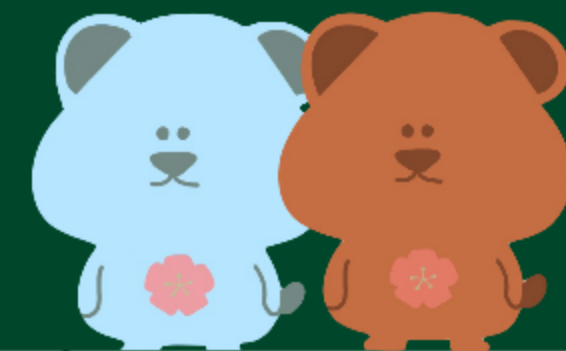




# 분류 Part 1.

3팀. 김소은 김수연 도하은 황사현



# 4.1 분류의 개요 및 4.2 결정 트리

황사현

## 4.1 분류의 개요



# 4.1 분류의 개요

## 지도학습

: 레이블(Label), 즉 명시적인 정답이 있는 데이터가 주어진 상태에서 학습하는 머신러닝 방식

## 분류(Classification)

: 학습 데이터로 주어진 데이터의 피처와 레이블 값(결정 값, 클래스 값)을 머신러닝 알고리즘으로 학습해 모델을 생성하고, 생성된 모델에 새로운 데이터 값이 주어졌을 때 미지의 레이블 값을 예측하는 것.

= 기존 데이터가 어떤 레이블에 속하는지 패턴을 알고리즘으로 인지한 뒤에 새롭게 관측된 데이터에 대한 레이블을 판별하는 것.

# 4.1 분류의 개요

## 분류를 구현할 수 있는 다양한 머신러닝 알고리즘

- 베이즈(Bayes) 통계와 생성 모델에 기반한 나이브 베이즈(Naïve Bayes)
- 독립변수와 종속변수의 선형 관계성에 기반한 로지스틱 회귀(Logistic Regression)
- 데이터 균일도에 따른 규칙 기반의 결정 트리(Decision Tree)
- 개별 클래스 간의 최대 분류 마진을 효과적으로 찾아주는 서포트 벡터 머신(Support Vector Machine)
- 근접 거리를 기준으로 하는 최소 근접(Nearest Neighbor) 알고리즘
- 심층 연결 기반의 신경망(Neural Network)
- 서로 다른(또는 같은) 머신러닝 알고리즘을 결합한 앙상블(Ensemble)

## 4.2 결정 트리



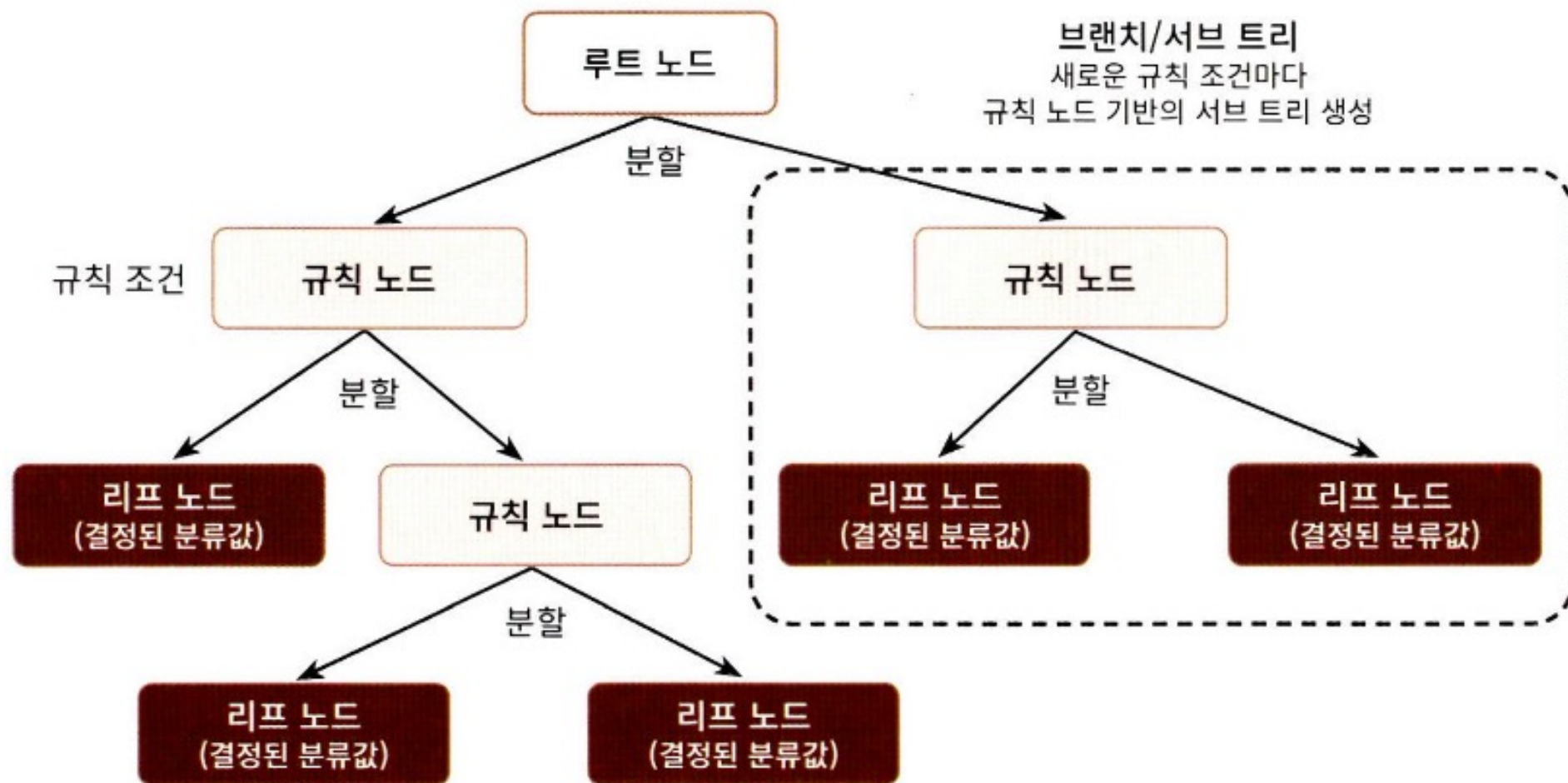
# 4.2 결정 트리

## 결정 트리(Decision Tree)

: 데이터에 있는 규칙을 학습을 통해 자동으로 찾아내  
트리(Tree) 기반의 분류 규칙을 만드는 것.

- 규칙을 가장 쉽게 표현하는 방법: if/else 기반으로 표현하기.
- 데이터의 어떤 기준을 바탕으로 규칙을 만들어야 가장 효율적인 “분류”가 될 것인가가 알고리즘의 성능을 크게 좌우함!

# 4.2 결정 트리의 구조



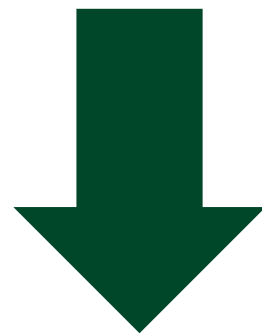
- 규칙 노드(Decision Node)
  - : 규칙 조건
  - 데이터 세트 내 피처가 결합해 규칙 조건을 만들 때마다 생성
- 리프 노드(Leaf Node)
  - : 결정된 클래스 값
- 서브 트리(Sub Tree)
  - 새로운 규칙 조건마다 생성



많은 규칙이 있다는 것

=

분류를 결정하는 방식이  
복잡해진다는 것



**과적합**으로 이어질 가능성!!

트리의 깊이(depth)가 깊어질수록  
결정 트리의 예측 성능이 저하될 가능성이 높아짐.



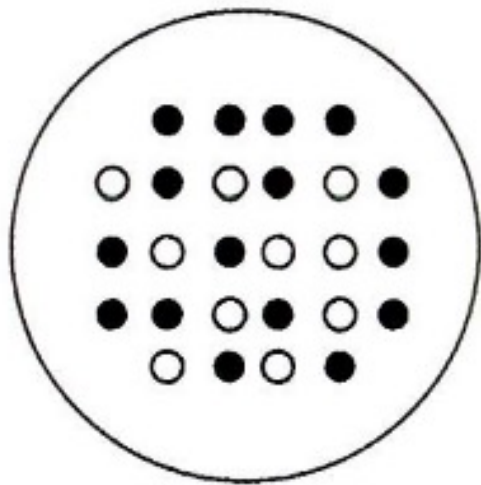
\* 최대한 많은 데이터 세트가 해당 분류에 속할 수 있도록 결정 노드의 규칙이 정해져야 함.

- 어떻게 트리를 분할(Split)할 것인가가 중요!

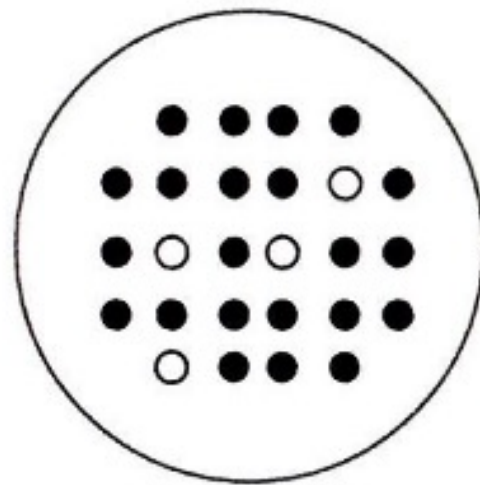
= 최대한 균일한 데이터 세트를 구성할 수 있도록 분할하는 것

Q. “균일한 데이터 세트”란?

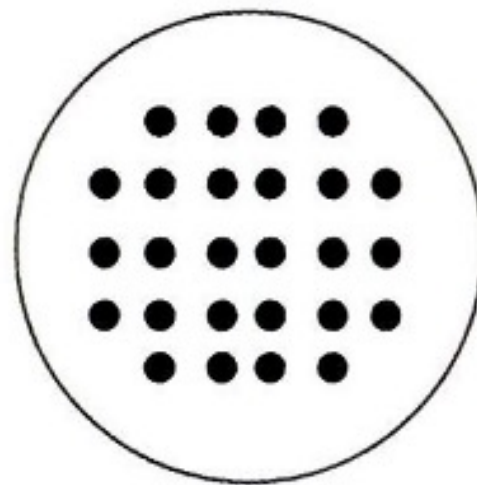
데이터 세트 A



데이터 세트 B



데이터 세트 C



답: C > B > A

\* **균일도** -> 데이터를 구분하는 데 필요한 정보의 양에 영향을 미침.



\* 결정 노드 -> 정보 균일도가 높은 데이터 세트를 먼저 선택할 수 있도록 규칙 조건을 만듦.

## [ 데이터 값 예측 과정 ]

1. 정보 균일도가 데이터 세트로 쪼개질 수 있도록 조건을 찾아 서브 데이터 세트를 만들고,
2. 이 서브 데이터 세트에서 균일도가 높은 자식 데이터 세트 쪼개는 방식을
3. 자식 트리로 내려가면서 반복.



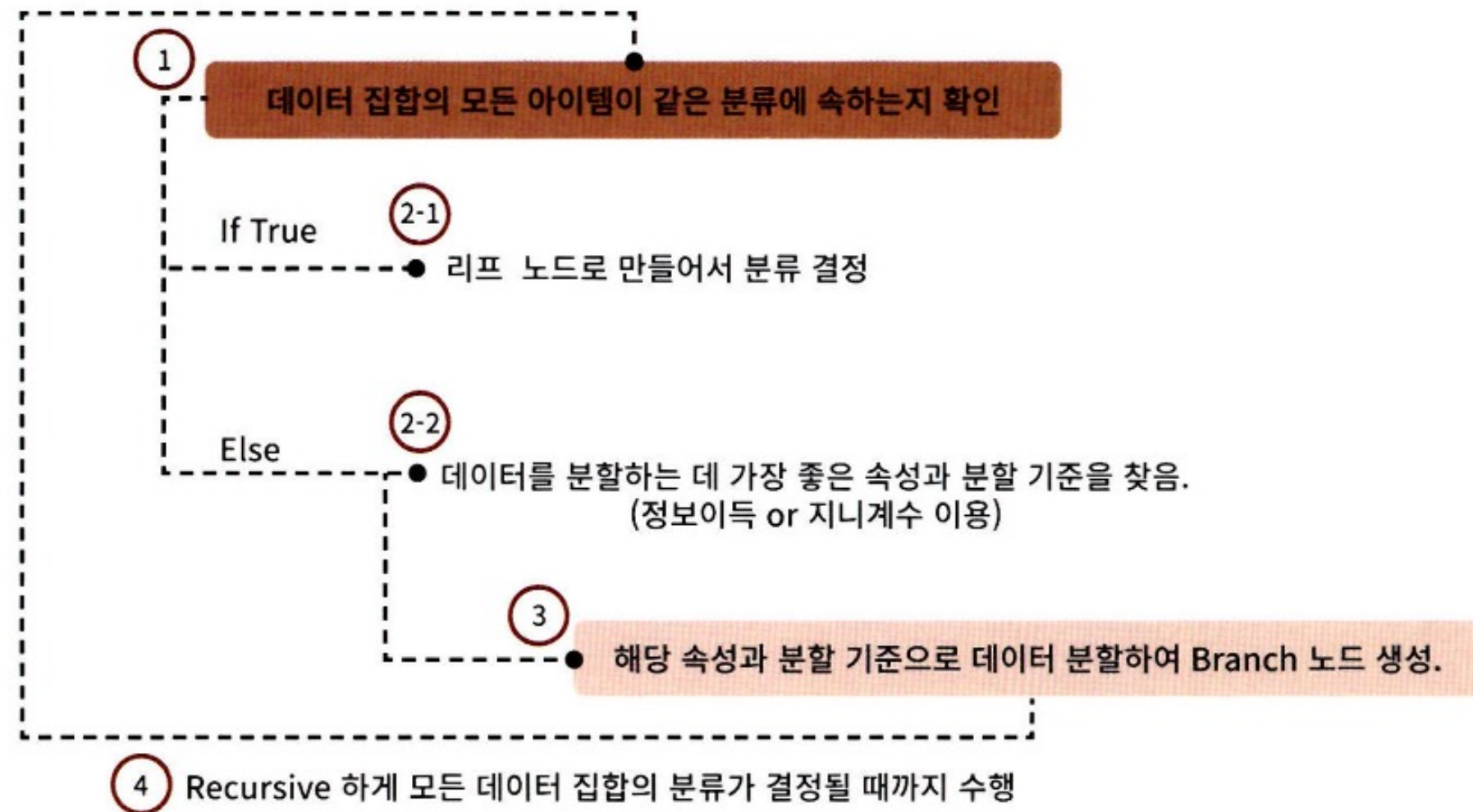
## \* 정보 균일도를 측정하는 대표적인 방법

= 엔트로피를 이용한 “정보 이득(Information Gain)지수”, “지니 계수”

- 정보 이득은 엔트로피라는 개념을 기반으로 합니다. 엔트로피는 주어진 데이터 집합의 혼잡도를 의미하는데, 서로 다른 값이 섞여 있으면 엔트로피가 높고, 같은 값이 섞여 있으면 엔트로피가 낮습니다. 정보 이득 지수는 1에서 엔트로피 지수를 뺀 값입니다. 즉,  $1 - \text{엔트로피 지수}$ 입니다. 결정 트리는 이 정보 이득 지수로 분할 기준을 정합니다. 즉, 정보 이득이 높은 속성을 기준으로 분할합니다.
- 지니 계수는 원래 경제학에서 불평등 지수를 나타낼 때 사용하는 계수입니다. 경제학자인 코라도 지니(Corrado Gini)의 이름에서 딴 계수로서 0이 가장 평등하고 1로 갈수록 불평등합니다. 머신러닝에 적용될 때는 지니 계수가 낮을수록 데이터 균일도가 높은 것으로 해석해 지니 계수가 낮은 속성을 기준으로 분할합니다.



- \* **DecisionTreeClassifier** : 결정 트리 알고리즘을 사이킷런에서 구현한 것!  
기본적으로 지니 계수를 이용해 데이터 세트를 분할함





# 4.2 결정 트리 모델의 특징

결정 트리 장점	결정 트리 단점
<ul style="list-style-type: none"><li>• 쉽다. 직관적이다</li><li>• 피처의 스케일링이나 정규화 등의 사전 가공 영향도가 크지 않음.</li></ul>	<ul style="list-style-type: none"><li>• 과적합으로 알고리즘 성능이 떨어진다. 이를 극복하기 위해 트리의 크기를 사전에 제한하는 튜닝 필요.</li></ul>

- 결정 트리는 학습 데이터 기반 모델의 정확도를 높이기 위해 조건을 계속 추가함.
- ➔ 트리 깊이가 계속 커짐.
  - ➔ 결국 복잡한 학습 모델이 만들어짐.
  - ➔ 실제 상황에 유연하게 대처하기 힘들.
  - ➔ 예측 성능이 떨어짐.
- ➔ 트리의 크기를 사전에 제한하는 것이 오히려 성능 튜닝에 더 도움이 됨.

# 4.2 결정 트리 파라미터

파라미터 명	설명
min_samples_split	<ul style="list-style-type: none"><li>• 노드를 분할하기 위한 최소한의 샘플 데이터 수로 과적합을 제어하는 데 사용됨.</li><li>• 디폴트는 2이고 작게 설정할수록 분할되는 노드가 많아져서 과적합 가능성 증가</li><li>• 과적합을 제어. 1로 설정할 경우 분할되는 노드가 많아져서 과적합 가능성 증가</li></ul>
min_samples_leaf	<ul style="list-style-type: none"><li>• 말단 노드(Leaf)가 되기 위한 최소한의 샘플 데이터 수</li><li>• Min_samples_split와 유사하게 과적합 제어 용도. 그러나 비대칭적(imbalanced) 데이터의 경우 특정 클래스의 데이터가 극도로 작을 수 있으므로 이 경우는 작게 설정 필요.</li></ul>
파라미터 명	설명
max_features	<ul style="list-style-type: none"><li>• 최적의 분할을 위해 고려할 최대 피처 개수. 디폴트는 None으로 데이터 세트의 모든 피처를 사용해 분할 수행.</li><li>• int 형으로 지정하면 대상 피처의 개수, float 형으로 지정하면 전체 피처 중 대상 피처의 퍼센트임</li><li>• 'sqrt'는 전체 피처 중 <math>\sqrt{\text{전체 피처 개수}}</math> 만큼 선정</li><li>• 'auto'로 지정하면 sqrt와 동일</li><li>• 'log'는 전체 피처 중 <math>\log_2(\text{전체 피처 개수})</math> 선정</li><li>• 'None'은 전체 피처 선정</li></ul>
max_depth	<ul style="list-style-type: none"><li>• 트리의 최대 깊이를 규정.</li><li>• 디폴트는 None. None으로 설정하면 완벽하게 클래스 결정 값이 될 때까지 깊이를 계속 키우며 분할하거나 노드가 가지는 데이터 개수가 min_samples_split보다 작아질 때까지 계속 깊이를 증가시킴.</li><li>• 깊이가 깊어지면 min_samples_split 설정대로 최대 분할하여 과적합할 수 있으므로 적절한 값으로 제어 필요.</li></ul>
max_leaf_nodes	<ul style="list-style-type: none"><li>• 말단 노드(Leaf)의 최대 개수</li></ul>

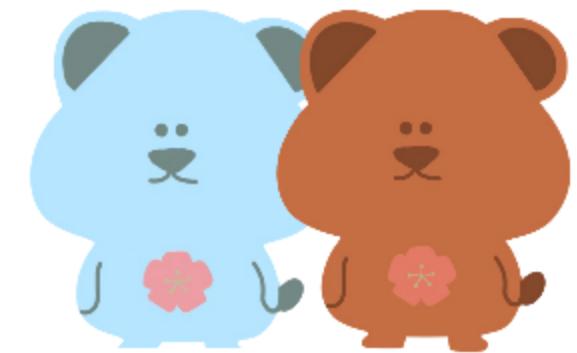


# 결정 트리 실습 및 4.3 앙상블 학습

김수연



# 결정 트리 실습



## 4.2 결정 트리 실습

```
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
```

중복된 피처명에 대해서 원본 피처명에 \_1 또는 \_2를 추가로 부여해 새로운 피처명을 가지는  
DataFrame을 반환하는 함수 생성

## 4.2 결정 트리 실습

```
def get_human_dataset():

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

    # 중복된 피처명을 수정한느 get_new_feature_name_df()를 이용, 신규 피처명 DataFrame 생성
    new_feature_name_df = get_new_feature_name_df(feature_name_df)

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

    # 학습 피처 데이터셋과 테스트 피처 데이터를 DataFrame으로 로딩. 칼럼명은 feature_name 적용
    X_train = pd.read_csv('./human_activity/train/X_train.txt', sep='\s+', names=feature_name)
    X_test = pd.read_csv('./human_activity/test/X_test.txt', sep='\s+', names=feature_name)

    # 학습 레이블과 테스트 레이블 데이터를 DataFrame으로 로딩. 칼럼명은 action으로 부여
    y_train = pd.read_csv('./human_activity/train/y_train.txt', sep='\s+', header=None, names=['action'])
    y_test = pd.read_csv('./human_activity/test/y_test.txt', sep='\s+', header=None, names=['action'])

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

X_train, X_test, y_train, y_test = get_human_dataset()
```

train 디렉터리에 있는 학습용 피처 데이터 세트와 레이블 데이터 세트, test 디렉터리에 있는 테스트용 피처 데이터 파일과 레이블 데이터 파일을 각각 학습/테스트용 DataFrame에 로드

## 4.2 결정 트리 실습

사이킷런의 `DecisionTreeClassifier`를 이용해 동작 예측 분류 수행  
먼저 하이퍼 파라미터는 모두 디폴트 값으로 설정하여 수행

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# 예제 반복 시마다 동일한 예측 결과 도출을 위해 random_state 설정
dt_clf = DecisionTreeClassifier(random_state=156)
dt_clf.fit(X_train, y_train)
pred = dt_clf.predict(X_test)
accuracy = accuracy_score(y_test, pred)
print('결정 트리 예측 정확도: {0:.4f}'.format(accuracy))

# DecisionTreeClassifier의 하이퍼 파라미터 추출
print('DecisionTreeClassifier 기본 하이퍼 파라미터:\n', dt_clf.get_params())
```

결정 트리 예측 정확도: 0.5848

# 4.2 결정 트리 실습

결정 트리의 깊이가 예측 정확도에 주는 영향을 살펴보자.

다음은 `GridSearchCV`를 이용해 사이킷런의 결정 트리의 깊이를 조절할 수 있는 하이퍼 파라미터인 `max_depth`를 6, 8, 10, 12, 16, 20, 24로 늘리며 예측 성능을 측정한 것이다.

```
from sklearn.model_selection import GridSearchCV

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

grid_cv = GridSearchCV(dt_clf, param_grid=params, scoring='accuracy', cv=5, verbose=1)
grid_cv.fit(X_train, y_train)
print('GridSearchCV 최고 평균 정확도 수치: {0:.4f}'.format(grid_cv.best_score_))
print('GridSearchCV 최적 하이퍼 파라미터: ', grid_cv.best_params_)
```

GridSearchCV 최고 평균 정확도 수치: 0.8526  
GridSearchCV 최적 하이퍼 파라미터: {'max\_depth': 8}

	param_max_depth	mean_test_score
0	6	0.850925
1	8	0.852557
2	10	0.850925
3	12	0.844124
4	16	0.852149
5	20	0.851605
6	24	0.850245

# 4.2 결정 트리 실습

결정 트리는 더 완벽한 규칙을 학습 데이터 세트에 적용하기 위해  
노드를 지속적으로 분할하며 깊이가 깊어지고 더욱 더 복잡한 모델이 된다..

	param_max_depth	mean_test_score
0	6	0.850925
1	8	0.852557
2	10	0.850925
3	12	0.844124
4	16	0.852149
5	20	0.851605
6	24	0.850245

깊어진 트리는 학습 데이터 세트에는  
올바른 예측 결과를 가져올지 모르지만,  
검증 데이터 세트에서는 오히려 과적합으로 인한  
성능 저하를 유발한다.

## 4.2 결정 트리 실습

별도의 테스트 데이터 세트에서 결정 트리의 정확도를 측정해보자.

```
max_depths = [6, 8, 10, 12, 16, 20, 24]

# max_depth 값을 변화시키면서 그때마다 학습과 테스트 세트에서의 예측 성능 측정
for depth in max_depths:
    dt_clf = DecisionTreeClassifier(max_depth=depth, random_state=156)
    dt_clf.fit(X_train, y_train)
    pred = dt_clf.predict(X_test)
    accuracy = accuracy_score(y_test, pred)
    print('max_depth = {0} 정확도: {1:.4f}'.format(depth, accuracy))
```

max\_depth가 8일 경우 약 87.07%로 가장 높은 정확도를 나타낸다.

앞의 GridSearchCV 예제와 마찬가지로 깊이가 깊어질수록 데이터 세트의 정확도는 더 떨어진다.

이처럼 하이퍼 파라미터를 이용해 깊이를 제어할 수 있어야 한다.

## 4.2 결정 트리 실습

`max_depth`와 `min_samples_split`을 같이 변경하며 정확도 성능을 튜닝해보자.

```
params = {  
    'max_depth' : [8, 12, 16, 20],  
    'min_samples_split' : [16, 24],  
}  
  
grid_cv = GridSearchCV(dt_clf, param_grid=params, scoring='accuracy', cv=5, verbose=1)  
grid_cv.fit(X_train, y_train)  
print('GridSearchCV 최고 평균 정확도 수치: {0:.4f}'.format(grid_cv.best_score_))  
print('GridSearchCV 최적 하이퍼 파라미터: ', grid_cv.best_params_)
```

GridSearchCV 최고 평균 정확도 수치: 0.8550

GridSearchCV 최적 하이퍼 파라미터: {'max\_depth': 8, 'min\_samples\_split': 16}



## 4.2 결정 트리 실습

별도 분리된 테스트 데이터 세트에 해당 하이퍼 파라미터를 적용해보자.

앞 예제의 GridSearchCV 객체인 `grid_cv`의 속성인 `best_estimator_`는 최적 하이퍼 파라미터인 `max_depth 8`, `min_samples_split 16`으로 학습이 완료된 Estimator 객체이다. 다음은 이를 이용해 테스트 데이터 세트에 예측을 수행한 것이다.

```
best_df_clf = grid_cv.best_estimator_  
pred1 = best_df_clf.predict(X_test)  
accuracy = accuracy_score(y_test, pred1)  
print('결정 트리 예측 정확도: {0:.4f}'.format(accuracy))
```

결정 트리 예측 정확도: 0.8718

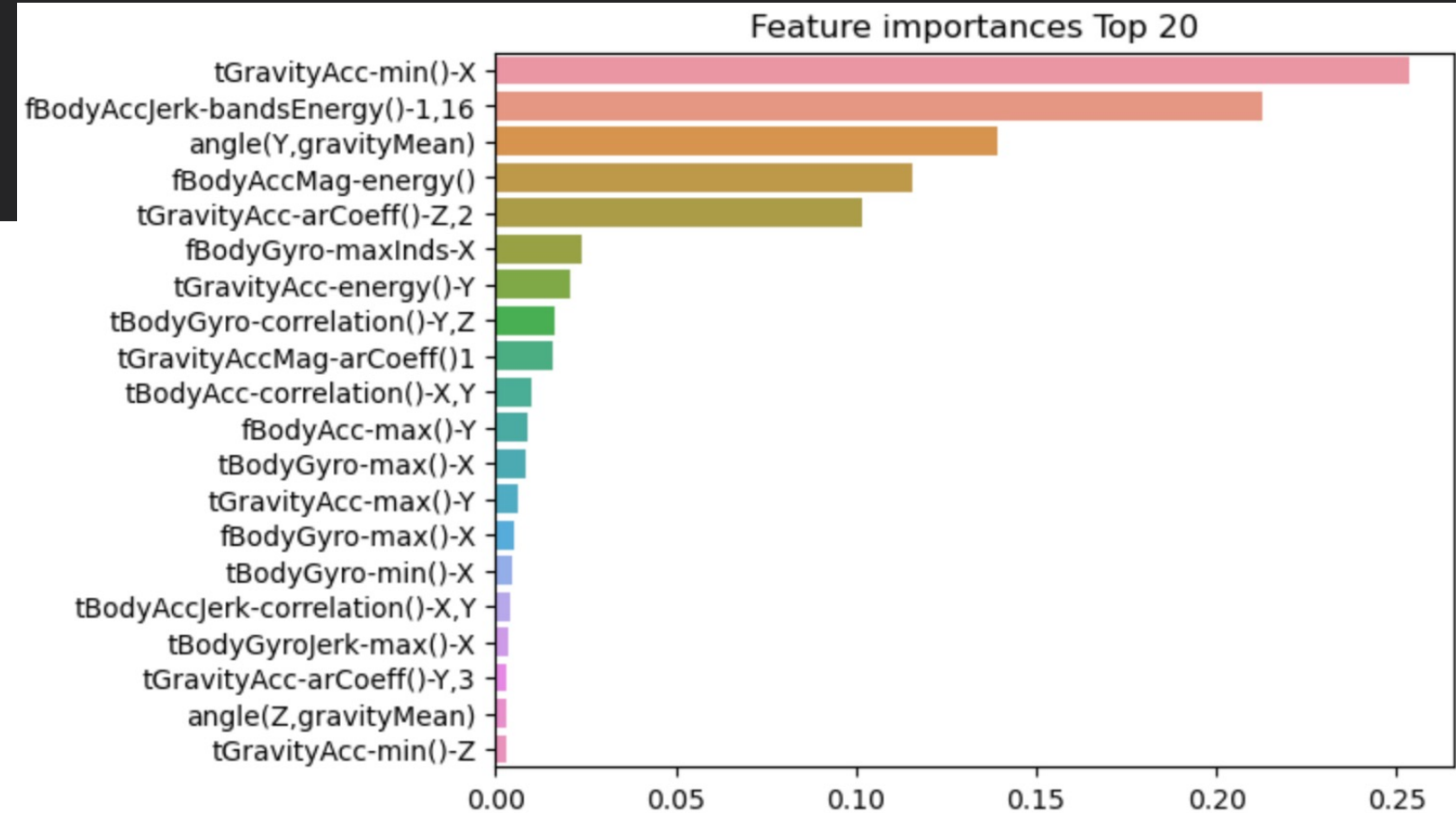
## 4.2 결정 트리 실습

마지막으로 결정 트리에서 각 피처의 중요도를 `feature_importances_` 속성을 이용해 알아보자.

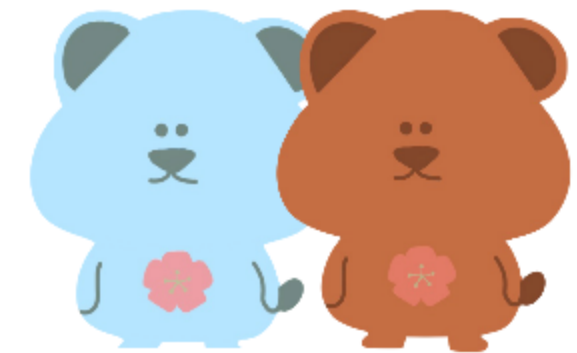
```
import seaborn as sns

ftr_importances_values = best_df_clf.feature_importances_
# Top 중요도로 정렬을 쉽게 하고, 시본(Seaborn)의 막대그래프로 쉽게 표현하기 위해 Series 변환
ftr_importances = pd.Series(ftr_importances_values, index=X_train.columns)
# 중요도값 순으로 Series를 정렬
ftr_top20 = ftr_importances.sort_values(ascending=False)[:20]
plt.title('Feature importances Top 20')
sns.barplot(x=ftr_top20, y=ftr_top20.index)
plt.show()
```

가장 높은 중요도를 가진 top5의 피처들이  
매우 중요하게 규칙생성에 영향을  
미치고 있다.



## 4.3 앙상블 학습



## 4.3 앙상블 학습

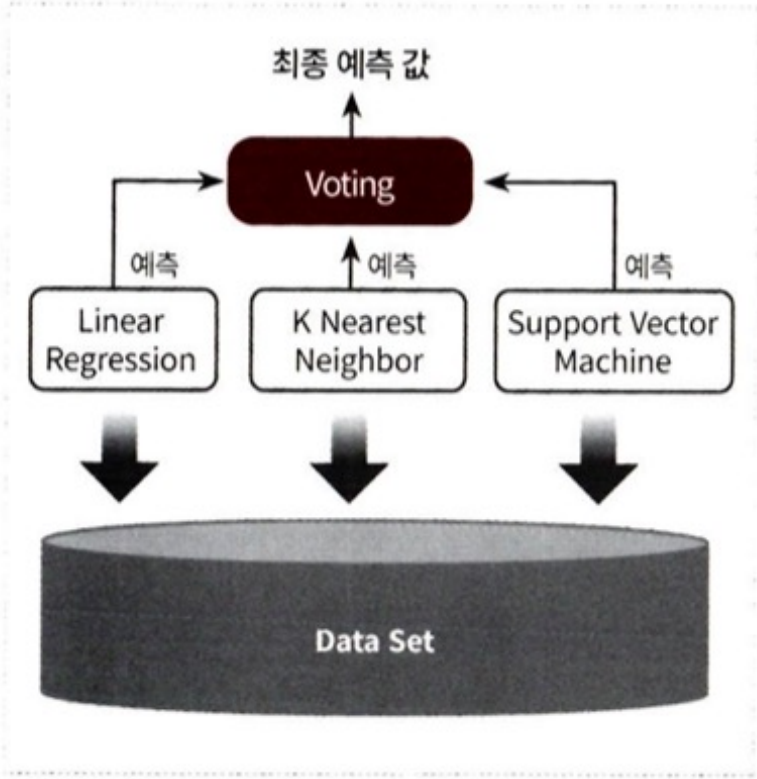
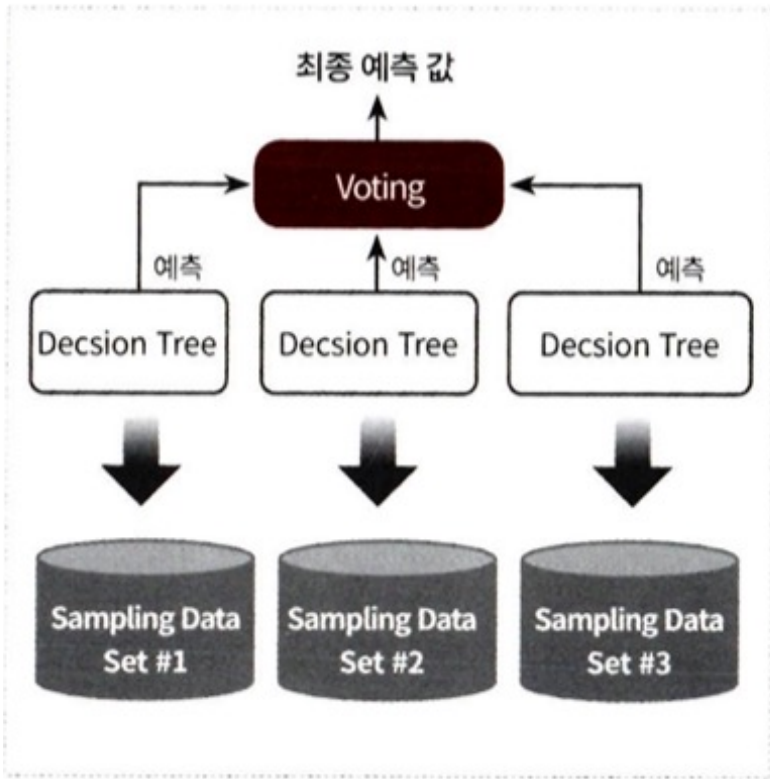
### 앙상블 학습(Ensemble Learning)

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

앙상블 학습의 목표는 단일 분류기보다 신뢰성이 높은 예측 값을 얻는 것이다.  
쉽고 편하면서도 강력한 성능을 보유하고 있는 것이 앙상블 학습의 특징!

앙상블의 학습 유형은 전통적으로 보팅(Voting), 배깅(Bagging), 부스팅(Boosting)의 세 가지로  
나눌 수 있으며, 이외에도 스태킹을 포함한 다양한 앙상블 방법이 있다.

# 4.3 앙상블 학습

보팅 (Voting)	배깅 (Bagging)
여러 개의 분류기가 투표를 통해 최종 예측 결과를 결정	
서로 다른 알고리즘을 가진 분류기를 결합	모두 같은 유형의 알고리즘을 기반으로 한 각각의 분류기가 데이터 샘플링을 서로 다르게 가져가면서 학습을 수행
 <p>Voting 방식</p> <p>The diagram illustrates the Voting ensemble method. At the bottom is a cylinder labeled 'Data Set'. Three arrows point upwards from this data set to three separate model boxes: 'Linear Regression', 'K Nearest Neighbor', and 'Support Vector Machine'. Each model box has an arrow labeled '예측' (Prediction) pointing to a central 'Voting' box. The 'Voting' box then has an arrow labeled '최종 예측 값' (Final Prediction Value) pointing to the top.</p>	 <p>Bagging 방식</p> <p>The diagram illustrates the Bagging ensemble method. At the bottom are three separate cylinders labeled 'Sampling Data Set #1', 'Sampling Data Set #2', and 'Sampling Data Set #3'. Each sampling set has an arrow pointing upwards to a corresponding 'Decsion Tree' box (note the typo in the image). Each 'Decsion Tree' box has an arrow labeled '예측' (Prediction) pointing to a central 'Voting' box. The 'Voting' box then has an arrow labeled '최종 예측 값' (Final Prediction Value) pointing to the top.</p>

## 4.3 앙상블 학습

**부스팅**은 여러 개의 분류기가 순차적으로 학습을 수행하되,  
앞에서 학습한 분류기가 예측이 틀린 데이터에 대해서는 올바르게 예측할 수 있도록  
다음 분류기에게는 가중치(weight)를 부여하면서 학습과 예측을 진행하는 것이다.  
예측 성능이 뛰어나 앙상블 학습을 주도하고 있다.  
ex. 그래디언트 부스트, XGBoost, LightGBM

**스태킹**은 여러 가지 다른 모델의 예측 결과값을 다시 학습 데이터로 만들어  
다른 모델로 재학습시켜 결과를 예측하는 방법이다.



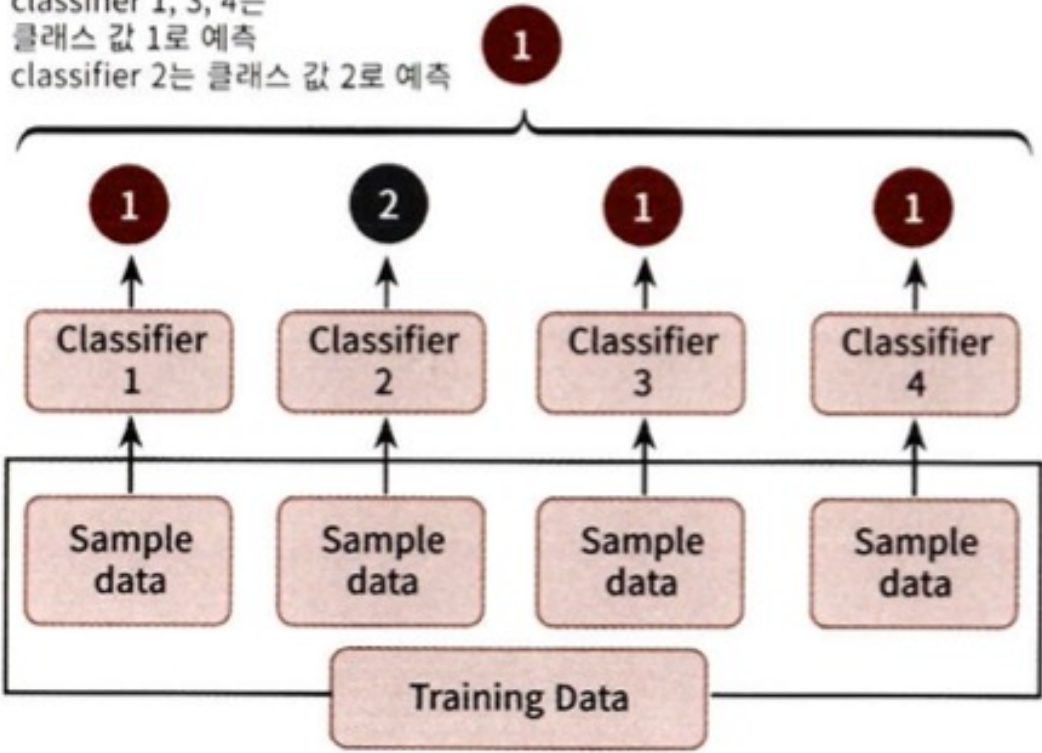
# 4.3 하드 보팅과 소프트 보팅

## 하드 보팅 (Hard Voting)

측한 결과값들 중 다수의 분류기가 결정한 예측값을 최종 보팅 결과값으로 선정하는 것

Hard Voting은 다수의 classifier 간 다수결로 최종 class 결정

클래스 값 1로 예측  
classifier 1, 3, 4는  
클래스 값 1로 예측  
classifier 2는 클래스 값 2로 예측

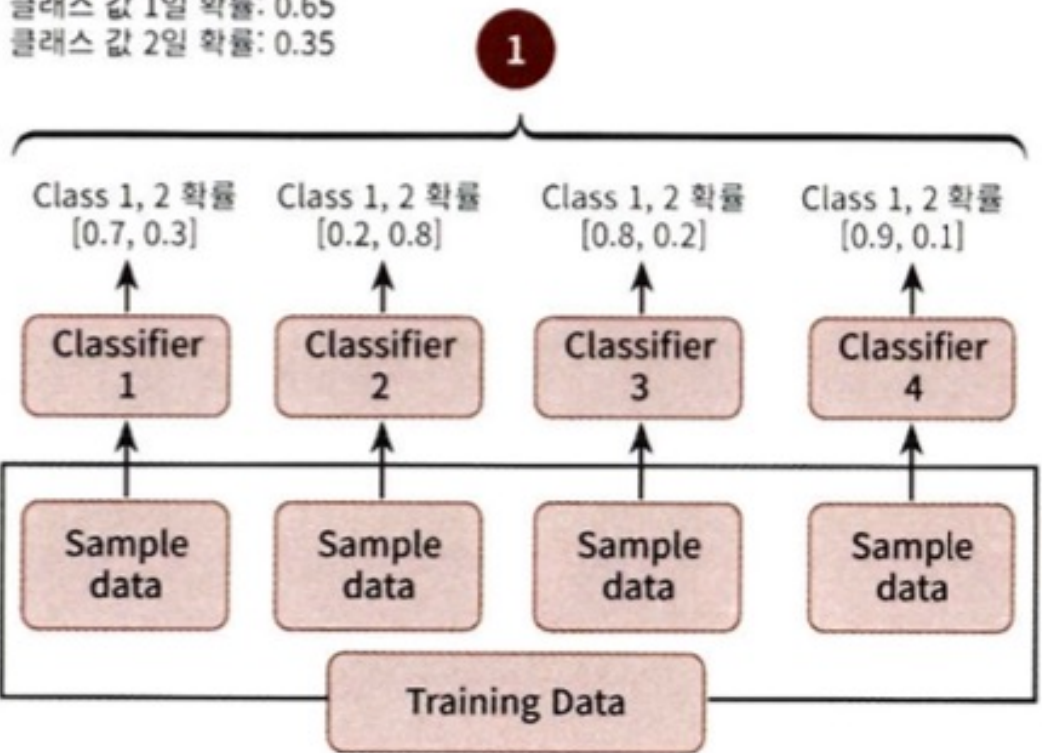


## 소프트 보팅 (Soft Voting)

분류기들의 레이블 값 결정 확률을 모두 더하고 이를 평균해서 이들 중 가장 높은 레이블 값을 최종 보팅 결과값으로 선정하는 것

Soft Voting은 다수의 classifier 들의 class 확률을 평균하여 결정

클래스 값 1로 예측  
클래스 값 1일 확률: 0.65  
클래스 값 2일 확률: 0.35



## 4.3 보팅 분류기

사이킷런은 `VotingClassifier` 클래스를 통해  
보팅 방식의 앙상블을 구현

`estimators`와 `voting` 값을 주요 인자로 입력 받는다.

`estimators`는 리스트 값으로 보팅에 사용될 여러 개의 `Classifier` 객체들을 튜플 형식으로 입력  
받으며 `voting`은 'hard' 시 하드 보팅, 'soft' 시 소프트 보팅 방식을 적용한다 (기본은 'hard'이다).



## 4.3 보팅 분류기

```
# 개별 모델은 로지스틱 회귀와 KNN
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)

# VotingClassifier 학습/예측/평가
vo_clf.fit(X_train, y_train)
pred = vo_clf.predict(X_test)
print('Voting 분류기 정확도: {0:.4f}'.format(accuracy_score(y_test, pred)))

# 개별 모델의 학습/예측/평가
classifiers = [lr_clf, knn_clf]
for classifier in classifiers:
    classifier.fit(X_train, y_train)
    pred = classifier.predict(X_test)
    class_name = classifier.__class__.__name__
    print('{0} 정확도: {1:.4f}'.format(class_name, accuracy_score(y_test, pred)))
```

Voting 분류기 정확도: 0.9474

LogisticRegression 정확도: 0.9386

KNeighborsClassifier 정확도: 0.9386

로지스틱 회귀와 KNN을 기반으로 하여 소프트 보팅 방식으로 새롭게 보팅 분류기를 만들었다.

보팅 분류기의 정확도가 더 높게 나타났다.

보팅으로 여러 개의 기반 분류기를 결합한다고 해서 무조건 기반 분류기보다 예측 성능이 향상되는 것은 아니지만 단일 ML 알고리즘보다 뛰어난 예측 성능을 가지는 경우가 많다.

## 4.3 보팅 분류기

결정 트리 알고리즘은 쉽고 직관적인 분류 기준을 가지고 있지만 정확한 학습을 위해 학습 데이터의 예외 상황에 집착한 나머지 오히려 과적합이 발생해 실제 테스트 데이터에서 예측 성능이 떨어지는 현상이 발생하기 쉽다.

그러나 앙상블 학습에서는 이 같은 결정 트리 알고리즘의 단점을 수십~수천 개의 매우 많은 분류기를 결합해 다양한 상황을 학습하게 함으로써 극복하고 있다.

이처럼 앙상블 학습을 통해 결정 트리 알고리즘의 장점은 그대로 취하고 단점을 보완하면서 편향-분산 트레이드오프의 효과를 극대화할 수 있다.



# 배깅 및 4.4 랜덤 포레스트

김소은

배경



## 배깅(Bagging)

각각의 분류기가 모두 같은 유형의 알고리즘 기반이지만, 데이터 샘플링을 서로 다르게 가져가면서 학습을 수행해 보팅을 수행

대표적인 배깅 방식:  
랜덤 포레스트 알고리즘

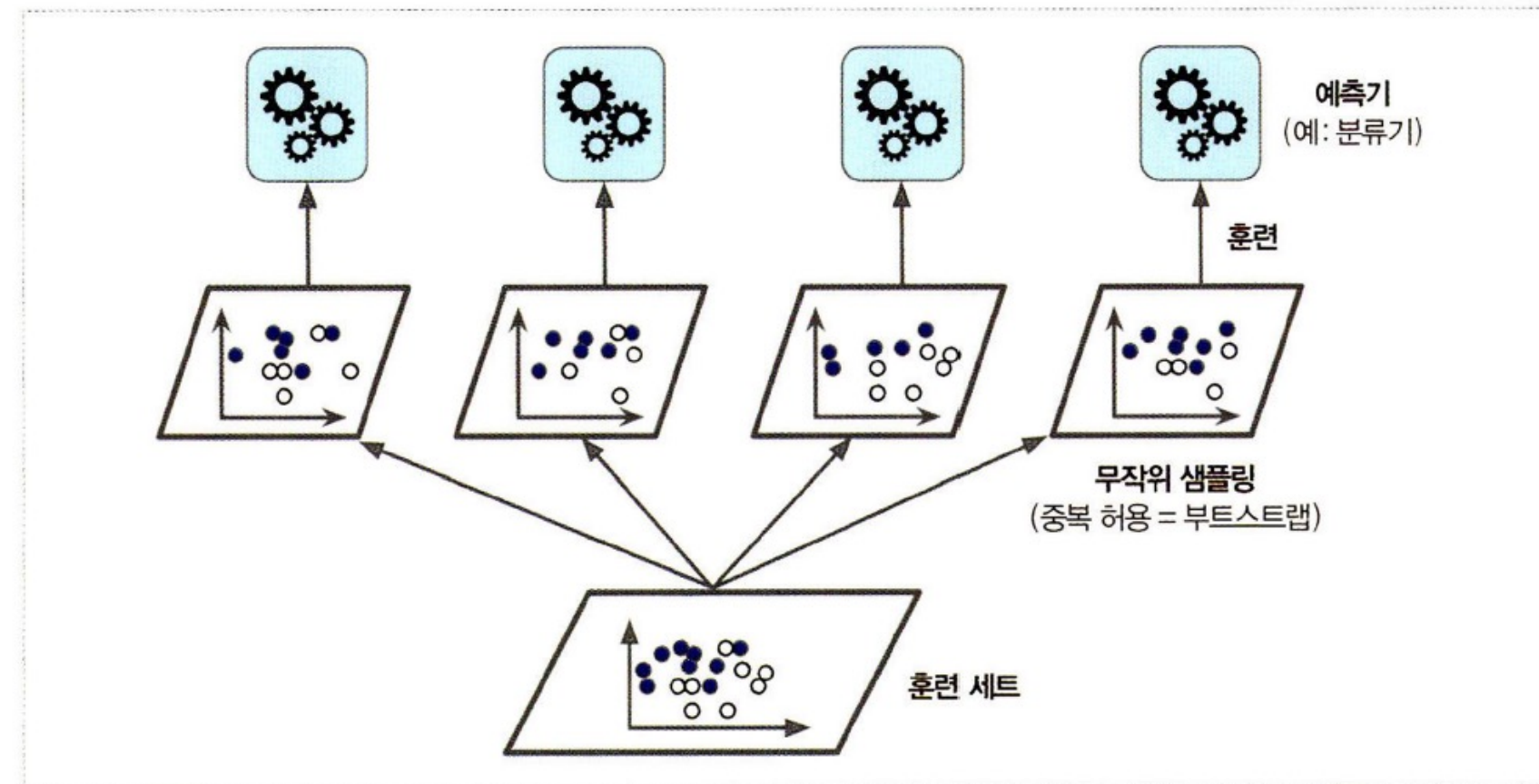


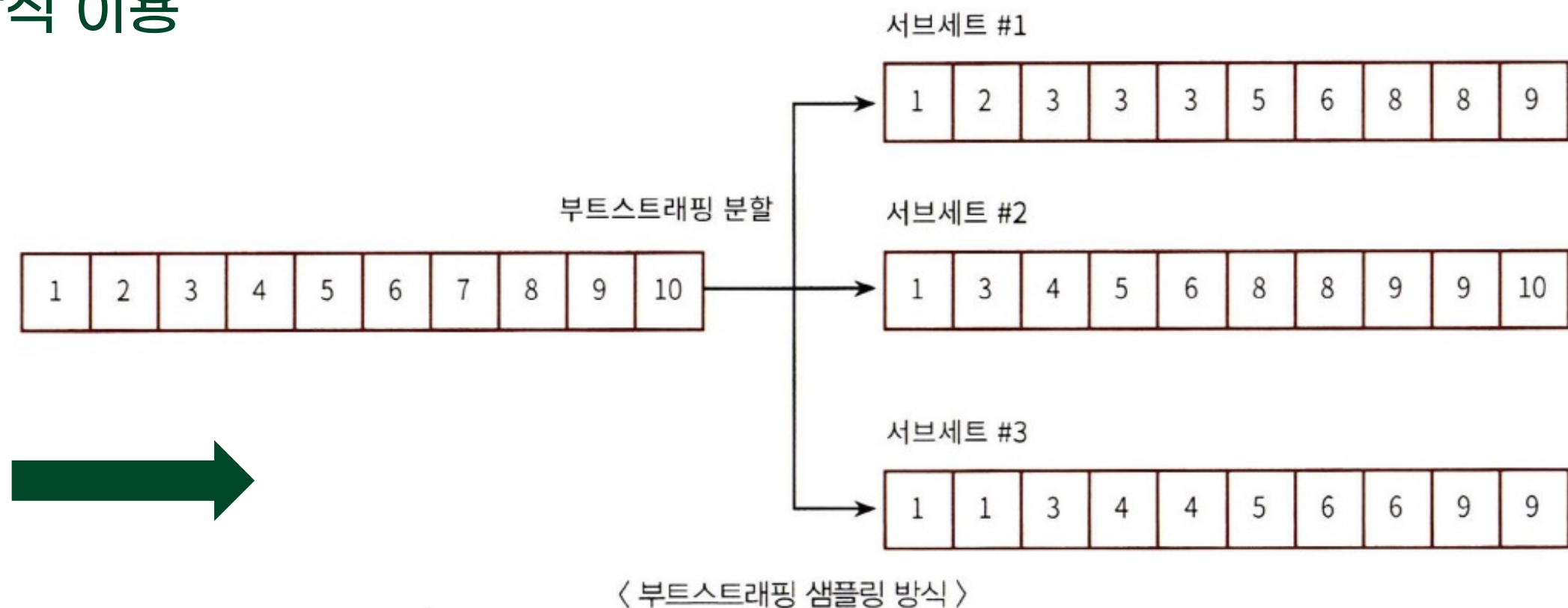
그림 7-4 배깅과 페이스팅은 훈련 세트에서 무작위로 샘플링하여 여러 개의 예측기를 훈련합니다.

## 부트 스트래핑 분할방식(Bagging)

개별 분류기에 할당된 학습 데이터는 원본 학습 데이터를 샘플링해 추출하는데,  
이렇게 개별 분류기에게 데이터를 샘플링해서 추출하는 방식

따라서 배깅에서는 부트스트래핑 방식 이용

원본 데이터의 건수가 10개인  
학습 데이터 세트에  
 $n\_estimators=3$ 으로 하이퍼  
파라미터를 부여하면 다음과 같이  
서브세트가 만들어진다.



사이킷런에서 분류의 경우 **BaggingClassifier** 제공  
회귀의 경우 BaggingRegressor

결정 트리 분류기 500개의 앙상블을 훈련시키는 코드)

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

Bootstrap=True  
#훈련세트에서 중복 허용  
n\_jobs=-1  
#가용한 모든 코어 사용

\* 페이스팅:  
중복을 허용하지 않는 샘플링 방식



# 배깅

## 결정 경계 비교

단일 결정 트리의 결정 경계 vs 500개의 트리 사용한 배깅 앙상블

- 앙상블의 예측이 일반화가 더 잘됨
- 앙상블은 비슷한 편향에서 더 작은 분산을 만듦

## Bootstrapping vs Pasting

부트스트래핑은 서브셋에 다양성 up  
-> 배깅이 페이스팅보다 편향 up

다양성  
-> 예측기들의 상관관계 down  
-> 앙상블 분산 down

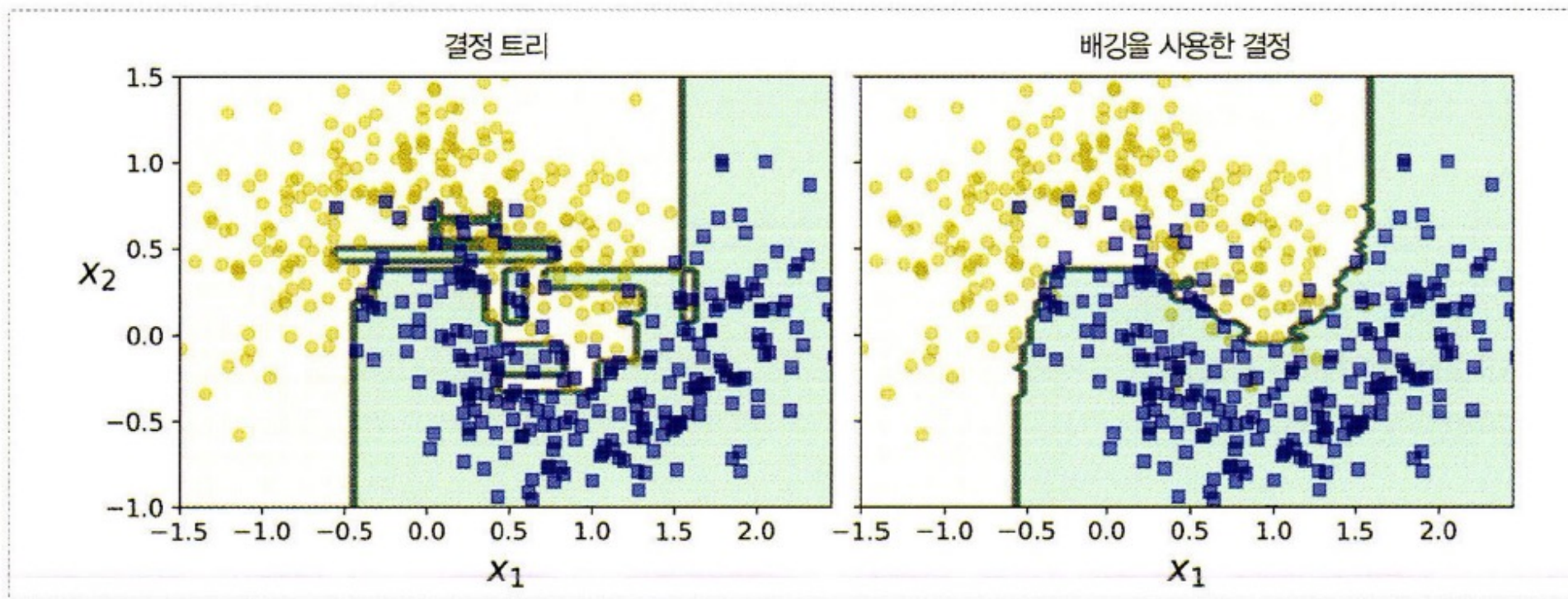


그림 7-5 단일 결정 트리(왼쪽)와 500개 트리로 만든 배깅 앙상블(오른쪽) 비교

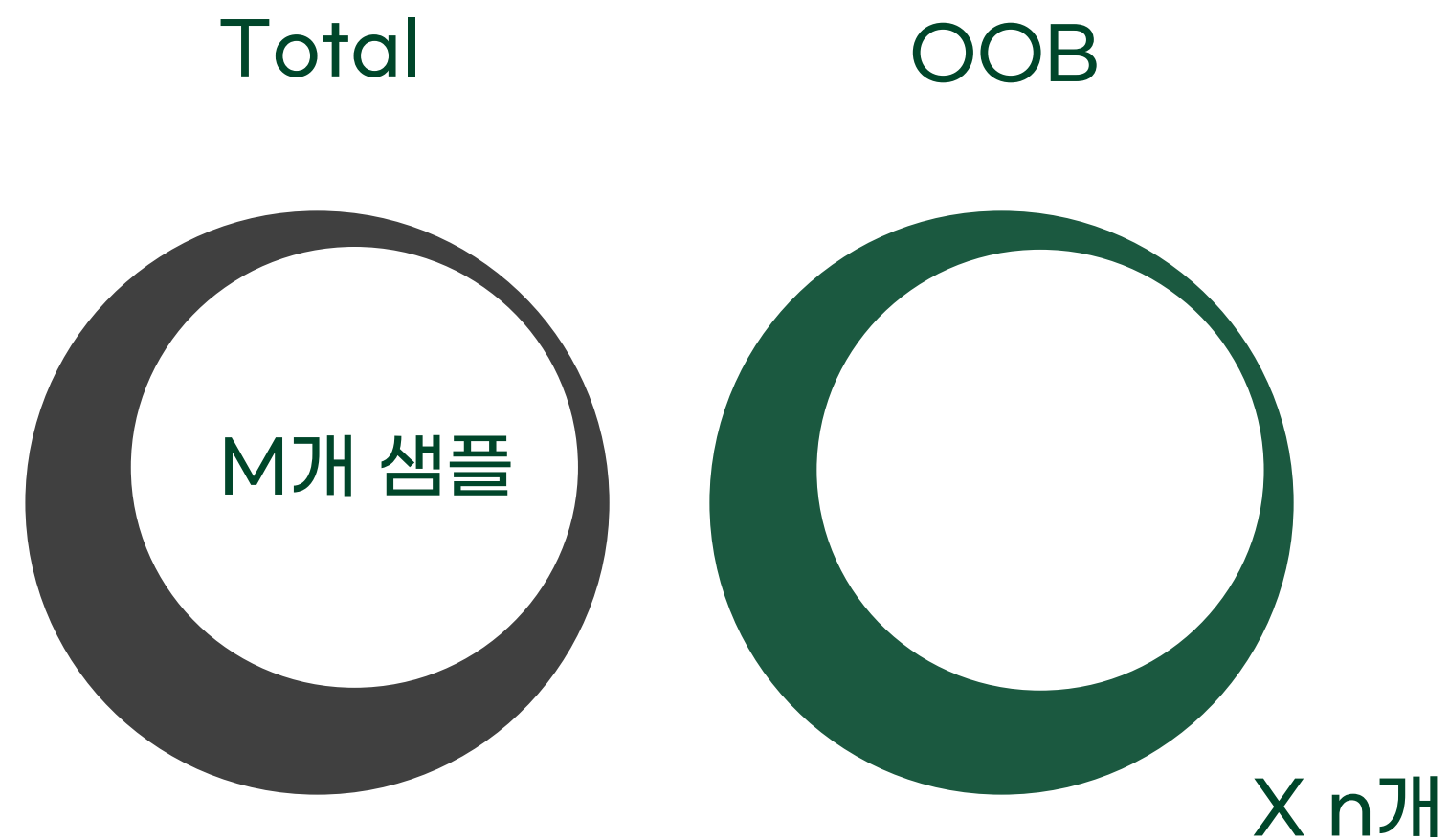
결론: 배깅 일반적 더 선호



# 배깅

## OOB(out-of-bag) 샘플

- 배깅 사용 시 선택되지 않은 훈련 샘플
- 평균적으로 각 예측기에 훈련 샘플의 63% 정도만 샘플링 되고 나머지 37% = oob 샘플



- 앙상블의 평가는 각 예측기의 oob 평가를  
평균해서 얻는다

$$(\bigcirc + \bigcirc + \bigcirc) / 3 = \text{앙상블의 평가}$$

- oob\_score=True -> 자동으로 oob 평가 수행

```
bag_clf = BaggingClassifier(  
    DecisionTreeClassifier(), n_estimators=500,  
    bootstrap=True, n_jobs=-1, oob_score=True)
```

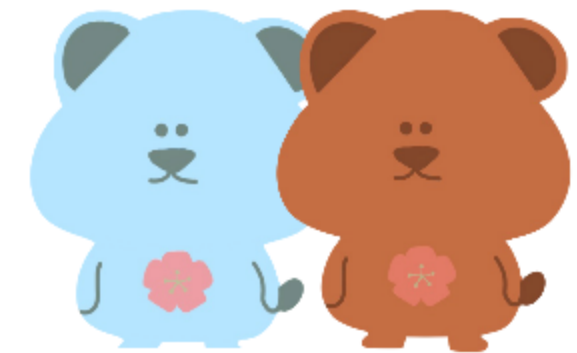
```
bag_clf.fit(X_train, y_train)  
bag_clf.oob_score_
```

```
0.9583333333333334
```

## 특성 샘플링

- BaggingClassifier는 특성 샘플링도 지원
  - 특성 샘플링에서 각 예측기는 무작위로 선택한 입력 특성의 일부분으로 훈련됨
  - max\_features, bootstrap\_features 두 매개변수로 조절
  - (이미지와 같은) 매우 고차원의 데이터셋을 다룰 때 유용
- 
- 랜덤 패치 방식 (random patches method):
    - 훈련 특성과 샘플을 모두 샘플링한 것
- 
- 서브스페이스 방식 (random subspaces method):
    - 훈련 샘플을 모두 사용하고 특성은 샘플링하는 것

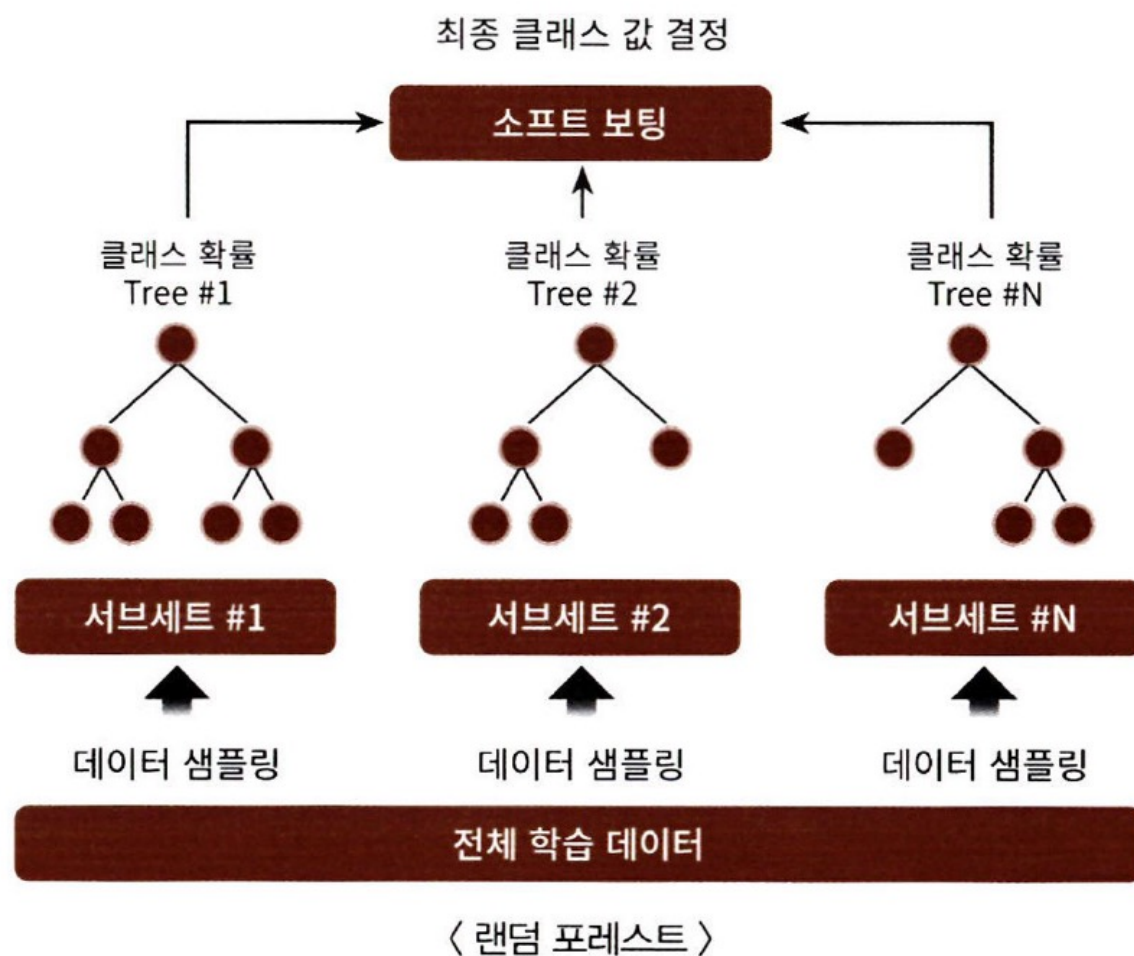
## 4.4 랜덤 포레스트



# 4.4 랜덤 포레스트

## 랜덤 포레스트

- 배깅의 대표적인 알고리즘
- 개별적인 분류기의 기반 알고리즘은 결정 트리이지만 개별 트리가 학습하는 데이터 세트는 전체 데이터에서 일부가 중첩되게 샘플링된 데이터 세트



랜덤 포레스트는 데이터가 중첩된 데이터 세트에 설정 트리 분류기를 각각 적용한다

# 4.4 랜덤 포레스트

사이킷런에서 RandomForestClassifier 제공

순서: 학습/테스트용 DataFrame 반환 ->  
랜덤 포레스트 학습 및 별도의 테스트 세트로 예측 성능 평가

```
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()를 이용해 학습/테스트용 DataFrame 반환
X_train, X_test, y_train, y_test = get_human_dataset()

#랜덤 포레스트 학습 및 별도의 테스트 세트로 예측 성능 평가
rf_clf = RandomForestClassifier(random_state=0)
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))
```

```
From sklearn.ensemble import
RandomForestClassifier
```

#랜덤 포레스트 정확도: 0.9108

# 4.4 랜덤 포레스트 하이퍼 파라미터 튜닝

트리 기반의 앙상블 알고리즘의 단점:

하이퍼 파라미터가 너무 많고, 튜닝 시간이 많이 소모된다

## GridSearchCV 이용해 랜덤 포레스트 하이퍼 파라미터 튜닝

```
from sklearn.model_selection import GridSearchCV

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

#RandomForestClassifier 객체 생성 후 GridSearchCV 수행
rf_clf = RandomForestClassifier(random_state=0, n_jobs=-1)
grid_cv = GridSearchCV(rf_clf, param_grid=params, cv=2, n_jobs=-1)
grid_cv.fit(X_train, y_train)

print('최적 하이퍼 파라미터:\n', grid_cv.best_params_)
print('최고 예측 정확도: {0:.4f}'.format(grid_cv.best_score_))
```

- n\_estimators = 100 -> 튜닝 시간 절약
- CV = 2 -> 최적 하이퍼 파라미터 구하기

```
최적 하이퍼 파라미터:
{'max_depth': 10, 'min_samples_leaf': 8, 'min_samples_split': 8, 'n_estimators': 100}
최고 예측 정확도: 0.9180
```



# SVM

도하연



# SVM 목차

---

**#01 SVM 개요**

**#02 선형 SVM 분류**

**#03 비선형 SVM 분류**

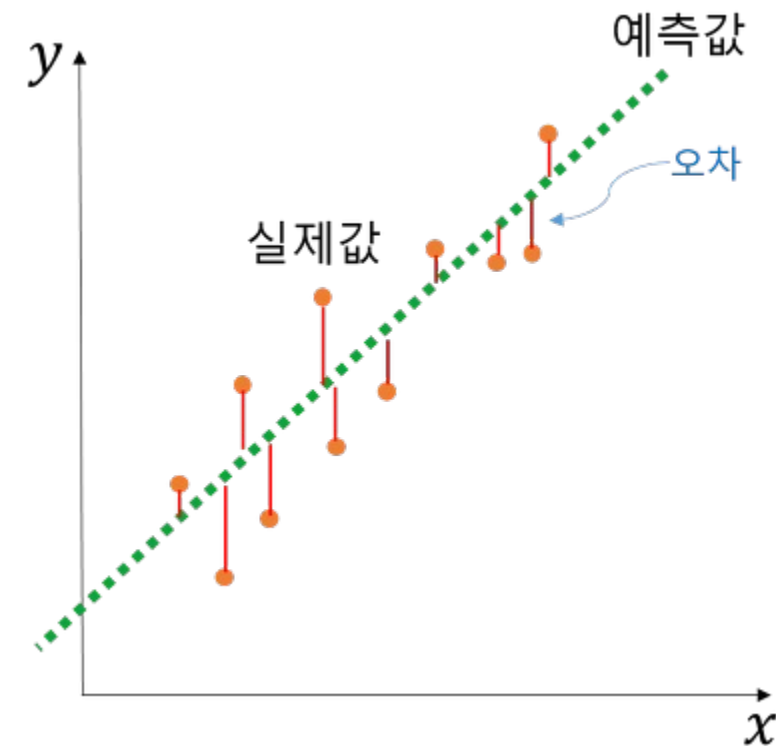
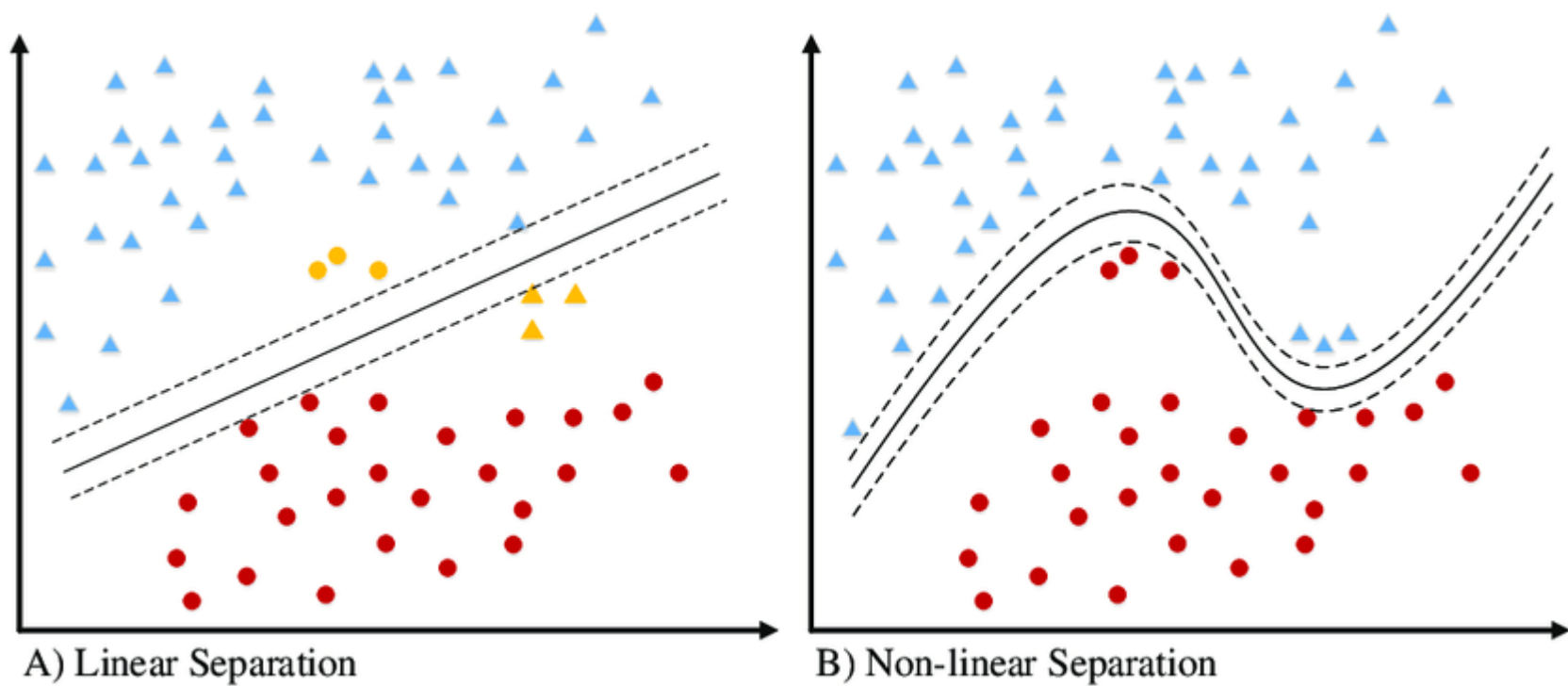


# SVM 개요



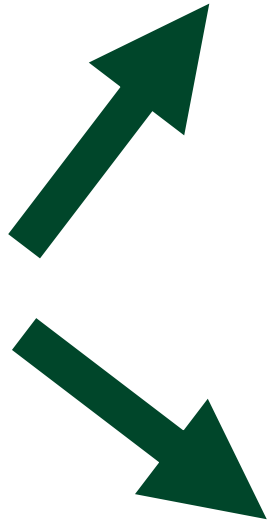
# #01 SVM 개요

- ✓ 선형, 비선형 분류, 회귀, 이상치 탐색에 사용할 수 있는 다목적 머신러닝 모델
- ✓ 복잡한 분류 문제 및 작거나 중간 크기의 데이터셋에 적합



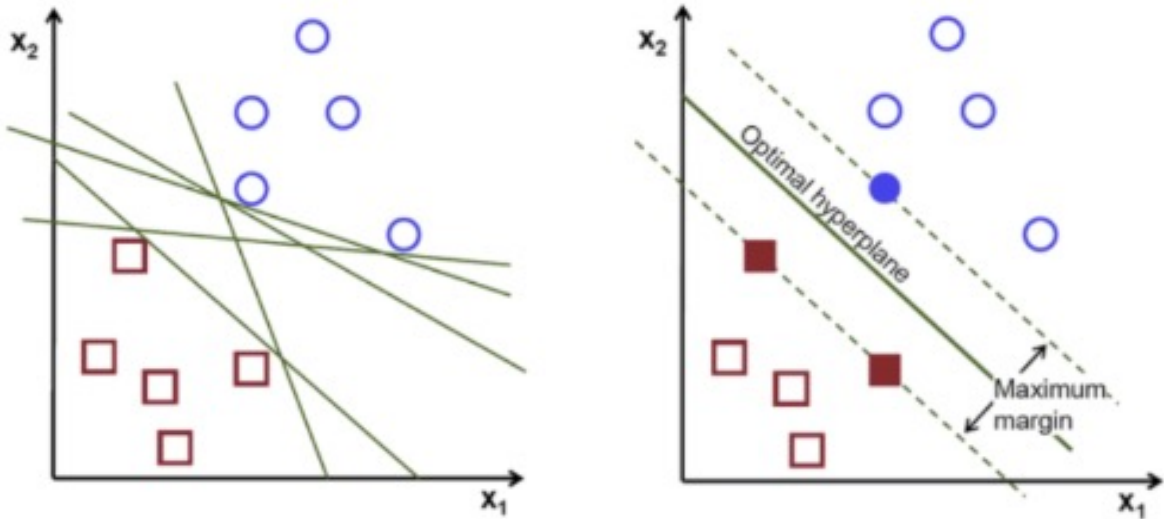
# #01 SVM 개요

SVM  
(Support Vector  
Machine)

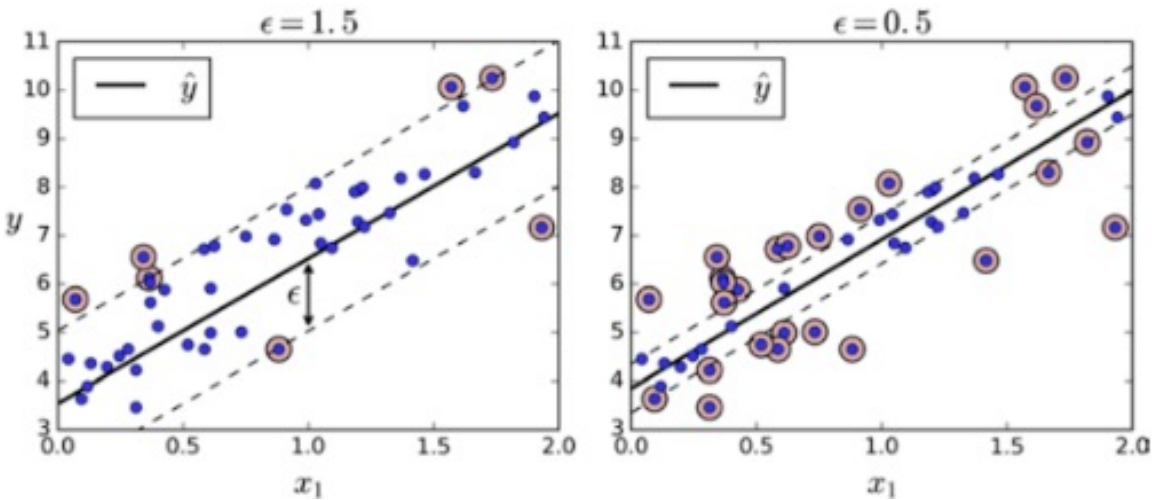


SVC  
(Support Vector  
Classification)

SVR  
(Support Vector  
Regression)



마진을 최대화하는 중간지점을 찾아  
각 그룹을 구분



마진 내부에 최대한 많은  
데이터가 들어가도록 학습

참고: <https://woono.tistory.com/111>  
<https://zephyrus1111.tistory.com/211>

# 선형 SVM 분류



# #02 선형 SVM 분류

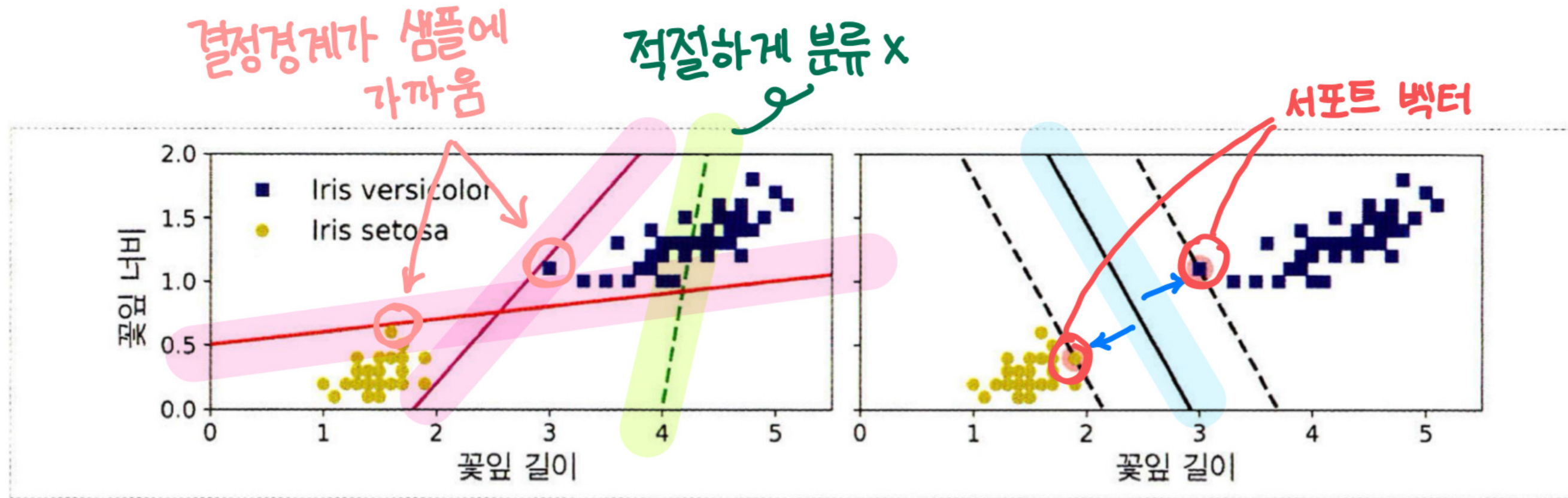


그림 5-1 라지 마진 분류 \* 선형 분류기의 결정 경계 \* SVM 분류기의 결정 경계

- 라지 마진 분류 -> 클래스 사이에 가장 폭이 넓은 도로 찾기
- 서포트 벡터 -> 도로 경계에 위치한 샘플에 의해 전적으로 결정

# #02 선형 SVM 분류

## ✓ 하드 마진 분류

- 모든 샘플이 도로 바깥쪽에 올바르게 분류되어 있음
- 데이터가 선형적으로 구분될 수 있어야 제대로 작동
- 이상치에 민감

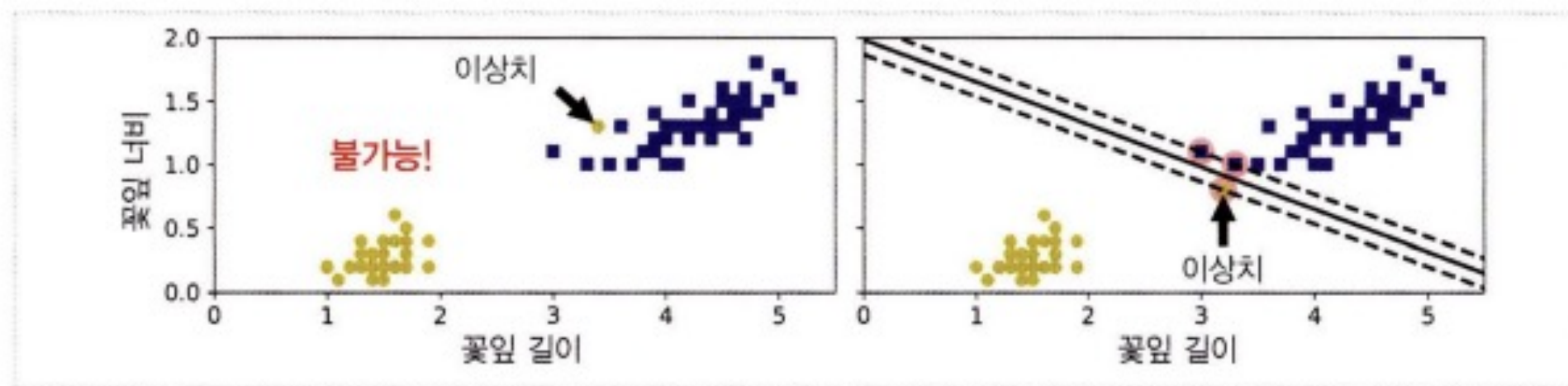


그림 5-3 이상치에 민감한 하드 마진



# #02 선형 SVM 분류

## ✓ 소프트 마진 분류

- 유연한 모델
- 도로의 폭 넓게 유지하기 & 마진 오류

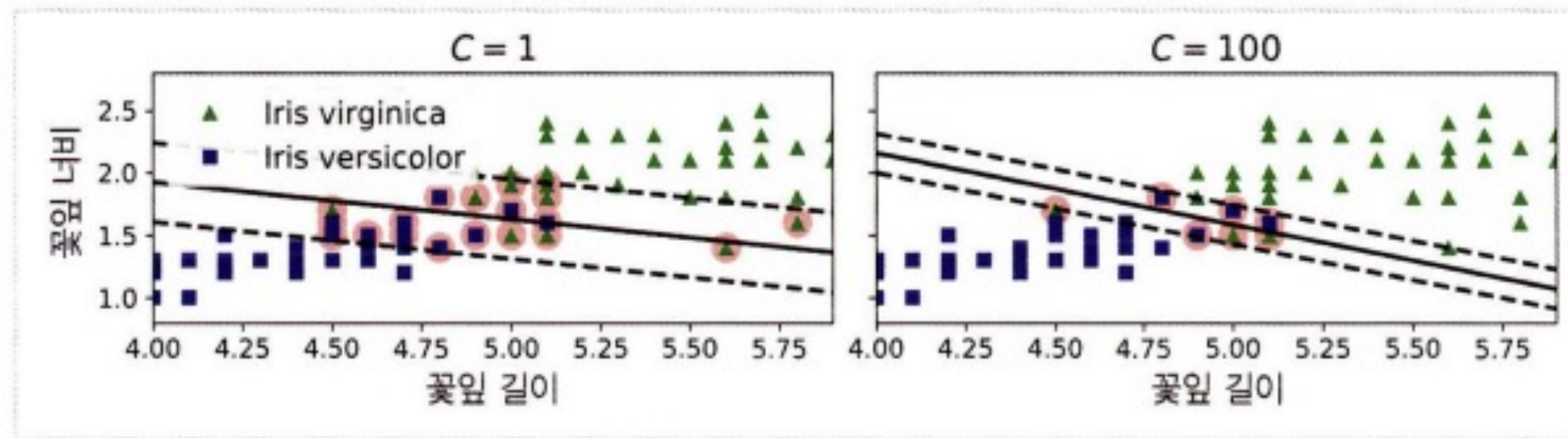


그림 5-4 넓은 마진(왼쪽) 대 적은 마진 오류(오른쪽)

# 비선형 SVM 분류



# #03 비선형 SVM 분류

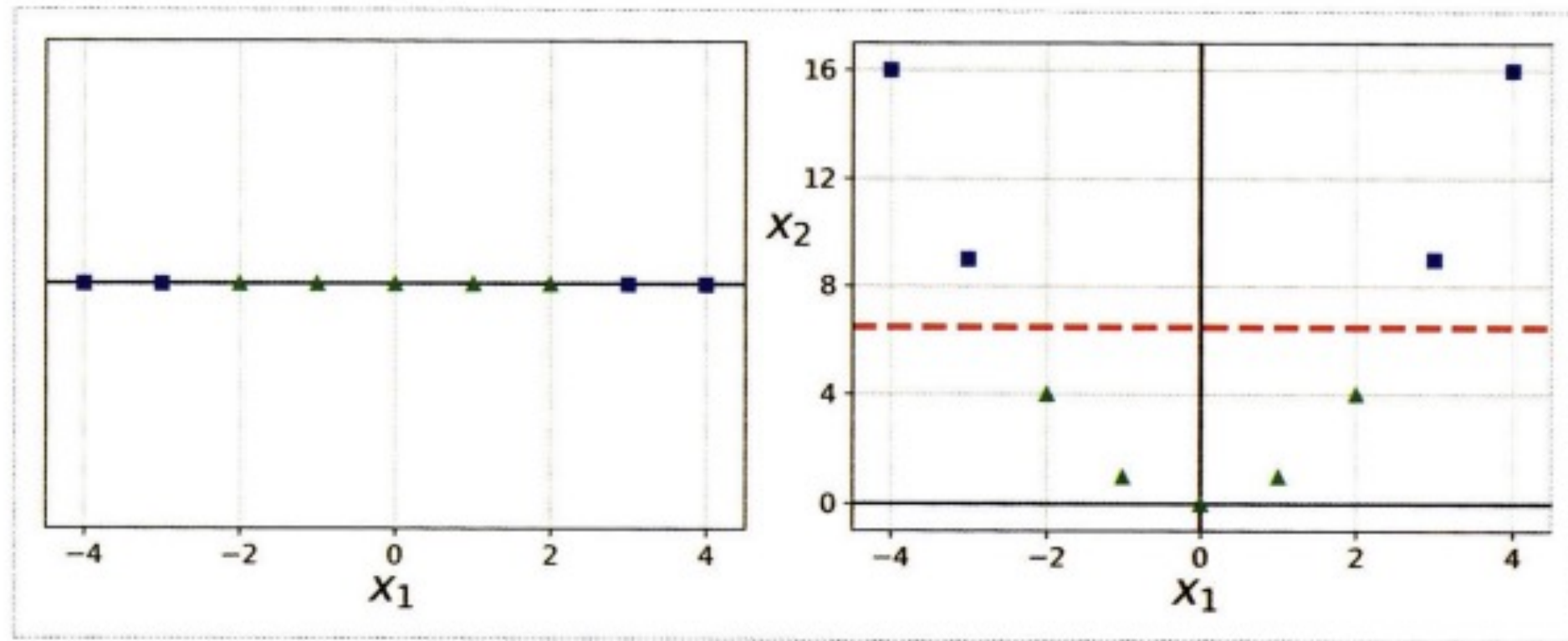


그림 5-5 특성을 추가하여 선형적으로 구분되는 데이터셋 만들기

- 선형적으로 분류 X
- 선형적으로 분류 O
- $X_1$
- $X_2 = (X_1)^2$



# #03 비선형 SVM 분류

## ✓ 다항식 커널

- 다항식 특성을 추가하는 것
- 낮은 차수의 다항식
  - > 복잡한 데이터셋 표현 X
- 높은 차수의 다항식 -> 속도 저하

## ✓ 커널 트릭

- 실제로는 특성을 추가하지 않으면서 다항식 특성을 추가한 것과 같은 결과
- Coef0 : 모델이 높은 차수와 낮은 차수에 얼마나 영향을 받을지 조절

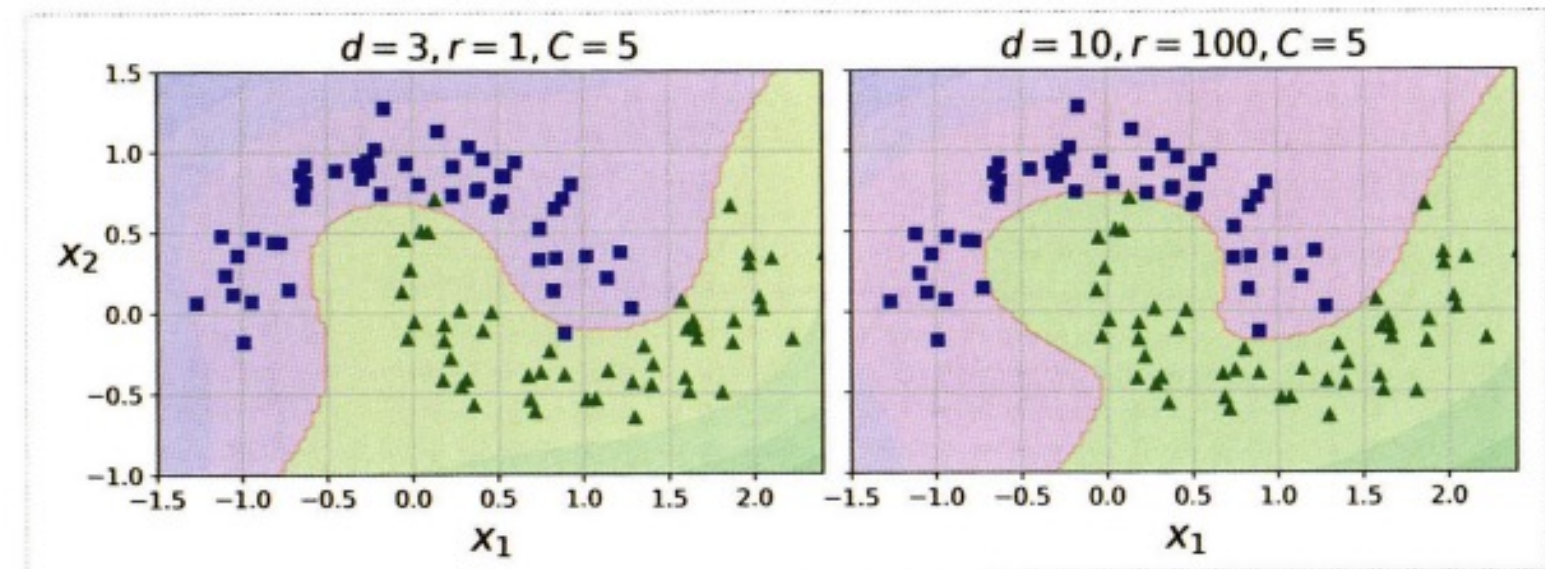


그림 5-7 다항식 커널을 사용한 SVM 분류기

# #03 비선형 SVM 분류

## ✓ 유사도 특성

- 각 샘플이 특정 랜드마크와 얼마나 닮았는지 특정하여 계산한 특성을 추가

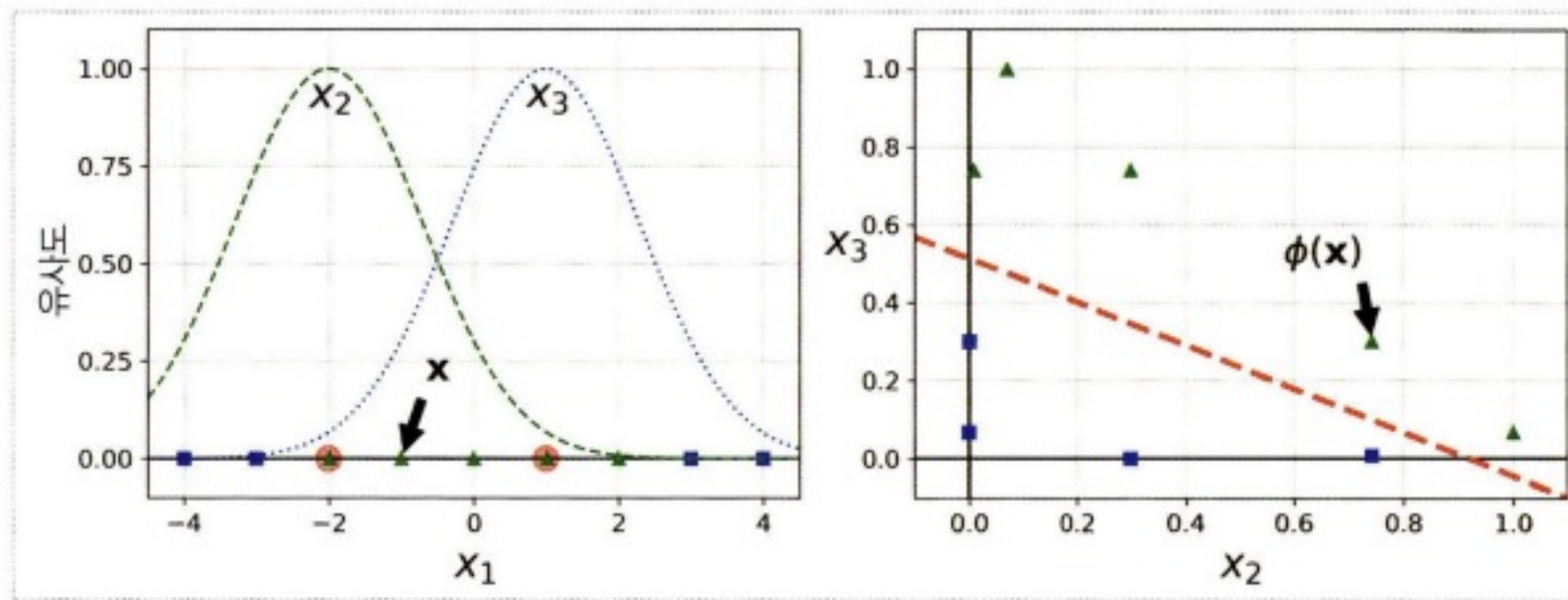


그림 5-8 가우시안 RBF를 사용한 유사도 특성



# #03 비선형 SVM 분류

## ✅ 가우시안 RBF 커널

- 가우시안 RBF 커널을 사용한 SVC 모델 -> 유사도 특성을 많이 추가하는 것과 비슷한 결과

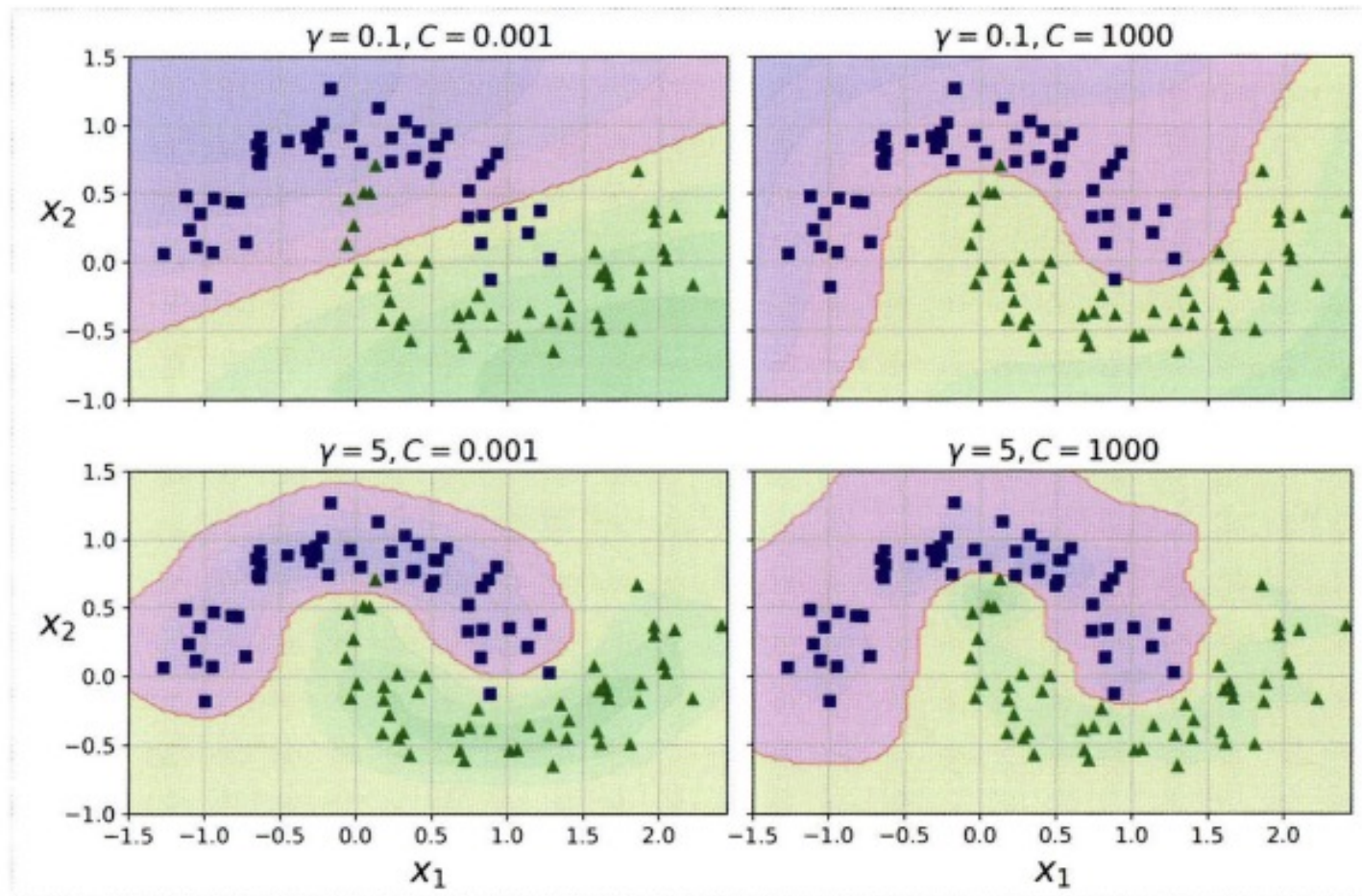


그림 5-9 RBF 커널을 사용한 SVM 분류기

gamma 값이 작으면 :

- 종 모양 그래프가 넓어짐
- 결정 경계가 부드러워진다.

gamma 값이 크면 :

- 종 모양 그래프가 좁아짐
- 결정 경계가 각 샘플을 따라 구불구불하게 휘어짐