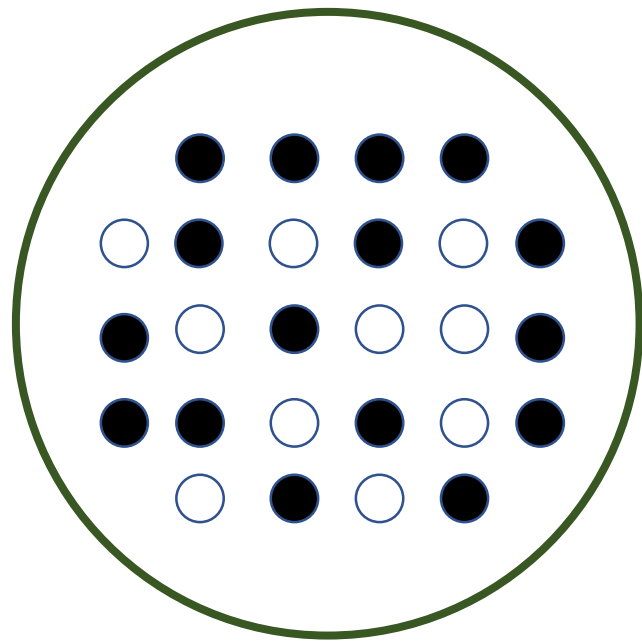
#4.2 결정 트리

결정 트리의 성능

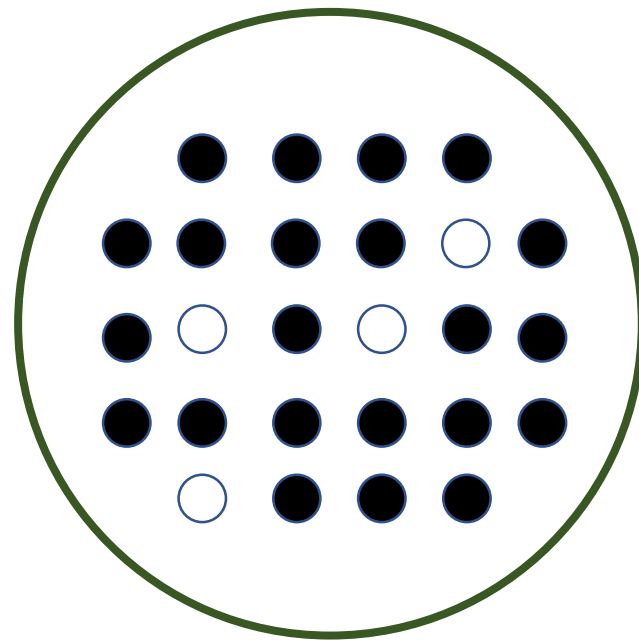
- 많은 규칙이 있다? = 분류 결정 방식이 더욱 복잡해진다 -> 과적합으로 이어짐
- 분류의 깊이가 깊어질수록 결정 트리의 예측 성능 저하 가능성 높음
- 적은 결정 노드로 높은 예측 정확도 => 데이터 분류할 때 최대한 많은 데이터 세트가 해당 분류에 속할 수 있도록 결정 노드의 규칙 정해야 함
- 어떻게 트리를 분할할 것인지가 중요 (최대한 균일한 데이터 세트 구성하도록 분할)

#4.2 결정 트리

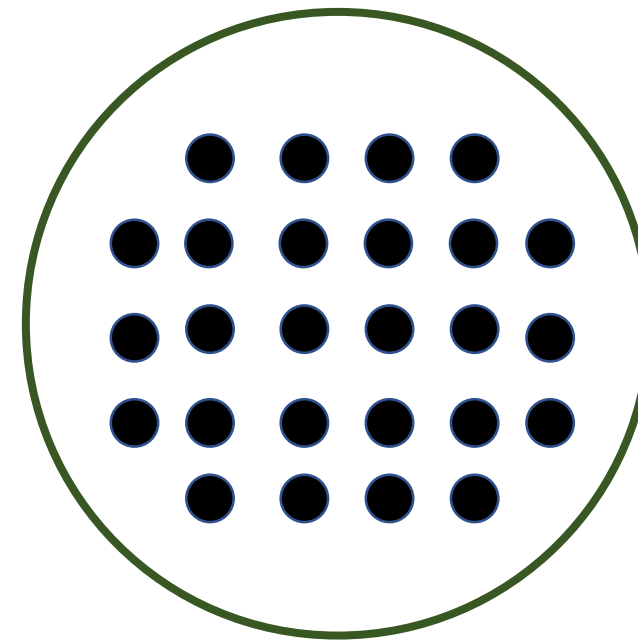
균일한 데이터 세트의 의미



데이터 세트 A



데이터 세트 B



데이터 세트 C

- **균일도** : $C > B > A$ (모두 검은 공으로 구성된 C가 가장 높음, 흰 공 검은 공 반반인 A가 가장 낮음)
- 균일도는 데이터를 구분하는 데 필요한 정보의 양에 영향을 미침
 - C에서 하나의 데이터를 뽑았을 때? 검은 공이라고 쉽게 예측
 - A에서 하나의 데이터를 뽑았을 때? 균일도가 낮고 혼잡도가 높아 데이터 판단 정보가 더 필요함

#4.2 결정 트리

효율적인 분류 방식

- 결정 노드 : 정보 균일도가 높은 데이터 세트를 먼저 선택할 수 있도록 규칙 조건 만들기
 - 1. 정보 균일도가 데이터 세트로 쪼개질 수 있도록 조건을 찾아 서브 데이터 세트 만들기
 - 2. 서브 데이터 세트에서 균일도가 높은 자식 데이터 세트 쪼개는 방식을 자식 트리로 내려가면서 반복하는 방식으로 데이터 예측
- 예) 30개의 레고 블록 '형태' 속성 : 동그라미, 네모, 세모 / '색깔' 속성 : 노랑, 빨강, 파랑
노랑색 블록 : 모두 동그라미 / 빨강과 파랑 블록 : 동그라미, 네모, 세모 섞여 있음
=> 가장 첫 번째로 만들어져야 하는 규칙 조건 : if 색깔== '노란색' (::노란색 블록은 모두 동그라미)

#4.2 결정 트리

정보의 균일도를 측정하는 대표적인 방법

1. 엔트로피를 이용한 정보 이득 지수

- **엔트로피**: 주어진 데이터 집합의 **혼잡도** (A의 경우 엔트로피가 높고, C의 경우 엔트로피가 낮음)
- 즉, 서로 다른 값이 섞여 있으면 엔트로피가 높고 같은 값이 섞여 있으면 엔트로피가 낮음
- **정보 이득 지수**: $1 - \text{엔트로피 지수}$
- 정보 이득 지수가 높은 속성을 기준으로 결정 트리를 분할함

2. 지니 계수

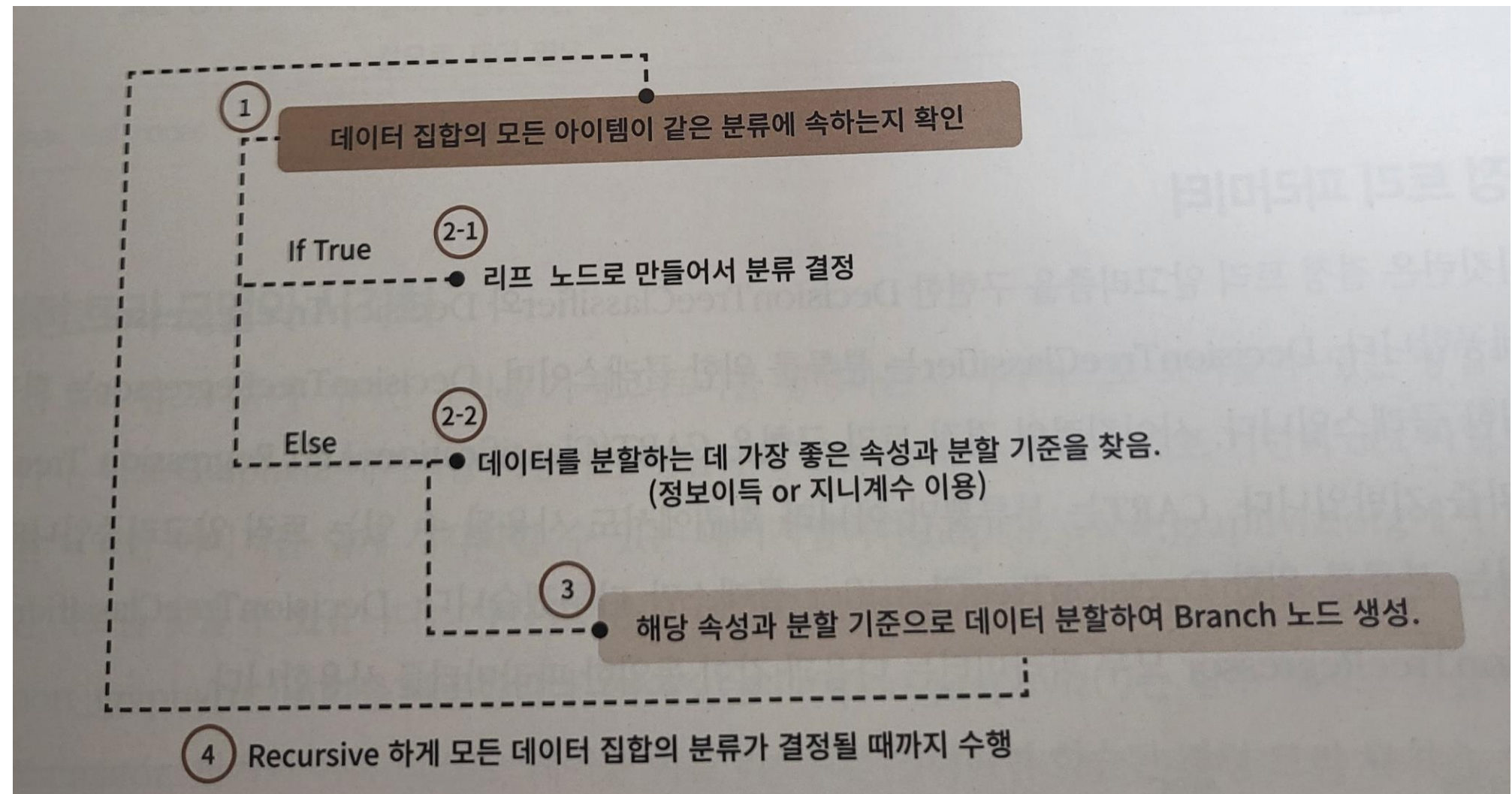
- 경제학에서 불평등 지수를 나타낼 때 사용하는 계수
- 머신러닝에 적용될 때는 지니 계수가 낮을수록 데이터 균일도가 높은 것으로 해석
- 지니 계수가 낮은 속성을 기준으로 분할

⇒ 데이터 균일도가 높은 = 지니 계수가 낮은 = 정보 이득 지수가 높은 속성을 기준으로 트리 분할

#4.2 결정 트리

결정 트리 알고리즘

- 사이킷런에서 구현한 DecisionTreeClassifier -> 지니 계수를 이용해 데이터 세트 분할
- 일반적인 알고리즘
 - 데이터 세트 분할하는 데 가장 좋은 조건(=정보 이득 높음 =지니 계수 낮음)을 찾아 자식 트리 노드에 걸쳐 반복적으로 분할
 - 데이터가 모두 특정 분류에 속하게 되면 분할 멈추고 분류 결정



#4.2 결정 트리

결정 트리 모델의 특징

장점(+)

- 정보의 '균일도' 라는 룰을 기반으로 하여 알고리즘이 쉽고 직관적
- 결정 트리가 룰이 매우 명확하고, 이에 기반해 어떻게 규칙 노드와 리프 노드가 만들어지는 지 알 수 있음
- 시각화로 표현까지 가능
- 특별한 경우를 제외하고 전처리 작업 필요 없음

단점(-)

- 과적합으로 정확도가 떨어짐
- 피처가 많고 균일도가 다양하게 존재할수록 트리의 깊이가 커지고 복잡해짐
- 복잡한 학습 모델은 결국 실제 상황에 유연하게 대처할 수 없음, 예측 성능 떨어짐

=> 트리의 크기를 사전에 제한하는 것이 오히려 성능 튜닝에 도움이 됨

#4.2 결정 트리

결정 트리 파라미터

사이킷런은 결정 트리 알고리즘을 구현한 DecisionTreeClassifier와 DecisionTreeRegressor 클래스 제공

DecisionTreeClassifier

- 분류를 위한 클래스

DecisionTreeRegressor

- 회귀를 위한 클래스

사이킷런의 결정 트리 구현 : CART(Classfication And Regression Trees) 알고리즘 기반

- 분류뿐만 아니라 회귀에서도 사용될 수 있는 트리 알고리즘
- DecisionTreeClassifier와 DecisionTreeRegressor 모두 동일한 파라미터 사용

파라미터 명	설명
min_sample_spell	노드를 분할하기 위한 최소한의 샘플 데이터 수로 과적합 제어에 사용
min_samples_leaf	말단 노드(Leaf)가 되기 위한 최소한의 샘플 데이터 수, 과적합 제어 용도
max_features	최적의 분할을 위해 고려할 최대 피쳐 개수, 디폴트는 None으로 모든 피쳐 사용해 분할 수행
max_depth	트리의 최대 깊이 규정, 디폴트는 None으로 완벽하게 클래스 결정 값이 될 때까지 깊이 키우며 분할 수행
max_leaf_nodes	말단 노드(Leaf)의 최대 개수

#4.2 결정 트리

결정 트리 모델의 시각화

결정 트리 모델 시각화 : Graphviz 패키지 사용 (그래프 기반의 dot 파일로 기술된 이미지를 쉽게 시각화)

사이킷런은 `export_graphviz()` API 제공

- 함수 인자로 학습이 완료된 Estimator, 피처의 이름 리스트, 레이블 이름 리스트를 입력 -> 학습된 결정 트리 규칙을 실제 트리 형태로 시각화하여 보여줌

#4.2 결정 트리

붓꽃 데이터 세트 결정 트리 모델 시각화

DecisionTreeClassifier를 이용해 학습 -> 규칙 트리 생성

```
[*]: from sklearn.tree import DecisionTreeClassifier
      from sklearn.datasets import load_iris
      from sklearn.model_selection import train_test_split
      import warnings
      warnings.filterwarnings('ignore')

      #DecisionTree Classifier 생성
      dt_clf = DecisionTreeClassifier(random_state=156)

      #붓꽃 데이터를 로딩하고, 학습과 테스트 데이터 세트로 분리
      iris_data = load_iris()
      X_train, X_test, y_train, y_test = train_test_split(iris_data.data, iris_data.target, test_size=0.2, random_state=11)

      #DecisionTreeClassifier 학습
      dt_clf.fit(X_train, y_train)
```

#4.2 결정 트리

붓꽃 데이터 세트 결정 트리 모델 시각화

`export_graphviz()` : Graphviz가 읽어 들여 그래프 형태로 시각화할 수 있는 출력 파일 생성
인자로 학습이 완료된 `estimator`, output 파일 명, 결정 클래스의 명칭, 피처의 명칭 입력

```
[*]: from sklearn.tree import export_graphviz

#export_graphviz()의 호출 결과로 out_file로 지정된 tree.dot 파일을 생성함.
export_graphviz(dt_clf, out_file="tree.dot", class_name=iris_data.target_names, \
                feature_names = iris_data.feature_names, impurity=True, filled=True)
```

#4.2 결정 트리

붓꽃 데이터 세트 결정 트리 모델 시각화

출력 파일 'tree.dot' 을 Graphviz의 파이썬 래퍼 모듈을 호출해 결정 트리의 규칙 시각화

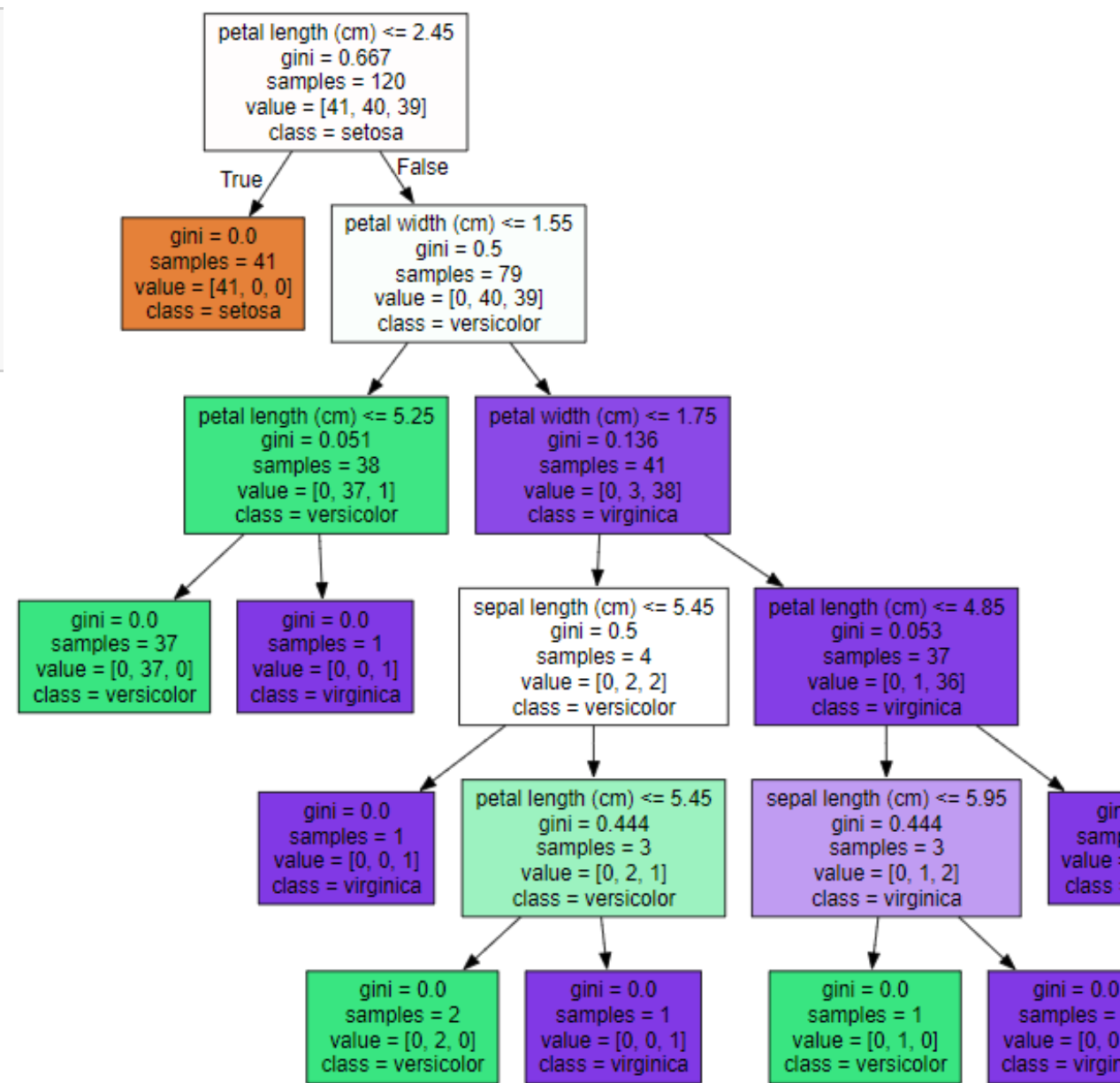
```
import graphviz
```

```
# 위에서 생성된 tree.dot 파일을 Graphviz 읽어서 Jupyter Notebook상에서 시각화
```

```
with open("tree.dot") as f:
```

```
    dot_graph = f.read()
```

```
graphviz.Source(dot_graph)
```



#4.2 결정 트리

예제로 보는 결정 트리 규칙 구성

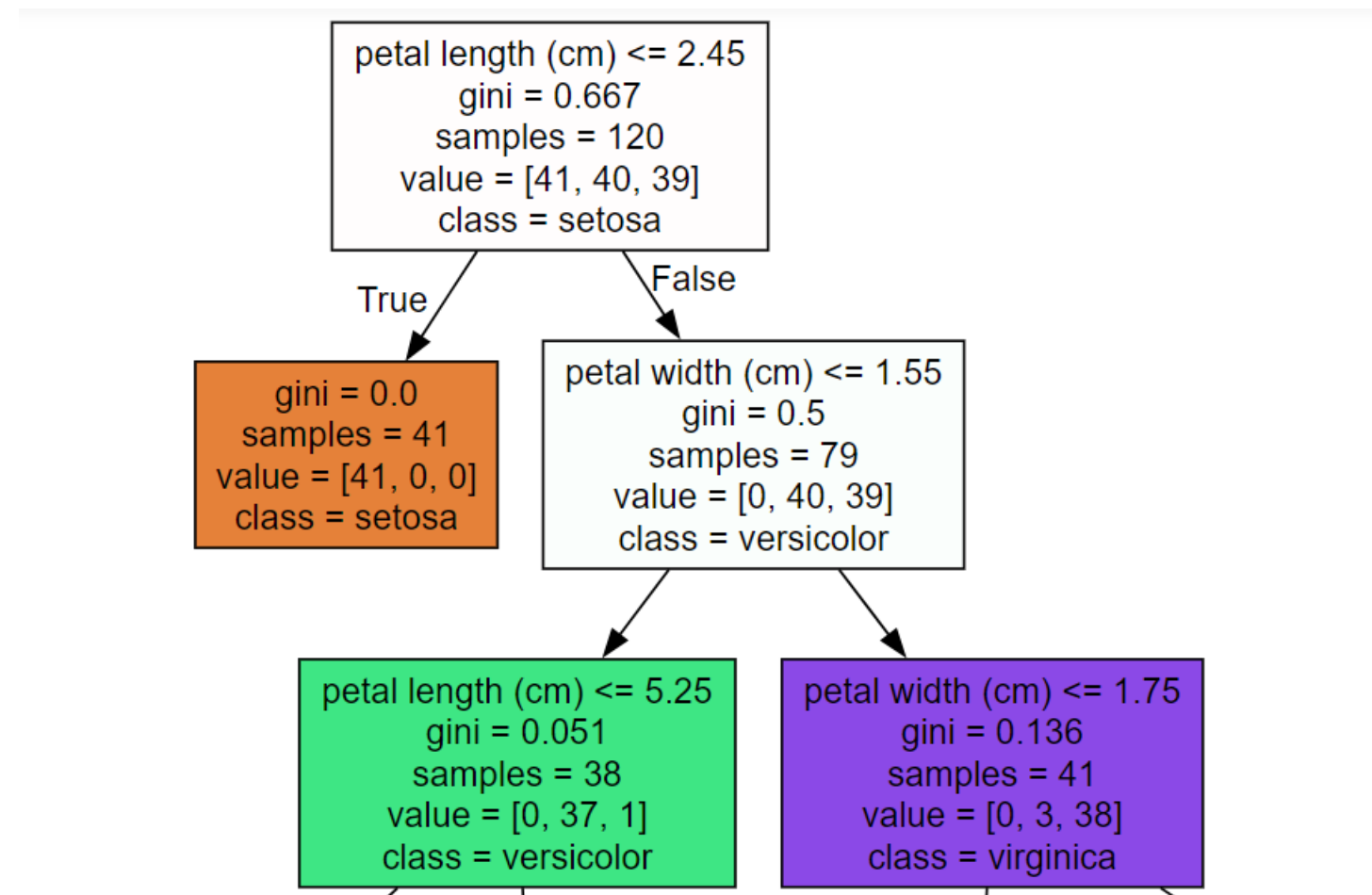
- 리프 노드
 - 더 이상 자식 노드가 없는 노드
 - 최종 클래스(레이블) 값이 결정되는 노드
 - 오직 하나의 클래스 값으로 최종 데이터가 구성되거나 리프 노드가 될 수 있는 하이퍼 파라미터 조건을 충족하면 됨
- 브랜치 노드
 - 자식 노드가 있는 노드
 - 자식 노드를 만들기 위한 분할 규칙 조건을 가짐
 - 노드 내 기술된 지표
 - $\text{petal length(cm)} \leq 2.45$: 피처의 조건이 있는 것은 자식 노드를 만들기 위한 규칙 조건 (없으면 리프 노드)
 - $\text{gini} = 0000$: 주어진 데이터 분포에서의 지니 계수

#4.2 결정 트리

예제로 보는 결정 트리 규칙 구성

- samples : 현 규칙에 해당하는 데이터 건수
- value = [] : 클래스 값 기반의 데이터 건수
- 예) 붓꽃 데이터 세트 클래스 값 : 0, 1, 2로 각각 품종을 가리킴

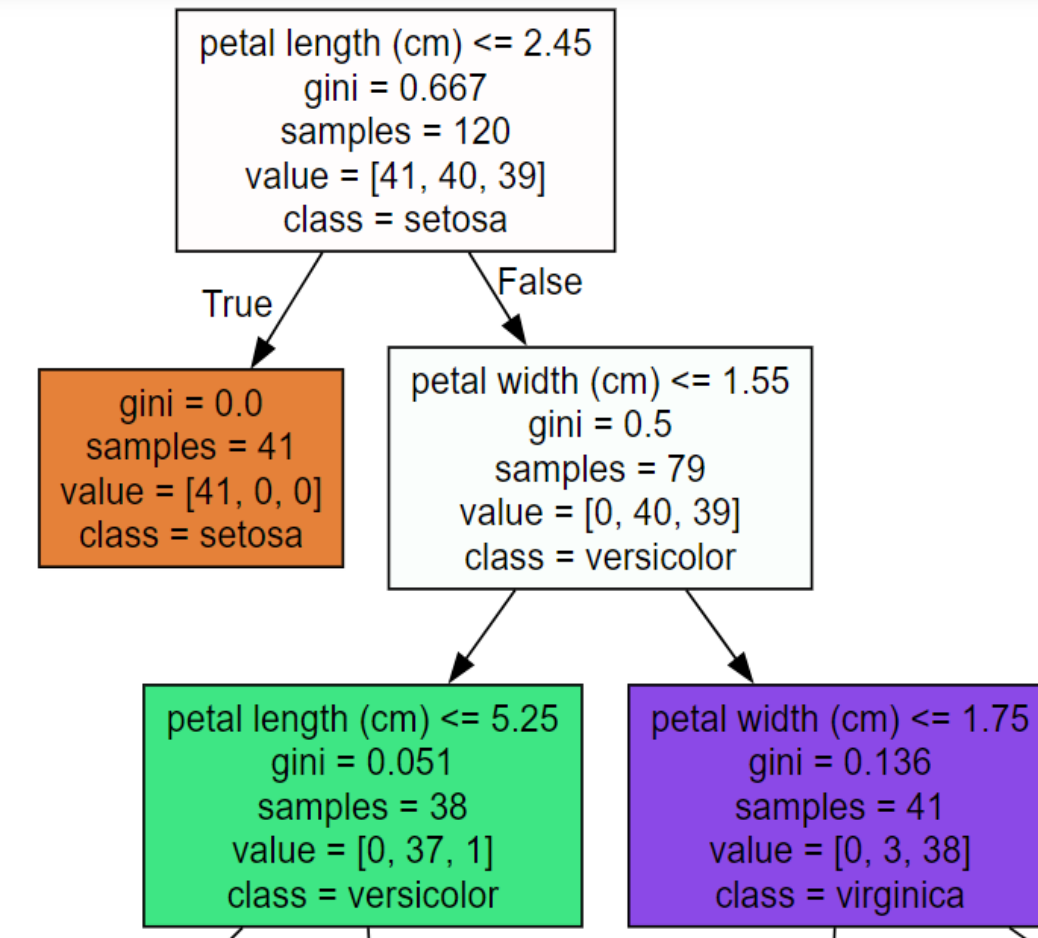
value = [41, 40, 39]라면 1번 품종 41개, 2번 품종 40개, 3번 품종 39개로 데이터가 구성되어 있음



#4.2 결정 트리

예제로 보는 결정 트리 규칙 구성

- 루트 노드 = 1번 노드
 - class = setosa는 하위 노드를 가질 경우 setosa의 개수가 41개로 가장 많다는 의미
- 2번 노드(주황색)
 - 모든 데이터가 setosa로 결정됨 -> 리프 노드가 됨 -> 더 이상의 규칙 만들 필요가 없음
- petal length (cm) ≤ 2.45 가 True인 규칙으로 생성된 리프 노드
- 3번 노드(흰색)
 - petal length (cm) ≤ 2.45 가 False인 규칙 노드
 - 2번 품종, 3번 품종이 40개, 39개로 여전히 지니 계수가 0.5로 높음 -> 다음 자식 브랜치 노드로 분기할 규칙 필요
 - petal width (cm) ≤ 1.55 규칙으로 자식 노드 생성

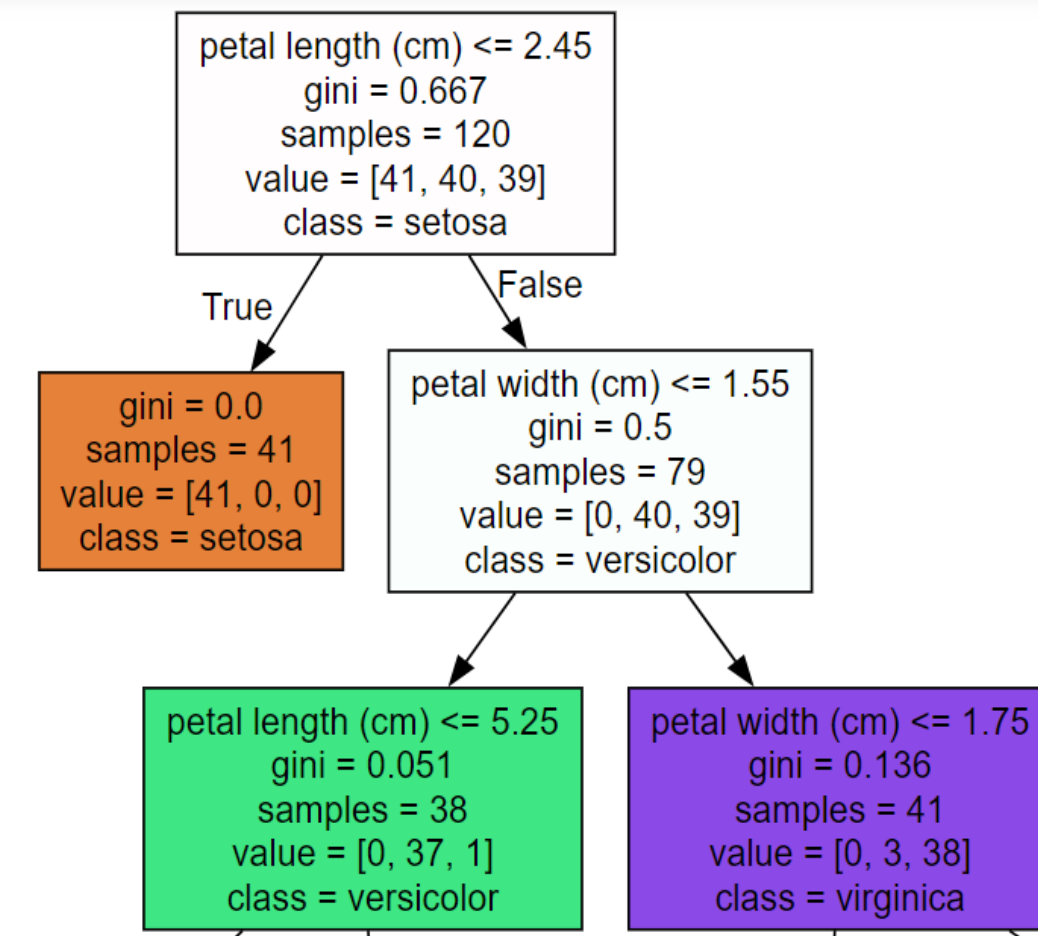


#4.2 결정 트리

예제로 보는 결정 트리 규칙 구성

주황색 0: Setosa, 초록색 1:Versicolor, 보라색 2:Virginica
(색이 짙어질수록 지니 계수가 낮고 해당 레이블에 속하는 샘플 데이터가 많다.)

- 4번 노드(연두색)
 - petal width ≤ 1.55 가 True인 규칙 노드
 - 38개의 데이터 중에 1개를 제외한 데이터가 versicolor
 - 지니 계수는 0.051로 매우 낮으나 여전히 2번 품종과 3번 품종이 혼재 -> petal length ≤ 5.25 라는 새 규칙으로 자식 노드 생성
- 5번 노드(보라색)
 - petal width ≤ 1.55 가 False인 규칙 노드
 - 41개의 데이터 중에 3개를 제외한 데이터가 virginica
 - 지니 계수는 0.136로 매우 낮으나 여전히 2번 품종과 3번 품종이 혼재 -> petal width ≤ 1.75 라는 새 규칙으로 자식 노드 생성
- 각 노드의 색 : 붓꽃 데이터의 레이블 값을 의미



#4.2 결정 트리

예제로 보는 결정 트리 규칙 구성

- 4번 노드(연두색)
 - 38개의 데이터 중에 1개를 제외한 데이터가 versicolor -> versicolor와 virginica를 구분하기 위해 다시 자식 노드 생성
 - ⇒ 규칙 생성 로직을 미리 제어하지 않으면 완벽하게 클래스 값을 구별해내기 위해 트리 노드를 계속해서 만들어 감
 - ⇒ 매우 복잡한 규칙 트리 생성, 모델이 쉽게 과적합 되는 문제 발생
 - ⇒ 하이퍼 파라미터로 복잡한 트리 생성을 막음

petal length (cm) ≤ 5.25
gini = 0.051
samples = 38
value = [0, 37, 1]
class = versicolor

- 결정 트리의 하이퍼 파라미터

- `max_depth` (결정 트리의 최대 트리 깊이 제어)

- 예) 제약 없음 -> `max_depth = 3`으로 깊이 줄이기

- `min_samples_split` (자식 노드를 분할해 만들기 위한 최소한의 샘플 데이터 개수)

- 예) `min_samples_split=4`, `samples = 30`이면 서로 다른 class 값이 있어도 split하지 않음

petal length (cm) ≤ 5.45
gini = 0.444
samples = 3
value = [0, 2, 1]
class = versicolor

sepal length (cm) ≤ 5.95
gini = 0.444
samples = 3
value = [0, 1, 2]
class = virginica

#4.2 결정 트리

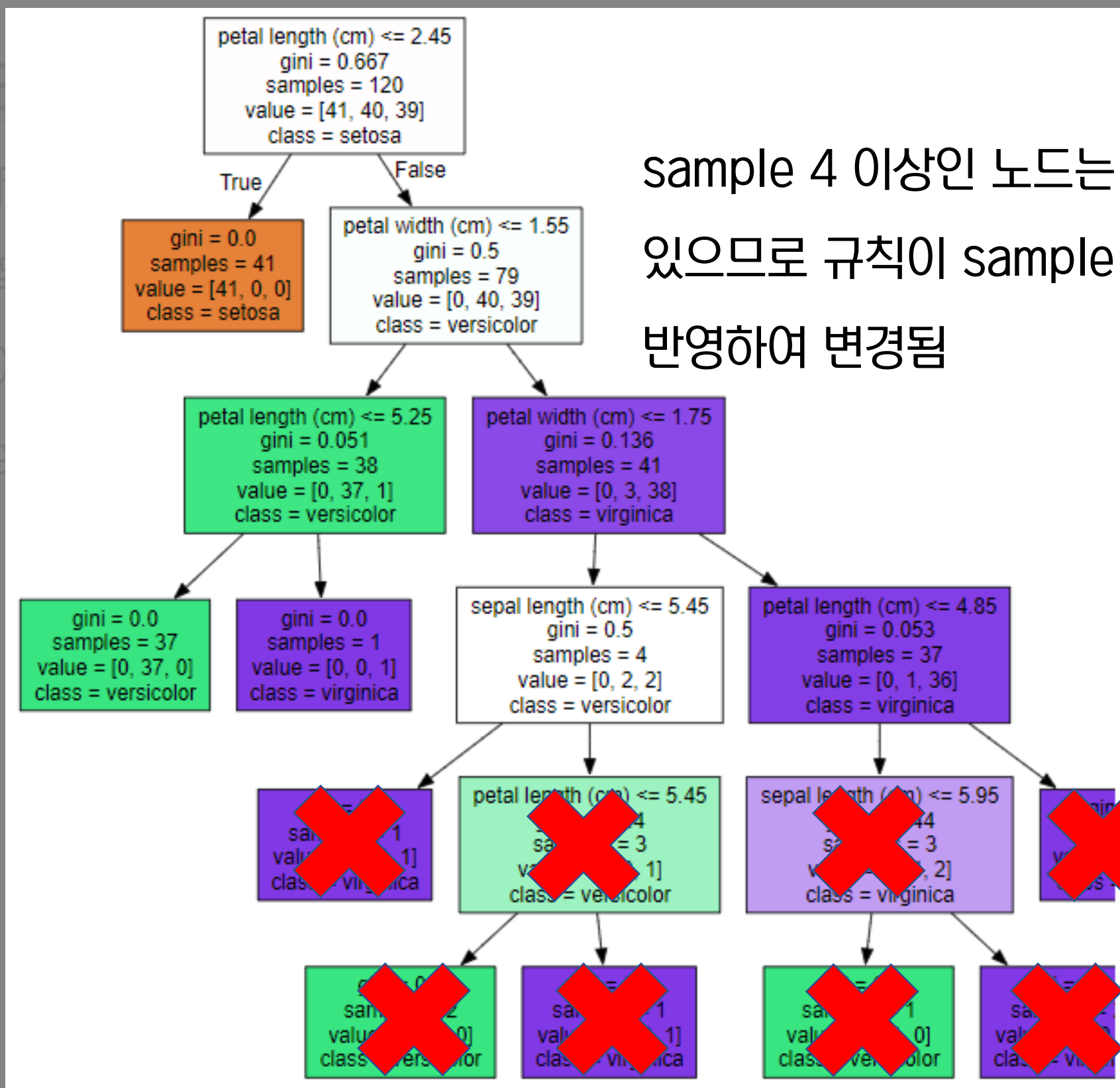
예제로 보는 결정 트리 규칙 구성

- `min_samples_leaf` (리프 노드가 될 수 있는 샘플 데이터 건수의 최소값)
 - 디폴트 : 1 -> 단독 클래스로만 되어 있거나 단 한 개의 데이터로 되어 있을 경우 리프 노드가 될 수 있음
 - 값을 키우면 더 이상 분할하지 않고 리프 노드가 될 수 있는 조건 완화
 - 예) `min_samples_leaf = 40`이면 샘플이 4 이하이면 리프 노드가 됨 -> 브랜치 노드 줄어들음, 결정 트리 간결하게 됨

#4.2 결정 트리

예제로 보는 결정 트리

- min_samples_leaf
- 디폴트 : 1 -> 단
- 값을 키우면 더
- 예) min_sample
- 간결하게 됨



sample 4 이상인 노드는 리프 클래스 노드가 될 수 있으므로 규칙이 sample 4인 노드를 만들 수 있는 상황을 반영하여 변경됨

#4.2 결정 트리

feature_importances_속성

- 결정 트리는 어떠한 속성을 규칙 조건으로 선택하느냐가 중요한 요건
- 중요한 몇 개의 피처가 명확한 규칙 트리를 만드는 데 크게 기여, 간결하고 이상치에 강한 모델 만들 수 있음
- 사이킷런은 피처의 중요한 역할 지표를 feature_importances_속성으로 제공
- 피처가 트리 분할 시 정보 이득이나 지니 계수를 얼마나 효율적으로 잘 개선시켰는지를 정규화된 값으로 표현한 것
- ndarray 형태로 값을 반환, 피처 순서대로 값이 할당됨
 - 예) feature_importances_가 [0.01667014, 0.02500521]라면 첫 번째 피처의 중요도가 0.01667014, 두 번째 피처의 중요도가 0.02500521
 - 일반적으로 값이 높을수록 해당 피처의 중요도가 높다는 의미

#4.2 결정 트리

feature_importances_속성

- 붓꽃 데이터 세트에서 피처별로 결정 트리 알고리즘에서 중요도 추출
- fit()으로 학습된 DecisionTreeClassifier 객체 변수인 dt_clf에서 feature_importances_속성을 가져와 피처별로 중요도 매핑, 막대그래프로 표현

```
In [6]: import seaborn as sns
import numpy as np
%matplotlib inline

# feature importance 추출
print("Feature importances:\n{0}".format(np.round(dt_clf.feature_importances_, 3)))

# feature별 importance 매핑
for name, value in zip(iris_data.feature_names , dt_clf.feature_importances_):
    print('{0} : {1:.3f}'.format(name, value))

# feature importance를 column 별로 시각화 하기
sns.barplot(x=dt_clf.feature_importances_ , y=iris_data.feature_names)
```

```
Feature importances:
[0.025 0.    0.555 0.42 ]
sepal length (cm) : 0.025
sepal width (cm)  : 0.000
petal length (cm) : 0.555
petal width (cm) : 0.420
```


#4.2 결정 트리

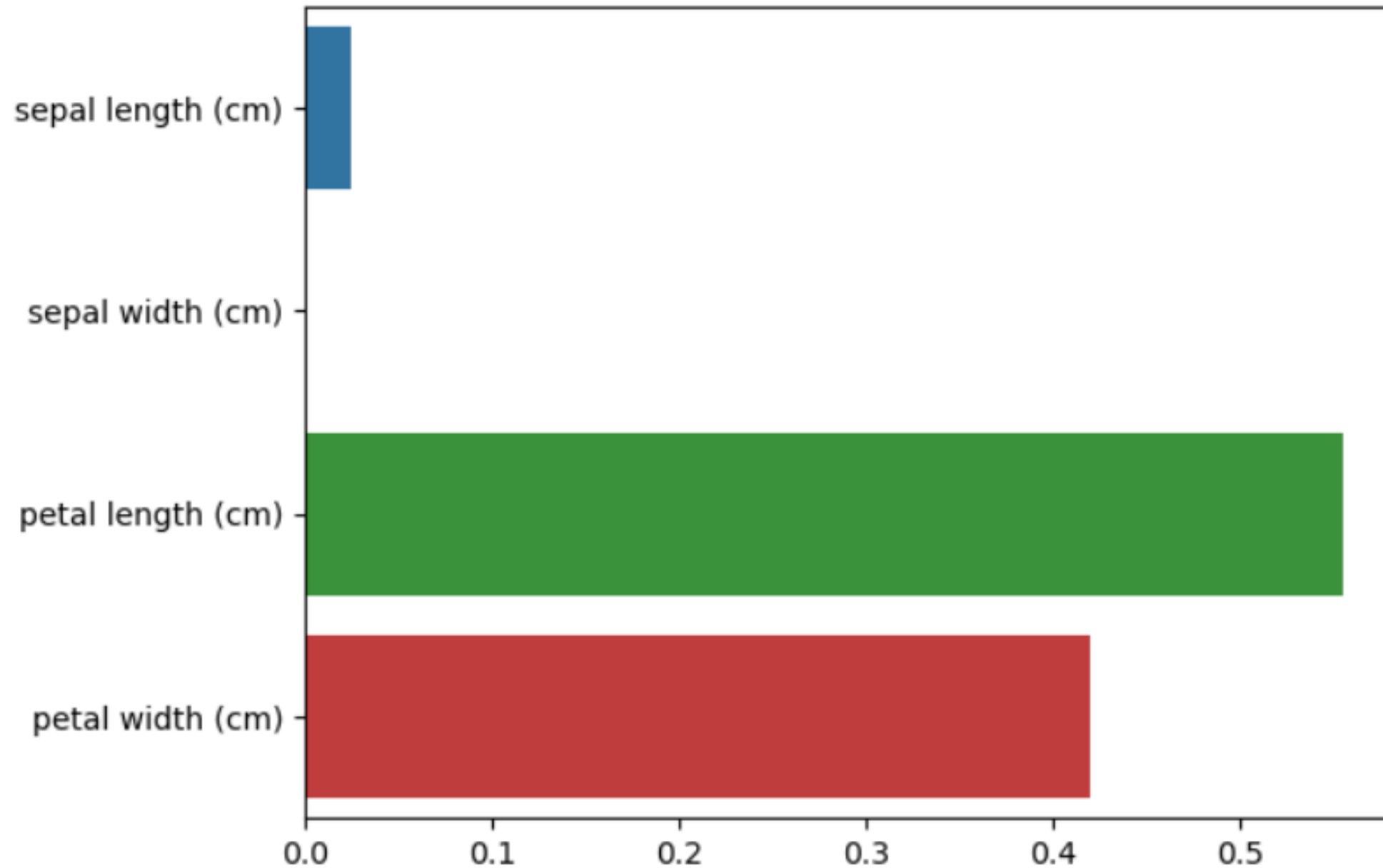
feature_importances_ 소스

- 붓꽃 데이터
- fit()으로
가져와 표

In [6]:

Out[6]: <Axes: >

여러 피쳐들 중 petal_length가 가장 피쳐 중요도가 높음



petal width (cm) = 0.420

#4.2 결정 트리

결정 트리 과적합(Overfitting)

- 사이킷런 make_classification() 함수를 이용해 2개의 피처가 3가지 유형의 클래스 값을 가지는 데이터 세트 생성, 그래프 형태로 시각화

```
from sklearn.datasets import make_classification
import matplotlib.pyplot as plt
%matplotlib inline

plt.title("3 Class values with 2 Features Sample data creation")

# 2차원 시각화를 위해서 feature는 2개, 결정값 클래스는 3가지 유형의 classification 샘플 데이터 생성.
X_features, y_labels = make_classification(n_features=2, n_redundant=0, n_informative=2,
                                          n_classes=3, n_clusters_per_class=1, random_state=0)

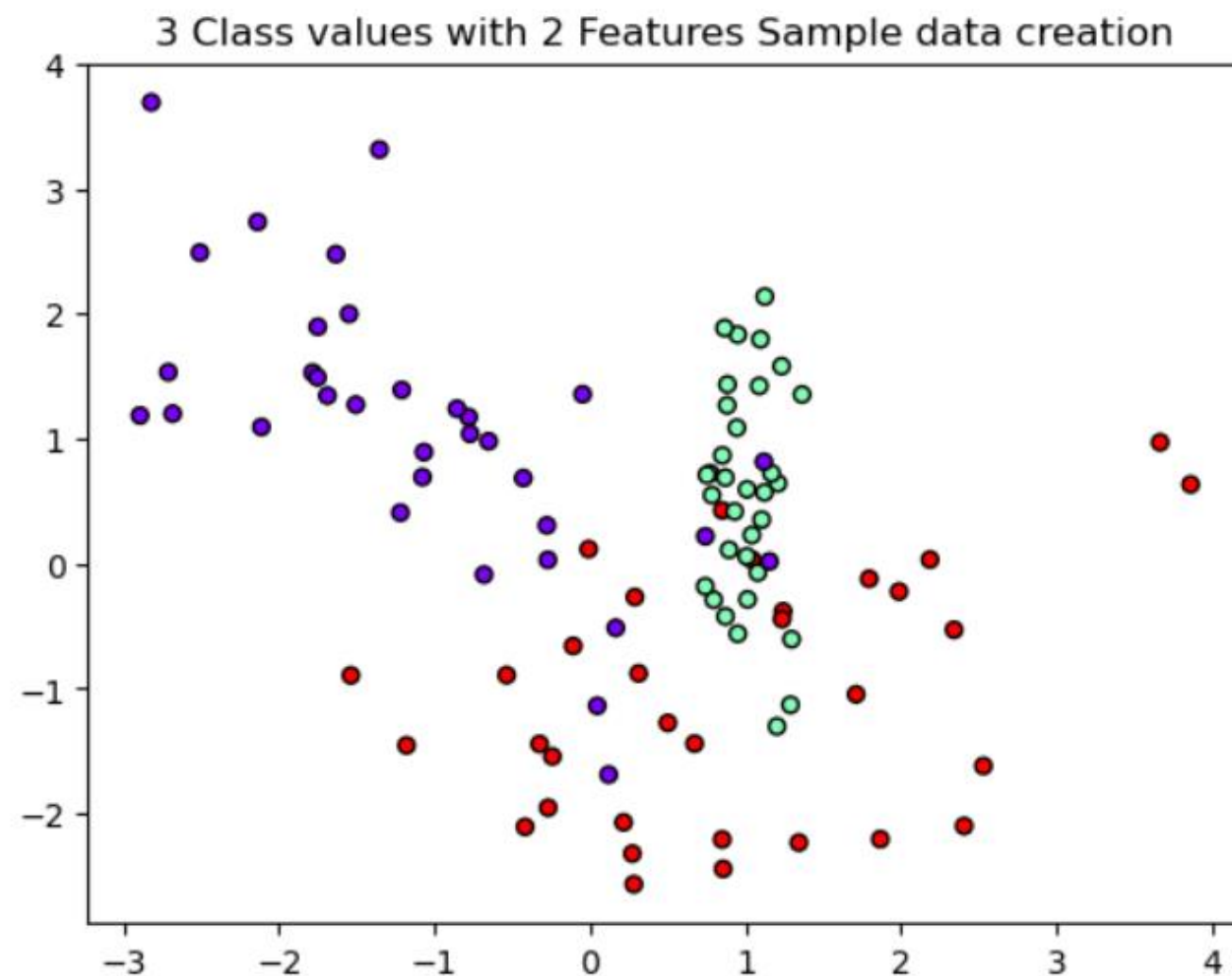
# plot 형태로 2개의 feature로 2차원 좌표 시각화, 각 클래스값은 다른 색깔로 표시됨.
plt.scatter(X_features[:, 0], X_features[:, 1], marker='o', c=y_labels, s=25, cmap='rainbow', edgecolor='k')
```

#4.2 결정 트리

결정 트리 과적합(Overfitting)

- 사이킷런 `make_classification()` 함수를 이용해 2개의 피처가 3가지 유형의 클래스 값을 가지는 데이터 세트 생성, 그래프 형태로 시각화

Out[7]: <matplotlib.collections.PathCollection at 0x2b441afc310>



각 피처가 X, Y축으로 나열된 2차원 그래프
3개의 클래스 값 구분은 색깔로 되어 있음

#4.2 결정 트리

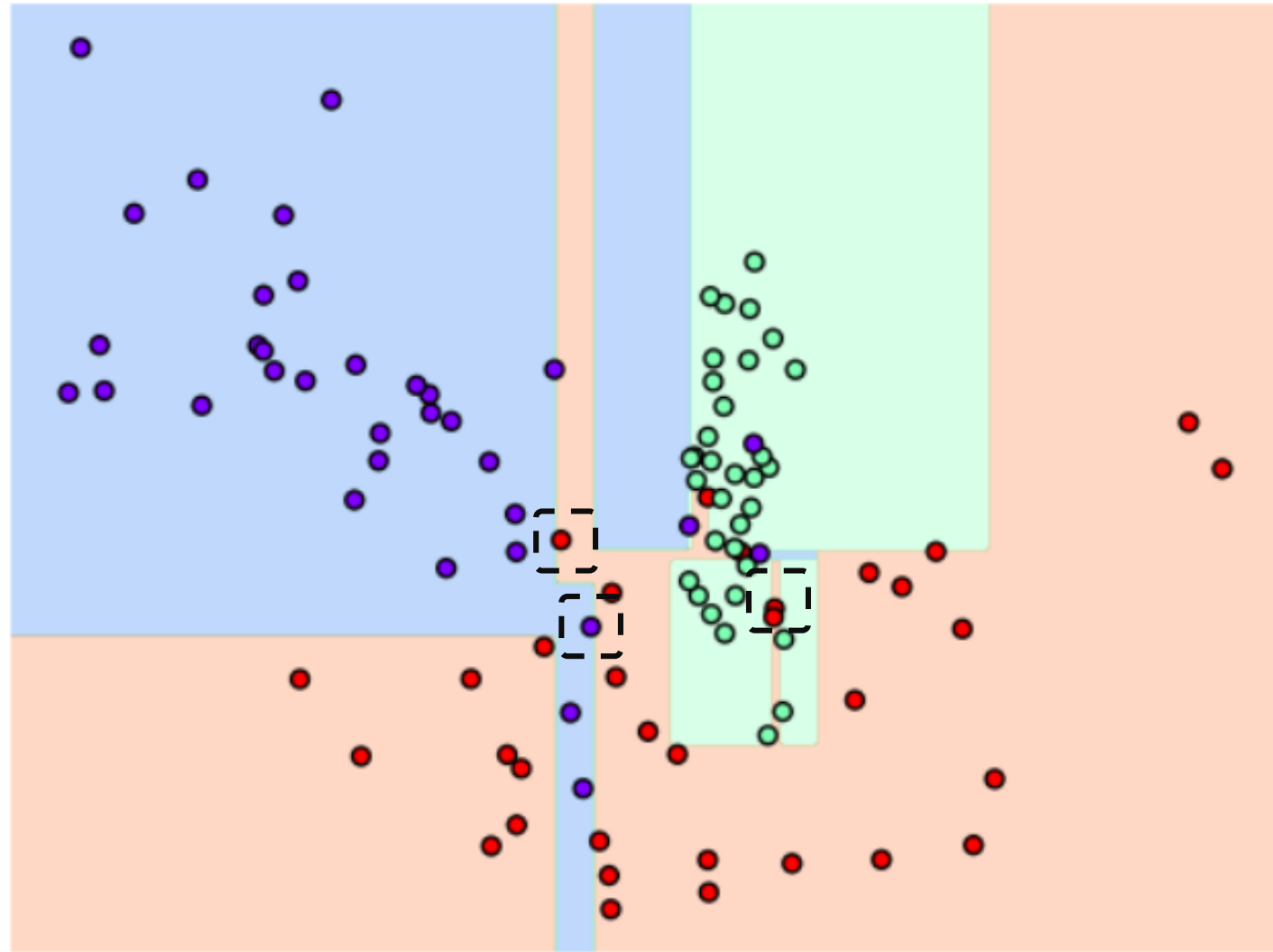
결정 트리 과적합(Overfitting)

- X_features와 y_labels 데이터 세트를 기반으로 결정 트리 학습
- 첫 번째 학습 : 결정 트리 하이퍼 파라미터를 디폴트로 설정, 결정 트리 모델 결정 기준 및 데이터 분류 확인
- visualize_boundary() : 머신러닝 모델이 클래스 값을 예측하는 결정 기준을 색상과 경계로 나타내
모델이 어떻게 데이터 세트를 예측 분류하는지 잘 이해할 수 있게 함

#4.2 결정 트리

```
In [9]: from sklearn.tree import DecisionTreeClassifier

# 특정한 트리 생성 제약없는 결정 트리의 Decision Boundary 시각화.
dt_clf = DecisionTreeClassifier(random_state=156).fit(X_features, y_labels)
visualize_boundary(dt_clf, X_features, y_labels)
```



학습

결정 트리 모델 결정 기준 및 데이터 분류 확인

측하는 결정 기준을 색상과 경계로 나타내

수 있게 함

일부 이상치 데이터까지 분류하기 위해 분할이 자주 일어남

-> 결정 기준 경계가 매우 많아지고 복잡함

-> 학습 데이터 세트 특성과 약간만 다른 형태의 데이터

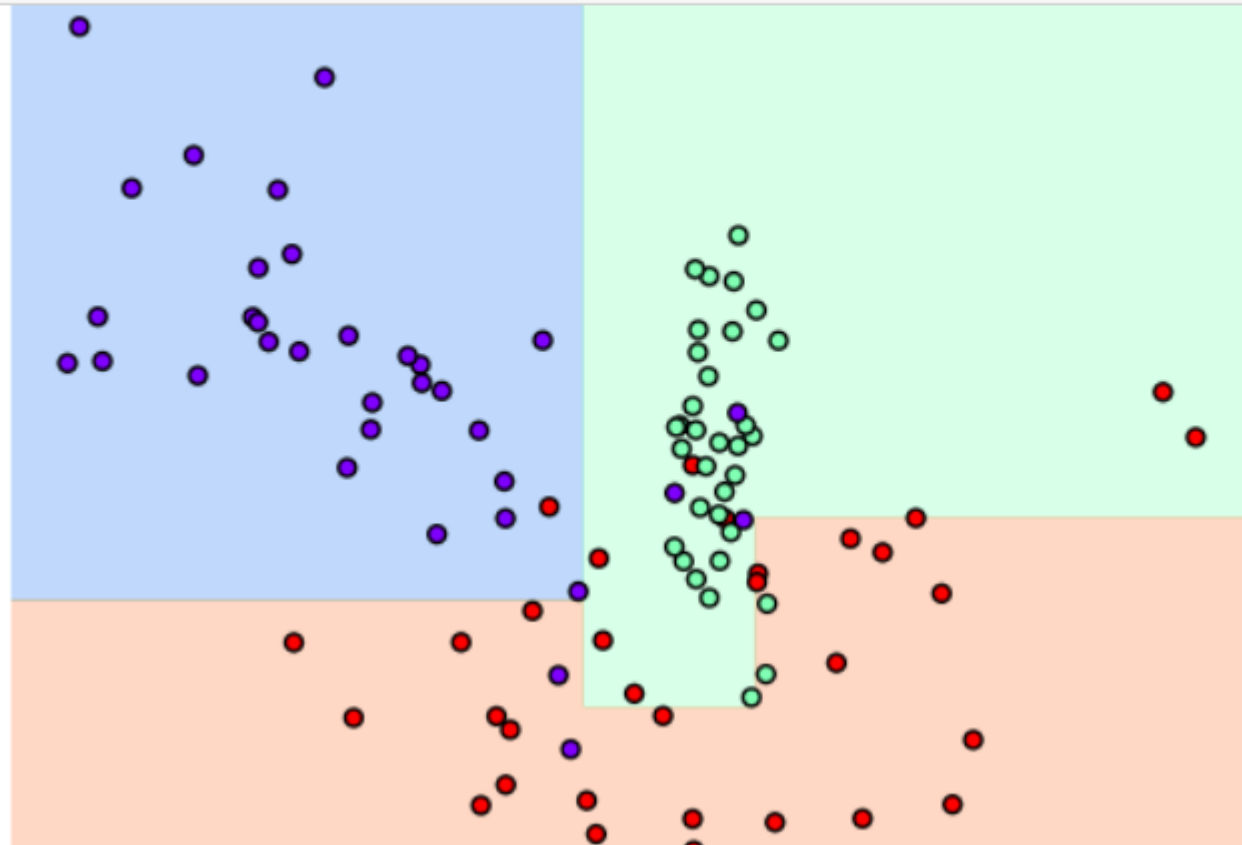
세트 예측 때 예측 정확도 떨어짐

#4.2 결정 트리

결정 트리 과적합(Overfitting)

- X_features와 y_labels 데이터 세트를 기반으로 결정 트리 학습
- 두 번째 학습 : 결정 트리 하이퍼 파라미터 중 min_samples_leaf = 6으로 변경, 결정 트리 모델 결정 기준 및 데이터 분류 확인

```
In [12]: # min_samples_leaf=6 으로 트리 생성 조건을 제약한 Decision Boundary 시각화  
dt_clf = DecisionTreeClassifier(min_samples_leaf=6, random_state=156).fit(X_features, y_labels)  
visualize_boundary(dt_clf, X_features, y_labels)
```



이상치에 크게 반응하지 않으며 조금 더 일반화된 분류
규칙에 따라 분류됨
-> 예측 성능은 첫 번째 모델보다 뛰어날 가능성 높음



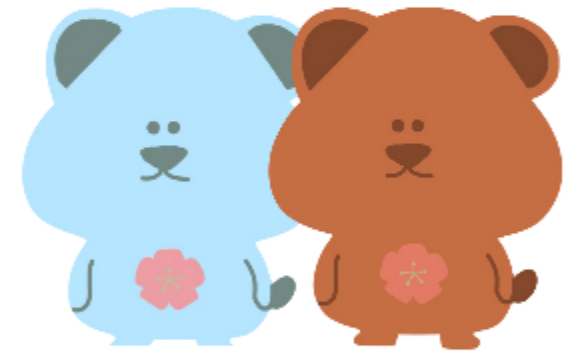
4장 02 결정 트리 실습 ~

03 앙상블 학습

강정인

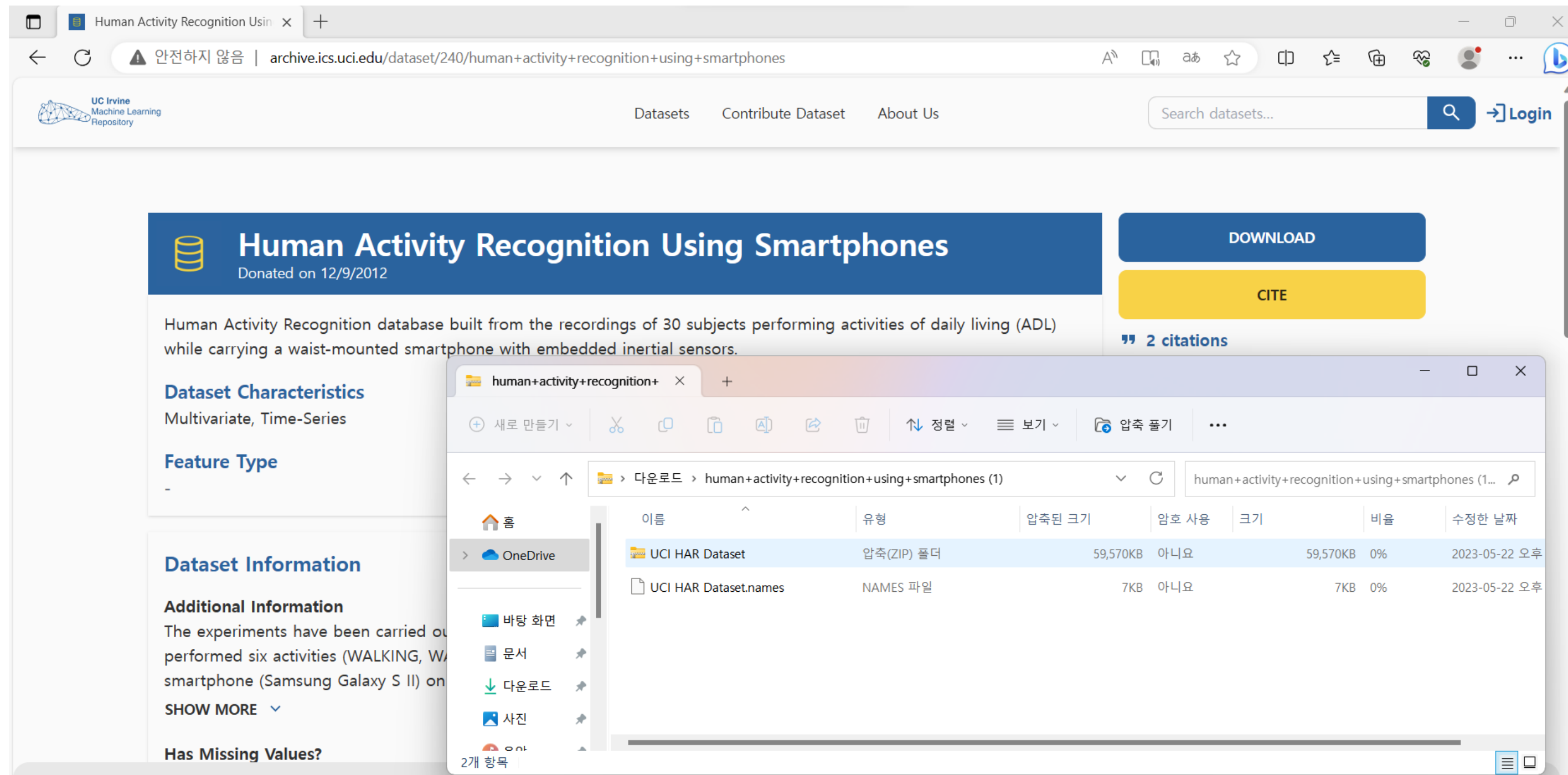
결정 트리 실습

-사용자 행동 인식 데이터 세트



#사용자 행동 인식 데이터 세트

- 스마트폰 센서를 장착한 뒤 사람의 동작과 관련된 여러 가지 피처를 수집한 데이터
- [Human Activity Recognition Using Smartphones - UCI Machine Learning Repository](https://archive.ics.uci.edu/dataset/240/human+activity+recognition+using+smartphones)에 접속 후 Data set zip Download



The screenshot displays the UCI Machine Learning Repository website for the 'Human Activity Recognition Using Smartphones' dataset. The page features a blue header with the dataset name and a 'Donated on 12/9/2012' note. Below the header, there is a description of the dataset: 'Human Activity Recognition database built from the recordings of 30 subjects performing activities of daily living (ADL) while carrying a waist-mounted smartphone with embedded inertial sensors.' The page also includes a 'Dataset Characteristics' section (Multivariate, Time-Series), a 'Feature Type' section, and a 'Dataset Information' section. A 'DOWNLOAD' button is prominently displayed in blue, and a 'CITE' button is in yellow. A '2 citations' badge is also visible. Overlaid on the bottom right is a file explorer window showing the downloaded files: 'UCI HAR Dataset' (59,570KB) and 'UCI HAR Dataset.names' (7KB).

이름	유형	압축된 크기	암호 사용	크기	비율	수정한 날짜
UCI HAR Dataset	압축(ZIP) 폴더	59,570KB	아니요	59,570KB	0%	2023-05-22 오후
UCI HAR Dataset.names	NAMES 파일	7KB	아니요	7KB	0%	2023-05-22 오후

#Data set 파일 구성

이름	수정한 날짜	유형
test	2023-09-17 오전 12:05	파일 폴더
train	2023-09-17 오전 12:05	파일 폴더
._DS_Store	2023-09-17 오전 12:05	DS_STORE 파일
._activity_labels	2023-09-17 오전 12:05	텍스트 문서
._features	2023-09-17 오전 12:05	텍스트 문서
._features_info	2023-09-17 오전 12:05	텍스트 문서
._README	2023-09-17 오전 12:05	텍스트 문서
._test	2023-09-17 오전 12:05	_TEST 파일
._train	2023-09-17 오전 12:05	_TRAIN 파일

Inertial Signals	2023-09-17 오전 12:05	파일 폴더	
._Inertial Signals	2023-09-17 오전 12:05	파일	1KB
._subject_test	2023-09-17 오전 12:05	텍스트 문서	1KB
._X_test	2023-09-17 오전 12:05	텍스트 문서	1KB
._y_test	2023-09-17 오전 12:05	텍스트 문서	1KB

Inertial Signals	2023-09-17 오전 12:05	파일 폴더	
._Inertial Signals	2023-09-17 오전 12:05	파일	1KB
._subject_train	2023-09-17 오전 12:05	텍스트 문서	1KB
._X_train	2023-09-17 오전 12:05	텍스트 문서	1KB
._y_train	2023-09-17 오전 12:05	텍스트 문서	1KB

1KB

1KB

1KB

#파일 DataFrame으로 로딩

- features.txt 파일을 DataFrame으로 로딩하기

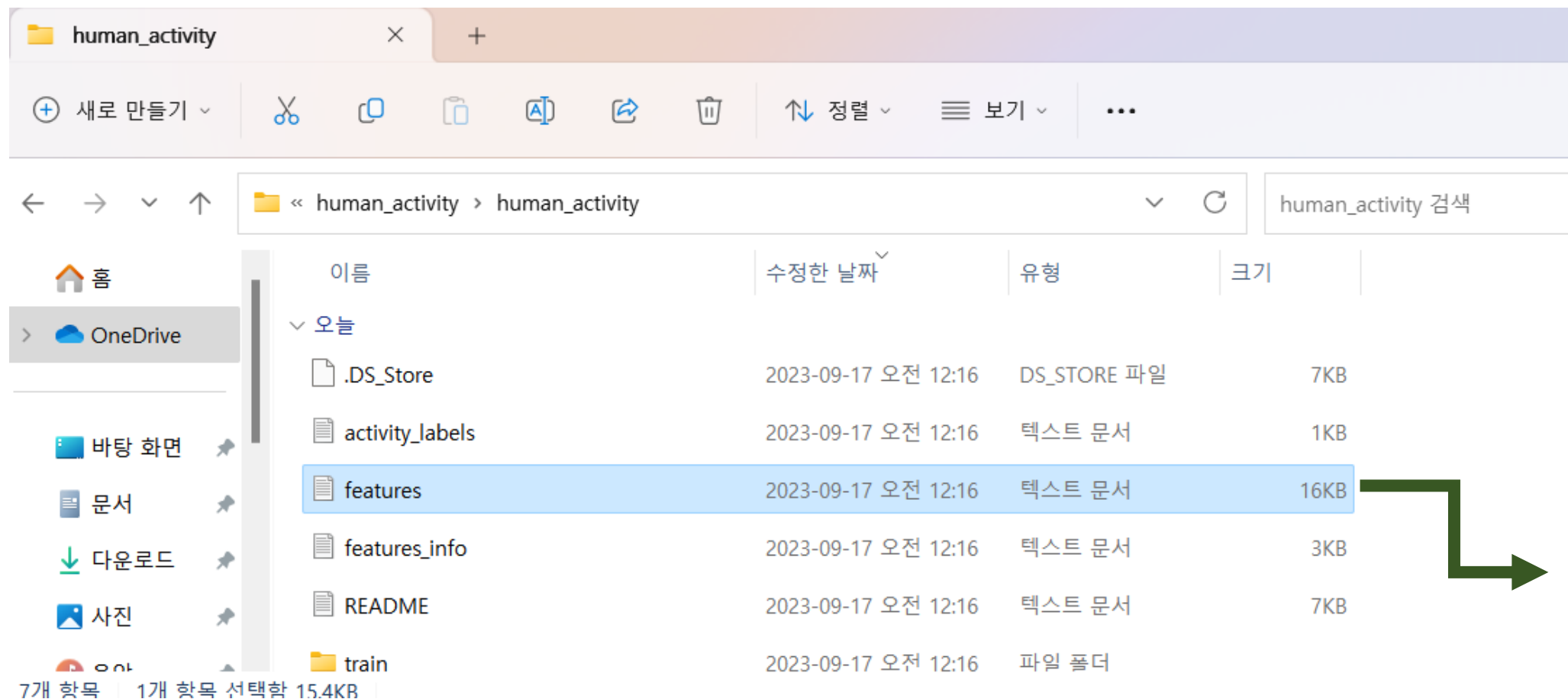
```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

# features.txt 파일에는 피쳐 이름 index와 피쳐명이 공백으로 분리되어 있음. 이를 DataFrame으로 로드.
feature_name_df = pd.read_csv('./human_activity/features.txt', sep='\\s+',
                              header=None, names=['column_index', 'column_name'])

# 피쳐명 index를 제거하고, 피쳐명만 리스트 객체로 생성한 뒤 샘플로 10개만 추출
feature_name = feature_name_df.iloc[:, 1].values.tolist()
print('전체 피쳐명에서 10개만 추출:', feature_name[:10])
```

전체 피쳐명에서 10개만 추출: ['tBodyAcc-mean()-X', 'tBodyAcc-mean()-Y', 'tBodyAcc-mean()-Z', 'tBodyAcc-std()-X', 'tBodyAcc-std()-Y', 'tBodyAcc-std()-Z', 'tBodyAcc-mad()-X', 'tBodyAcc-mad()-Y',

#중복된 피쳐명



이름	수정한 날짜	유형	크기
.DS_Store	2023-09-17 오전 12:16	DS_STORE 파일	7KB
activity_labels	2023-09-17 오전 12:16	텍스트 문서	1KB
features	2023-09-17 오전 12:16	텍스트 문서	16KB
features_info	2023-09-17 오전 12:16	텍스트 문서	3KB
README	2023-09-17 오전 12:16	텍스트 문서	7KB
train	2023-09-17 오전 12:16	파일 폴더	

```
1 tBodyAcc-mean()-X
2 tBodyAcc-mean()-Y
3 tBodyAcc-mean()-Z
4 tBodyAcc-std()-X
5 tBodyAcc-std()-Y
6 tBodyAcc-std()-Z
7 tBodyAcc-mad()-X
8 tBodyAcc-mad()-Y
9 tBodyAcc-mad()-Z
10 tBodyAcc-max()-X
11 tBodyAcc-max()-Y
12 tBodyAcc-max()-Z
13 tBodyAcc-min()-X
14 tBodyAcc-min()-Y
15 tBodyAcc-min()-Z
16 tBodyAcc-sma()
17 tBodyAcc-energy()-X
18 tBodyAcc-energy()-Y
19 tBodyAcc-energy()-Z
20 tBodyAcc-iqr()-X
21 tBodyAcc-iqr()-Y
22 tBodyAcc-iqr()-Z
23 tBodyAcc-entropy()-X
24 tBodyAcc-entropy()-Y
25 tBodyAcc-entropy()-Z
26 tBodyAcc-arCoeff()-X,1
27 tBodyAcc-arCoeff()-X,2
28 tBodyAcc-arCoeff()-X,3
29 tBodyAcc-arCoeff()-X,4
30 tBodyAcc-arCoeff()-Y,1
31 tBodyAcc-arCoeff()-Y,2
32 tBodyAcc-arCoeff()-Y,3
33 tBodyAcc-arCoeff()-Y,4
34 tBodyAcc-arCoeff()-Z,1
35 tBodyAcc-arCoeff()-Z,2
36 tBodyAcc-arCoeff()-Z,3
37 tBodyAcc-arCoeff()-Z,4
38 tBodyAcc-correlation()-X,Y
39 tBodyAcc-correlation()-X,Z
40 tBodyAcc-correlation()-Y,Z
```

#중복된 피처명

```
feature_dup_df = feature_name_df.groupby('column_name').count()
print(feature_dup_df[feature_dup_df['column_index'] > 1].count())
feature_dup_df[feature_dup_df['column_index'] > 1].head()
```

column_index 42
dtype: int64

중복된 피처명 42개

column_index

column_name

fBodyAcc-bandsEnergy0-1,16	3
fBodyAcc-bandsEnergy0-1,24	3
fBodyAcc-bandsEnergy0-1,8	3
fBodyAcc-bandsEnergy0-17,24	3
fBodyAcc-bandsEnergy0-17,32	3



#중복된 피처명 처리

- 중복된 피처명에 대해서 원본 피처명에 _1 또는 _2를 추가
- 새로운 피처명을 가지는 DataFrame을 반환하는 함수인 get_new_feature_name_df()를 생성

```
def get_new_feature_name_df(old_feature_name_df):  
    feature_dup_df = pd.DataFrame(data=old_feature_name_df.groupby('column_name').cumcount(),  
                                  columns=['dup_cnt'])  
    feature_dup_df = feature_dup_df.reset_index()  
    new_feature_name_df = pd.merge(old_feature_name_df.reset_index(), feature_dup_df, how='outer')  
    new_feature_name_df['column_name'] = new_feature_name_df[['column_name', 'dup_cnt']].apply(lambda x : x[0]+'_'+str(x[1])  
                                                    if x[1] >0 else x[0] , axis=1)  
    new_feature_name_df = new_feature_name_df.drop(['index'], axis=1)  
    return new_feature_name_df
```

#학습,테스트용 피쳐 파일 로딩

```
import pandas as pd

def get_human_dataset():

    # 각 데이터 파일들은 공백으로 분리되어 있으므로 read_csv에서 공백 문자를 sep으로 할당.
    feature_name_df = pd.read_csv('./human_activity/features.txt', sep='\s+',
                                   header=None, names=['column_index', 'column_name'])

    # 중복된 피쳐명을 수정하는 get_new_feature_name_df()를 이용, 신규 피쳐명 DataFrame 생성.
    new_feature_name_df = get_new_feature_name_df(feature_name_df)

    # DataFrame에 피쳐명을 컬럼으로 부여하기 위해 리스트 객체로 다시 변환
    feature_name = new_feature_name_df.iloc[:, 1].values.tolist()

    # 학습 피쳐 데이터 셋과 테스트 피쳐 데이터를 DataFrame으로 로딩. 컬럼명은 feature_name 적용
    X_train = pd.read_csv('./human_activity/train/X_train.txt', sep='\s+', names=feature_name)
    X_test = pd.read_csv('./human_activity/test/X_test.txt', sep='\s+', names=feature_name)

    # 학습 레이블과 테스트 레이블 데이터를 DataFrame으로 로딩하고 컬럼명은 action으로 부여
    y_train = pd.read_csv('./human_activity/train/y_train.txt', sep='\s+', header=None, names=['action'])
    y_test = pd.read_csv('./human_activity/test/y_test.txt', sep='\s+', header=None, names=['action'])

    # 로드된 학습/테스트용 DataFrame을 모두 반환
    return X_train, X_test, y_train, y_test
```

```
X_train, X_test, y_train, y_test = get_human_dataset()
```

#학습용 피쳐 데이터 세트 분석

```
▶ print('## 학습 피쳐 데이터셋 info()')  
print(X_train.info())
```

**7352개의 레코드
561개의 피쳐**

```
↳ ## 학습 피쳐 데이터셋 info()  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 7352 entries, 0 to 7351  
Columns: 561 entries, tBodyAcc-mean()-X to angle(Z,gravityMean)  
dtypes: float64(561)  
memory usage: 31.5 MB  
None
```

```
[20] print(y_train['action'].value_counts())
```

```
6      1407  
5      1374  
4      1286  
1      1226  
2      1073  
3       986  
Name: action, dtype: int64
```

**레이블값 1,2,3,4,5,6 개
분포도는 고르게 분포**

#동작 예측 분류 수행

- 사이킷런의 `DecisionTreeClassifier`를 이용해 동작 예측 분류 수행
- 하이퍼 파라미터의 값 추출

```
[▶] # @title 기본 제목 텍스트
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# 예제 반복 시마다 동일한 예측 결과 도출을 위해 random_state 설정
dt_clf = DecisionTreeClassifier(random_state=156)
dt_clf.fit(X_train , y_train)
pred = dt_clf.predict(X_test)
accuracy = accuracy_score(y_test , pred)
print('결정 트리 예측 정확도: {0:.4f}'.format(accuracy))

# DecisionTreeClassifier의 하이퍼 파라미터 추출
print('DecisionTreeClassifier 기본 하이퍼 파라미터:\n', dt_clf.get_params())
```

```
☞ 결정 트리 예측 정확도: 0.8548
DecisionTreeClassifier 기본 하이퍼 파라미터:
{'ccp_alpha': 0.0, 'class_weight': None, 'criterion': 'gini', 'max_depth': None, 'max_features': None,
```

#예측 정확도

- 결정 트리의 깊이(Tree Depth)가 예측 정확도에 주는 영향
- 결정 트리의 경우 리프 노드가 될 수 있는 적합한 수준이 될 때까지 트리의 분할 수행
- GridSearchCV를 이용해 하이퍼 파라미터 max_depth의 값을 변화시킴
- Min_samples_split는 16으로 고정 후 max_depth를 6,8,10,12,16,20,24로 계속 늘리면서 예측 성능 측정

```
▶ from sklearn.model_selection import GridSearchCV

params = {
    'max_depth' : [ 6, 8 ,10, 12, 16 ,20, 24]
}

grid_cv = GridSearchCV(dt_clf, param_grid=params, scoring='accuracy', cv=5, verbose=1 )
grid_cv.fit(X_train , y_train)
print('GridSearchCV 최고 평균 정확도 수치:{0:.4f}'.format(grid_cv.best_score_))
print('GridSearchCV 최적 하이퍼 파라미터:', grid_cv.best_params_)
```

Fitting 5 folds for each of 7 candidates, totalling 35 fits
GridSearchCV 최고 평균 정확도 수치:0.8549
GridSearchCV 최적 하이퍼 파라미터: {'max_depth': 8}

#예측 성능 변화 관찰



```
# GridSearchCV객체의 cv_results_ 속성을 DataFrame으로 생성.
```

```
cv_results_df = pd.DataFrame(grid_cv.cv_results_)
```

```
# max_depth 파라미터 값과 그때의 테스트(Evaluation)셋, 학습 데이터 셋의 정확도 수치 추출
```

```
cv_results_df[['param_max_depth', 'mean_test_score']]
```

	param_max_depth	mean_test_score
0	6	0.847662
1	8	0.854879
2	10	0.852705
3	12	0.845768
4	16	0.847127
5	20	0.848624
6	24	0.848624



#결정 트리의 정확도 측정



```
max_depths = [ 6, 8 ,10, 12, 16 ,20, 24]
# max_depth 값을 변화 시키면서 그때마다 학습과 테스트 셋에서의 예측 성능 측정
for depth in max_depths:
    dt_clf = DecisionTreeClassifier(max_depth=depth, min_samples_split=16, random_state=156)
    dt_clf.fit(X_train , y_train)
    pred = dt_clf.predict(X_test)
    accuracy = accuracy_score(y_test , pred)
    print('max_depth = {0} 정확도: {1:.4f}'.format(depth , accuracy))
```

max_depth = 6 정확도: 0.8551

max_depth = 8 정확도: 0.8717

max_depth = 10 정확도: 0.8599

max_depth = 12 정확도: 0.8571

max_depth = 16 정확도: 0.8599

max_depth = 20 정확도: 0.8565

max_depth = 24 정확도: 0.8565

#정확도 성능 튜닝

- max_depth와 min_samples_split 둘 다 변경

```
▶ params = {  
    'max_depth' : [ 8 , 12, 16 ,20],  
    'min_samples_split' : [16, 24],  
}  
  
grid_cv = GridSearchCV(dt_clf, param_grid=params, scoring='accuracy', cv=5, verbose=1 )  
grid_cv.fit(X_train , y_train)  
print('GridSearchCV 최고 평균 정확도 수치: {0:.4f}'.format(grid_cv.best_score_))  
print('GridSearchCV 최적 하이퍼 파라미터:', grid_cv.best_params_)
```

Fitting 5 folds for each of 8 candidates, totalling 40 fits

GridSearchCV 최고 평균 정확도 수치: 0.8549

GridSearchCV 최적 하이퍼 파라미터: {'max_depth': 8, 'min_samples_split': 16}

max_depth가 8, min_samples_split이 16일 때 정확도 85.49%

#결정 트리의 예측 정확도

- `best_estimator_` :
최적 하이퍼 파라미터인 `max_depth8`, `min_samples_split 16`으로 학습이 완료된 Estimator 객체



```
best_df_clf = grid_cv.best_estimator_  
pred1 = best_df_clf.predict(X_test)  
accuracy = accuracy_score(y_test , pred1)  
print('결정 트리 예측 정확도:{0:.4f}'.format(accuracy))
```

결정 트리 예측 정확도:0.8717

`max_depth 8`, `min_samples_split 16`일 때 테스트 데이터 세트의 예측 정확도는 약 87.17%

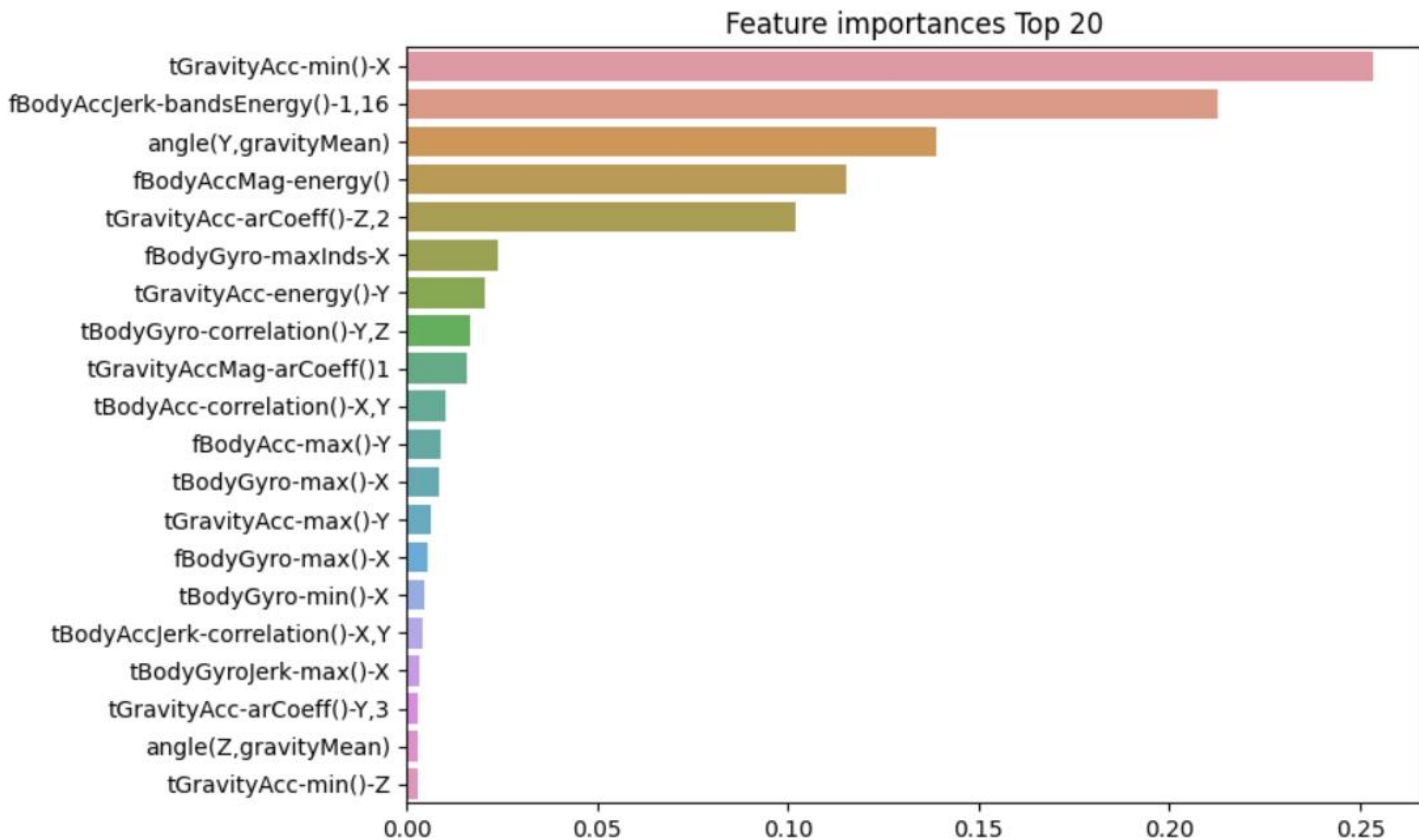
#피처의 중요도

- Feature의 중요도를 feature_importances_ 속성을 이용해 알아보기
- 중요도 높은 순으로 Top 20 피처를 막대그래프로 표현

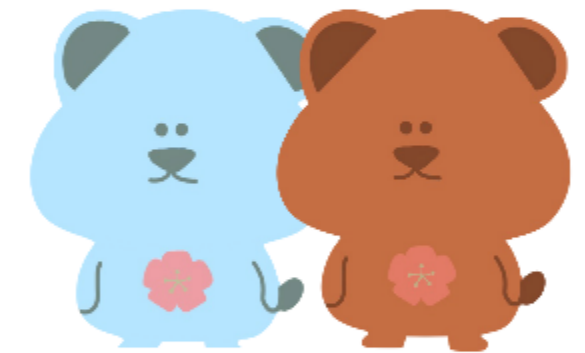
```
import seaborn as sns

ftr_importances_values = best_df_clf.feature_importances_
# Top 중요도로 정렬을 쉽게 하고, 시본(Seaborn)의 막대그래프로 쉽게 표현하기 위해 Series변환
ftr_importances = pd.Series(ftr_importances_values, index=X_train.columns )
# 중요도값 순으로 Series를 정렬
ftr_top20 = ftr_importances.sort_values(ascending=False)[:20]
plt.figure(figsize=(8,6))
plt.title('Feature importances Top 20')
sns.barplot(x=ftr_top20 , y = ftr_top20.index)
plt.show()
```

#중요도 높은 순으로 막대 그래프로 표현



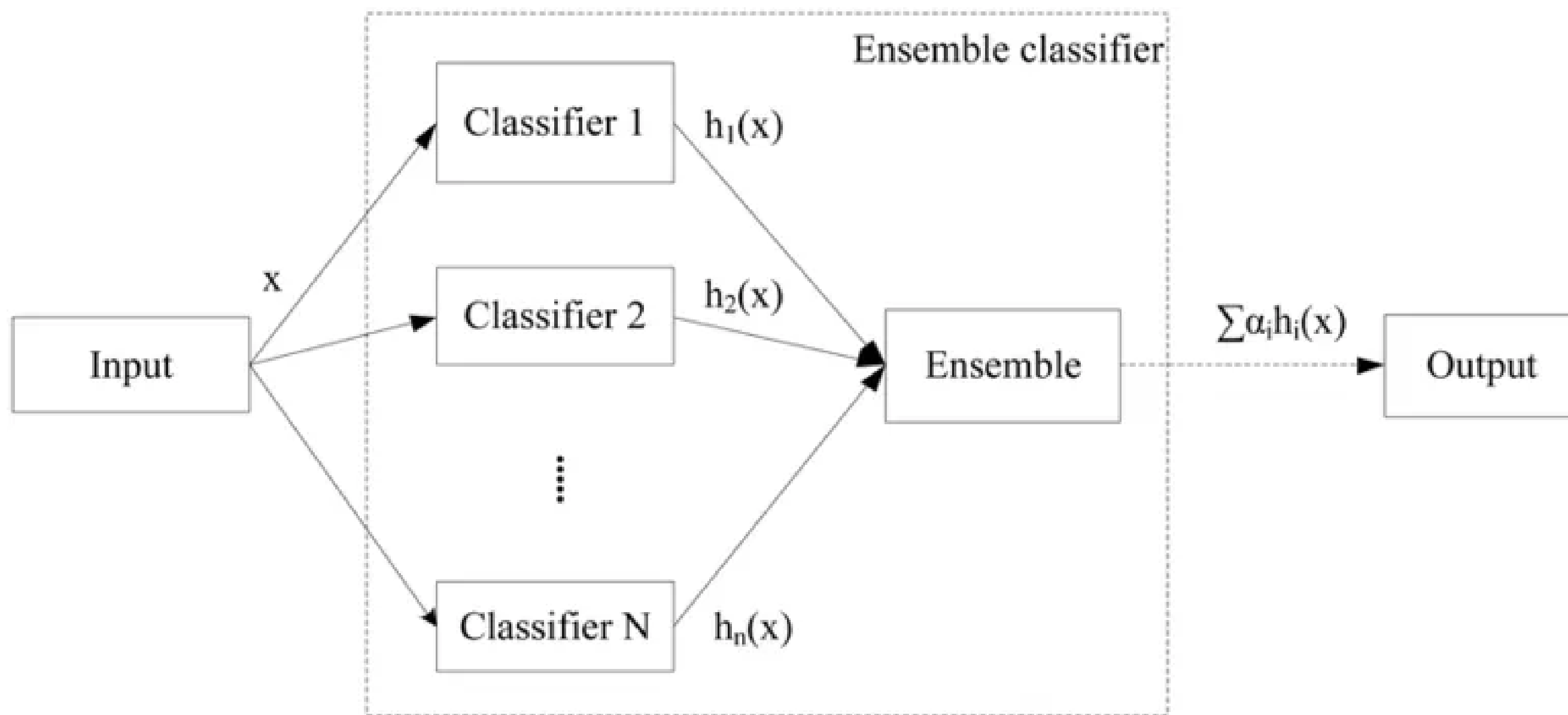
03 앙상블 학습



#앙상블 학습의 개요

여러 개의 분류기를 생성하고 그 예측을 결합함으로써 보다 정확한 최종 예측 도출

-단일 분류기보다 신뢰성의 높은 예측값을 얻음



#앙상블 학습의 유형

- 일반적으로 보팅(Voting), 배깅(Bagging), 부스팅(Boosting)으로 구분하며 이외에 스택킹 기법 등이 있음
- 배깅은 랜덤 포레스트(Random Forest) 알고리즘이 대표적
- 부스팅은 에이다 부스팅, 그래디언트 부스팅, XGBoost, LightGBM 등이 있음
- 정형 데이터의 분류나 회귀에서는 GBM 부스팅 계열의 앙상블이 높은 예측 성능을 나타냄

#앙상블 학습의 유형_보팅과 배깅

- 보팅과 배깅

- 여러 개의 분류기가 투표를 통해 최종 예측 결과를 결정하는 방식

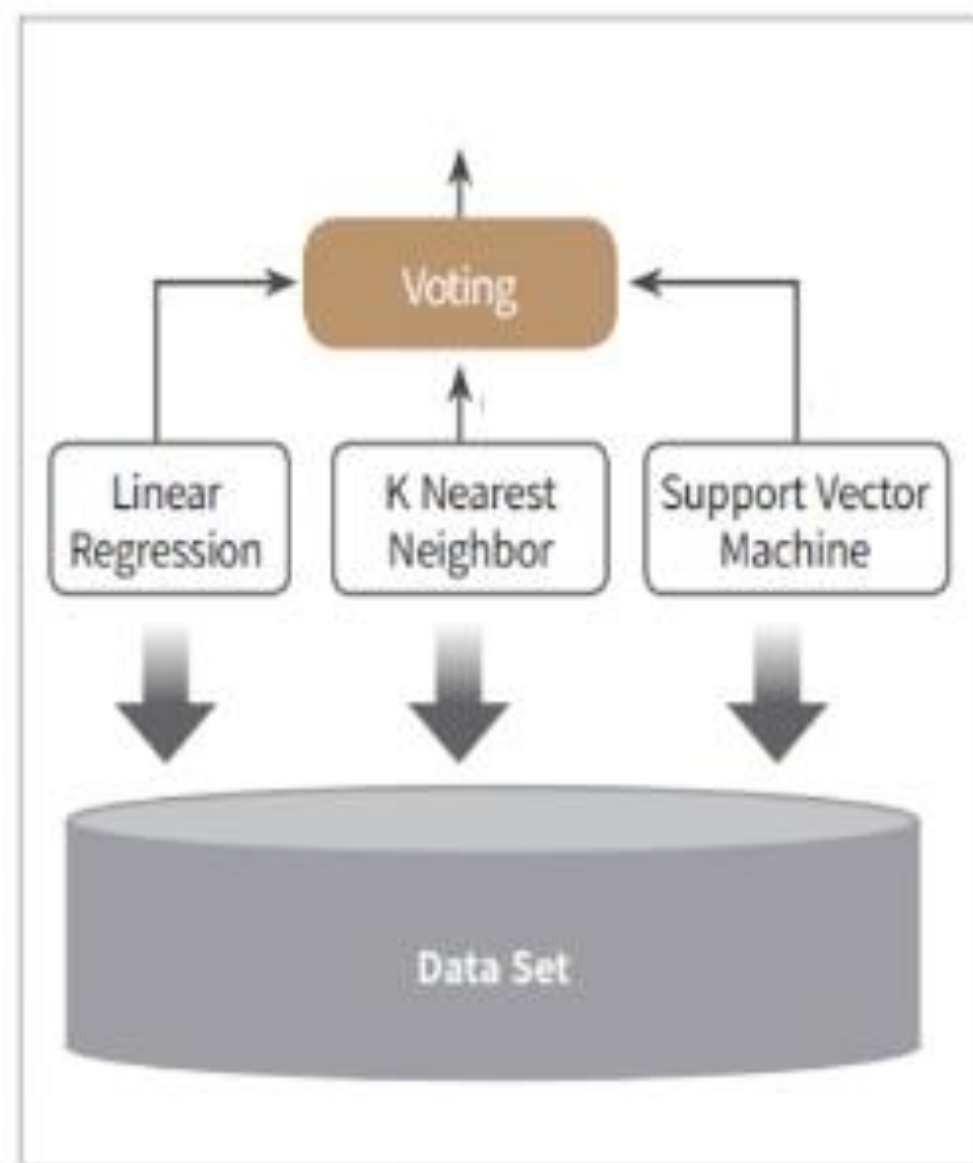
- 보팅

- 서로 다른 알고리즘을 가진 분류기 결합

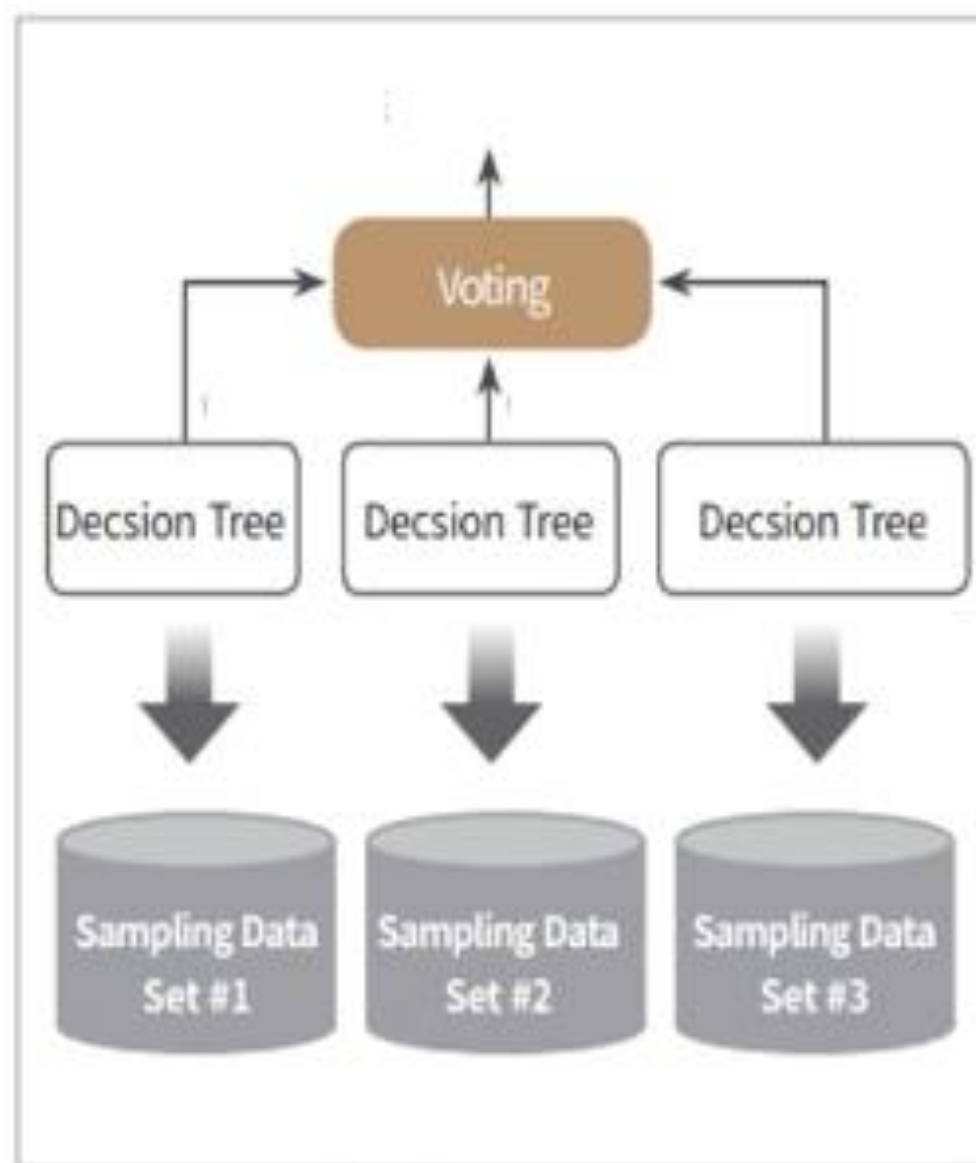
- 배깅

- 각각의 분류기가 모두 같은 유형의 알고리즘 기반
- 데이터 샘플링을 서로 다르게 가져가면서 학습 수행
- 랜덤 포레스트 알고리즘

#분류기 도식화



Voting



Bagging

보팅 분류기

‘선형 회귀, K 최근접 이웃, 서포트 벡터 머신’이라는 3개의 ML알고리즘이 같은 데이터 세트에 대해 학습하고 예측한 결과를 보팅을 통해 최종 예측 결과 산정

배깅 분류기

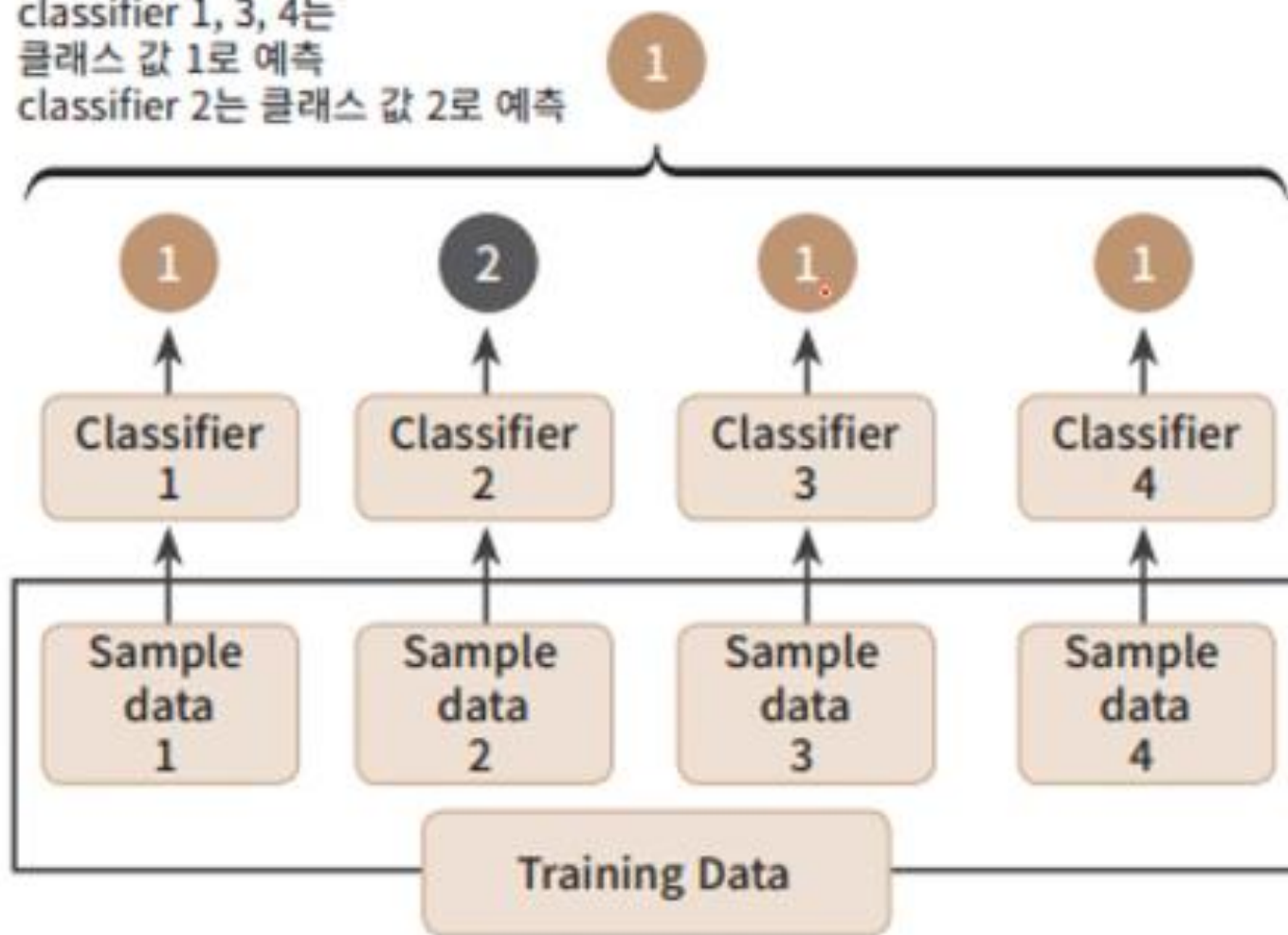
개별 분류기가 부트 스트래핑 방식으로 샘플링된 데이터 세트에 대해 학습을 통해 개별적인 예측을 수행한 결과를 보팅을 통해 최종 예측 결과 선정

#보팅 유형

- 하드 보팅 - 다수결의 원칙
- 소프트 보팅 - 확률을 평균하여 결정

Hard Voting은 다수의 classifier 간 다수결로 최종 class 결정

클래스 값 1로 예측
classifier 1, 3, 4는
클래스 값 1로 예측
classifier 2는 클래스 값 2로 예측

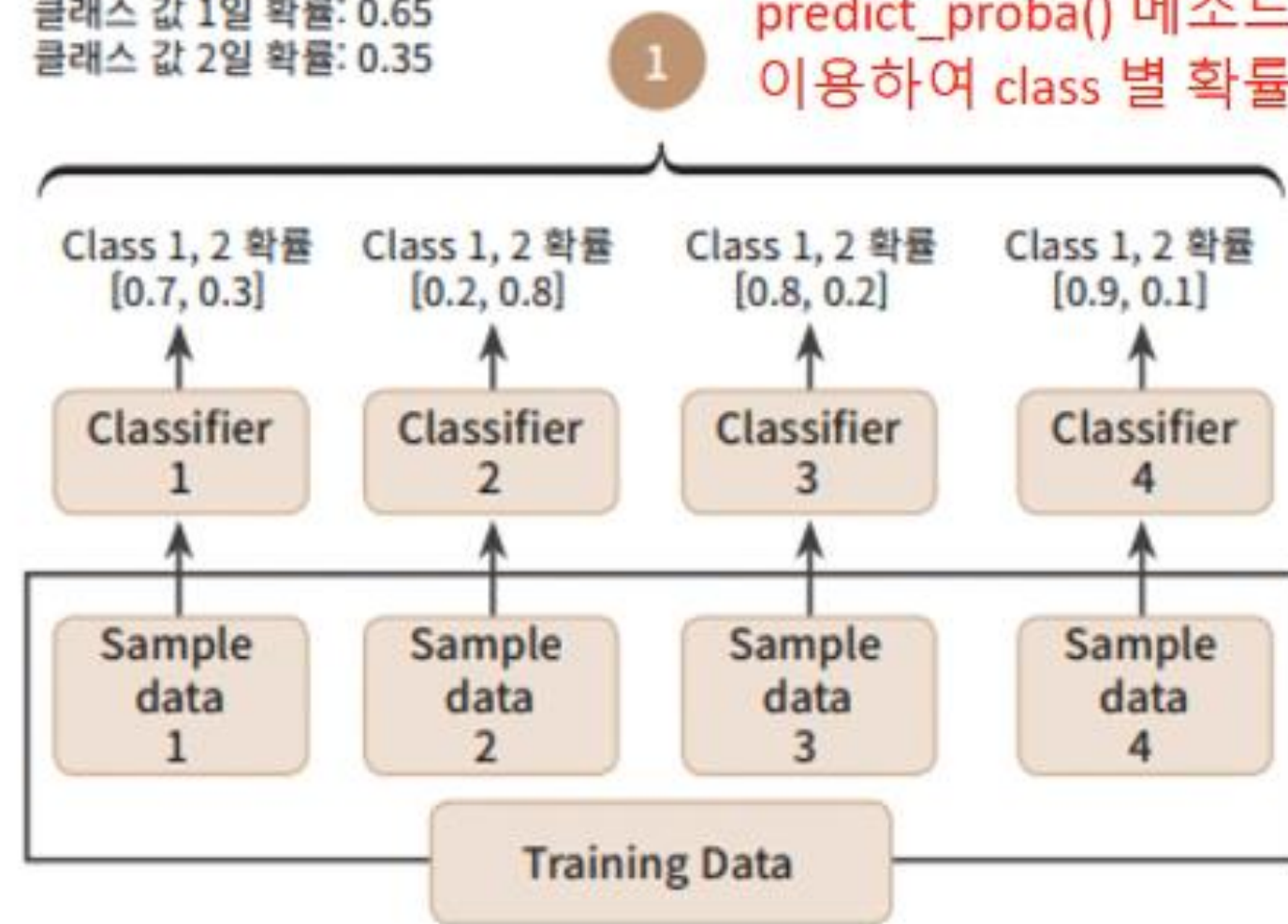


<하드 보팅>

Soft Voting은 다수의 classifier 들의 class 확률을 평균하여 결정

클래스 값 1로 예측
클래스 값 1일 확률: 0.65
클래스 값 2일 확률: 0.35

**predict_proba() 메소드를
이용하여 class 별 확률 결정**



<소프트 보팅>

#보팅 분류기

사이킷런은 보팅 방식의 앙상블을 구현한 VotingClassifier 클래스 제공

위스콘신 유방암 데이터 세트 예측 분석

- 유방암의 악성종양, 양성종양 여부를 결정하는 이진 분류 데이터 세트
- 종양의 크기, 모양 등의 형태와 관련한 피쳐
- 로지스틱 회귀와 KNN 기반으로 보팅 분류기 생성

#위스콘신 데이터 세트 예측 분석

```
import pandas as pd

from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

cancer = load_breast_cancer()

data_df = pd.DataFrame(cancer.data, columns=cancer.feature_names)
data_df.head(3)
```

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	mean fractal dimension	...	worst radius	worst texture	worst perimeter
0	17.99	10.38	122.8	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	0.07871	...	25.38	17.33	184.6
1	20.57	17.77	132.9	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	0.05667	...	24.99	23.41	158.8
2	19.69	21.25	130.0	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	0.05999	...	23.57	25.53	152.5

3 rows x 30 columns

#소프트 방식의 보팅 분류기 생성

- *VotingClassifier* 클래스는 주요 생성 인자로 *estimators*와 *voting* 값 입력
- *Estimators*는 리스트 값으로 보팅에 사용될 여러 개의 *Classifier* 객체들을 튜플 형식으로 입력

```
[4] # 개별 모델은 로지스틱 회귀와 KNN 임.  
lr_clf = LogisticRegression(solver='liblinear')  
knn_clf = KNeighborsClassifier(n_neighbors=8)  
  
# 개별 모델을 소프트 보팅 기반의 앙상블 모델로 구현한 분류기  
vo_clf = VotingClassifier( estimators=[('LR',lr_clf),('KNN',knn_clf)] , voting='soft' )  
  
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,  
                                                    test_size=0.2 , random_state= 156)  
  
# VotingClassifier 학습/예측/평가.  
vo_clf.fit(X_train , y_train)  
pred = vo_clf.predict(X_test)  
print('Voting 분류기 정확도: {0:.4f}'.format(accuracy_score(y_test , pred)))  
  
# 개별 모델의 학습/예측/평가.  
classifiers = [lr_clf, knn_clf]  
for classifier in classifiers:  
    classifier.fit(X_train , y_train)  
    pred = classifier.predict(X_test)  
    class_name= classifier.__class__.__name__  
    print('{0} 정확도: {1:.4f}'.format(class_name, accuracy_score(y_test , pred)))
```

```
Voting 분류기 정확도: 0.9561  
LogisticRegression 정확도: 0.9474  
KNeighborsClassifier 정확도: 0.9386
```

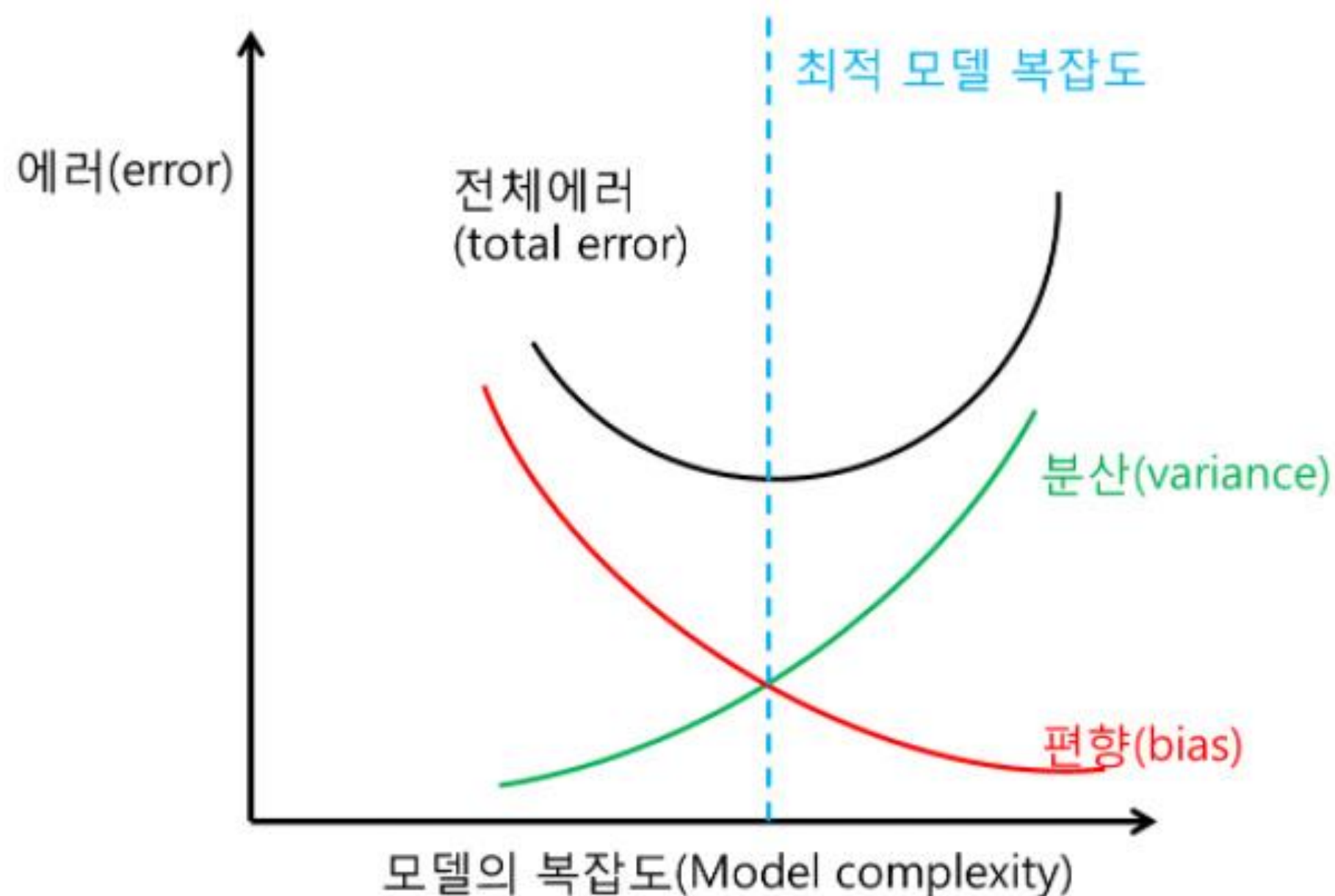
#ML 모델의 성능

- 보팅과 배깅 등의 앙상블 방법은 다른 단일 ML 알고리즘보다 예측 성능이 뛰어남
- 현실 세계는 다양한 변수와 예측이 어려운 규칙으로 구성
- 다양한 관점을 가진 알고리즘이 서로 결합하면 더 나은 성능을 실제 환경에서 끌어낼 수 있음

#ML 모델의 성능

- ML 모델은 어떻게 높은 유연성을 가지고 현실에 대처할 수 있는가가 중요한 평가요소

· '편향-분산 트레이드오프' 극복이 중요



#ML 모델의 성능

- 배깅과 부스팅

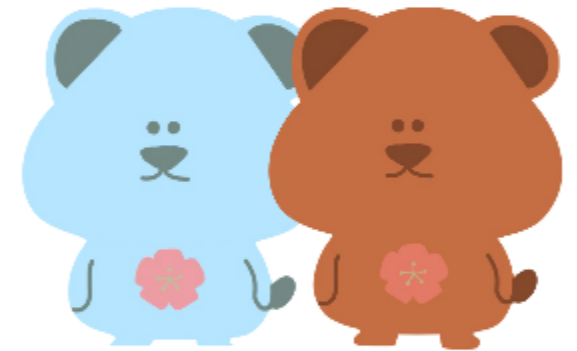
- 결정 트리 알고리즘 기반
- 예외 상황에 집착해 과적합 발생 가능성이 있음



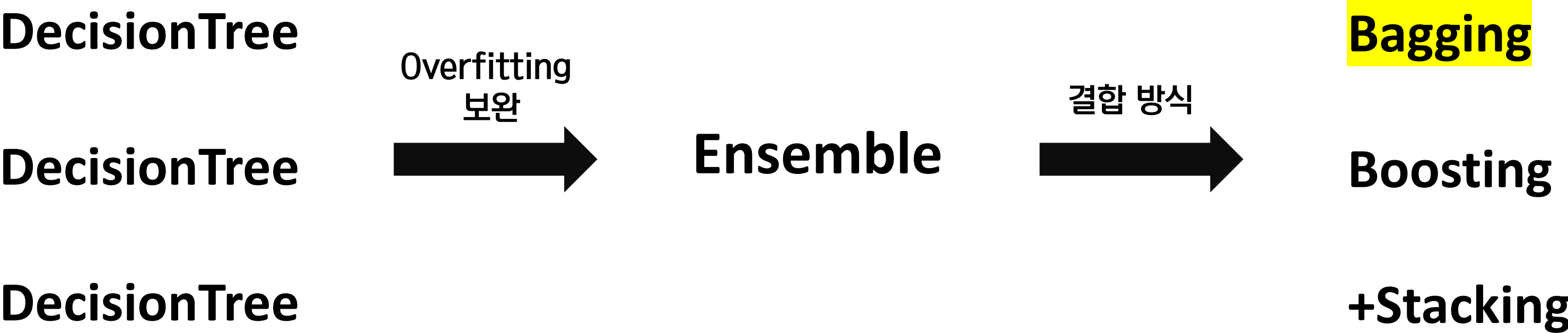
- 앙상블 학습에서는 매우 많은 분류기를 결합해 다양한 상황을 학습함으로써 극복
- 결정 트리의 단점인 과적합을 수십~수천개의 많은 분류기를 결합해 보완하고 장점인 직관적인 분류 기준은 강화됨

04 랜덤 포레스트 + SVM

2271034 신유진



랜덤 포레스트의 개요 및 실습



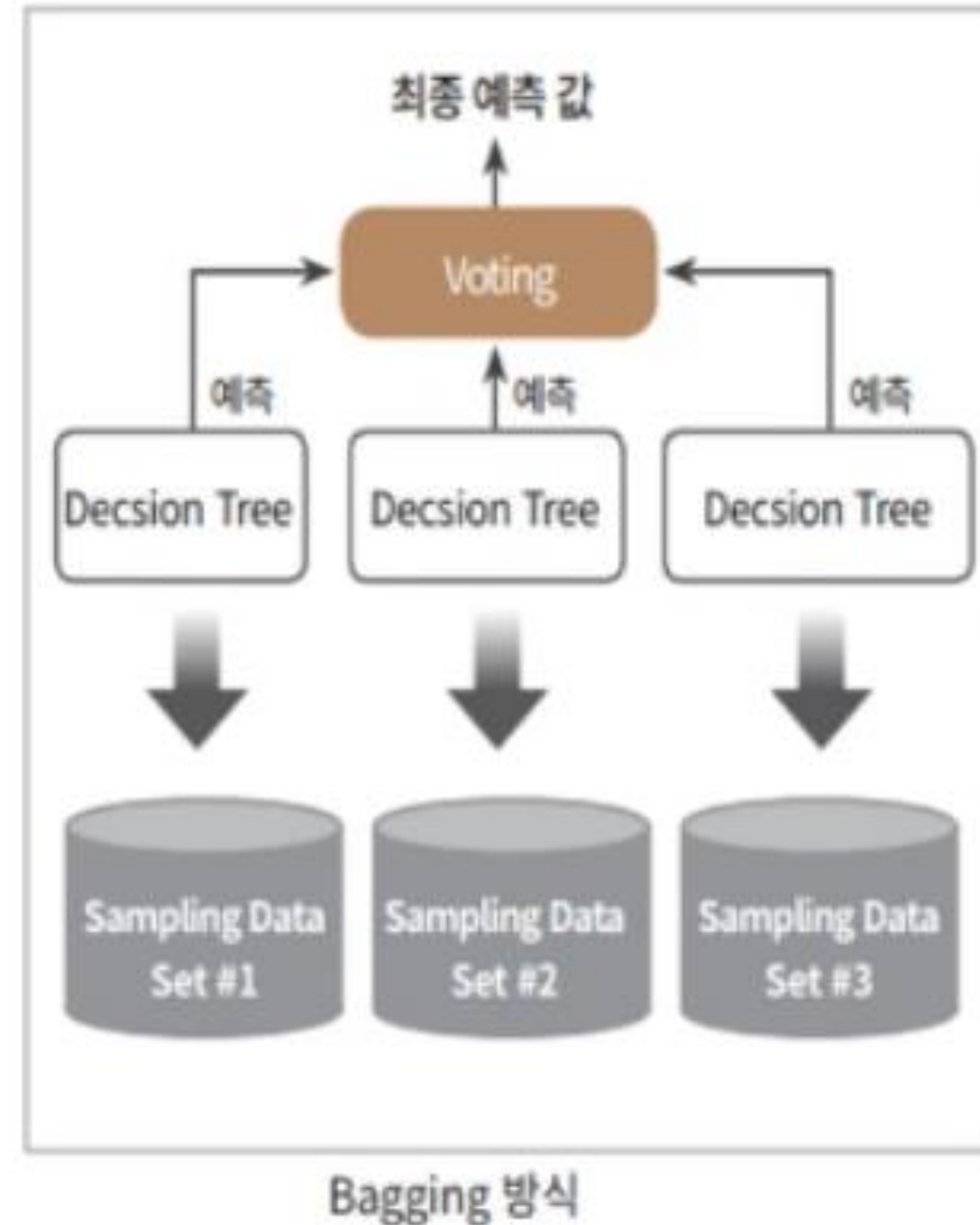
랜덤 포레스트의 개요 및 실습

#1 배깅(bagging)

- 같은 알고리즘, 다른 데이터 세트로 여러 개의 분류기 학습
- 보팅(voting)으로 최종결정
- 랜덤 포레스트(Random Forest)

#2 결정 트리 기반의 랜덤 포레스트

- 직관적
- 빠른 속도와 높은 예측 성능
- + 부스팅기반 앙상블 알고리즘



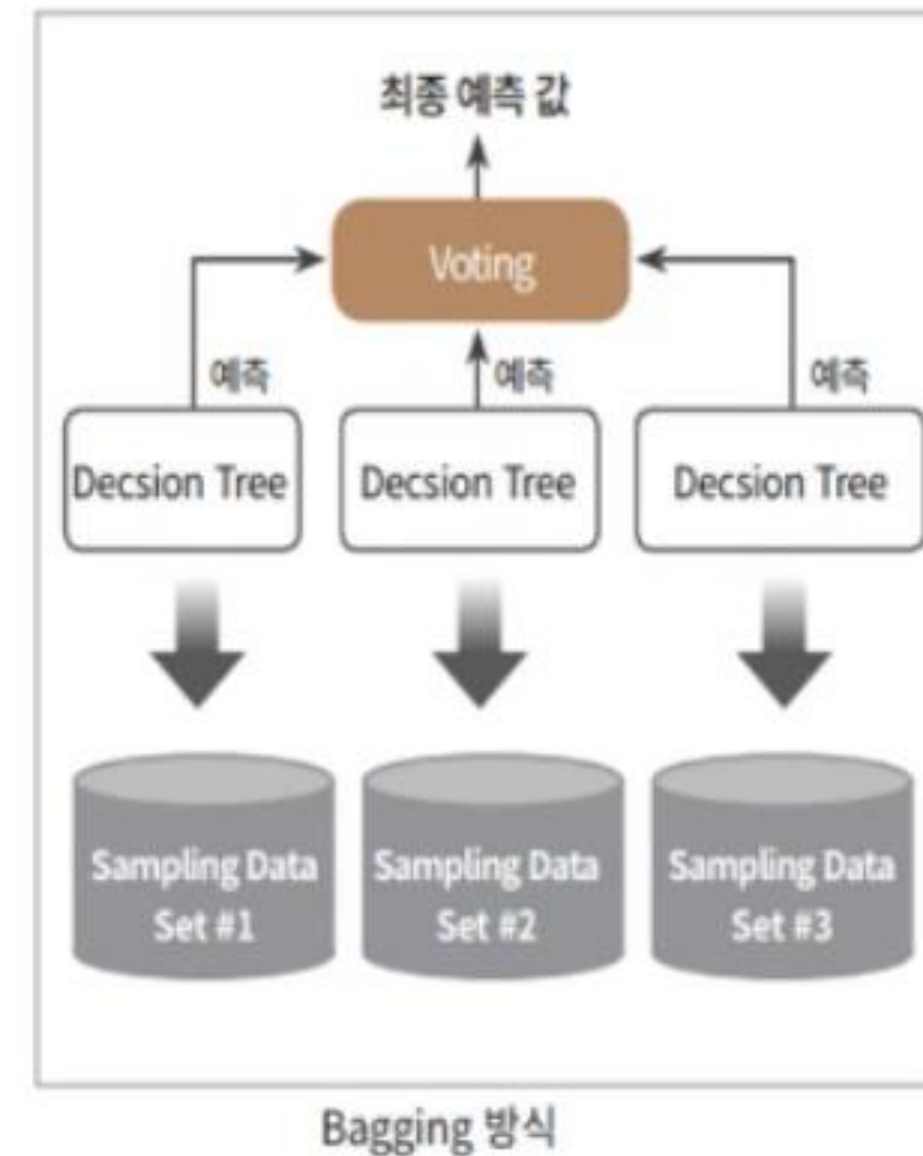
랜덤 포레스트의 개요 및 실습

#3 랜덤 포레스트

1) 배깅 - 데이터 샘플링

2) 개별 학습

3) 보팅 - 최종 결정



개별 분류기가 학습할 데이터 - 부트스트래핑(bootstrapping)

개별 분류기의 기반 알고리즘 : 결정 트리

랜덤 포레스트의 개요 및 실습

#4 부트스트래핑(bootstrapping) 분할 방식

: 랜덤 포레스트(**배깅**)의 데이터 샘플링

- **중복을 허용한 무작위 재추출**

: 일부가 중첩된 데이터 세트

- bagging = bootstrap aggregating

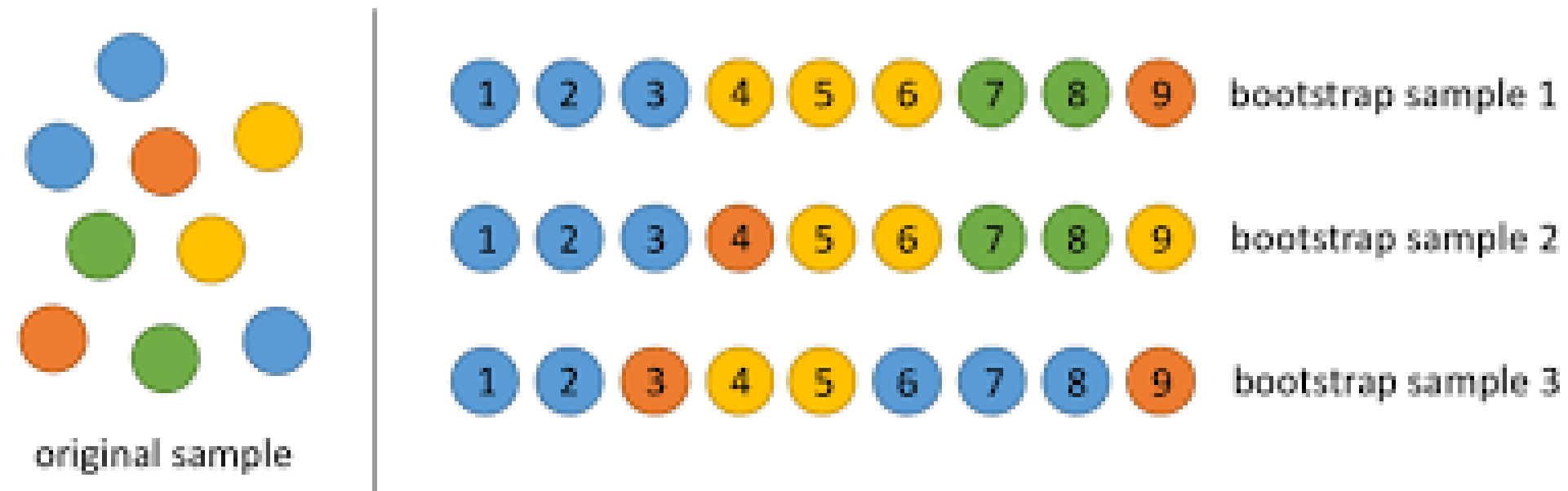
+ 통계학 bootstrap)

- 중복 허용된 작은 데이터 세트

> 각 데이터 세트의 통계량 계산

랜덤 포레스트의 개요 및 실습

- 원본 데이터 건수 9
- 3개의 결정 트리 기반 RandomForest
: n_estimators = 3



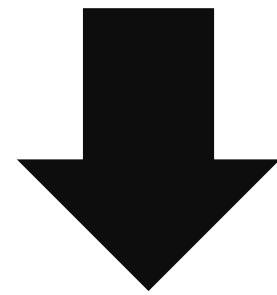
샘플링된 데이터 건수는 전체 데이터 건수와 동일,

개별 데이터 중첩

랜덤 포레스트의 개요 및 실습

Random Forest

Bootstrapping 방식으로 분할된 서브세트로
분류기 각각 학습



많은 분류기의 결과 결합으로

과적합에 대한 안정성 확보

이상치/예외 상황에 유연한 대처

랜덤 포레스트의 개요 및 실습

#5 sklearn-RandomForestClassifier, 실습

: 랜덤 포레스트 기반 분류 지원 클래스

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import pandas as pd
import warnings
warnings.filterwarnings('ignore')
```

앙상블 알고리즘 – 랜덤포레스트

경고문구 모두 무시

랜덤 포레스트의 개요 및 실습

```
# 결정 트리에서 사용한 get_human_dataset()을 이용해 학습/테스트용 DataFrame 반환
X_train, X_test, y_train, y_test = get_human_dataset()

# 랜덤 포레스트 학습 및 별도의 테스트 세트로 예측 성능 평가
rf_clf = RandomForestClassifier(random_state=0)
rf_clf.fit(X_train, y_train)
pred = rf_clf.predict(X_test)
accuracy = accuracy_score(y_test, pred)

print('랜덤 포레스트 정확도: {0:.4f}'.format(accuracy))
```

- get_human_dataset() > 사용자 행동 데이터 세트의 학습/테스트용 DataFrame 로딩
- 랜덤 포레스트 각 분류기 학습 후 정확도 측정(평균)

랜덤 포레스트 정확도 : 0.9108

랜덤 포레스트 하이퍼 파라미터 및 튜닝

#1 튜닝 : 트리 기반 앙상블 알고리즘의 단점

- 다수의 하이퍼 파라미터

트리 기반 자체

배깅, 부스팅, 학습, 정규화 등을 위한 파라미터

> 튜닝 시간 소모

```
(n_estimators: int = 100, *, criterion: str = "gini", max_depth: Any | None = None, min_samples_split: int = 2, min_samples_leaf: int = 1, min_weight_fraction_leaf: float = 0, max_features: str = "sqrt", max_leaf_nodes: Any | None = None, min_impurity_decrease: float = 0, bootstrap: bool = True, oob_score: bool = False, n_jobs: Any | None = None, random_state: Any | None = None, verbose: int = 0, warm_start: bool = False, class_weight: Any | None = None, ccp_alpha: float = 0, max_samples: Any | None = None) → None
```

A random forest classifier.

Hyperparameter	Meaning	Default or recommended values
num.trees	The number of decision trees that constitute the forest	500
importance	The variable importance measure based on type of problem. For classification, when you set it to impurity, the default used is Gini Index	recommended for classification: "impurity"
mtry	The number of features to be considered for each tree	auto
splitrule	The rule by which each split is considered in a tree, the impurity measure to separate to one class from another in our target variable.	Gini
max.depth	The maximum depth of the tree i.e. the maximum number of edges from root node to terminal node. This regularizes for overfitting in each tree.	Default is no limit and 5 is recommended.
min.node.size	Min number of instances in terminal node Equivalent to setting depth of tree in Python	Default 1 for classification, 5 for regression, 3 for survival, and 10 for probability.
replace	Sampling with replacement (Bootstrapping) or not	True (use full dataset for bootstrapping)
sampsize	The sample size for each for bootstrap sample.	if (replace) nrow(x) else ceiling(.632*nrow(x)) If replacement is true, then all rows of x are used. Else 63.2% of the training data is bootstrapped every time.

[Hyperparameter tuning of Random Forest in both R and Python | by Ammu Bharat Ram | Medium](#)

2023-09-17 검색

랜덤 포레스트 하이퍼 파라미터 및 튜닝

#2 주요 하이퍼 파라미터

- n_estimators

결정 트리 개수 지정

학습시간 고려 필요

default = 100 교재 오류?

- max_feature

트리 분할 시 참조할 피쳐 수

default = 'auto' ('sqrt', \sqrt{n})

(결정 트리 : 'None')

- max_depth , min_samples_leaf

```
RandomTreeClassifier(n_estimators,  
  
max_depth, min_samples_leaf,  
  
min_samples_split, random_state)
```

랜덤 포레스트 하이퍼 파라미터 및 튜닝

#3 GridSearchCV 하이퍼 파라미터 튜닝, 실습

- n_estimators = 100(default)
- cv = 2 : 교차 검증 2회로
- > 시간 절약 위해, 파라미터 최적화 후
n_estimator 재설정
- n_jobs = -1 : 멀티 코어 환경, 모든 코어 사용
- > (RandomForest) CPU 병렬 처리 효과적 수행 가능

```
params={  
    'n_estimators':[100],  
    'max_depth': [6,8,10,12],  
    'min_samples_leaf': [8,12,18],  
    'min_samples_split': [8,16,20]  
}
```

(파라미터별) $1*4*3*3 = 36$ 회 수행

28 CPU Core 기반 리눅스 환경 - 6초

개인 노트북 - 약 5분

랜덤 포레스트 하이퍼 파라미터 및 튜닝

```
# RandomForestClassifier 객체 생성 후 GridSearchCV 수행
rf_clf = RandomForestClassifier(random_state=0, n_jobs=-1)
grid_cv = GridSearchCV(rf_clf, param_grid=params, cv=2, n_jobs=-1)
grid_cv.fit(X_train, y_train)

print('최적 하이퍼 파라미터:\n', grid_cv.best_params_)
print('최고 예측 정확도: {0:.4f}'.format(grid_cv.best_score_))
```

최적 하이퍼 파라미터:

```
{'max_depth': 10, 'min_samples_leaf': 8, 'min_samples_split': 8, 'n_estimators': 100}
```

- GridSearchCV이용해 미리 설정한 params의 하이퍼 파라미터 조합별로 학습 수행

n_jobs = -1 추가

Best_score_ 최적 하이퍼 파라미터 반환

> 최고 예측 정확도 : 0.9168

랜덤 포레스트 하이퍼 파라미터 및 튜닝

- 확인한 최적 파라미터로 RandomForestClassifier 재설정

```
rf_clf1 = RandomForestClassifier(n_estimators = 300, max_depth=10, min_samples_leaf=8, \
                                min_samples_split=8, random_state=0)
```

다시 학습 수행 후, 정확도 측정

```
rf_clf1.fit(X_train, y_train)
pred = rf_clf1.predict(X_test)
print('예측 정확도: {0:.4f}'.format(accuracy_score(y_test, pred)))
```

예측 정확도: 0.9165

랜덤 포레스트 하이퍼 파라미터 및 튜닝

- 피쳐 중요도 시각화 : seaborn.barplot

RandomForestClassifier.feature_importance 속성 > 피쳐 중요도 확인

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

ftr_importances_values = rf_clf1.feature_importances_
ftr_importances = pd.Series(ftr_importances_values, index = X_train.columns)
ftr_top20 = ftr_importances.sort_values(ascending=False)[:20]

plt.figure(figsize=(8,6))
plt.title('Feature importances Top 20')
sns.barplot(x=ftr_top20, y=ftr_top20.index)

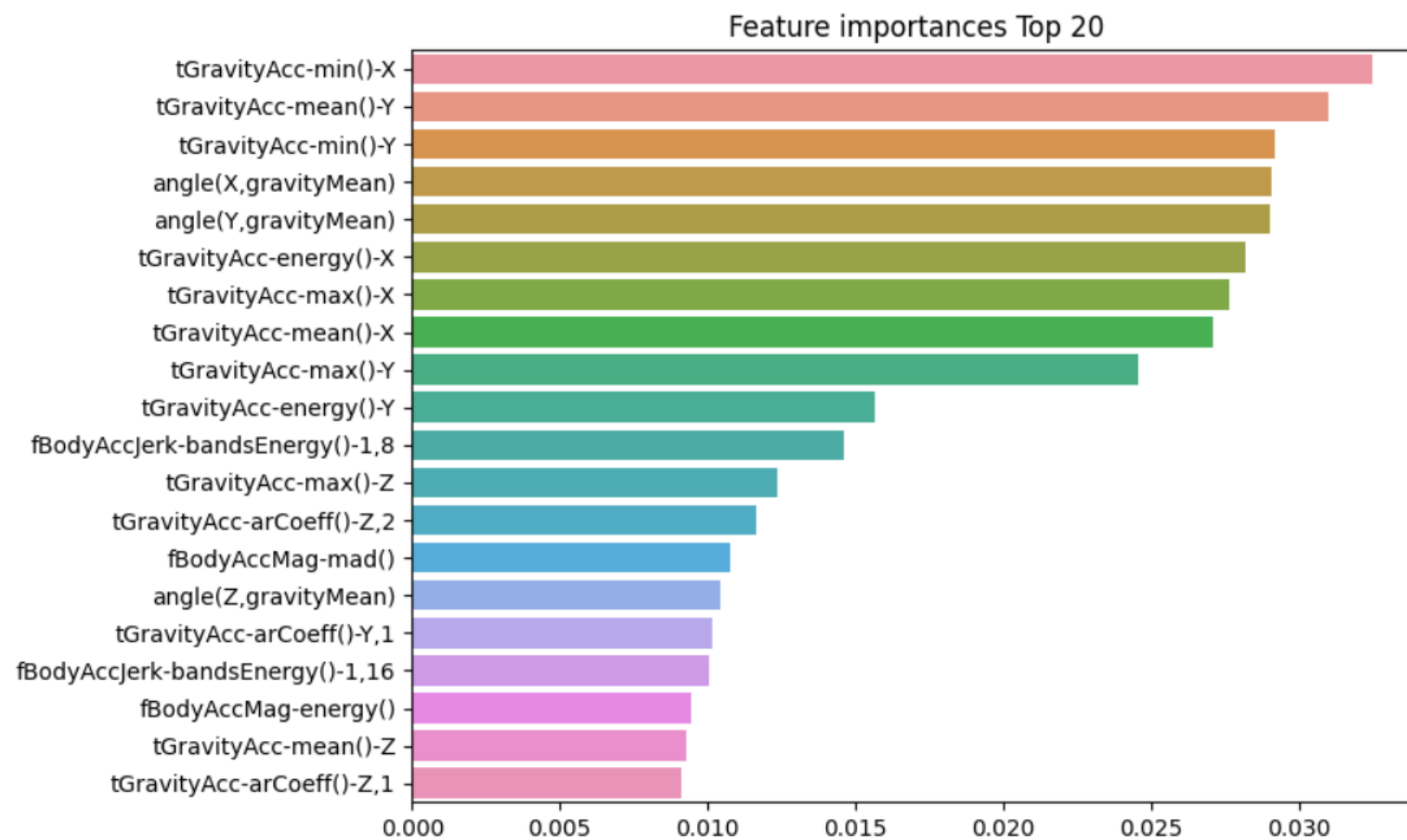
plt.show()
```

%matplotlib inline 코드로 코랩에서 그래프 즉시 확인

랜덤 포레스트 하이퍼 파라미터 및 튜닝

- `.sort_values(ascending=False)[:20]`

: 중요 20개 피쳐 내림차순으로 정렬하여 그래프 그리기



tGravityAcc-min()-X
tGravityAcc-mean()-X
tGravityAcc-min()-X

...

%matplotlib inline 코드로 코랩에서 즉시 확인한 그래프

SVM(Support Vector Machine)

#0 SVM

- 분류, 회귀 위한 지도학습 모델

SVC : 분류 서포트 벡터 머신, SVR : 회귀 서포트 벡터 머신

- 이진 분류에 좋은 성능
- Decision boundary(결정경계) 정의, 이때 서포트 벡터(support vector) 사용

서포트 벡터 : 군집의 중심 ~ 클래스 구분 경계까지의 거리 정보를 담는 데이터 포인트

두 군집으로부터 동일한 거리

- (python) sklearn, PyML, LIBSVM
- Linear SVM (선형 모델) / NonLinear SVM (비선형 모델)

SVM(Support Vector Machine)

#1 Linear SVM Classification

- 선형 Decision boundary
- street : 결정경계에 평행한 선
- large margin classification : street 간격 최대로 fit

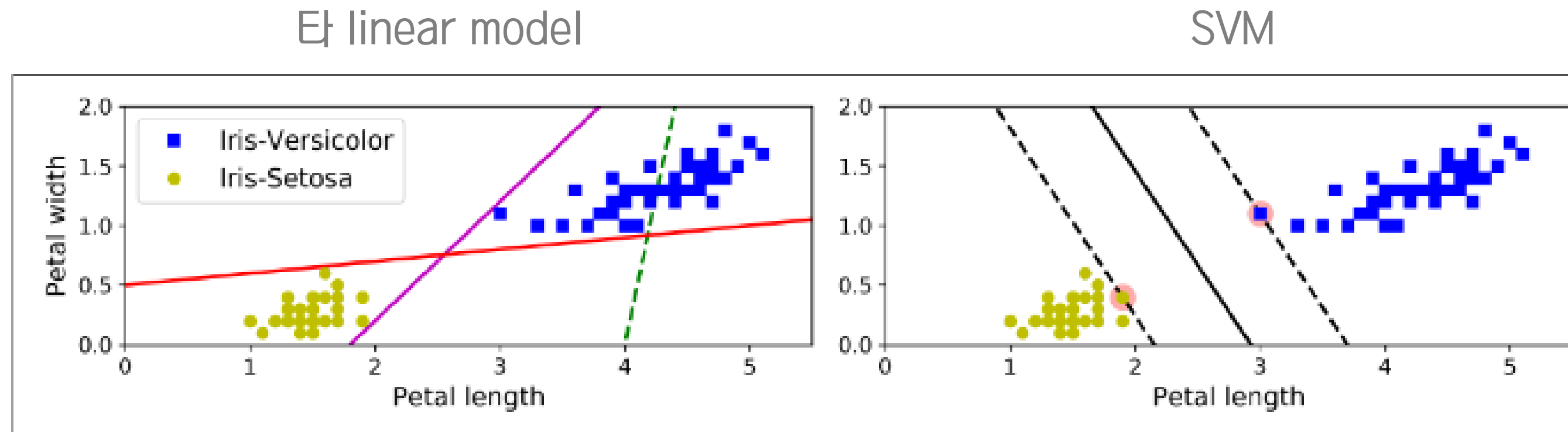


Figure 5-1. Large margin classification

SVM(Support Vector Machine)

#2 SVM의 스케일 민감성

- 특징점(feature)의 스케일(최대~최소 범위, x/y축)이 결과에 큰 영향

> 적절한 **스케일링 필요성**

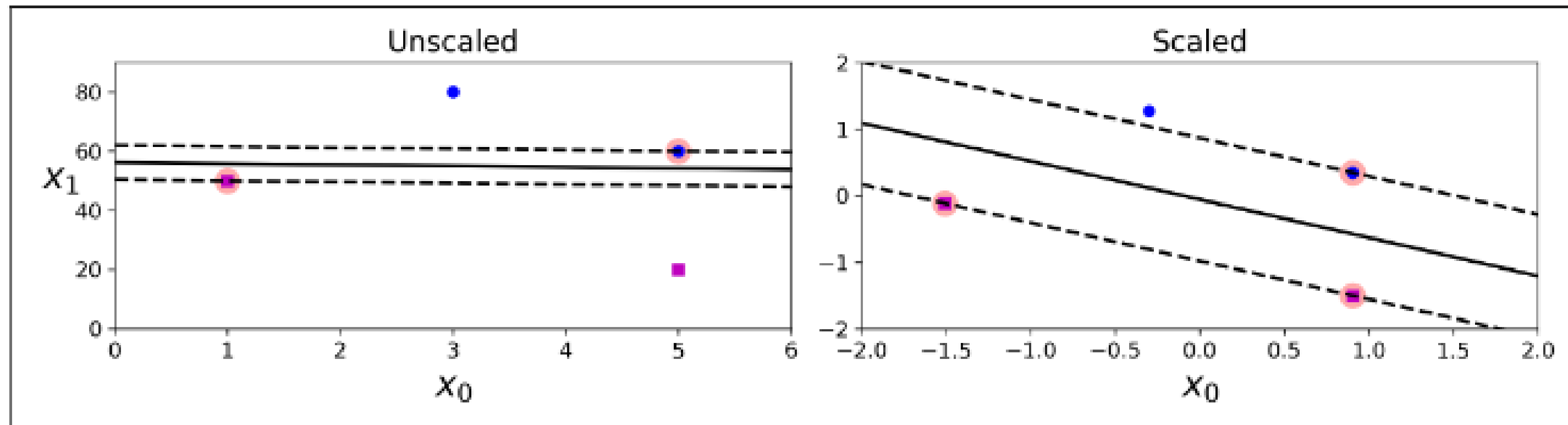


Figure 5-2. Sensitivity to feature scales

SVM(Support Vector Machine)

#3 Hard Margin Classification : 모든 데이터를 이분하는 엄격한 결정 선

> 직선으로 분류되어야 함

> outlier(이상치) 에 민감

새로운 데이터에 적용 어려움

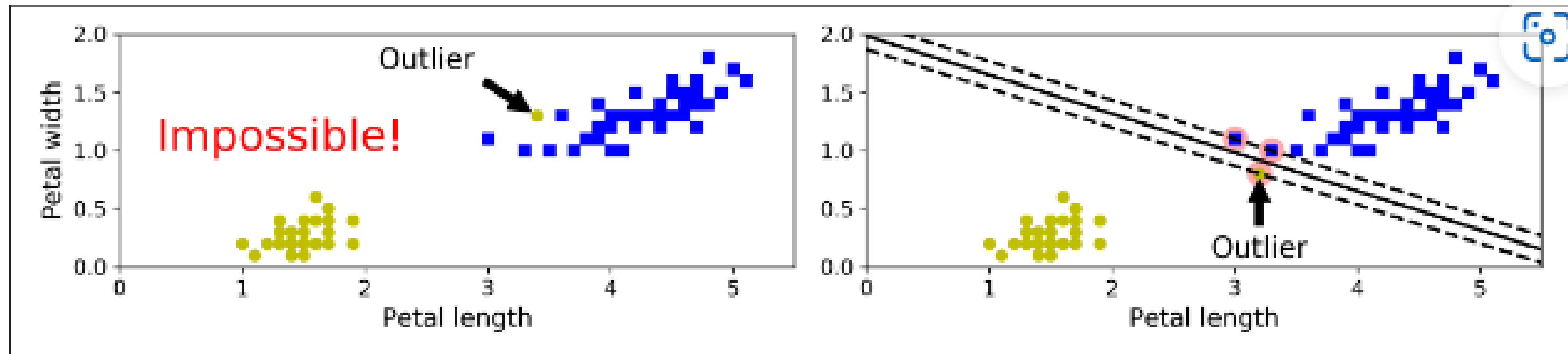


Figure 5-3. Hard margin sensitivity to outliers

SVM(Support Vector Machine)

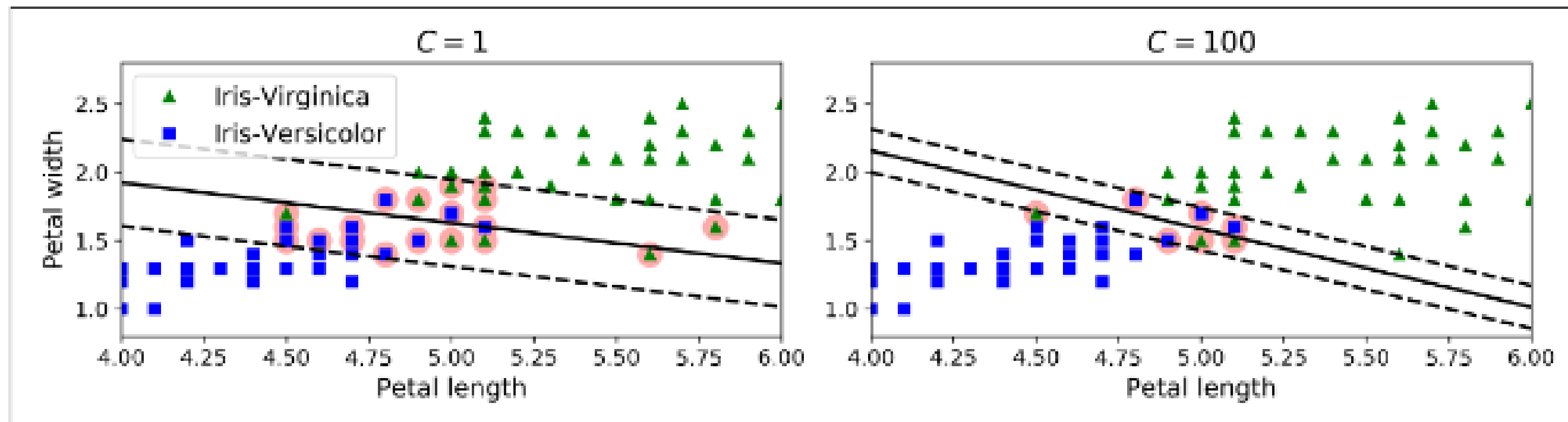
#4 Soft margin classification

for) street 최대 간격 유지 및 margin violations 제한

- **parameter C** > street 간격 및 margin violation 수 조절

> hard 보다 새로운 데이터에 유연하게 적용

C 작으면
Street 넓고
m.v. 많아짐



C 크면
Street 좁고
m.v. 적어짐

Figure 5-4. Large margin (left) versus fewer margin violations (right)

SVM(Support Vector Machine)

#4 Soft margin classification

- (python) sklearn.svm 의 **LinearSVC**로 구현

Logistic Regression과 달리 확률 제공

- + SVC(kernal= 'linear', C1) 또는

SGDClassifier(loss="hinge", alpha=1/(m*C)) 큰 데이터셋에 유리 로도 구현 가능

```
from sklearn.svm import LinearSVC
```

```
svm_clf = Pipeline([  
    ("scaler", StandardScaler()),  
    ("linear_svc", LinearSVC(C=1, loss="hinge")),  
])
```

```
svm_clf.fit(X,y)
```

SVM(Support Vector Machine)

#5 Nonlinear Support Vector Machine

: 직선으로 분류되지 않는 경우, 효과적인 분류 필요한 경우

차원을(분류에 고려할 feature) 추가 (2차원 > 3차원에 매핑 – 가우시안, RBF 커널 등 사용)

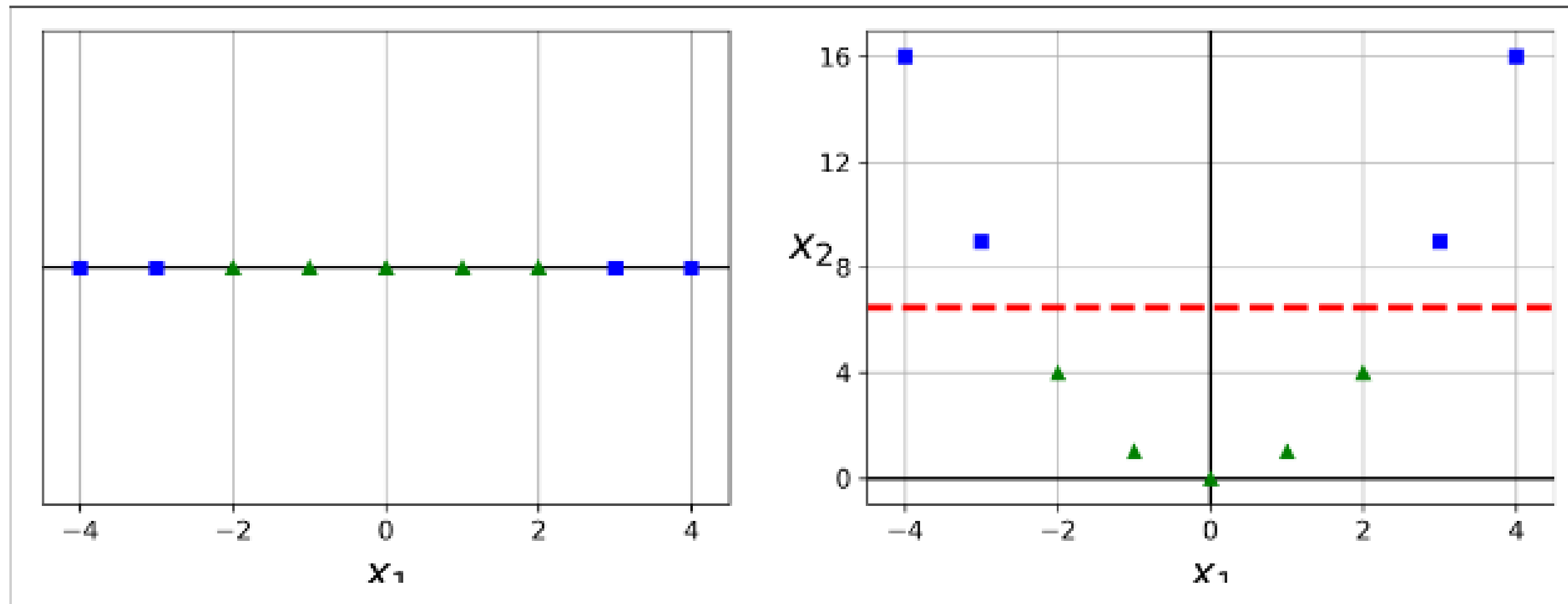


Figure 5-5. Adding features to make a dataset linearly separable

SVM(Support Vector Machine)

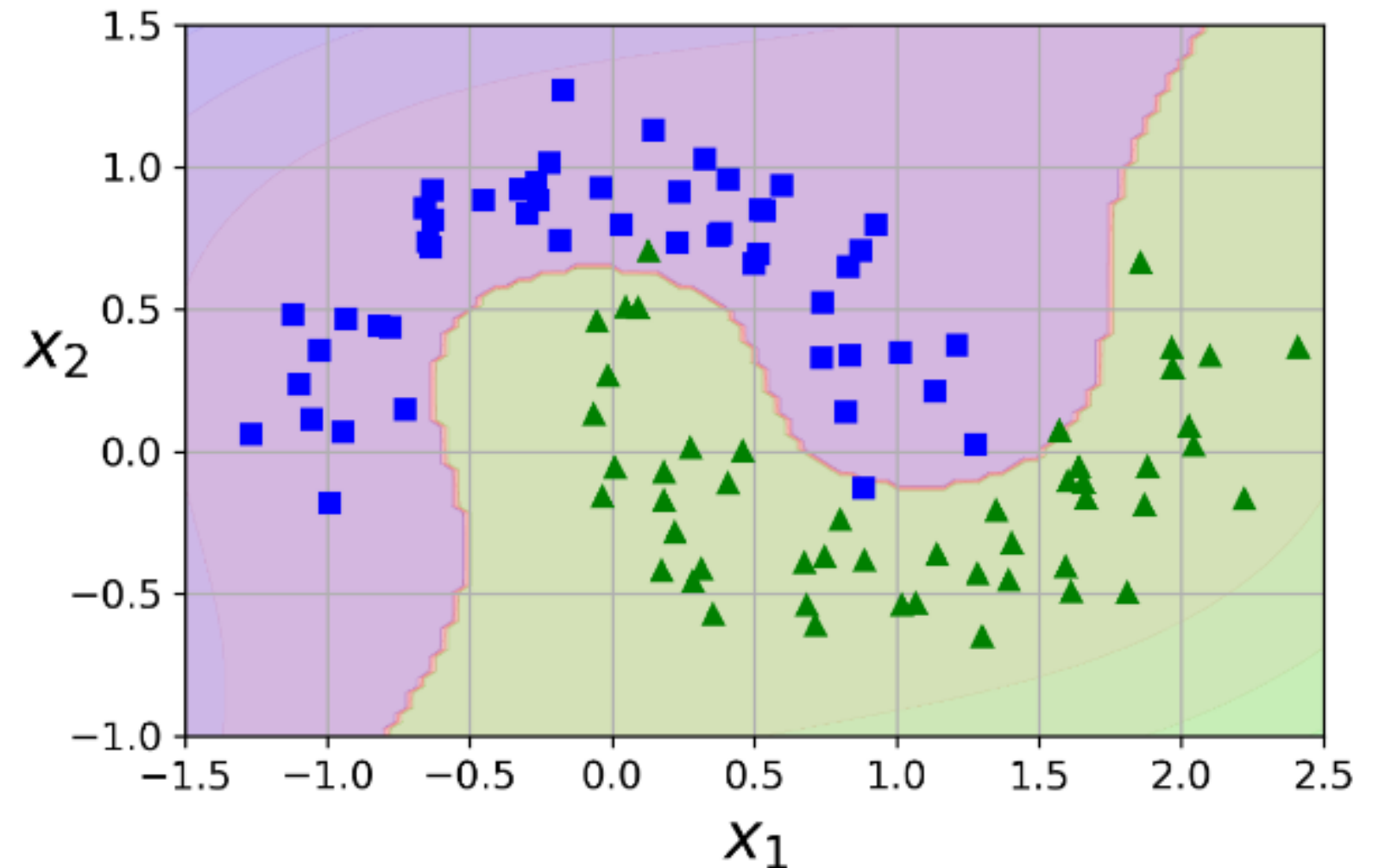
#5 Nonlinear Support Vector Machine

- Pipeline = PolynomialFeatures transforer + StandardScaler + LinearSVC

만들어 구현

```
poly_svm_clf = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scaler", StandardScaler()),
    ("svm_clf", LinearSVC(C=10, loss="hinge"))
])

poly_svm_clf.fit(X,y)
```



3차원으로 만들어진 곡선 경계

SVM(Support Vector Machine)

For Nonlinear Support Vector Machine

#6 Polynomial Kernel : Polynomial degree

- Polynomial degree : 데이터에 fit한 모델 Overfitting 가능성

> 모델 느려짐

**** kernel trick **** 제공

feature 추가 X

feature 추가한 것과 같은 결과

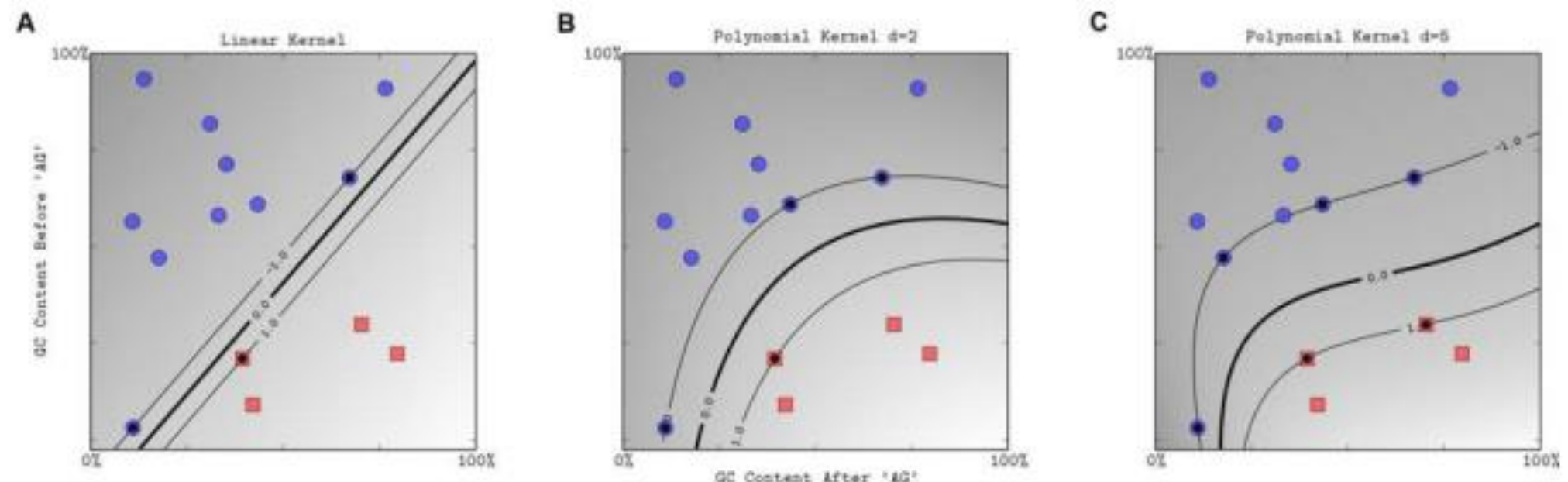


Figure 6. The effect of the degree of a polynomial kernel. The polynomial kernel of degree 1 leads to a linear separation (A). Higher-degree polynomial kernels allow a more flexible decision boundary (B,C). The style follows that of Figure 3.
doi:10.1371/journal.pcbi.1000173.g006

<https://doi.org/10.1371/journal.pcbi.1000173> 2023-09-17검색

: 고차원 매핑과 내적을 한번에

저차원 > 고차원 + 고차원 > 저차원

SVM(Support Vector Machine)

For Nonlinear Support Vector Machine

#7 Adding Similarity Features

: **Similarity functions** 사용 하여 특징점 추가 > **landmark** 와의 유사성 측정하여 분류

Similarity functions : 유사도 결정 함수

landmark : 유사도 결정 기준 인스턴스

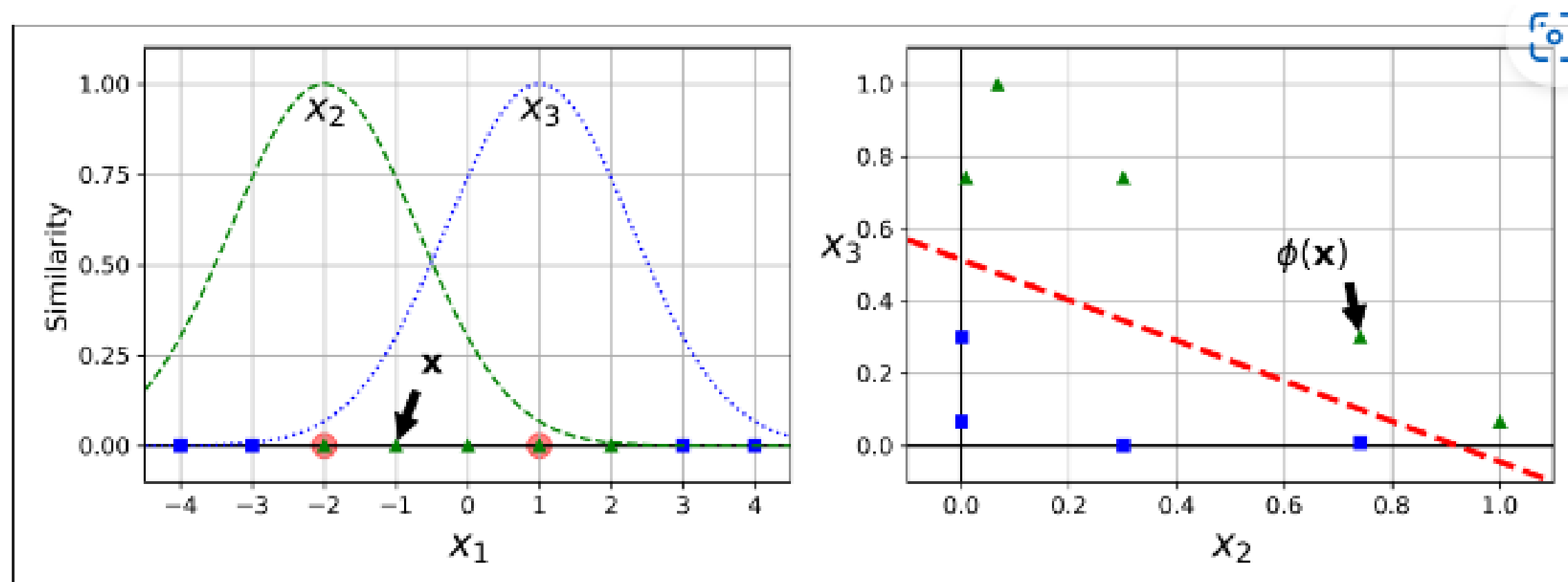


Figure 5-8. Similarity features using the Gaussian RBF

SVM(Support Vector Machine)

For Nonlinear Support Vector Machine

#8 Gaussian RBF Kernel 사용

- similarity func. 많은 연산으로 소요시간 많아짐

> kernel trick 사용

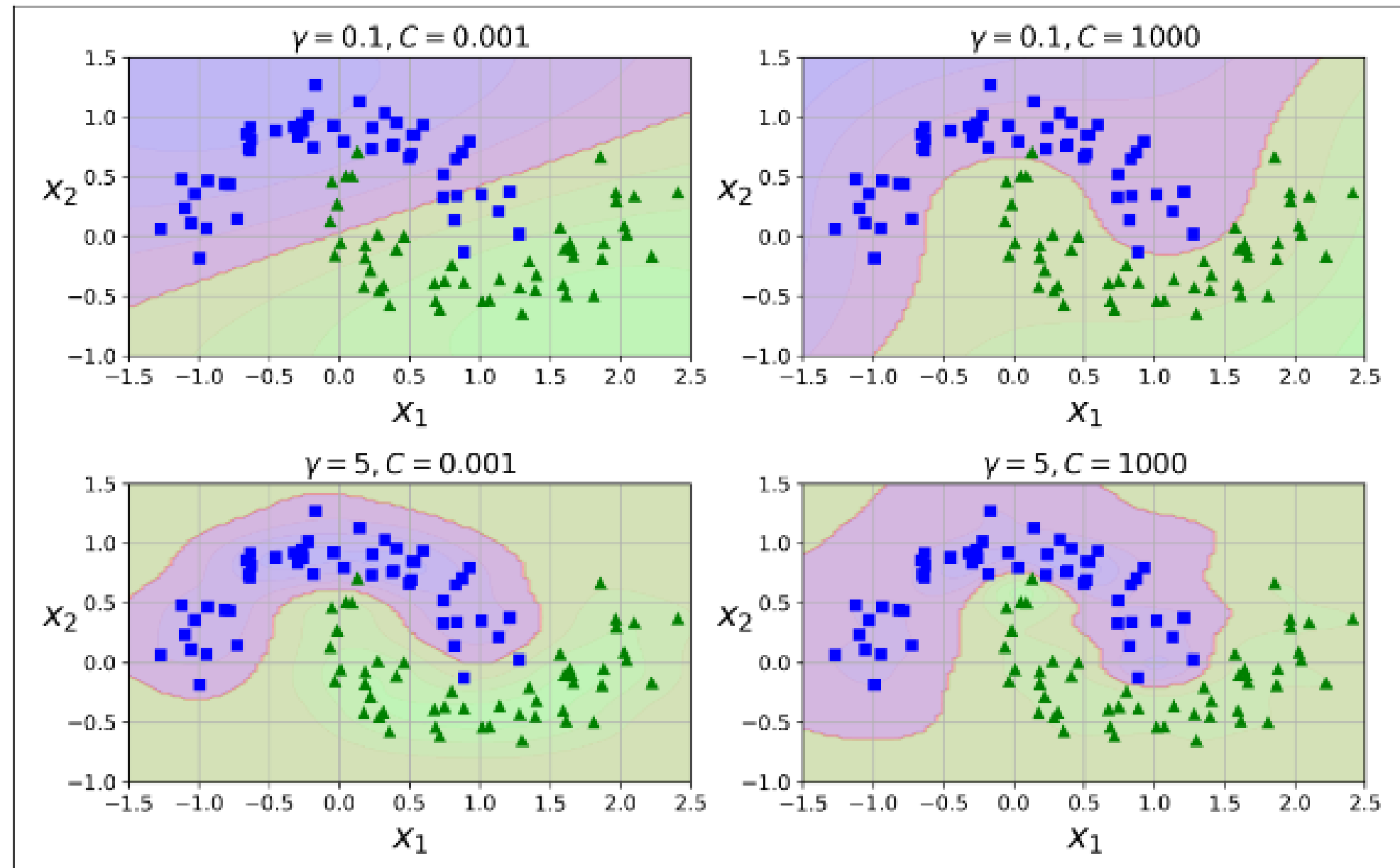
> (parameter) C

> gamma 조절

: 앞 함수의 벨 크기 조절

커질수록 벨 너비 좁아져

overfitting



SVM(Support Vector Machine)

For Nonlinear Support Vector Machine

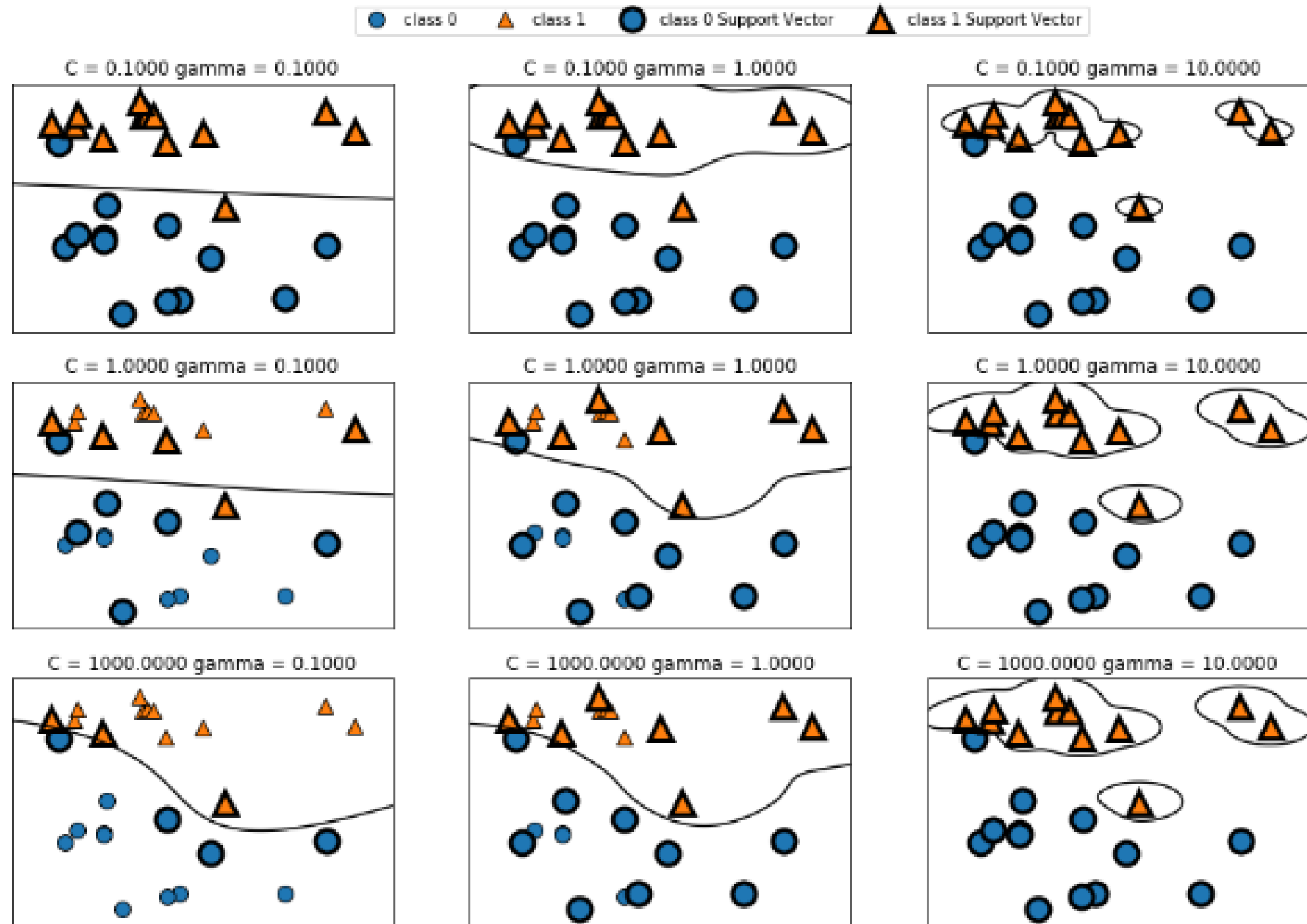
#8 Gaussian RBF Kernel 사용

- gamma 변수

: 하나의 데이터가
영향을 미치는 범위

- C 변수

: 결정경계가 데이터에
맞춰지는 정도



THANK YOU

