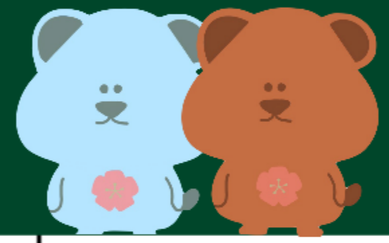
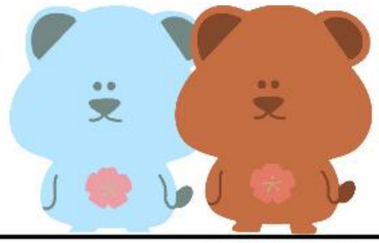


2주차 발표

김경민, 김도하, 남유림

EWHA EURON

목차



EWHA EURON

01.분류의 개요

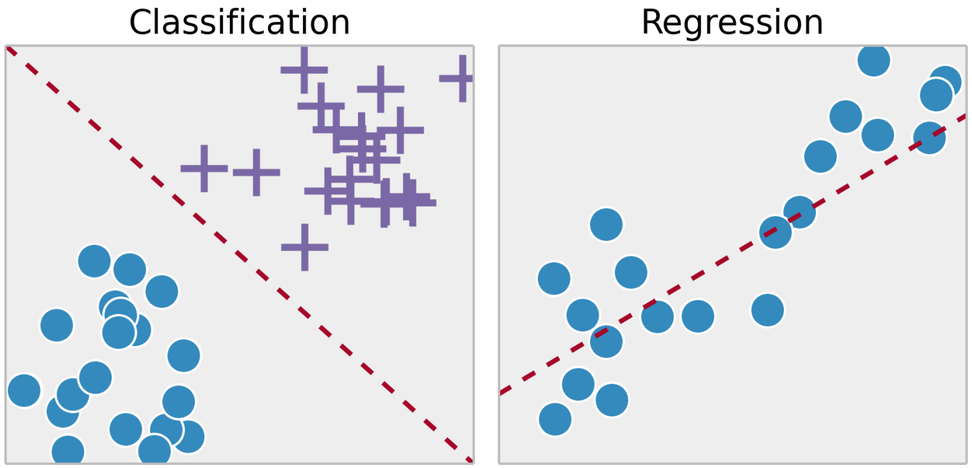
EWHA EURON

1.1 지도학습 VS 비지도학습

지도학습(Supervised Learning)	비지도학습(Unsupervised Learning)
명확한 정답이 주어진 데이터를 먼저 학습한 뒤 미지의 정답을 예측하는 방식	모델에 **정답 없이 데이터만** 제공한 상태에서 데이터의 패턴을 알아내는 방식
입력값(X data)가 주어지면 입력값에 대한 Label(Y data)를 주어 학습시킴	정답 라벨이 없는 데이터를 비슷한 특징끼리 **군집화**하여 새로운 데이터에 대한 결과를 예측하는 방법
강아지 사진 X(input data)를 주고 이 사진은 강아지라고 분류해주어 Y값 Label을 알려줌	여러 자동차의 사진을 분류하지 않고 많이 준비한 뒤 자동차 데이터를 머신러닝에게 주어 학습시킴
분류, **회귀**, 추천 시스템, 시각/음성 감지/인지, 텍스트 분석, NLP	**클러스터링**, 차원 축소

1.2 지도학습-분류, 회귀

<table>
<tr><th colspan="2">분류(Classification)</th><th>회귀(Regression)</th></tr>
<tr><td colspan="2">레이블이 연속되지 않고 서로 단절되어 있는
이산적인 값의 카테고리를 구분하는 문제</td><td>예측해야 하는 레이블이 연속적인 값을
갖고 있을 때 연속된 값을 예측</td></tr>
<tr><td>이진분류</td><td>다중분류</td><td rowspan="3">ex. 체중 데이터를 바탕으로 신장 예측</td></tr>
<tr><td>Y/N처럼 두가지의 답으로 분류</td><td>이진분류에서 답의 개수만 증가한 분류의 형태</td></tr>
<tr><td>ex.
이미지 데이터를 바탕으로
강아지 사진인지,
고양이 사진인지 분류</td><td>ex. 식물 종 분류</td></tr>
</table>



*분류의 다양한 머신러닝 알고리즘
: 나이브 베이즈 로지스틱 회귀, 결정 트리,
서포트 벡터 머신, 최소 근접 알고리즘, 신경망, 앙상블

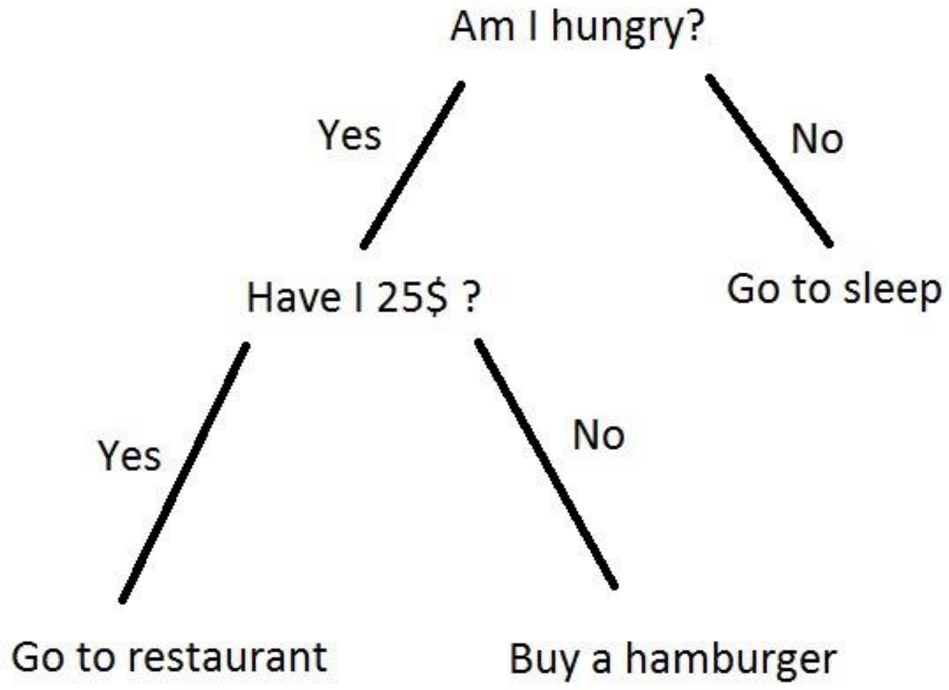
02.결정 트리

EWHA EURON

2.1 결정 트리

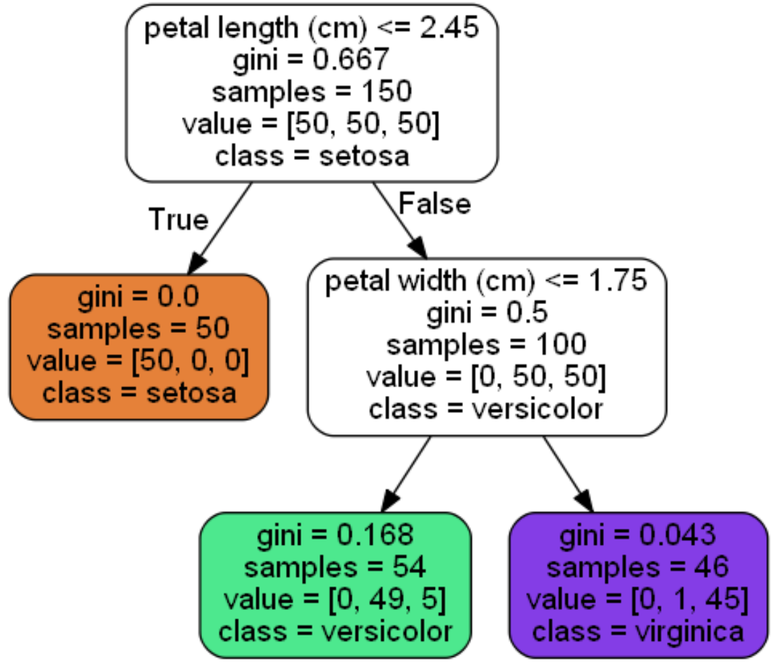
결정 트리(Decision Tree)

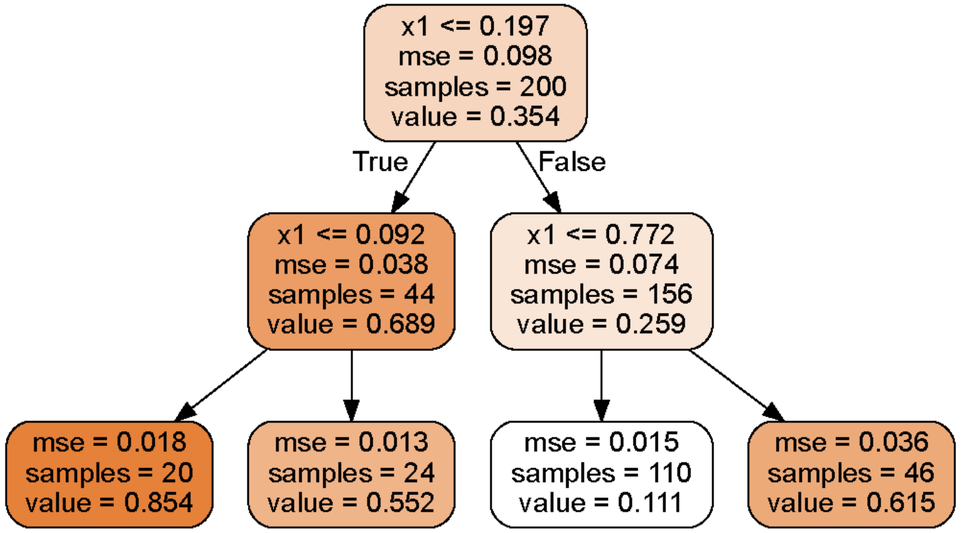
: 데이터에 있는 **규칙**을 학습을 통해 자동으로 찾아내 트리(Tree) 기반의 분류 규칙을 만드는 것
: **데이터의 어떤 기준을 바탕으로 규칙을 만들어야 가장 효율적인 분류가 될 것인가**가
알고리즘의 성능을 크게 좌우함



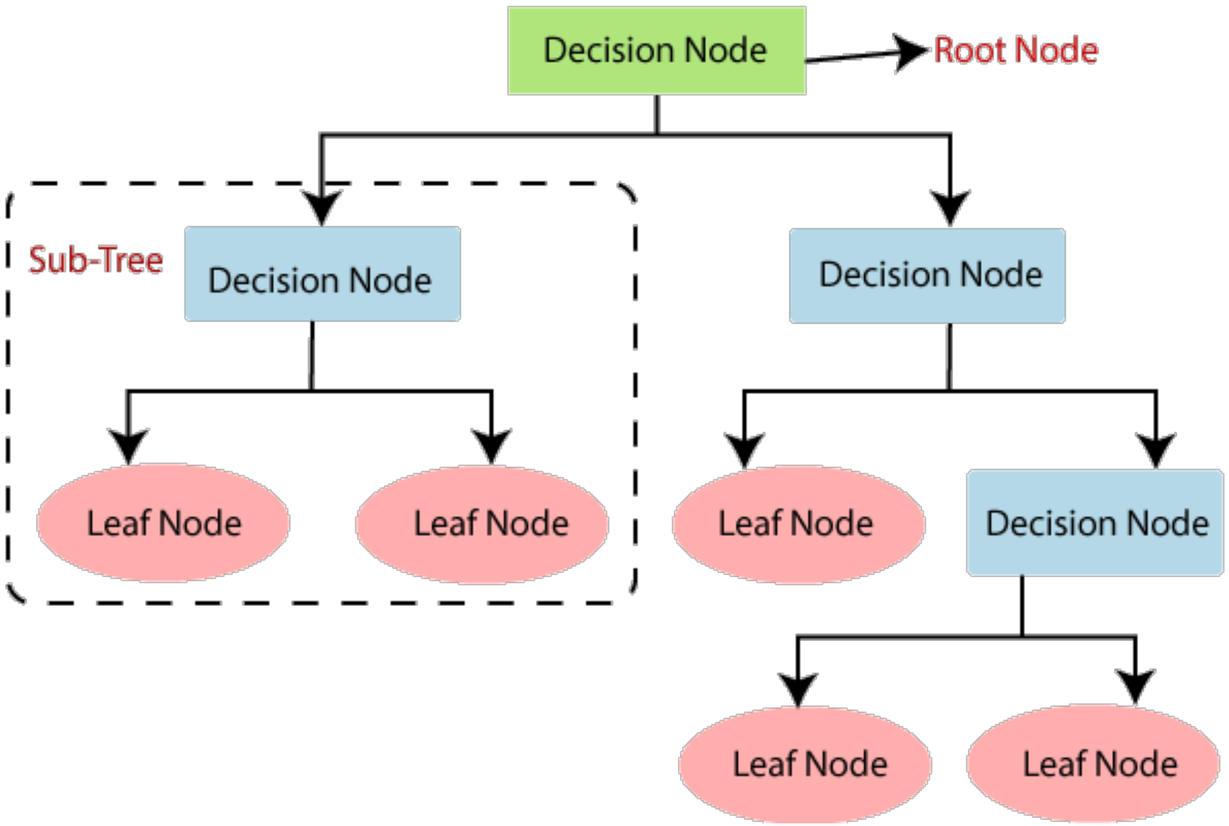
2.2 결정 트리의 종류

DecisionTreeClassifier	DecisionTreeRegressor
분류를 위한 결정 트리	회귀를 위한 결정 트리
리프 노드에 속한 특정 클래스의 레이블을 결정	각 노드에서 클래스를 예측하는 대신 어떤 값을 예측
불순도를 최소화하는 방향으로 분할	MSE를 최소화하는 방향으로 분할





2.3 결정 트리의 구조



루트 노드
: 깊이가 0인 맨 위의 노드

규칙 노드(Decision Node)
: 규칙 조건을 만드는 노드
: 자식 노드로 분할됨

리프 노드(Leaf Node)
: 최종 클래스(레이블) 값이 결정되는 노드

브랜치/서브 트리(Sub Tree)
: 새로운 규칙 조건마다 생성됨

*트리의 깊이(depth)가 깊어질수록(= 규칙이 많을수록)
결정 트리의 예측 성능이 저하될 가능성↑

2.3 결정 트리의 구조

생성 과정 및 규칙
: Root node → Leaf nodes
: **정보 균일도가 높은 데이터 세트**로 **쪼개질 수 있도록 조건**을 찾아 서브 데이터 세트를 만들고
다시 서브 데이터 세트에서 균일도가 높은 자식 데이터 세트로 쪼개는 방식을 자식 트리로 내려가면서
반복하는 방식으로 데이터 값을 예측
: Root 노드는 모든 데이터를 고려한다고 가정하며 하위 노드(자식 노드)로 내려가며 데이터들이 분류됨

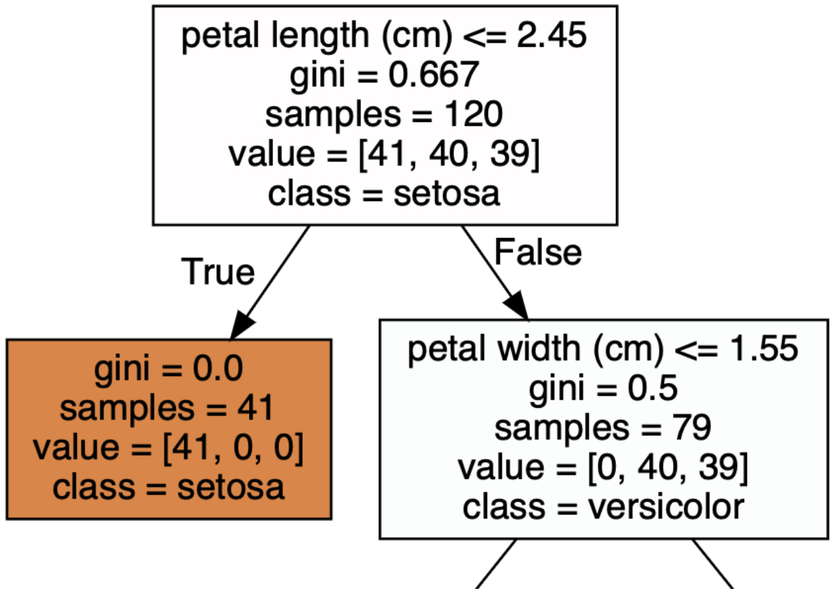
1. 노드에서 고려할 데이터가 **이미 하나의 Class**에만 속해 있거나,
더 이상 고려할 feature가 없으면 Leaf node로 여기고 자식 노드를 생성하지 않음
→ 고려할 feature가 더 이상 없어서 leaf 노드가 된 경우, 가장 많은 수의 class를 결과 값으로 채택

2. 각 노드에서 고려할 feature 선택 시, **데이터들을 가장 잘 나눠주는 feature를 선택**
이때 Leaf node에서 **완벽**하게 데이터들이 나눠진다는 **보장X**

3. 선택된 feature에 대한 **조건 별로(값마다) 자식 노드를 생성**합니다.

4. 각 자식 노드에서는 해당 조건을 만족하는 데이터만 고려하여 1부터 **반복**

2.3 결정 트리의 구조



1. Petal Length <=2.45 **규칙**으로 **자식 노드 생성**

2. 깊이 1의 오른쪽 리프 노드에서 **모든 데이터**가 Setosa로 결정되므로 **클래스가 결정된 리프 노드**가 되고 더 이상의 규칙을 만들 필요X

3. 깊이 1의 왼쪽 리프 노드에서는 **정보 균일도가 높은 데이터 세트**로 **쪼개질 수 있도록 조건에 따라 분할하여** 다시 리프 노드를 만듦

2.4 결정 트리의 성능

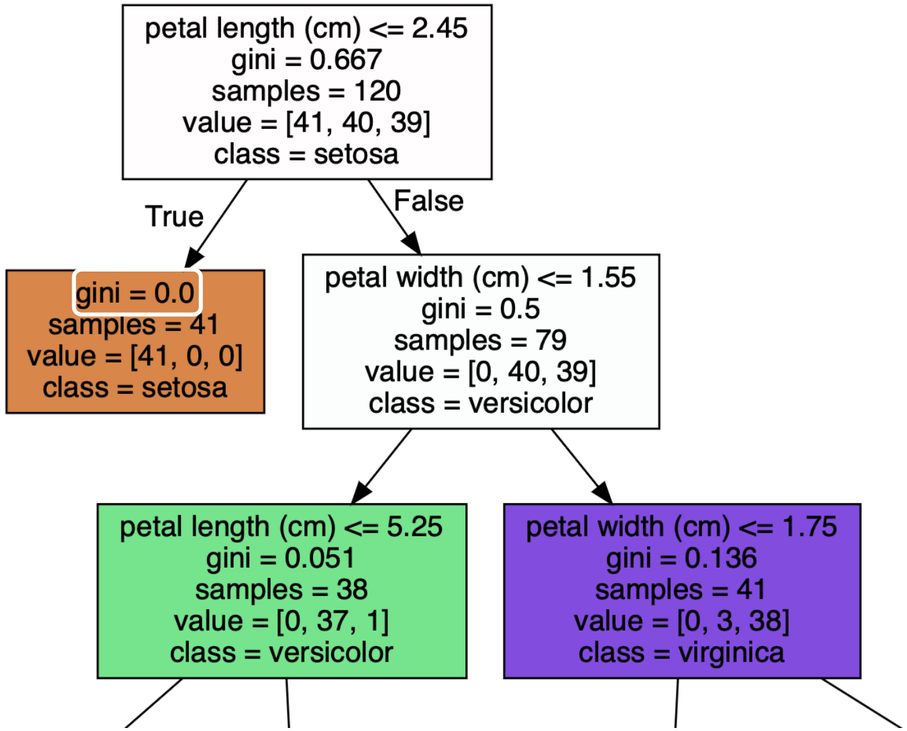
- **최대한 균일한 데이터 세트**를 구성할 수 있도록
즉, 최대한 많은 데이터 세트가 해당 분류에 속할 수 있도록 트리를 **분할(Split)**하는 것이 중요

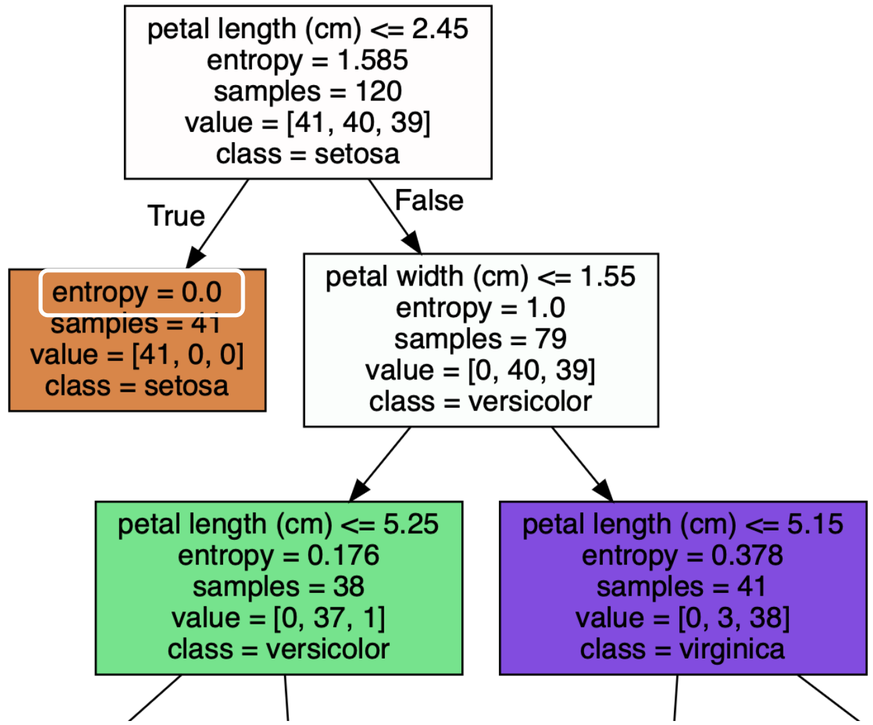
- 균일도(불순도:Impurity)
  : 복잡성을 의미
  : 해당 범주 안에 서로 다른 데이터가 얼마나 섞여 있는 지를 뜻함
  : 다양한 개체들이 섞여 있을수록 불순도가 높아짐

- 정보의 균일도를 측정하는 대표적인 두가지 방법

정보 이득(Information Gain) 지수	**지니 계수**
데이터 집합의 혼잡도인 엔트로피 이용	경제학에서 불평등 지수 나타낼 때 사용됨
서로 같은 값 섞여 있음 → 엔트로피 ↓	0이 가장 평등, 1로 갈수록 불평등
1 – 엔트로피 : **높을수록 데이터 균일**	지니 계수가 **낮을수록 균일도가 높음**

2.4 결정 트리의 성능 - 균일도





DecisionTreeClassifier( )
DecisionTreeClassifier(**criterion**, splitter, max_depth, min_samples_split, min_samples_leaf, min_weight_fraction_leaf, max_features, random_state, max_leaf_nodes, min_impurity_decrease, min_impurity_split, class_weight, presort)
*criterion를 통해 지니 계수 또는 엔트로피(entropy)를 기준으로 한 결정 트리를 출력할 수 있음 (default : criterion = gini)
*critierion에 따라 규칙 조건, 균일도 모두 차이가 나타남

2.5 결정 트리의 특징

장점	단점
ML 알고리즘 중 직관적이며 이해하기 쉬움	약한 학습기
데이터의 스케일링이나 정규화 등의 사전 가공의 영향이 매우 적음	예측 성능을 향상 시키기 위해 복잡한 규칙 구조를 가져야 함
시각화 가능하므로 분류 과정 파악 용이	피처多, 균일도↓ → 트리의 깊이 깊어짐
회귀/분류, 수치형/범주형 데이터에 모두 사용 가능	과적합이 발생해 예측 성능이 저하될 가능성이 있음
∴ 트리의 크기를 사전에 제한하는 것이 성능 튜닝에 도움이 됨	

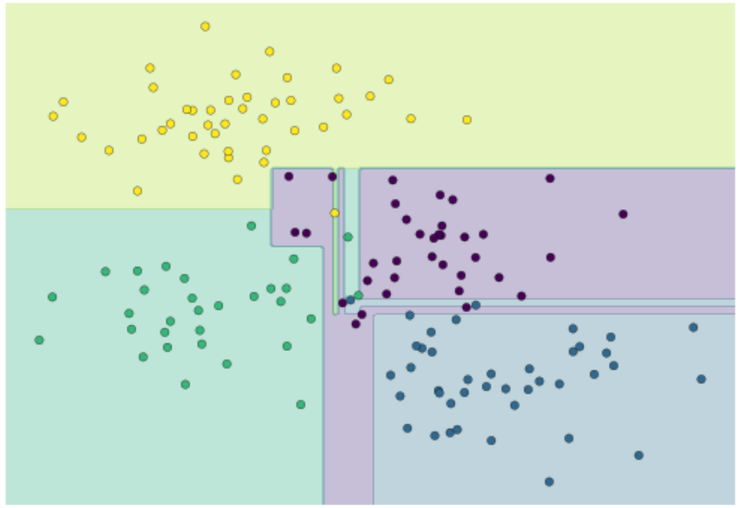
2.6 결정 트리의 파라미터

min_samples_split	- 노드를 분할하기 위한 최소한의 샘플 데이터 수 - 과적합 제어 - 디폴트는 2, 작게 설정할수록 분할되는 노드 ↑ → 과적합 가능성 ↑
min_saples_leaf	- 밑단 노드 Leaf 가 되기 위한 최소한의 샘플 데이터 수 - 과적합 제어 - 비대칭적 데이터의 경우 특정 클래스의 데이터가 작을 가능성0 → 작게 설정
max_feature	- 최적의 분할을 위해 고려할 최대 피처 개수(디폴트 = None, 모든 피처를 사용해 분할) - int 형 → 대상 피처의 개수, float 형 → 전체 피처 중 대상 피처의 퍼센트 - 'sqrt' : 전체 피처 중 sqrt(전체 피처 개수), 'auto' 로 지정하면 sqrt 와 동일 - 'log' 는 전체 피처 중 log2 (전체 피처 개수) 선정
max_depth	- 트리의 최대 깊이를 규정 - 디폴트 = None : 완벽히 클래스 결정 값이 될 때까지 깊이 계속 분할/노드가 가지는 데이터 수가 min_samples_split 보다 작아질 때까지 계속 깊이를 증가시킴 *과적합할 수 있으므로 적절한 값으로 제어 필요
max_leaf_nodes	밑단 노드 Leaf 의 최대 개수

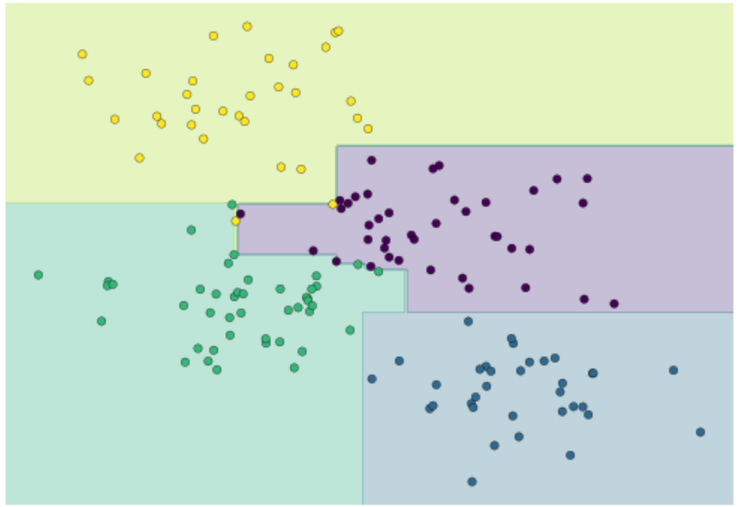
* min으로 시작하는 매개변수를 증가시키거나 max로 시작하는 매개변수를 감소시키면 모델에 규제가 커짐

2.7 과적합(Overfitting)

과적합(Overfitting)
: 모델이 학습 데이터에만 과도하게 최적화되어, 실제 예측을 다른 데이터로 수행할 경우에는 예측 성능이 떨어지는 경우



엄격한 분할 기준



일반화된 분류 규칙

엄격한 분할 기준 → 결정 기준 경계 **많아지고 복잡**
⇒ 학습 데이터 세트의 특성과 약간만 다른 형태의 데이터 세트를 예측하면 예측 정확도 ↓
이상치에 크게 반응하지 않으면서 좀 더 **일반화된 분류 규칙**에 따라 분류 → 예측 성능이 더 뛰어날 가능성 ↑
학습 데이터에만 지나치게 최적화된 분류 기분은 오히려 테스트 데이터 세트에서 정확도 떨어뜨릴 수 있기 때문
⇒ 하이퍼 파라미터의 변경을 통해 결정 기준 경계를 완화

2.8 결정 트리 실습 – 사용자 행동 인식

1. 데이터 전처리(.txt → DataFrame, 중복된 피처명 → 새로운 피처명을 가지는 DataFrame으로 반환
2. 학습/테스트용 DataFrame 로딩 및 반환
3. DecisionTreeClassifier를 이용해 동작 예측 분류 수행 → 학습, 예측, 평가
4. 결정 트리 정확도 성능 튜닝 : GridSearchCV를 이용해 max_depth를 조절하며 예측 성능 확인
5. feature_importatnces_속성을 통해 결정 트리에서 각 피처의 중요도 알아보기

03.앙상블 학습

EWHA EURON

3.1 학습 개요

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

- 정형 데이터 분류 시에는 앙상블이 딥러닝 보다 뛰어난 성능을 나타냄.
  (ex. 랜덤 포레스트, 그래디언트 부스팅, XGBoost, LightGBM, 스태킹(Stacking))

- 학습 유형은 보팅(Voting), 배깅(Bagging), 부스팅(Boosting)의 세가지로 나눌 수 있으며, 이 외에도 스태킹을 포함한 앙상블 기법이 있음.

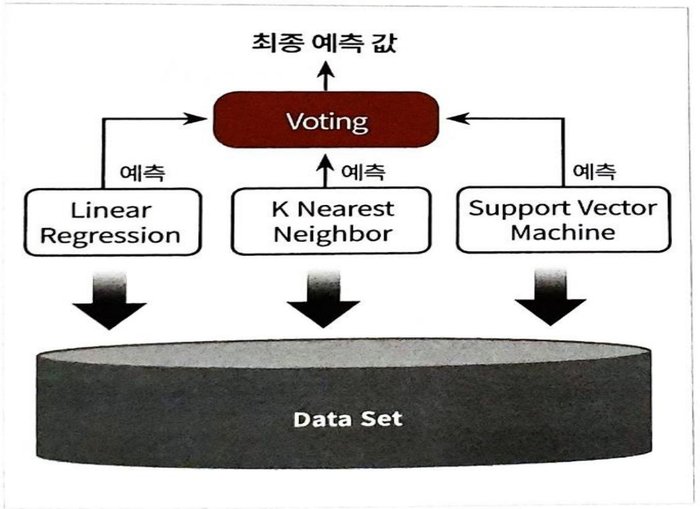
3.1 학습 개요

보팅 : 일반적으로 서로 다른 알고리즘(선형 회귀, K 최근접 이웃, 서포트 벡터 머신)을 가진 분류기를 결합하는 것. 같은 데이터 세트에 대해 학습.

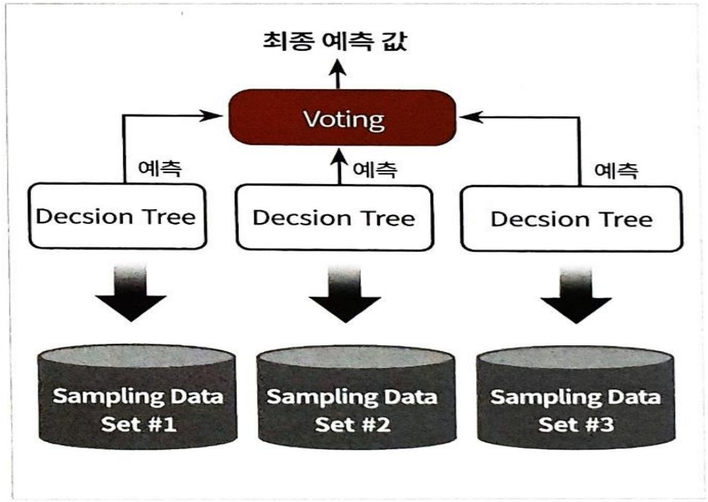
배깅 : 각각의 분류기가 모두 같은 유형의 알고리즘 기반이지만, 데이터 샘플링을 서로 다르게 가져가면서 학습을 수행. (+ 중첩 허용)

ex) 랜덤 포레스트 알고리즘.

cf. 부스트스트래핑(Booststrapping) : 개별 Classifier에게 데이터를 샘플링해서 추출하는 방식



Voting 방식



Bagging 방식

3.1 학습 개요

부스팅 : 여러 개의 분류기가 순차적으로 학습을 수행하되, 앞에서 학습한 분류기가 예측이 틀린 데이터에 대해서는 올바르게 예측할 수 있도록 다음 분류기에는 가중치를 부여하면서 학습과 예측을 진행.

ex) 그래디언트 부스트, XGBoost(eXtra Gradient Boost), LightGBM(Light Gradeint Boost)

스태킹 : 여러 가지 다른 모델의 예측 결괏값을 다시 학습 데이터로 만들어서 다른 모델(메타 모델)로 재학습 시켜 결과를 예측.

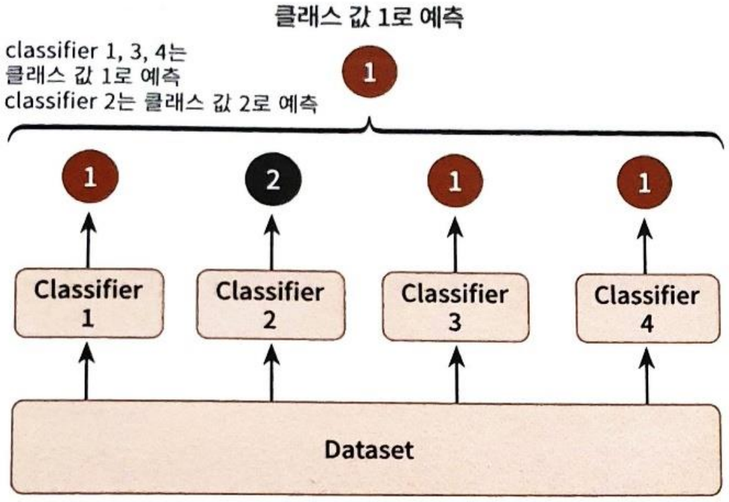
3.2 보팅 유형(Hard Voting & Soft Voting)

1. 하드 보팅(Hard Voting)

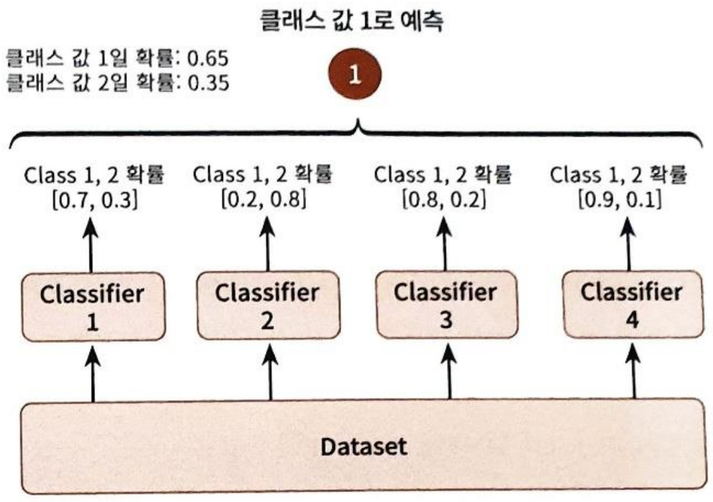
- 예측한 결괏값들 중 다수의 분류기가 결정한 예측값을 최종 보팅 결괏값으로 선정.

2. 소프트 보팅(Soft Voting)

- 분류기들의 레이블 값 결정 확률을 모두 더하고 이를 평균해서 이들 중 확률이 가장 높은 레이블 값을 최종 보팅 결괏값으로 선정. 일반적으로 소프팅 보팅이 하드 보팅보다 예측 성능이 좋아서 더 많이 사용됨.



<하드 보팅>



<소프트 보팅>

3.3 보팅 분류기

위스코신 유방암 데이터 세트 예측 분석

- 유방암의 악성종양, 양성종양 여부를 결정하는 이진 분류 데이터 세트
- 종양의 크기, 모양 등의 형태와 관련된 많은 피처를 가짐

(1) 모듈 및 데이터 로딩

```
import pandas as pd

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

cancer = load_breast_cancer()

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

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points
0	17.99	10.38	122.8	1001.0	0.11840	0.27760	0.3001	0.14710
1	20.57	17.77	132.9	1326.0	0.08474	0.07864	0.0869	0.07017
2	19.69	21.25	130.0	1203.0	0.10960	0.15990	0.1974	0.12790

3 rows × 30 columns

리스트 값으로 보팅에 사용될 여러 개의 Classifier 객체들을 튜플 형식으로 입력 받음.

(2) 보팅 분류기,로지스틱 회귀, KNN을 이용한 정확도 산출

hard를 입력하면, 하드 보팅
soft를 입력하면 소프트 보팅
(기본은 'hard')

```
# 개별 모델은 로지스틱 회귀와 KNN임.
lr_clf = LogisticRegression(solver="liblinear")
knn_clf = KNeighborsClassifier(n_neighbors = 8)

# 개별 모델을 소프트 보팅 기반의 앙상블 모델로 구현한 분류기
vo_clf = VotingClassifier(estimators= [("LR", lr_clf), ("KNN",knn_clf)] ,voting = 'soft')

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target, test_size = 0.2,random_state=156)

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

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

```
Voting 분류기 정확도:0.9561
LogisticRegression 정확도: 0.9474
KNeighborsClassifier 정확도: 0.9386
```

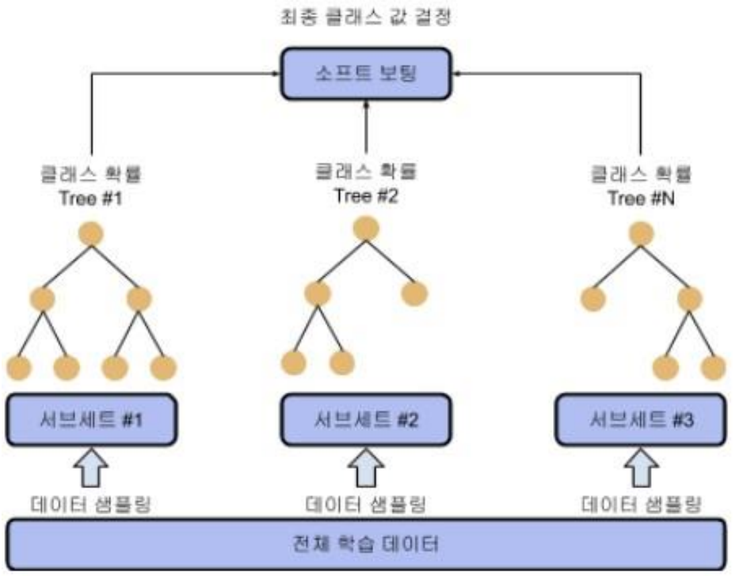
04. 랜덤 포레스트

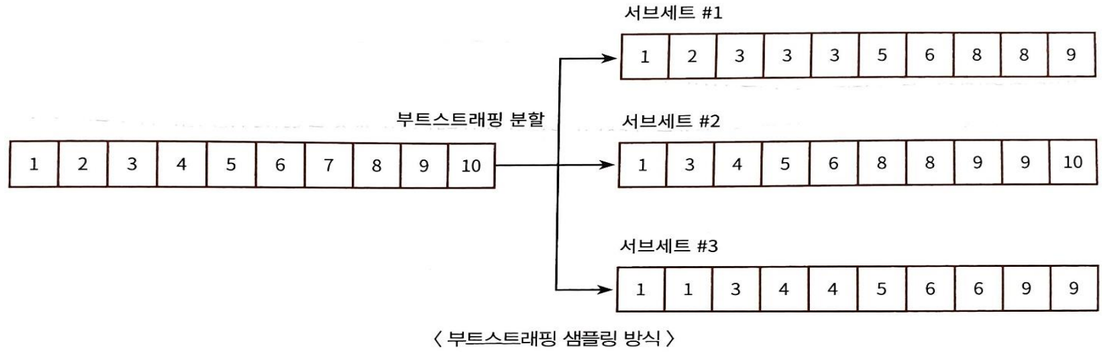
EWHA EURON

4.1 개요 및 실습

- 여러 개의 결정 트리 분류기가 전체 데이터에서 배깅 방식으로 각자의 데이터를 샘플링해 개별적으로 학습을 수행한 뒤 최종적으로 모든 분류기가 보팅을 통해 예측 결정.

- 개별적인 분류기의 기반 알고리즘은 결정 트리지만, 개별 트리가 학습하는 데이터 세트는 전체 데이터에서 일부가 중첩되게 샘플링되는 데이터 세트(부스트스트래핑 분할 방식)





〈 부트스트래핑 샘플링 방식 〉

4.1 개요 및 실습

[사용자 행동 인식 데이터 세트]
- RandomForestClassifier를 이용해 예측

```
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
```

```
import pandas as pd

def get_human_dataset( ):

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

    # 중복된 피처명을 수정하는 get_new_feature_name_df()를 이용, 신규 피처명 DataFrame생성.
    new_feature_name_df = get_new_feature_name_df(feature_name_df)

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

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

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

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


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

```
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, max_depth=8)
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 : 0.9196
```

4.2 하이퍼 파라미터 및 튜닝

앙상블 알고리즘의 단점 :
1. 하이퍼 파라미터가 너무 많고, 튜닝을 위한 시간이 많이 소모 됨.
2. 튜닝 후 예측 성능이 크게 향상되는 경우가 많지 않음.

[랜덤 포레스트] - 하이퍼 파라미터

n_estimaotrs	-결정 트리의 개수를 지정함. (디폴트는 10) -많이 설정할수록 좋은 성능을 기대할 수 있지만, 계속 증가시킨다고 성능이 무조건 향상되는 건 아님. - 늘릴수록 학습 수행 시간이 오래 걸림.
max_features	- 결정 트리의 max_features 파라미터와 같음. 하지만 Randomforestclassifer의 기본 max_features는 'None'이 아니라 'sqrt'와 같음. - 랜덤포레스트의 트리를 분할하는 피처를 참조할 때, 전체 피처가 아니라 sqrt만큼 참조함. ex) 전체 피처가 16이면, 분할을 위해선 4개 참조.
max_depth, min_samples_leat, min_samples_split	- 결정 트리에서 과적합을 개선하기 위해 사용되는 파라미터가 랜덤 포레스트에도 똑같이 적용됨.

4.2 하이퍼 파라미터 및 튜닝

GridSearchCV 이용

```
from sklearn.model_selection import GridSearchCV

params = {
    'n_estimators':[100],
    'max_depth' : [6, 8, 10, 12],
    'min_samples_leaf' : [8, 12, 18 ],
    'min_samples_split' : [8, 16, 20]
}
# RandomForestClassifier 객체 생성 후 GridSearchCV 수행
rf_clf = RandomForestClassifier(random_state=0, n_jobs=-1)
grid_cv = GridSearchCV(rf_clf , param_grid=params , cv=2, n_jobs=-1 )
grid_cv.fit(X_train , y_train)

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

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

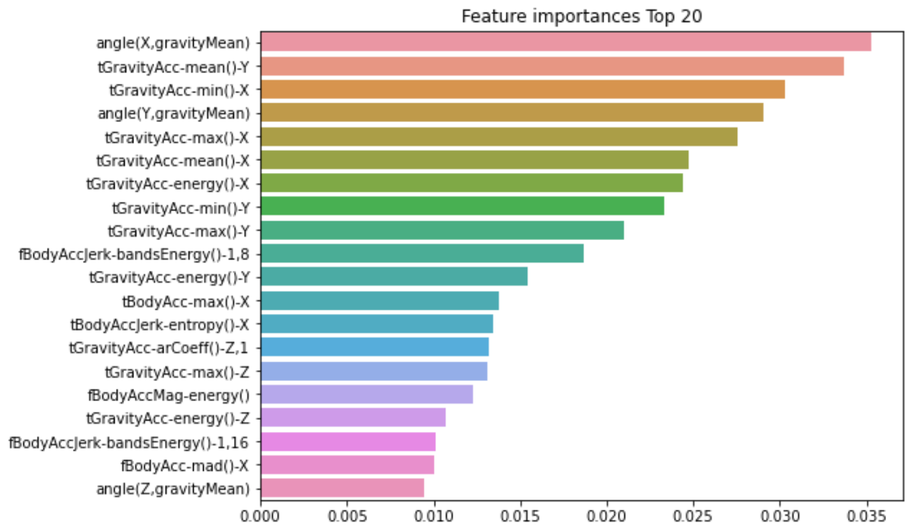
RandomForestCalssifier 학습

```
rf_clf1 = RandomForestClassifier(n_estimators=100, min_samples_leaf=6,max_depth=16,
                                 min_samples_split=2, 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: 0.9260

피처 중요도

```
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()
```



4.3 랜덤 포레스트 Vs 배깅

배깅	랜덤 포레스트
- Boostrap Aggregation - Training set의 개수를 늘린 다음, 각 training set을 가지고 학습을 시키고 각 training set으로부터 나온 예측 결과를 regression의 경우 평균 내어 전체 예측 결과로 사용하고 classification의 경우 다수결의 원칙 사용(복원추출 사용)	- 배깅 방법 사용하고 각각의 bootstrapped training set을 훈련시키는 base learner가 decision tree인 경우 - 배깅과 다른 점 : 배깅의 base learner가 decision tree인 경우, 배깅은 설명 변수에 있는 전체 feature를 모두 사용하여 decision tree의 노드를 분할하는 반면, 랜덤 포레스트는 자동적으로, 설명 변수에 있는 feature 가운데 일부만을 무작위로 골라내어 그 feature들만 사용하여 노드를 분할함. -> 설명 변수의 영향력을 낮출 수 있어, overfitting의 위험을 줄일 수 있음.

출처 : 배깅(Bagging) / 랜덤 포레스트(Ran.. : 네이버블로그 (naver.com)

4.4 랜덤 포레스트 회귀

```
estimator = RandomForestRegressor(n_estimators=10, bootstrap=True, criterion='mse', max_depth=None,
                                  max_leaf_nodes=None, min_samples_split=2, min_samples_leaf=1, max_features='auto')
```

주요 하이퍼 파라미터

n_estimators	렌덤포레스트를 구성하는 결정나무의 개수. 디폴트는 100
criterion	결정 나무의 노드를 분지할 때 사용하는 불순도 측정 방식. mse와 rmse mae 중 하나로 입력.
max_depth	결정나무의 최대 깊이. None으로 입력하며 잎 노드가 완전 순수해지거나 모든 잎 노드에 min_sampoles_split_samples 보다 적은 수의 샘플을 포함할 때까지 결정나무를 학습시킴.
max_features	결정나무를 분지할 때 고려하는 특징 수 혹은 비율. 디폴트는 sqrt로 특정 개수에 루트를 씌운 값임.
max_samples	각 결정 나무를 학습하는 데 사용할 샘플의 수 혹은 비율.

$$MSE = \frac{1}{N}\sum_{i}^{N}(pred_i - target_i)^2$$

$$RMSE = \sqrt{\frac{1}{N}\sum_{i}^{N}(pred_i - target_i)^2}$$

$$MAE = \frac{1}{N}\sum_{i}^{N}|(pred_i - target_i)|$$

4.4 랜덤 포레스트 회귀

데이터 로드

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor

x_data = np.array([
    [2, 1],
    [3, 2],
    [3, 4],
    [5, 5],
    [7, 5],
    [2, 5],
    [8, 9],
    [9, 10],
    [6, 12],
    [9, 2],
    [6, 10],
    [2, 4]
])
y_data = np.array([3, 5, 7, 10, 12, 7, 13, 13, 12, 13, 12, 6])
```

데이터 전처리 및 모델 생성, 학습

```
x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.3, random_state=777)

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

```
RandomForestRegressor()
```

모델 검증 및 예측

```
print(model.score(x_train, y_train))

print(model.score(x_test, y_test))

x_test = np.array([
    [4, 6]
])

y_predict = model.predict(x_test)

print(y_predict[0])
```

```
0.9810486008836524
0.8006804347826088
6.65
```

출처
랜덤 포레스트(회귀)의 하이퍼 파라미터 튜닝 (tistory.com)
언제 MSE, MAE, RMSE를 사용하는가?. 제목에 열거한 RMSE, MSE, MAE는 딥러닝 모델을 최적화 하는… | by Jaeyoung Cheong | Medium
강의 02 RandomForestRegressor 모델 - 토닥토닥 sklearn - 머신러닝 (wikidocs.net)

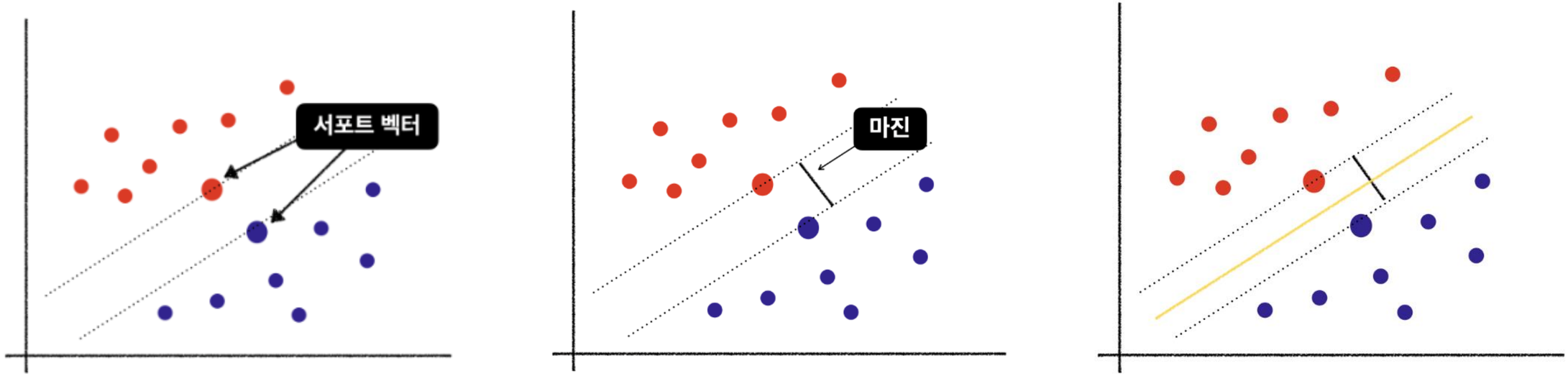
05. 서포트 벡터 머신

EWHA EURON

#5.1 Linear SVM Classification

1. 서포트 벡터 머신(Support Vector Machines, SVM)

- 여러 집단들을 가장 잘 분류할 수 있는 최적의 선, 결정 경계를 찾기 위한 모델
- 서포트 벡터 : 영역의 경계 부분 데이터를 기준으로 한 평행한 두 직선
- 마진 : 두 집단 사이의 거리 -> 최적의 결정경계는 마진을 최대화



#5.1 Linear SVM Classification

1. 서포트 벡터 머신(Support Vector Machines, SVM)

- 스케일링

: 스케일링에 따라 데이터의 위치가 달라지고, 그에 따라 결정 경계도 달라지므로 적절한 스케일링이 필요

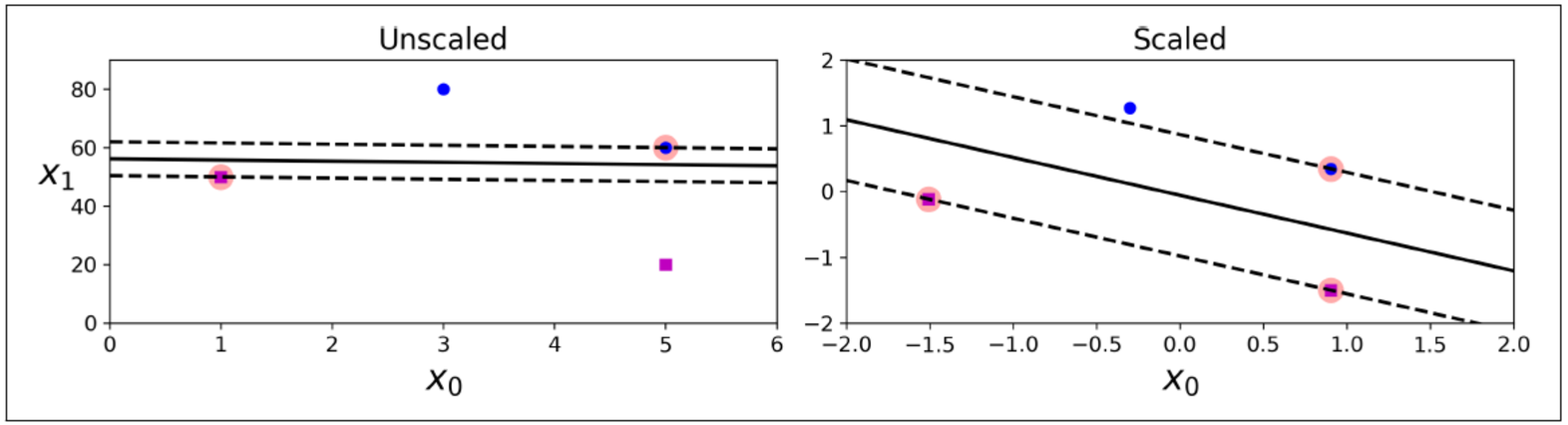


Figure 5-2. Sensitivity to feature scales

#5.1 Linear SVM Classification

2. 하드 마진 SVM

- 이상치를 허용하지 않고 가능한 멀리 경계를 형성하는 모델

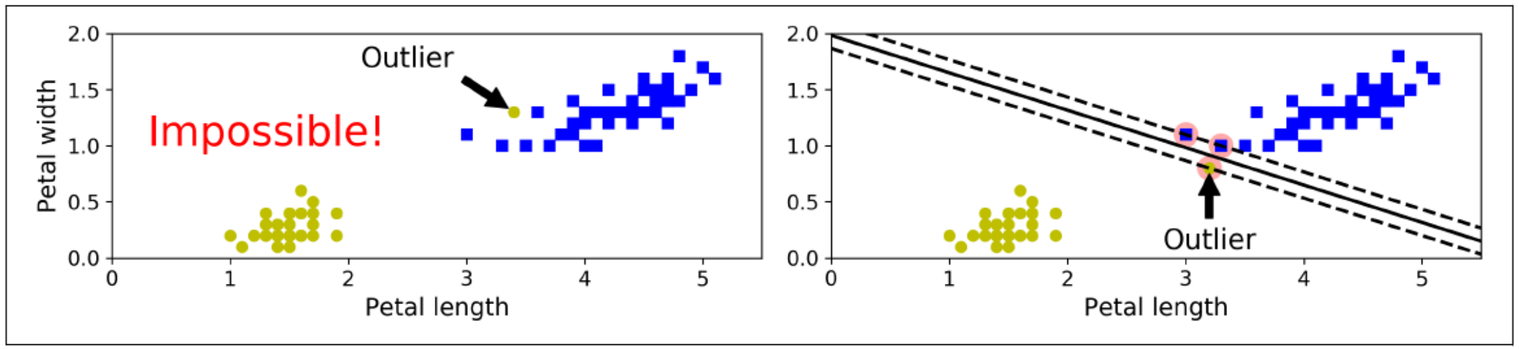


Figure 5-3. Hard margin sensitivity to outliers

- 극단적인 이상치가 있으면 분류 불가능
- 다른 클래스와 근접한 이상치가 있으면 마진이 매우 좁아짐 -> 분류 일반화 어려움

#5.1 Linear SVM Classification

3. 소프트 마진 SVM

- 하드 마진보다 조금 더 유연한 모델
- 이상치들을 어느정도 허용하면서 결정 경계 설정

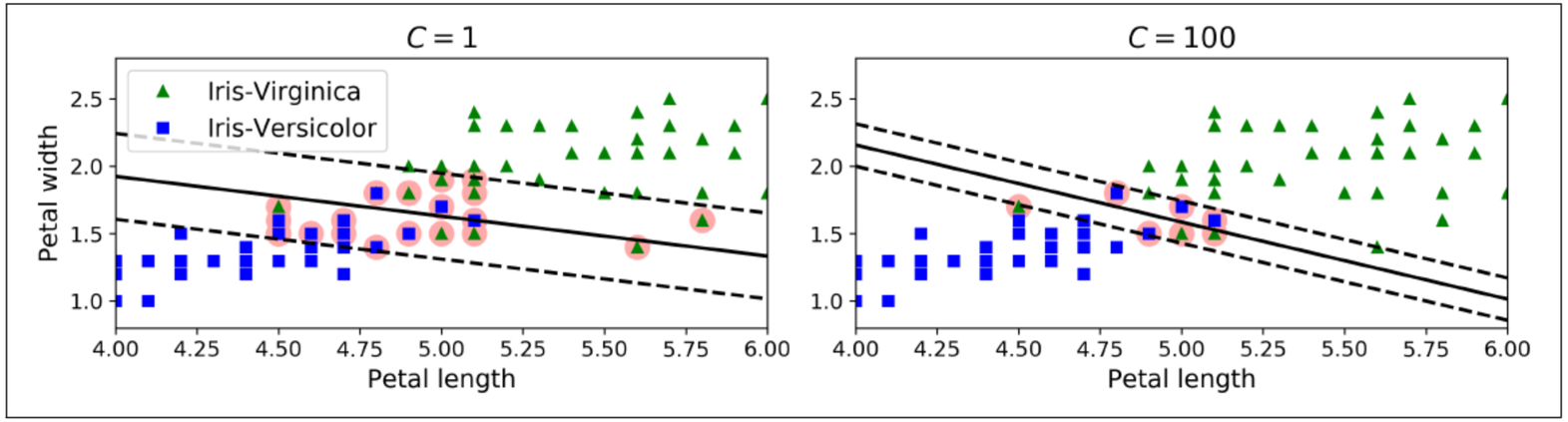


Figure 5-4. Large margin (left) versus fewer margin violations (right)

- 마진 오류(margin violation) : 데이터가 서포트 벡터 사이 혹은 반대편에 있는 경우
- 마진을 가능한 한 넓게 유지 & 마진 오류 사이 적절한 균형을 조절하는 것이 필요

#5.1 Linear SVM Classification

4. 사이킷런 SVC 클래스

```
from sklearn.svm import SVC
```

- 주요 파라미터

(1) c : 마진 오류를 얼마나 허용할 것인지 설정

- 값이 클수록 마진이 넓어지고 마진 오류 증가
- 값이 작을수록 마진이 좁아지고 마진 오류 감소

(2) kernel : 커널 함수 종류 지정

- 'linear', 'poly', 'rbf', 'sigmoid'

(3) gamma : 커널 계수 지정

- 'poly', 'rbf', 'sigmoid' 일 때 유효

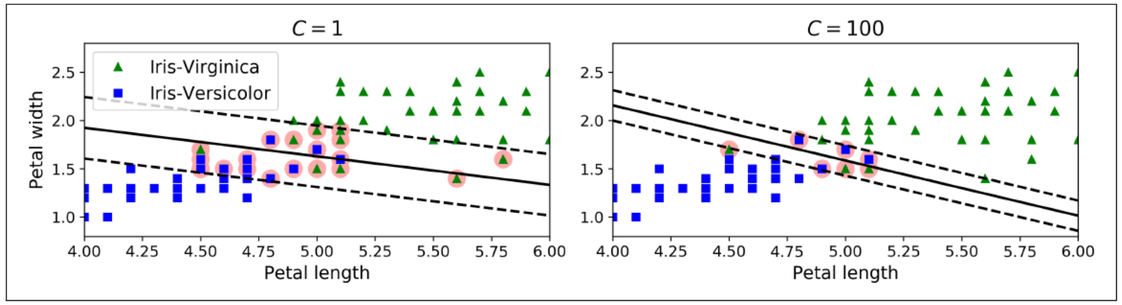


Figure 5-4. Large margin (left) versus fewer margin violations (right)

#5.2 Nonlinear SVM Classification

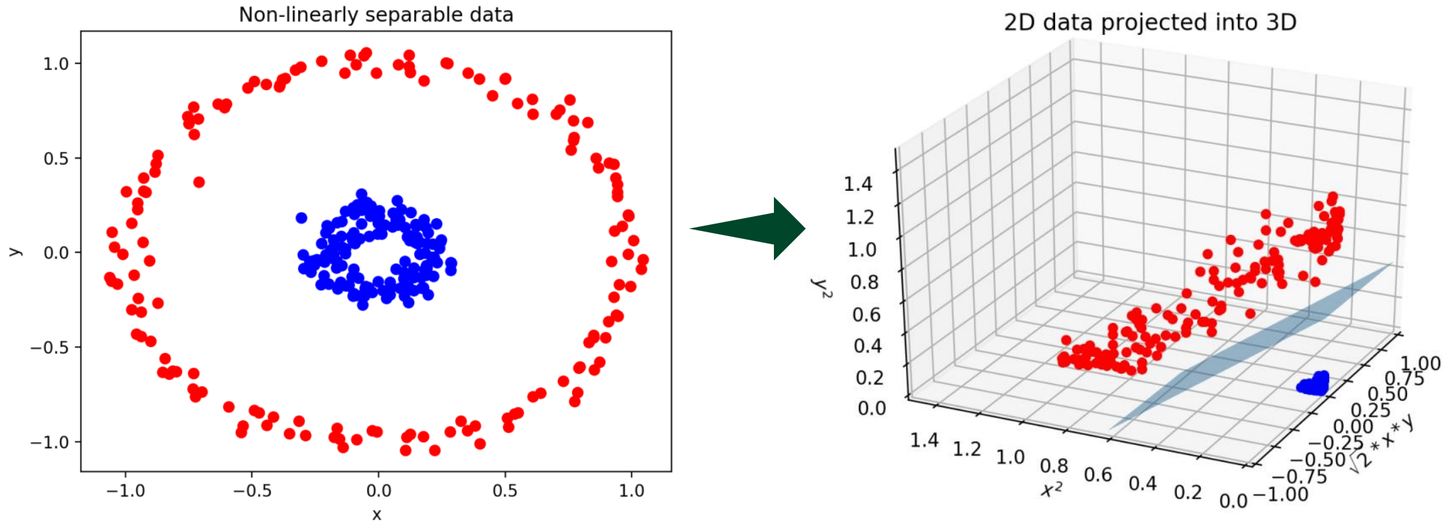
1. Kernel 함수

- 저차원에서 해결하기 어려운 문제를 고차원으로 변환시켜 문제를 해결할 때 사용하는 함수
- 커널 트릭 : 실제로 특성을 추가하지 않고도 특성을 추가한 것과 같은 효과를 내는 방법

(1) ‘linear’ : 선형 SVM

(2) ‘poly’ : 다항식 커널

- 2차원을 3차원으로 변환



(3) ‘rbf’ : 방사 기저 함수(가우시안 커널) -> 무한한 차원으로 변환

(4) ‘sigmoid’ : 시그모이드 함수

#5.2 Nonlinear SVM Classification

1. Kernel 함수

```
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)
```

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

rbf_kernel_svm_clf.fit(X,y)
```

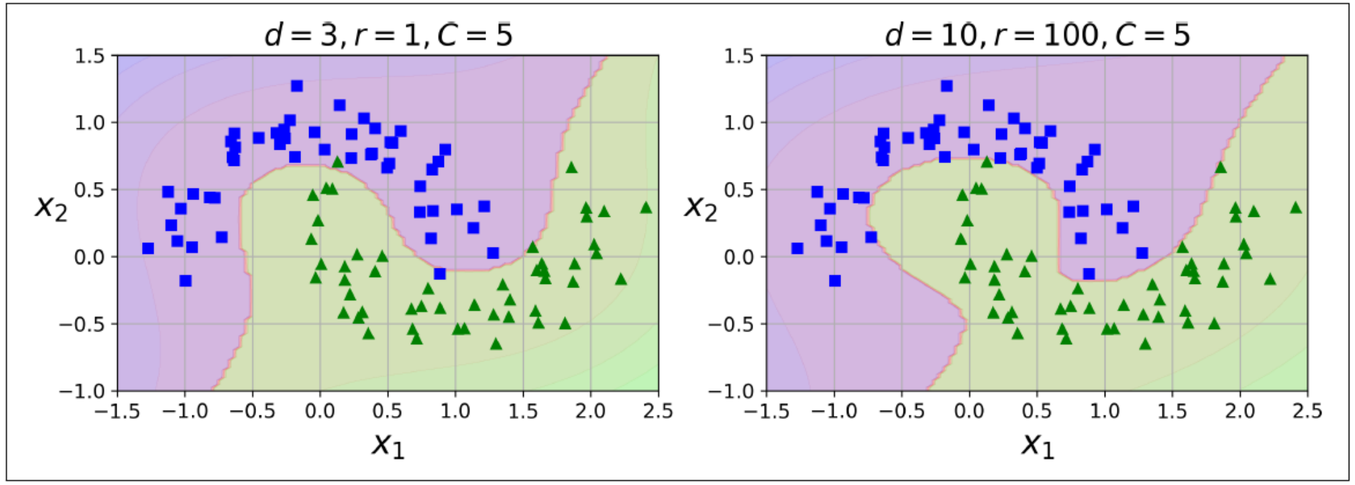


Figure 5-7. SVM classifiers with a polynomial kernel

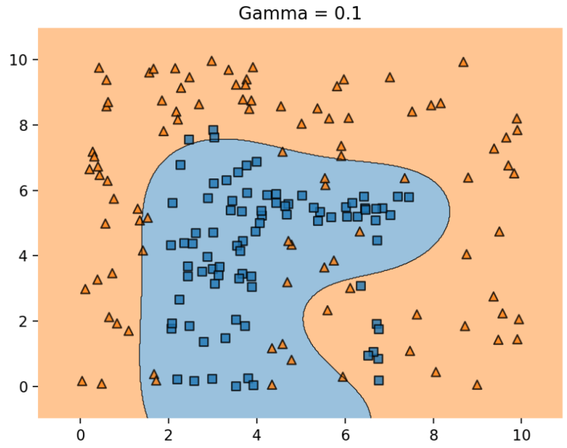
#5.2 Nonlinear SVM Classification

2. 파라미터 gamma

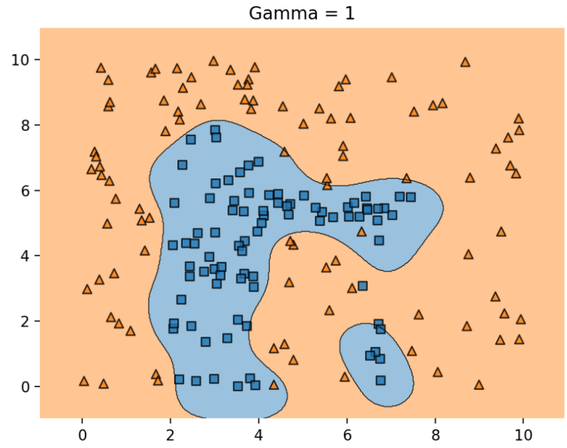
- 결정 경계를 얼마나 유연하게 그을 것인지 조절

  - 값을 높이면 학습 데이터에 많이 의존 -> 결정 경계 구불구불

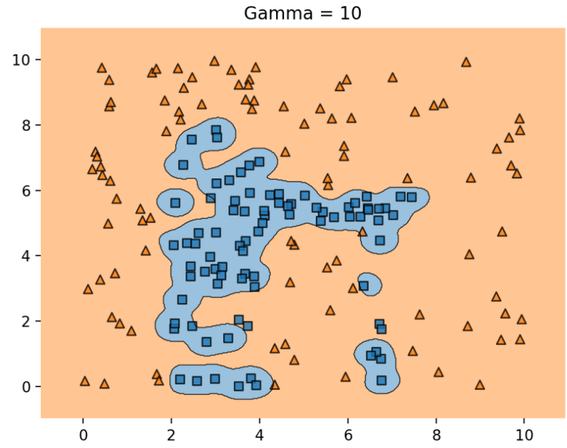
  - 값을 낮추면 학습 데이터에 덜 의존 -> 직선에 가까운 결정 경계



gamma 값이 너무 낮은 경우
-> 언더피팅 발생



gamma 값이 적당한 경우



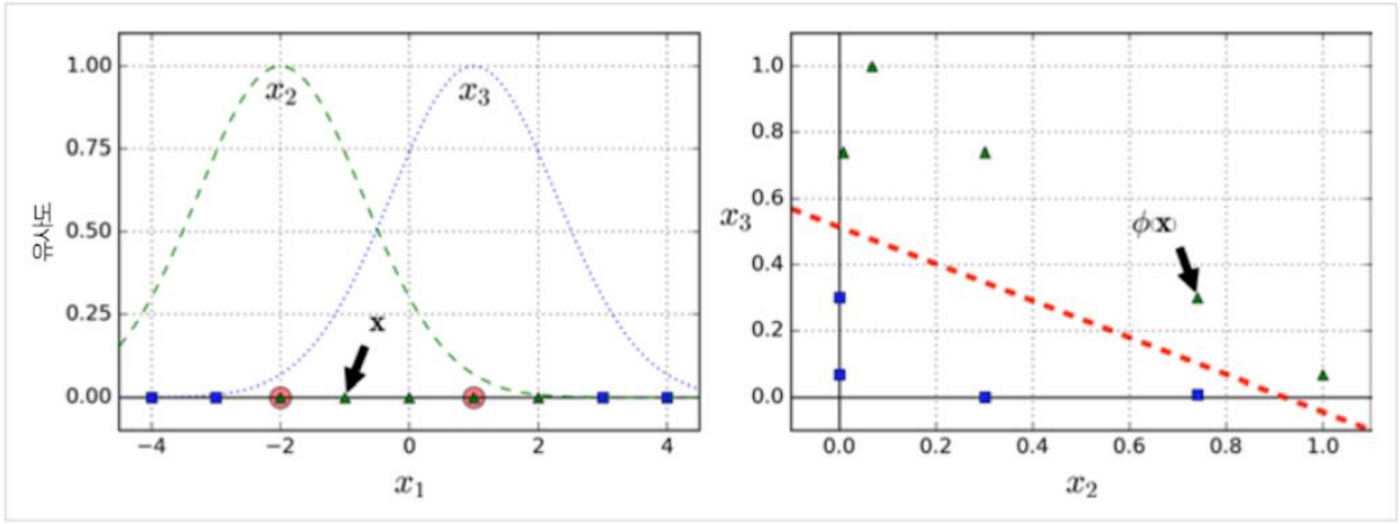
gamma 값이 너무 높은 경우
-> 오버피팅 발생

참고 링크 : https://hleecaster.com/ml-svm-concept/

#5.2 Nonlinear SVM Classification

3. 유사도 특성

- 각 샘플이 각 특정 랜드마크와 얼마나 닮았는지 측정하는 유사도 함수로 계산한 특성



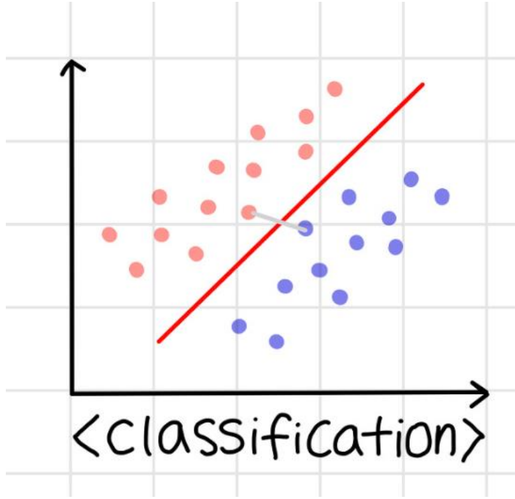
랜드마크 : X1 = -2, x 1 = 1
RBF 함수를 유사도 함수로 특성을 계산하여
변환된 데이터셋

- 랜드마크 선택 : 간단히 데이터셋에 있는 모든 표본 위치에 랜드마크 설정
  - 차원이 매우 커지기 때문에 선형적으로 구분될 가능성이 높아짐
  - 학습 데이터 세트가 매우 큰 경우 동일한 크기의 아주 많은 특성이 만들어진다는 단점

#5.3 SVM Regression

SVM 분류 vs 회귀 비교

- 분류
  - 데이터 간 거리가 가장 멀어지게 마진 설정
  - 마진이 클수록 분류 일반화 좋음 -> 소프트 마진 사용



- 회귀
  - 데이터를 대표하는 직선을 만드는 것이 목표
  - 마진이 좁을수록 데이터를 아우를 수 있는 회귀선 -> 하드 마진 사용
  - 사이킷런 SVR 클래스 사용 `from sklearn.svm import SVR`

#5.4 Under the Hood

1. 결정함수와 목적함수

(1) 결정 함수 : $w^T x + b = w_1 x_1 + \cdots + w_n x_n + b$

- w : 피처 가중치 벡터 / b : 편향

- 함수 값이 positive -> 예측된 클래스를 positive(1)로 분류 / 그렇지 않다면 negative(0)로 분류

- SVM 학습 : 마진 오류를 피하면서 마진을 가능한 한 넓게 만드는 w와 b의 값을 찾는 것

(2) 목적 함수

- 마진을 가장 크게 하는 결정 경계의 방향(w)을 찾는 함수

- margin = 2 / $\| w \|$

#5.5 서포트 벡터 머신 실습

```
# 라이브러리 설치
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
```

```
# iris 데이터 로드 후 split
iris_data = load_iris()
X_train,X_test,y_train,y_test = train_test_split(iris_data.data,iris_data.target,test_size=0.2,random_state=11)
```

```
# 선형 SVM 분류 모델
c = [0.5, 1, 5, 10, 100]
for thisC in c:
  lin_svm = SVC(kernel='linear',C=thisC)
  lin_svm.fit(X_train, y_train)
  pred = lin_svm.predict(X_test)
  accuracy = accuracy_score(y_test, pred)
  print("C : {0}, 예측 정확도 : {1:.4f}".format(thisC, accuracy))
```

```
C : 0.5, 예측 정확도 : 0.9667
C : 1, 예측 정확도 : 1.0000
C : 5, 예측 정확도 : 1.0000
C : 10, 예측 정확도 : 1.0000
C : 100, 예측 정확도 : 0.9667
```

Linear SVM 모델 학습

- Kernel : ‘linear’
- C : for 반복문을 이용하여 예측 정확도 비교

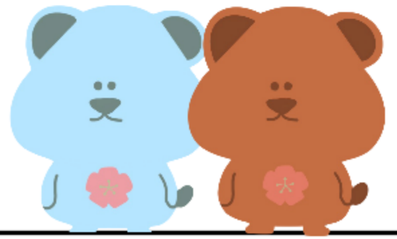
```
from sklearn.model_selection import GridSearchCV
param={'C':[1,5,10,20,40,100],
       'gamma':[.1, .25, .5, 1]}
GS=GridSearchCV(SVC(kernel='rbf'),param, cv=5)
GS.fit(X_train, y_train)
print(GS.best_params_)
print(GS.best_score_)

estimator = GS.best_estimator_
pred = estimator.predict(X_test)
print('테스트 데이터셋 정확도 : {0:.4f}'.format(accuracy_score(y_test, pred)))
```

```
{'C': 5, 'gamma': 0.1}
0.975
테스트 데이터셋 정확도 : 1.0000
```

비선형 RBF SVM 모델 학습

- Kernel : ‘rbf’
- C, gamma : GridSearchCV를 이용하여 최적의 파라미터 결정

THANK YOU



EWHA EURON