



# Aichemist Session

CHAP 04 분류(1)

# CONTENTS

## 분류

01. 결정 트리

02. 앙상블 학습

03. 랜덤 포레스트

04. GBM

→ 3주차는 여기까지만!

05. XGBoost

06. LightGBM

07. 베이지안 최적화 기반의 HyperOpt를 이용한 하이퍼 파라미터 튜닝

08. 스택킹 앙상블

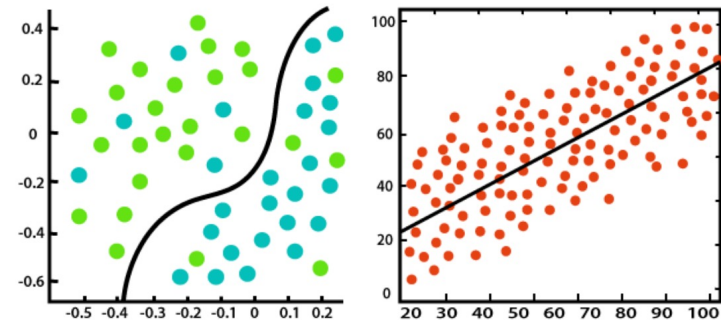
# 분류

: 학습데이터로 주어진 데이터의 피처와 레이블 값(결정 값, 클래스 값)을

머신러닝 알고리즘으로 학습해 모델을 생성하고, 그 모델로 새로운 데이터의 레이블 예측

지도학습!

<div>분류</div> <div>(Classification)</div>		<div>회귀</div> <div>(Regression)</div>
예측하고자 하는 값이 범주형 변수인 경우 연속되지 않은 이산적 카테고리 구분 문제		예측하고자 하는 값이 실수인 경우 연속된 값 예측 문제
이진 분류	다중 분류	ex) 체중 데이터를 바탕으로 신장 예측
0,1 2개의 결정 클래스 값	3개 이상의 결정 클래스 값	
ex) 이미지 데이터를 바탕으로 강아지/고양이 사진 분류	ex) 식물 종 분류	





# 01.

## 결정 트리

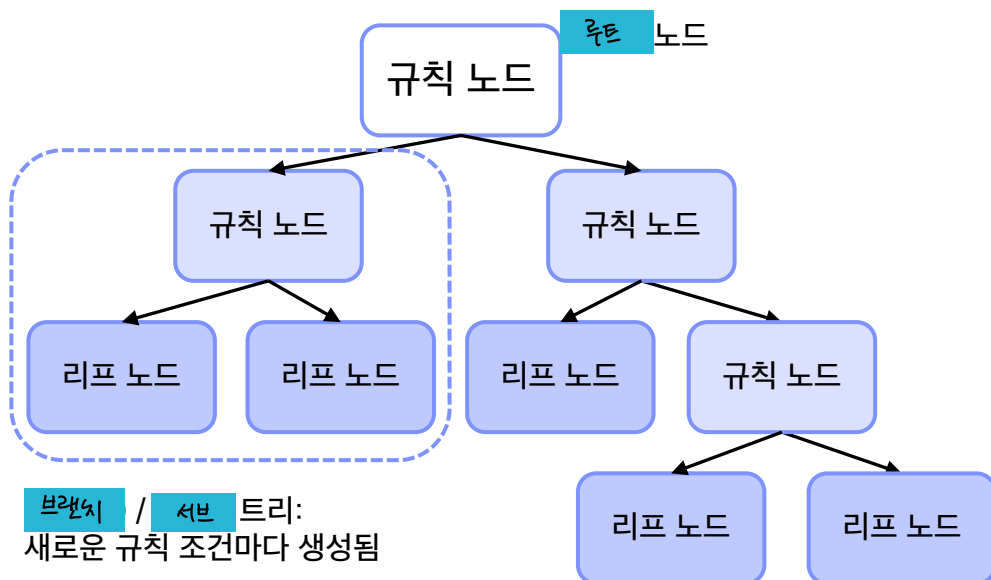
# 결정 트리

: 데이터에 있는 규칙을 학습을 통해 찾아내 트리 기반 분류 규칙을 만드는 것

데이터를 어떻게 **분류** 할지 결정하는 기준이 모델의 성능을 결정짓는 핵심 요소

깊이가 깊어지면 과적합 $\uparrow$ , 예측 성능 $\downarrow$   $\rightarrow$  최대한 **균일한** 데이터 세트 구성하도록 분할 필요

## • 결정 트리 구조



- **루트** 노드: 깊이가 0인 맨 위의 노드
- **규칙** 노드: 규칙조건을 만드는 노드  
= 결정 노드
- **리프** 노드: 최종 클래스 값이 결정되는 노드  
= 말단 노드

## 결정 트리 특징

장점	단점
<ul style="list-style-type: none"><li>- ML 알고리즘 중 직관적이며 이해하기 쉬움</li><li>- 데이터 사전 가공의 영향이 매우 적음</li><li>- 시각화가 가능하여 분류 과정을 이해하기 쉬움 (Graphviz 패키지)</li><li>- 범주형/수치형 데이터 모두 처리 가능</li></ul>	<ul style="list-style-type: none"><li>- 예측 성능 향상을 위한 규칙 추가 → 복잡한 학습 모델, 유연성↓ → 과적합에 취약</li><li>- 선형 관계 모델링에 제한적</li></ul>

# 정보 균일도

- 균일도 : 데이터 세트 내의 데이터가 얼마나 (일관된) 값을 가지는지를 나타내는 척도

데이터 세트의 균일도 ↑, 데이터 구분에 필요한 정보의 양↓

결정 노드는 정보 균일도가 **높은** 데이터 세트를 먼저 선택하도록 조건을 만듦

- 정보 균일도 측정 방법

정보 이득	지니 계수
엔트로피 ↓, 균일도↑ 1 - 엔트로피 지수 정보 이득 <b>높은</b> 속성 기준 분할	(평등) 0 ~ 1 (불평등) 지니 계수↓, 균일도↑ 지니 계수 <b>낮은</b> 속성 기준 분할

## 생성 과정 및 규칙

- Root node → Leaf nodes

정보 균일도가 **높은** 데이터 세트로 분할되는 조건을 찾아 서브 데이터 세트를 만들고 다시 서브 데이터 세트에서 위 방식을 반복하는 방식으로 데이터 값을 예측

1. 노드의 모든 데이터가 이미 하나의 Class에 속함 or 더 이상 고려할 feature가 없음

→ **리프** 노드로 만들고 분할 종료

(고려할 feature가 더 이상 없어서 리프 노드가 된 경우, 가장 많은 수의 Class를 결과 값으로 채택)

2. 각 노드에서 고려할 feature 선택 시, 데이터들을 가장 잘 나눠주는 feature를 선택

(이때 Leaf node에서 완벽하게 데이터들이 나눠진다는 보장X)

3. 선택된 feature에 대한 조건별 자식 노드 생성

4. 각 자식 노드에서 1부터 반복



## 결정 트리 파라미터

- CART의 DecisionTreeClassifier

DecisionTreeClassifier(criterion, splitter, max\_depth, min\_samples\_split, min\_samples\_leaf, min\_weight\_fraction\_leaf, max\_features, random\_state, max\_leaf\_nodes, min\_impurity\_decrease, class\_weight)

\* min으로 시작하는 매개변수를 증가시키거나 max로 시작하는 매개변수를 감소시키면 모델에 규제가 커짐

## 결정 트리 파라미터

<code>min_samples_split</code>	노드를 분할하기 위한 최소한의 샘플 데이터 수 과적합 제어 default: 2, 작게 설정할 수록 분할되는 노드 ↑ → 과적합 가능성 ↑
<code>min_samples_leaf</code>	리프 노드가 되기 위한 최소한의 샘플 데이터 수 과적합 제어 비대칭적 데이터의 경우 특정 클래스의 데이터가 작을 가능성 0 → 작게 설정
<code>max_features</code>	최적의 분할을 위해 고려할 최대 피처 개수 (default: none, 모든 피처를 사용해 분할) Int 형 : 대상 피처의 개수 / float 형 : 전체 피처 중 대상 피처의 퍼센트 sqrt : 전체 피처중 sqrt(전체 피처 개수), 'auto'로 지정하면 sqrt와 동일 log 는 전체 피처중 log2(전체 피처 개수) 선정
<code>max_depth</code>	트리의 최대 깊이를 규정 default: none, 완벽히 클래스 결정 값이 될 때까지 계속 분할 / 노드가 가지는 데이터 수가 min_samples_split보다 작아질 때까지 계속 깊이를 증가시킴 *과적합 주의, 적절한 값으로 제어 필요
<code>max_leaf_nodes</code>	리프 노드 최대 개수

## 결정 트리 파라미터

criterion	노드 분할 기준 설정. gini (default) : 지니불순도 / entropy : 정보 이득
splitter	각 노드에서 분할을 선택하는데 사용되는 전략 best (default): 최선의 분할 / random : 무작위 분할 선택
random_state	난수 생성기의 시드, 모델 재현성을 위해 사용
min_weight_fraction_leaf	리프 노드가 되기 위해 필요한 (모든 입력 샘플에 대한) 최소 가중치의 합계 비율을 설정, 과적합 제어 default: 0.0, 가중치 비율에 제한을 두지 않음
min_impurity_decrease	분할로 인해 불순도가 이 값보다 크게 감소하는 경우에만 분할
class_weight	클래스 가중치 특정 클래스의 샘플에 더 많은 가중치 줄 때 사용 balanced : 클래스 빈도의 역수에 비례하여 가중치가 자동으로 조정

# 붓꽃 데이터 세트 분류 예제

```
# 모듈 임포트
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import export_graphviz
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings('ignore')

# DecisionTreeClassifier 생성
dt_clf = DecisionTreeClassifier(random_state=156)

# 붓꽃 데이터 로딩, 학습 / 테스트 데이터 세트 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=156)

# DecisionTreeClassifier 학습
dt_clf.fit(X_train, y_train)

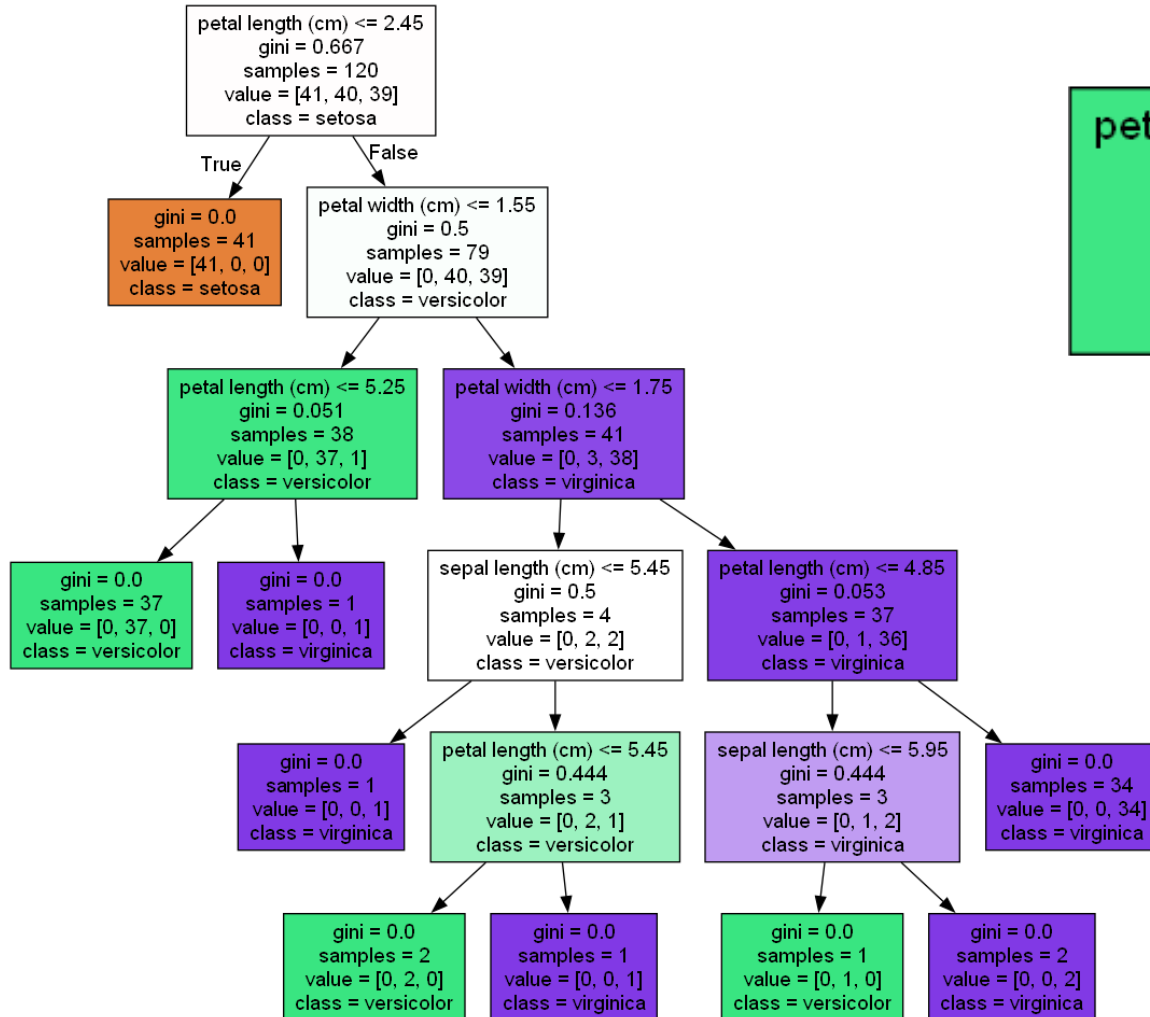
# 학습된 Classifier로 tree.dot 생성
export_graphviz(dt_clf, out_file="tree.dot", class_names=iris_data.target_names,
                feature_names=iris_data.feature_names, impurity=True, filled=True)
```

- 모듈 임포트
- 결정 트리 Estimator 생성
- 데이터 로딩 및 분리
- 학습 및 tree.dot 생성

**impurity:** 시각화에 노드의 불순도 정보가 나타  
남

**filled:** 노드의 클래스가 구분되도록  
색을 칠함

# 붓꽃 데이터 세트 분류 예제



- 규칙
- 현 데이터 분포에서의 지니 계수
- 현 규칙에 해당하는 데이터 건수
- 클래스 값 기반 데이터 건수
- 클래스 값 (최다)

노드 색: 레이블 값 의미,

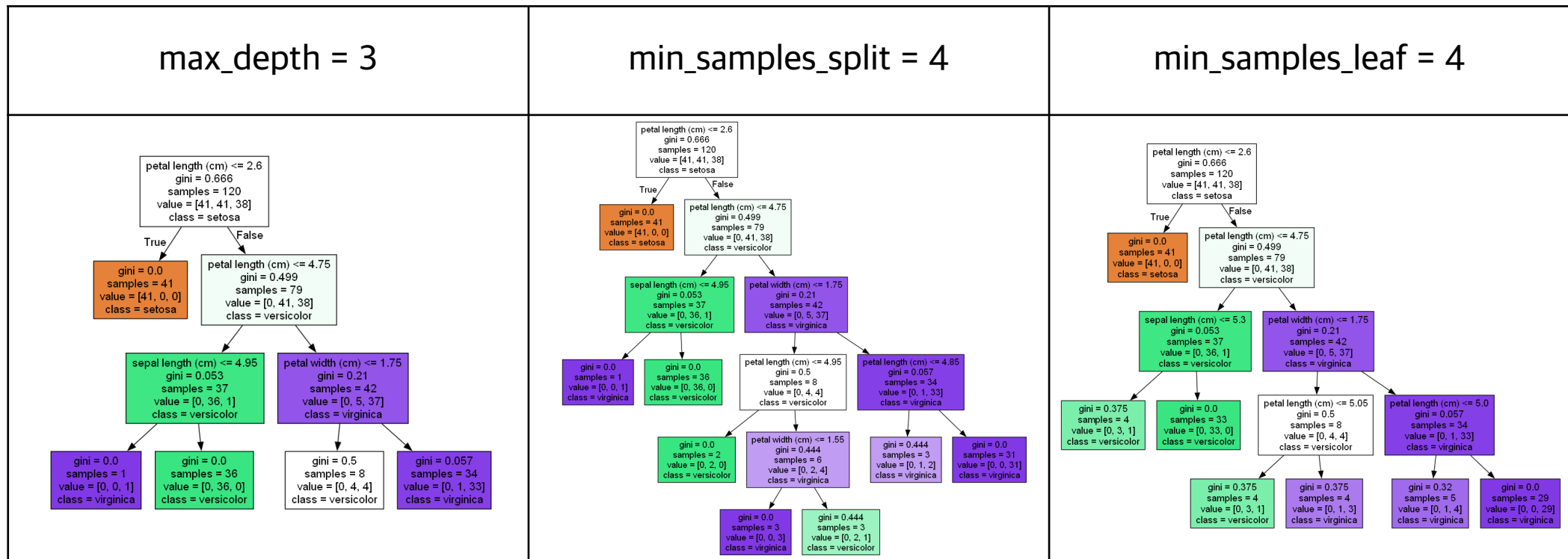
색이 진할수록 지니계수가 낮음

# 붓꽃 데이터 세트 분류 예제

- 하이퍼 파라미터 조절에 따른 결정 트리 변화

복잡한 규칙 트리 → 모델 과적합

그러므로, 하이퍼 파라미터 제어를 통해 과적합 방지, 간결하고 이상치에 강한 모델 만들 수 있음



# 붓꽃 데이터 세트 분류 예제

- 피처별 중요도 확인

```
# 피처별 중요도 확인

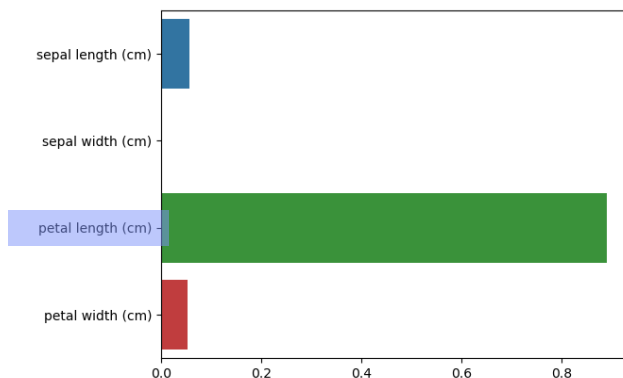
import seaborn as sns
import numpy as np
%matplotlib inline

# 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 column별로 시각화
sns.barplot(x=dt_clf.feature_importances_, y=iris_data.feature_names)
```

```
Feature importances:
[0.058 0.      0.89  0.053]
sepal length (cm) : 0.058
sepal width (cm) : 0.000
petal length (cm) : 0.890
petal width (cm) : 0.053
```



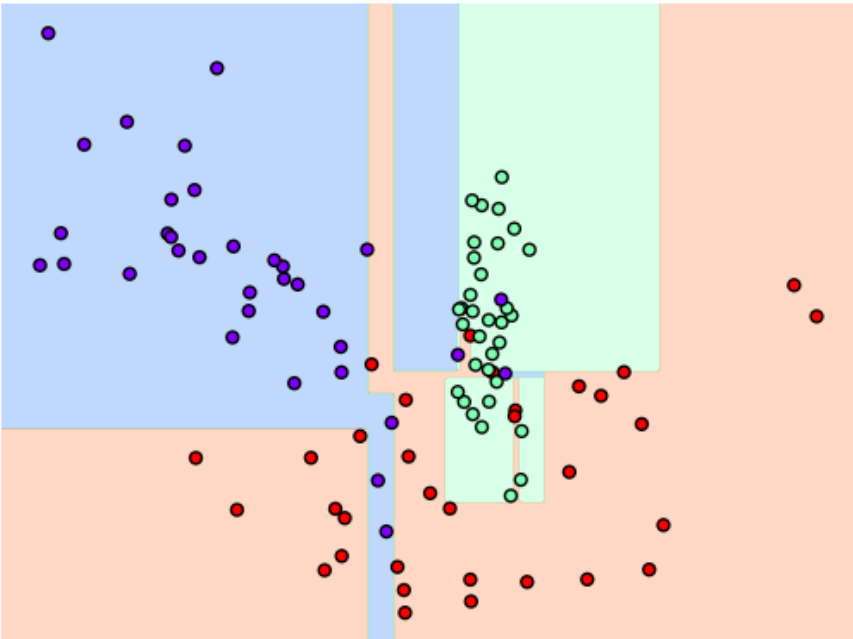
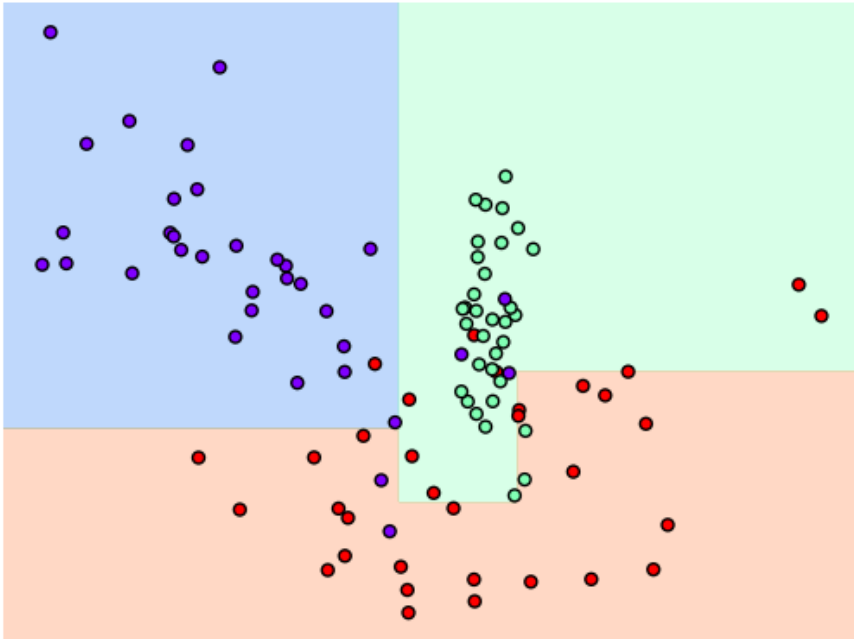
.feature\_importances\_ :

결정 트리 알고리즘이 학습을 통해 규칙을 정하는 데 있어 피처의 중요한 역할 지표 제공

피처가 트리 분할 시 정보 이득 또는 지니 계수를 얼마나 효율적으로 개선시켰는지를 정규화된 값으로 표현

값↑, 피처의 중요도↑

## 결정 트리 과적합(Overfitting)

생성 제약 없음	min_samples_leaf = 6
	
일부 이상치 데이터도 분류하기 위한 분할로 결정 기준 경계가 매우 많아짐 → 유연성↓ → 예측 정확도↓	이상치에 크게 반응하지 않으면서 좀 더 일반화된 분류 규칙에 따라 분류됨 → 예측 정확도↑



## 실습 - 사용자 행동 인식 데이터 세트

- (1) 데이터 전처리
- (2) 학습/테스트용 DataFrame 로딩
- (3) DecisionTreeClassifier 로 학습/예측/평가
- (4) 하이퍼 파라미터 튜닝
- (5) feature\_importances\_ 로 각 피쳐 중요도 확인

# 실습 - 사용자 행동 인식 데이터 세트

## (1) 데이터 전처리

### - 피처명 데이터 로드, 살펴보기

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

# features.txt 파일에는 피처 이름 index, 피처명이 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'])

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

```
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
```

전체 피처명에서 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', 'tBodyAcc-mad()-Z', 'tBodyAcc-max()-X']

# 실습 - 사용자 행동 인식 데이터 세트

## (1) 데이터 전처리

### - 중복된 피처명 확인 및 변환 함수 정의

```
# 중복된 피처명 확인

# 피처명 기준으로 그룹화하고, 그룹별 카운트(같은 이름 세기)하여 feature_dup_df에 저장
feature_dup_df = feature_name_df.groupby('column_name').count()
# feature_dup_df에서 같은 이름이 2개 이상인 경우를 필터링하여 수를 출력
print(feature_dup_df[feature_dup_df['column_index'] > 1].count())
# 중복된 피처명 살펴보기
feature_dup_df[feature_dup_df['column_index'] > 1].head()
```

```
column_index    42
dtype: int64
```

column_name	column_index
fBodyAcc-bandsEnergy()-1,16	3
fBodyAcc-bandsEnergy()-1,24	3
fBodyAcc-bandsEnergy()-1,8	3
fBodyAcc-bandsEnergy()-17,24	3
fBodyAcc-bandsEnergy()-17,32	3

```
# 중복된 피처명 변환하여 DataFrame 반환하는 함수 정의 (_1, _2...)
```

```
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
```

# 실습 - 사용자 행동 인식 데이터 세트

## (2) 학습/테스트용 DataFrame 로딩

### - 학습, 테스트 데이터를 DataFrame에 로딩하여 반환하는 함수 정의

*# 학습/테스트용 데이터 DataFrame에 로딩하는 함수 정의*

```
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 피처명을 수정하는 get_new_feature_name_df()를 이용, 신규 피처명 DataFrame 생성.
    new_feature_name_df = get_new_feature_name_df(feature_name_df)

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

    # 학습 피처 데이터 셋과 테스트 피처 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()`

# 실습 - 사용자 행동 인식 데이터 세트

## (2) 학습/테스트용 DataFrame 로딩

### - 학습용 피쳐, 레이블 데이터 세트 살펴보기

```
# 학습용 피쳐 데이터 세트 살펴보기
print('## 학습 피쳐 데이터셋 info()')
print(X_train.info())
```

```
## 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
```

전부 float64 → 인코딩 필요 없음

```
# 학습용 레이블 데이터 세트 살펴보기
print(y_train['action'].value_counts())
```

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

레이블 값이 고르게 분포되어 있음

# 실습 - 사용자 행동 인식 데이터 세트

## (3) DecisionTreeClassifier로 학습/예측/평가

### - 디폴트 하이퍼 파라미터로 수행

```
# 디폴트 설정으로 DecisionTreeClassifier 학/예/평
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# 예제 반복 시 마다 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, 'max_leaf_nodes': None, 'min_
impurity_decrease': 0.0, 'min_samples_leaf': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'random_state': 156, 'splitt
er': 'best'}
```

# 실습 - 사용자 행동 인식 데이터 세트

## (4) 하이퍼 파라미터 튜닝

- max\_depth 조절 (GridSearchCV 검증용 데이터 세트의 정확도 평균 수치)

```
from sklearn.model_selection import GridSearchCV

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

grid_cv = GridSearchCV(dtclf, 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, 'min\_samples\_split': 16}

```
# GridSearchCV 객체의 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']]
```

↓ 5개 CV 세트에서 검증용 데이터 세트의 정확도  
평균 수치

8일 때 정점, 이후로 떨어짐

	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

# 실습 - 사용자 행동 인식 데이터 세트

## (4) 하이퍼 파라미터 튜닝

- max\_depth 조절 (테스트 데이터 세트에서 정확도)

```
# 테스트 데이터 세트에서 max_depth 에 따른 정확도 측정
max_depths = [ 6, 8 ,10, 12, 16 ,20, 24]
# max_depth 값을 변화 시키면서 그때마다 학습하고 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

8일 때 약 87.16%로 가장 높음



# 실습 - 사용자 행동 인식 데이터 세트

## (4) 하이퍼 파라미터 튜닝

### - max\_depth, min\_samples\_split 조절

```
# max_depth, min_samples_split 조절
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}

8, 16일 때 가장 높음

### - 최적 하이퍼 파라미터로 학습된 모델로 예측

```
# 최적 하이퍼 파라미터로 학습된 estimator 객체 best_estimator_로 테스트 데이터에 예측 수행
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

# 실습 - 사용자 행동 인식 데이터 세트

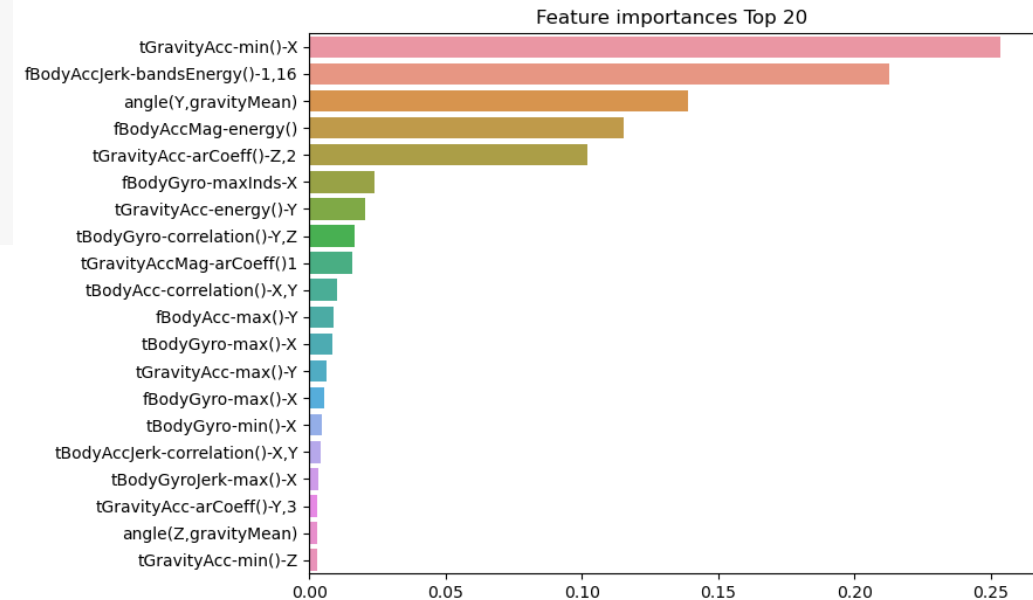
(5) feature\_importances\_ 로 각 피쳐 중요도 확인

- Top 20 피쳐 막대그래프

```
# 중요도 높은 Top 20 피쳐 막대그래프
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()
```

Top 5 피쳐가 큰 영향



---

---

---

# 02.

앙상블 학습

## 앙상블 학습

: 여러 개의 분류기를 생성하고 그 예측을 **결합** 해 보다 정확한 최종 예측을 도출하는 기법

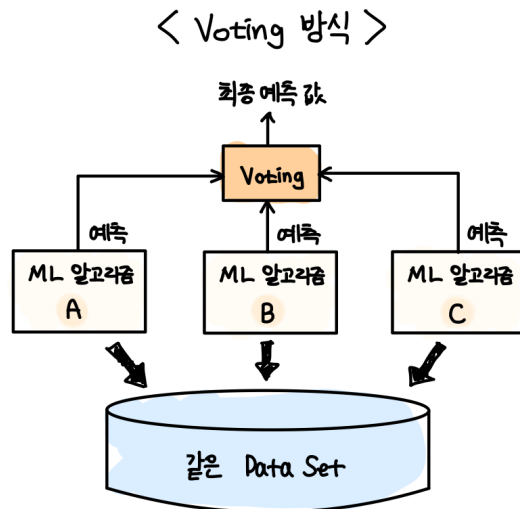
목표: 다양한 분류기의 예측 결과를 결합해 단일 분류기보다 신뢰성이 높은 예측값을 얻는 것  
정형 데이터 분류 시에 딥러닝보다 뛰어난 성능 나타냄  
랜덤 포레스트, XGBoost, LightGBM, 스택킹 ...

학습 유형: 보팅(Voting), 배깅(Bagging), 부스팅(Boosting) 외에 스택킹 등

# 앙상블 학습 유형

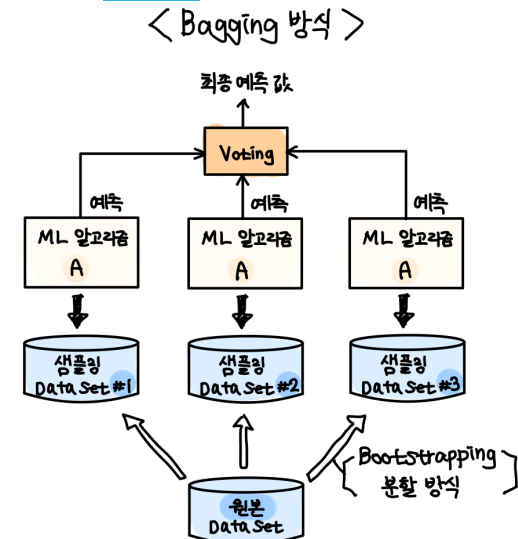
## 보팅

- 여러 개의 **다른** 알고리즘 가진 분류기가 투표를 통해 예측 결과 결정
- **같은** 데이터 세트 학습, 예측한 결과로 투표



## 배깅

- 여러 개의 **같은** 알고리즘 가진 분류기가 투표를 통해 예측 결과 결정
- **부트스트래핑** 방식으로 **개별** 데이터 세트 학습 : 원본 학습 데이터를 샘플링해 개별 분류기에 할당
- 데이터 세트 간 **중첩** 허용



## 앙상블 학습 유형

부스팅	스태킹
<ul style="list-style-type: none"><li>- 여러 개의 분류기가 순차적으로 학습 수행, 앞에서 예측 틀린 데이터에 대해 다음 분류기에게 가중치 부여하면서 학습, 예측 진행</li><li>- 그래디언트 부스트, XGBoost, LightGBM</li></ul>	<ul style="list-style-type: none"><li>- 여러 가지 다른 모델의 예측 결과값을 다시 학습 데이터로 만들어 다른 모델(메타 모델)로 재학습시켜 결과 예측</li></ul>

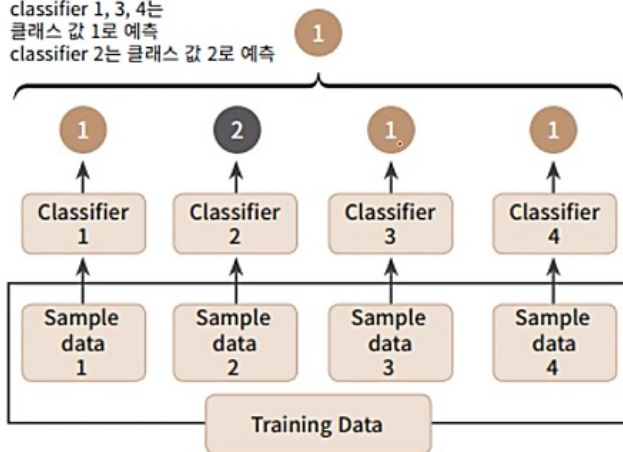
# 보팅 유형

## 하드 보팅 (Hard Voting)

- 다수의 분류기가 결정한 예측값을 최종 보팅값으로 설정

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>

## 소프트 보팅 (Soft Voting)

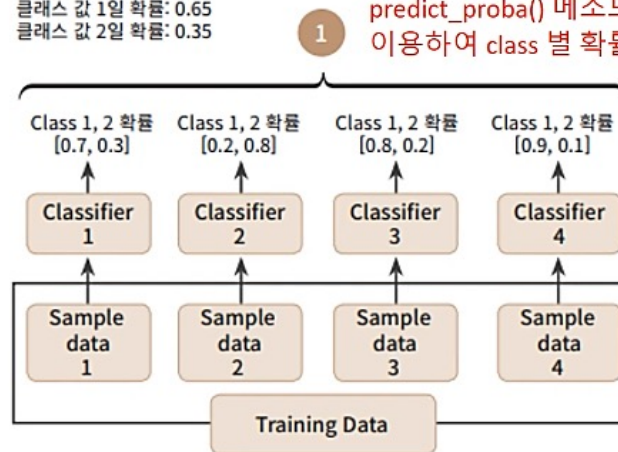
- 분류기들의 레이블 값 결정 확률을 평균해서 이들 중 확률이 가장 높은 레이블 값을 최종 보팅 결과값으로 선정

- 일반적으로 하드 보팅보다 예측 성능 좋음

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>

# 실습 - 위스콘신 유방암 데이터 세트

## (1) 모듈 임포트, 데이터 로딩 및 확인

```
# 모듈 임포트, 데이터 로딩
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)
# 데이터 살펴보기
data_df.head(3)
```

유방암의 악성종양, 양성종양 여부를 결정하는 이진 분류 데이터 세트

종양의 크기, 모양 등의 형태와 관련된 많은 피처가짐

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

3 rows × 30 columns



# 실습 - 위스콘신 유방암 데이터 세트

## (2) 소프트 보팅 방식 분류기 학습/예측/평가

```
# 소프트 보팅 분류기 생성, 학/예/평

# 개별 모델은 로지스틱 회귀와 KNN
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

전반적으로 단일 ML 알고리즘보다 예측 성능 뛰어남  
(항상 X)

많은 분류기 결합해 유연성 증가

---

---

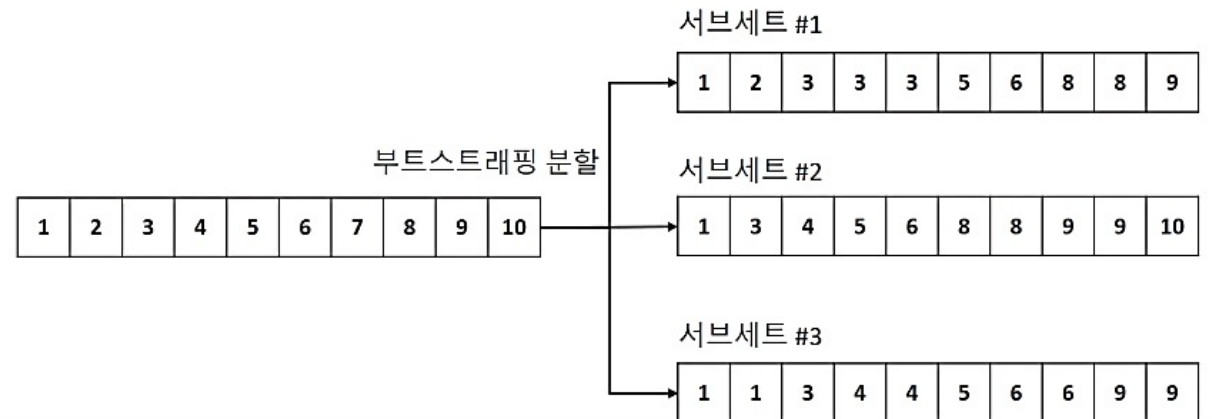
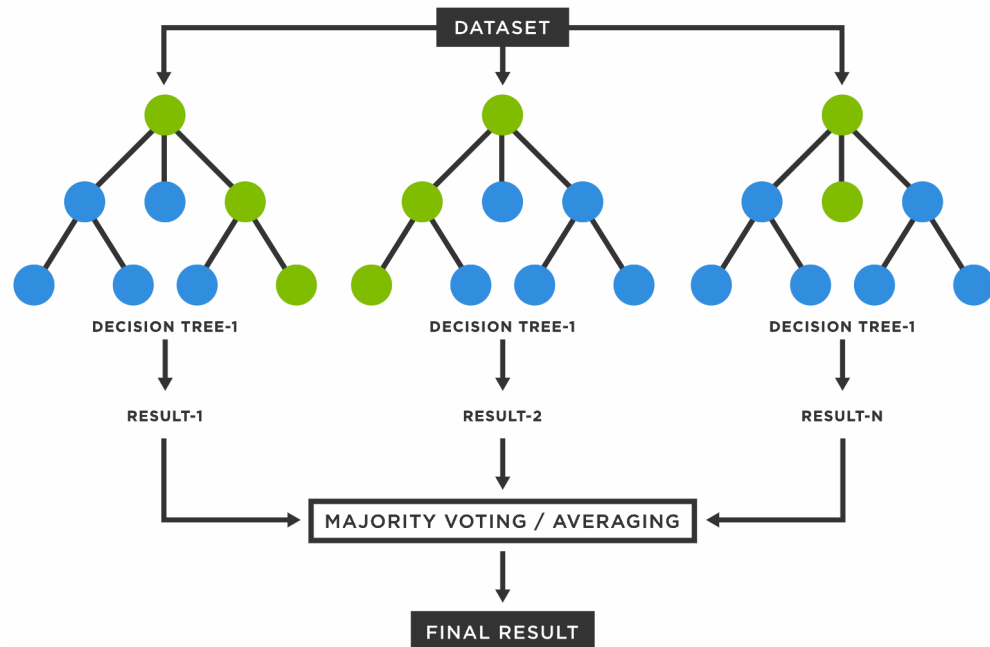
---

# 04.

## 랜덤 포레스트

# 랜덤 포레스트

: 여러 개의 **결정 트리** 분류기가 전체 데이터에서 배깅 방식으로 각자의 데이터를 샘플링해  
개별적으로 학습을 수행한 뒤 최종적으로 모든 분류기가 **보팅**을 통해 예측 결정



서브 데이터 세트 : 전체 데이터 건수와 동일, 개별 데이터가 중복

## 실습 - 사용자 행동 인식 데이터 세트

- (1) 모듈 임포트, 함수 정의
- (2) RandomForestClassifier 학습/예측/평가
- (3) 하이퍼 파라미터 튜닝
- (4) 피처 중요도 시각화

# 실습 - 사용자 행동 인식 데이터 세트

## (1) 모듈 임포트, 함수 정의

```
# 필요 모듈 임포트, 함수 정의
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
import pandas as pd
import warnings
warnings.filterwarnings('ignore')

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

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 피처명을 수정하는 get_new_feature_name_df()를 이용, 신규 피처명 DataFrame 생성.
    new_feature_name_df = get_new_feature_name_df(feature_name_df)

    # DataFrame에 피처명을 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()
```

결정 트리 실습에서 정의한 함수 재사용

# 실습 - 사용자 행동 인식 데이터 세트

## (2) RandomForestClassifier 학습/예측/평가

```
# 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 셋으로 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: {:.4f}'.format(accuracy))
```

랜덤 포레스트 정확도: 0.9253

# 실습 - 사용자 행동 인식 데이터 세트

## (3) 하이퍼 파라미터 튜닝

```
# 하이퍼 파라미터 튜닝
from sklearn.model_selection import GridSearchCV
```

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

```
# 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}  
최고 예측 정확도: 0.9180

```
# 테스트 데이터에서 예측, 평가
rf_clf1 = RandomForestClassifier(n_estimators=300, max_depth=10, min_samples_leaf=8, #
                                min_samples_split=8, 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: 0.9165

트리 기반 앙상블 알고리즘 단점: 효율성 떨어짐

많은 하이퍼 파라미터 ⇒ 시간 소모↑, 근데 튜닝 후 예측 성능 향상

경우 별로 없음

**n\_estimator** : 결정 트리의 개수 지정, default=10

증가 시, 좋은 성능 기대 가능 but 무조건 성능 향상X, 학습 수행 시간↑

**max\_features** : 최적의 분할 위해 고려할 최대 피쳐 개수

결정 트리의 max\_features 와 같음, 근데 기본이 auto(=sqrt)

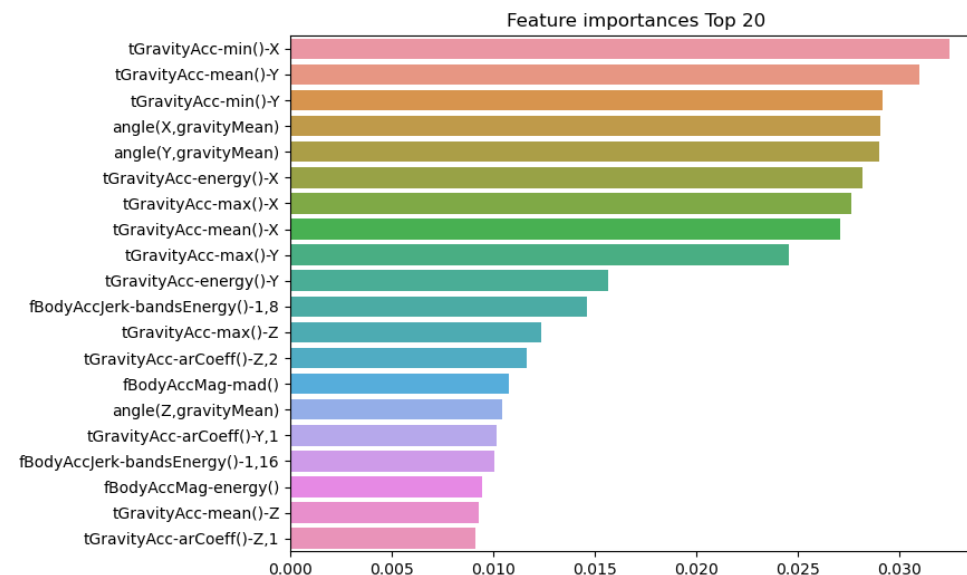
# 실습 - 사용자 행동 인식 데이터 세트

## (4) 피쳐 중요도 시각화

```
# 피쳐 중요도 시각화
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)
fig1 = plt.gcf()
plt.show()
plt.draw()
fig1.savefig('rf_feature_importances_top20.tif', format='tif', dpi=300, bbox_inches='tight')
```







04.

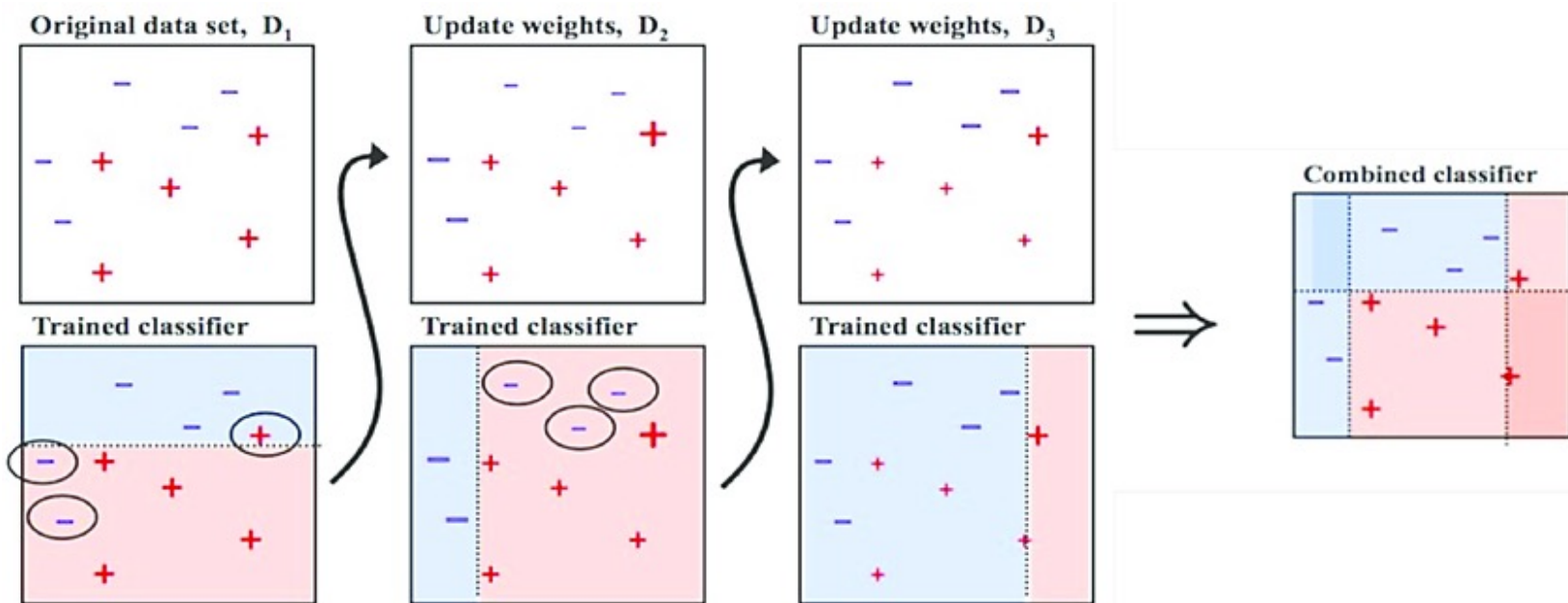
GBM

# AdaBoost

부스팅 알고리즘: 잘못 예측한 데이터에 **가중치** 부여해 오류 개선하면서 학습하는 방식

① AdaBoost(Adaptive boosting) : 오류 데이터에 가중치 부여하면서 부스팅 수행,

약한 학습기가 순차적으로 오류값에 대해 가중치를 부여한 예측 결정 기준을 모두 **결합** 해 예측을 수행



1. 약한 학습기가 분류 기준 1로  
데이터 분류

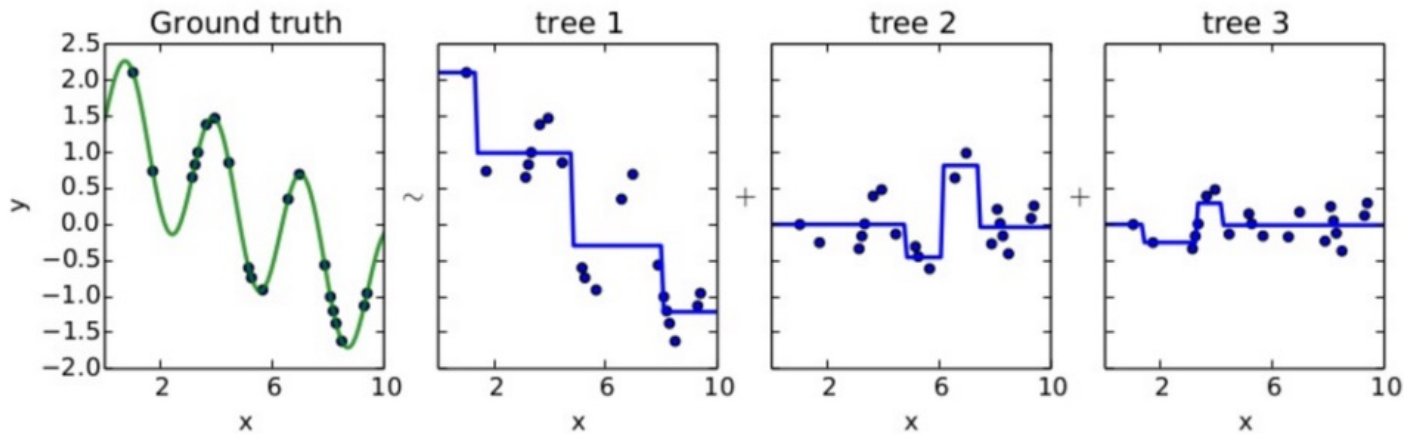
2. 오류 데이터에 대해 가중치 값 부여

3. 1-2 과정 반복

4. 약한 학습기가 순차적으로 오류값에 대해  
가중치를 부여한 예측 결정 기준을 모두 결합  
해 예측을 수행

# GBM

② GBM : 에이다 부스트와 유사, 가중치 업데이트를 **경사하강법** 이용



$h(x)$  : 오류값

$y$  : 실제값

$x_n$  : 데이터의 피쳐

$F(x)$  : 예측값

오류식:  $h(x) = y - F(x) \Rightarrow$  **오류식**의 최소화

경사하강법은 다음 챕터 '회귀'에서 자세히 설명합니다!

반복 수행을 통해 오류를 최소화할 수 있도록 가중치의 업데이트 값을 도출하는 기법

# GBM 하이퍼 파라미터

- GradientBoostClassifier 하이퍼 파라미터

n\_estimators, max\_depth, max\_features ... 트리 기반 파라미터

loss	경사 하강법에서 사용할 비용 함수 지정 default: deviance
learning_rate	GBM이 학습을 진행할 때마다 적용하는 학습률 weak learner가 순차적으로 오류값을 보정하는 데 적용 default: 0.1
n_estimators	weak learner의 개수 커질수록 예측 성능이 좋아짐 ( 일정 수준까지 ) 개수가 너무 많아지면 수행 시간이 오래 걸림
subsample	weak learner가 학습에 사용하는 데이터의 샘플링 비율 과적합이 염려되는 경우, 1보다 작은 값으로 설정

# 실습 - 사용자 행동 인식 데이터 세트

## (1) GradientBoostClassifier 학습/예측/평가

```
# 모듈 임포트
from sklearn.ensemble import GradientBoostingClassifier
import time
import warnings
warnings.filterwarnings('ignore')

# 학습/테스트 데이터 세트 로드
X_train, X_test, y_train, y_test = get_human_dataset()

# GBM 수행 시간 측정. 시작 시간 설정
start_time = time.time()

# 모델 학/예/평(정확도)
gb_clf = GradientBoostingClassifier(random_state=0)
gb_clf.fit(X_train, y_train)
gb_pred = gb_clf.predict(X_test)
gb_accuracy = accuracy_score(y_test, gb_pred)

print('GBM 정확도: {0:.4f}'.format(gb_accuracy))
print("GBM 수행 시간: {0:.1f} 초 ".format(time.time() - start_time))
```

GBM 정확도: 0.9389  
GBM 수행 시간: 4872.9 초

앞의 랜덤 포레스트 예제에 이어서 작성

수행 시간이 길

# GBM 정리

- 랜덤 포레스트보다,  
예측 성능 조금 뛰어남, **수행 시간** 오래 걸림(병렬 처리 지원X), 하이퍼 파라미터 튜닝 노력 더 필요
- 과적합에 강함
- GBM 기반 ML 패키지
  - **XGBoost**
  - **LightGBM**



수고하셨습니다