Week3_분류(4.1~4.4)

분류의 개요

✏️ **분류 (Classification)**

: 학습 데이터의 피처(X)와 레이블(y)값을 머신러닝 알고리즘으로 학습
→ 예측 모델 생성
→ 테스트 값에 대해 예측 수행

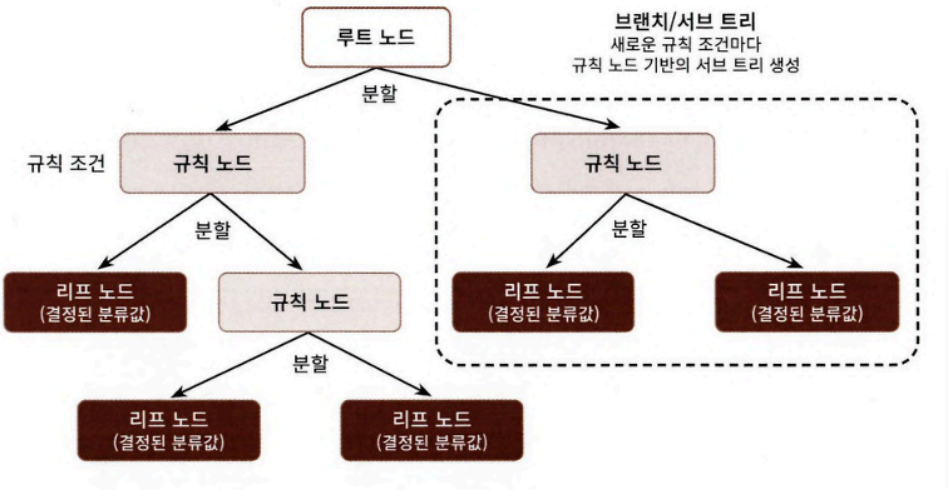
- *분류 머신러닝 알고리즘*
  Ex) 나이브 베이즈, 로지스틱 회귀,
  결정 트리, 서포트 벡터 머신, 최소 근접 알고리즘, 신경망, 앙상블
- **앙상블 (Ensemble)**: 알고리즘 여러 개를 결합한 것
  - 배깅(Bagging) ex. 랜덤 포레스트
  - 부스팅(Boosting) ex. 그래디언트 부스팅 (Gradient Boosting)

    — XgBoost, LightGBM 등의 알고리즘의 등장으로 기존 그래디언트 부스팅의 단점(수행 시간)이 보완되면서 정형 데이터 분류 영역에서 활용도가 높아지고 있음

결정 트리 (Decision Tree)

: 데이터에 있는 규칙(if-else)을 학습을 통해 자동으로 찾아내 트리 기반의 분류 규칙을 만드는 것

- 결정 트리의 **구조**



  - 규칙 노드 (Decision Node): 데이터 규칙 조건
  - 리프 노드 (Leaf Node): 최종 클래스(레이블) 값이 결정되는 노드
    — 조건: 하나의 클래스 값으로 최종 데이터 구성 / 리프 노드가 되는 하이퍼 파라미터 조건 충족

⚠️ 많은 규칙
→ 트리의 깊이가 깊어짐 = 분류 결정 방식이 복잡해짐

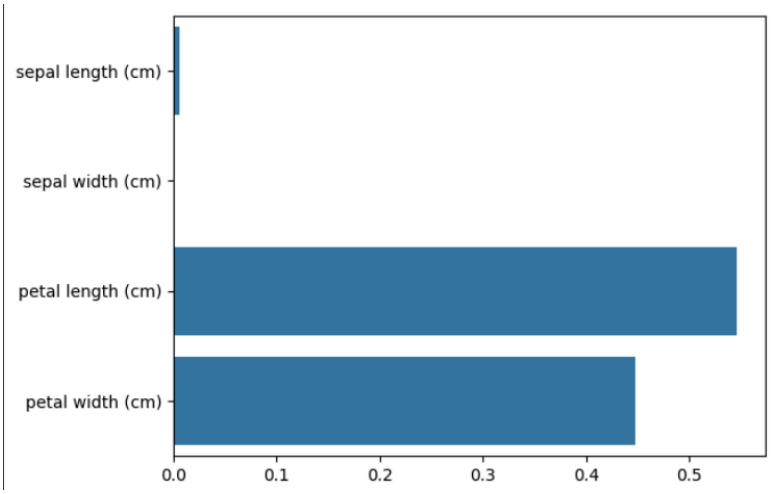
→ 과적합으로 인해 결정 트리의 예측 성능이 저하될 가능성 ⬆️
⇒ 가능한 적은 결정 노드로 높은 예측 정확도 만들기

= 정보 균일도가 높은 데이터 세트를 선택하는 규칙 조건으로 트리를 분할하기

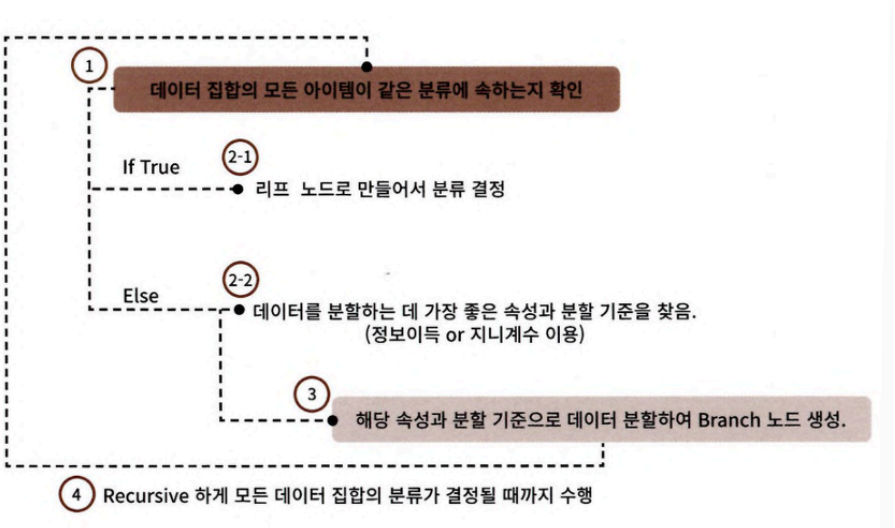
- **정보 균일도** ⬆️ = 데이터를 구분하는 데 필요한 정보의 양 ⬇️
  - 정보 균일도 측정 방법
    - 정보 이득 (Information Gain) 지수: 1 - (엔트로피지수)

      데이터가 균일할수록 엔트로피가 낮음 → 정보 이득 지수가 높음
    - **지니 계수**: 불평등 지수를 나타낼 때 사용하는 계수 - 0: 가장 평등 1: 가장 불평등

      데이터가 균일할수록 지니 계수가 낮음
  - DecisionTreeClassifier 객체의 `feature_importances_` 속성
    : ⬇️ 결정 노드 ⬆️ 정확도를 갖는 규칙을 정하기 위한
    각 피처 별 중요도를 ndarray로 반환
    : 해당 피처가 정보 이득/지니 계수를 얼마나 효율적으로 개선시켰는지에 대한 정규화된 값



column별 feature_importances_를 시각화한 것
→ petal length가 가장 중요한 피처임을 알 수 있다



결정 트리의 분류 결정 과정

결정 트리 모델의 특징

👍 장점

- 알고리즘이 쉽고 직관적: '균일도'라는 룰이 명확함
- 데이터 전처리 작업이 필요X

👎 단점

- 과적합으로 알고리즘 성능이 떨어짐

  ⇒ 트리의 크기를 사전에 제한 필요

결정 트리 파라미터

** 사이킷런 결정 트리 분류 - `DecisionTreeClassifier`, 회귀 - `DecisionTreeRegressor`

→ CART(Classification And Regression Trees) 알고리즘 기반

min_sample_split	- **노드를 분할하기 위한 최소한의 샘플 데이터 수** - 과적합 제어: min_sample_split ⬇️ → 분할 노드 ⬆️ →과적합 가능성 ⬆️ - default=2
min_samples_leafs	- **분할 후 자식 노드의 최소한의 샘플 데이터 수** - 과적합 제어: min_samples_leafs ⬆️ → 노드 분할 ⬇️ - 비대칭적(imbalanced) 데이터의 경우에는 작게 설정해야 함
max_features	- **최적 분할을 위해 고려할 최대 피처 개수** - int 지정: 대상 피처 개수 / float 지정: 대상 피처의 % - default=None: 데이터 세트의 모든 피처를 사용해 분할 - 'sqrt'=sqrt(전체 피처 개수) - 'auto'='sqrt' - 'log'=log(전체 피처 개수)
max_depth	- **트리 최대 깊이** 규정 - default=None: 완벽한 클래스 결정 값이 되거나 노드의 데이터 개수가 min_samples_split보다 작아질 때까지 깊이 증가 = 최대 분할
max_leaf_nodes	- **말단 노드의 최대 개수**

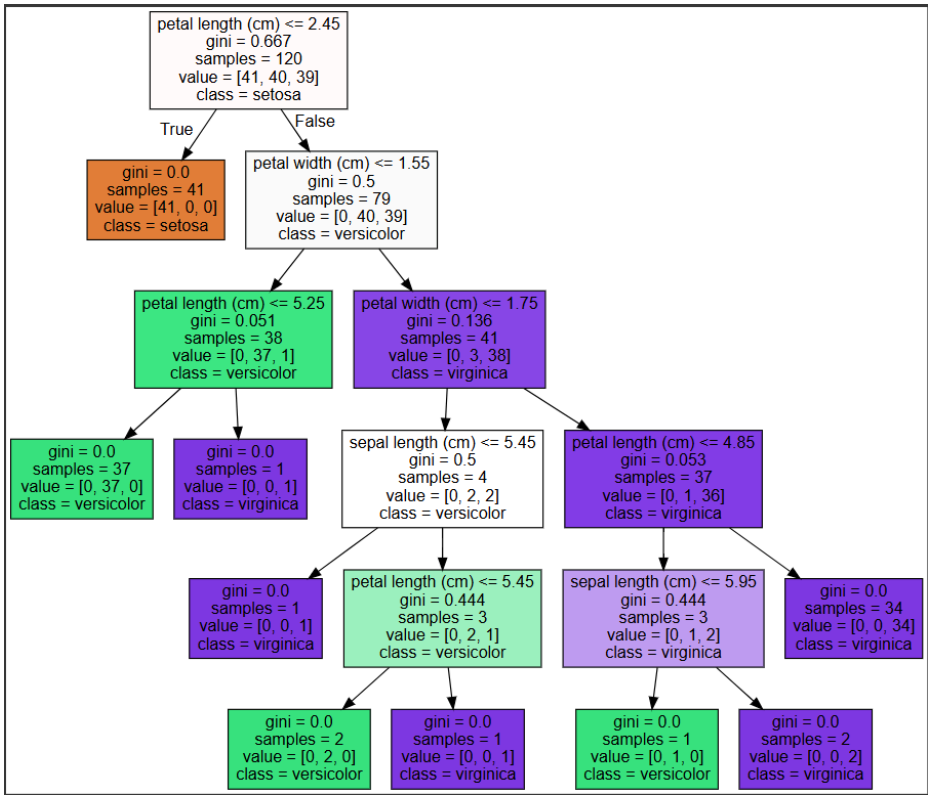
결정 트리 모델의 시각화

`export_graphviz()`: Graphviz API

- 파라미터: Estimator, 피처 이름 리스트, 레이블 이름 리스트
- 학습된 결정 트리 규칙 → 트리 형태로 시각화

export_graphviz()로 결정 트리를 시각화하기

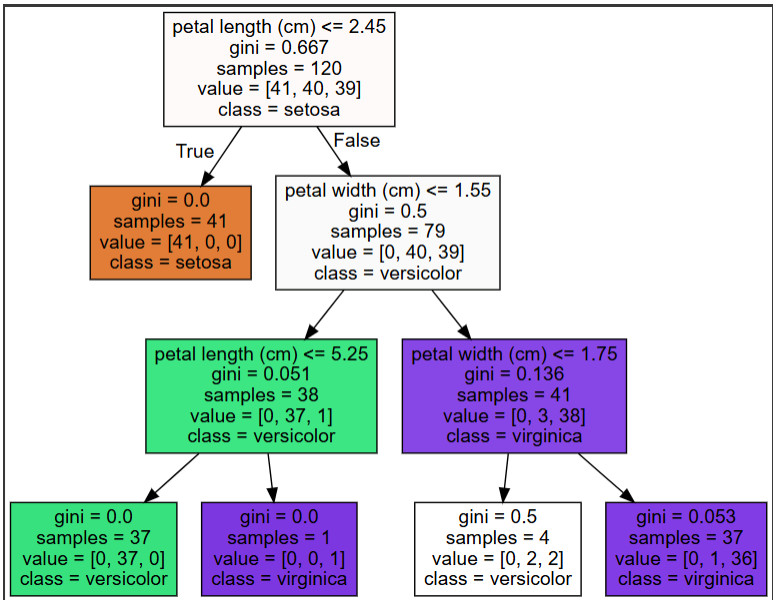
- 과적합 제어하지 않은 결정 트리



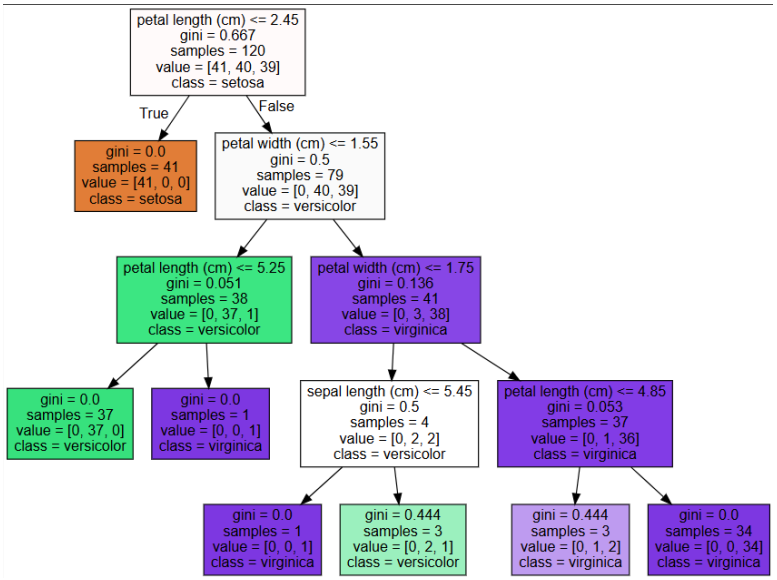
- `petal length(cm) ≤ 2.45` 등: 자식 노드를 만들기 위한 피처 규칙 조건. 조건이 없으면 리프 노드
- `gini`: value=[]로 주어진 데이터 분포에서의 지니 계수
- `samples`: 현 규칙에 해당하는 데이터 건수
- `value` = []: 클래스(레이블) 값 기반 데이터 건수 *(각 레이블 별 데이터 건수)*
- 노드 색: 붓꽃 데이터의 레이블 값. 색이 짙을수록 지니 계수 ⬇️, 해당 레이블의 샘플 데이터 ⬆️

노드 번호	1	2	3
samples	120개 : 전체 데이터 수	41개 : setosa 클래스로 결정된 리프 노드	79개 : 분할이 더 필요한 데이터
value	[41, 40, 39]	[41, 0, 0] : setosa만 존재	[0, 40, 39] : setosa를 제외한 레이블
gini	0.667	0.0 : 레이블이 하나 → 완전 균등	0.5 : 레이블이 2개
class	setosa : 하위 노드를 가질 경우 setaso의 개수가 41개로 제일 많음	setosa	versicolor

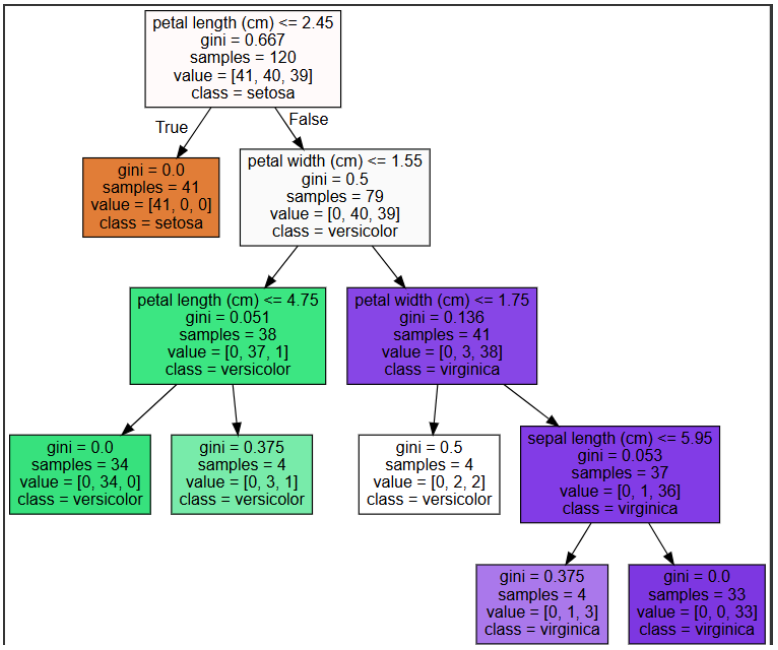
- 파라미터로 과적합 제어된 결정 트리
  1. `max_depth=3`
     → 트리 깊이가 3일 때 노드 생성 멈춤



2. min_samples_split=4
   → 샘플 개수가 4개 미만일 때는 더이상 노드를 생성하지 않음



3. min_samples_leaf=4
   → 자식 노드의 샘플 개수가 4개 미만이 될 때는 생성하지 않음

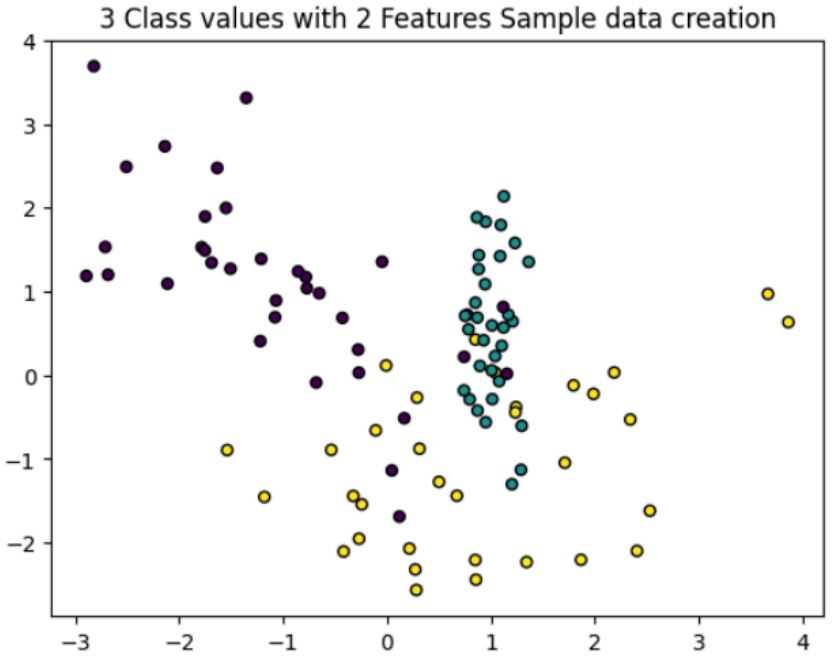


결정 트리 과적합(Overfitting)

결정 트리의 과적합 문제 확인하기

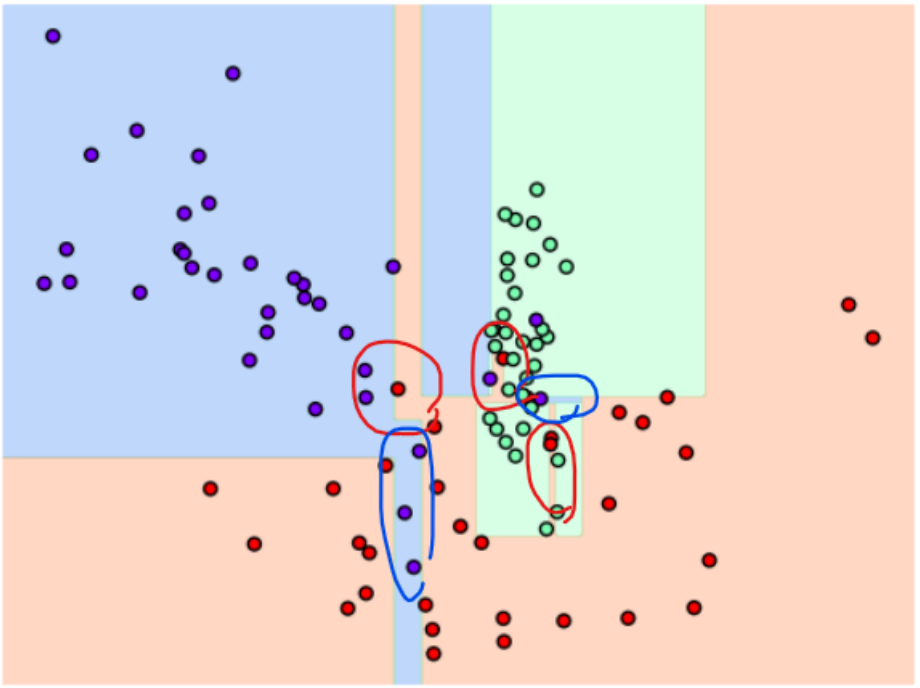
분류용 데이터 생성

```
# 분류용 데이터 세트 생성
from sklearn.datasets import make_classification
import matplotlib.pyplot as plt
%matplotlib inline

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

# 피처 2개 클래스 3가지 유형의 분류 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)
plt.scatter(X_features[:, 0], X_features[:, 1], marker='o', c=y_labels, s=25, edgecolor='k')
```



1. 디폴트 DecisionTreeClassifier가 분류한 레이블 값 경계로 확인하기

```
# 결정 트리 하이퍼 파라미터 = default
dt_clf = DecisionTreeClassifier(random_state=156).fit(X_features, y_labels)
visualize_boundary(dt_clf, X_features, y_labels)
```

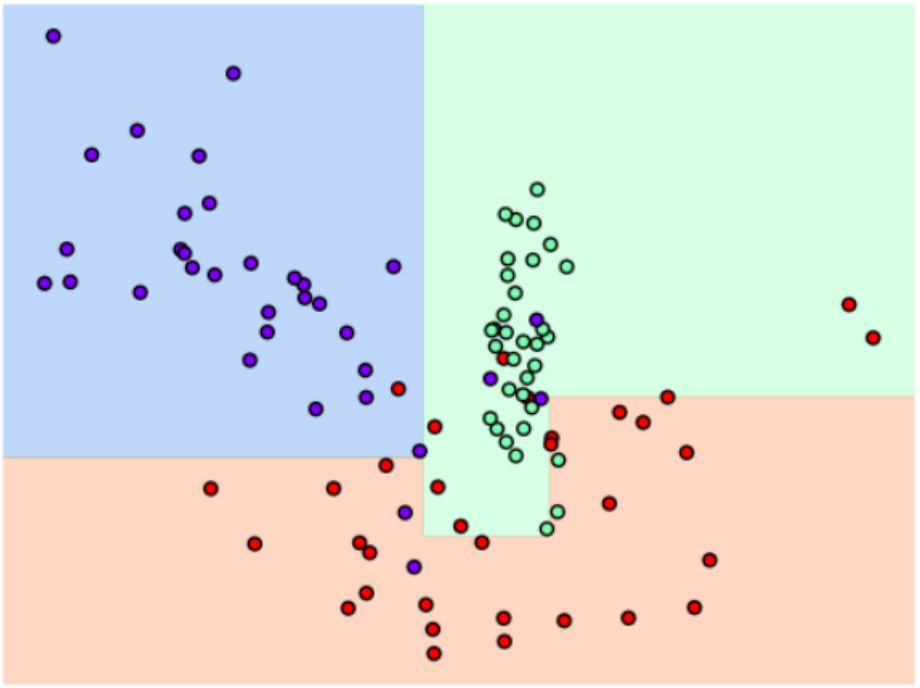


⇒ **과적합** 발생

: 학습 데이터에만 지나치게 최적화되어있어 테스트 데이터를 분류할 때 오히려 정확도가 떨어지게 됨

2. min_samples_leaf=6인 DecisionTreeClassifier

```
# min_samples_leaf=6으로 트리 생성 조건 제약
dt_clf = DecisionTreeClassifier(min_samples_leaf=6, random_state=156).fit(X_features, y_labels)
visualize_boundary(dt_clf, X_features, y_labels)
```



더 일반적인 분류 규칙에 따라 분류된 모델

앙상블 학습 (Ensemble Learning)

: 여러 개의 Classifier를 생성 → 예측을 결합 → 정확한 최종 예측을 도출하는 기법
⇒ 단일 분류기보다 신뢰성이 높은 예측값을 얻고자 함

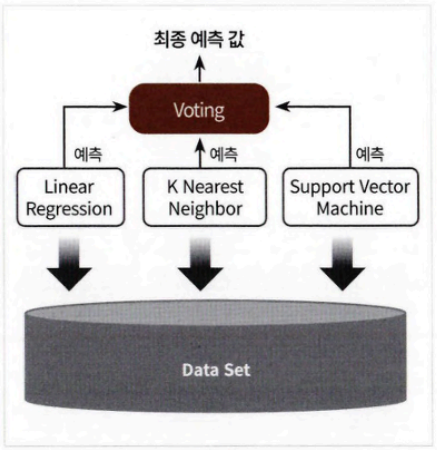
= 편향-분산 트레이드오프의 효과를 극대화

- *편향: 학습 알고리즘에서 잘못된 가정을 했을 때 발생하는 오차 → underfitting*
- *분산: 데이터 세트에 작은 변동 때문에 발생하는 오차 → overfitting*

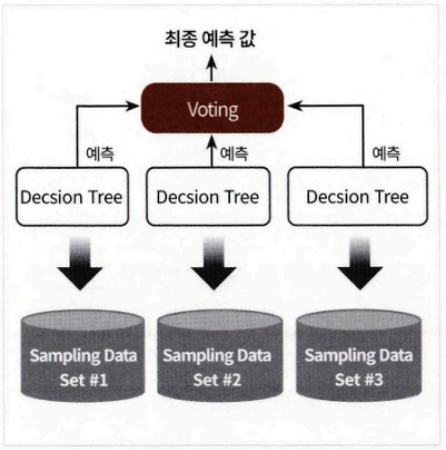
ex) 랜덤 포레스트, 그래디언트 부스팅 알고리즘
— XGBoost, LightGBM, Stacking 등

앙상블 학습의 유형

- **보팅(Voting)**
  : 서로 다른 알고리즘 + 같은 데이터 세트 학습&예측 →
  보팅
- **배깅(Bagging)**
  : 단일 알고리즘(결정 트리) + 서로 다른(부트스트래핑 샘플링) 데이터 세트 각자 학습&예측 →
  보팅
  - 부트스트래핑(Bootstrapping) 분할 방식
    : 개별 Classifier에게 원본 학습 데이터를 샘플링해서 추출하는 방식
    — 샘플간 중복 데이터 허용



Voting 방식



Bagging 방식

- **부스팅(Boosting)**

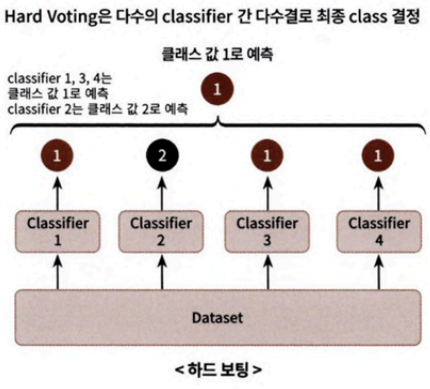
  : 여러 분류기가 순차적으로, 다음 분류기에게 가중치를 부여하면서 학습&예측

  - ex. 그래디언트 부스트, XGBoost, LightGBM

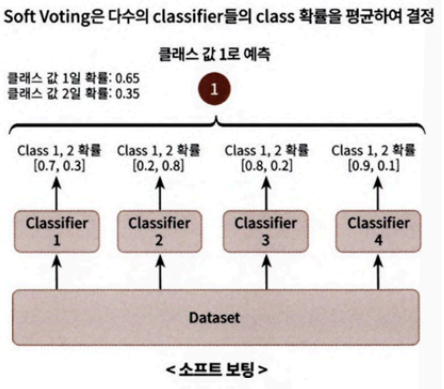
- 스태킹
  : 여러 가지 다른 모델의 예측 결괏값을 다시 학습 데이터로 만듦
  → 메타 모델로 재학습시켜 결과를 예측

보팅 유형 - 하드 보팅 & 소프트 보팅

	하드 보팅	소프트 보팅
결괏값 선정 방법	다수결 원칙 다수의 분류기가 결정한 예측값으로 선정	분류기들의 각 레이블 값 결정 확률의 평균이 높은 레이블 값을 선정 → 더 자주 사용



<하드 보팅>



<소프트 보팅>

보팅 분류기 (Voting Classifier)

** 사이킷런 VotingClassifier 클래스: 보팅 방식의 앙상블 구현

✅ VotingClassifier 생성 인자

- estimators=[('해당 분류기에 붙일 이름', Classifier1), ('분류기 이름', Classifier2), ...]
  : 보팅에 사용될 Classifier 객체 튜플의 리스트

- voting='hard' OR ='soft'
  : 보팅 방식 지정 (default='hard')

✅ **보팅 방식의 앙상블로 위스콘신 유방암 데이터 세트 예측 분석하기**

: 로지스티 회귀, KNN Classifier를 사용한 소프트 보팅 방식

```
# 로지스틱 회귀, KNN -> 소프트 보팅 방식으로 분류기 생성

# 개별 모델: 로지스틱 회귀+KNN
lr_clf = LogisticRegression(solver='liblinear')
knn_clf = KNeighborsClassifier(n_neighbors=8)

# 개별 모델 -> 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 모델 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
```

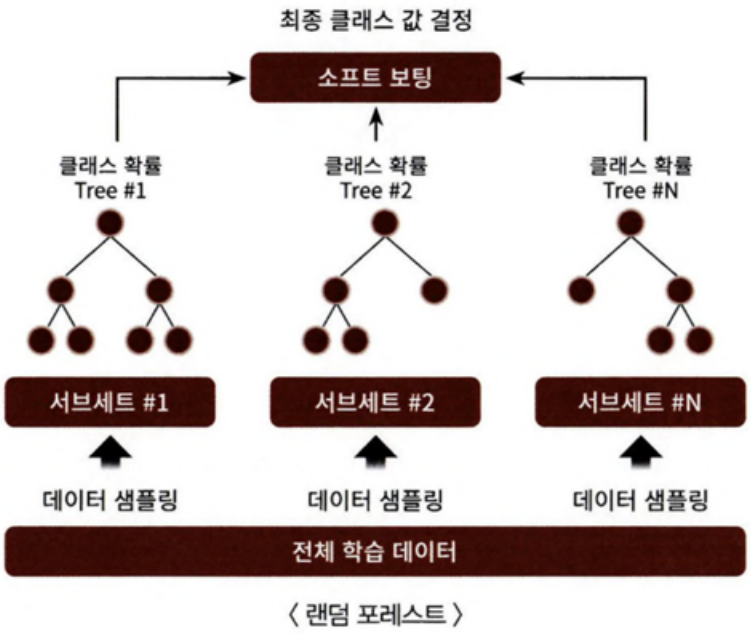
랜덤 포레스트

배깅 중 하나

— 같은 알고리즘 Classifier 다른 data 보팅

✏️ **랜덤 포레스트**

: 데이터가 중첩된(부트스트래핑 샘플링) 개별 데이터 세트에 결정 트리 분류기 각각 적용



〈 랜덤 포레스트 〉

** 사이킷런 RandomForestClassifier 클래스

하이퍼 파라미터

파라미터	설명
n_estimators	- 랜덤 포레스트에서의 결정 트리 개수 - default=10
max_features	- 최적 분할을 위해 고려할 최대 피처 개수 - default='auto'='sqrt' → 트리 분할 피처 참조할 때 sqrt(전체피처개수)만큼 참조
+) max_depth, min_samples_leaf, min_samples_split 등 결정 트리의 파라미터 적용 가능	

⚠️ 트리 기반 앙상블 알고리즘에는 하이퍼 파라미터가 너무 많아서 튜닝 시간이 많이 소모됨. 시간 투자에 비해 예측 성능이 크게 향상되지도 않음. 랜덤 포레스트의 하이퍼 파라미터는 결정 트리에서 사용되는 파라미터가 대부분이기 때문에, 그나마 적은 편에 속한다.

✅ **랜덤 포레스트 예측 모델 생성 + 하이퍼 파라미터 튜닝하기**

1. 별다른 튜닝 없이 랜덤 포레스트 학습 및 예측 성능 평가

```python
# 사용자 행동 인식 데이터 세트를 RandomForestClassifier로 예측하기
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import pandas as pd
import warnings
warnings.filterwarnings('ignore')

# 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, max_depth=8)
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))
```

```
랜덤 포레스트 정확도: 0.9220
```

2. GridSearchCV로 하이퍼 파라미터 튜닝 후 학습 및 예측 성능 평가

```python
# RandomForestClassifier 하이퍼 파라미터 튜닝하기
from sklearn.model_selection import GridSearchCV

params = {
    'max_depth': [8,16,24],
    'min_samples_leaf': [1,6,12],
    'min_samples_split': [2,8,16]
}

# RandomForestClassifier 객체 생성 후 GridSearchCV 수행
rf_clf = RandomForestClassifier(n_estimators=100, random_state=0, n_jobs=-1) # n_jobs=
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': 16, 'min_samples_leaf': 6, 'min_samples_split': 16}
최고 예측 정확도: 0.9157
```

```python
# 최적 하이퍼 파라미터로 rf_clf 학습하기
rf_clf1 = RandomForestClassifier(n_estimators=100, min_samples_leaf=6, max_depth=16,
                                 min_samples_split=2, random_state=0)
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: 0.9253
```