



앙상블

4조 이한울 임현진 정기윤 황정민

목차


앙상블이란

배깅

스택킹

보팅

부스팅

A modern home office with a wooden desk, a black desk lamp, a laptop, and framed art on a dark wall. The room has a large window on the left, a dark wall on the right, and a white cabinet in the background. The desk is light-colored wood, and the lamp is black with gold accents. The laptop is black with a white Apple logo. The wall is dark grey or black, and there are four framed abstract art pieces. The window shows a view of a residential area with trees and buildings.

1

앙상블이란?

앙상블이란

앙상블

여러 개의 분류기(Classifier)를 생성하고 그 예측을 결합함으로써 보다 **정확한 최종 예측**을 도출하는 기법



- ▷ 즉 여러 개의 단일 모델들의 평균치를 내거나, 투표를 해서 다수결에 의한 결정을 하는 등 여러 모델들의 집단 지성을 활용하여 더 나은 결과를 도출해낸다.
- ▷ 개별로 학습한 여러 개의 모델(weak learner)을 조합하여 일반화(generalization) 성능을 향상시킬 수 있다.

앙상블이란

앙상블 유형

보팅 Voting	배깅 Bagging	부스팅 Boosting
여러 종류의 알고리즘을 사용한 각각의 결과에 대해 투표를 통해 최종 결과를 예측하는 방식	같은 알고리즘에 대해 데이터 샘플을 다르게 두고 학습을 수행해 보팅을 수행하는 방식	여러 개의 알고리즘이 순차적으로 학습을 하되, 그 다음번 알고리즘에 가중치를 부여하여 학습과 예측을 진행하는 방식
하드 보팅 소프트 보팅	Random forest	AdaBoost GBM

A modern home office with a wooden desk, a black desk lamp, a laptop, and framed art on a dark wall. The room features a large window on the left, a dark wall with four framed abstract art pieces, and a white shelf with various decorative items. The overall aesthetic is minimalist and contemporary.

2

보팅(Voting)

보팅 (Voting)

뜻 그대로 “투표”를 통해 결과 도출하는 방식

서로 다른 알고리즘이 도출해낸

결과물에 대하여 최종 투표하는 방식으로 결과 예측

-hard voting

-soft voting

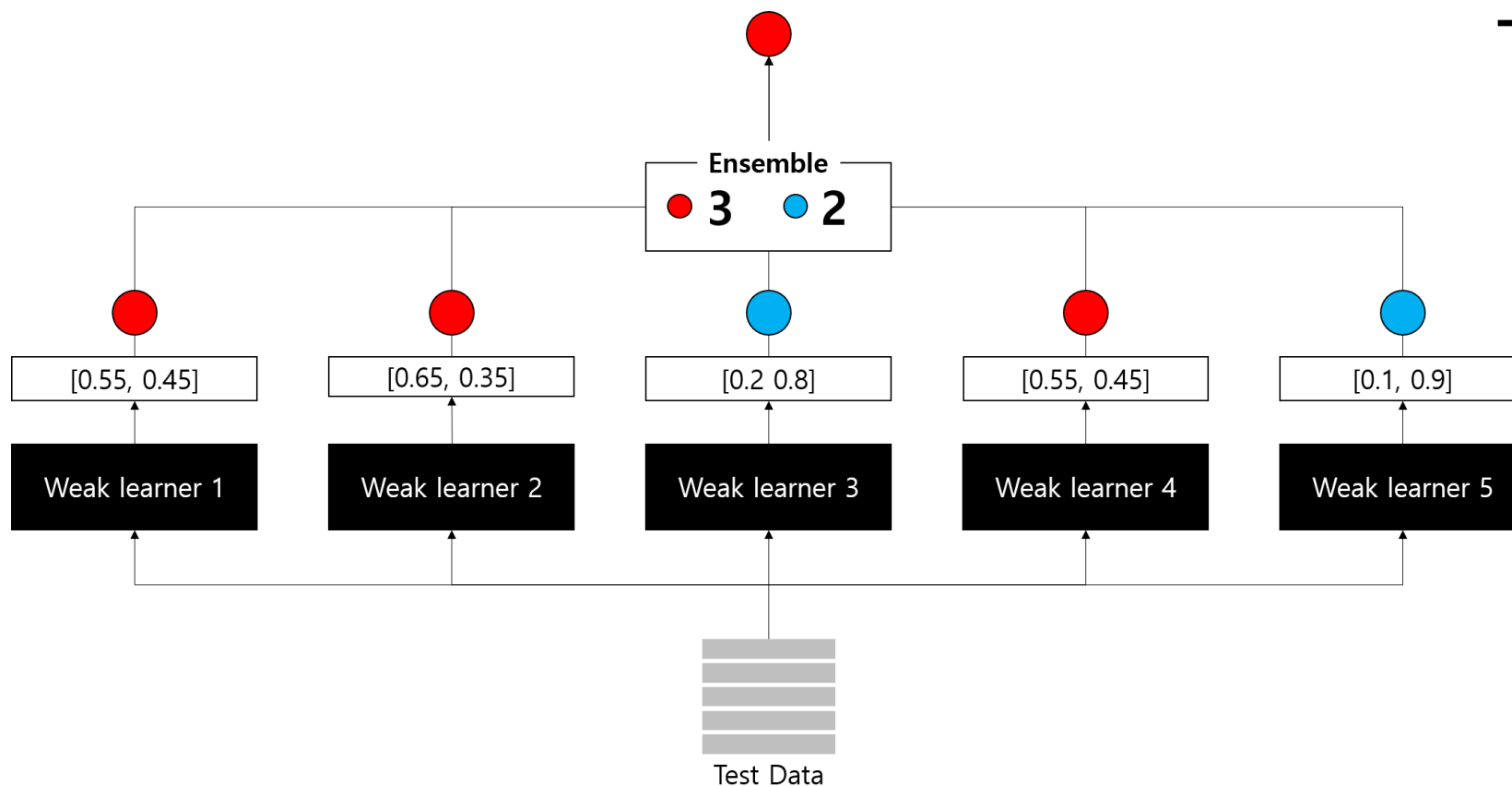


보팅 (Voting)

Hard voting

각 알고리즘들의 예측 결과값을 바탕으로 다수결 투표

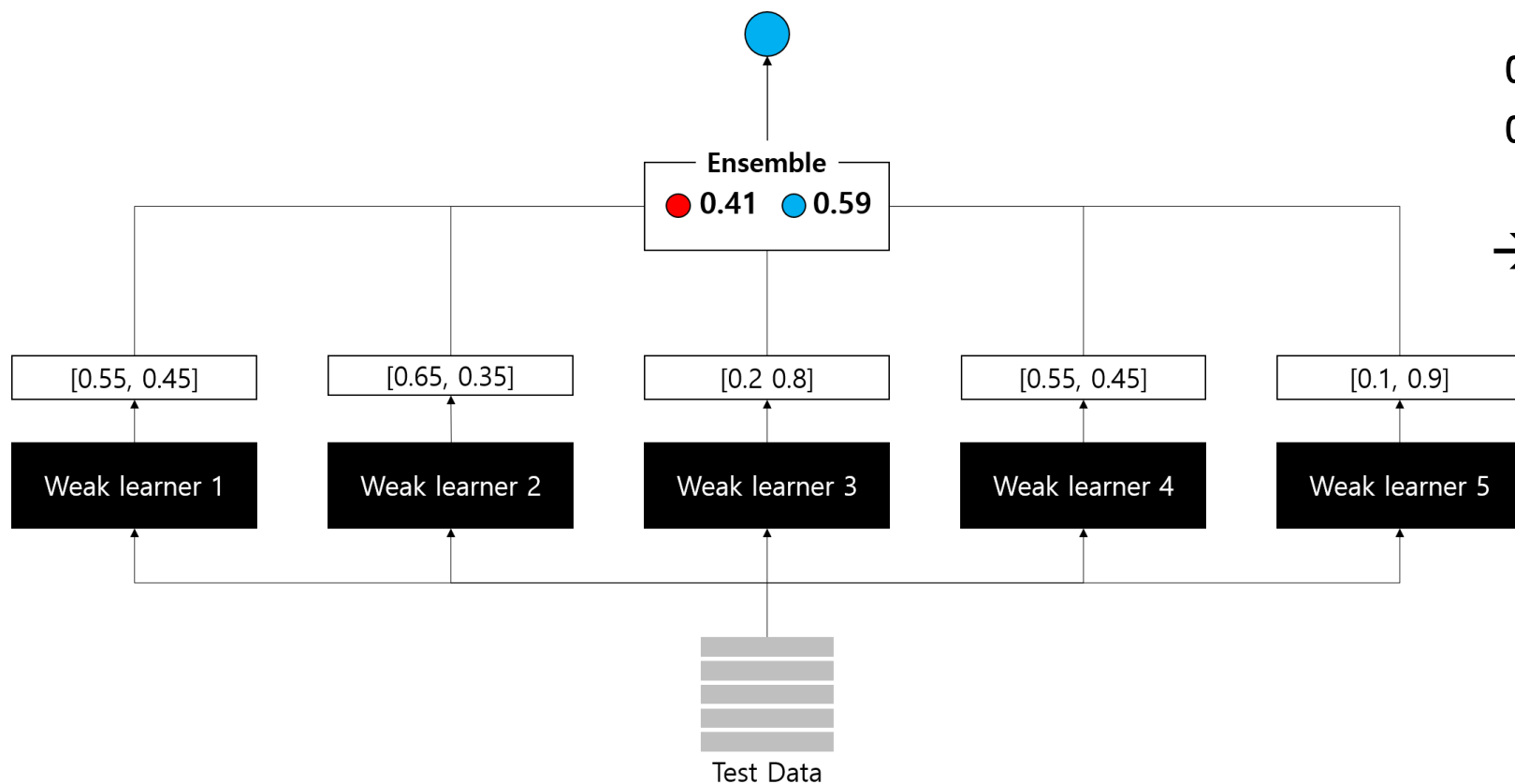
→ 다수결에 따라 최종 결과 **빨강**



보팅 (Voting)

Soft voting

각 알고리즘의 예측 확률값의 평균을 바탕으로 선택



예측확률값 평균(파랑): 0.59
예측확률값 평균(빨강): 0.41

→ 최종 결과 파랑

보팅 (Voting)

사이킷런에서 제공되는 위스콘신 유방암 데이터 세트를 이용해
보팅방식의 앙상블을 적용하기

```
import pandas as pd

from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
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
```

```
#유방암 데이터셋 불러오기
cancer = load_breast_cancer()
```

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

보팅 (Voting)

실습

```
#보팅 적용을 위한 개별모델 만들기(개별 모델은 각각 로지스틱회귀, KNN, random forest)
lr_clf = LogisticRegression(solver='lbfgs', class_weight='balanced', max_iter=10000)
knn_clf = KNeighborsClassifier(n_neighbors=8)
rf_cf = RandomForestClassifier(random_state=0)
```

```
#train, test 데이터 나누기
X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, test_size=0.2, random_state=156)
```

```
#개별 모델을 소프트 보팅 기반 앙상블 모델로 구현하기
vo_soft_clf = VotingClassifier(estimators=[('LR', lr_clf), ('KNN', knn_clf), ('RF', rf_cf)], voting='soft')
```

```
#소프트보팅 모델 학습
vo_soft_clf.fit(X_train, y_train)
pred_s = vo_soft_clf.predict(X_test)
```

```
#개별 모델을 하드 보팅 기반 앙상블 모델로 구현하기
vo_hard_clf = VotingClassifier(estimators=[('LR', lr_clf), ('KNN', knn_clf), ('RF', rf_cf)], voting='hard')
```

```
#하드보팅 모델 학습
vo_hard_clf.fit(X_train, y_train)
pred_h = vo_hard_clf.predict(X_test)
```

보팅 (Voting)

실습

```
: #모델 평가
print("Soft Voting 분류기 정확도", accuracy_score(y_test, pred_s))
print("Hard Voting 분류기 정확도", accuracy_score(y_test, pred_h))

classifiers = [lr_clf, knn_clf, rf_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)))
```

Soft Voting 분류기 정확도 0.9736842105263158
Hard Voting 분류기 정확도 0.9736842105263158
LogisticRegression 정확도: 0.9561
KNeighborsClassifier 정확도: 0.9386
RandomForestClassifier 정확도: 0.9561

보팅 분류기 정확도 > 개별 모델 정확도

하지만 항상 앙상블 기법이 무조건 더 좋은 성능이 나오지는 않는다.

또한 hard voting보다 soft voting이 일반적으로 성능 결과가 잘 나오는 편이라 더 합리적인 방법이다.

A modern office interior with a wooden desk, a black desk lamp, a laptop, and framed art on a dark wall. The scene is dimly lit, with light coming from a window on the left. A large white number '3' is overlaid on the image.

3

배깅(Bagging)

배깅 (Bagging)

Bootstrap + Aggregating

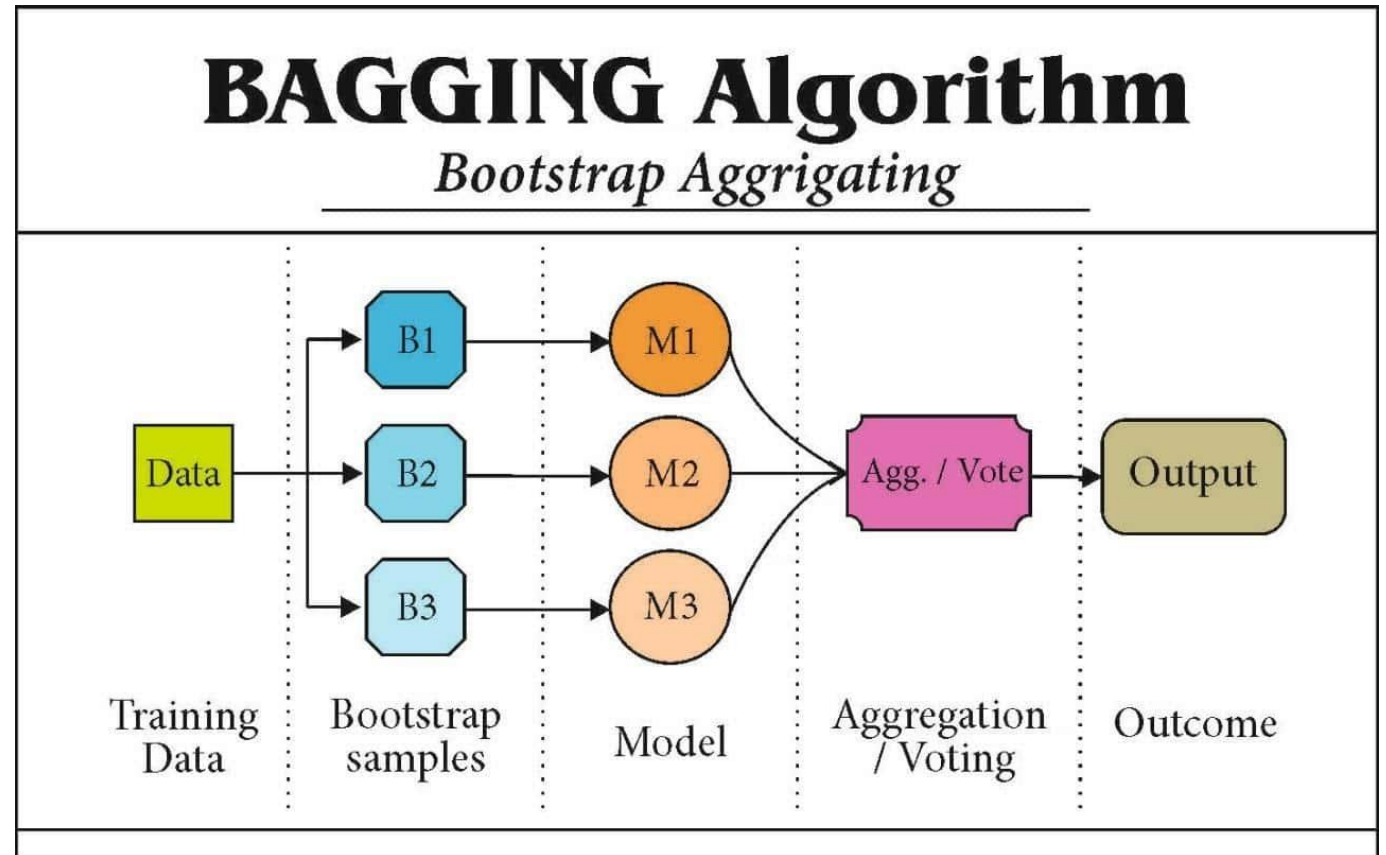
Random sampling

집계

같은 알고리즘에 대해

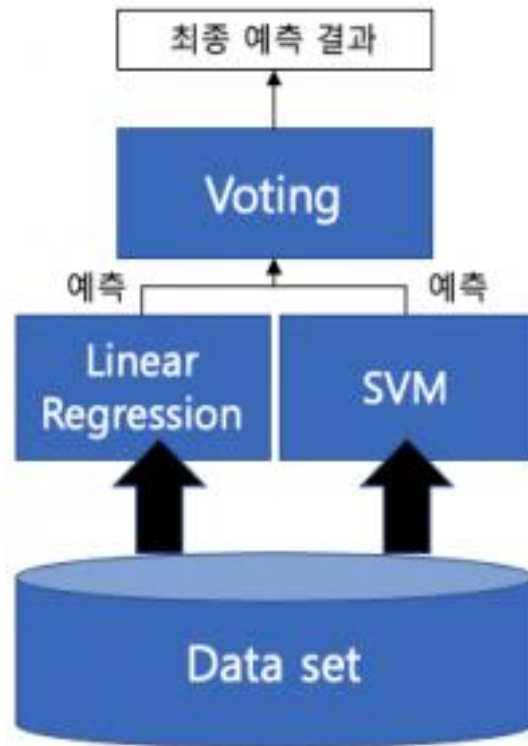
서로 다른 학습데이터로 학습시킨 후

투표를 통해 예측값을 구하는 방법

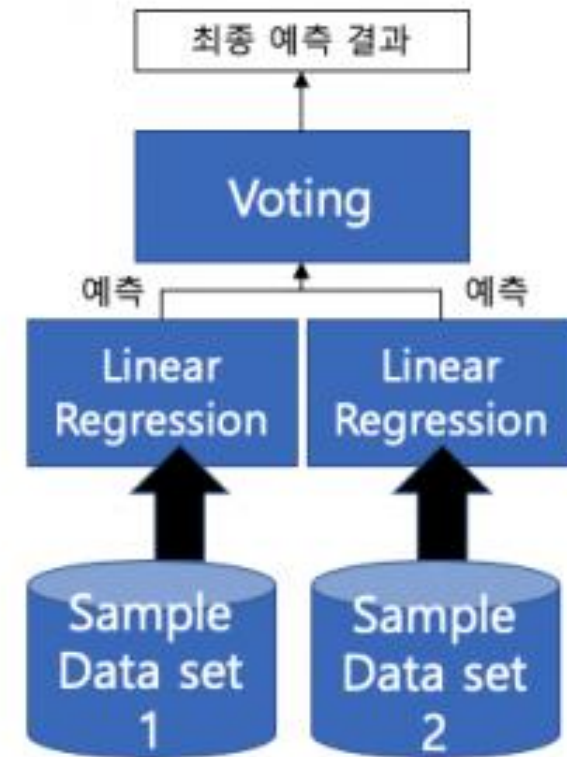


배깅 (Bagging)

voting



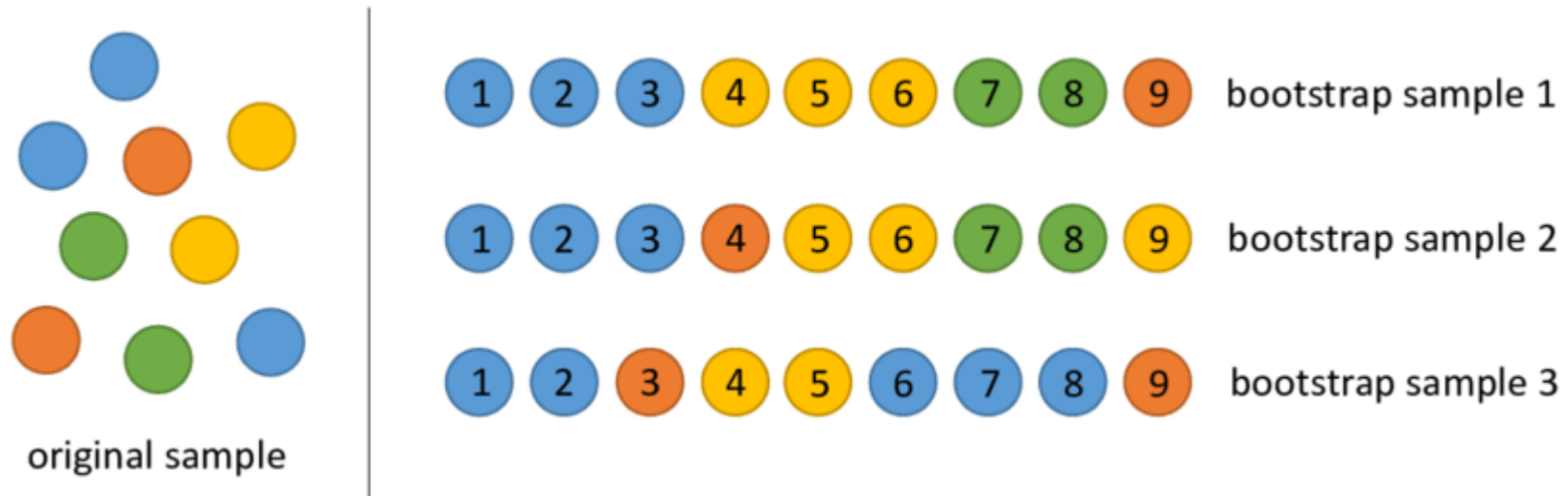
bagging



배깅 (Bagging)

Bootstrap

: random sampling 하는 방법, 단 복원추출



- ▷ 주어진 데이터가 충분하지 않을 때 데이터의 양의 임의적으로 늘림
- ▷ 데이터를 약간 편향되도록 샘플링하는 기법으로 샘플링을 통해 편향을 높여서 분산이 높은 모델의 과대적합을 줄이는 효과 제공

배깅 (Bagging)

Bootstrap

이론적으로 한 개체가 하나의 bootstrap에 한번도 선택되지 않을 확률

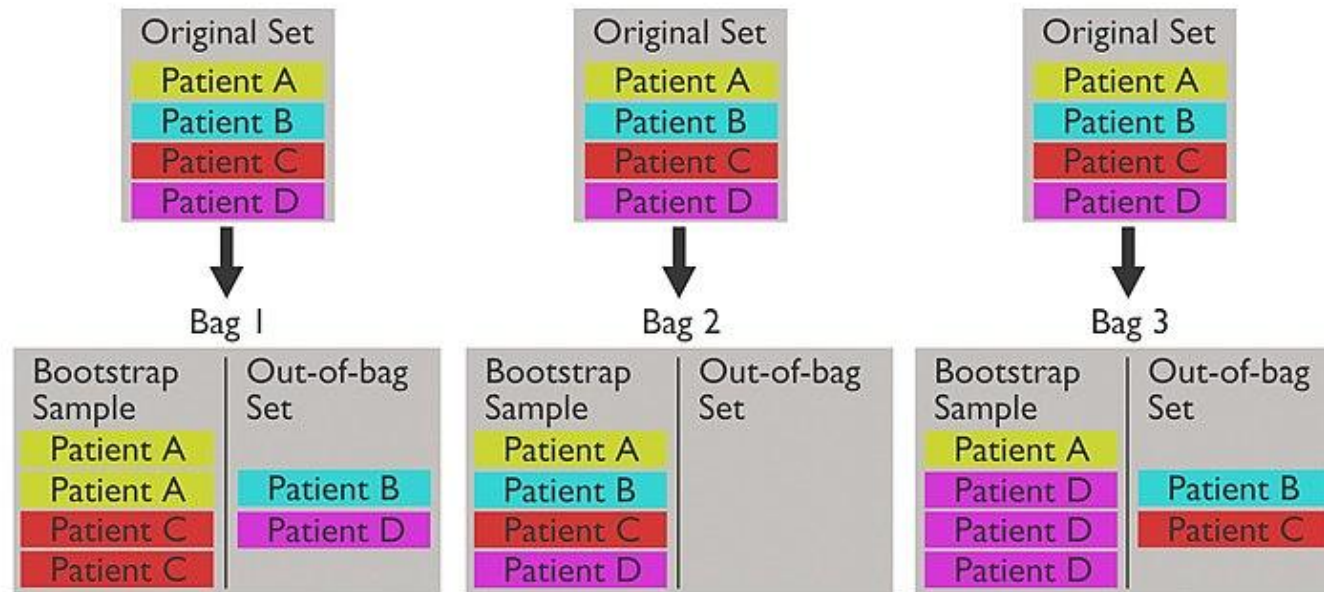
$$p = \left(1 - \frac{1}{N}\right)^N \rightarrow \lim_{N \rightarrow \infty} \left(1 - \frac{1}{N}\right)^N = e^{-1} = 0.368$$



배깅 (Bagging)

Oob error

$$p = \left(1 - \frac{1}{N}\right)^N \rightarrow \lim_{N \rightarrow \infty} \left(1 - \frac{1}{N}\right)^N = e^{-1} = 0.368$$



⇒ Out-Of-Bag Data

배깅 (Bagging)

Aggregating

여러 분류 모델이 예측한 값들을 조합해서 하나의 결론을 도출하는 과정

- Majority voting

$$Ensemble(\hat{y}) = \underset{i}{\operatorname{argmax}} \left(\sum_{j=1}^n I(\hat{y}_j = i), i \in \{0, 1\} \right)$$

Training Accuracy	Ensemble population	P(y=1) for a test instance	Predicted class label
0.80	Model 1	0.90	1
0.75	Model 2	0.92	1
0.88	Model 3	0.87	1
0.91	Model 4	0.34	0
0.77	Model 5	0.41	0
0.65	Model 6	0.84	1
0.95	Model 7	0.14	0
0.82	Model 8	0.32	0
0.78	Model 9	0.98	1
0.83	Model 10	0.57	1

$$\sum_{j=1}^n I(\hat{y}_j = 0) = 4$$

$$\sum_{j=1}^n I(\hat{y}_j = 1) = 6$$

$$Ensemble(\hat{y}) = 1$$

연속형 변수 → 평균

범주형 변수 → 다중투표

복습) 의사결정 트리 (Decision Tree)

의사결정 트리

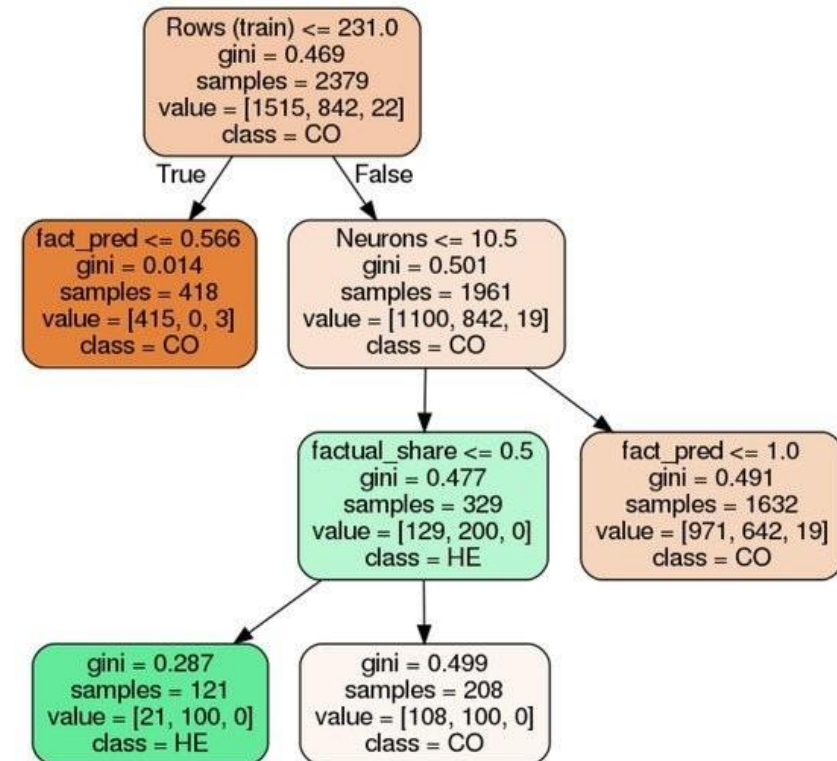
- 데이터에 내재되어 있는 패턴을 변수의 조합으로 나타내는 예측 / 분류 모델을 나무의 형태로 표현하는 기법
- 변수들로 기준을 만들고, 분류된 집단의 성질을 통해 새로운 관측 값을 추정하는 모형

〈장점〉

- 결과의 해석이 용이
- 직관적
- 범용성

〈단점〉

- 계층적 구조로 인해 중간에 에러가 발생하면 다음 단계로 전파
- 학습데이터의 미세한 변동이 결과에 큰 영향을 미침
- 높은 과적합의 위험

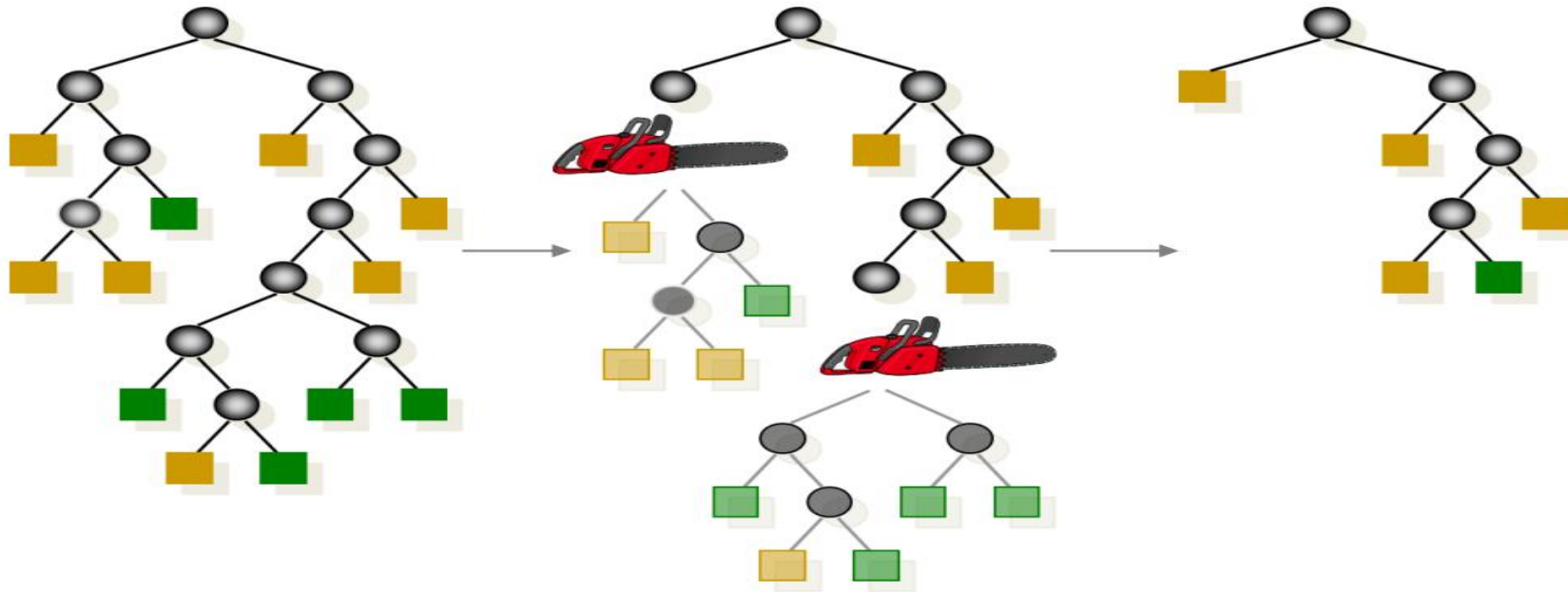


복습) 의사결정 트리 (Decision Tree)

의사결정 트리의 과적합을 피하기 위한 방법으로는

-> 초모수를 조정한 가지치기

-> 의사결정 트리를 base 모델로 하는 랜덤 포레스트



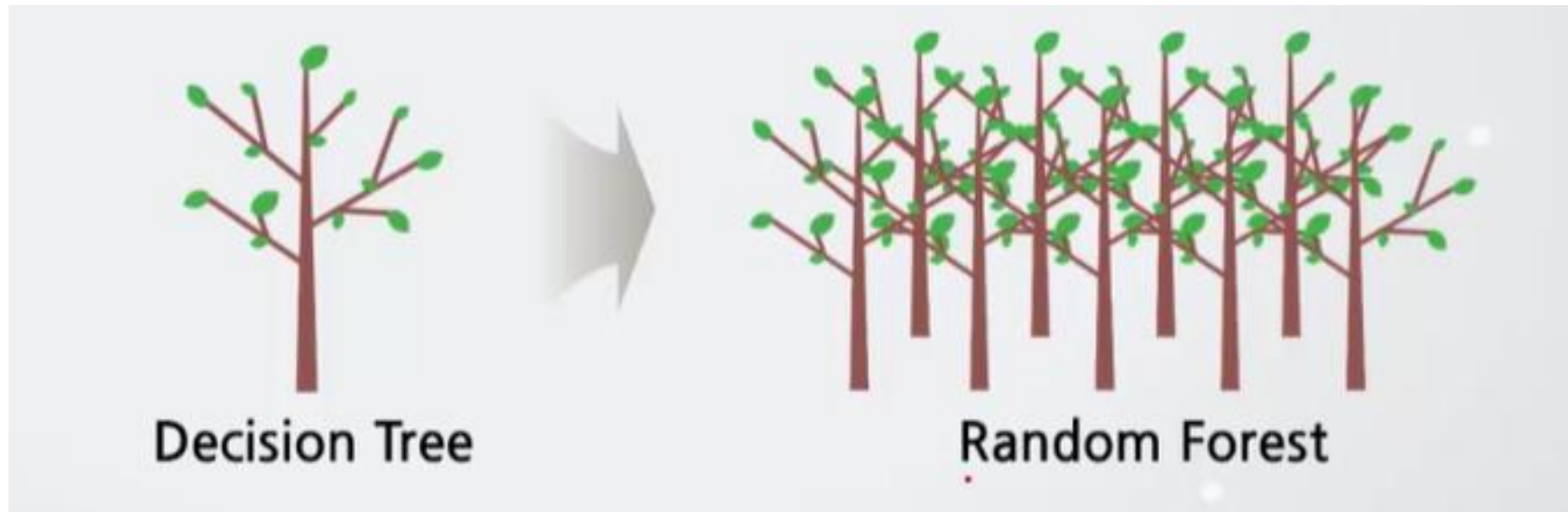
▷노드를 분할하기 위한 최소한의 데이터 샘플 수 (`min_samples_split`),
Leaf 노드가 되기 위한 최소한의 샘플 수(`min_samples_leaf`) 증가시키는 방법

복습) 의사결정 트리 (Decision Tree)

의사결정 트리의 과적합을 피하기 위한 방법으로는

-> 초모수를 조정한 가지치기

-> 의사결정 트리를 base 모델로 하는 랜덤 포레스트

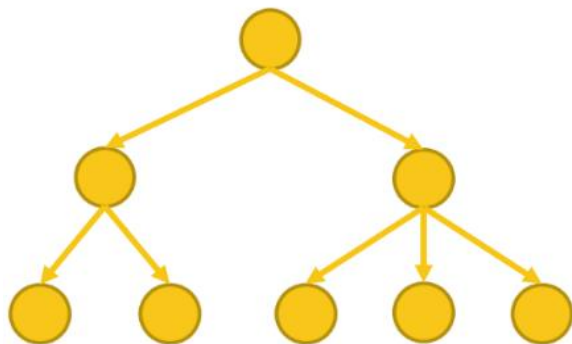


랜덤포레스트(Random Forest)

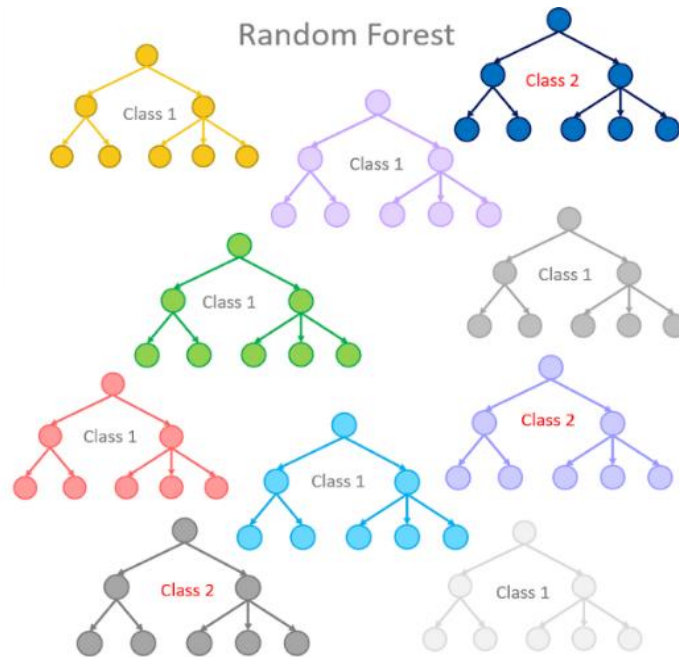
Random forest

- 배깅의 대표적 예시
- 의사결정 트리의 단점인 과적합을 회피할 목적으로 의사결정 트리를 base 모델로 하는 앙상블 모델
- 다수의 의사결정 트리 모델에 의한 예측/분류값을 종합하는 방법
- 하나의 의사결정 트리보다 높은 예측 정확성을 보임

Single Decision Tree



Random Forest



랜덤포레스트(Random Forest)

1) 라이브러리 임포트

```
from sklearn import datasets
from sklearn import tree
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score

import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# sklearn 모델의 동일한 결과 출력을 위한 코드
np.random.seed(5)
```

2) 손글씨 데이터 로드 및 feature와 target 나누기

```
mnist = datasets.load_digits()
features, labels = mnist.data, mnist.target
```

3) 교차 검증 평균 정확도를 계산하는 함수 정의

```
def cross_validation(classifier, features, labels):
    cv_scores = []

    for i in range(10):
        scores = cross_val_score(classifier, features, labels, cv=10, scoring='accuracy')
        cv_scores.append(scores.mean())

    return cv_scores
```

사용하려는 분류기 지정

정확도 계산

랜덤포레스트(Random Forest)

4) 교차 검증 평균 정확도를 계산

```
np.random.seed(5)
dt_cv_scores = cross_validation(tree.DecisionTreeClassifier(), features, labels)
dt_cv_scores
```

```
[0.8280229671011794,
0.8235630043451273,
0.8224674115456239,
0.8235692116697703,
0.8341464928615766,
0.8185692116697704,
0.8241247672253259,
0.8180136561142148,
0.8235630043451272,
0.8258038485412786]
```

의사결정 트리를 사용한
10번의 교차 검증 평균 정확도

```
#의사결정 트리 정확도
np.mean(dt_cv_scores)
```

0.8241843575418993

```
np.random.seed(5)
rf_cv_scores = cross_validation(RandomForestClassifier(), features, labels)
rf_cv_scores
```

```
[0.9487988826815641,
0.947135319677219,
0.9543575418994413,
0.94768156424581,
0.950471756672874,
0.9515704531346989,
0.9504593420235878,
0.9454655493482308,
0.9487833643699565,
0.9515704531346989]
```

랜덤 포레스트를 사용한
10번의 교차 검증 평균 정확도

```
#랜덤포레스트 정확도
np.mean(rf_cv_scores)
```

0.949629422718808

랜덤포레스트(Random Forest)

5) plot을 위한 데이터 프레임 생성

```
cv_list = [  
    ['random_forest', rf_cv_scores],  
    ['decision_tree', dt_cv_scores],  
]  
df = pd.DataFrame.from_dict(dict(cv_list))  
df
```

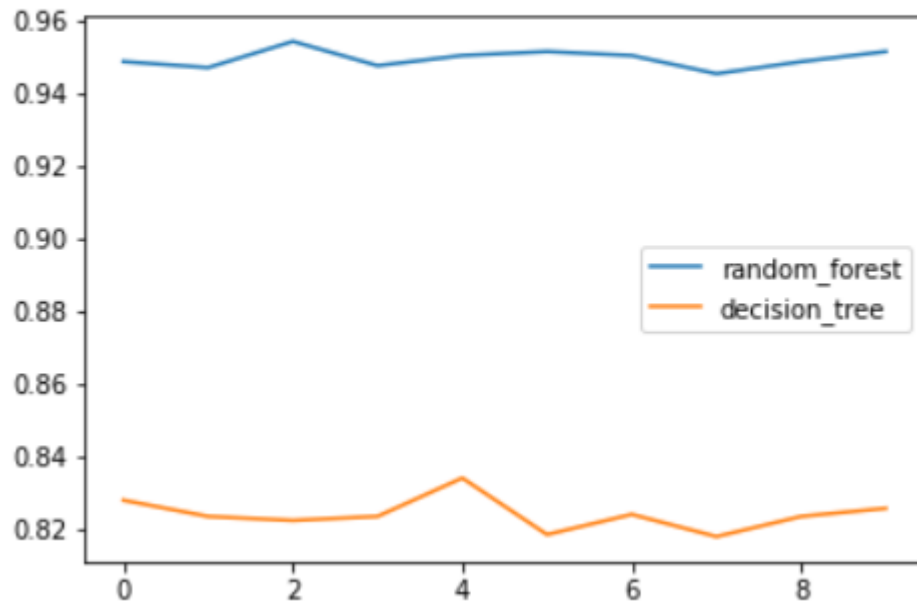
	random_forest	decision_tree
0	0.948799	0.828023
1	0.947135	0.823563
2	0.954358	0.822467
3	0.947682	0.823569
4	0.950472	0.834146
5	0.951570	0.818569



6) 그래프 출력

```
df.plot()
```

<matplotlib.axes._subplots.AxesSubplot at 0x7f6cd5042990>



랜덤 포레스트가 의사결정트리보다 정확도가 높은 것을 확인

랜덤포레스트(Random Forest)

랜덤 포레스트의 핵심 keyword

무작위성 “random” & 다양성 “diversity”

무작위성은 트리를 더욱 다양하게 만들고,

다양성은 분산을 낮추어(Less Overfitting) 더 훌륭한 모델을 만들어낸다.

랜덤포레스트(Random Forest)

랜덤 포레스트 단계

1. 부트스트랩을 통한 학습 데이터 샘플링

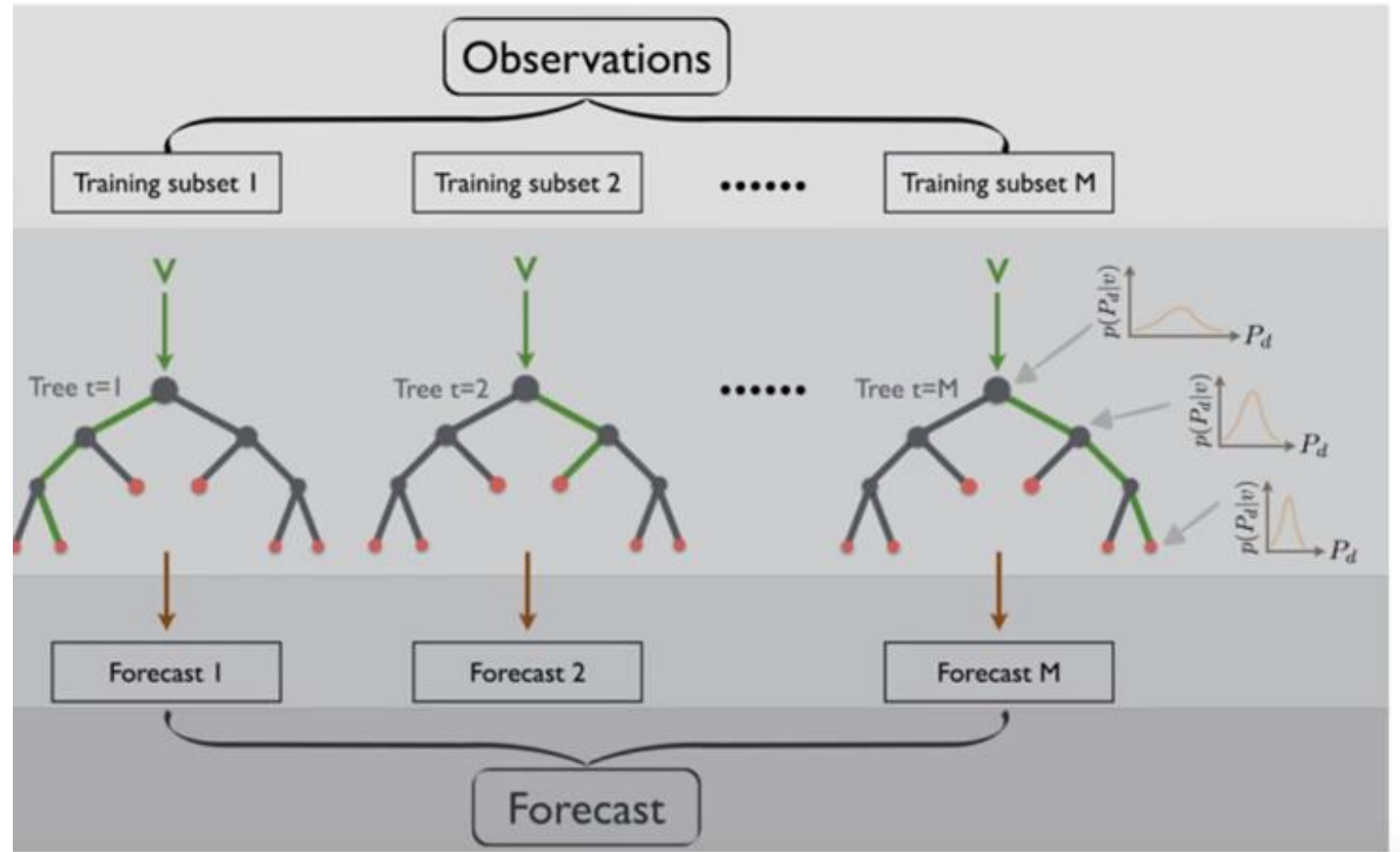
- > 다양성

2. 개별 의사결정 트리 훈련

-> 무작위성

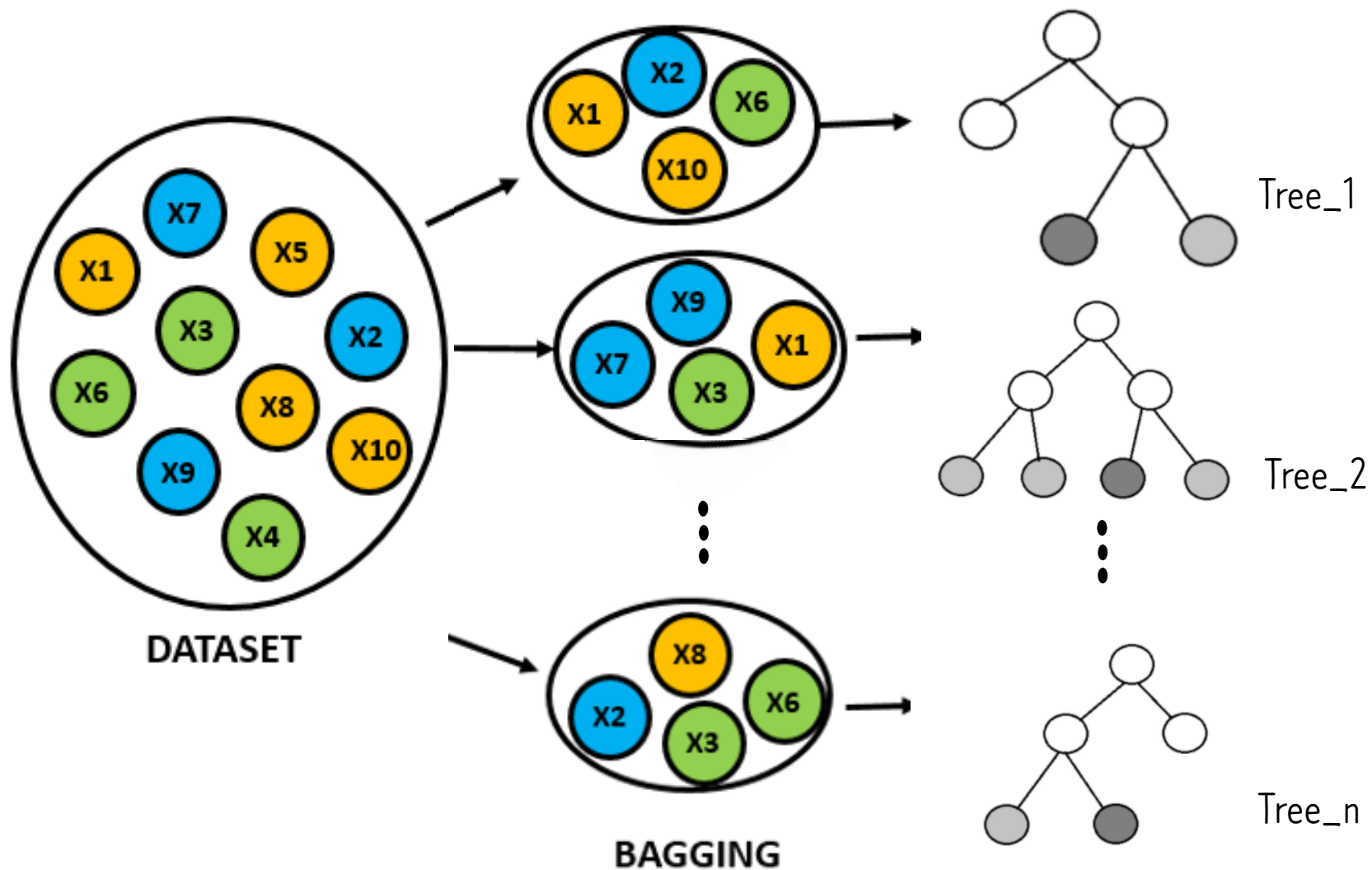
3. 개별 모델을 통한 예측

4. 결과 도출



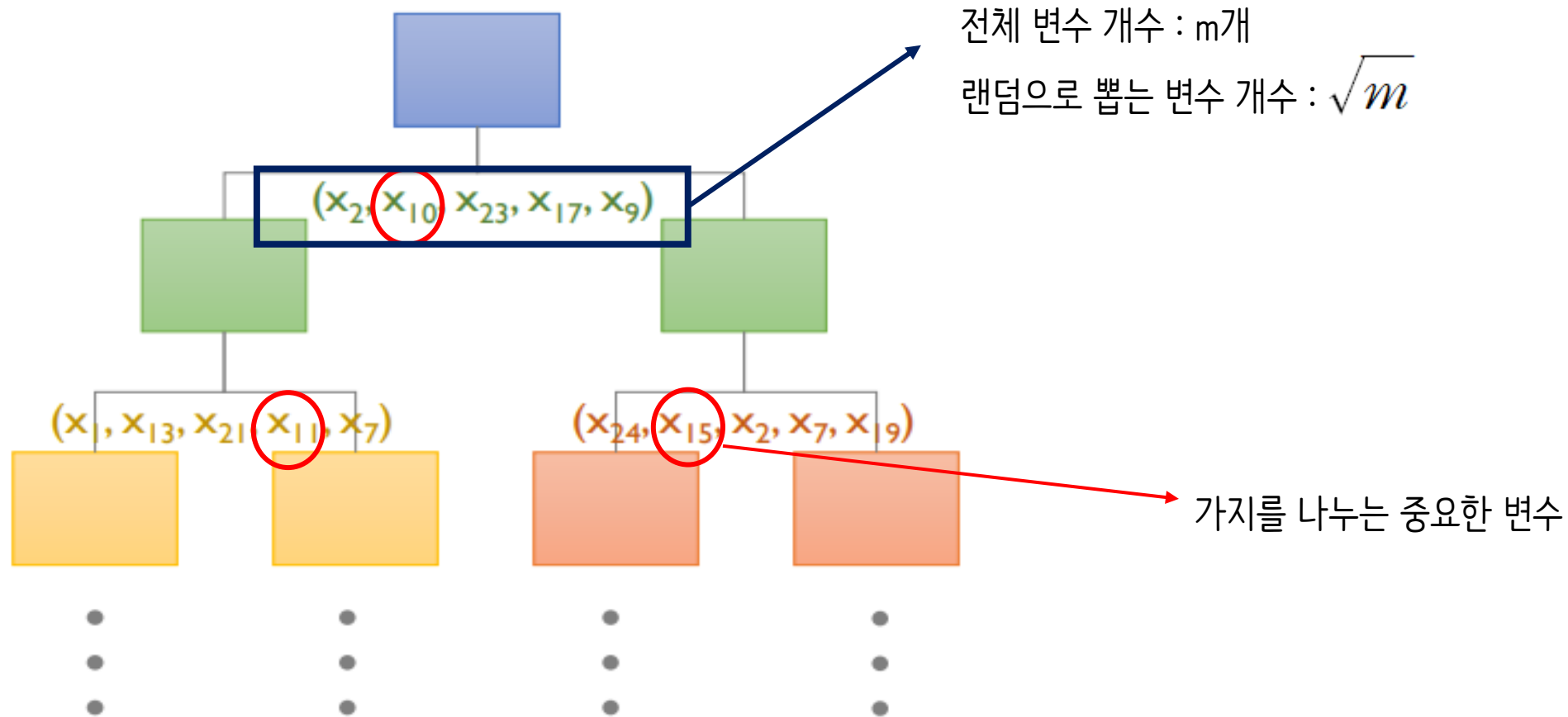
랜덤 포레스트

1. 부트스트랩을 통한 학습 데이터 샘플링 - > 다양성



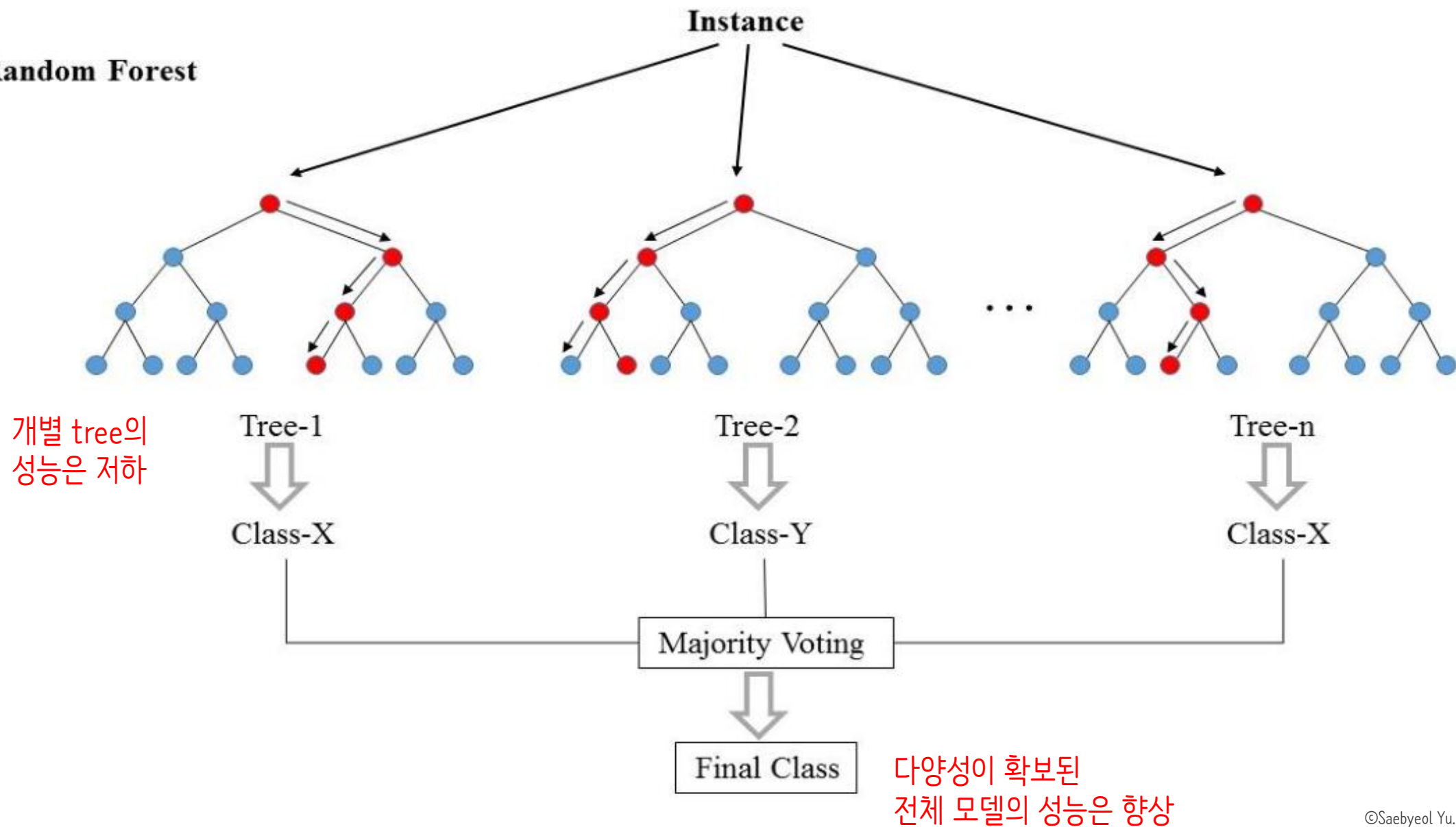
랜덤포레스트

2. 개별 의사결정 트리 훈련 -> 무작위성



랜덤포레스트

Random Forest



랜덤포레스트

변수 중요도

*OOB 데이터 : Bootstrap 샘플링 과정에서 선택되지 못한 데이터
= 학습에 사용되지 않은 데이터
= 검증 데이터 셋

*OOB score : 각 분류기에 OOB 샘플을 검증 데이터를 적용하여 도출한 정확도의 평균 → 모델 평가 지표

1. OOB 데이터를 이용해 각 트리의 OOB error 계산 : r_i
2. OOB 데이터의 특정 변수 X_i 를 임의로 섞은 데이터에 대한 OOB error 계산: e_i

3. $d_i = e_i - r_i, i = 1, 2, \dots, t$

$$\bar{d} = \frac{1}{t} \sum_{i=1}^t d_i$$

$$s_d^2 = \frac{1}{t-1} \sum_{i=1}^t (d_i - \bar{d})^2$$

4. 변수 중요도 $v_i = \frac{\bar{d}}{s_d}$

i 번째 변수의 데이터를 random하게 배치한 OOB data

X_A	X_B	X_C	Y
xa1	xb1	xc1	y1
xa2	xb2	xc2	y2
xa3	xb3	xc3	y3
xa4	xb4	xc4	y4
xa5	xb5	xc5	y5
xa6	xb6	xc6	y6

랜덤포레스트

랜덤 포레스트 _ 초모수

파라미터 명	설명
n_estimators	<ul style="list-style-type: none">- 결정트리의 갯수를 지정- Default = 10- 무작정 트리 갯수를 늘리면 성능 좋아지는 것 대비 시간이 걸릴 수 있음
min_samples_split	<ul style="list-style-type: none">- 노드를 분할하기 위한 최소한의 샘플 데이터수→ 과적합을 제어하는데 사용- Default = 2 → 작게 설정할 수록 분할 노드가 많아져 과적합 가능성 증가
min_samples_leaf	<ul style="list-style-type: none">- 리프노드가 되기 위해 필요한 최소한의 샘플 데이터수- min_samples_split과 함께 과적합 제어 용도- 불균형 데이터의 경우 특정 클래스의 데이터가 극도로 작을 수 있으므로 작게 설정 필요
max_features	<ul style="list-style-type: none">- 최적의 분할을 위해 고려할 최대 feature 개수- Default = 'auto' (결정트리에서는 default가 none이었음)- int형으로 지정 → 피쳐 갯수 / float형으로 지정 → 비중- sqrt 또는 auto : 전체 피쳐 중 $\sqrt{\text{피쳐개수}}$ 만큼 선정- log : 전체 피쳐 중 $\log_2(\text{전체 피쳐 개수})$ 만큼 선정
max_depth	<ul style="list-style-type: none">- 트리의 최대 깊이- default = None→ 완벽하게 클래스 값이 결정될 때 까지 분할또는 데이터 개수가 min_samples_split보다 작아질 때까지 분할- 깊이가 깊어지면 과적합될 수 있으므로 적절히 제어 필요
max_leaf_nodes	리프노드의 최대 개수

의사결정 트리의 하이퍼파라미터
+
배깅, 부스팅, 학습, 정규화 등의
하이퍼 파라미터 추가

→ 튜닝 할 파라미터 증가

결정 트리에서는 노드를 분할 할 때
모든 feature를 고려 : default = none
Vs

랜덤 포레스트에서는 랜덤으로 선택 된
Feature를 고려 : default = 'auto'
→ \sqrt{m}

랜덤 포레스트 실습

Human Activity 데이터 사용

-> 5개 데이터셋 (feature, X_train, y_train, X_test, y_test)

1) 데이터 전처리

#1-1) feature 셋 불러오기 및 데이터 확인

```
feature_name_df = pd.read_csv('./features.txt', sep='#s+',  
                               header=None, names=['column_index', 'column_name'])
```

```
print(feature_name_df.shape)
```

```
feature_name_df.groupby('column_name').count().sort_values('column_index')
```

tBodyGyro-arCoeff()-X,4	1
...	...
fBodyAccJerk-bandsEnergy()-25,48	3
fBodyAccJerk-bandsEnergy()-33,48	3
fBodyAccJerk-bandsEnergy()-41,48	3

- ▷ feature셋에 총 561개의 feature 존재
- ▷ groupby()를 통해 동일한 컬럼 이름을 갖는 특성들이 있음을 확인

랜덤포레스트

1) 데이터 전처리

#1-2) 중복된 feature name을 구분하기 위한 함수 정의

```
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
```

▷ 예를 들어 “A” 라는 feature name이 3개 존재 -> 순차적으로 A, A_1, A_2으로 변경하여 중복을 피함

fBodyAcc-bandsEnergy()-1,16
fBodyAcc-bandsEnergy()-1,16
fBodyAcc-bandsEnergy()-1,16

fBodyAcc-bandsEnergy()-1,24
fBodyAcc-bandsEnergy()-1,24
fBodyAcc-bandsEnergy()-1,24

fBodyAcc-bandsEnergy()-1,16
fBodyAcc-bandsEnergy()-1,16_1
fBodyAcc-bandsEnergy()-1,16_2

fBodyAcc-bandsEnergy()-1,24
fBodyAcc-bandsEnergy()-1,24_1
fBodyAcc-bandsEnergy()-1,24_2

랜덤포레스트

1) 데이터 전처리

#1-3) 데이터셋을 불러오기 위한 함수 정의

```
def get_human_dataset( ):

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

    #중복된 feature명을 수정하는 get_new_feature_names_df()함수 이용
    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('./X_train.txt', sep='#s+', names=feature_name)
    X_test = pd.read_csv('/content/drive/MyDrive/data/X_test.txt', sep='#s+', names=feature_name)

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

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

#데이터셋 불러오기
X_train, X_test, y_train, y_test = get_human_dataset()
```

랜덤포레스트

1) 데이터 전처리

```
#1-4) 학습 데이터 차원 확인  
X_train.shape , y_train.shape
```

```
((7352, 561), (7352, 1))
```

```
#1-4) 테스트 데이터 차원 확인  
X_test.shape , y_test.shape
```

```
((2947, 561), (2947, 1))
```

```
X_train.head()
```

	tBodyAcc- mean()-X	tBodyAcc- mean()-Y	tBodyAcc- mean()-Z	tBody sto
0	0.288585	-0.020294	-0.132905	-0.99
1	0.278419	-0.016411	-0.123520	-0.99
2	0.279653	-0.019467	-0.113462	-0.99
3	0.279174	-0.026201	-0.123283	-0.99
4	0.276629	-0.016570	-0.115362	-0.99

5 rows × 561 columns

```
y_train
```

	act ion
0	5
1	5
2	5
3	5
4	5
...	...
7347	2
7348	2
7349	2
7350	2
7351	2

7352 rows × 1 columns

▷ 학습데이터와 테스트 데이터의 차원을 확인

▷ 데이터 형태 확인

* Y (target)값은 1~6의 행동 분류 범주 label

랜덤포레스트

2) 랜덤 포레스트 학습

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

```
# 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: {0:.4f}'.format(accuracy))
```

랜덤 포레스트 정확도: 0.9253

- ▷ 랜덤 포레스트를 사용하기 위해 사이킷 런의 앙상블로부터 RandomForestClassifier을 импорт
 - ▷ 랜덤 포레스트 모델 rf_clf을 생성하고 학습 데이터를 훈련
 - ▷ Predict를 통해 X_test의 예측값 저장
- 하이퍼파라미터를 지정하지 않고 훈련 시킨 랜덤 포레스트의 정확도 : 0.9253

랜덤포레스트

3) 랜덤 포레스트 하이퍼 파라미터 튜닝

```
from sklearn.model_selection import GridSearchCV
params = {
    'n_estimators':[50],
    '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)
#하이퍼 파라미터
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': 12, 'min_samples_leaf': 18, 'min_samples_split': 8, 'n_estimators': 50 }

최고 예측 정확도: 0.9173

▷ GridSearchCV를 이용해서 파라미터 튜닝

▷ 튜닝시간을 절약하기 위해서 n_estimator는 50, cv=2로 설정하여 최적의 파라미터를 구한 후 n_estimator를 추후에 증가시켜 성능 평가

▷ 멀티 코어 cpu 환경에서 **n_jobs = -1** 파라미터를 추가하면 모든 CPU 코어를 이용해 학습 가능

랜덤포레스트

3) 랜덤 포레스트 하이퍼 파라미터 튜닝

```
#최적 파라미터 조합에서 n_estimators를 300으로 증가
rf_clf1 = RandomForestClassifier(n_estimators = 300 , max_depth = 12, min_samples_leaf =18,
                                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 : 0.9186

▷최적의 파라미터 조합을 유지하면서 base tree 개수인 **n_estimators**를 300으로 증가->예측 정확도가 조금 향상 됨
(기본적으로 개별 tree 개수가 많아지면 랜덤 포레스트 성능이 향상되지만 , 무한정으로 많아지면 학습 시간도 길어짐에 유의 !)

랜덤포레스트

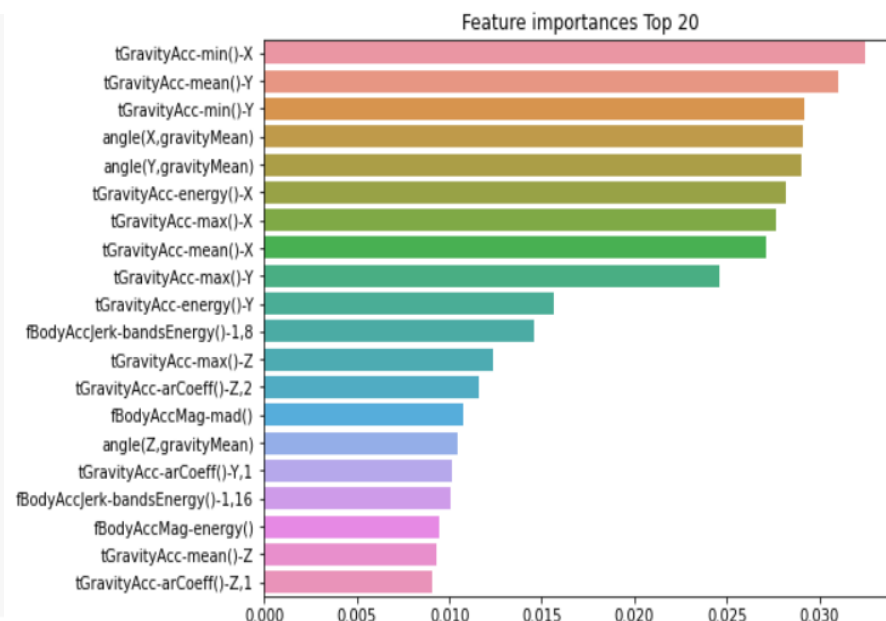
4) 알고리즘이 선택한 피처의 중요도 시각화

```
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)
plt.show()
```

- ▷ 피처의 중요도는 feature_importances_에 저장
- ▷ 의사결정 트리의 장점인 직관적 해석이 랜덤포레스트에서도 유지
- * 휴먼 행동을 분류하는데 가장 중요한 피처는 “tGravityAcc-min()-X”



- > 상위 20개의 피처 확인 한 결과

4

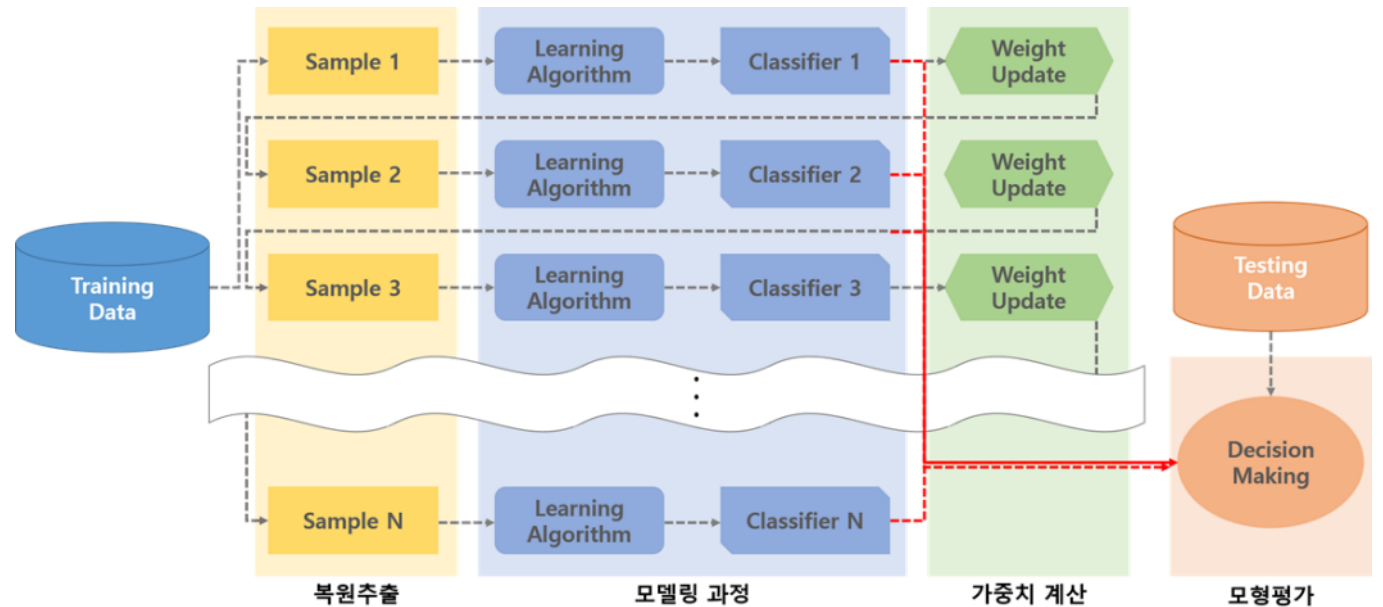
부스팅(Boosting)



Boosting이란?

Weak learner들을 순차적으로 여러 개 결합하여 성능을 높이는 알고리즘이다.

순차적 학습과 가중 투표가
Boosting의 가장 큰 특징!

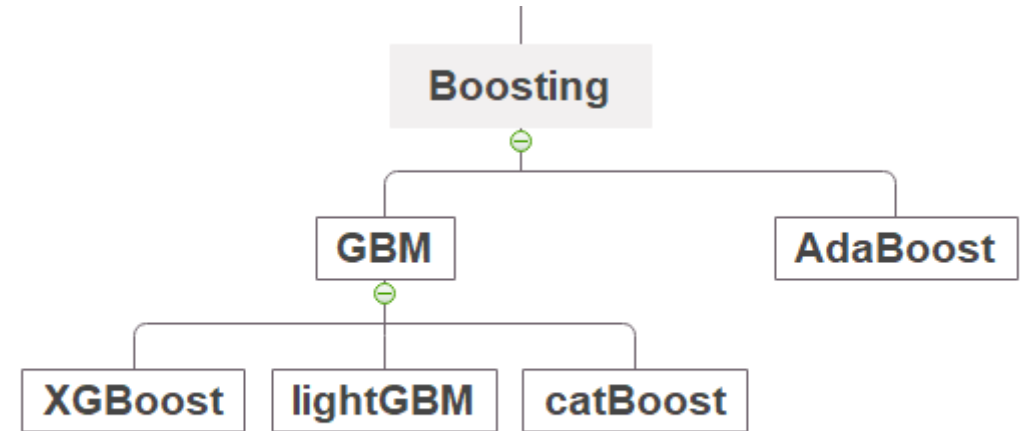


부스팅

Bagging과 Boosting

Boosting의 종류

Boosting 기법에는 AdaBoost, Gradient Boosting(GBM), LightGBM, XGBoost 등이 있다.



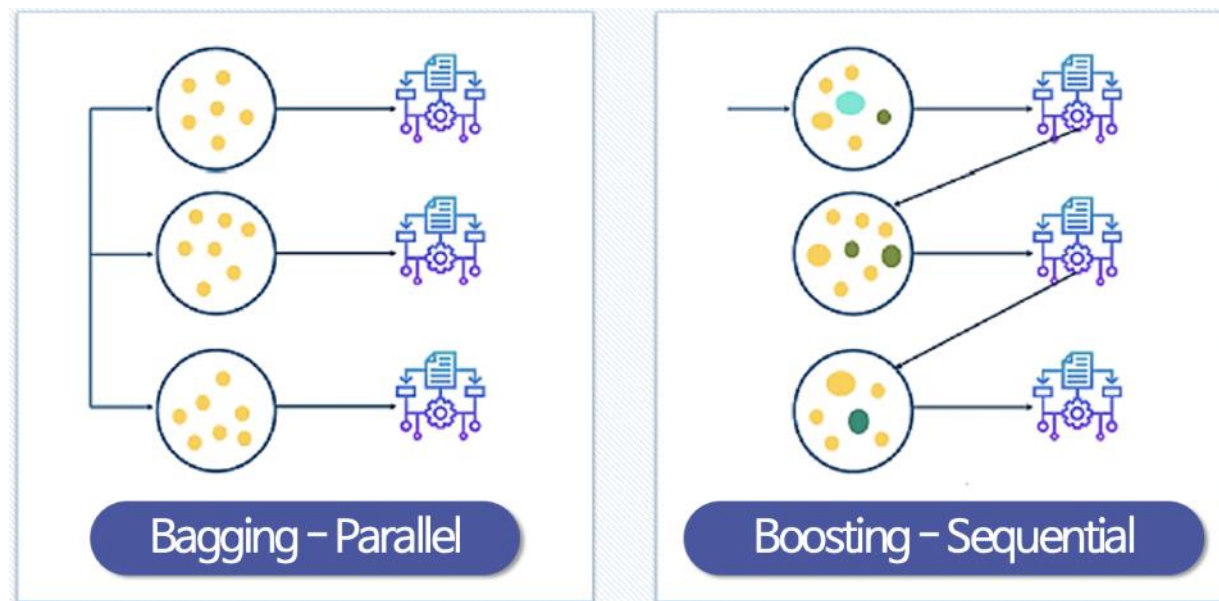
부스팅

Bagging과 Boosting

Boosting과 Bagging의 비교

배깅은 병렬적으로 학습이 이뤄지지만
부스팅은 순차적으로 학습이 이뤄진다.

순차적 학습은 부스팅의 가장 큰 특징!



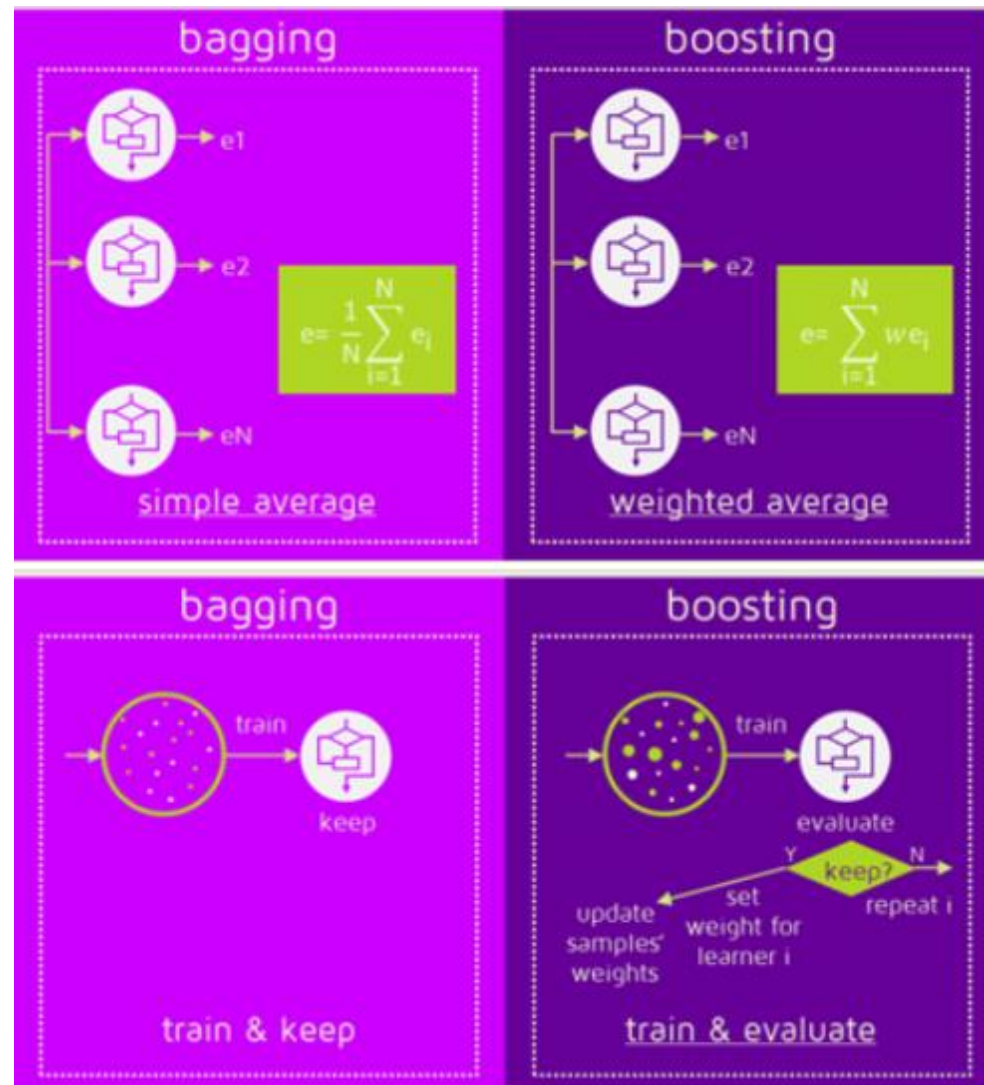
부스팅

Bagging과 Boosting

Boosting과 Bagging의 비교

배깅은 $1/n$ 으로 가중치를 주지만, 부스팅은 오차가 큰 개체에 더 높은 가중치를 부여함으로써 모델의 성능을 높이하고자 한다

배깅은 트레이닝 셋을 만들어 업데이트 없이 가지고 있는 반면, 부스팅은 트레이닝 셋을 만든 후에 업데이트 및 조정하는 과정이 존재한다.



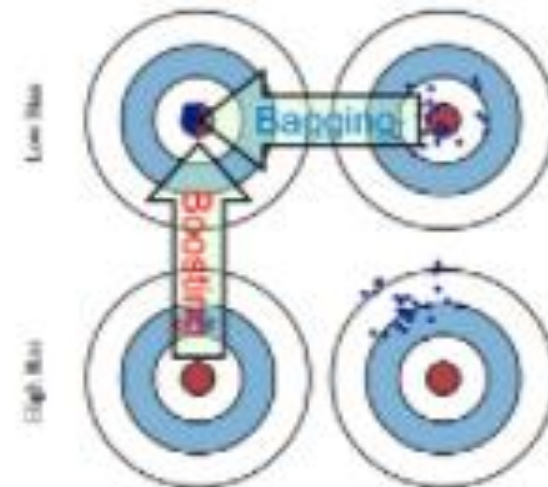
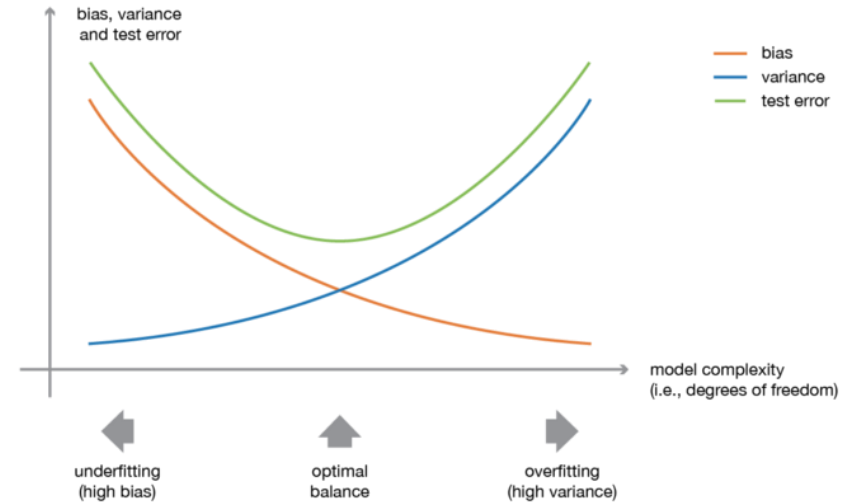
부스팅

Bagging과 Boosting

Boosting과 Bagging의 비교

Bagging의 경우, Bias가 낮은 모델들을 이용해서 variance를 낮추는 반면, Boosting은 variance가 낮은 모델들을 합쳐서 bias를 줄인다.

이는 Bagging은 각각의 모델들이 서로 독립적이지만 Boosting은 이전 모델의 오류를 고려해 가중치를 조정하기 때문이다.



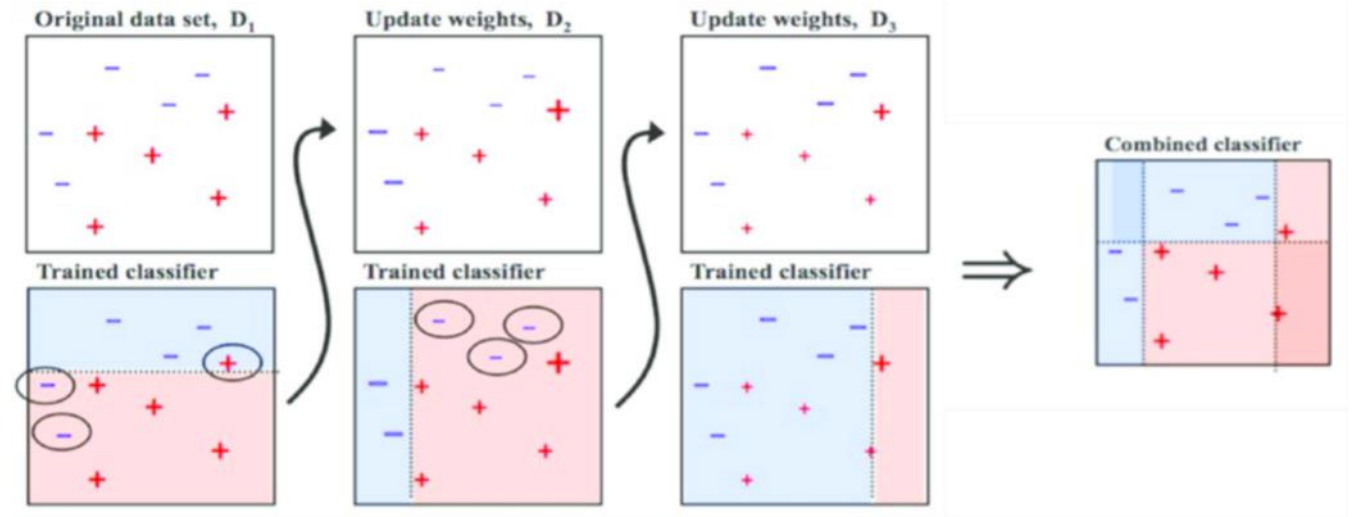
The Famous Bias/Varian Tradeoff

Adaboost란?

Adaptive + Boosting

약한 분류기들을 순차적으로 학습시키고, 이들을 조합하여 강한 분류기의 성능을 향상시키는 것이다.

이전 모델이 잘못 분류한 데이터들의 가중치를 높임으로써 다음 모델이 잘못 분류한 데이터에 더 집중할 수 있도록 한다.

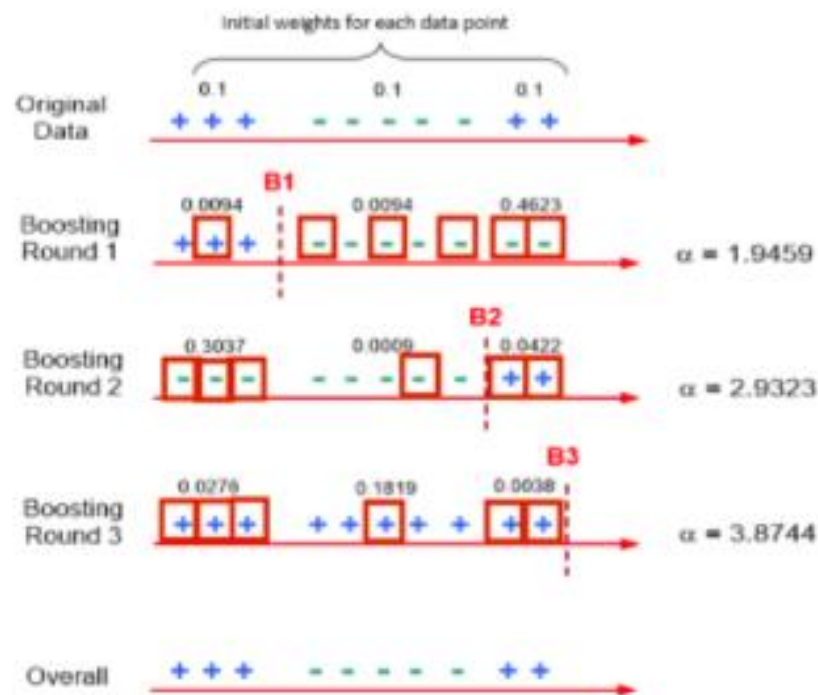


$$H(x) = \alpha_1 h_1(x) + \alpha_2 h_2(x) + \dots + \alpha_t h_t(x) = \sum_{t=1}^T \alpha_t h_t(x)$$

Adaboost

▪ AdaBoost (Adaptive Boost)의 예시

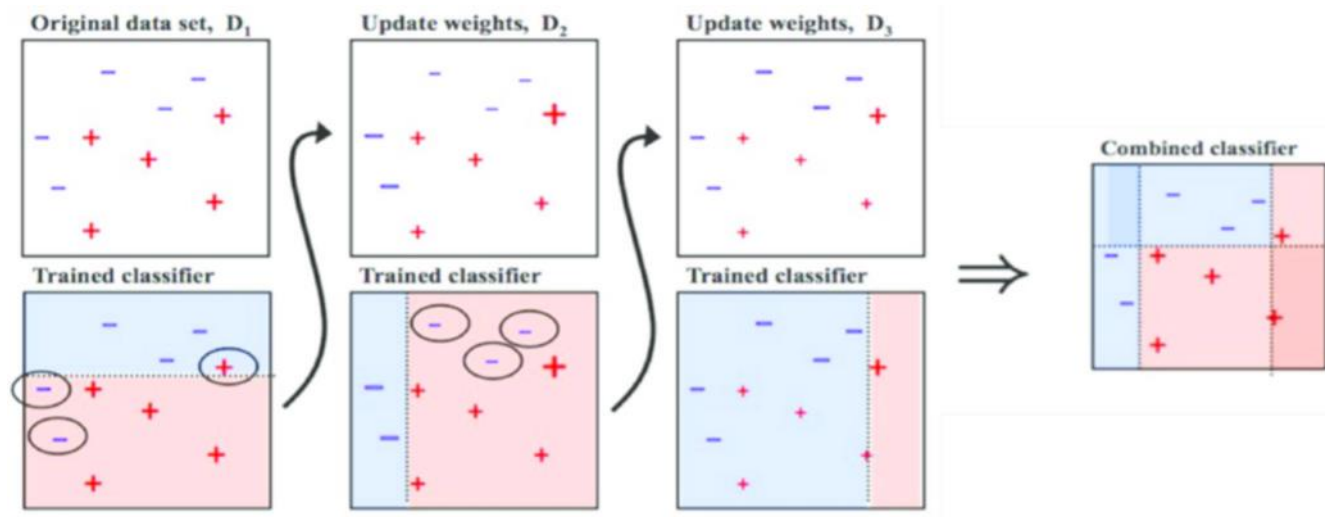
- ① 모든 데이터에 대해 가중치를 동일하게 0.1로 설정
- ② Round 1에서 빨간색 네모의 데이터가 수집되며 이를 기반으로 분류 기준값인 B1을 설정(B1보다 작은 경우 +, 큰 경우 -로 분류)
- ③ 데이터 i 에 대해 m 번째 round에서의 가중치를 업데이트 (오분류된 데이터에 가중치를 크게, 정분류된 데이터에 가중치를 작게 설정)
- ④ 업데이트한 가중치의 확률로 샘플을 재수집
- ⑤ 4번의 결과로 Round 2의 빨간색 네모의 데이터가 수집
- ⑥ 수집된 데이터로 모델을 학습한 결과 B2가 분류 기준값으로 도출됨
- ⑦ 가중치를 업데이트
- ⑧ 이와 같은 방법을 설정한 반복 횟수만큼 반복
- ⑨ 예시에서 Round 3까지의 결과를 종합하면 Original Data와 동일한 결과가 도출됨



부스팅

Adaboost의 알고리즘

1. 동일한 가중치 부여
2. Amount of Say를 구함
3. 샘플가중치를 업데이트한다.
(가중치의 합이 1이 아니면 특정 수를 곱해 1이 되도록 조정해준다.)
4. 이 과정을 반복한다.



$$H(x) = \alpha_1 h_1(x) + \alpha_2 h_2(x) + \dots + \alpha_t h_t(x) = \sum_{t=1}^T \alpha_t h_t(x)$$

부스팅

Adaboost의 알고리즘

1. 동일한 가중치 부여
2. Amount of Say를 구함

옆의 그림에서 빨간색이 잘못 분류된 case라고 하면, $\text{totalerror} = 1/8$ 따라서 Amount of Say는 약 0.97

Chest Pain	Blocked Arteries	Patient Weight	Heart Disease	Sample Weight
Yes	Yes	205	Yes	1/8
No	Yes	180	Yes	1/8
Yes	No	210	Yes	1/8
Yes	Yes	167	Yes	1/8
No	Yes	156	No	1/8
No	Yes	125	No	1/8
Yes	No	168	No	1/8
Yes	Yes	172	No	1/8

$$\text{Amount of Say} = \frac{1}{2} \log\left(\frac{1 - \text{Total Error}}{\text{Total Error}}\right)$$

부스팅

Adaboost의 알고리즘

3. 샘플 가중치를 업데이트한다.

잘못 분류된 sample의 경우 위의 경우처럼 weight를 조정해 가중치를 증가시켜주고, 잘 분류했던 sample의 경우 아래의 경우처럼 weight를 조정해 가중치를 감소시켜준다.

계산해보면

잘못 분류했던 sample:

$$(1/8) * e^{(0.97)} = (1/8) * 2.64 = 0.33$$

잘 분류했던 sample:

$$(1/8) * e^{(-0.97)} = (1/8) * 0.38 = 0.05$$

Chest Pain	Blocked Arteries	Patient Weight	Heart Disease	Sample Weight
Yes	Yes	205	Yes	1/8
No	Yes	180	Yes	1/8
Yes	No	210	Yes	1/8
Yes	Yes	167	Yes	1/8
No	Yes	156	No	1/8
No	Yes	125	No	1/8
Yes	No	168	No	1/8
Yes	Yes	172	No	1/8

New Sample Weight = sample weight $\times e^{\text{amount of say}}$

This is the formula we will use to *increase* the **Sample Weight** for the sample that was *incorrectly* classified.

Chest Pain	Blocked Arteries	Patient Weight	Heart Disease	Sample Weight
Yes	Yes	205	Yes	1/8
No	Yes	180	Yes	1/8
Yes	No	210	Yes	1/8
Yes	Yes	167	Yes	1/8
No	Yes	156	No	1/8
No	Yes	125	No	1/8
Yes	No	168	No	1/8
Yes	Yes	172	No	1/8

New Sample Weight = sample weight $\times e^{-\text{amount of say}}$

This is the formula we will use to *decrease* the **Sample Weights**.

부스팅

Adaboost의 알고리즘

3. 가중치의 합이 1이 되지 않으면...

하지만 이 경우, 가중치의 합이 0.68로 1이 되지 않는다. 따라서 다음과 같은 과정을 통해 가중치를 조정 해준다.

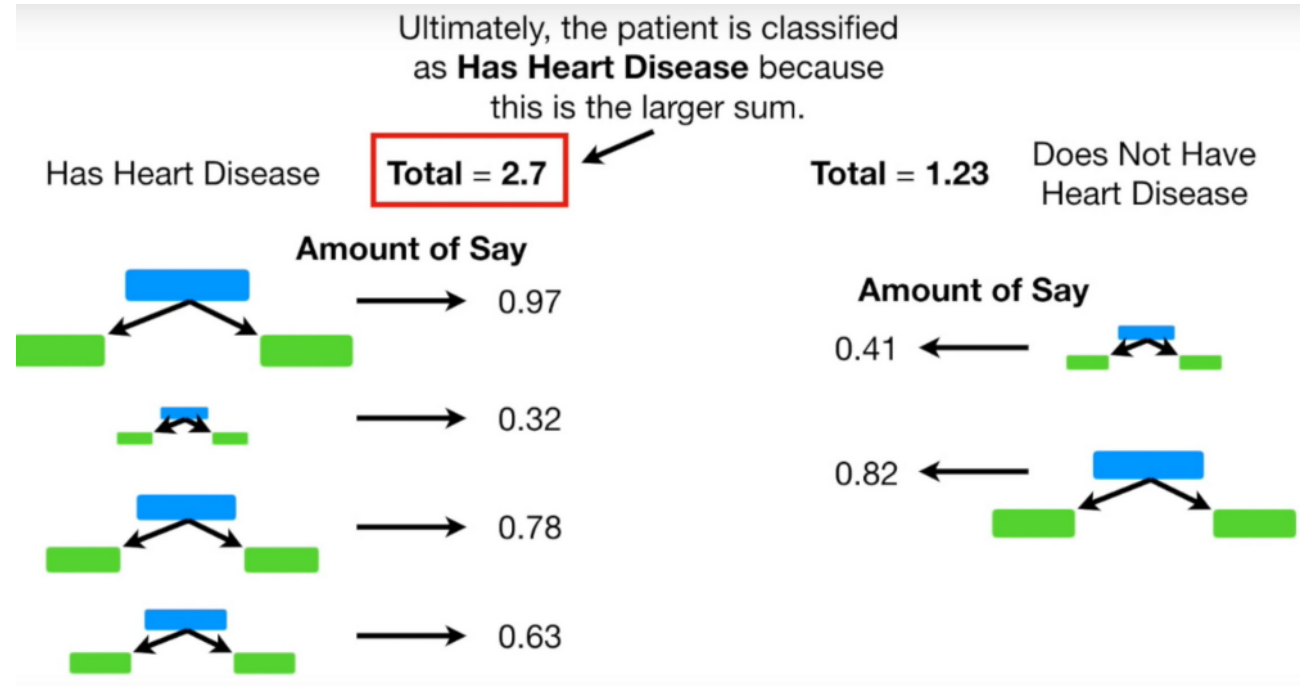
$$0.33 \rightarrow 0.33/0.68 = 0.49$$

$$0.05 \rightarrow 0.05/0.68 = 0.07$$

Chest Pain	Blocked Arteries	Patient Weight	Heart Disease	Sample Weight	New Weight	Norm. Weight
Yes	Yes	205	Yes	1/8	0.05	0.07
No	Yes	180	Yes	1/8	0.05	0.07
Yes	No	210	Yes	1/8	0.05	0.07
Yes	Yes	167	Yes	1/8	0.33	0.49
No	Yes	156	No	1/8	0.05	0.07
No	Yes	125	No	1/8	0.05	0.07
Yes	No	168	No	1/8	0.05	0.07
Yes	Yes	172	No	1/8	0.05	0.07

4. 이 과정을 반복한다.

이 과정을 반복한 후, 각각의 클래스로 분류한 모델들의 Amount of Say의 합을 구한 후, 그 값이 가장 큰 클래스로 분류한다.



$$H(x) = \alpha_1 h_1(x) + \alpha_2 h_2(x) + \dots + \alpha_t h_t(x) = \sum_{t=1}^T \alpha_t h_t(x)$$

부스팅

AdaBoost의 하이퍼파라미터

AdaBoost의 하이퍼파라미터		
번호	이름	설명
1	base_estimators	학습에 사용되는 알고리즘 Default: 깊이가 1인 DecisionTree
2	n_estimators	생성할 weak learner의 개수 Default=50
3	learning_rate	학습률(0~1) weak learner가 순차적으로 오류 값을 보정해 나갈 때 적용하는 계수 Default = 1

Adaboost의 장점과 단점

장점

과적합 되는 경향을 줄일 수 있다.

정확도가 높다

단점

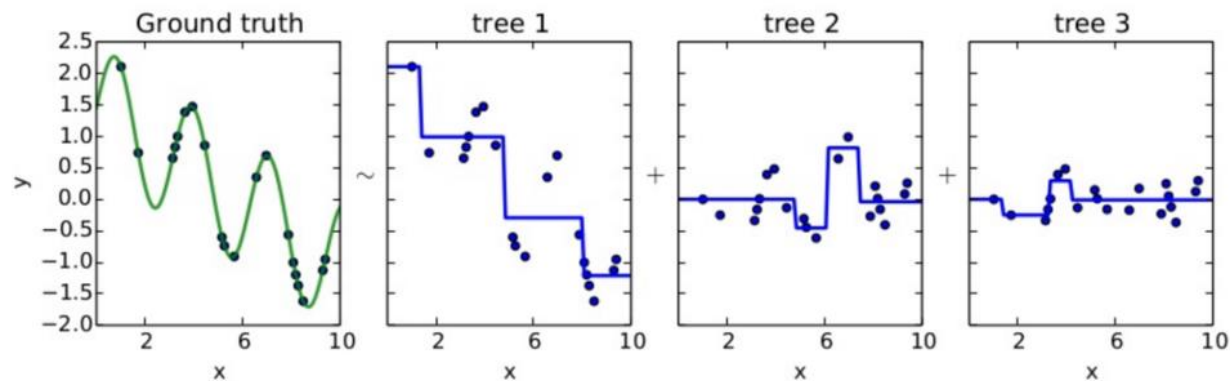
노이즈 데이터 및 이상치에 민감하다.

Gradient boost란?

Gradient Descent(경사하강법) + Boosting

Adaboost와 비슷하지만 Gradient Boost의 경우

가중치 업데이트를 경사하강법을 이용하여 진행

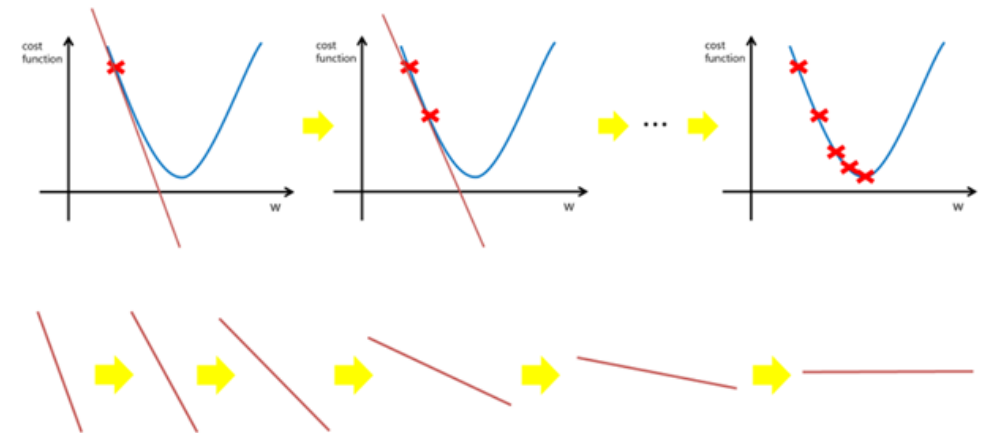
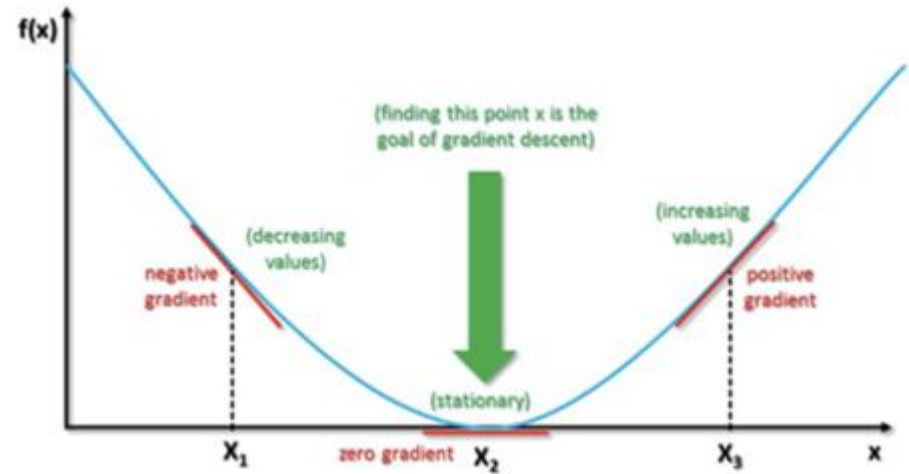


부스팅

경사하강법

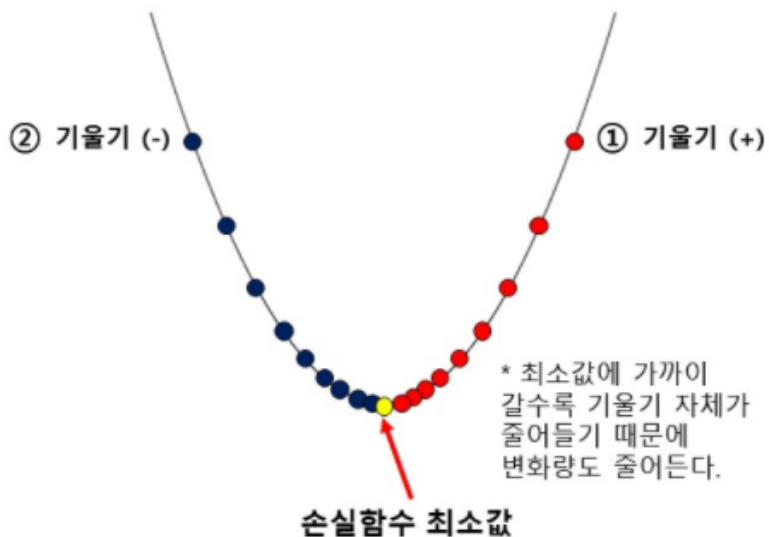
경사하강법이란?

함수의 기울기(경사)를 구하고, 기울기가 낮은 쪽으로 계속 이동시켜 최적값에 이를 때까지 반복하는 방법



경사하강법의 진행과정

1. 시작점을 선택
2. 비용(손실)함수 계산(J)
3. 파라미터 값 업데이트
4. 반복학습



경사하강법의 과정을 식으로 나타내면 아래와 같다.

For each update on Loss Function J :

$$x = x - r * \frac{dJ}{dx}$$

변수 x에 순간기울기의 일정비율(r, learning rate) 만큼을 빼주는 것이다. 그래프에서 1번 지점에서는 기울기가 +이기 때문에 최소값에 다다르기 위해서는 x값을 감소 시켜야 한다. 기존의 x값에 기울기 +값을 빼주면 x 값이 감소하기 때문에 최소값에 가까이 가게 된다.

부스팅

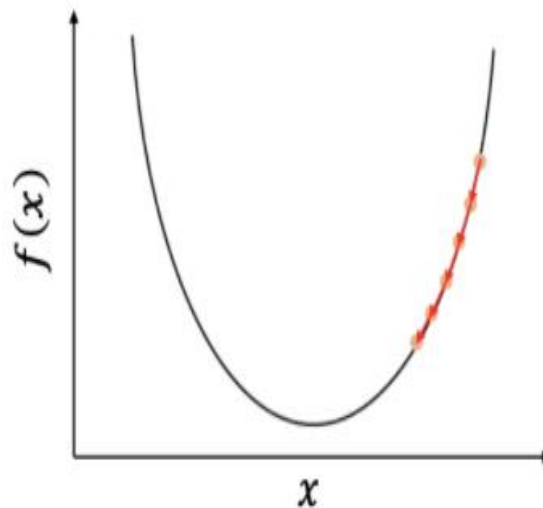
경사하강법

학습률

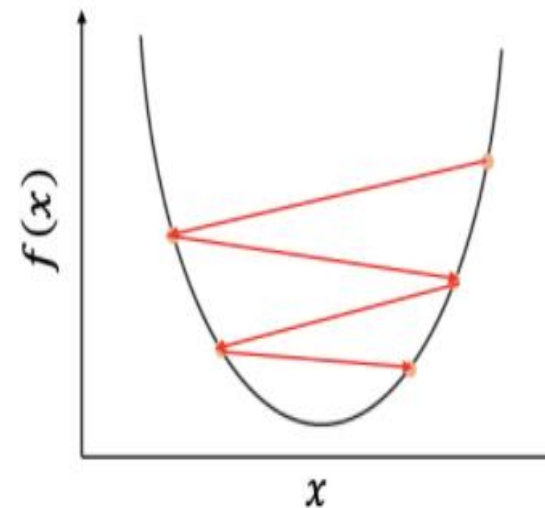
학습률은 한 번에 얼마나 이동을 할 것인지 결정한다(step size).

학습률이 너무 크면 최솟값으로 수렴하지 못하고, 함수 값이 계속 커지는 상황이 발생할 수 있다.

학습률이 너무 작으면 최적의 값을 구하는데 소요되는 시간이 오래 걸린다.



α 가 너무 작은 경우 (수렴이 늦음)



α 가 너무 큰 경우 (진동)

Gradient Boost 알고리즘

- $Loss$ 함수를 다음과 같이 정의

$$L(y_i, f(x_i)) = \frac{1}{2}(y_i - f(x_i))^2$$

- 이때의 $Loss$ 함수의 기울기(gradient)

$$\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} = \frac{\partial [\frac{1}{2}(y_i - f(x_i))^2]}{\partial f(x_i)} = f(x_i) - y_i$$

- Negative gradient = Residual

$$-(f(x_i) - y_i) = y_i - f(x_i)$$

- Negative gradient(residual)를 최소화 시키면서 학습 시키기때문에 gradient boosting이라 부름

일반적으로 손실함수로 제곱오차를 사용하지만
절대오차등을 사용할 수도 있다.

Gradient Boost 알고리즘 예시

모든 타겟 값의 평균을 추정 값이라고 하면 $(88 + 76 + 56 + 73 + 77 + 57) / 6 = 71.2$

이 값으로 예측한 값과 실제 값의 차이 (error)를 반영한 새로운 트리를 만듦

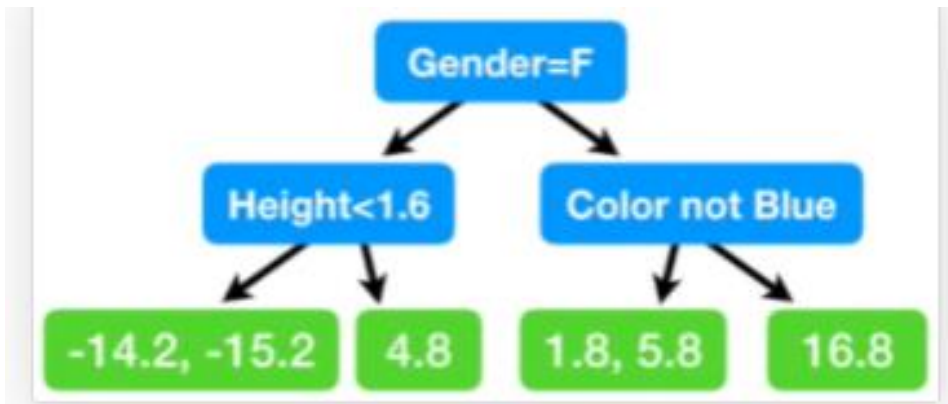
Height (m)	Favorite Color	Gender	Weight (kg)
1.6	Blue	Male	88
1.6	Green	Female	76
1.5	Blue	Female	56
1.8	Red	Male	73
1.5	Green	Male	77
1.4	Blue	Female	57

부스팅

Gradient Boost 알고리즘

Gradient Boost 알고리즘 예시

트리를 만들어보면 아래의 그림과 같다.



Height (m)	Favorite Color	Gender	Weight (kg)	Residual
1.6	Blue	Male	88	16.8
1.6	Green	Female	76	4.8
1.5	Blue	Female	56	-15.2
1.8	Red	Male	73	1.8
1.5	Green	Male	77	5.8
1.4	Blue	Female	57	-14.2

Gradient Boost 알고리즘 예시

두 개 이상의 값이 있으면 평균값으로
채워준다.



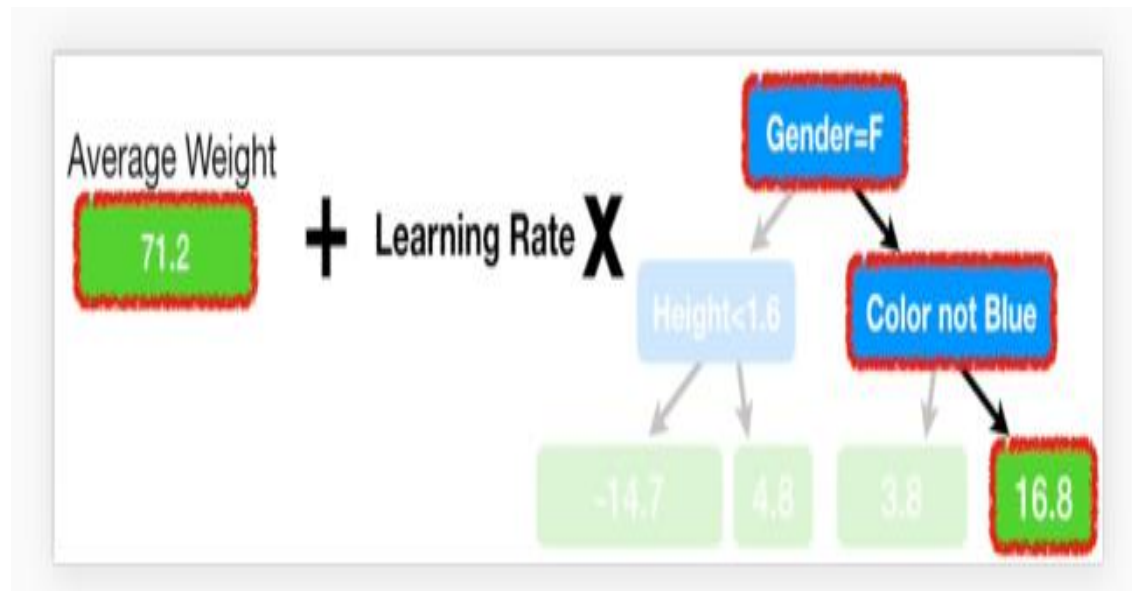
부스팅

Gradient Boost 알고리즘

Gradient Boost 알고리즘 예시

원래의 값에 트리의 예측값을 더해 새로운 예측값을 찾아본다. 이 때, 트리의 예측값에 학습률을 곱해서 더해준다.

학습률이 0.1이면 새로운 예측값은 $71.2 + (0.1 * 16.8) = 72.9$ 가 된다.



부스팅

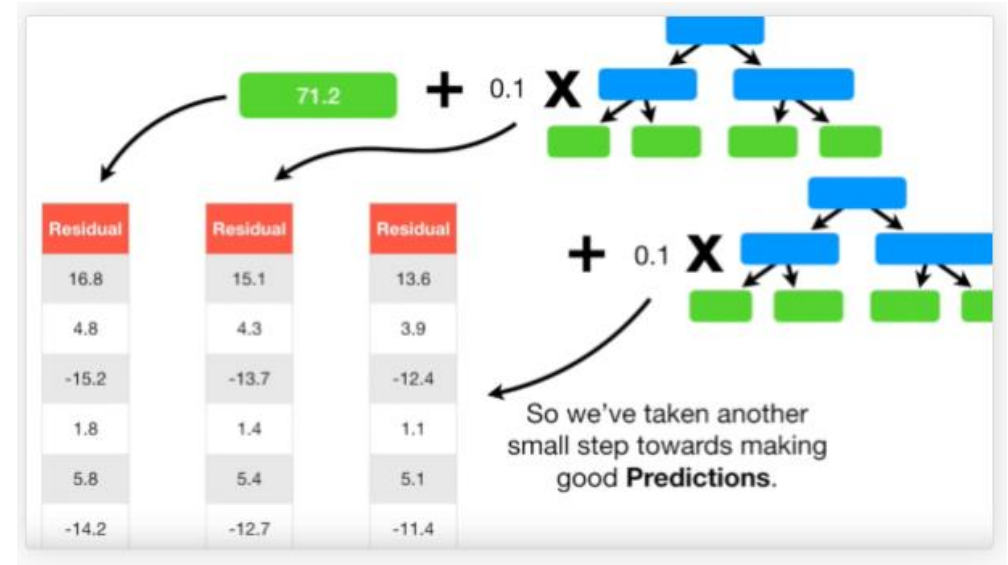
Gradient Boost 알고리즘

Gradient Boost 알고리즘 예시

새로운 예측 값들을 통해서 다시 잔차들을 구해보면 다음과 같다.

새로운 예측값들로부터 나온 잔차들을 가지고 트리를 만들고, 원래 값에 트리의 예측값에 학습률을 곱해 더 해주는 과정을 반복한다.

Height (m)	Favorite Color	Gender	Weight (kg)	Residual
1.6	Blue	Male	88	15.1
1.6	Green	Female	76	4.3
1.5	Blue	Female	56	-13.7
1.8	Red	Male	73	1.4
1.5	Green	Male	77	5.4
1.4	Blue	Female	57	-12.7

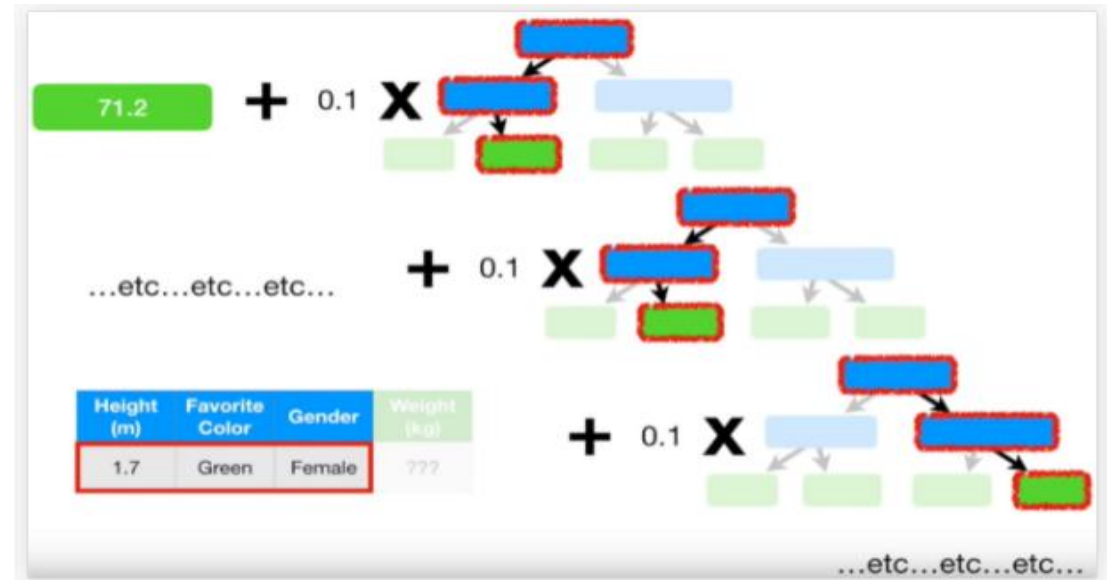


부스팅

Gradient Boost 알고리즘

Gradient Boost 알고리즘 예시

이 과정을 계속하다보면 잔차는 계속 줄어들게 된다.



부스팅

GBM의 하이퍼파라미터

GBM의 하이퍼파라미터		
번호	이름	설명
1	max_depth	트리의 최대 깊이 너무 크면 과적합이 발생(보통 3~10정도로 설정) 범위 0~ Inf Default=3
2	min_samples_split	노드를 분할하기 위한 최소한의 샘플 데이터 수 값이 작을수록 과적합 가능성 증가 Default=2
3	min_samples_leaf	리프노드가 되기 위한 최소한의 샘플 데이터 수 값이 작을수록 과적합 가능성 증가 Default=1
4	max_leaf_nodes	리프노드의 최대 개수 Default = None(제한 없음)

부스팅

GBM의 하이퍼파라미터

GBM의 하이퍼파라미터		
번호	이름	설명
5	loss	경사하강법에서 용할 손실함수를 지정 Default: deviance
6	n_estimators	생성할 트리의 개수를 지정 값이 커지면 성능은 좋아질 수도 있지만 시간이 오래 걸림 Default=100
7	learning_rate	학습률(0~1) Default=0.1
8	subsample	개별트리가 학습에 사용하는 데이터 샘플링 비율(0~1) Default=1(전체 데이터)

Gradient Boosting의 장점과 단점

장점

예측성능이 뛰어나다.

특성의 스케일을 조정하지 않아도 된다.

단점

훈련 시간이 길다.

하이퍼파라미터를 잘 정해주어야 한다.

과적합의 위험이 있다.

XGBoost란?

Extreme Gradient Boosting :

GBM의 성능을 높여서 만든 알고리즘

Gradient Boost가 병렬학습이 지원되도록 구현한

라이브러리가 XGBoost

성능과 효율이 좋아서 많이 사용된다.



XGBoost의 장점

GBM보다 빠르다 (병렬수행)

과적합규제 (강한 내구성)

예측성능이 뛰어나다

다양한 옵션을 제공한다(ex)조기중단)

결측값을 자체 처리 할 수 있다

나무 가지치기를 통해 분할 수를 더 줄여준다



부스팅

XGBoost 설치

1. 명령 프롬프트에 `pip install xgboost`를 통해서 설치할 수 있다.

```
C:\Users\ksslk>pip install xgboost
Collecting xgboost
  Downloading xgboost-1.5.1-py3-none-win_amd64.whl (106.6 MB)
    | 106.6 MB 2.2 MB/s
Requirement already satisfied: scipy in c:\users\ksslk\anaconda3\lib\site-packages (from xgboost) (1.7.1)
Requirement already satisfied: numpy in c:\users\ksslk\anaconda3\lib\site-packages (from xgboost) (1.21.2)
WARNING: Error parsing requirements for python-language-server: [Errno 2] No such file or directory: 'c:\users\ksslk\anaconda3\lib\site-packages\python_language_server-0.35.1.dist-info\METADATA'
WARNING: Error parsing requirements for python-jsonrpc-server: [Errno 2] No such file or directory: 'c:\users\ksslk\anaconda3\lib\site-packages\python_jsonrpc_server-0.4.0.dist-info\METADATA'
WARNING: Error parsing requirements for pathtools: [Errno 2] No such file or directory: 'c:\users\ksslk\anaconda3\lib\site-packages\pathtools-0.1.2.dist-info\METADATA'
Installing collected packages: xgboost
Successfully installed xgboost-1.5.1
```

2. 만약 위의 코드를 통해서도 잘 설치가 되지 않으면, 아나콘다 프롬프트에서 아래의 코드를 실행하도 된다.

```
conda install -c anaconda py-xgboost
```

General 파라미터

전체 기능을 가이드 한다.
Default 값을 바꾸는 일이 거의 없다.

General Parameter		
번호	이름	설명
1	booster trree	트리기반 모델(gbtree) 와 선형모델 (gblienear)중 선택 Default = 'gbtree'
2	nthread	CPU 실행 스레드 개수 조정 Default는 전체를 다 사용하는 것

Boost Parameter

트리 최적화, 부스팅, 정규화 등과 관련
된 파라미터들
이들을 잘 튜닝하는 것이 중요

Boost parameter		
번호	이름	설명
1	learning rate	학습률(0~1) Default=0.1
2	n_estimators	생성할 weak learner의 수 Default=100
3	min_child_weight	GBM의 min_samples_leaf와 유사 과적합 조절 용도 범위 0 ~ Inf Default=1
4	gamma	트리의 리프 노드를 추가적으로 나눌지 결정할 최소 손실 감소값 해당 값보다 큰 손실이 감소된 경우에 리프 노드 분리 값이 클수록 과적합 방지 Default=0

Boost Parameter

Boost parameter		
번호	이름	설명
5	early_stopping_rounds	조기 종단을 위한 반복 횟수, N번 반복하는 동안 성능 평가 지표가 향상되지 않으면 반복을 멈춤. Default=None
6	max_depth	너무 크면 과적합이 발생(보통 3~10정도로 설정) 0으로 지정하면 깊이의 제한 x 범위 0~ Inf Default=3
7	subsample	GBM의 subsample과 동일 과적합을 조절 용도 범위 0~1 일반적으로 0.5~1로 설정 Default=1
8	colsample_bytree	GBM의 max_features와 유사 트리 생성에 필요한 피처의 샘플링에 사용 피처가 많을때 과적합 조절에 사용 범위: 0~1 Default=1

Boost Parameter

Boost parameter		
번호	이름	설명
9	reg_lambda	L2 정규화 적용값 Default=1
10	reg_alpha	L1 정규화 적용값 Default=0
11	scale_pos_weight	불균형 데이터 셋의 균형을 유지 Default=1

Learning Task 파라미터

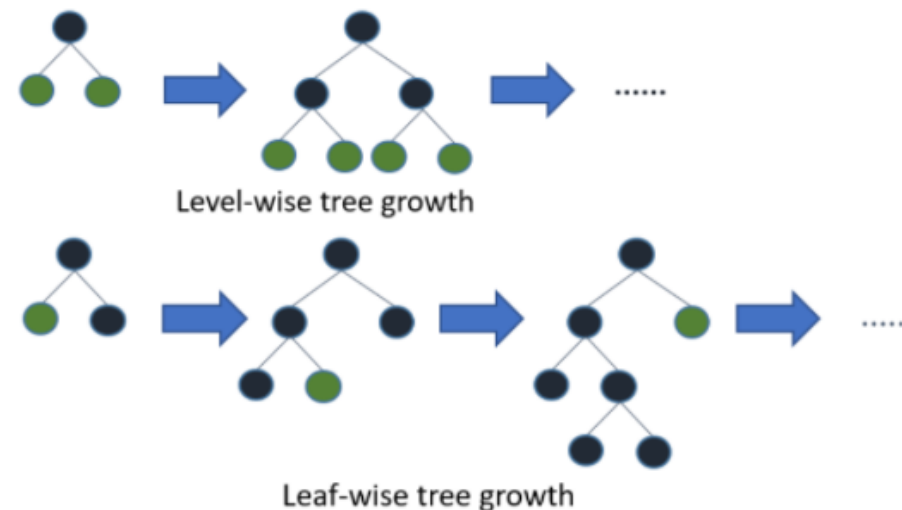
학습 수행 시의 객체함수,
평가를 위한 지표 등을 설정

Learning Task Parameter		
번호	이름	설명
1	Objective	목적함수 종류 'reg:linear': 회귀 binary:logistic: 이진분류 multi:softmax: 다중분류, 클래스 반환 multi:softprob: 다중분류, 확률반환 Default='reg:linear'
2	eval_metric	평가지표 rmse, mae, logloss, error, auc, ... Default: 회귀 → rmse, 분류 → error
3	eval_set	성능 평가에 사용하는 데이터셋 Default=None

Light GBM이란?

GBM의 성능을 높인 XGBoost역시도 시간이 오래 걸림

LightGBM은 기존의 GBM계열의 트리가 사용하는
'균형 트리 분할(level-wise)방식'이 아닌 '리프 중심
트리 분할(leaf-wise)방식'을 사용해 시간과 메모리를
절약



Light GBM 하이퍼파라미터

Light GBM을 설치하는 방법은 XGBoost와 동일

Light GBM은 적은 데이터에 대해 과적합이
발생하기 쉬우므로 하이퍼파라미터를 잘
조절해 주어야한다.

파라미터명 (파이썬 래퍼)	파라미터명 (사이킷런 래퍼)	설명
num_iterations (100)	n_estimators (100)	- 반복 수행 트리개수 지정 - 너무 크면 과적합 발생
learning_rate (0.1)	learning_rate (0.1)	- 학습률
max_depth (-1)	max_depth (-1)	- 최대 깊이 - default → 깊이에 제한이 없음
min_data_in_leaf (20)	min_child_samples (20)	- 최종 리프 노드가 되기 위한 레코드수 - 과적합 제어용
num_leaves (31)	num_leaves (31)	- 하나의 트리가 가지는 최대 리프 개수
boosting (‘gbdt’)	boosting_type (‘gbdt’)	- gbdt : 일반적인 그래디언트부스팅 트리 - rf : 랜덤포레스트
bagging_fraction (1.0)	subsample (1.0)	- 데이터 샘플링 비율 - 과적합 제어용
feature_fraction (1.0)	colsample_bytree (1.0)	- 개별트리 학습시 선택되는 피쳐 비율 - 과적합 제어용
lambda_l2 (0)	reg_lambda (0)	- L2 Regularization 적용 값 - 피쳐 개수가 많을 때 적용을 검토 - 클수록 과적합 감소 효과
lambda_l1 (0)	reg_alpha (0)	- L1 Regularization 적용 값 - 피쳐 개수가 많을 때 적용을 검토 - 클수록 과적합 감소 효과
objective	objective	- ‘reg:linear’ : 회귀 - binary:logistic : 이진분류 - multi:softmax : 다중분류, 클래스 반환 - multi:softprob : 다중분류, 확률반환

데이터 가져오기

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

```
url = 'https://raw.githubusercontent.com/hmkim312/datas/main/HAR/features.txt'
```

```
feature_name_df = pd.read_csv(url, sep = '\s+', header = None, names = ['column_index', 'column_name'])
```

```
feature_name = feature_name_df.iloc[:,1].values.tolist()
```

```
X_train = pd.read_csv('https://raw.githubusercontent.com/hmkim312/datas/main/HAR/X_train.txt', sep = '\s+', header = None)
```

```
X_test = pd.read_csv('https://raw.githubusercontent.com/hmkim312/datas/main/HAR/X_test.txt', sep = '\s+', header = None)
```

```
y_train = pd.read_csv('https://raw.githubusercontent.com/hmkim312/datas/main/HAR/y_train.txt', sep = '\s+', header = None, names = ['action'])
```

```
y_test = pd.read_csv('https://raw.githubusercontent.com/hmkim312/datas/main/HAR/y_test.txt', sep = '\s+', header = None, names = ['action'])
```

```
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
X_train.columns = feature_name
```

```
X_test.columns = feature_name
```

```
X_train.head()
```

Out [1]:

	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	...	fBodyBodyGyroJerkMag-meanFreq()	fBodyBodyGyroJerkMag-meanFreq()
0	0.288585	-0.020294	-0.132905	-0.995279	-0.983111	-0.913526	-0.995112	-0.983185	-0.923527	-0.934724	...	-0.074323	-0.074323
1	0.278419	-0.016411	-0.123520	-0.998245	-0.975300	-0.960322	-0.998807	-0.974914	-0.957686	-0.943068	...	0.158075	0.158075
2	0.279653	-0.019467	-0.113462	-0.995380	-0.967187	-0.978944	-0.996520	-0.963668	-0.977469	-0.938692	...	0.414503	0.414503
3	0.279174	-0.026201	-0.123283	-0.996091	-0.983403	-0.990675	-0.997099	-0.982750	-0.989302	-0.938692	...	0.404573	0.404573
4	0.276629	-0.016570	-0.115362	-0.998139	-0.980817	-0.990482	-0.998321	-0.979672	-0.990441	-0.942469	...	0.087753	0.087753

5 rows × 561 columns

Adaboost

```
In [2]: from sklearn.ensemble import AdaBoostClassifier
        from sklearn.metrics import accuracy_score
        import time
        import warnings

        warnings.filterwarnings('ignore')
        start_time = time.time()

        clf = AdaBoostClassifier(random_state=10)
        clf.fit(X_train, y_train)
        pred = clf.predict(X_test)

        print('ACC : ', accuracy_score(y_test, pred))
        print('AdaBoost 수행 시간: {:.1f}초'.format(time.time() - start_time))
```

```
ACC :  0.5310485239226331
AdaBoost 수행 시간: 28.9초
```

Adaboost-깊이 수정

```
In [3]: from sklearn.ensemble import AdaBoostClassifier
        from sklearn.tree import DecisionTreeClassifier
        from sklearn.metrics import accuracy_score
        import time
        import warnings

        warnings.filterwarnings('ignore')
        start_time = time.time()

        tree_model = DecisionTreeClassifier(max_depth=20)
        clf = AdaBoostClassifier(base_estimator = tree_model, n_estimators=100, random_state=0)
        clf.fit(X_train, y_train)
        pred = clf.predict(X_test)

        print('ACC : ', accuracy_score(y_test, pred))
        print('hyper parameter 수정 후 AdaBoost 수행 시간: {:.1f}초'.format(time.time() - start_time))

ACC :  0.8568035290125552
hyper parameter 수정 후 AdaBoost 수행 시간: 4.9초
```

GBM

```
In [4]: from sklearn.ensemble import GradientBoostingClassifier
        from sklearn.metrics import accuracy_score
        import time
        import warnings

        warnings.filterwarnings('ignore')

        start_time = time.time()
        gb_clf = GradientBoostingClassifier(random_state=13)
        gb_clf.fit(X_train, y_train)
        gb_pred = gb_clf.predict(X_test)

        print('ACC : ', accuracy_score(y_test, gb_pred))
        print('GBM 수행 시간: {:.1f}초'.format(time.time() - start_time))
```

```
ACC : 0.9385816084153377
GBM 수행 시간: 756.9초
```

GBM-grid search

#시간이 많이 걸리므로 생략하셔도 됩니다!

```
In [5]: from sklearn.model_selection import GridSearchCV

        params = {
            'n_estimators': [100, 500],
            'learning_rate': [0.05, 0.1]
        }

        start_time = time.time()
        grid = GridSearchCV(gb_clf, param_grid=params, cv=2, verbose=1, n_jobs=-1)
        grid.fit(X_train, y_train)
        print('Fit time : ', time.time()-start_time)
```

```
Fitting 2 folds for each of 4 candidates, totalling 8 fits
Fit time : 6736.362127065659
```

XGBoost

grid.best_params를 learning_rate=0.01, n_estimators: 100로 바꾸시면 됩니다!

```
In [15]: from xgboost import XGBClassifier
```

```
start_time = time.time()
xgb = XGBClassifier(grid.best_params_, max_depth=3)
xgb.fit(X_train.values, y_train)
print('Acc : ', accuracy_score(y_test, xgb.predict(X_test.values)))
print('XGBoost 수행 시간: {:.1f}초'.format(time.time() - start_time))
```

```
[17:10:02] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.5.1/src/learner.cc:1115: Starting in XGBoost 1.3.0, the default evaluation metric used with the objective 'multi:softprob' was changed from 'merror' to 'mlogloss'. Explicitly set eval_metric if you'd like to restore the old behavior.
Acc : 0.9494401085850017
XGBoost 수행 시간: 48.5초
```


XGBoost-early stopping추가

```
In [19]: from xgboost import XGBClassifier
```

```
evals = [(X_test.values, y_test)]
```

```
start_time = time.time()
```

```
xgb = XGBClassifier(grid.best_params_, max_depth = 3)
```

```
xgb.fit(X_train.values, y_train, early_stopping_rounds = 5, eval_set=evals)
```

```
print('Acc : ', accuracy_score(y_test, xgb.predict(X_test.values)))
```

```
print('XGBoost 수행 시간: {:.1f}초'.format(time.time() - start_time))
```

```
[17:21:16] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.5.1/src/learner.cc:1115: Starting in XGBoost 1.3.0, the default evaluation metric used with the objective 'multi:softprob' was changed from 'merror' to 'mlogloss'. Explicitly set eval_metric if you'd like to restore the old behavior.
```

```
[0]    validation_0-mlogloss:1.23408
```

```
[1]    validation_0-mlogloss:0.95022
```

```
[49]   validation_0-mlogloss:0.75956
```

```
[50]   validation_0-mlogloss:0.13196
```

```
[51]   validation_0-mlogloss:0.13097
```

```
[52]   validation_0-mlogloss:0.13088
```

```
[53]   validation_0-mlogloss:0.13078
```

```
[54]   validation_0-mlogloss:0.13096
```

```
[55]   validation_0-mlogloss:0.13125
```

```
[56]   validation_0-mlogloss:0.13171
```

```
[57]   validation_0-mlogloss:0.13119
```

```
[58]   validation_0-mlogloss:0.13203
```

```
Acc : 0.9463861554122837
```

```
XGBoost 수행 시간: 35.7초
```

LGBM

```
In [20]: from lightgbm import LGBMClassifier

start_time = time.time()
lgbm = LGBMClassifier(n_estimator=400)
lgbm.fit(X_train.values, y_train, early_stopping_rounds=10, eval_set=evals)
print('Acc : ', accuracy_score(y_test, grid.best_estimator_.predict(X_test.values)))
print('LightGBM 수행 시간: {:.1f}초'.format(time.time() - start_time))
```

```
[1] valid_0's multi_logloss: 1.4404
[2] valid_0's multi_logloss: 1.21574
[3] valid_0's multi_logloss: 1.04795
[4] valid_0's multi_logloss: 0.913299
[5] valid_0's multi_logloss: 0.812686
[6] valid_0's multi_logloss: 0.725964
[7] valid_0's multi_logloss: 0.652995
[8] valid_0's multi_logloss: 0.591598
[9] valid_0's multi_logloss: 0.539383
[10] valid_0's multi_logloss: 0.49699
[11] valid_0's multi_logloss: 0.45999
[12] valid_0's multi_logloss: 0.42899
[13] valid_0's multi_logloss: 0.39999
[14] valid_0's multi_logloss: 0.37499
[15] valid_0's multi_logloss: 0.35199
[16] valid_0's multi_logloss: 0.33199
[17] valid_0's multi_logloss: 0.31399
[18] valid_0's multi_logloss: 0.29799
[19] valid_0's multi_logloss: 0.28399
[20] valid_0's multi_logloss: 0.27199
[21] valid_0's multi_logloss: 0.26199
[22] valid_0's multi_logloss: 0.25399
[23] valid_0's multi_logloss: 0.24699
[24] valid_0's multi_logloss: 0.24199
[25] valid_0's multi_logloss: 0.23899
[26] valid_0's multi_logloss: 0.23699
[27] valid_0's multi_logloss: 0.23599
[28] valid_0's multi_logloss: 0.23599
[29] valid_0's multi_logloss: 0.23699
[30] valid_0's multi_logloss: 0.23899
[31] valid_0's multi_logloss: 0.24199
[32] valid_0's multi_logloss: 0.24699
[33] valid_0's multi_logloss: 0.25399
[34] valid_0's multi_logloss: 0.26199
[35] valid_0's multi_logloss: 0.27199
[36] valid_0's multi_logloss: 0.28399
[37] valid_0's multi_logloss: 0.29799
[38] valid_0's multi_logloss: 0.31399
[39] valid_0's multi_logloss: 0.33199
[40] valid_0's multi_logloss: 0.35199
[41] valid_0's multi_logloss: 0.37499
[42] valid_0's multi_logloss: 0.39999
[43] valid_0's multi_logloss: 0.42899
[44] valid_0's multi_logloss: 0.45999
[45] valid_0's multi_logloss: 0.49699
[46] valid_0's multi_logloss: 0.539383
[47] valid_0's multi_logloss: 0.591598
[48] valid_0's multi_logloss: 0.652995
Acc : 0.9419748897183576
LightGBM 수행 시간: 6.8초
```

A modern home office with a wooden desk, a black desk lamp, a laptop, and framed art on a dark wall. The room features a large window on the left, a dark wall with four framed abstract art pieces, and a white desk. A large white number '5' is overlaid on the image.

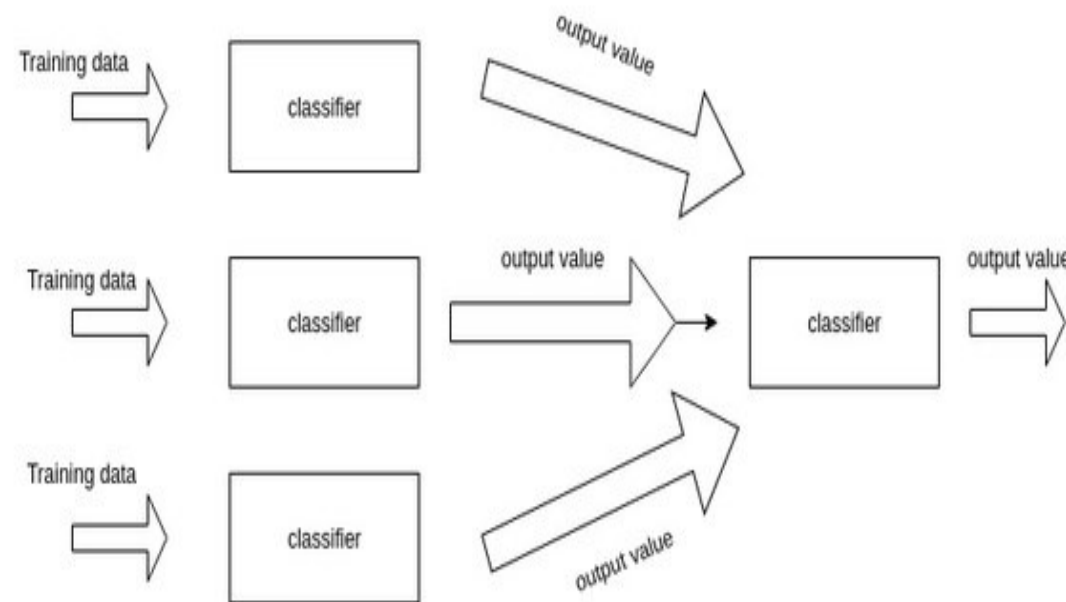
5

스택킹(Stacking)

Stacking이란?

여러 개의 개별 모델들이 예측을 진행한 후,
그 예측한 데이터를 다시 최종 모델(메타모델)의
training set으로 사용

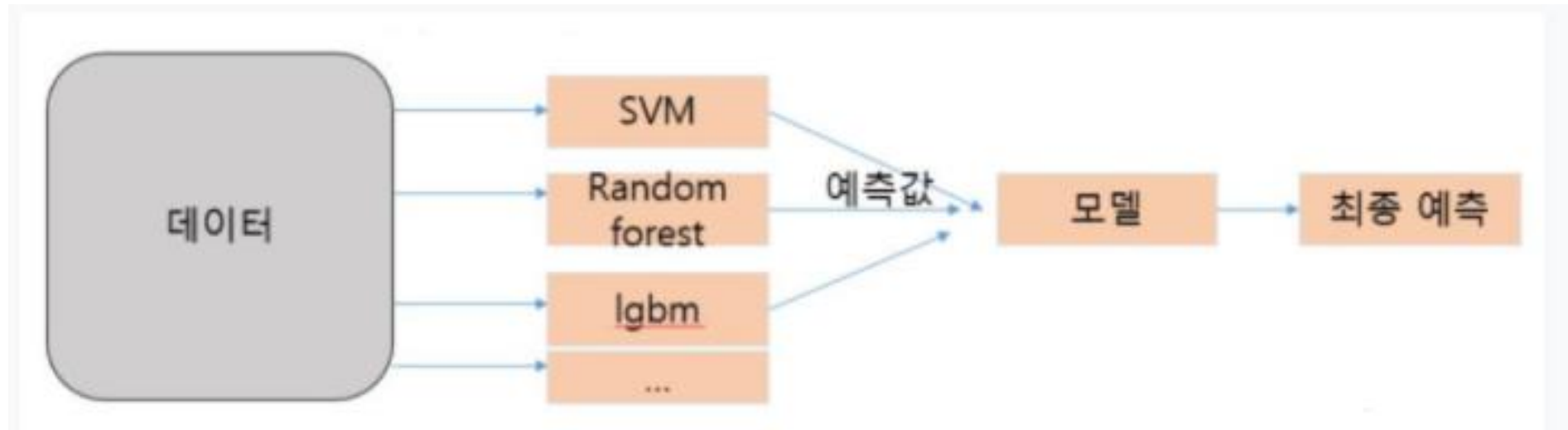
최종모델(메타모델)의 예측값이 최종 예측값이 됨.



스택킹

Stacking 앙상블

Stacking 예시

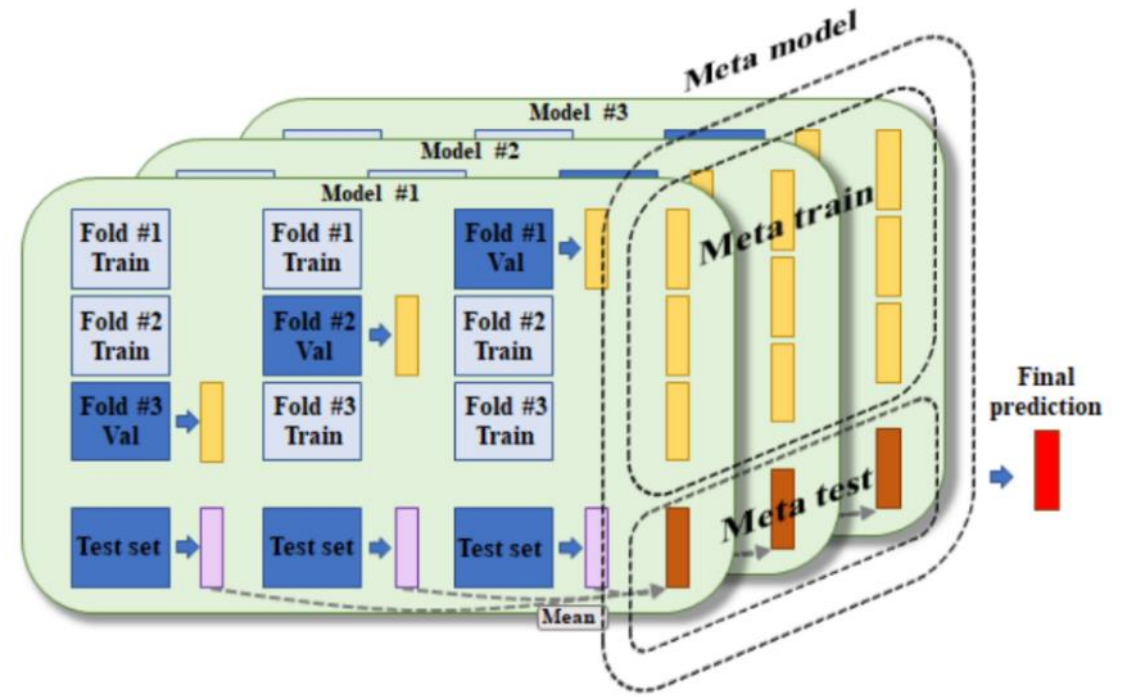


스택킹

Stacking 앙상블

CV세트 기반의 Stacking

Stacking은 과적합 문제 발생가능성이 크다
CV 세트 기반의 Stacking 앙상블이 많이
사용된다.



CV세트 기반의 Stacking의 원리

1. Train Set을 N개의 Fold로 나눈다.(5개로 나눴다고 가정)
2. 4개를 학습을 위한 데이터 폴드로 사용해 개별모델을 학습,
1개를 검증을 위한 데이터 폴드로 사용해 데이터를 예측 후 결과를 저장한다.
3. 2의 과정을 5번 반복한 후 예측 값들의 평균으로 최종 결과 값을 생성한다.
4. 3에서 구한 결과값을 학습데이터로 사용해 메타 모델을 학습시키고, 예측을 진행한다.