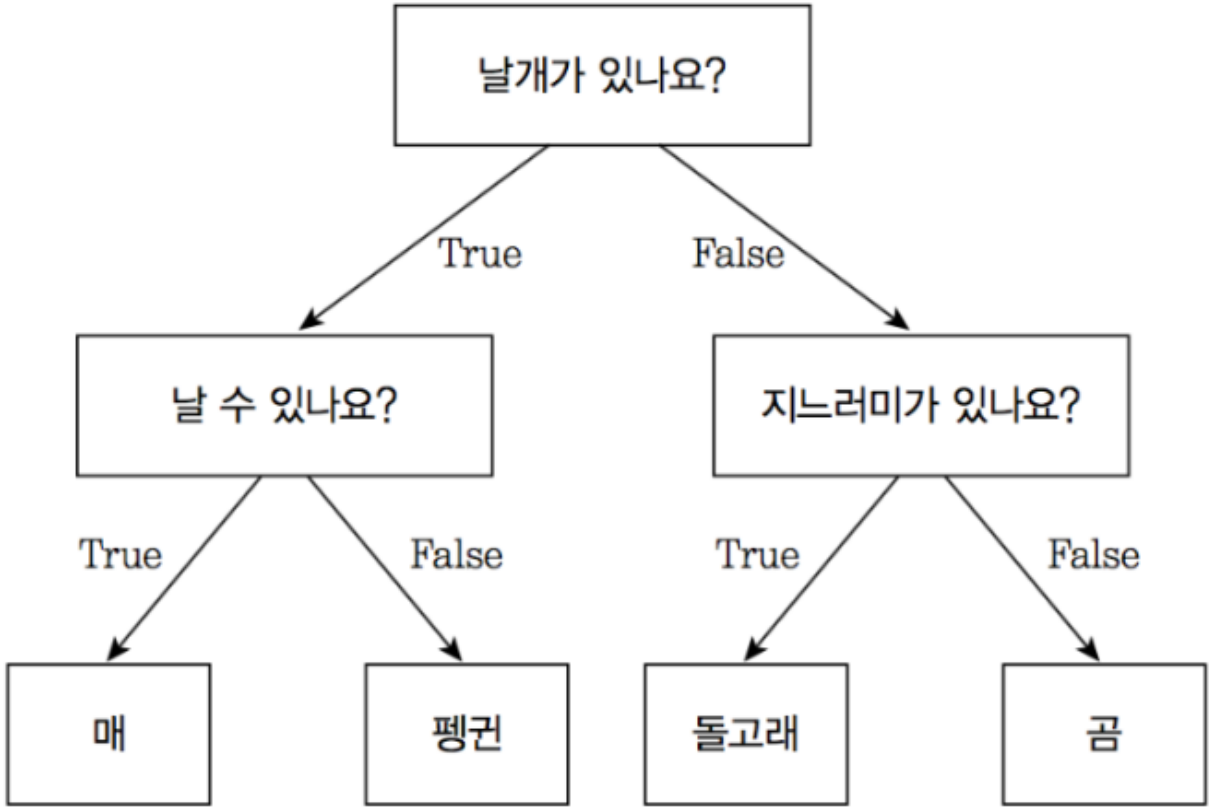
# Decision Tree

# Decision Tree

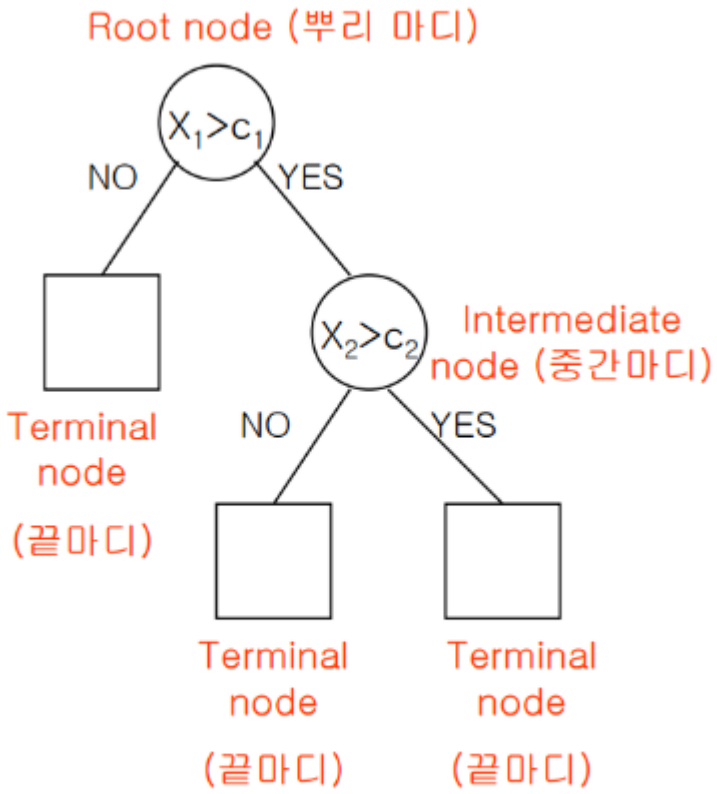



결정트리(의사결정나무, 의사결정트리)는 특정기준(질문)에 따라서 데이터를 구분하는 모델이며 분류와 회귀 문제에서 가장 널리 사용되는 모델이다.

날개가 있나요? /
날수 있나요? /
지느러미가 있나요? /
를 사용해서 4개의 클래스를 구분하는 모델을 만들었다.

출처 : 텐서플로우 블로그

# Decision Tree



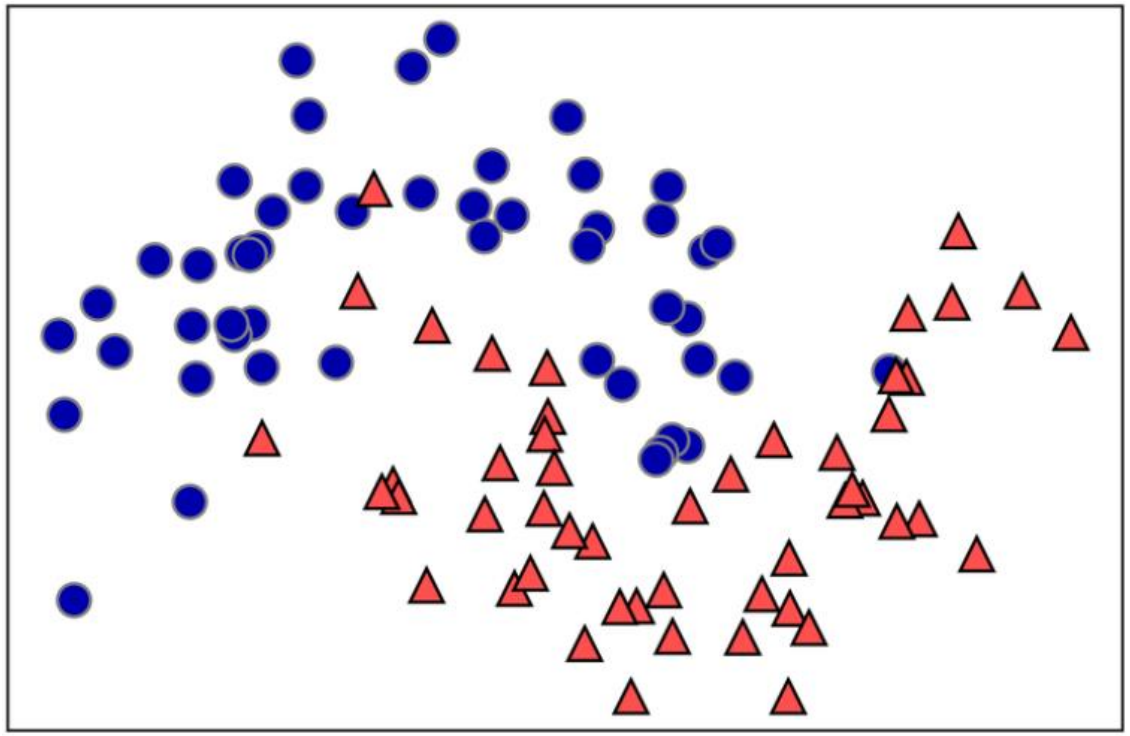

출처: ratsgo's blog

결정트리 구조

결정트리에서 질문이나 네모상자를 노드(Node) 라고 한다.
맨 위의 노드(첫질문)를 Root Node라 하고
각 질문에 대한 마지막 노드를 Leaf Node라 한다.
엣지(edge)는 질문의 답과 질문을 연결하는 선을 말한다.

전체적인 모양이 나무를 뒤집에 놓은 것과 닮았다고 해서
붙여진 이름이 Decision Tree이다.

# Decision Tree

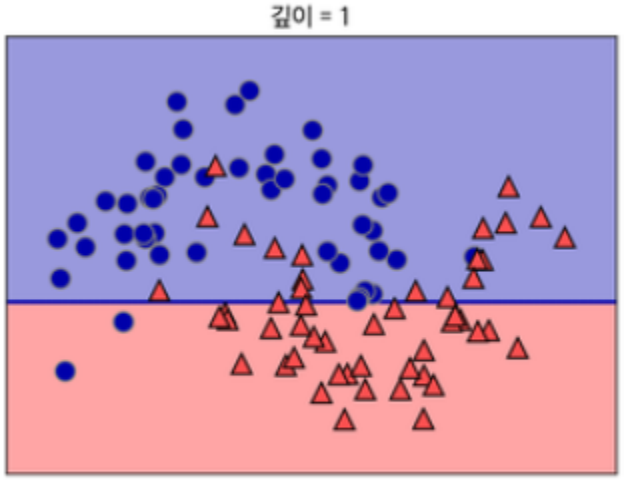


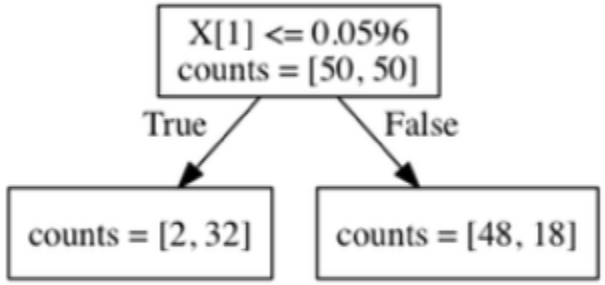
결정 트리를 적용할 반달 모양의 데이터셋

2차원 데이터 셋을 분류하는 결정 트리이다.
이 데이타 셋은 각 클래스에 데이터 포인트가 50개씩 있고 반달 두 개가 포개진 듯한 모양을 하고 있다.

앞의 동물 구분의 예처럼 예/아니오 형태의 특성으로 구분되지 않고 연속적인 데이터에 적용하는 특성으로 구분된다.
질문은 feature i는 특정한 값 a보다 큰가?
와 같은 형태를 주로 띈다.

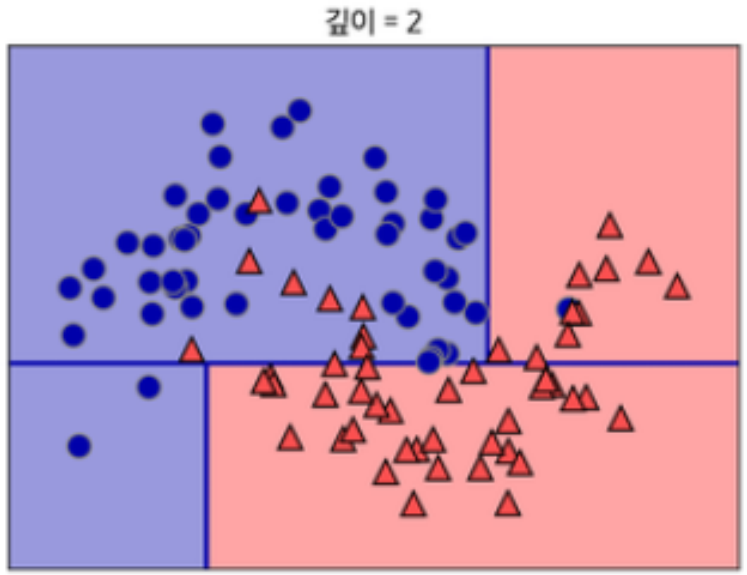
# Decision Tree

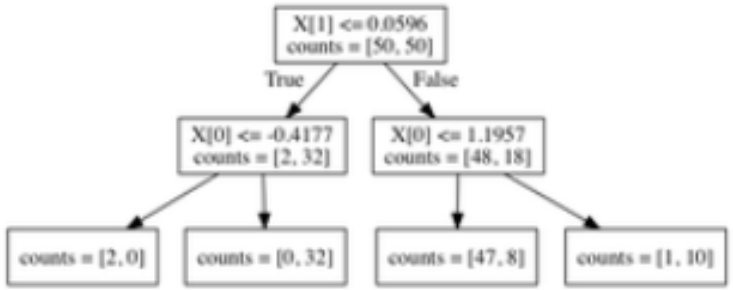



출처: 텐서 플로우 블로그

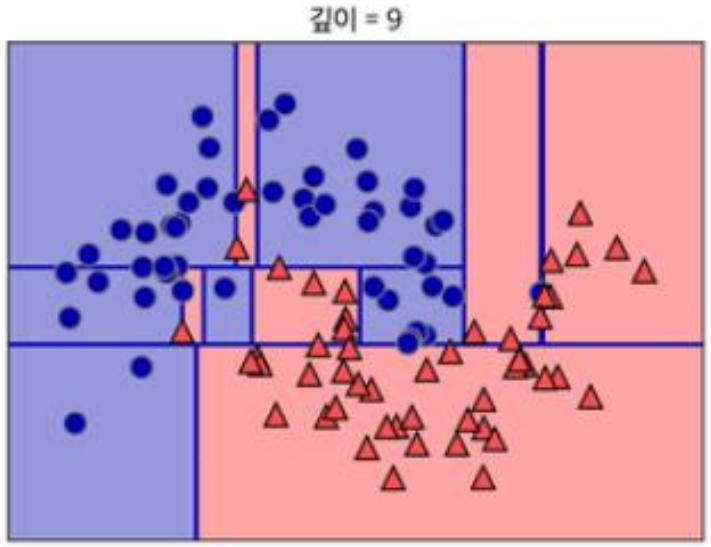
# Decision Tree




출처: 텐서 플로우 블로그

# Decision Tree

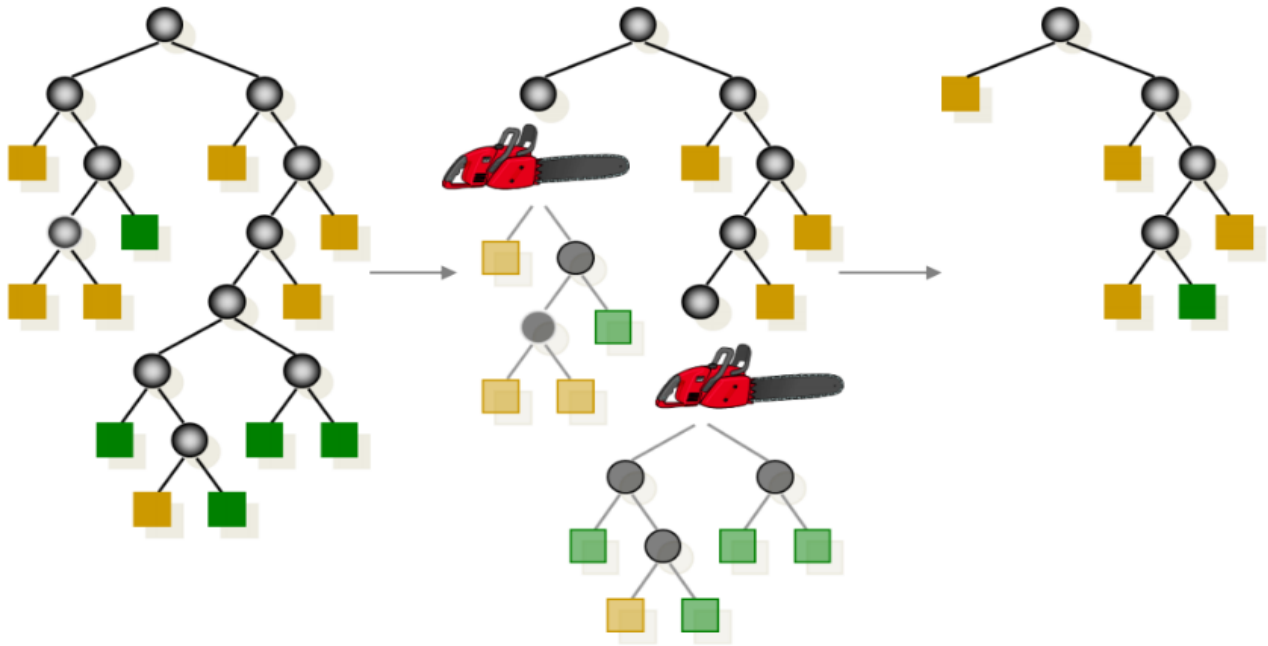



출처: 텐서 플로우 블로그

# Decision Tree

## 가지치기(pruning)

가지치기란 최대트리로 형성된 결정트리의 특정 노드 밑의 하부 트리를 제거하여 일반화 성능을 높히는 것을 의미한다.



출처: https://ratsgo.github.io/machine learning/2017/03/26/tree/

# Decision Tree

## 가지치기(Pruning)

오버핏팅을 막기 위한 방법으로 가지치기를 한다.
트리에 가지가 너무 많으면 오버핏팅이라 볼 수 있다.

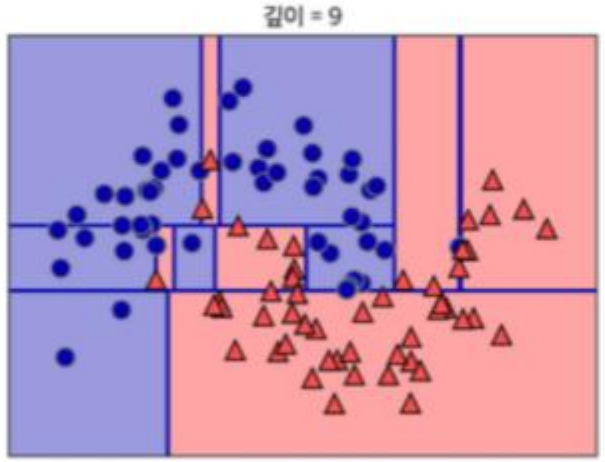
가지치기란 말 그대로 나무의 가지를 치는 작업을 말하는데
가지의 depth, leaf node의 최대갯수, 노드가 분할하는 최대 개수같은
속성을 제한하는 방법이다.

min_sample_split 파라미터를 조정하여 한 노드에 들어있는 최소
데이터 수를 정해줄 수 있다.

# Decision Tree

## 가지치기(Pruning)

- min_sample_split : 한 노드가 갖는 데이터의 개수 제한
- max_depth : tree의 깊이 제한




출처: 텐서 플로우 블로그

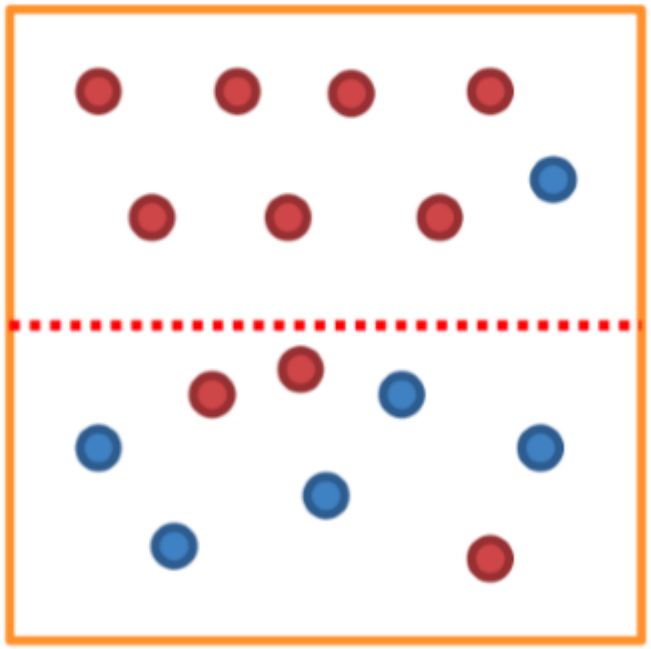
# Decision Tree

## 알고리즘: 엔트로피(Entropy), 불순도(Impurity)

- 불순도(Impurity)란 해당 범주 안에 서로 다른 데이터가 얼마나 섞여 있는지를 뜻함
- 엔트로피(Entropy)는 불순도(Impurity)를 수치적으로 나타낸 척도
  - 엔트로피가 1이면 불순도가 최대

$$\text{Entropy} = -\sum_i (p_i) \log_2 (p_i)$$

- 결정 트리는 불순도를 최소화(혹은 순도를 최대화)하는 방향으로 학습을 진행



# Decision Tree

## 실습

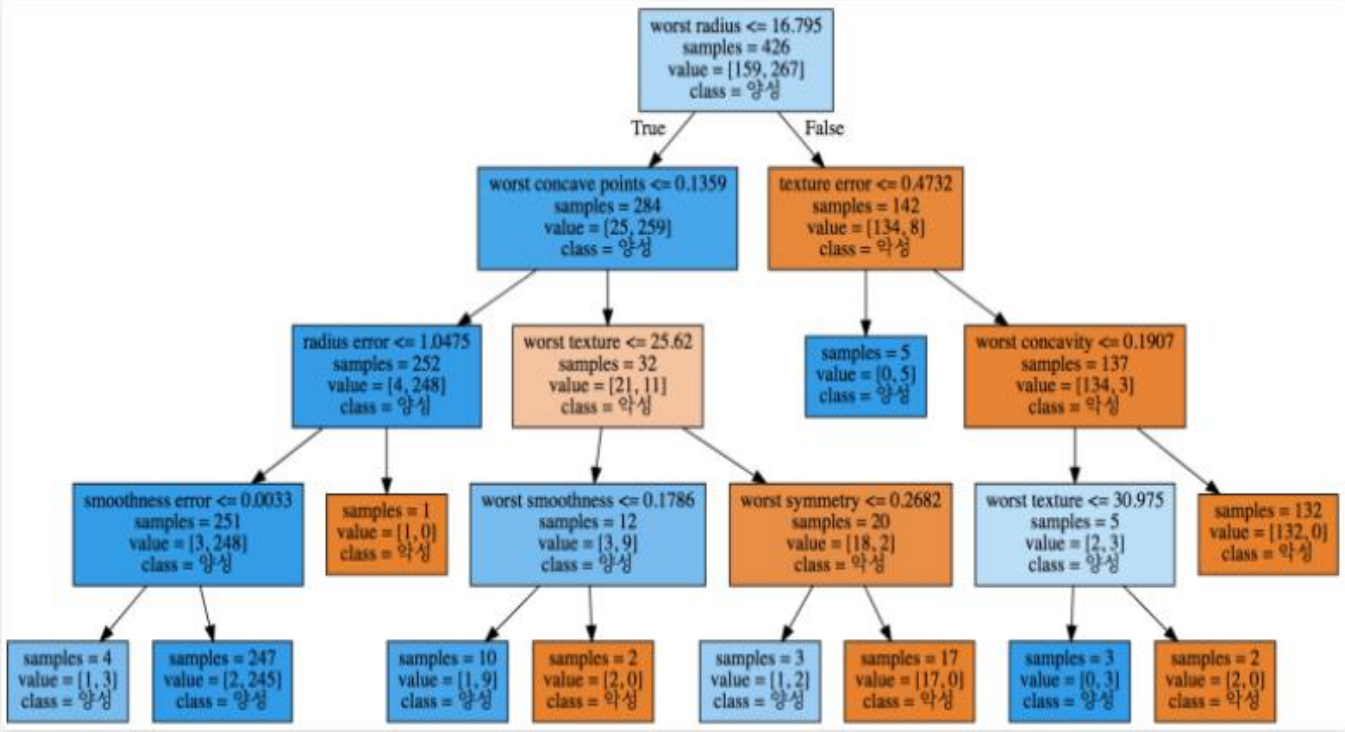
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)
print("훈련 세트 정확도: {:.3f}".format(tree.score(X_train, y_train)))
print("테스트 세트 정확도: {:.3f}".format(tree.score(X_test, y_test)))

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

max_depth 적용 후

```
tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(X_train, y_train)

print("훈련 세트 정확도: {:.3f}".format(tree.score(X_train, y_train)))
print("테스트 세트 정확도: {:.3f}".format(tree.score(X_test, y_test)))

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

# Decision Tree

```
import graphviz

with open("tree.dot") as f:
    dot_graph = f.read()
display(graphviz.Source(dot_graph))
```



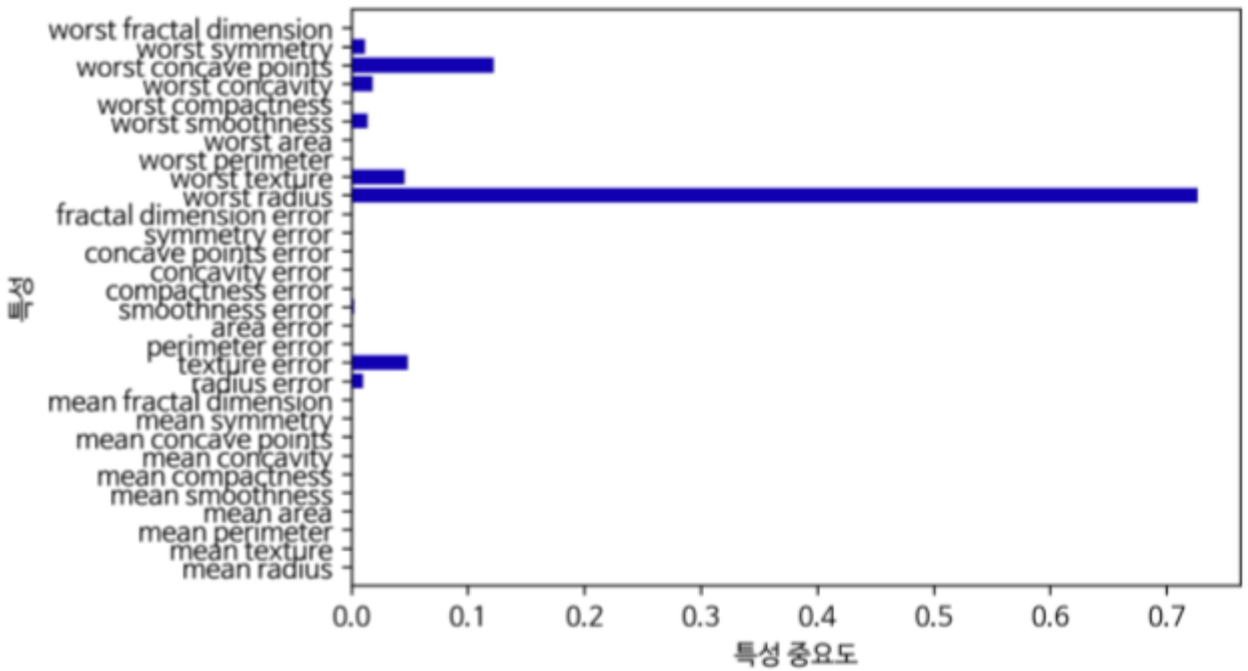

트리를 시각화하면 알고리즘의 예측이 어떻게 이뤄지는지 잘 이해할 수 있으며 비전문가에게 머신러닝 알고리즘을 설명하기에 좋습니다. 그러나 여기서 보듯이 깊이가 4만 되어도 트리는 매우 장황해집니다. 트리가 더 깊어지면(10 정도의 깊이는 보통입니다) 한눈에 보기가 힘들어집니다. 트리를 조사할 때는 많은 수의 데이터가 흐르는 경로를 찾아보면 좋습니다.

[그림 2-27]의 각 노드에 적힌 samples는 각 노드에 있는 샘플의 수를 나타내며 value는 클래스당 샘플의 수를 제공합니다. 루트 노드의 오른쪽 가지를 따라가면(worst radius > 16.795) 악성 샘플이 134개, 양성 샘플이 8개인 노드를 만듭니다. 이 방향의 트리 나머지는 이 8개의 양성 샘플을 더 세부적으로 분리합니다. 첫 노드에서 오른쪽으로 분리된 142개 샘플 중 거의 대부분(132개)이 가장 오른쪽 노드로 갑니다.

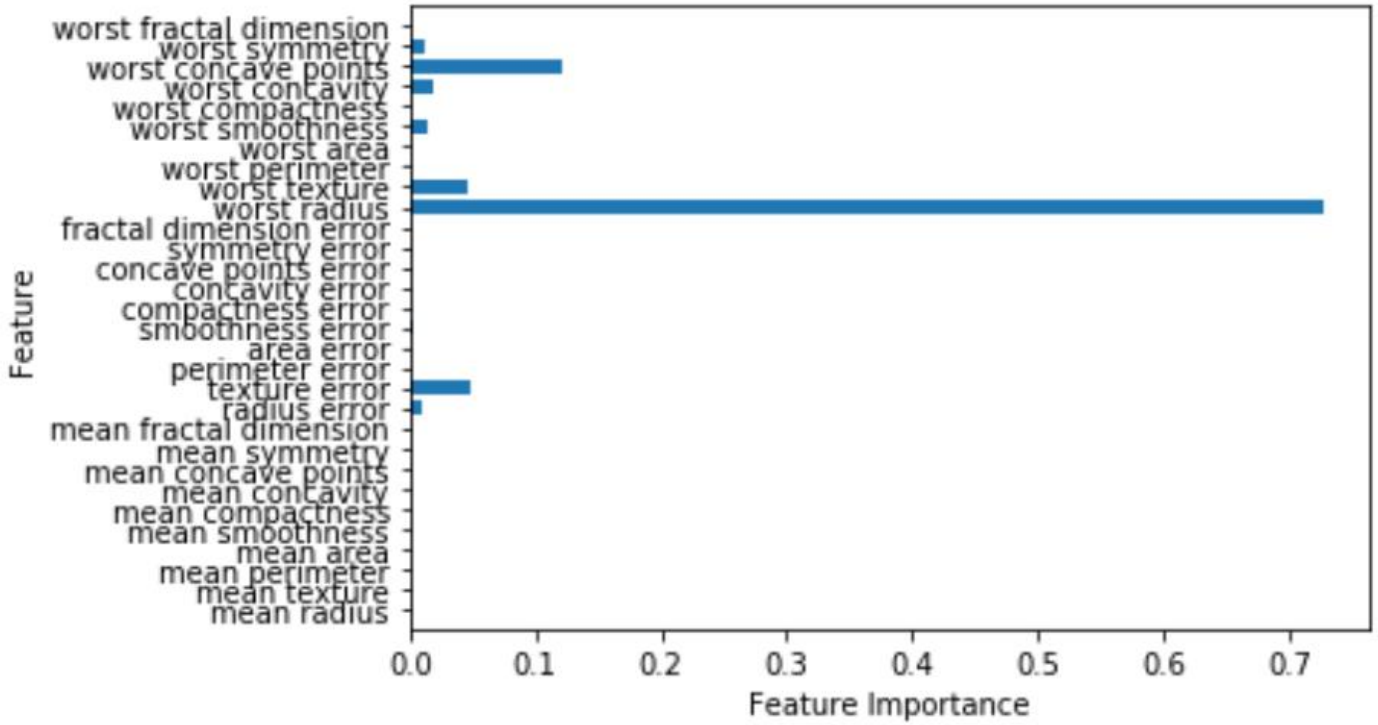
# Decision Tree

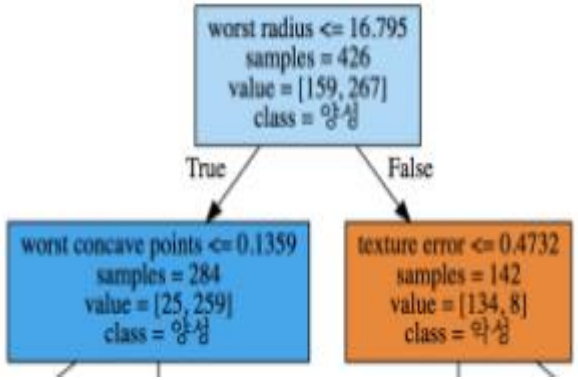
## 트리의 특성 중요도

- **특성 중요도**feature importance : 트리를 구성하는데 얼마나 중요한 속성인가를 평가 값.
- 이 값은 0과 1 사이의 값, 특성 중요도의 전체 합은 1임
- print("특성 중요도:\n{}".format(tree.feature_importances_))
- 특성 중요도: [ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.01 0.048 0. 0. 0.002 0. 0. 0. 0. 0. 0.727 0.046 0. 0. 0.014 0. 0.018 0.122 0.012 0. ]



# Decision Tree



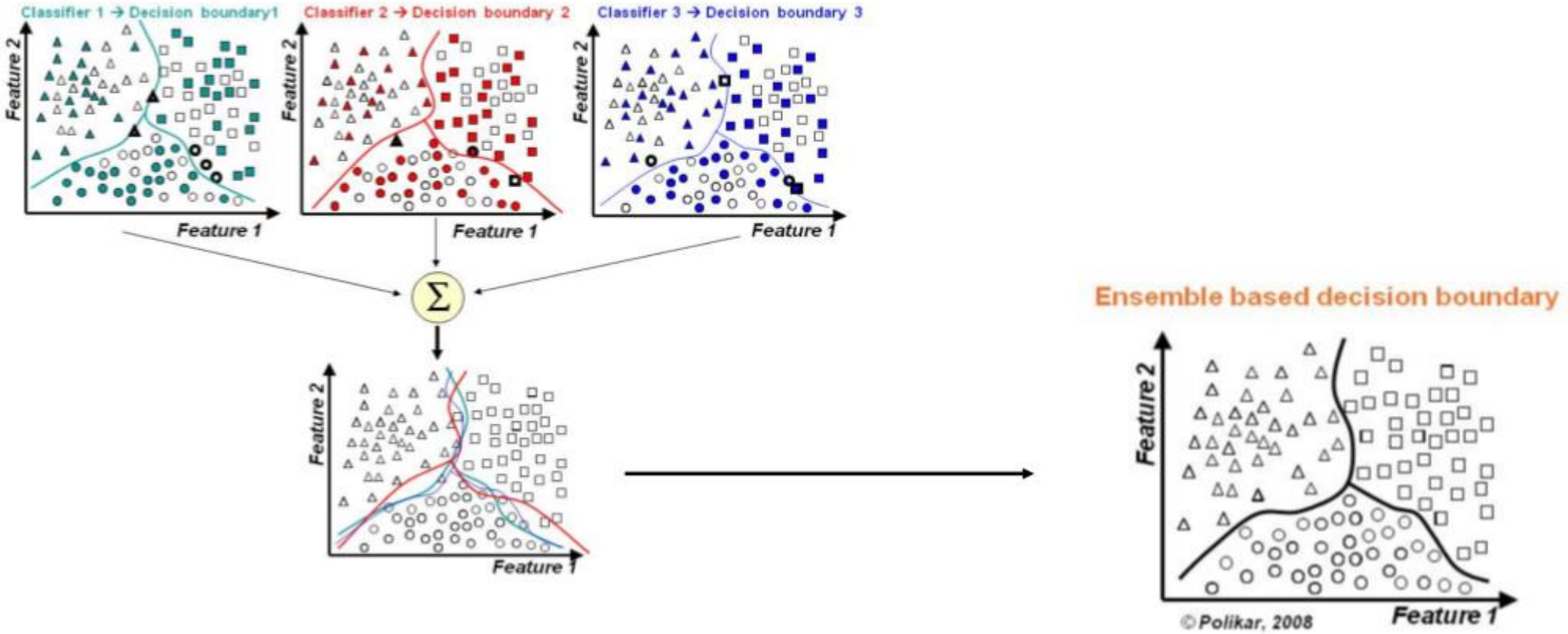




첫 번째 노드에서 사용한 특성("worst radius")이 가장 중요한 특성으로 나타납니다. 이 그래프는 첫 번째 노드에서 두 클래스를 꽤 잘 나누고 있다는 우리의 관찰을 뒷받침해줍니다 그러나 어떤 특성의 feature_importance_ 값이 낮다고 해서 이 특성이 유용하지 않다는 뜻은 아닙니다. 단지 트리가 그 특성을 선택하지 않았을 뿐이며 다른 특성이 동일한 정보를 지니고 있어서일 수 있습니다.

# Ensemble
# Random Forest

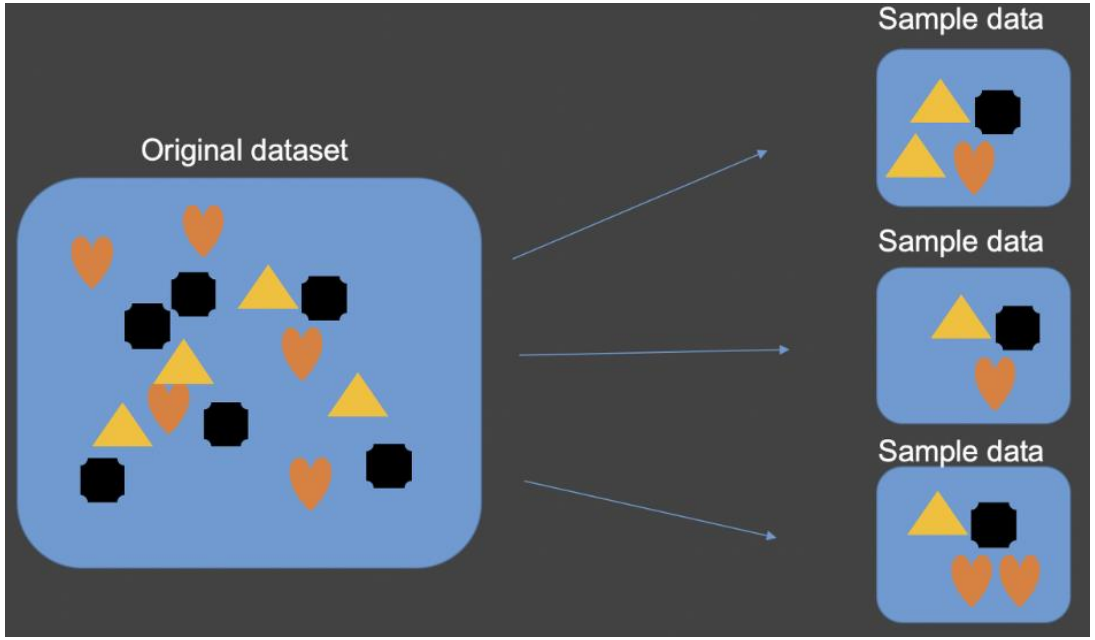
# 앙상블 알고리즘

- Bagging



# 부트스트랩 (Bootstrap Sampling)

Random Forest는 여러 개의 Decision Tree로 이뤄진 숲.
학습 데이타셋에서 무작위로 추출된 중복된 데이터를 꺼내어 각각의 DecisionTree를 생성
이때 Scikit-learn 에서 제공하는 RandomForest API는 bootstrap 원리가 적용된다.




총 3개의 Sampling Data를 만들었고
Decision Tree가 3개 생성됨

Decision Tree에서 나온 결과들을
다수결의 원칙 혹은 평균에 의해서
하나의 분류기로 구상하는 것을
Bagging이라 부른다.

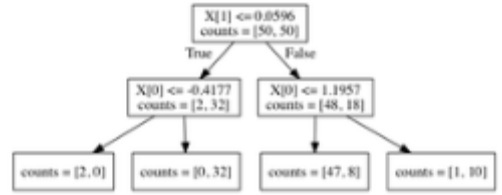
# 앙상블 알고리즘이란?

- Feature 중 랜덤으로 일부의 Feature만 선택해서 하나의 결정 트리를 만들고, 또 랜덤으로 일부의 Feature를 선택해서 또 다른 결정 트리를 만들고... 이렇게 계속 반복하여 여러 개의 결정 트리를 만들 수 있습니다. 결정 트리 하나마다 예측 값을 내놓겠죠. 여러 결정 트리들이 내린 예측 값들 중 가장 많이 나온 값을 최종 예측값으로 정합니다. 다수결의 원칙에 따르는 것입니다.

- 이렇게 의견을 통합하거나 여러 가지 결과를 합치는 방식을 앙상블(Ensemble)
  - 즉, 문제를 풀 때도 한 명의 똑똑한 사람보다 100 명의 평범한 사람이 더 잘 푸는 원리

# Random Forest

- 앙상블 기법 중의 하나임
- 결정트리 여러 개를 묶어서 다수결로 투표함




1 3 3

다수결로 투표함

예측값 3

# Random Forest

- 결정 트리의 주요 단점은 훈련 데이터에 과대적합되는 경향이 있다는 것
- 랜덤 포레스트는 이 문제를 회피할 수 있는 방법

- 랜덤 포레스트는기본적으로 여러 결정 트리의 묶음
- 각 트리는 비교적 예측을 잘 하구 있지만 일부에 과대 적합하다는 경향을 가지고 있음에 기초를 둠
- 서로 다른 방향으로 과대적합된 트리를 많이 만들면 그 결과를 평균냄으로써 과대적합된 양을 줄일 수 있음

- 이러한 전략은 구현하기 위해서는 결정 트리를 많이 만들어야 함
- 각각의 트리는 타깃 예측을 잘 해야하고 다른 트리와는 구별되어야 함

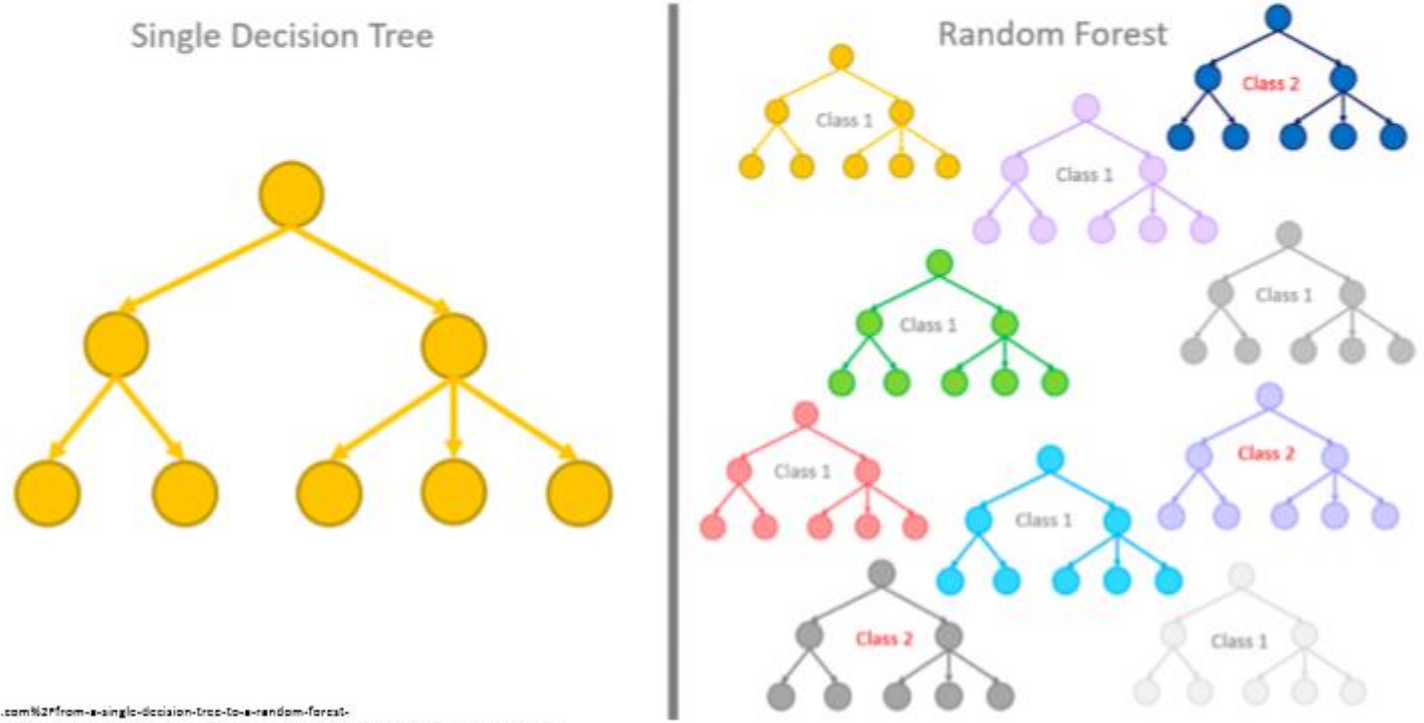
- **랜덤 포레스트에서 트리를 랜덤하게 만드는 방법은 두가지 방법**

**1. 데이터를 무작위로 선택하기**

**2. 분할 테스트에서 특성을 무작위로 선택하기**

# Random Forest

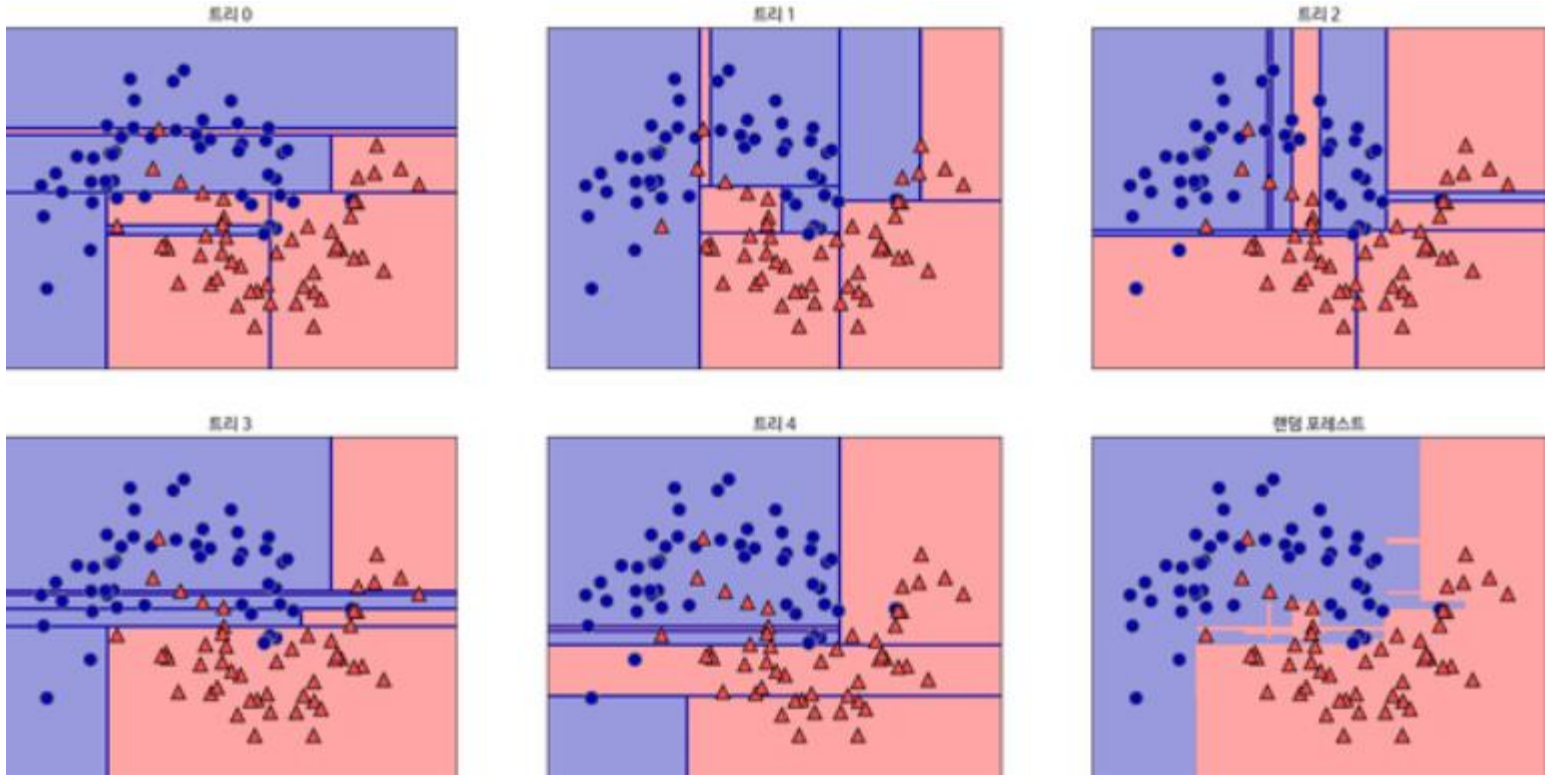
- 랜덤 포레스트의 포레스트는 숲(Forest).
- 결정 트리는 트리는 나무(Tree).
- 결정 트리(Decision Tree)가 모여 랜덤 포레스트(Random Forest)를 구성




https://www.google.com/url?sa=i&url=https%3A%2F%2Ftowardsdatascience.com%2Ffrom-a-single-decision-tree-to-a-random-forest-b9523bc65147&psig=AOvVaw2aMkTDO5ZUtcvCN20f655&ust=1589586599434000&source=images&cd=vfe&ved=0CAIQjRxqFwoTCLCgyMSZrukCFQAAAAAdAAAAABAD

# Random Forest

나무가 모여 숲을 이룹니다. 즉, 결정 트리(Decision Tree)가 모여 랜덤 포레스트(Random Forest)를 구성합니다. 결정 트리 하나만으로도 머신러닝을 할 수 있습니다. 하지만 결정 트리의 단점은 훈련 데이터에 오버피팅이 되는 경향이 있다는 것입니다. 여러 개의 결정 트리를 통해 랜덤 포레스트를 만들면 오버피팅 되는 단점을 해결할 수 있습니다.




가장 마지막 그림이 랜덤포레스트의 Decision Boundary이고, 나머지는 결정트리 각각의 Decision Boundary입니다.

결정트리 경계는 모호하고 overfitting되어있는 반면에 결정트리경계를 평균 내어서 만든 랜덤포레스트의 경계는 보다 깔끔합니다

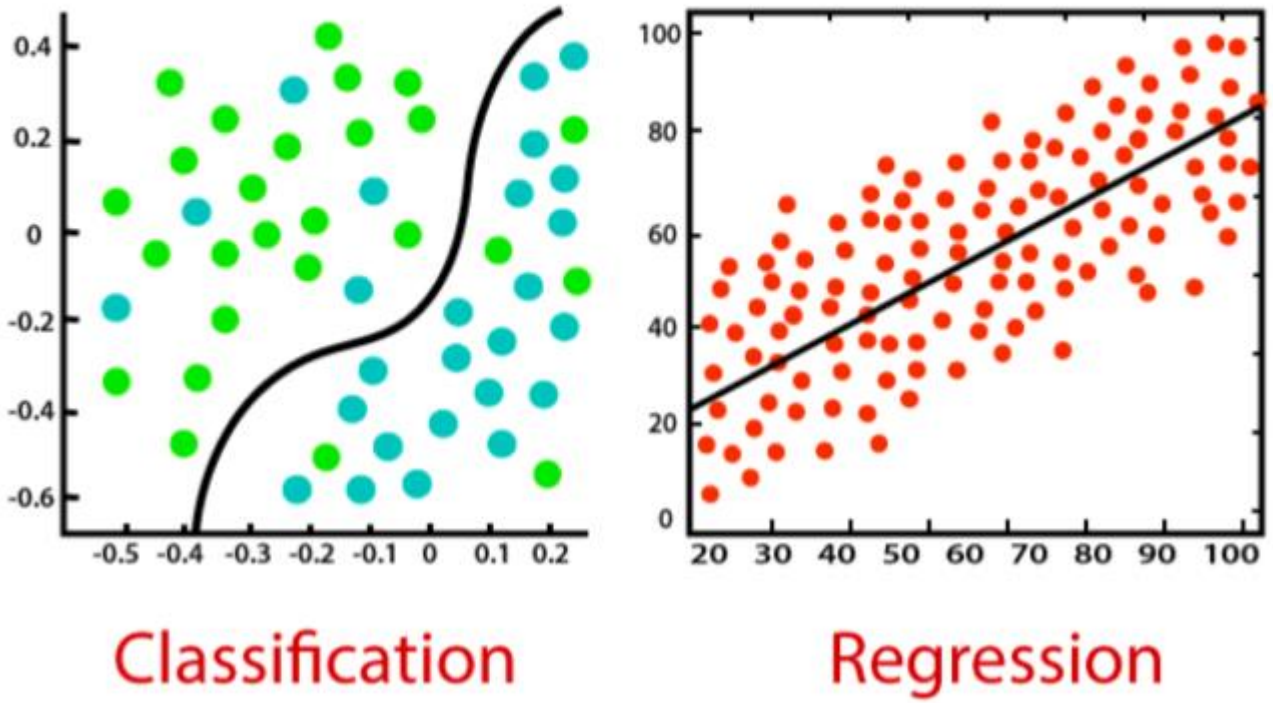
# Random Forest Parameter

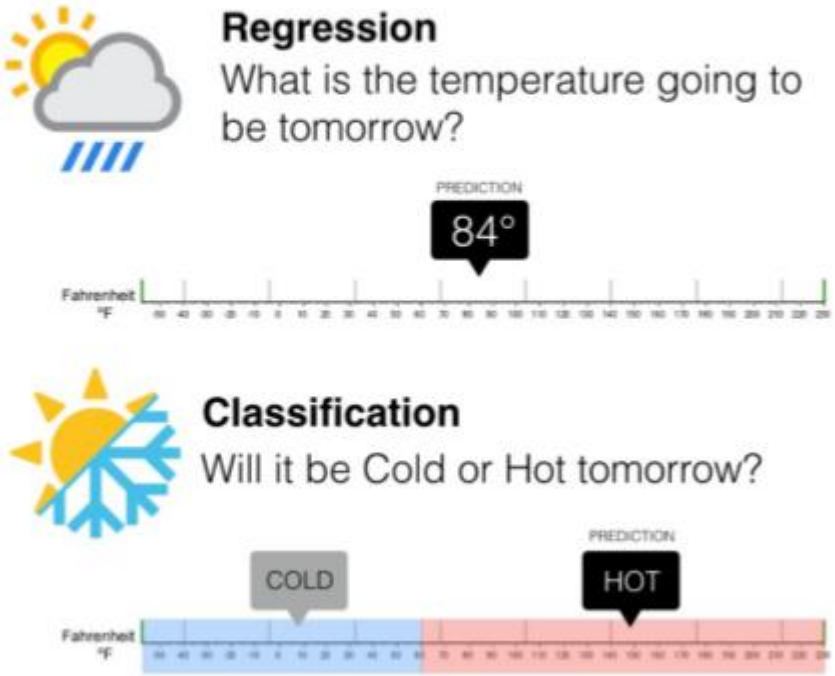
- **n_estimators:** 랜덤 포레스트 안의 결정 트리 갯수
  - n_estimators는 클수록 좋음. 결정 트리가 많을수록 더 깔끔한 Decision Boundary생성. 하지만 메모리와 훈련 시간이 증가

- **max_features:** 무작위로 선택할 Feature의 개수
  - max_features=n_features이면 30개의 feature 중 30개의 feature 모두를 선택해 결정 트리 생성
  - max_features 값이 크면 랜덤 포레스트의 트리들이 매우 비슷해지고, 가장 두드러진 특성에 맞게 예측을 할 것임
  - max_features 값이 작으면 랜덤 포레스트의 트리들이 서로 매우 달라질 것이고, 오버피팅이 줄어들 것임

# Random Forest

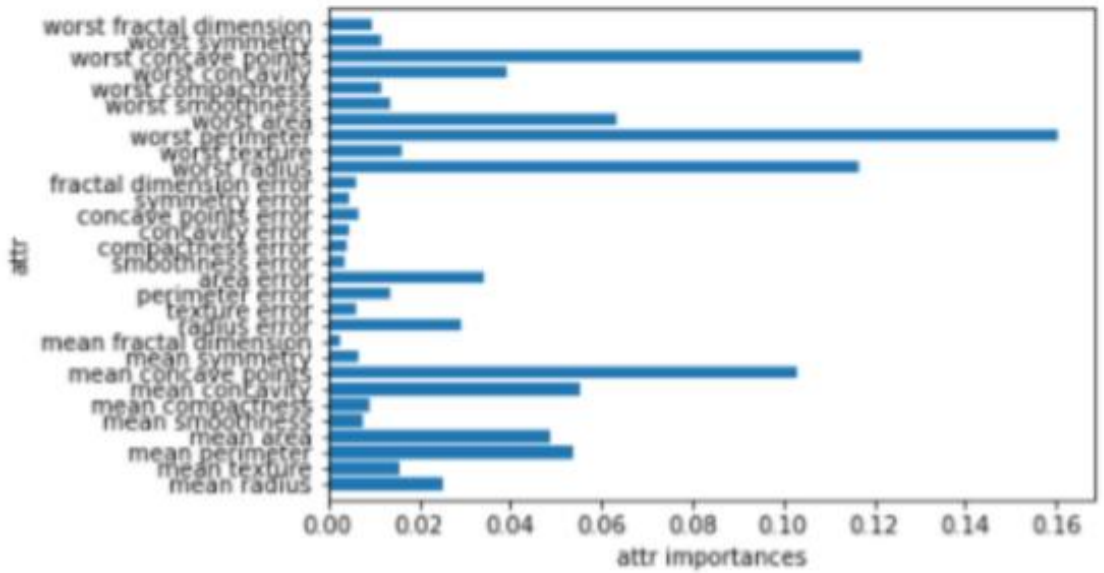
- RandomForestClassifier
- RandomForestRegressor



# Random Forest implement

훈련 세트 정확도 : 1.000
테스트 세트 정확도 : 0.972
특성 중요도 :
[0.02515433 0.01563844 0.05372655 0.04861645 0.007690078 0.00936994
 0.05539489 0.10305394 0.0065771  0.002827708 0.02921459 0.00607814
 0.01342868 0.034201174 0.00360641 0.00432096 0.00448775 0.00657502
 0.00460597 0.00627095 0.11657269 0.01603133 0.16027724 0.0634688
 0.01356448 0.01164113 0.03923725 0.11711756 0.01164259 0.00960721]



# Random Forest pros

- **회귀와 분류에 있어서 랜덤 포레스트는 현재 가장 널리 사용되는 머신러닝 알고리즘**
- 랜덤 포레스트는 성능이 매우 뛰어나고 매개변수 튜닝을 많이 하지 않아도 잘 작동하며, 데이터의 스케일을 맞출 필요도 없음
- 기본적으로 랜덤 포레스트는 단일 트리의 단점을 보완하고 장점은 가지고 있음

- 대량의 데이터셋에서 랜덤 포레스트 모델을 만들 때 다소 시간이 걸릴 수 있지만 CPU코어가 많다면 손쉽게 병렬 처리할 수있음
- **n_jobs 매개변수를 이용하여 사용할 코어 수를 지정**
  - (n_jobs=-1로 지정하면 컴퓨터의 모든 코어를 사용)

# Random Forest

## 주의할점

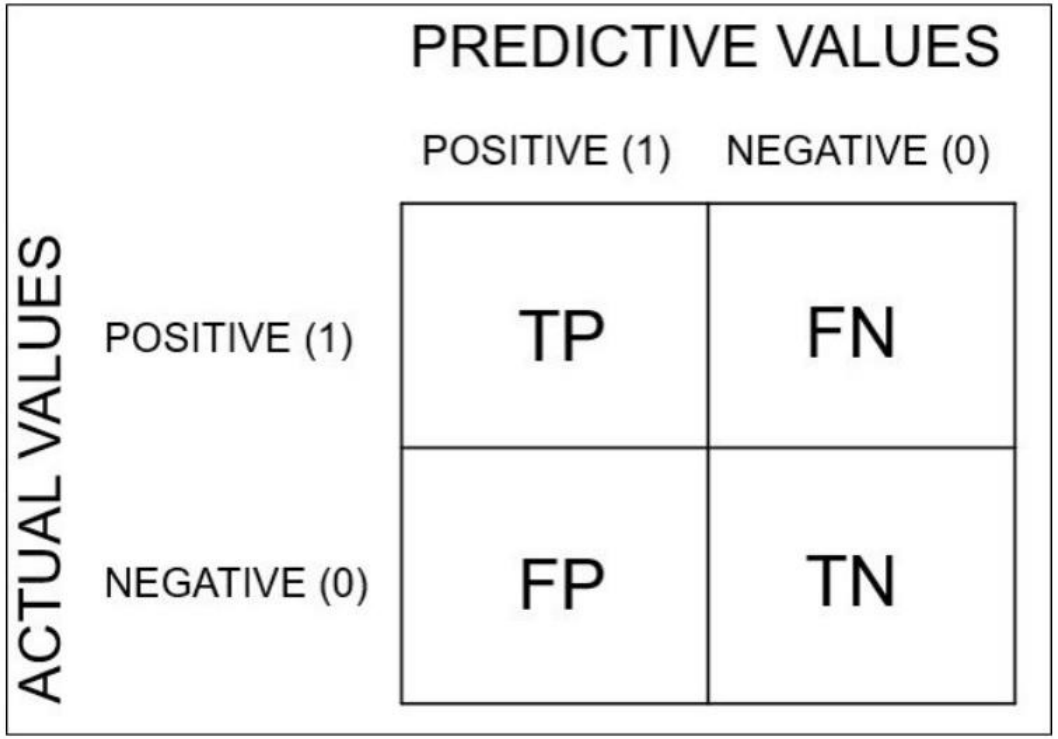
- 랜덤 포레스트는 랜덤하기때문에 random_state를 다르제 지정하면 전혀 다른 모델이 만들어짐
- 당연히 랜덤 포레스트의 트리가 많을수록 random_state값의 변화에 따른 변동이 적음
- **랜덤 포레스트는 텍스트 데이터와 같이 매우 차원이 높고 희소한 데이터에는 잘 작동하지 않음**
- 이러한 데이터에는 선형 모델이 더 적합
- 메모리를 많이 사용하기에 훈련과 예측이 느림

# Matrix Confusion

# RF(Random Forest) 예제, 예측결과

- **Confusion Matrix**

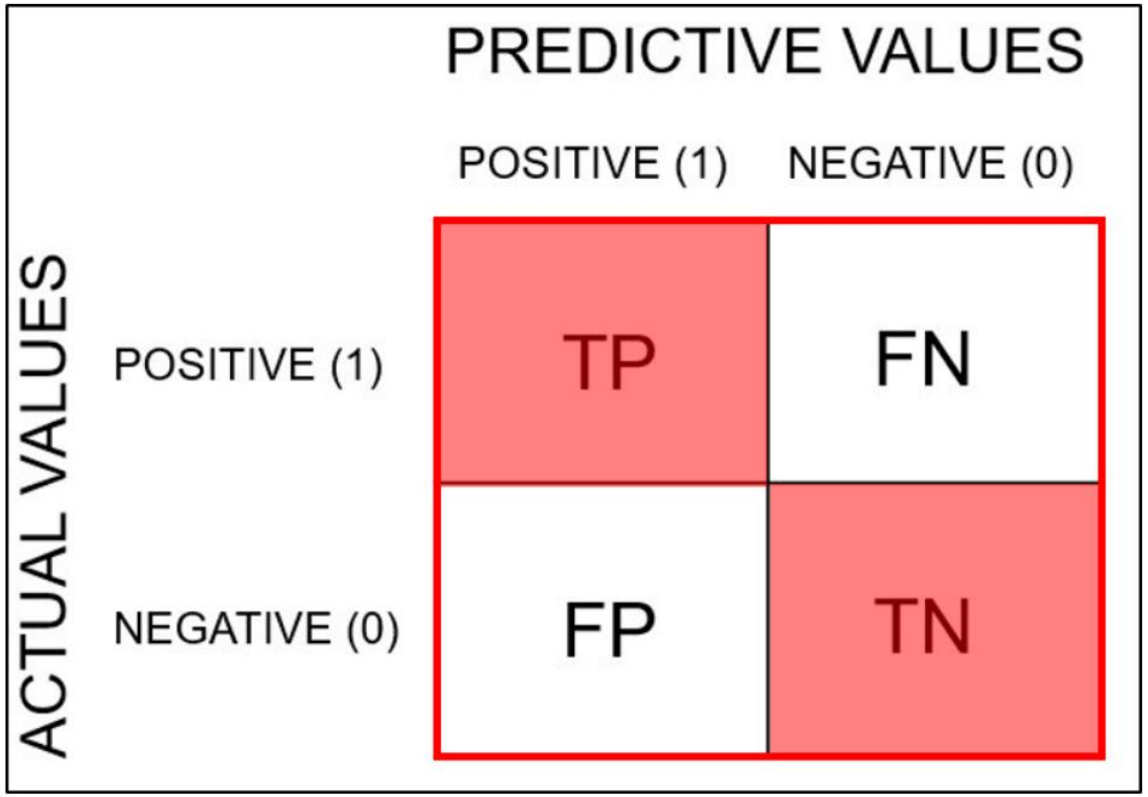
학습을 통해서 얻은 예측값을 실제값과 비교하기 위한 Matrix
모델의 정확도 | 실제값을 정확히 예측한 값의 정밀도 | 재현도를 확인할 수 있다



# RF(Random Forest) 예제, 예측결과

- **Confusion Matrix – Accuracy**

  모델이 얼마나 바르게 분류했는가의 비율로 Matrix에서 대각선 부분이 해당된다.



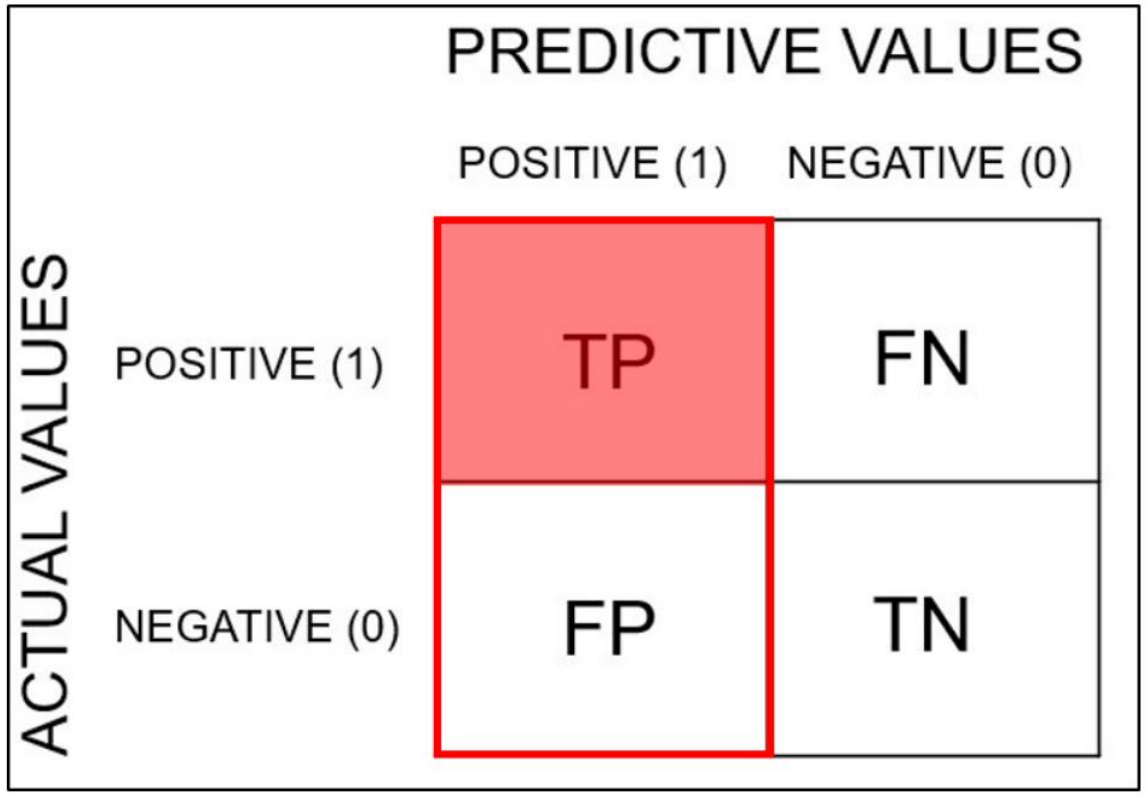

**TP** 실제 **T**인 정답을 **T**라고 예측 (예측-정답 일치)
**TN** 실제 **F**인 정답을 **F**라고 예측 (예측-정답 일치)

**FN** 실제 **T**인 정답을 **F**라고 예측 (예측-정답 불일치)
**FP** 실제 **F**인 정답을 **T**라고 예측 (예측-정답 불일치)

# RF(Random Forest) 예제, 예측결과

- **Confusion Matrix – Precision**

모델이 True로 분류한것중 실제로 True인 값의 비율, Positive 정답률

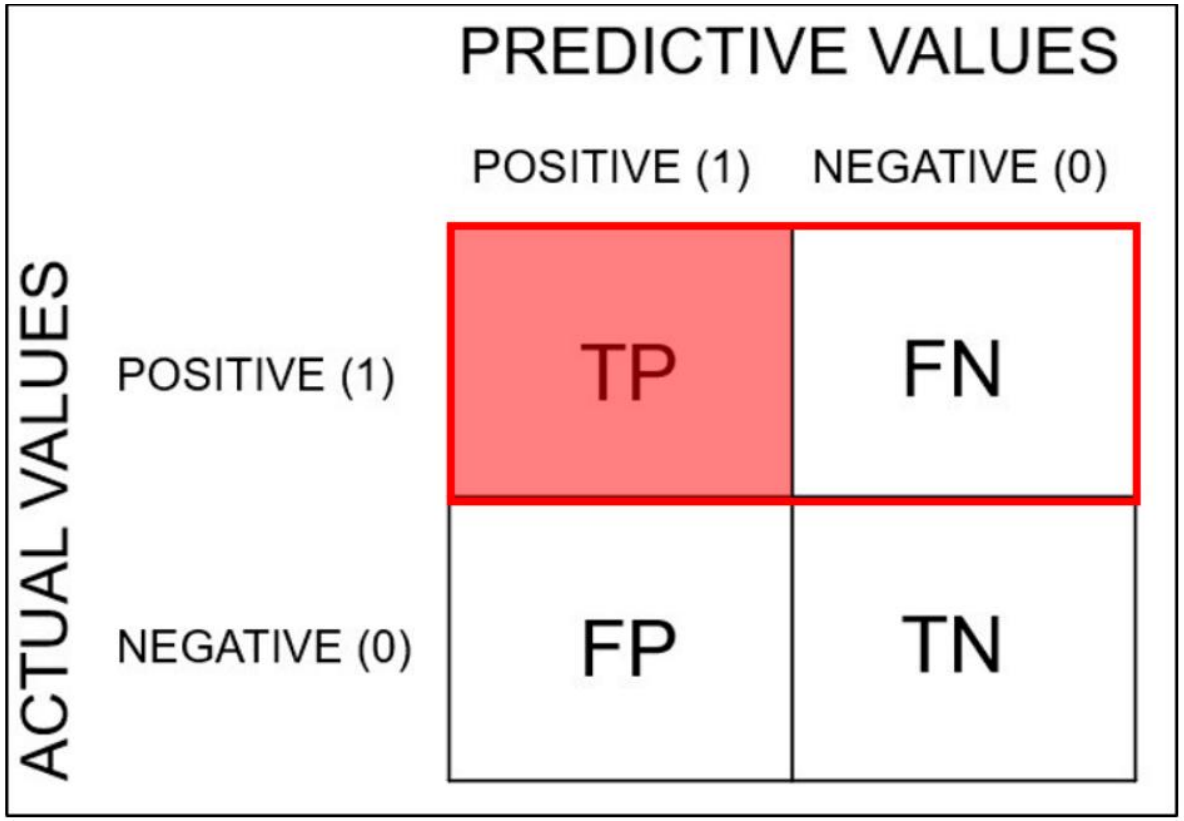



Matrix에서 왼편의 붉은선에 해당됨.
열을 비교

# RF(Random Forest) 예제, 예측결과

- **Confusion Matrix – Recall**

실제값이 True인 것중에 모델이 True로 분류한 비율

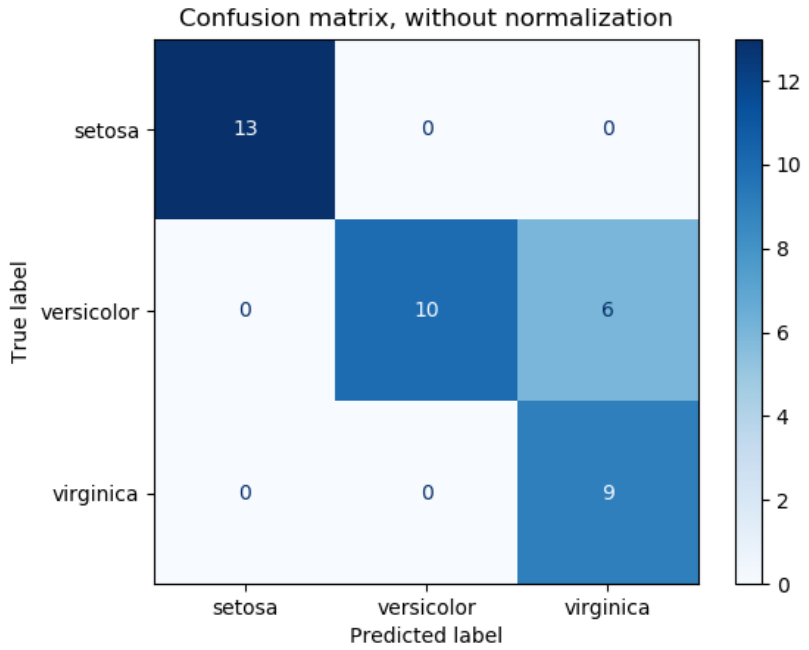



Matrix에서 왼편의 붉은선에 해당됨.

# RF(Random Forest) 예제, 예측결과

## Confusion Matrix

- Classification 머신러닝 모델이 제대로 작동을 했는지 혼동을 했는지 알아볼 수 있는 행렬
- 행(row)는 실제 클래스, 열(column)은 예측한 클래스



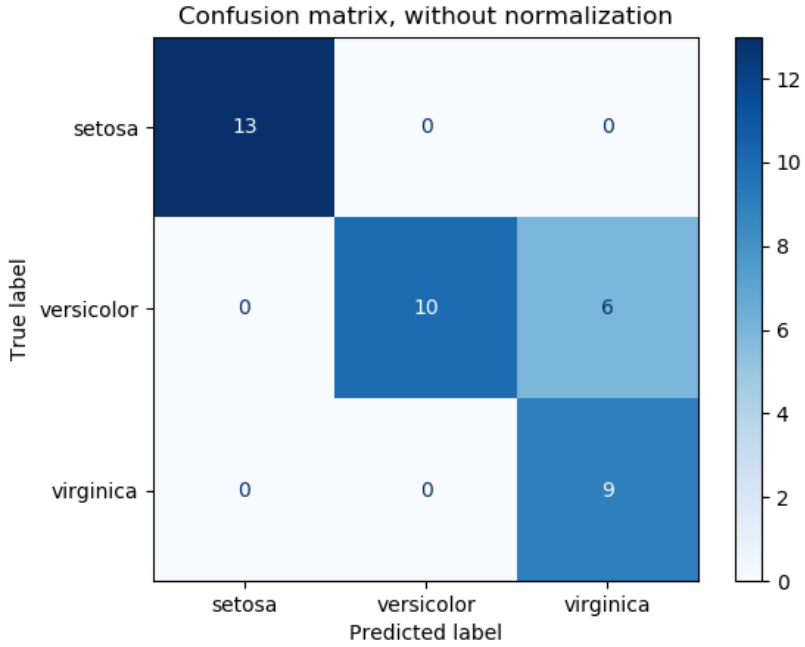

### 1. Basic confusion matrix

- seaborn의 heatmap은 confusion matrix를 시각화 하는 함수
- 색상이 밝으면 높은 숫자, 색상이 어두우면 낮은 숫자를 나타냄

# RF(Random Forest) 예제, 예측결과

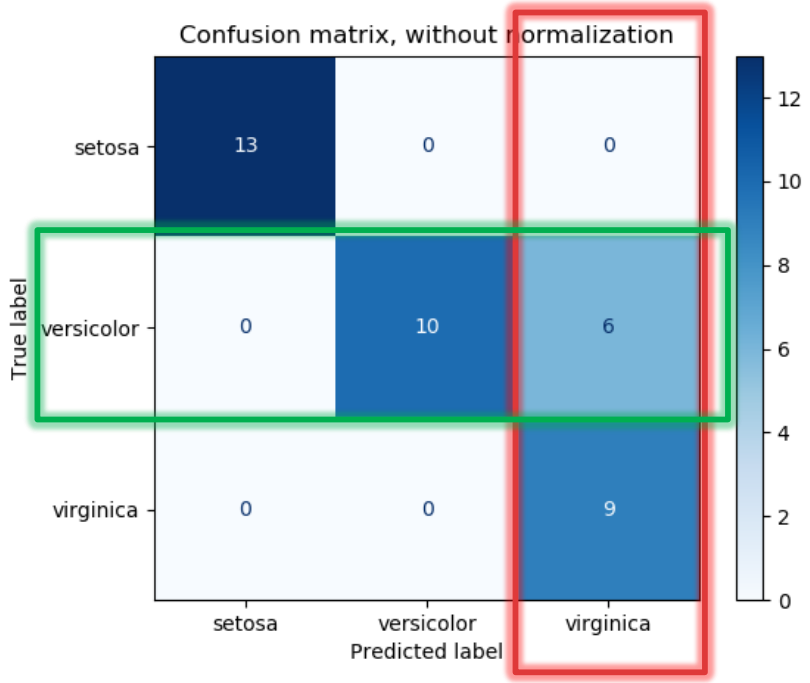
- **Confusion Matrix**

```
In [11]: from sklearn.metrics import confusion_matrix

         cm = pd.DataFrame(confusion_matrix(y_test, predicted), col
         sns.heatmap(cm, annot=True)

Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x7f70f7448320>
```

# RF(Random Forest) 예제, 예측결과

- **Precision / recall / Accuracy**



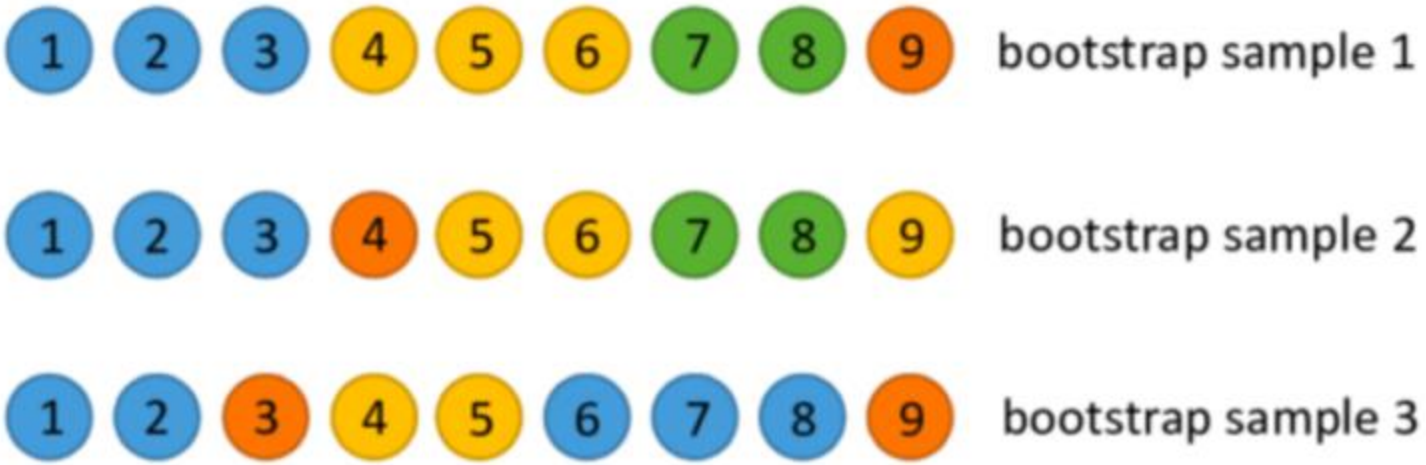

```
In [40]: print(classification_report(prediction, y_test))
```

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 1 | 0.00 | 0.00 | 0.00 | 8 |
| 2 | 0.73 | 1.00 | 0.85 | 22 |
| micro avg | 0.73 | 0.73 | 0.73 | 30 |
| macro avg | 0.37 | 0.50 | 0.42 | 30 |
| weighted avg | 0.54 | 0.73 | 0.62 | 30 |

# Random Forest
# &
# Bootstrap

# 부트 스트랩(bootstrap sampling)

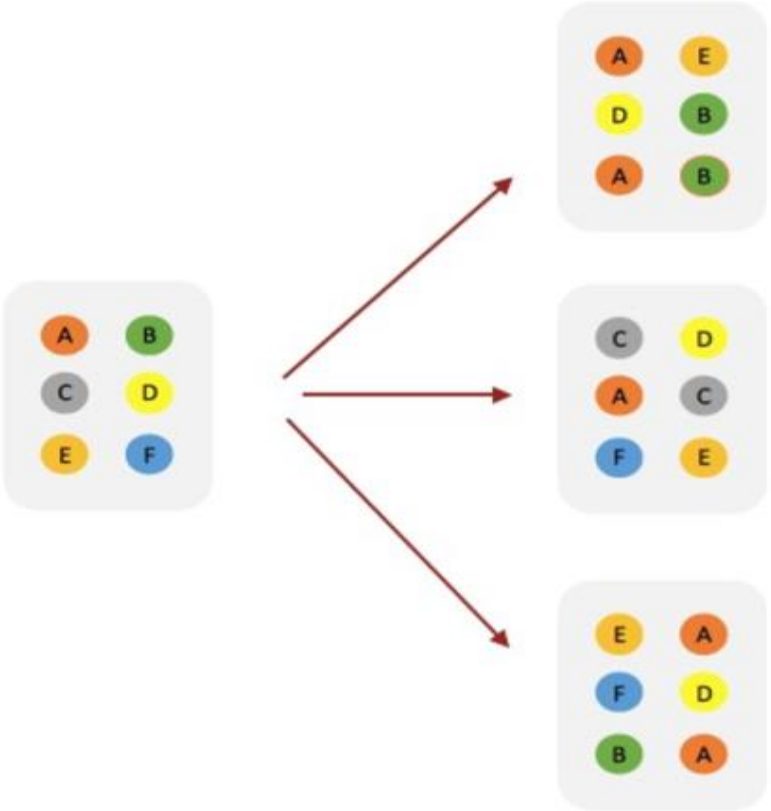
original sample




In statistics, **bootstrapping** is any test or metric that relies on random sampling with replacement.

# Bagging

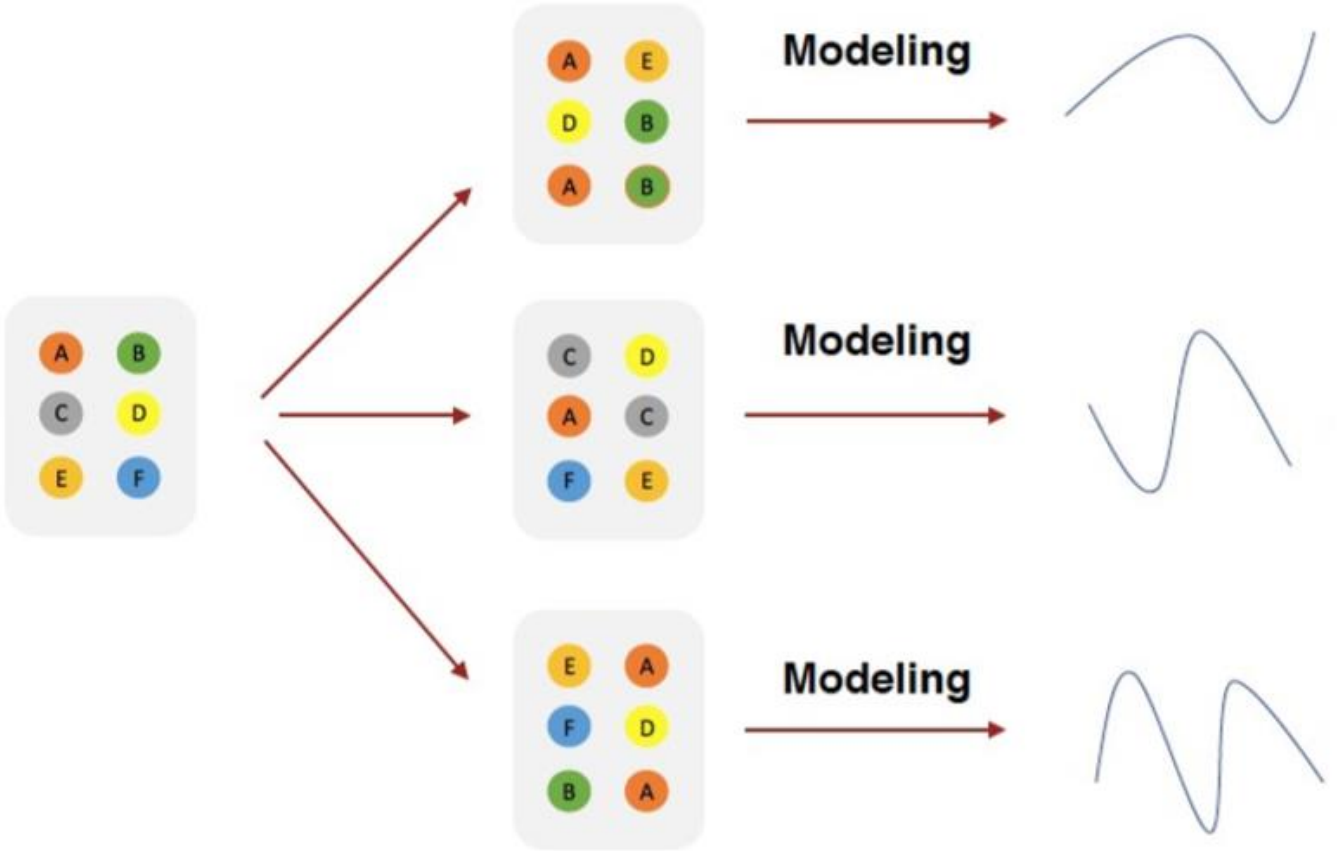
**Bagging은 Bootstrap Aggregating의 줄임말입니다**




**Random Sampling with replacement**

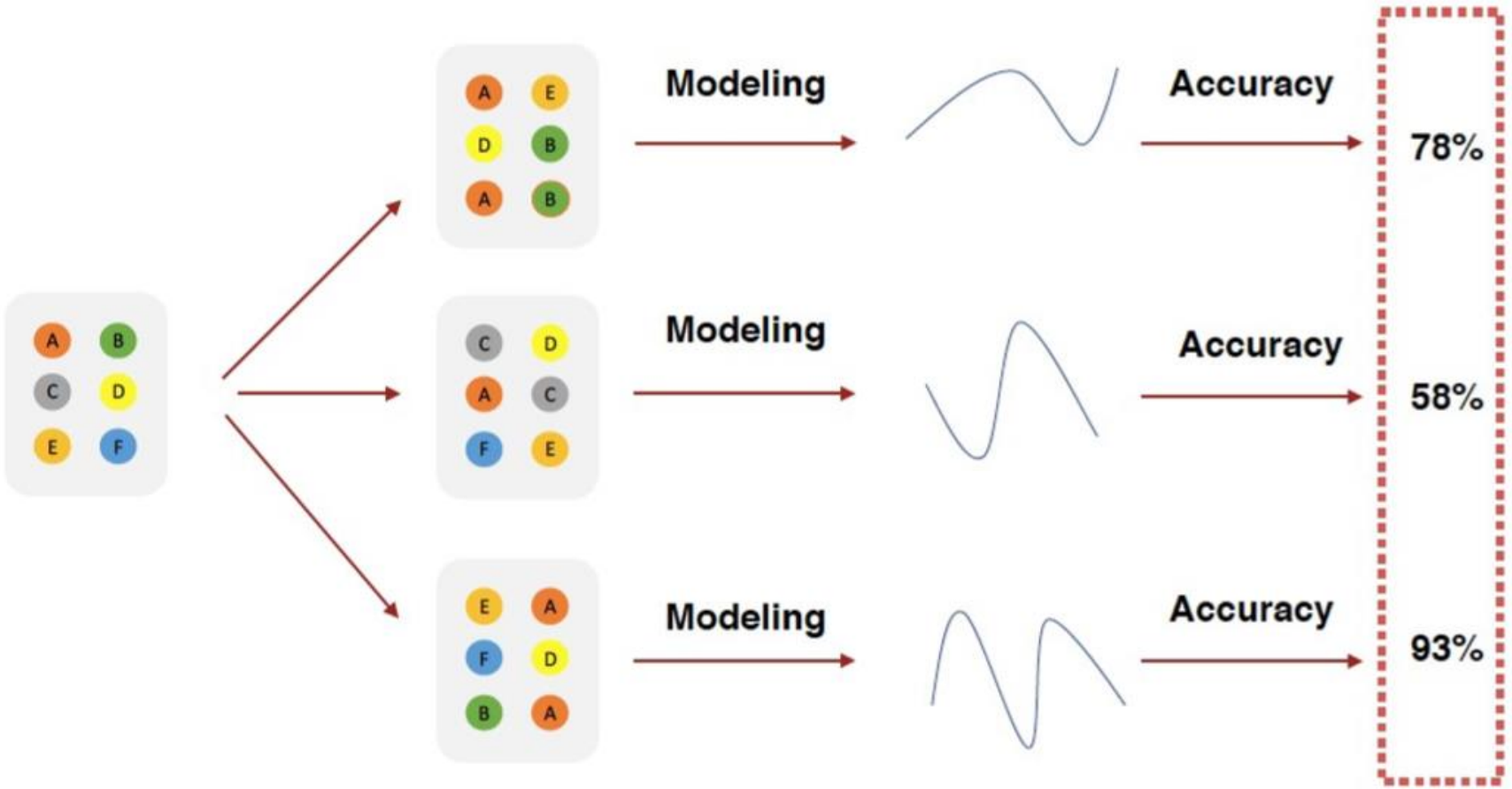
# Bagging

추출된 표본들에 각각 모델 (ex. Decision Tree)를 적합시켜 모델을 만들 수 있습니다



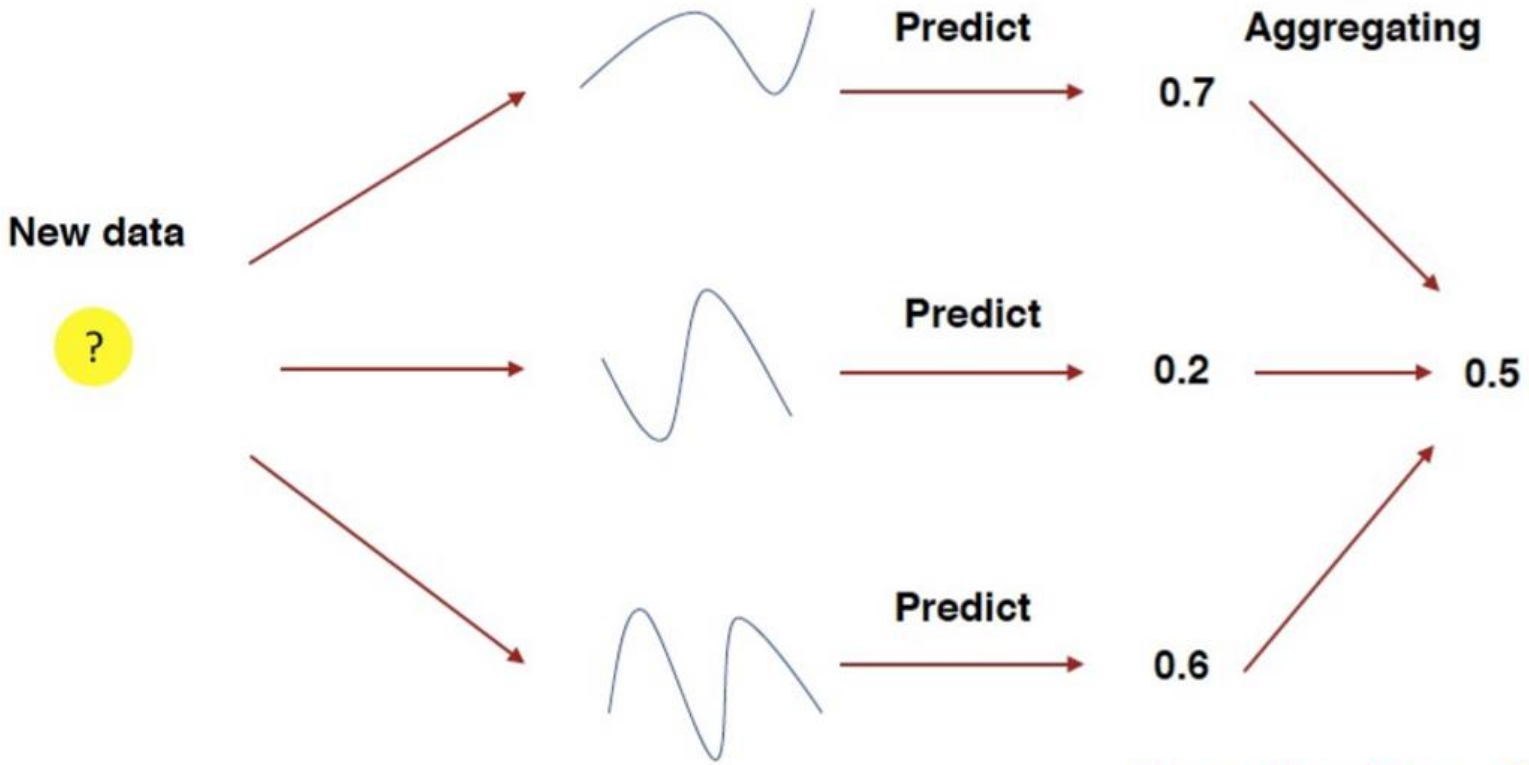
# Bagging

그리고 각각의 모델들은 예측 성능을 가지고 있습니다
이 값을 바탕으로 예측 성능을 얼마나 믿을 수 있는지 측정할 수 있습니다



# Bagging

각각의 예측 결과를 합쳐 하나의 예측결과로 만들 수 있습니다



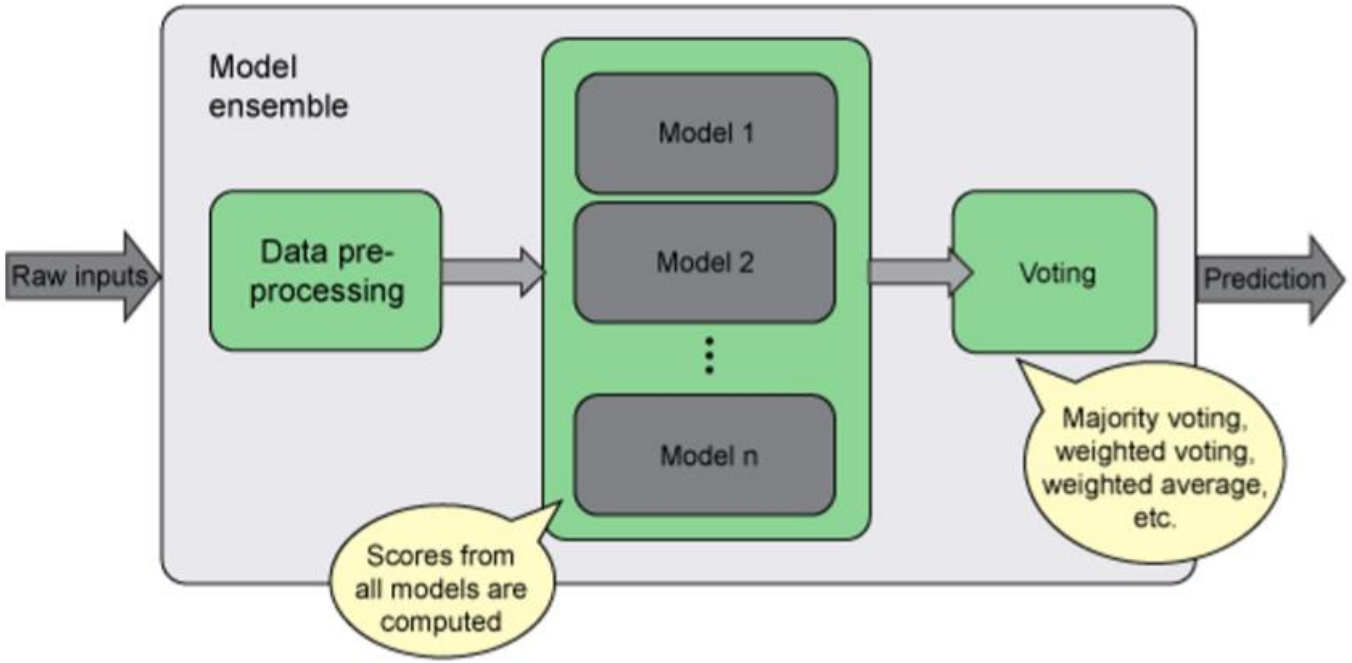

Classification : 다수결
Regression : 평균

# Ensemble

지금까지 설명한 모든것은 Ensemble 기법중의 하나이다.

**"Random Forest is a Ensemble…"**

**여러 모델을 이용하여 결과를 예측하고, 이를 결합하는 것을 Ensemble이라고 합니다**

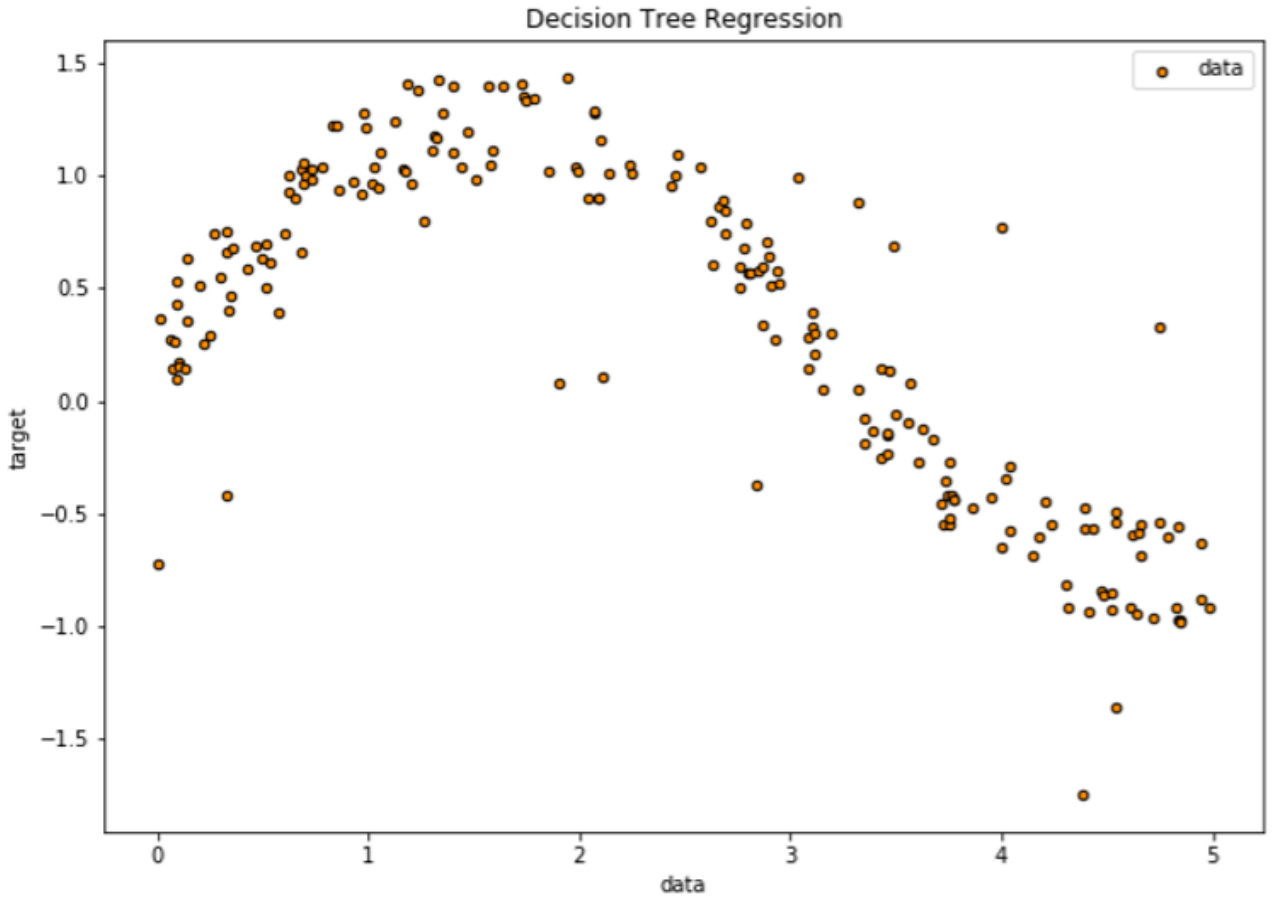



- **Error 최소화**
다양한 모델의 결과를 종합하여 전반적 오류를 줄여줌

- **모델별 한계를 극복**
모델별로 가지고 있는 한계를 여러가지 모델의 결과를 종합해 극복

# DecisionTree의 한계점

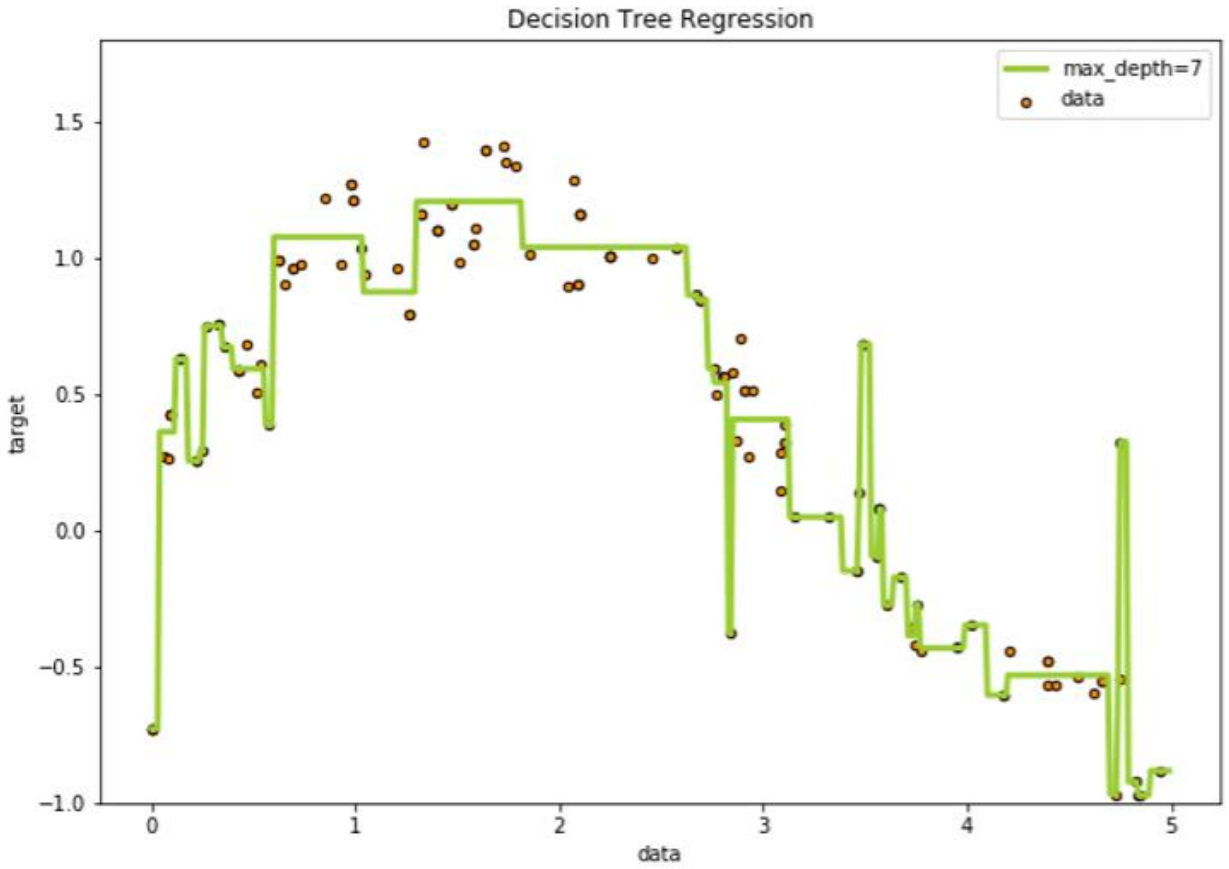
**Decision Tree는 Variance가 큰 모델입니다**
**Input이 조금이라도 변하면 모델이 크게 변합니다**




**아웃라이어가 있는 위와 같은 2차원 데이터가 있다고 해봅시다**

# DecisionTree의 한계점

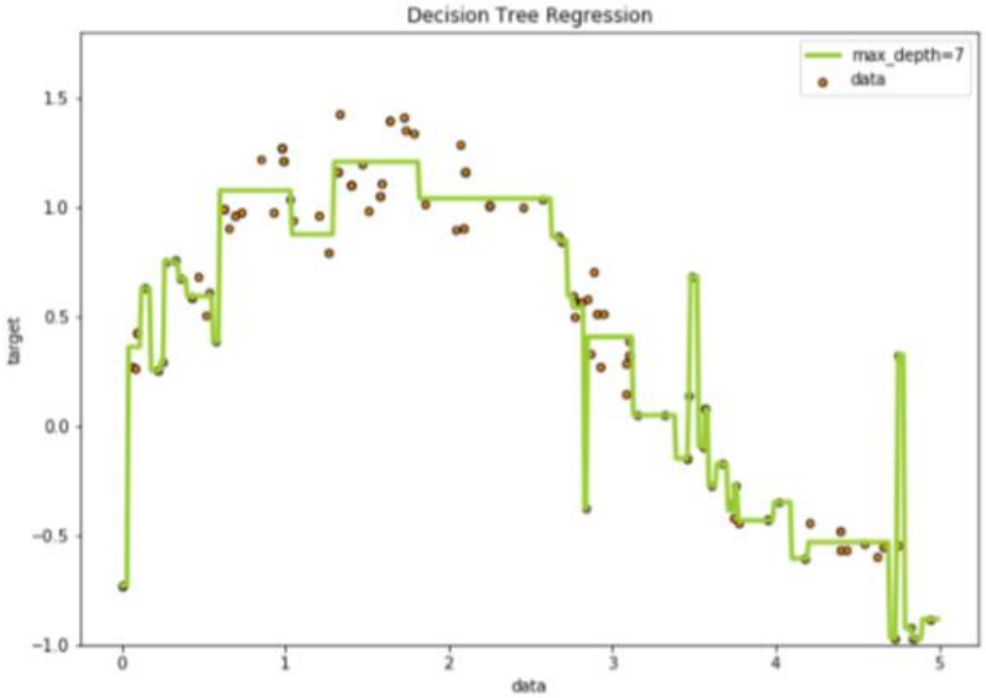
**위의 원본데이터에서 70% 샘플링을 진행한 뒤 Decision Tree를 적합**
**그리고 각각의 x값에 대해 y값을 예측하면 다음과 같은 그래프가 나옵니다**

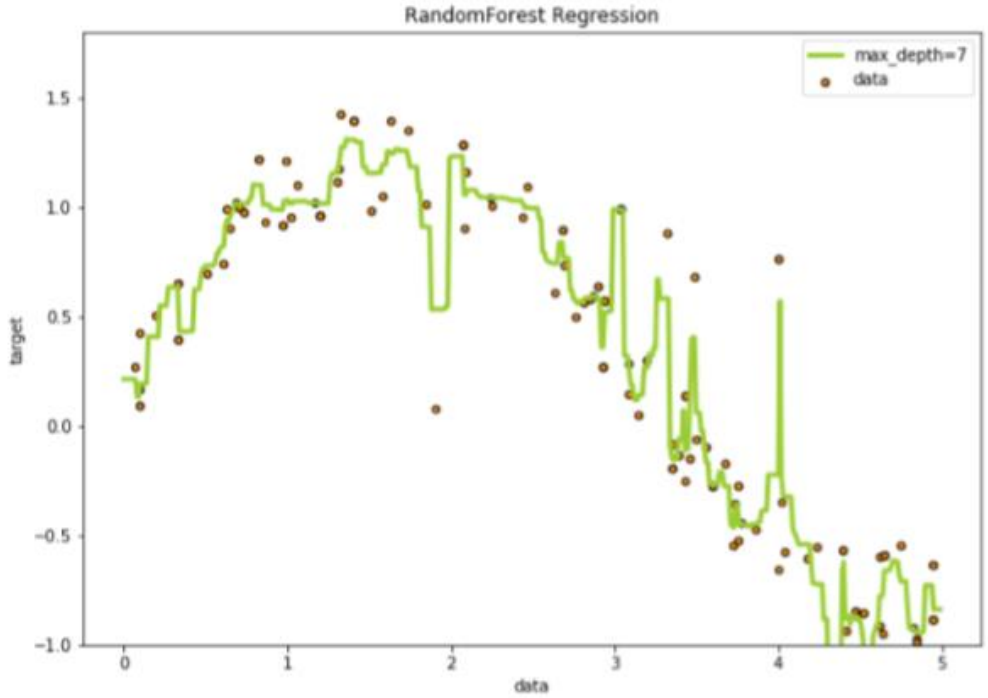



**Input 데이터에 큰 영향을 받음**
**= 모델의 Variance가 큼**

# RandomForest가 이를 극복하는 방법

**Decision Tree 여러개를 결합하여 Variance (변동성)을 낮춘다**
**tree처럼 outlier의 값을 그대로 따라가지 않고 모함수와 더 비슷해지는 것을 볼 수 있다**



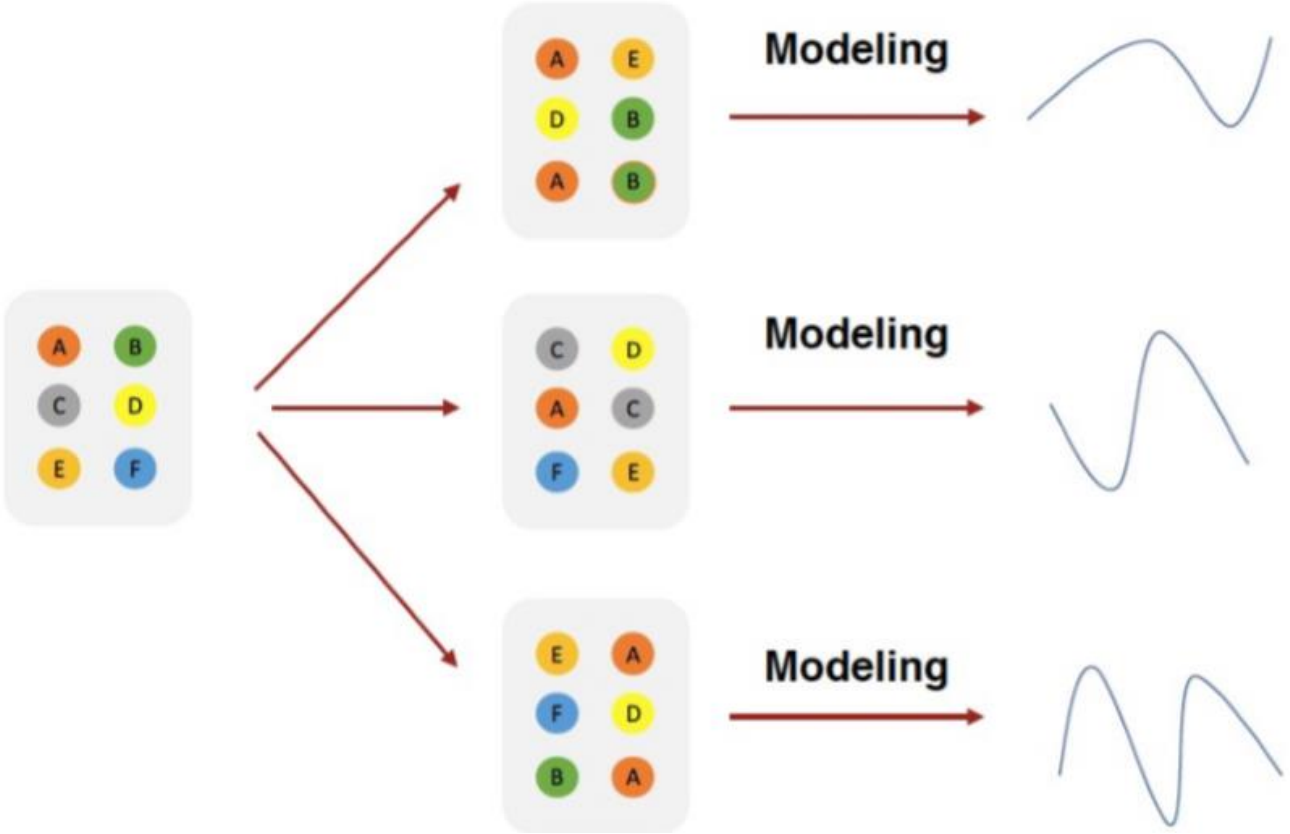




**High Variance**

**Low Variance**

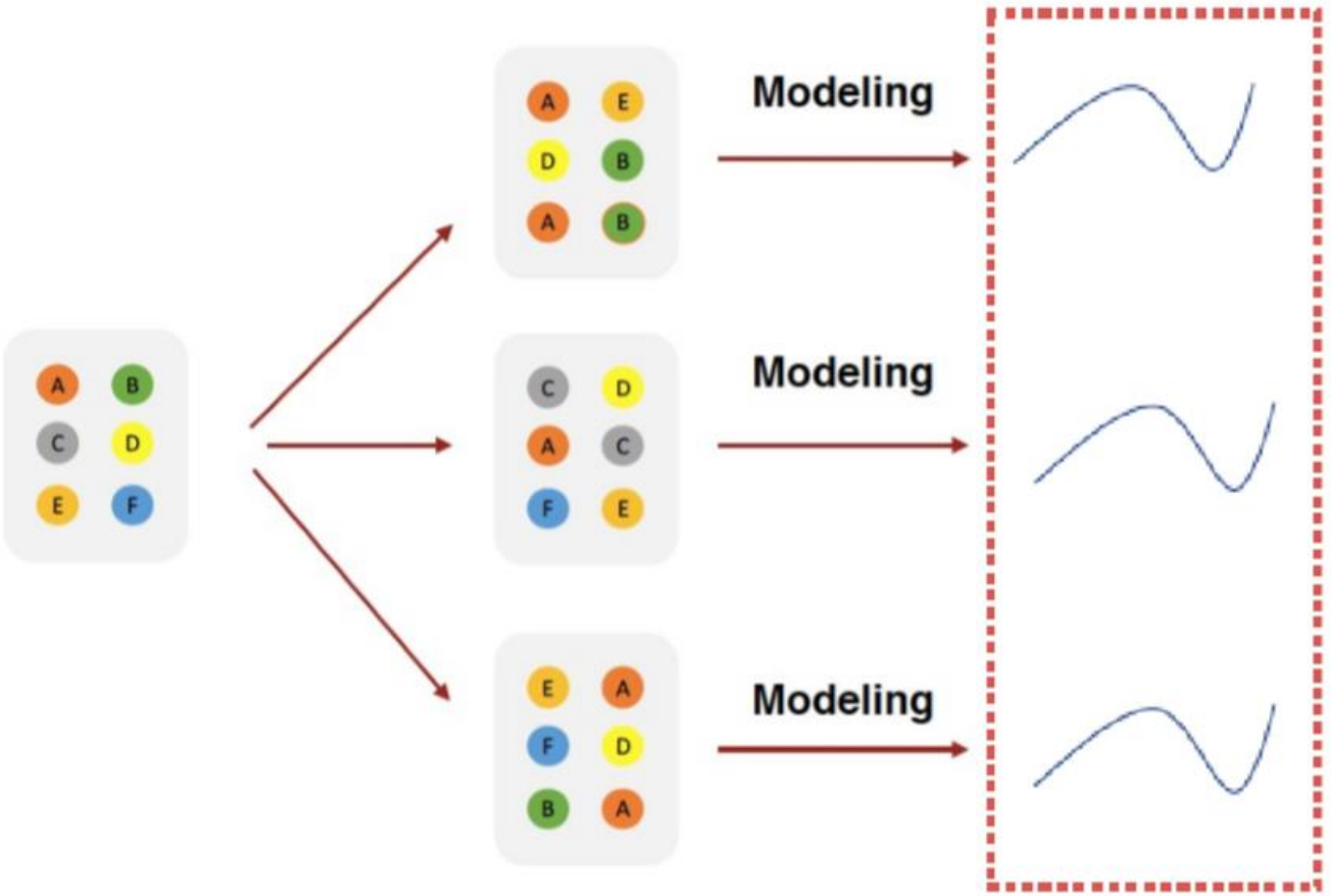
# Bagging의 문제점

다음과 같이 같은 데이터에 대해 다양한 모델이 나오는 것이 이상적입니다



# Bagging의 문제점

하지만 다음과 같이 모델이 전부 비슷하게 만들어지면 합치는 의미가 없어집니다

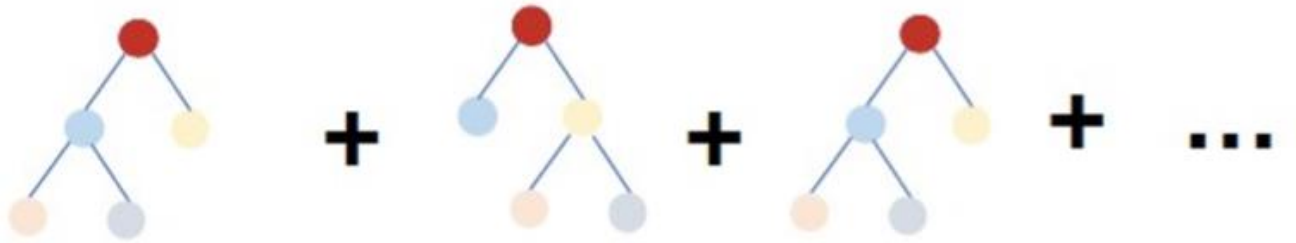


# Bagging의 문제점

강력한 설명변수가 있다면 tree가 그 변수로 인해 대부분 비슷해집니다

이상적인 Bagging

강력한 설명변수가 있을 때

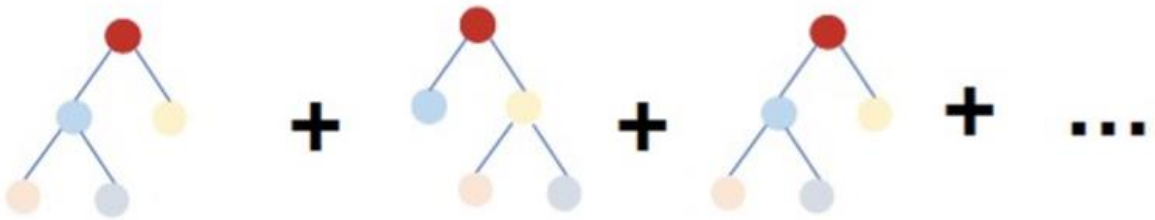


# Bagging의 문제점

강력한 설명변수가 있다면 tree가 그 변수로 인해 대부분 비슷해집니다

이상적인 Bagging

강력한 설명변수가 있을 때



트리간에 강력한 상관관계가 생겨 이를 ensemble해도 분산이 감소하지 않습니다

# Boosting

- AdaBoost, XGBoost, GradientBoost

# Boosting

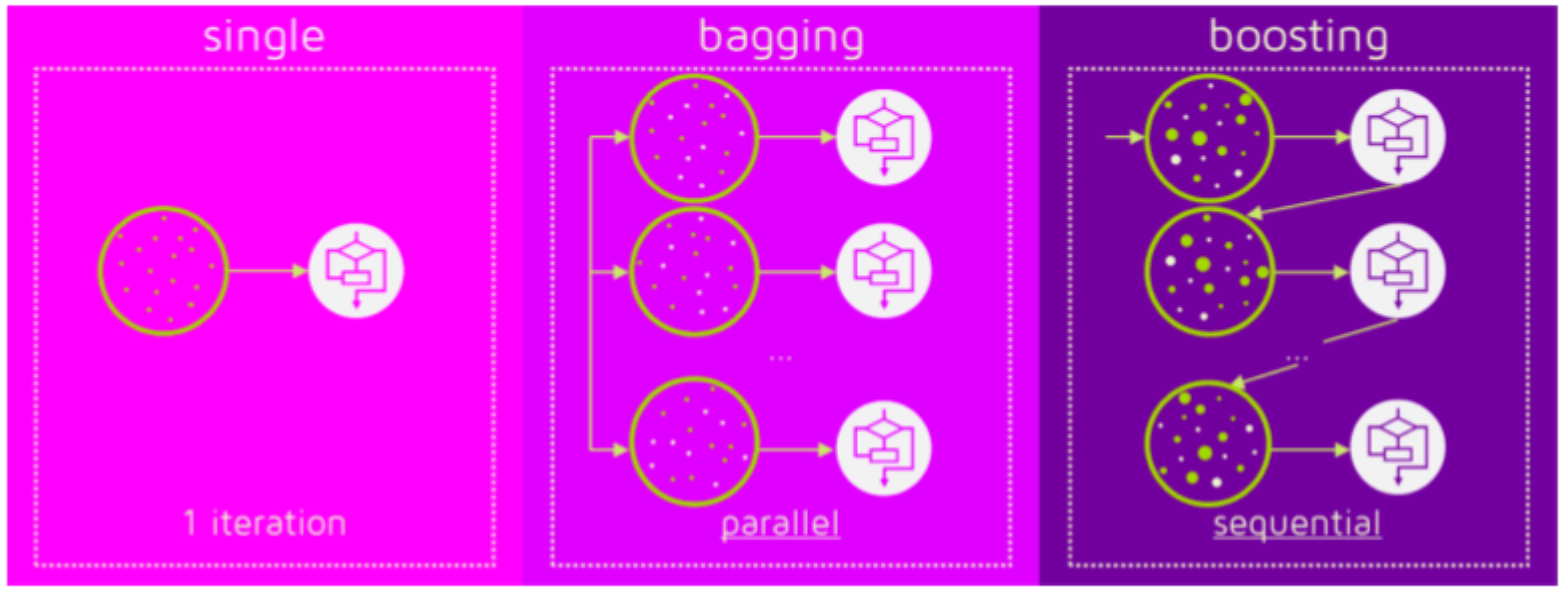
## Basic Idea of Boosting

- 직역하면, 약한 학습기, 우리는 데이터를 통해 모델링을 하고 결국에는 새로운 데이터가 들어왔을 때, 예측이나 분류를 해주기 원한다. 따라서 모델은 학습 데이터에 너무 편향되면 안된다(overfitting). 하나의 모델이 학습 데이터에 overfitting 되는 것을 막기 위해 약한 모델을 여러개 결합시켜 그 결과를 종합한다는게 기본적인 앙상블의 아이디어다.

- 부스팅은 여기에 sequential이 추가된다. 즉 연속적인 weak learner, 바로 직전 weak learner의 error를 반영한 현재 weak learner를 잡겠다는 아이디어이다. 이 아이디어는 GBM에서 loss를 계속 줄이는 방향으로 weak learner를 잡는다는 개념으로 확장된다.

# Boosting

- Bagging이 일반적인 모델을 만드는데 집중되어있다면, Boosting은 맞추기 어려운 문제를 맞추는데 초점이 맞춰져 있음
- 수학 문제를 푸는데 9번 문제가 엄청 어려워서 계속 틀렸다고 가정할 때, Boosting 방식은 9번 문제에 가중치를 부여해서 9번 문제를 잘 맞춘 모델을 최종 모델로 선정




- Bagging이 병렬로 학습하는 반면, Boosting은 순차적으로 학습시킴
- 학습이 끝나면 나온 결과에 따라 가중치가 재분배

# Boosting Algorithm Implement

- 중요한 매개변수로는 이전 트리의 오차를 얼마나 강하게 보정할 것인지를 제어하는 learning_rate.
  - 학습률이 크면 트리는 보정을 항하게 하기 때문에 복잡한 모델이 됨.
- n_estimators값을 키우면 앙상블에 트리가 더 많이 추가되어 모델의 복잡도가 증가

```
# coding: utf-8
from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import mglearn
import numpy as np

cancer = load_breast_cancer()

# 훈련/테스트 세트로 나누기
X_train, X_test, y_train, y_test = train_test_split(cancer.data,cancer.target, random_state=0)

gbrt = GradientBoostingClassifier(random_state=0)

gbrt.fit(X_train,y_train)
print("훈련 세트 정확도 : {:.3f}".format(gbrt.score(X_train,y_train)))
print("테스트 세트 정확도 : {:.3f}".format(gbrt.score(X_test,y_test)))
```

```
훈련 세트 정확도 : 1.000
테스트 세트 정확도 : 0.958
```

# Boosting Algorithm Implement

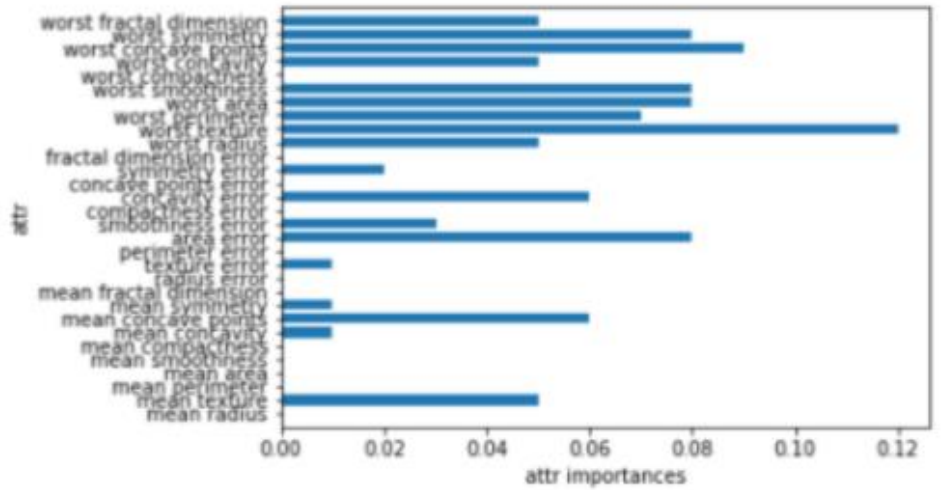
```
cancer = load_breast_cancer()

# 훈련/테스트 세트로 나누기
X_train, X_test, y_train, y_test = train_test_split(cancer.data,cancer.target, random_state=0)

gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train,y_train)

print("훈련 세트 정확도 : {:.3f}".format(gbrt.score(X_train,y_train)))
print("테스트 세트 정확도 : {:.3f}".format(gbrt.score(X_test,y_test)))
# 훈련 세트 정확도 : 1.000
# 테스트 세트 정확도 : 0.958
```

```
# 훈련 세트의 정확도가 100%이므로 과대적합되었다.
# 과대적합을 막기위해 사전 가지치기를 한다.
gbrt = GradientBoostingClassifier(random_state=0,max_depth=1)
gbrt.fit(X_train,y_train)

print("훈련 세트 정확도 : {:.3f}".format(gbrt.score(X_train,y_train)))
print("테스트 세트 정확도 : {:.3f}".format(gbrt.score(X_test,y_test)))
# 훈련 세트 정확도 : 0.991
# 테스트 세트 정확도 : 0.972

# 과대적합을 막기위해 학습률을 낮춘다
gbrt = GradientBoostingClassifier(random_state=0,learning_rate=0.01)
gbrt.fit(X_train,y_train)

print("훈련 세트 정확도 : {:.3f}".format(gbrt.score(X_train,y_train)))
print("테스트 세트 정확도 : {:.3f}".format(gbrt.score(X_test,y_test)))
# 훈련 세트 정확도 : 0.988
# 테스트 세트 정확도 : 0.965
```

# 특성의 중요도

- gbrt.feature_importances_

```
훈련 세트 정확도 : 0.991
테스트 세트 정확도 : 0.972
특성 중요도 :
[0.    0.05 0.    0.    0.    0.    0.01 0.06 0.01 0.    0.    0.01 0.    0.08
 0.03 0.    0.06 0.    0.02 0.    0.05 0.12 0.07 0.08 0.08 0.    0.05 0.09
 0.08 0.05]
```



# Boosting Algorithm Pros/Cons & Parameter

- 그래디언트 부스팅 결정 트리는 지도 학습에서 가장 강력하고 널리 사용하는 모델 중 하나임
- **단점은 매개변수 조정을 잘해야한다. 또한 훈련 시간이 길다. 트리기반 특성상 고차원 데이터에는 잘 작동하지 않음**
- **장점으로는 특성의 스케일을 조정하지 않아도 되고, 연석적인 특성에서도 잘 동작함**

- **매개변수**
  - **n_estimators가 클수록 모델이 복잡해지고 과대적합될 가능성이 높아짐**
  - **learning_rate를 낮추면 비슷한 복잡도의 모델을 만들이 위해 더 많은 트리를 추가해야 함**
  - 따라서 n_estimators를 맞추고 나서 적절한 learning_rate를 찾는것이 좋음
- max_depth: 보통 5보다 깊어지지 않게 사용

# Bias VS Variance

머신러닝 모델의 에러는 두가지로 분류할 수 있습니다. 바로 편향(Bias)과 분산(Variance) 입니다. 그리고 이 두가지 개념은 서로 다르게 움직입니다. 종종 Bias를 해결하면 Variance가 올라가고, Variance를 해결하면 Bias가 올라갑니다. 이 현상을 바로 **"Bias - Variance Tradeoff"(편향-분산 트레이드오프)**라고 합니다.

## Bias

Bias란 학습데이터를 충분히 표현할 수 없기 때문에 발생합니다.

높은 Bias를 보이는 모델은 **underfitting**이 된 상태입니다.

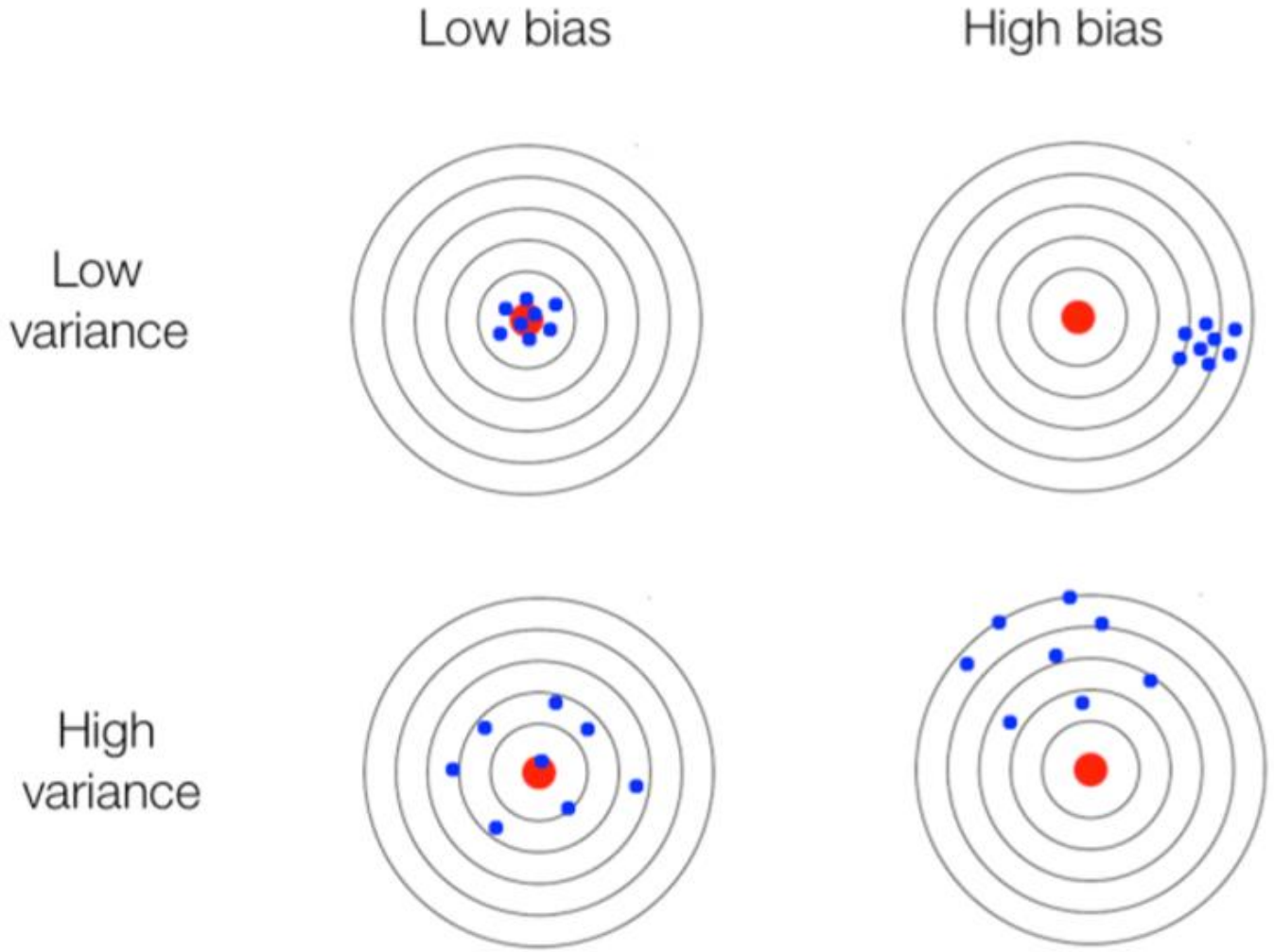
## Variance

Variance란 트레이닝 데이터에 너무 민감하게 반응하여 발생합니다.

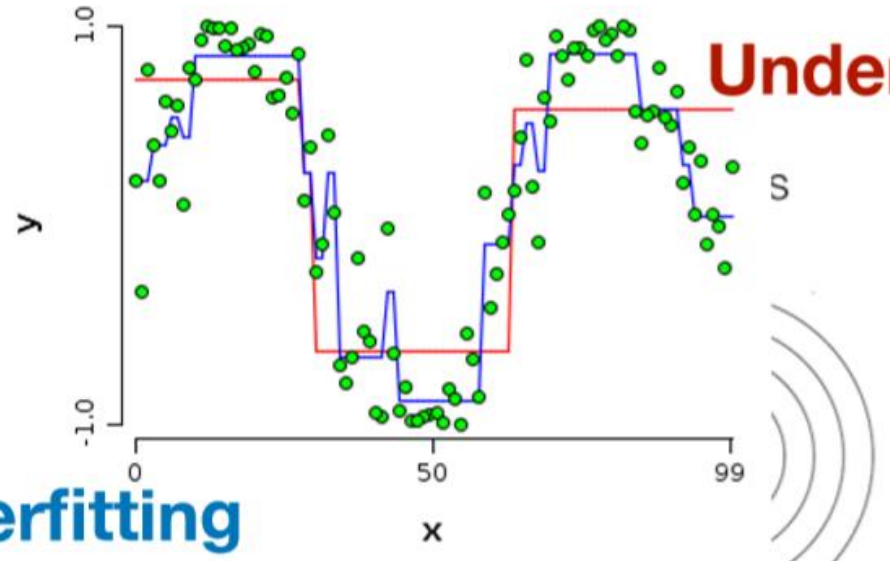
높은 Variance를 띄는 모델은 **overfitting**이 된 상태를 말합니다.

자료 출처 https://brunch.co.kr/@chris-song/32

# Bias vs Variance



# Bias vs Variance

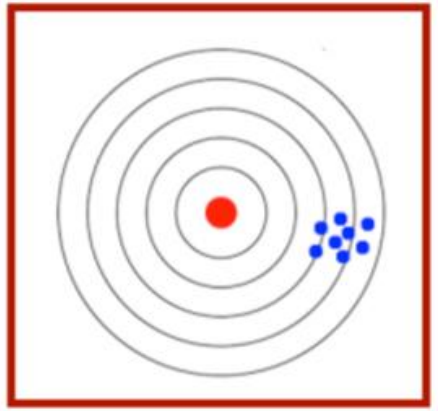



Underfitting

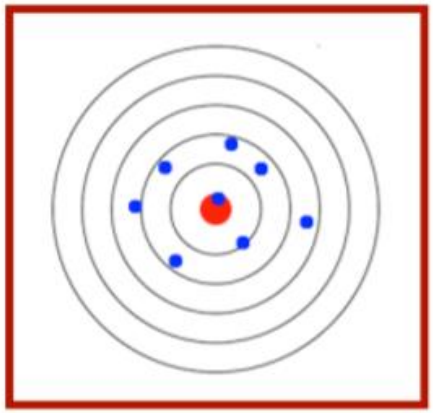
Overfitting

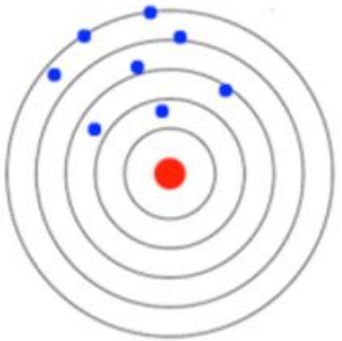
s

High bias

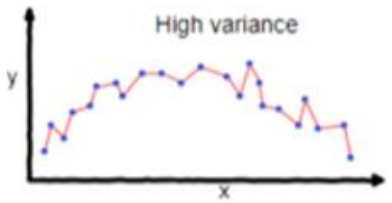


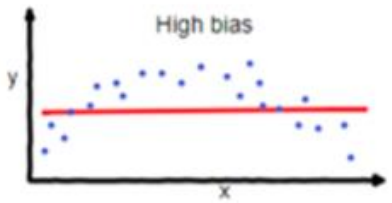
High variance

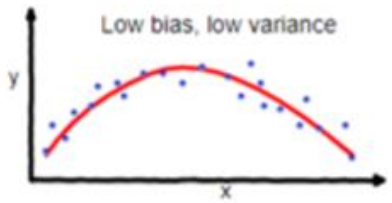




# Bias vs Variance



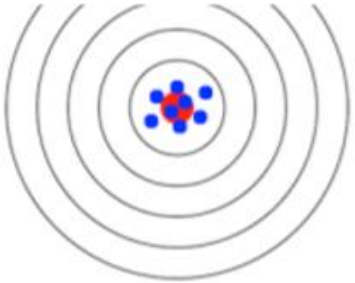

ligh bias

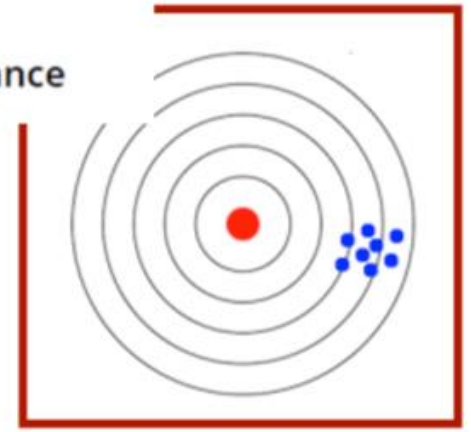
**overfitting**

**underfitting**

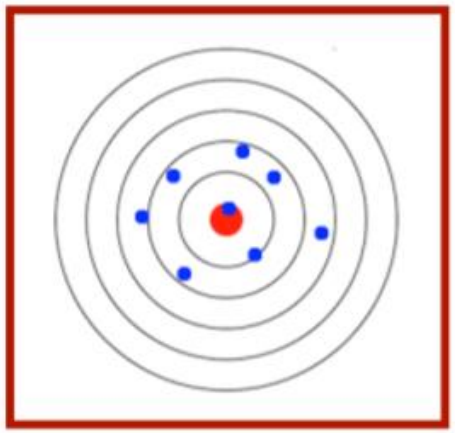
**Good balance**

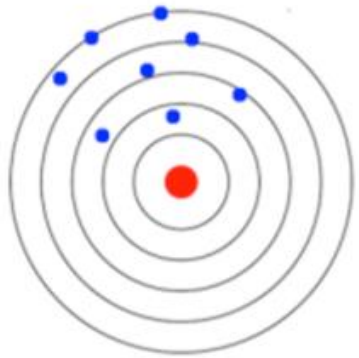
Low variance



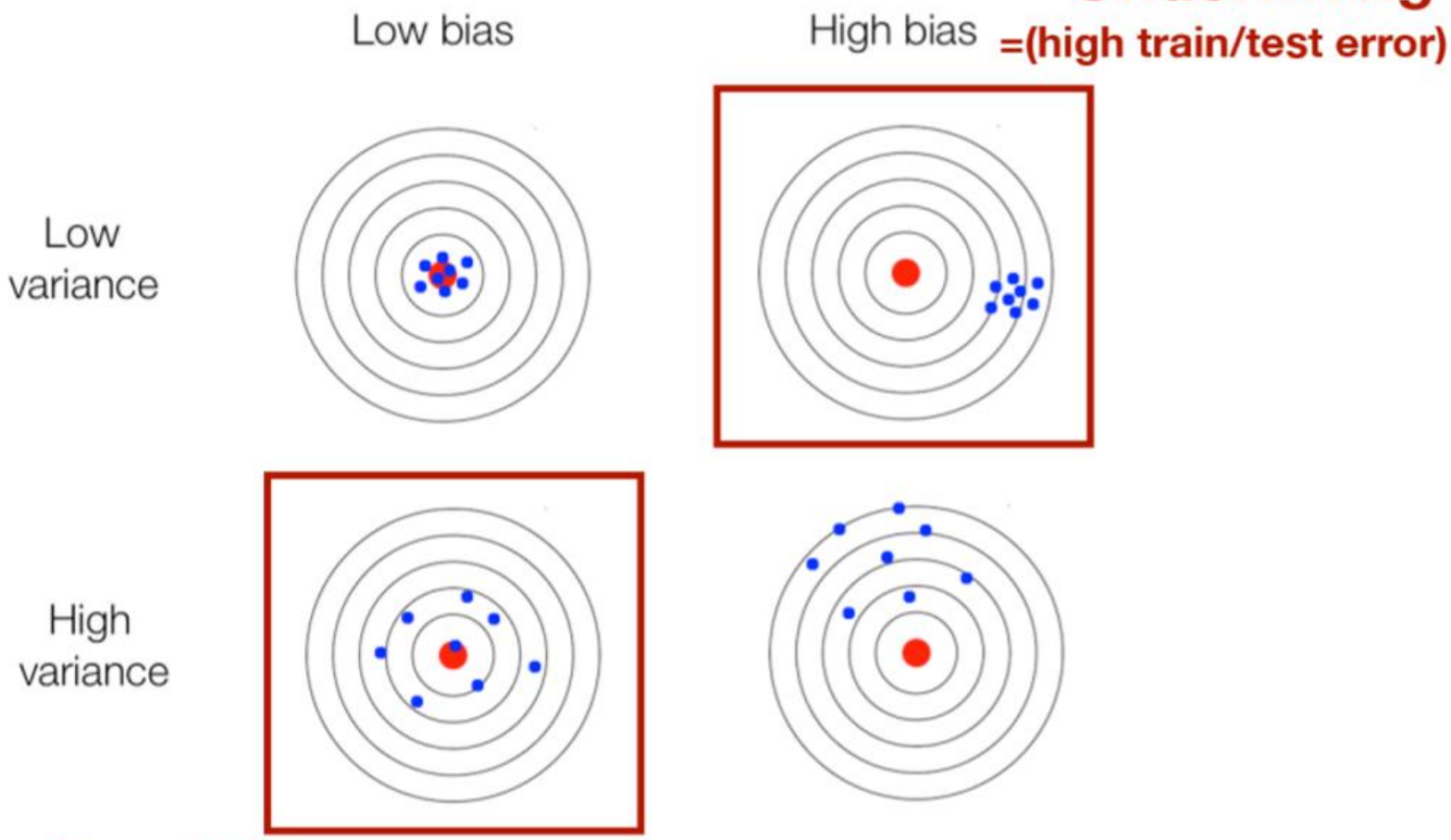


High variance



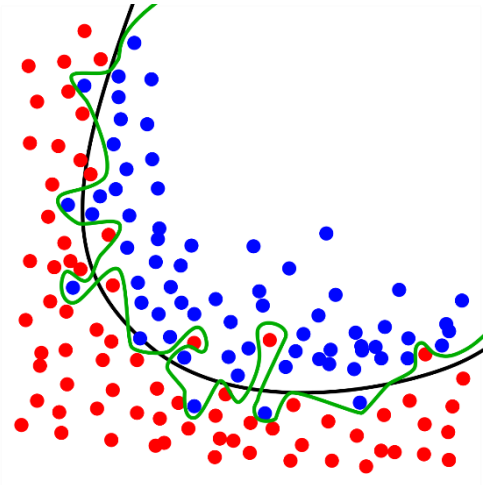


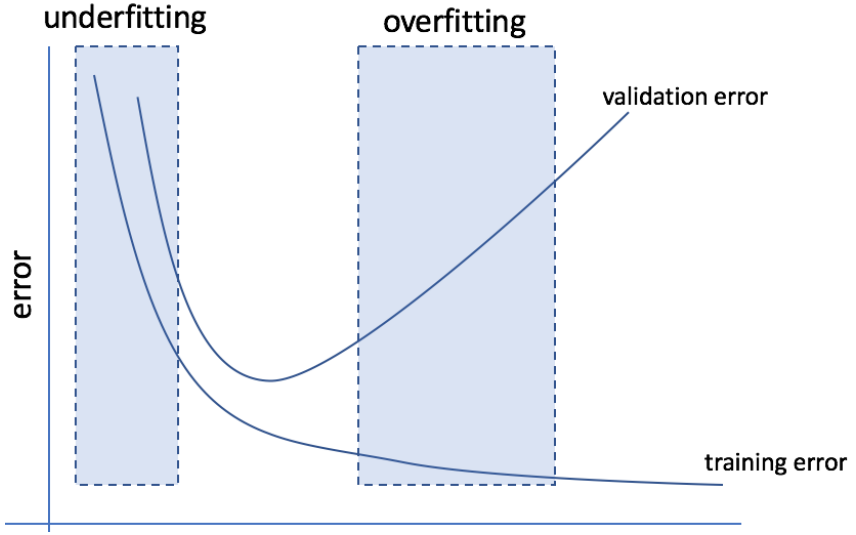
# Bias vs Variance



# Bias vs Variance

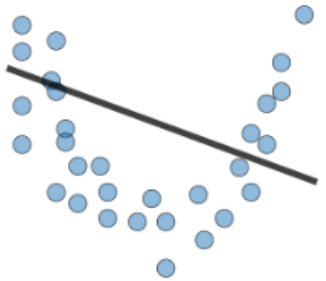
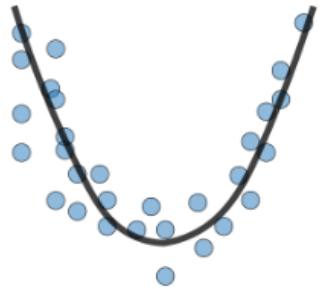
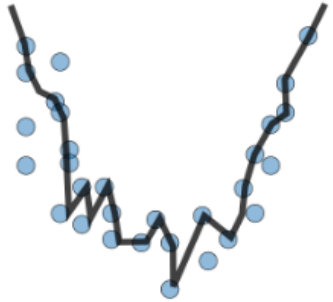
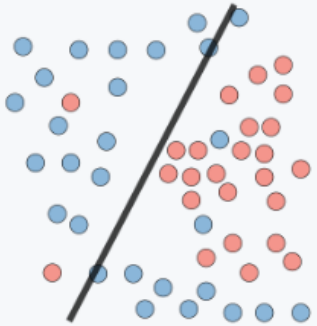
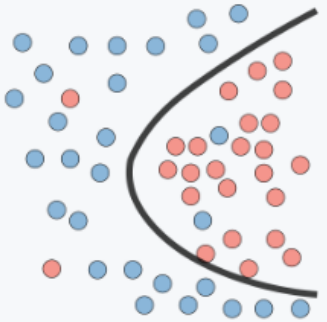
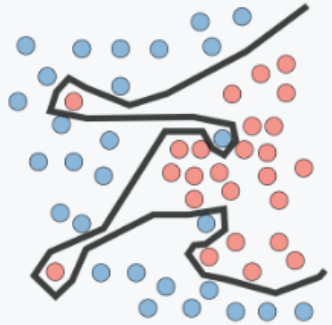
- 모델 복잡도가 커지면
  - 구불구불한 n차원 곡선으로 경계 구분함
  - Overfitting(과적합)이 일어남
  - 학습오차는 매우 적은 편이나, 일반화가 잘 안 되어 예측오차가 크다



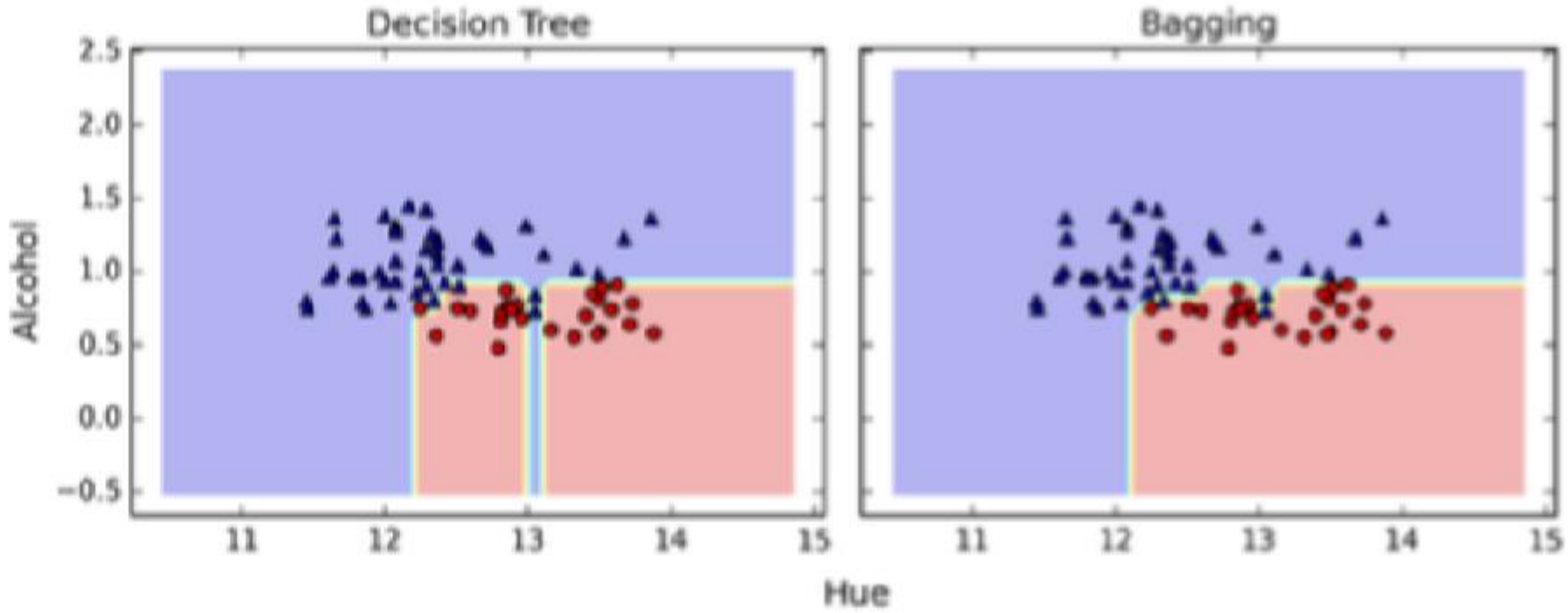


# Bias vs Variance

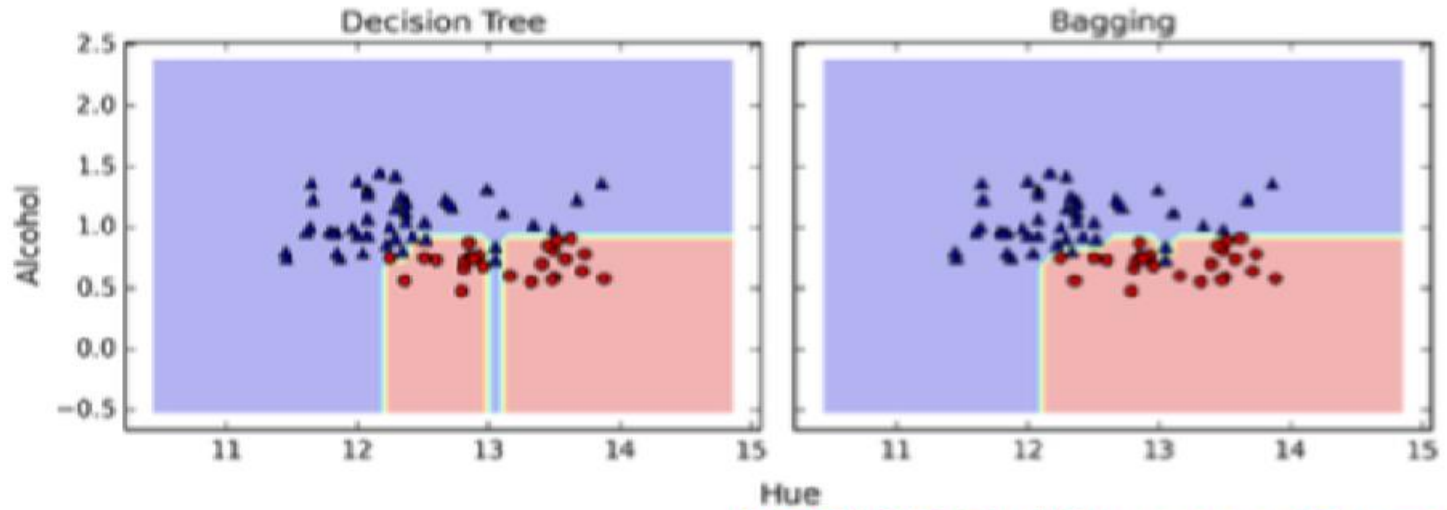
## 모델 복잡도가 커지면


| | Underfitting | Just right | Overfitting |
|---|---|---|---|
| 증상 | • 높은 학습 오류<br>• 테스트 오류에 가까운 학습 오류<br>• 높은 편향 | • 테스트 에러 보다 약간 낮은 학습 오류 | • 매우 낮은 학습 오류<br>• 테스트 오류보다 훨씬 낮은 학습 오류<br>• 높은 분산 |
| 회귀 일러스트레이션 |  |  |  |
| 분류 일러스트레이션 |  |  |  |
| 가능한 처리방법 | • 모델 복잡화<br>• 특징 추가<br>• 학습 증대 | | • 정규화 수행<br>• 추가 데이터 수집 |

# Bagging(+Random Forest)는 Variance를 감소시켜준다.



# Bagging(+Random Forest)는 Variance를 감소시켜준다.




반면에 앞으로 나올 boosting은 bias를 감소시켜준다.