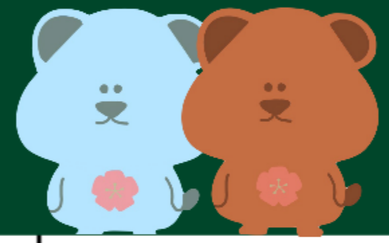
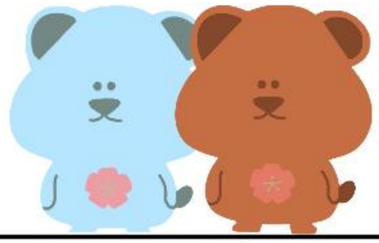


2주차 발표

DA팀 오수진 오연재 이서영

EWHA EURON

목차



EWHA EURON

01. 분류, 앙상블

EWHA EURON

1.1 지도학습 vs. 비지도학습

지도학습(Supervised Learning) vs. 비지도학습(Un-supervised Learning)

- **지도학습:** 정답이 있는(레이블 된) 데이터를 활용해 데이터를 학습시키는 것
  - 입력값(x)이 주어지면 입력값에 대한 label(y)를 주어 학습시키는 것
  - 예시) 고양이 사진(input data)을 주고, 이 사진은 고양이야(정답-label data)라고 알려주는 학습 방식

- **비지도학습:** 정답 레이블이 없는 데이터를 비슷한 특징끼리 군집화하여 새로운 데이터에 대한 결과를 예측하는 것
  - 지도학습에서 적절한 feature를 찾아내기 위한 전처리 방법으로 비지도 학습을 쓰기도 함
  - 라벨링 되어있지 않은 데이터로부터 패턴이나 형태를 찾아야 해서 지도학습보다는 조금 난이도가 있음
  - 예시) 고양이, 병아리, 호랑이 사진을 비지도학습 시키면 → (각 사진이 무슨 동물인지 정답(label)을 알려주지 않았기 때문에) 이 동물이 '무엇' 이라고 정의내릴 수 없지만, 비슷한 단위로 군집화 해줌.
    (다리가 4개인 고양이와 호랑이를 한 분류로, 다리가 4개이고 목이 긴 기린은 다른 분류,,, 나누어 놓을 것)

1.1 지도학습 vs. 비지도학습

지도학습(Supervised Learning) vs. 비지도학습(Un-supervised Learning)

지도학습(Supervised Learning)	비지도학습(Un-supervised Learning)
분류	클러스터링
회귀	차원 축소
추천 시스템	연관 규칙 학습
시각/음성 감지/인지	
텍스트 분석, NLP	

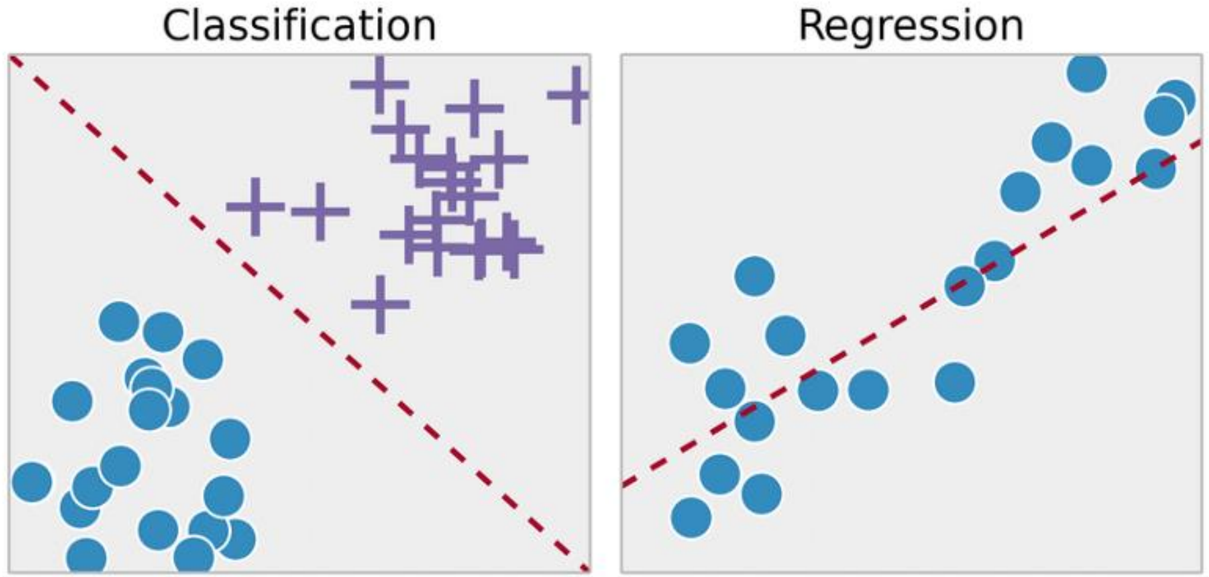
1.2 지도학습-분류, 회귀

지도학습 – 분류, 회귀

분류 : 주어진 데이터를 정해진 카테고리(label)에 따라 분류하는 문제	회귀 : 데이터들의 예측변수라 불리는 특성을 기준으로, 연속된 값(그래프)를 예측하는 문제
목표변수(y)가 범주형 변수	목표변수(y)가 수치형 변수
데이터를 다른 그룹으로 잘 분류했는지	값을 잘 예측하는 규칙을 만들어냈는지

분류
나이브 베이즈
로지스틱 회귀
결정트리
서포트벡터머신(SVM)
앙상블 학습
심층 신경망

<다양한 분류 알고리즘>



1.3 앙상블 학습

앙상블 학습
: 여러 개의 분류기를 생성하고, 그 예측을 결합하여 정확한 예측 도출하는 기법

- 강력한 하나의 모델을 사용하는 대신, 여러 개의 약한 모델 조합 → 더 정확한 예측에 도움
- 앙상블 학습 유형: 보팅(Voting), 배깅(Bagging), 부스팅(Boosting)
- 앙상블의 기본 알고리즘으로 일반적으로 사용되는 것은 결정트리(→ 여러 개의 결정 트리를 결합하여 하나의 결정 트리보다 더 좋은 성능을 냄)

결정 트리 - 장점	결정 트리 - 단점
쉽고 유연하게 적용가능한 알고리즘	과적합 문제
사전 가공의 영향 적음	

- 하지만, 과적합이 앙상블에서는 장점으로 적용됨.

  앙상블은 매우 많은 여러 개의 약한 학습기를 결합해 확률적으로 보완과 오류가 발생한 부분에 대해 가중치를 계속 업데이트 하면서 예측 성능을 향상시키는데, 결정 트리가 좋은 약한 학습기가 되기 때문!

02. 결정 트리

EWHA EURON

2.1 결정 트리

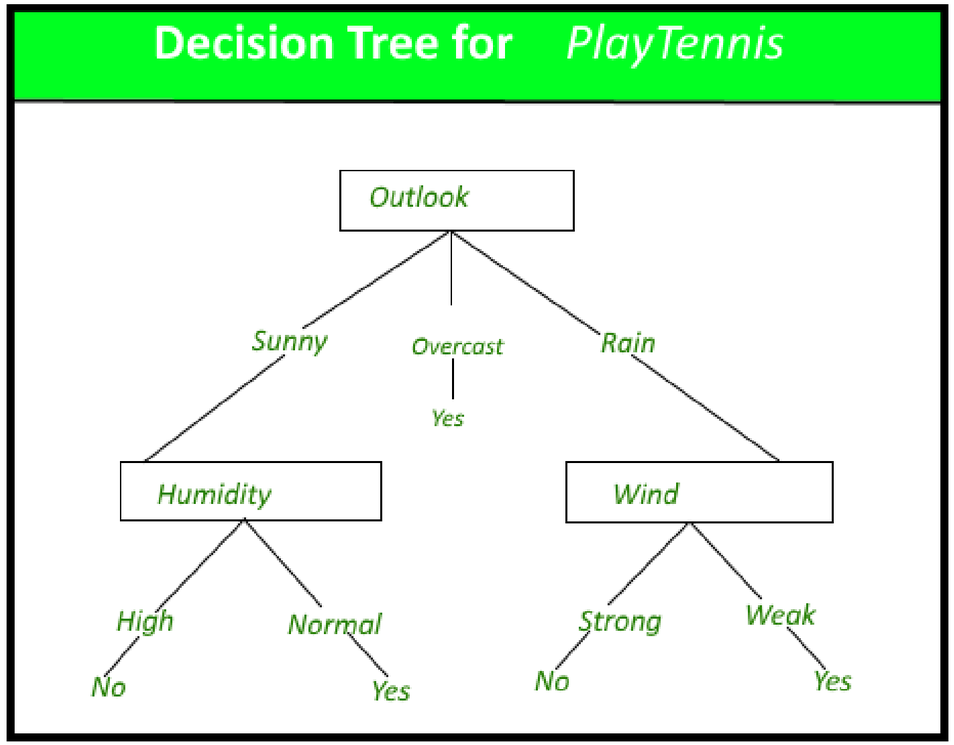
결정 트리란?

: 데이터에 있는 규칙을 학습을 통해 자동으로 찾아내 트리 기반의 분류 규칙을 만들어 레이블 분류하는 모델

- 결정 트리가 높은 예측 정확도를 가지려면 최대한 많은 데이터가 클래스에 속해야 함

트리를 분할하는 기준

- 균일도 : 여러 범주가 섞여 있는 정도 (불순도와 비슷한 개념)
- 같은 데이터가 많이 섞여 있으면 균일도가 높다고 말함
- 데이터를 구분하기 위해 필요한 정보의 양에 큰 영향을 줌



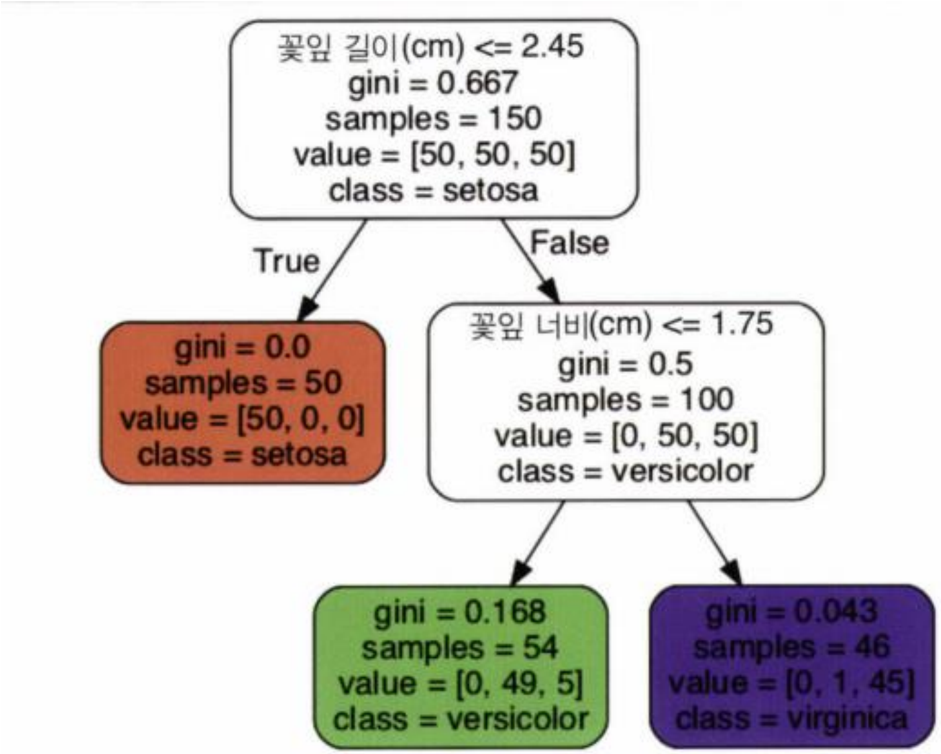
2.1 결정 트리

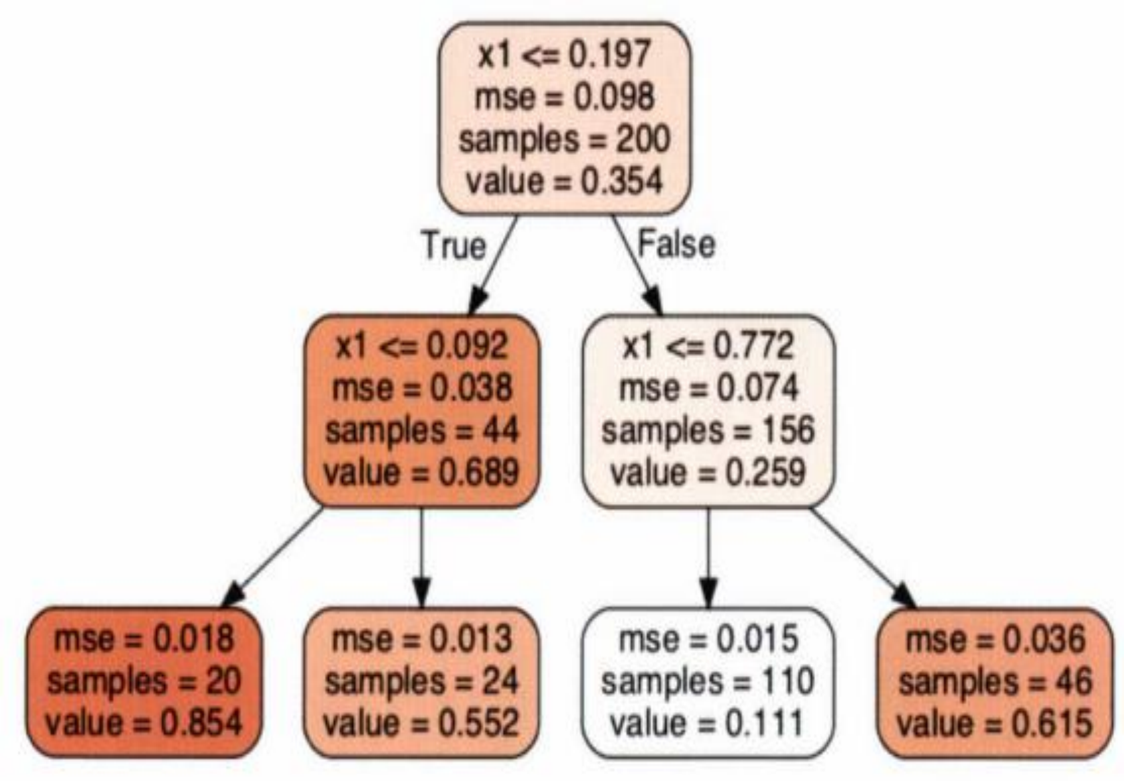
결정 트리 종류

DecisionTreeClassifier
: 클래스 분류 목적

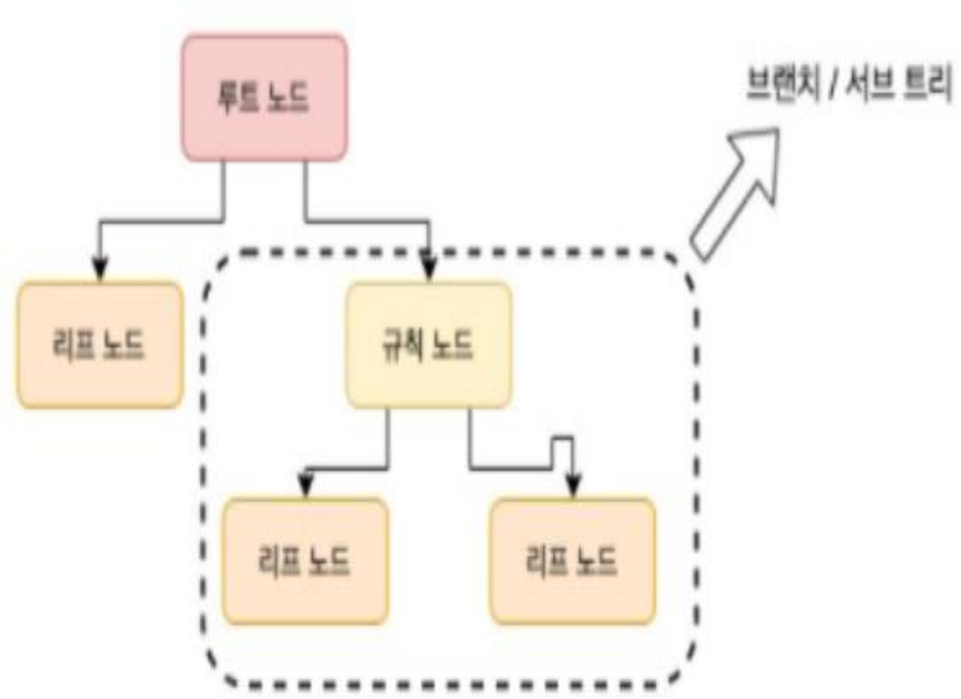
DecisionTreeRegression
: 분류 트리와 달리 각 노드에서 클래스 대신 값을 예측

- 예측 값과 가능한 많은 샘플이 가까이 있도록 분할
- 클래스나 지수 대신 MSE(평균 제곱 오차)





2.2 결정 트리 구조



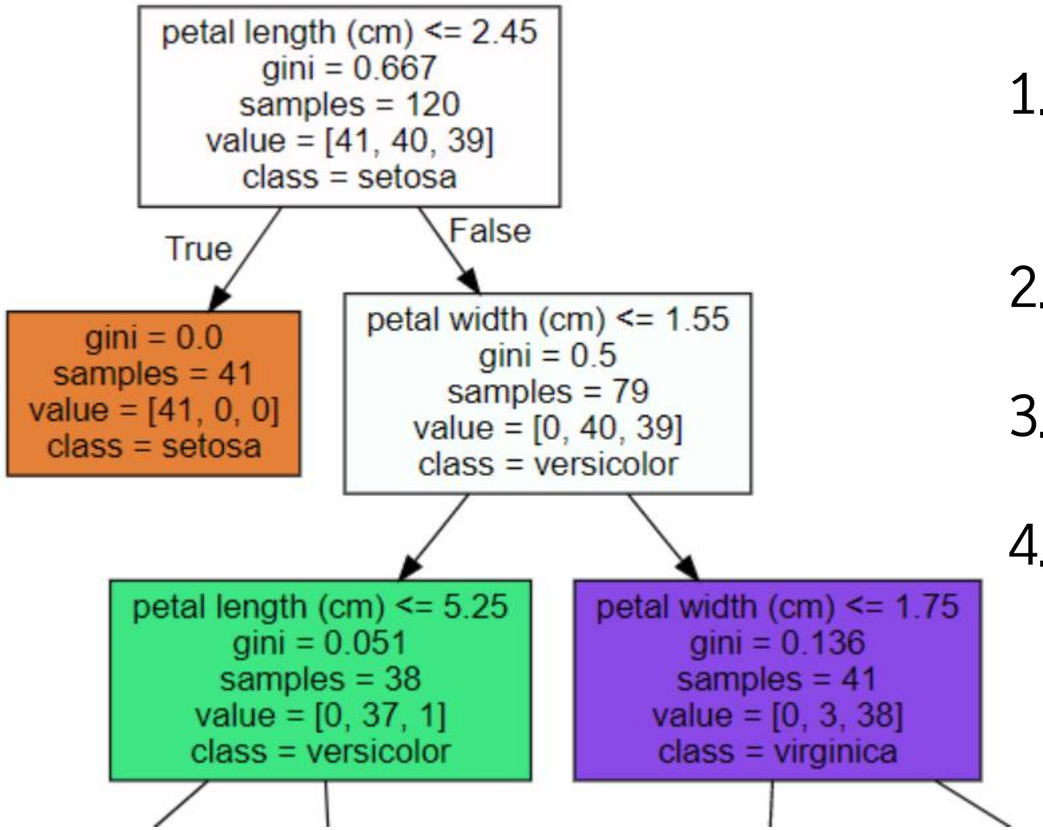
루트 노드 : 깊이가 0인 맨 위의 노드
규칙 노드 : 규칙 조건이 포함된 노드, 자식 노드로 분할됨
리프 노드 : 분류 값이 결정된 노드, 자식 노드가 없음
브랜치/서브 트리 : 새로운 규칙 조건이 생길 때마다 생성됨

** 트리의 깊이가 깊어질수록(피처가 많고 균일도가 다양할수록) 결정 트리의 예측 성능이 저하될 가능성 높음

2.2 결정 트리 구조

생성 과정

: 데이터 세트의 모든 값이 같은 분류라면 리프 노드로, 아니라면 규칙에 따라 분할하여 브랜치 노드 생성



1. 루트 노드에서 'petal length <= 2.45'가 규칙 조건에 해당. 규칙 조건을 만족하는/불만족하는 데이터가 각각 두 노드로 분리
2. 주황색 노드를 보면 규칙 조건이 없고 데이터 세트의 모든 값이 같은 분류이기 때문에 리프 노드, 분할 x
3. 다시 데이터를 분할하는데 좋은 기준/속성 찾아 분할하여 브랜치 노드 생성
4. 데이터가 모두 분류에 속할 때까지 과정 반복

2.3 균일도

균일도 측정

: 결정 노드는 정보 균일도가 높은 데이터 세트를 먼저 선택하도록 규칙 조건을 만듦

1) 정보이득 지수
   - 데이터의 혼잡도인 엔트로피를 이용
   - 같은 값이 섞여 있으면 혼잡도가 낮으니 엔트로피도 낮음
   - 정보이득지수 = 1-엔트로피 지수 (높을수록 균일)
   - 정보 이득이 높은 속성 기준으로 분할

$$H_i = -\sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^{n} p_{i,k} \log_2(p_{i,k})$$

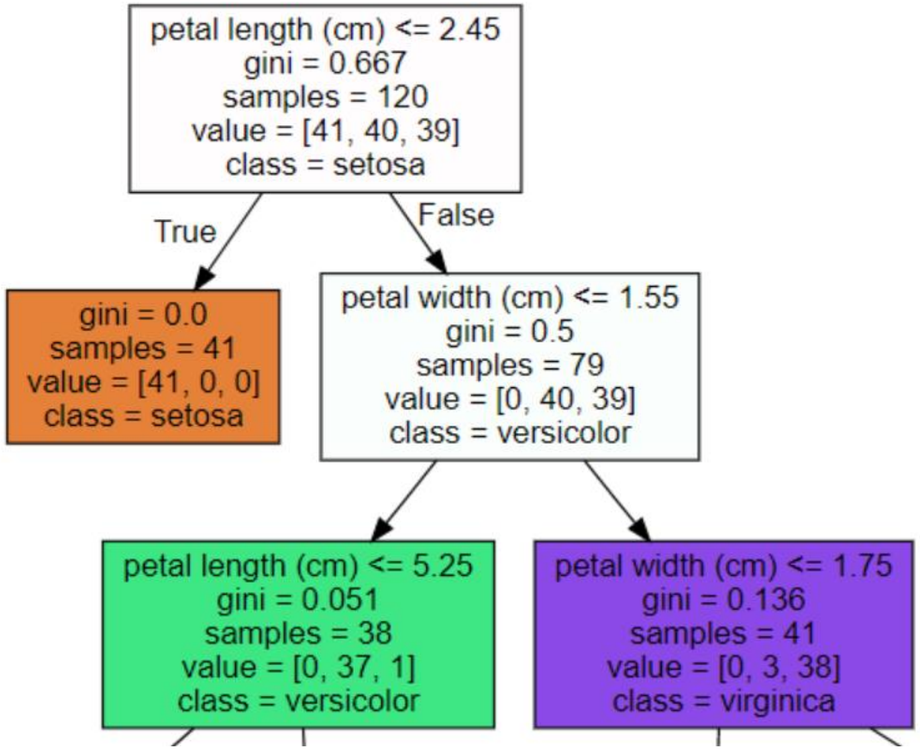
2) 지니 계수
   - 경제학에서 유래
   - 모든 샘플이 같은 클래스에 있다면 0(순수/평등), 1로 갈수록 불평등
   - 낮을수록 균일하므로 지니 계수가 낮은 속성 기준 분할
   - DecisionTreeClassifier에서 사용

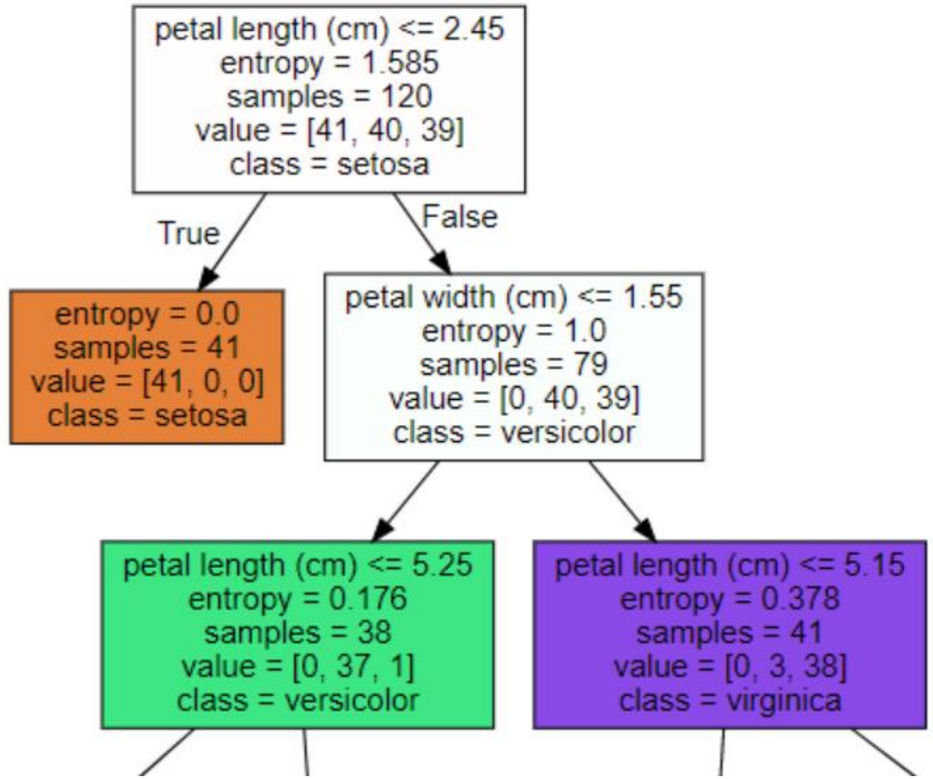
$$G_i = 1 - \sum_{k=1}^{n} {p_{i,k}}^2$$

** p(i,k)는 i번째 노드에 있는 훈련 샘플 중 클래스 k에 속한 샘플 비율

** 지니 계수는 계산이 빠르고, 엔트로피는 균형 잡힌 트리를 만든다는 장점 보유

2.3 균일도





** DecisionTreeClassifier의 criterion 매개변수를 "entropy"로 지정하면 지니 계수 대신 엔트로피를 기준으로 한 결정 트리를 볼 수 있음
** 두 보라색 노드를 보면 규칙 조건이 다르고, 균일도 기준 역시 지니와 엔트로피로 다르게 명시되어 있음

2.4 결정 트리 특징

결정 트리 장단점

1) 장점
- 알고리즘 쉽고 직관적
- 시각화 가능 -> 분류 과정 파악 쉬움
- 균일도만 주로 신경쓰기 때문에 전처리 작업 불필요
- 회귀와 분류, 수치형과 범주형 데이터 모두 가능

2) 단점
- 과적합으로 인한 정확도 감소
- 피처 많고, 균일도 낮을수록 트리의 깊이가 커져 성능 저하

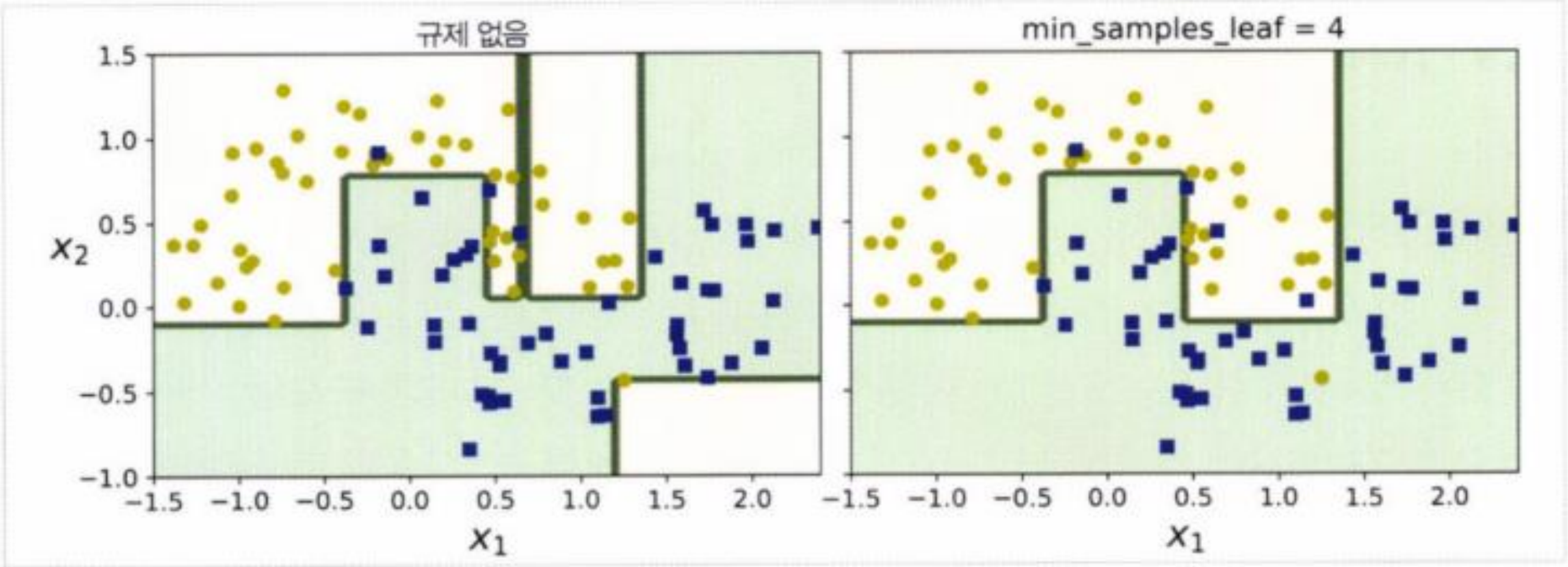
3) 해결 방법
- 튜닝을 통해 트리의 크기 사전 제한

2.5 결정 트리 파라미터

min_samples_split	- 노드를 분할하기 위한 최소 샘플 데이터 수 - 디폴트 2, 작을수록 과적합의 가능성 - 과적합 제어 용도
min_samples_leaf	- 말단 노드가 되기 위한 최소 샘플 데이터 수 - 과적합 제어 용도 - 비대칭적 데이터 → 특정 클래스 데이터가 극소일 가능성 존재하므로 작게 설정
max_features	- 최적의 분할을 위한 최대 피처 개수 - 디폴트 none (전체 피처 사용) - int형→ 피처 개수, float형→ 퍼센트 - sqrt, log, auto 등
max_depth	- 최적의 분할을 위한 최대 피처 개수 - 디폴트 none (전체 피처 사용) - int형→ 피처 개수, float형→ 퍼센트 - sqrt, log, auto 등
max_leaf_nodes	- 말단 노드의 최대 개수

** min_으로 시작하는 매개변수를 증가시키거나 max_로 시작하는 매개변수를 감소시키면 규제가 커짐

2.6 과적합 (Overfitting)



- moons 데이터셋
- 이상치 데이터까지 분류하기 위해 결정 기준 경계가 복잡해져 과적합된 것을 확인 가능
- 리프 생성 규칙을 완화하는 코드를 통해 결정 기준 경계가 더 일반화되도록 만들기
- 테스트 데이터는 학습 데이터와 다른 데이터 세트일 것이기 때문에 학습 데이터에 지나치게 최적화된 모델은 테스트에서의 정확도를 떨어트림

2.7 CART 훈련 알고리즘

: 매번 CART 비용 함수가 최소가 되도록 훈련 세트를 분할
- 사이킷런에서 결정 트리 훈련을 위해 사용
- 양쪽을 깔끔하게(가장 순수하게) 분리해낸 정도를 수치화한 것
- 탐욕적 알고리즘(좋은 솔루션 제공하지만 최적 솔루션 보장 X)

식 6-2 분류에 대한 CART 비용 함수

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

여기서 $G_{\text{left/right}}$는 왼쪽 /오른쪽 서브셋의 불순도

$m_{\text{left/right}}$는 왼쪽 /오른쪽 서브셋의 샘플 수

- 불순도는 지니 계수 혹은 엔트로피 이용
- 최대 깊이가 되거나 불순도를 줄이는 분할을 찾을 수 없을 때 중지
- max_depth 외에도 min_samples_split, min_samples_leaf, min_weight_fraction_leaf, max_leaf_nodes 등의 파라미터를 통해 중지 조건 설정 가능

2.7 CART 훈련 알고리즘

CART 훈련 알고리즘(Classification and Regression Tree)

: 매번 CART 비용 함수가 최소가 되도록 훈련 세트를 분할

- 불순도 대신 평균제곱오차를 최소화하도록 분할

식 6-4 회귀를 위한 CART 비용 함수

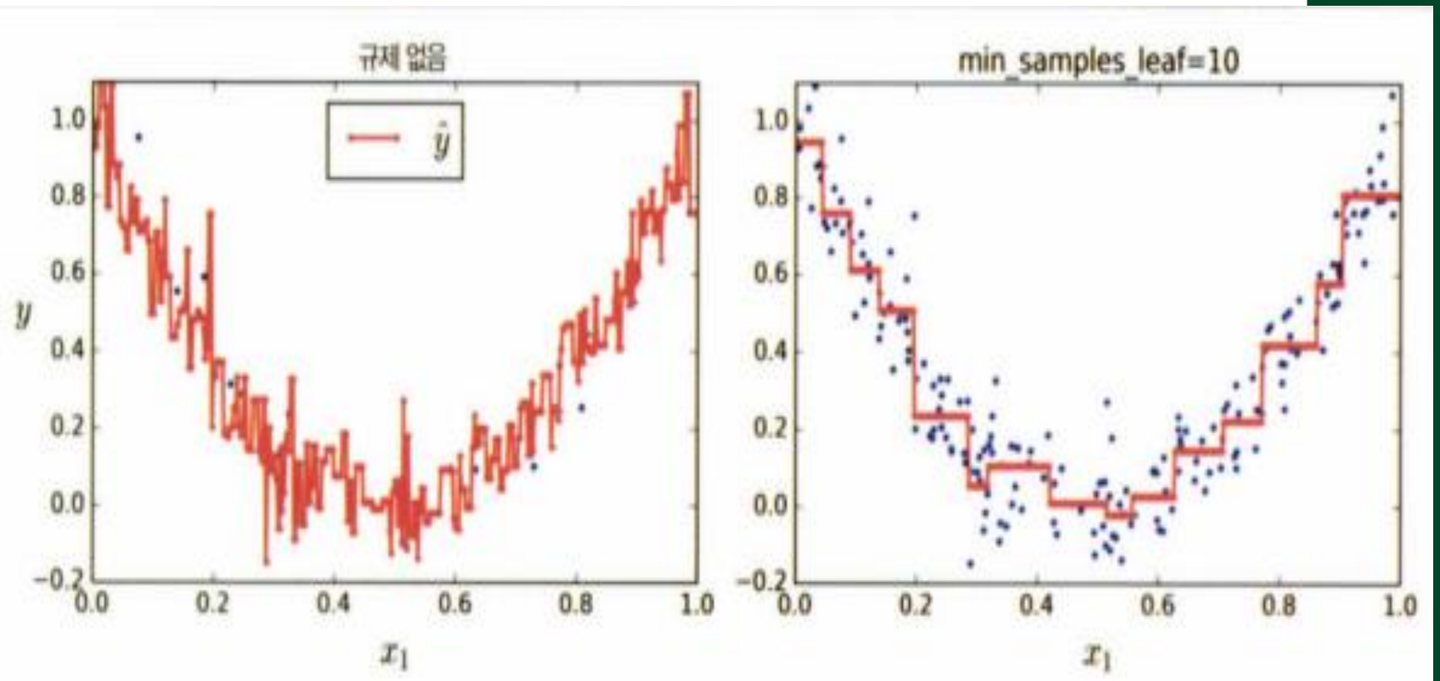
$$J(k, t_k) = \frac{m_{\text{left}}}{m}\text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m}\text{MSE}_{\text{right}}$$

여기서

$$\text{MSE}_{\text{node}} = \sum_{i \in \text{node}} \left(\hat{y}_{\text{node}} - y^{(i)}\right)^2$$

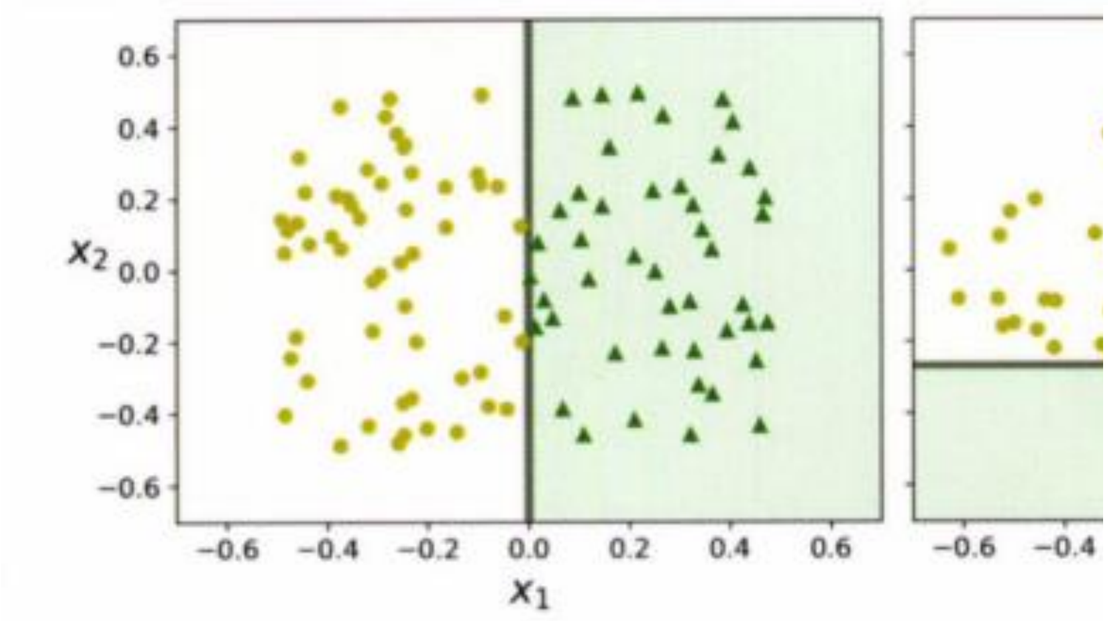
$$\hat{y}_{\text{node}} = \frac{1}{m_{\text{node}}} \sum_{i \in \text{node}} y^{(i)}$$

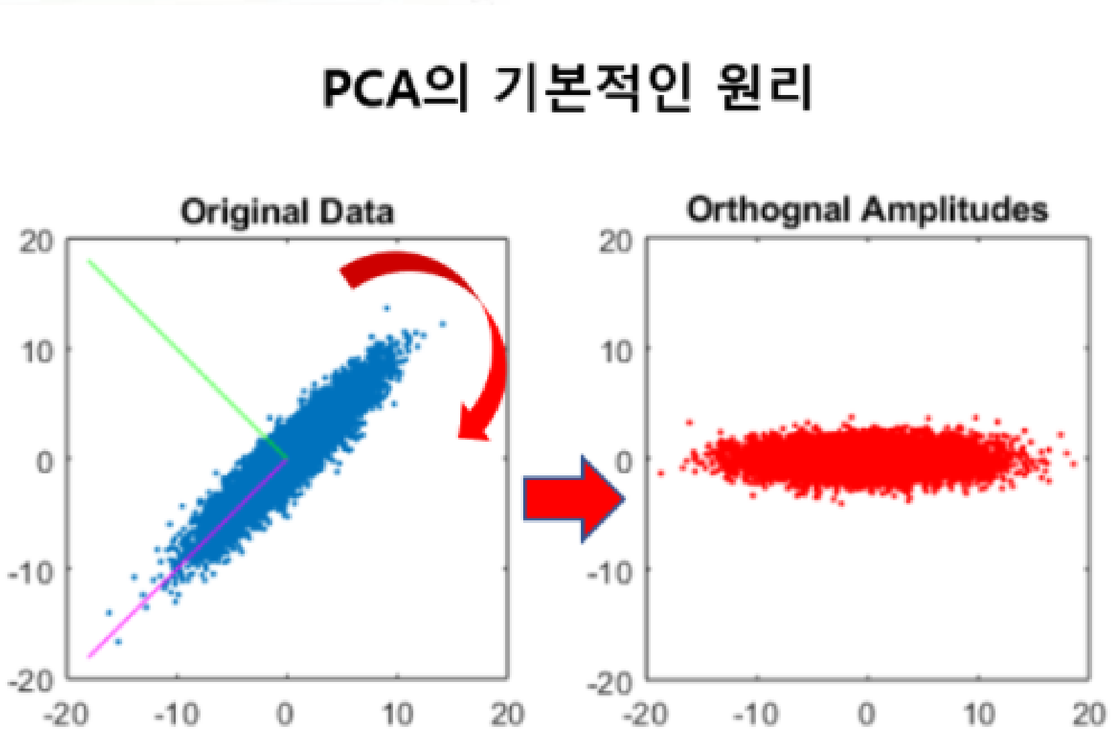
- 회귀 트리에서도 과적합되기 쉬우므로 모델 규제 필요



2.8 불안정성

- 결정 트리는 해석이 쉽고, 예측 속도나 성능이 좋은 모델이지만 계단 모양의 결정 경계를 만들기 때문에 훈련 세트의 회전에 민감





데이터 회전을 통한 특성 축소

- 왼쪽과 다르게 회전한 오른쪽 그림의 결정 트리는
- 이와 같은 문제 해결을 위해 PCA 사용

2.9 결정 트리 실습 순서

교재의 사용자 행동 인식 데이터 참고

1) 데이터 전처리
2) train, test 데이터 가져오기
3) 결정트리 학습/예측/평가
4) 결정트리 개선 : GridSearchCV를 이용해서 여러 파라미터로 시도 -> 베스트 모델 찾기
5) feature_importances_ 이용하여 각 피처의 중요도 확인

03. 앙상블 학습

EWHA EURON

3.1 앙상블과 정형 데이터

앙상블의 개념

여러 개의 분류기(classifier)를 생성하여 예측을 결합하여 정확한 최종 예측을 도출하는 기법. (집단지성)

대표적으로, voting, bagging, boosting이 존재한다.

정형 데이터 분류

이미지, 영상, 음성 등의 비정형 데이터(딥러닝에 뛰어남), 대부분의 정형 데이터(앙상블이 뛰어남)
최신의 앙상블 모델인 XGboost, LightGBM을 알면 정형 데이터의 분류, 회귀 분야의 예측에
뛰어난 모델 가능.

3.2 보팅 방식 / 배깅 방식

보팅(voting) 방식 / 배깅(bagging) 방식

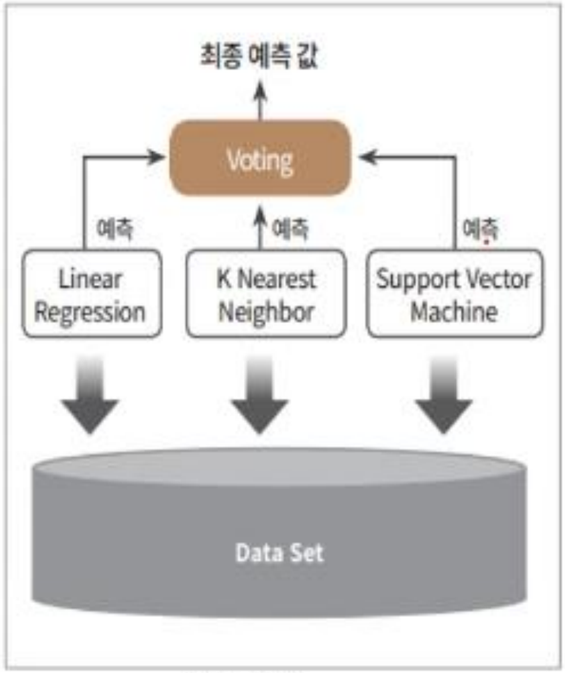
보팅(voting) 방식: 서로 다른 알고리즘을 가진 분류기를 결합, 같은 데이터 세트에 대해 학습

배깅(bagging) 방식: 같은 유형 기반의 알고리즘을 가진 분류기를 결합, 모두 다른 데이터 샘플링에 대해 학습.

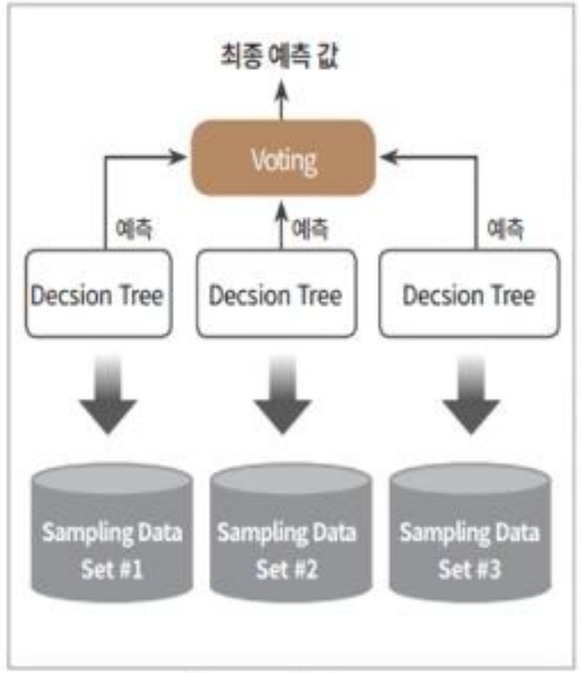
Ex) random forest

+) bootstrapping 분할 방식: 배깅 방식에서 개별 classifier에게 개별적인 데이터를 추출하는 방식.

+) 배깅 방식은 데이터의 중첩을 허용



Voting 방식



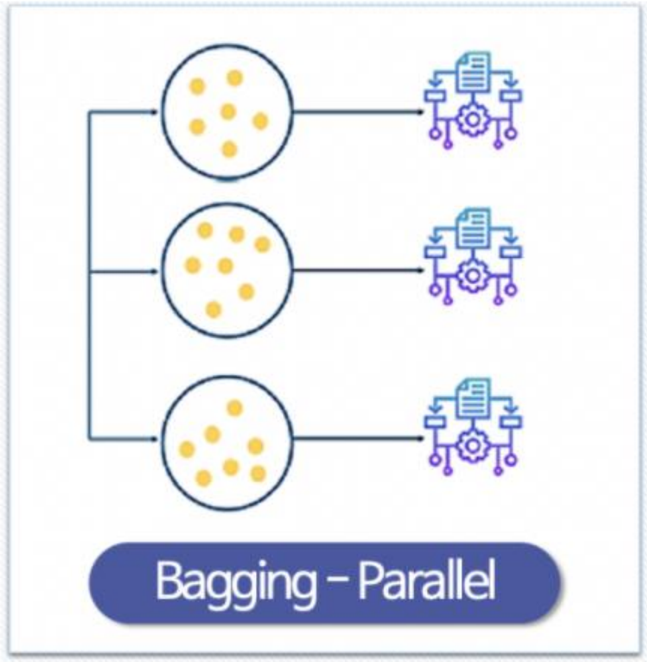
Bagging 방식

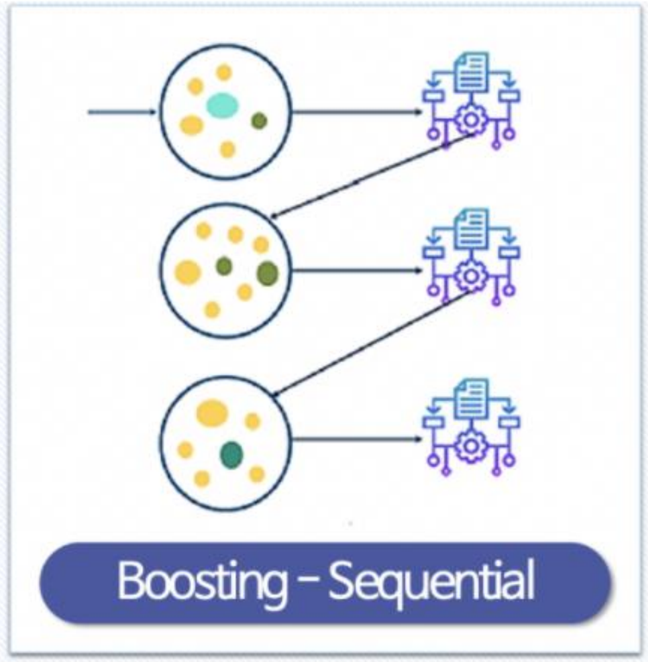
추가적으로…
부스팅 방식: 여러 개의 분류기가 순차적으로 학습,
앞의 예측이 틀리면 다음 분류기에 가중치를 부여
Ex) 그래디언트 부스트, XGBoost, LightGBM

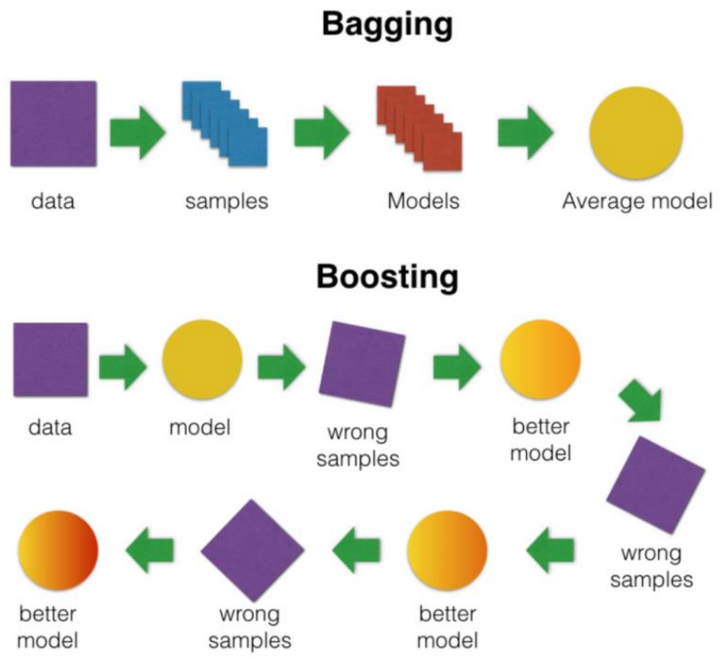
3.3 배깅 방식 / 부스팅 방식

부스팅 방식: 여러 개의 분류기가 순차적으로 학습, 앞의 예측이 틀리면 다음 분류기에 가중치를 부여
Ex) 그래디언트 부스트(경사하강법), Ada boost: 약한 학습기를 이용

배깅 방식은 한 번에 병렬적으로 결과를 얻고, 부스팅은 순차적으로 진행된다.





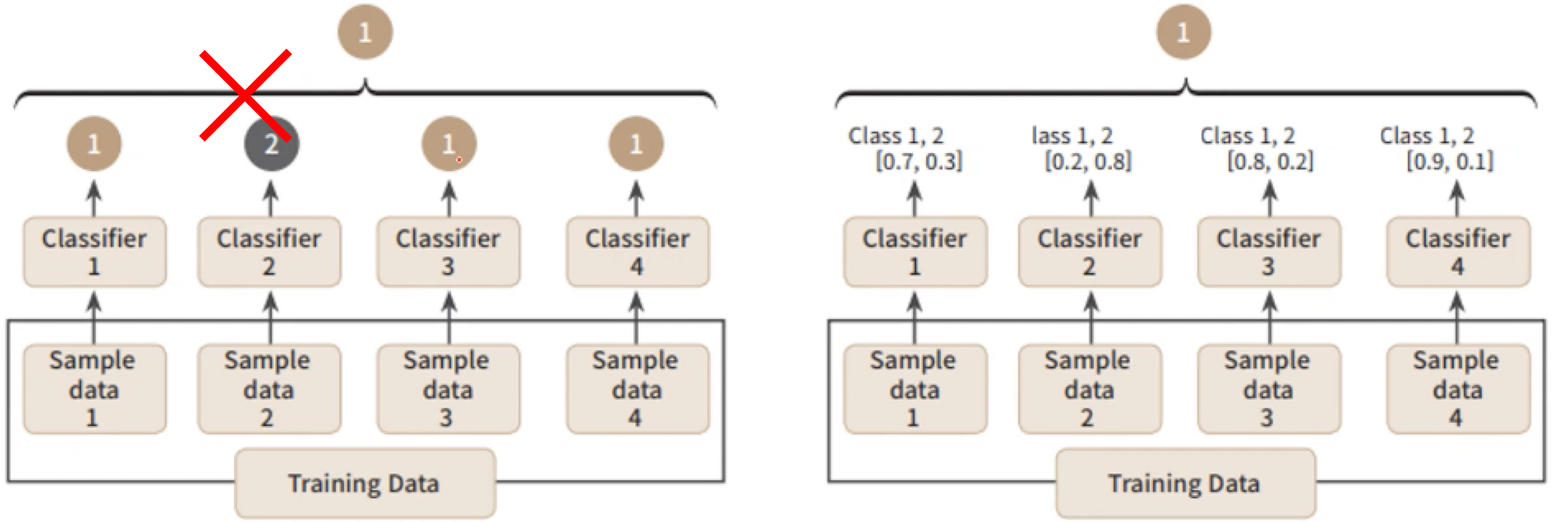


3.4 하드 보팅 / 소프트 보팅

Hard voting / soft voting

Hard voting: 다수결의 원칙과 비슷. 여러 개의 분류기가 낸 결괏값 중에서 많은 것을 선택

Soft voting: 분류기들의 레이블 값을 더하여 평균을 내어 가장 높은 레이블 값을 선택, 일반적으로 soft voting이 보팅 방법으로 적용된다. (일반적으로 소트프 보팅이 예측 성능이 더 좋다.)



좌) 하드 보팅
우) 소프트 보팅

3.5 보팅 분류기 작동(소프트 보팅)

보팅 분류기 작동해보기(소프트 보팅)

1) 필요한 모듈과 데이터 로드 필요한 모듈과 데이터 로드

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

```
cancer=load_breast_cancer()
```

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

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness
0	17.99	10.38	122.8	1001.0	0.11840	0.27760
1	20.57	17.77	132.9	1326.0	0.08474	0.07864
2	19.69	21.25	130.0	1203.0	0.10960	0.15990

3 rows × 30 columns

2) estimator: 보팅에 사용될 객체들, voting: 보팅 방식 선택(디폴트: hard)

```
# 개별 모델은 로지스틱 회귀와 KNN임
lr_clf=LogisticRegression()
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)
```

3) 정확도 산출

```
# Voting CLassifier 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: 0.947368
LogisticRegression 정확도: 0.9386
KNeighborsClassifier 정확도: 0.9386
```

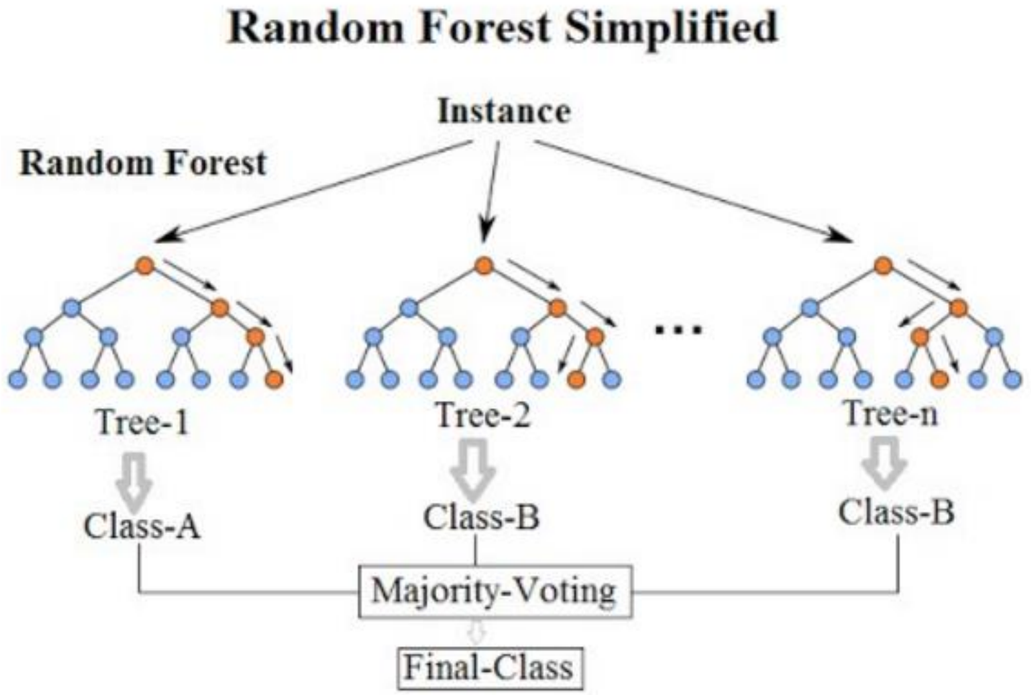
04. 랜덤 포레스트

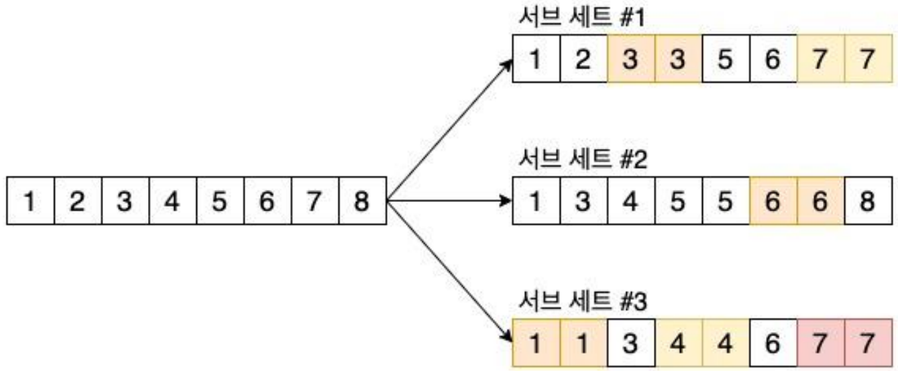
EWHA EURON

4.1 랜덤 포레스트

랜덤 포레스트란…

배깅에 속하며 같은 알고리즘을 여러 개 이용하고 다양한 데이터 셋을 이용(중첩 발생).
개별적인 분류기의 기반은 겨정 트리이지만 개별 트리가 학습하는 데이터 세트는 전체 중
일부가 중첩되게 샘플링 된 것. (부트스트래핑 방식)





n_estimator=30이라고 하면 3개의 결정 트리 기반으로 학습하며
위와 같이 3개의 subset을 생성된다.

서브세트 데이터 건수는 전체 데이터 건수와 동일하지만 중첩되어
만들어진다

4.2 랜덤 포레스트 – 하이퍼 파라미터

랜덤 포레스트 - 하이퍼 파라미터 및 튜닝

하이퍼 파라미터(튜닝으로 인한 시간 오래 걸림)가 너무 많다. 또한 튜닝 후 예측 성능이 크게 향상되지 않기도 한다.

n_estimator: 결정 트리의 개수를 지정(디폴트는 10개, 무조건 많다고 좋은 것이 아님)

max_features: sqrt이고 전체 피처가 16개 이면 분할을 위해 4개를 참조한다.

max_depth, mins_smaples_leaf는 과적합 방지.

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

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

```
랜덤 포레스트 정확도: 0.9108
```

```
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: 0.9168
```

```
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
```

EWHA EURON

4.2 랜덤 포레스트 – 하이퍼 파라미터

랜덤 포레스트 - 하이퍼 파라미터 및 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: 0.9168
```

```
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
```

GridSerachCV를 이용하여 파라미터를 튜닝

여러 parameters를 수정하면서 예측 정확도를 구해본다.

n_estimators:100 로 설정 후, 지정한 parameters
중, 최고 예측 정확도를 나타내는 조건을 구한다.

평균 정확도: 91.68%

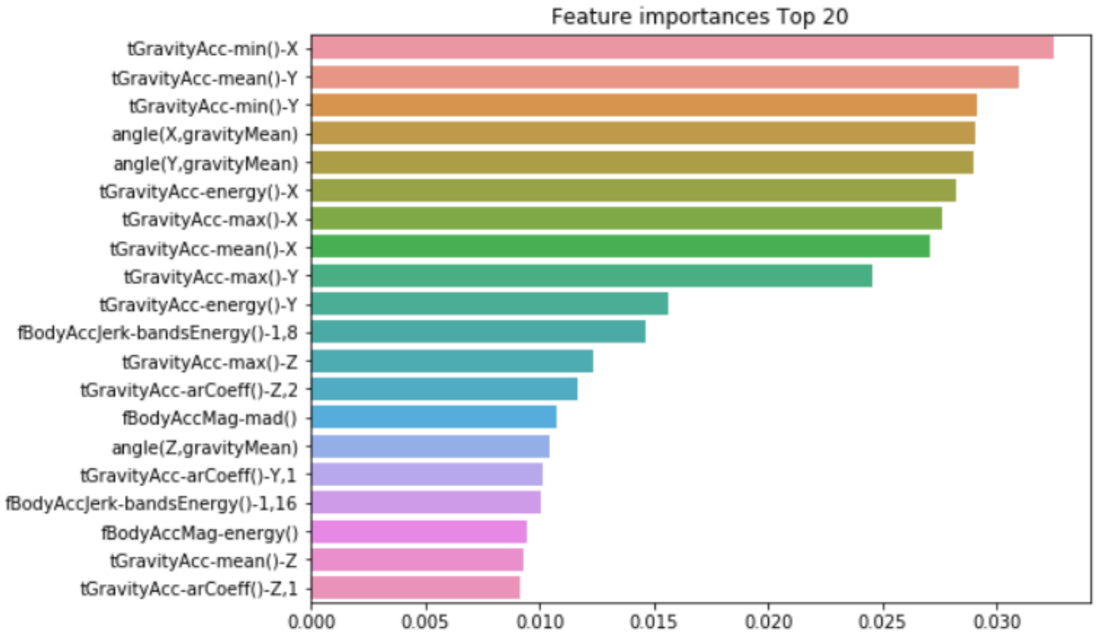
→ n_estimators: 300 로 변경하고 최적화 하이퍼 파라미터로 다시 성능을 측정한다.

정확도: 91.65%

4.3 랜덤 포레스트 – feature importance

랜덤 포레스트 – feature importance

```
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()
```



몇 개의 피처가 명확한 규칙 트리를 만드는 데 크게 기여하는지 확인

값이 높을수록 중요도가 높다는 의미

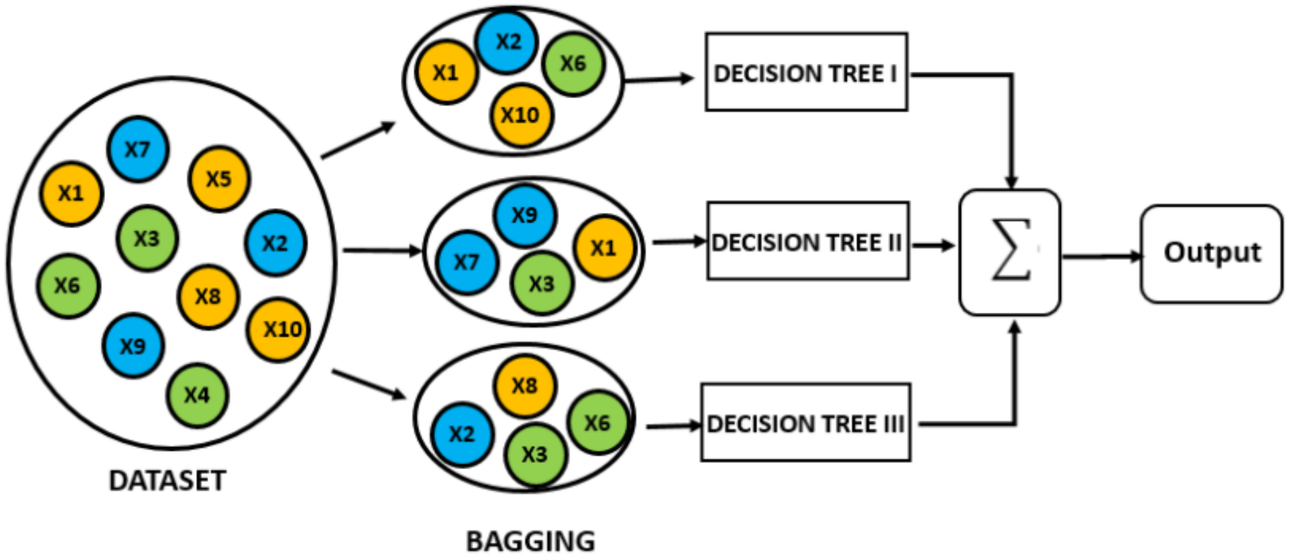
4.4 랜덤 포레스트 vs 배깅

랜덤포레스트, 배깅 알고리즘

Bootstrap: 전체 데이터에서 일부가 중첩되게 샘플링된 데이터 세트로 분할, 모두 동일한 알고리즘으로 이루어진 분류기를 사용한다.

Aggregating: Bootstrap에서 나온 결과를 통합한다.

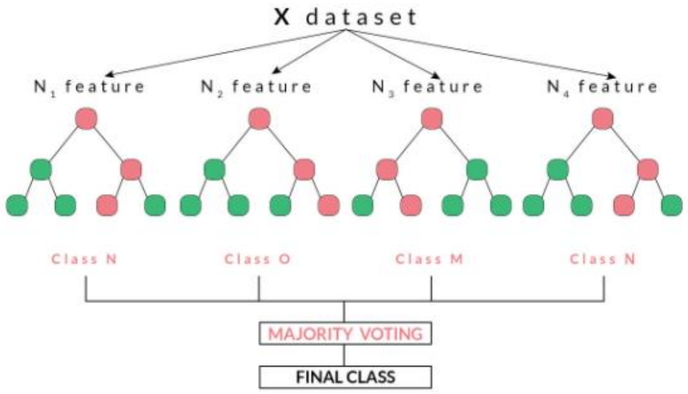
→ 최종적으로 모든 분류기가 보팅을 통해 예측 결정을 한다.

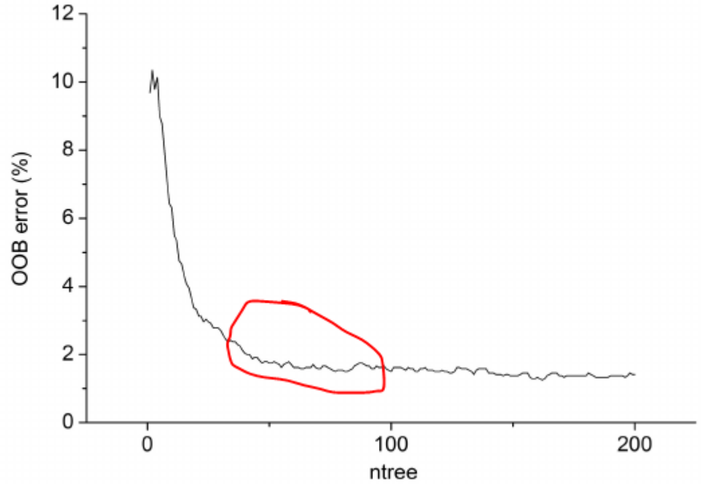


4.4 랜덤 포레스트 vs 배깅

랜덤포레스트, 배깅 알고리즘

여러 의사결정나무를 생성한 후에 다수결 또는 평균에 따라 출력 변수를 예측하는 알고리즘
의사결정나무와 bagging을 혼합한 형태임
부트스트랩을 이용하여 학습 집합에서 다양한 샘플을 추출함
입력 변수 중 일부의 입력 변수만 사용





- **트리의 개수(ntree)가 많을 수록 정확도는 개선되지만 elbow point(기울기가 급변하는 지점)가 존재**
  : elbow point 지점 이상 개수의 트리까지는 생성할 필요가 없음 (불필요한 트리 생성으로 인한 알고리즘 성능 저하)

출처: https://neocarus.tistory.com/entry/Random-forest-작동-원리 [Passive Incomer's Active Life]

4.5 랜덤 포레스트 회귀

+) 랜덤 포레스트 회귀

의사 결정 트리의 과적합을 줄이고 정확도를 향상시킨다.
범주형 값과 연속형 값 모두에서 잘 작동한다.
결과를 결합하기 위해 여러 의사 결정 트리에 적합하므로 많은 계산 능력과 리소스가 필요하다.

Import RandomForestRegressor
Import train_test_split
Import mean_absolute_error

X, y값을 지정 → train_test_split(X, y) → model.fit(X=x_train, y=y_train)

y_pred=model.predict(x_test)

MSE와 RMSE를 통해 오차를 구한다. (오차가 작을수록 좋음)

MSE

$$\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2$$

L2-norm

회귀 모델의 주요 손실함수

예측값과 실제값의 차이인 오차들의 제곱 평균으로 정의한다.

제곱을 하기 때문에 특이치(아웃라이어)에 민감하다.

RMSE

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}$$

MSE에 root를 씌운 값

참고: Regression - 모델 평가 : MSE, MAE, RMSE, RMSLE, R-Squared (tistory.com)
회귀분석의 네가지 방법, 선형회귀/의사결정트리/랜덤포레스트/SVM – GIS Developer
랜덤포레스트 (Random Forest) (tistory.com)

```
from sklearn.ensemble import RandomForestRegressor # 회귀트리(모델)
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_california_housing, load_boston
from sklearn.metrics import mean_absolute_error
import numpy as np

X, y = load_boston(return_X_y=True)     # return : X, y값 지정

boston = load_boston()
X = boston.data
y = boston.target

colnames = boston.feature_names
```

```
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

```
model = RandomForestRegressor()
model.fit(X = x_train, y = y_train)
```

```
RandomForestRegressor()
```

```
y_pred = model.predict(x_test)

mse = mean_absolute_error(y_test, y_pred)
print('평균제곱근 오차 mse: ', mse)
rmse = (np.sqrt(mse))
print('평균 제곱근 편차 rmse: ', rmse)
```

```
평균제곱근 오차 mse:  2.295671052631579
평균 제곱근 편차 rmse:  1.5151472049380479
```

05. SVM(서포트 벡터 머신)

EWHA EURON

5.1 SVM(서포트 벡터 머신)

SVM(서포트 벡터 머신)

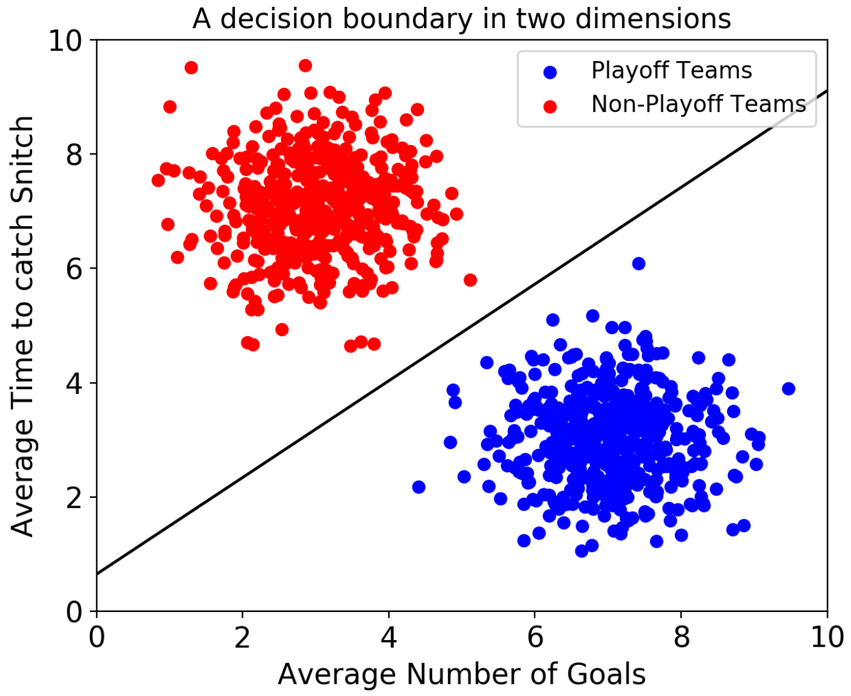
결정 경계(Decision Boundary), 즉 분류를 위한 기준 선을 정의하는 모델

→ 분류되지 않은 새로운 점이 나타나면 경계의 어느 쪽에 속하는지 확인해서 분류 과제를 수행하게 됨.

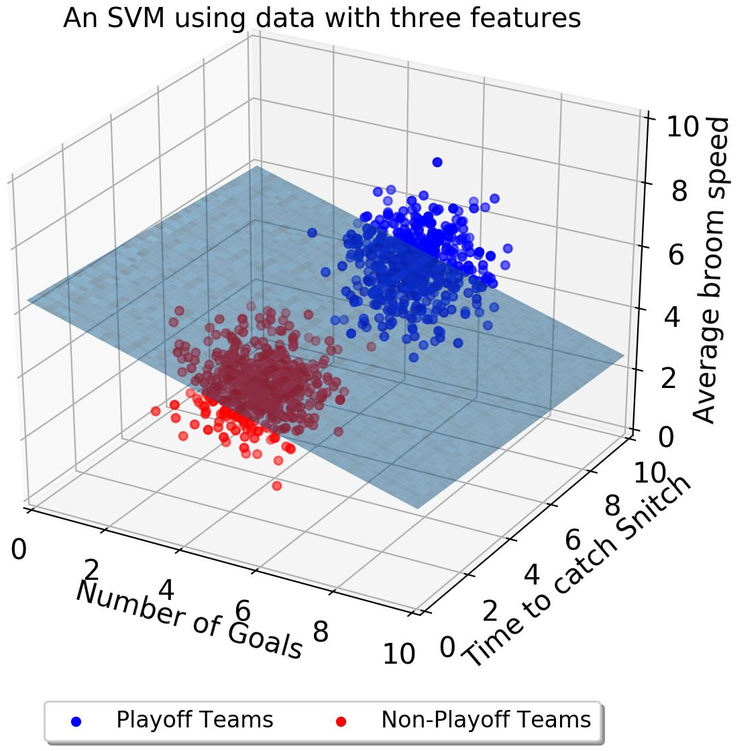
- 복잡한 분류 문제에 잘 맞음
- 작거나 중간 크기의 데이터셋에 적합

5.1 SVM(서포트 벡터 머신)

결정 경계(Decision Boundary): 분류를 위한 기준 선



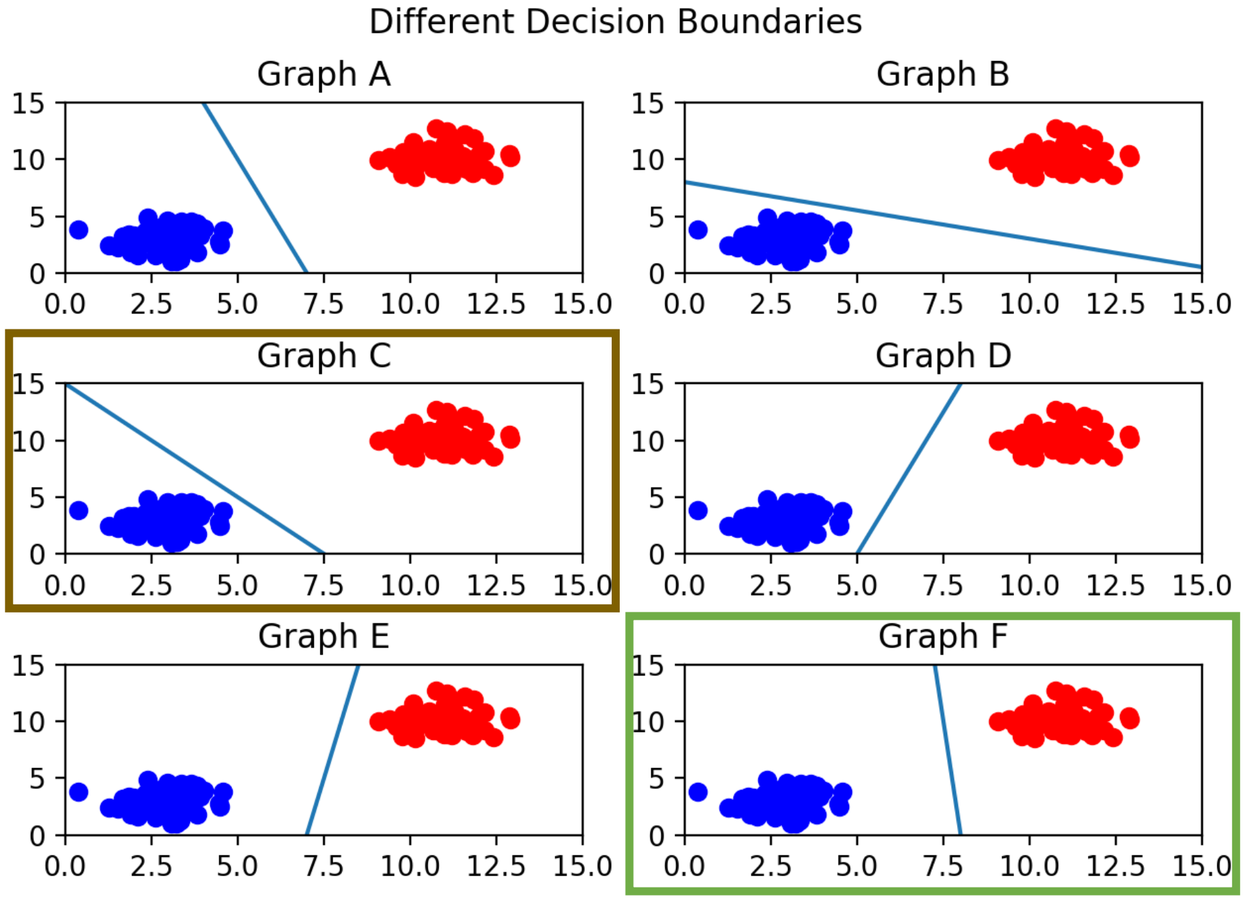
데이터 속성이 두개인 경우 → 결정경계: 선



데이터 속성이 세개인 경우 → 결정경계: 평면

5.1 SVM(서포트 벡터 머신)

최적의 결정 경계?



→ Graph F가 결정 경계로 가장 적절해 보임,
두 클래스 사이의 거리가 가장 멀기 때문!

➔ 결정경계가 데이터 군으로부터
최대한 멀리 떨어질 수록 좋다!

5.1 SVM(서포트 벡터 머신)

특성 스케일 *SVM은 특성의 스케일에 민감함

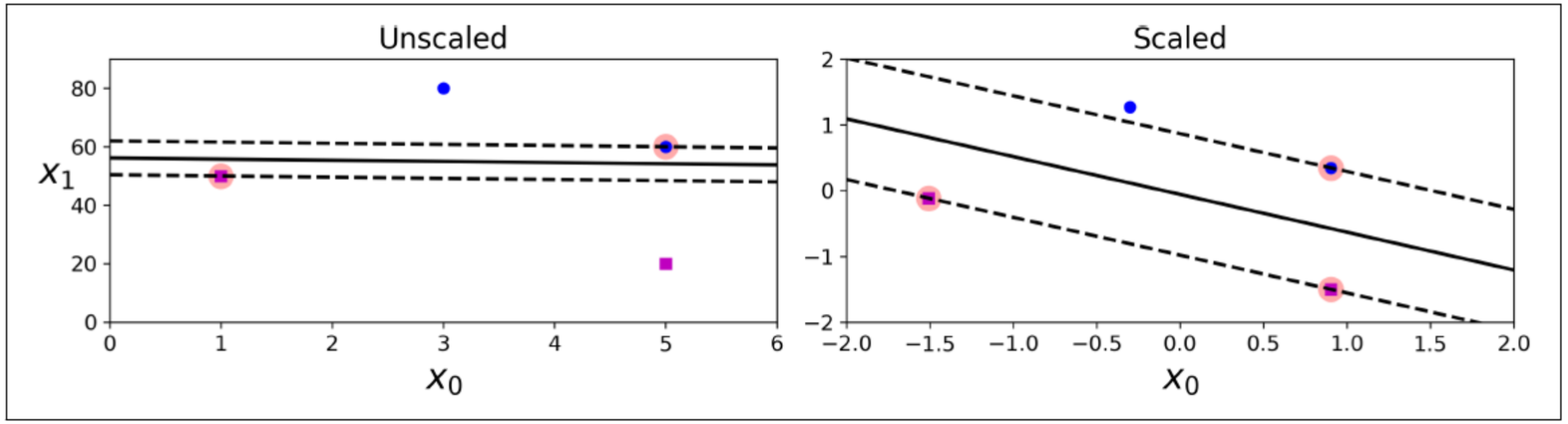


Figure 5-2. Sensitivity to feature scales

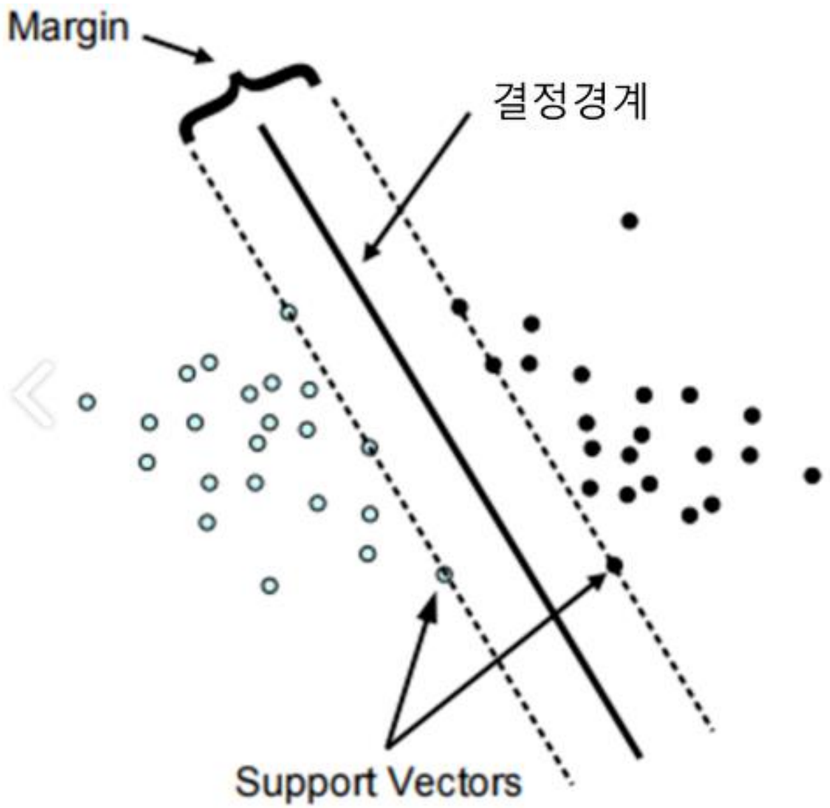
수직축의 스케일이
수평축의 스케일보다 훨씬 커서 가장
넓은 도로가 거의 수평에 가깝게 됨

특성의 스케일 조정(e.g. StandardScaler)
➔ 결정 경계 훨씬 좋아짐

5.1 SVM(서포트 벡터 머신)

Margin(마진): 결정 경계와 서포트 벡터 사이의 거리

(support vectors: 결정 경계와 가까이 있는 데이터 포인트
→ 이 데이터들이 경계를 정의하는 결정적인 역할을 함)

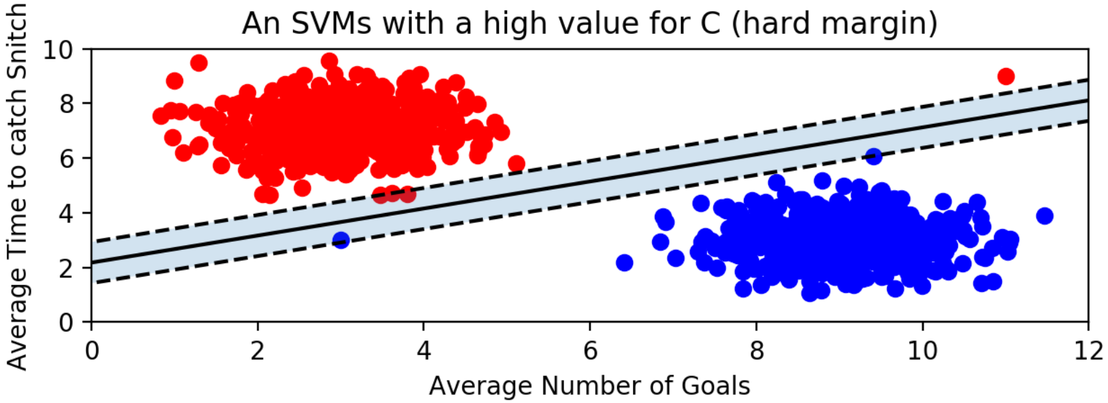


- 마진 = 점선으로부터 결정경계(실선)까지의 거리
- 최적의 결정 경계 = 마진 최대화
- SVM 분류기를 클래스 상에서 가장 폭이 넓은 도로를 찾는 것, 라지 마진 분류라고도 함

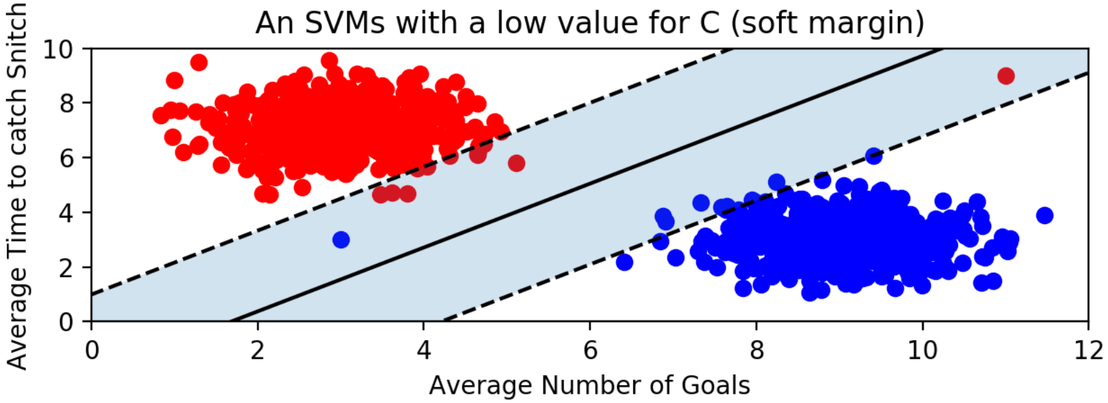
5.1 SVM(서포트 벡터 머신)

하드 마진 vs 소프트 마진(이상치 허용 정도) → 파라미터 'C'로 조절

데이터를 올바르게 분류하면서 마진의 크기를 최대화하려면 이상치 잘 다루는 것이 중요함



하드 마진
: 이상치에 민감하게 반응
→ 마진 ↓ ➔ train error ↓, 오버피팅 문제



소프트 마진
: 이상치에 대해 상대적으로 너그러운 기준
-> 마진 ↑ ➔ train error ↑, 언더피팅 문제

EWHA EURON

5.1 SVM(서포트 벡터 머신)

하드 마진 vs 소프트 마진(이상치 허용 정도) → **파라미터 'C'로 조절**

C: 마진의 넓이와 마진오류 사이의 trade off를 결정하는 파라미터 (즉, 모델의 오류 허용 정도)

➔ C ↓ → 소프트 마진(오류 허용) / C ↑ → 하드마진(오류 허용 안함)

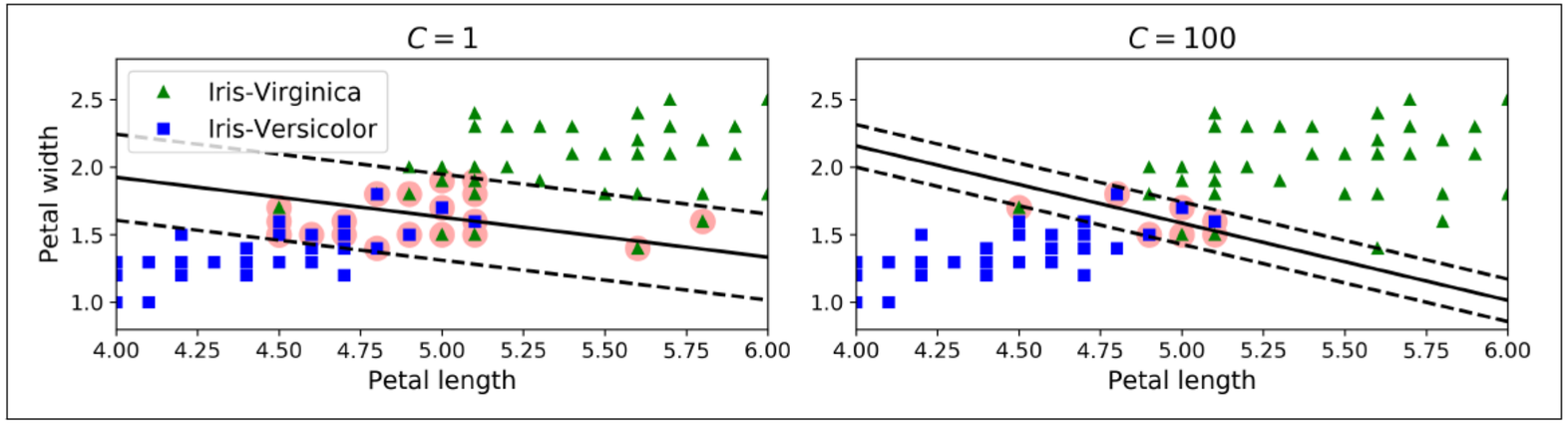
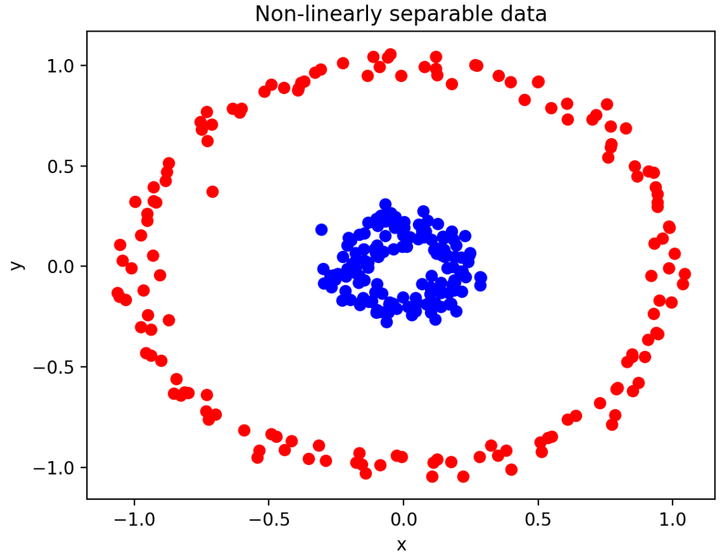


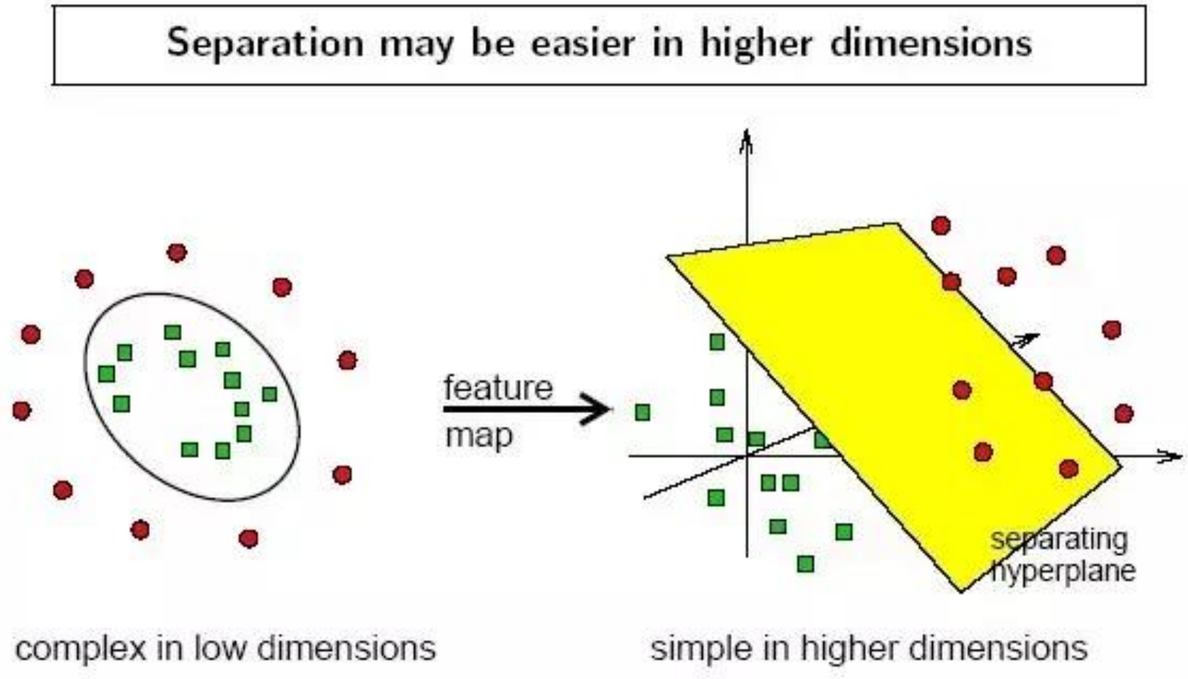
Figure 5-4. Large margin (left) versus fewer margin violations (right)

5.2 비선형 SVM

비선형 SVM: 선형적으로 분류할 수 없는 비선형 데이터셋을 다룰 때 활용



비선형 데이터



원래 데이터 차원을 고차원으로 변환시킨 후, 거기서 선형SVM적용 (즉, linear separated한 데이터로 변환하자!)

→ 일반적으로 엄청나게 높은 임의의 고차원으로 데이터를 변환하면 linearly separable하게 됨

→ 차원은 높이되, 계산량은 많이 높아지지 않도록 **커널 트릭** 사용

(커널 트릭: 실제로는 특성을 추가하지 않으면서 다항식 특성을 많이 추가한 것과 같은 결과를 얻을 수 있음)

5.2 비선형 SVM

커널(kernel) → 선형, 비선형 SVM 지정

선형 SVM

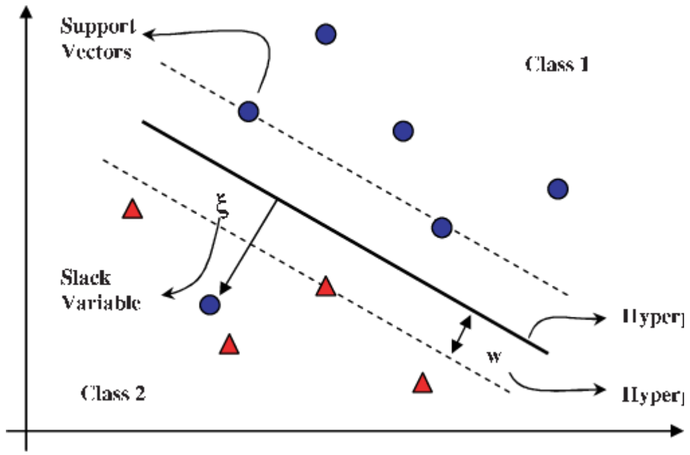
1) LinearSVC (→ 속도 더 빠름!) 2) linear 커널

```
from sklearn.svm import LinearSVC

svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("linear_svc", LinearSVC(C=1, loss="hinge", random_state=42))])

svm_clf.fit(X, y)
```

스케일에 민감

```
from sklearn.svm import SVC
classifier = SVC(kernel = 'linear')
```

* loss = "hinge"란?



모든 데이터를 항상 두 쪽으로 분류할 수 없음
→ 잘못 분류된 인스턴스 존재!
➔ Error case에 벌칙 주기
  1) error case에 해당하는 인스턴스 개수 기준으로 벌칙
  2) 멀리 떨어진 error case 인스턴스에 더 강한 벌칙
  → Hinge Loss (더 많이 사용하는 방식)

https://yngie-c.github.io/machine%20learning/2020/04/26/SVM/

5.2 비선형 SVM

비선형SVM

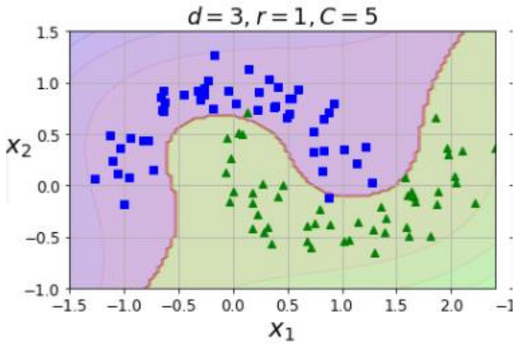
1) 다항식 커널(Polynomial Kernel) 2) 가우시안 RBF 커널 3) 시그모이드 커널

다항식 커널(Polynomial Kernel)

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))])
poly_kernel_svm_clf.fit(X,y)
```

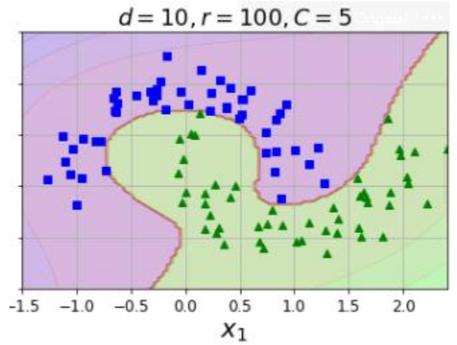
매개변수 coef0는 모델이 높은 차수와 낮은 차수에 얼마나 영향 받을지 조절 (상수항r) : 고차항의 영향 조절 가능

3차 다항식 커널




```
poly100_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=10, coef0=100, C=5))])
poly100_kernel_svm_clf.fit(X, y)
```

10차 다항식 커널



5.2 비선형 SVM

가우시안 RBF 커널 → 무한한 차원으로 변환

```
rbf_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))])

rbf_kernel_svm_clf.fit(X,y)
```

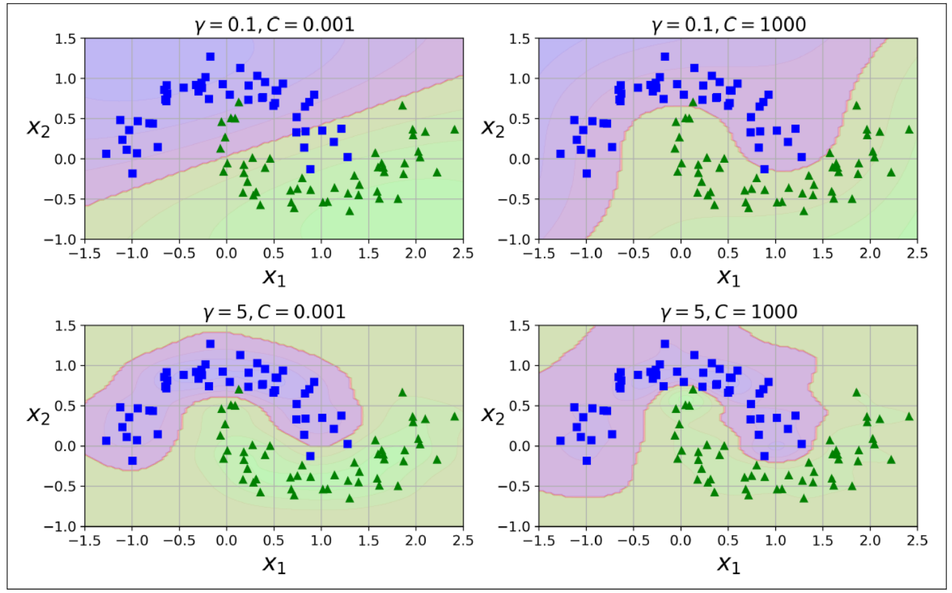


Figure 5-9. SVM classifiers using an RBF kernel

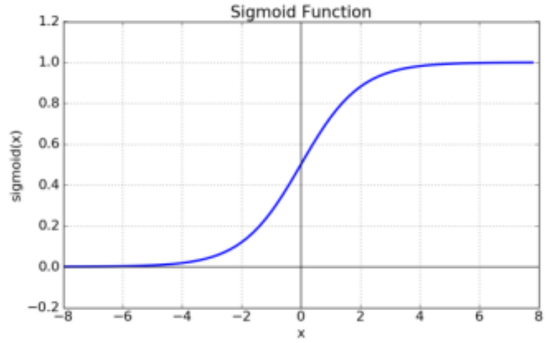
파라미터: gamma & C

- gamma: 하나의 데이터 샘플이 영향력을 행사하는 거리 결정(결정 경계의 유연한 정도 결정)
  - Gamma ↑ → 각 샘플의 영향 범위 작아짐 → 결정 경계 구불구불 → 오버피팅 위험
  - Gamma ↓ → 각 샘플의 영향 범위 넓어짐 → 결정 경계 부드러워짐 → 언더피팅 위험

➔ 언더피팅 → Gamma ↑ / 오버피팅 → Gamma ↓ ➔ 규제의 역할 함!

5.2 비선형 SVM

시그모이드 커널




```
svclassifier = SVC(kernel='sigmoid', degree=8)
svclassifier.fit(X_train, y_train)

y_pred = svclassifier.predict(X_test)
print('score : ',svclassifier.score(X_test,y_test))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

어떤 커널 사용?

1. 선형 커널 먼저 시도

   LinearSVC가 SVC(kernel = “linear”) 보다 훨씬 빠름

2. 훈련 세트가 너무 크지 않다면 가우시안 RBF 커널도 시도하면 좋고,

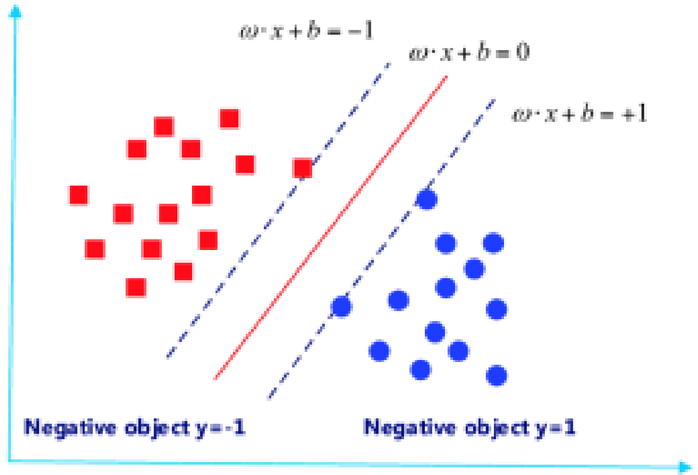
   대부분의 경우 이 커널이 잘 들어맞음. (주로 RBF 커널 많이 사용함)

5.3 SVM 회귀

SVM회귀와 SVM분류의 목표 비교

SVM 분류: 일정한 마진 오류 안에서

두 클래스 간의 도로 폭이 가능한 한 최대가 되도록 하는 것

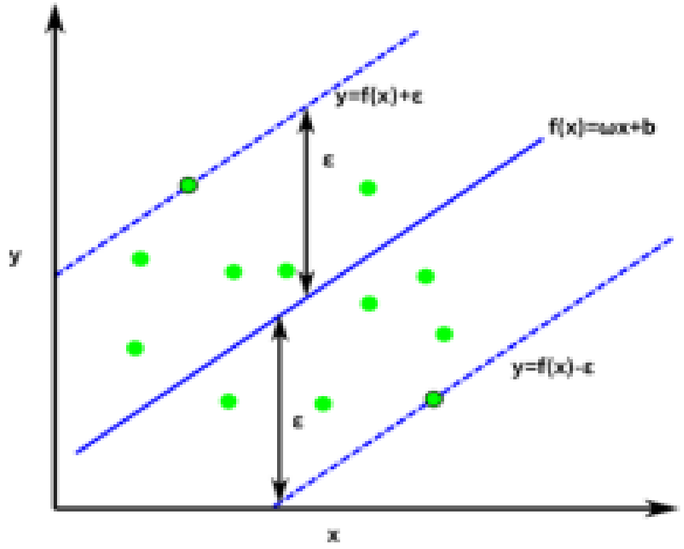


SVM 회귀: 제한된 마진 오류 안에서

도로 폭을 최대한 넓혀서

도로 위에 가능한 많은 샘플을 포함하도록 학습

**회귀 모델의 마진 위반 사례: 도로 밖에 위치한 샘플



5.3 SVM 회귀

SVM 선형 회귀

- 선형 회귀 모델을 SVM을 이용하여 구현
- 마진 안에서는 훈련 샘플이 추가되어도 모델의 예측에는 변함이 없음 → ε(epsilon)에 민감하지 않음
- ε(epsilon)으로 마진 조절

```
from sklearn.svm import LinearSVR

svm_reg = LinearSVR(epsilon=1.5, random_state=42)
svm_reg.fit(X, y)
```

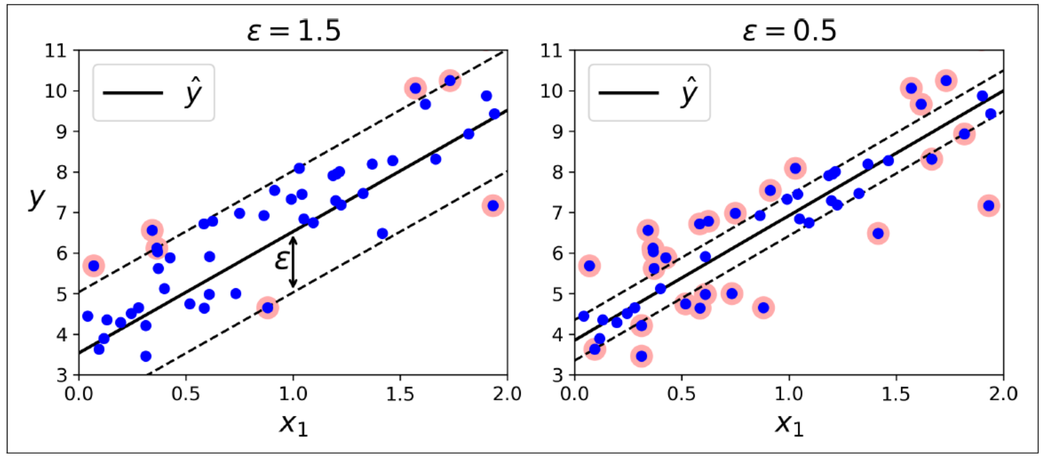


Figure 5-10. SVM Regression

왼: 마진 크게, 오: 마진 작게

5.3 SVM 회귀

SVM 비선형 회귀

- 커널 트릭을 활용해 비선형 회귀 모델 구현
- 마진 안에서는 훈련 샘플이 추가되어도 모델의 예측에는 변함이 없음 → ε(epsilon)에 민감하지 않음
- ε(epsilon)으로 마진 조절
- 하이퍼파라미터 C(정규화 매개 변수) 커질수록 마진 오차에 민감 → 오버피팅 위험

```
from sklearn.svm import SVR

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1, gamma="scale")
svm_poly_reg.fit(X, y)
```

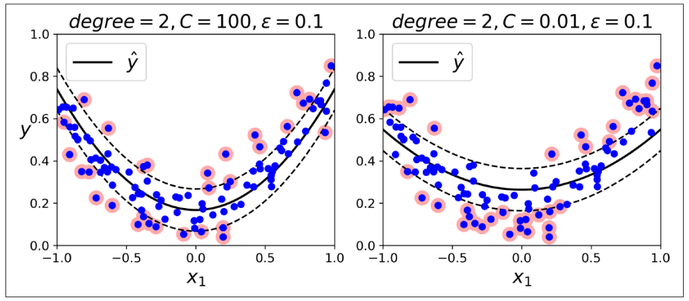


Figure 5-11. SVM regression using a 2nd-degree polynomial kernel

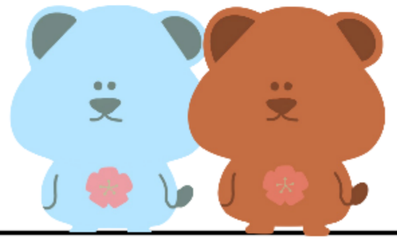
왼쪽: 규제 거의 x, 오른쪽: 규제 많음

5.4 SVM 장단점

SVM 장단점

장점	단점
회귀와 분류 문제 모두 사용 가능함	데이터가 너무 많으면 속도가 느리고 메모리적으로 힘듦
비선형 분리 데이터를 커널 트릭을 사용하여 분류 모델링 가능	확률 추정치를 제공하지 않음
고차원 공간에서 원활하게 작동	선형 커널은 선형의 분리 가능한 데이터인 경우 로지스틱 회귀 분석과 거의 유사함
텍스트 분류 및 이미지 분류에 효과적	

THANK YOU



EWHA EURON