



Ch.2 사이킷런으로 시작하는 머신러닝

Ch.3 평가

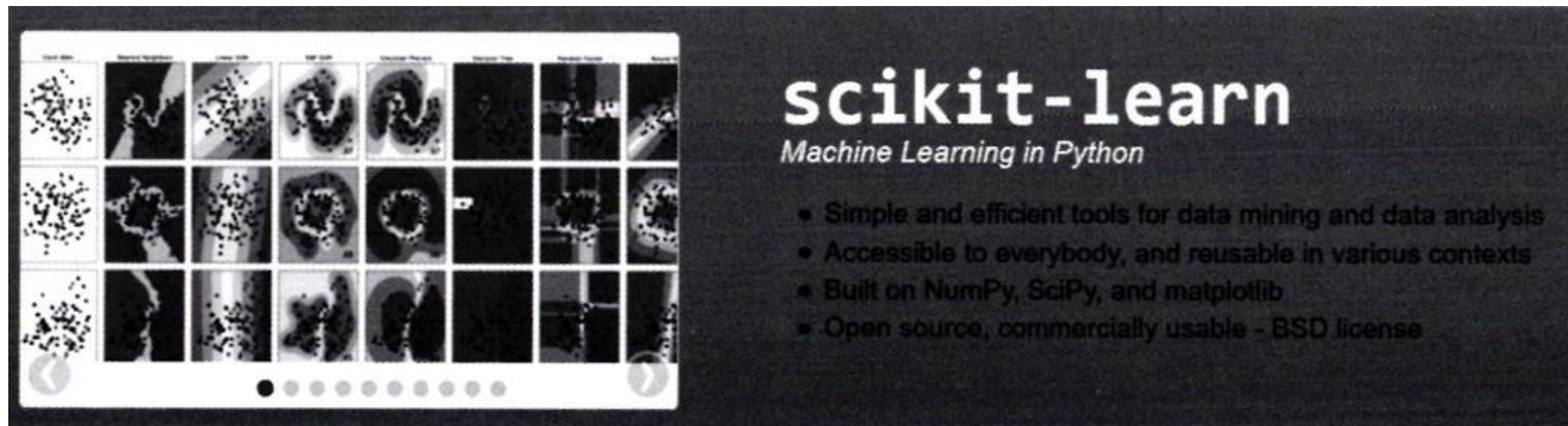
2팀 한송희 곽준희 우정연

Ch02. 사이킷런으로 시작하는 머신러닝



#2.1 사이킷런 소개와 특징

사이킷런(scikit-learn): 대표적인 파이썬 머신러닝 라이브러리



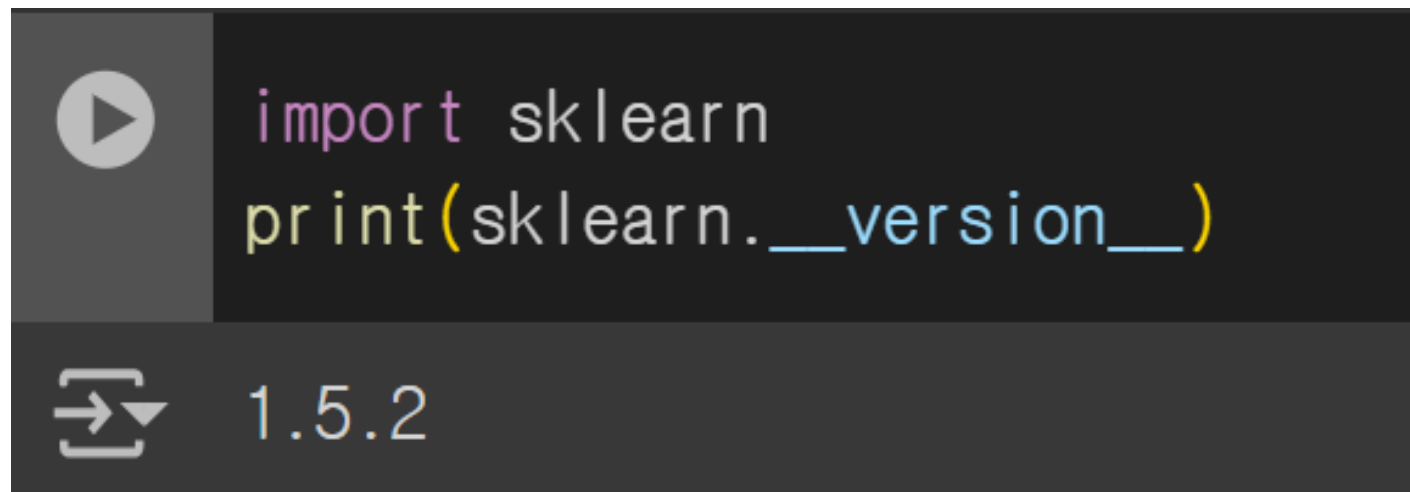
<특징>

- 가장 쉽고 파이썬스러운 API제공
- 머신러닝을 위한 다양한 알고리즘&개발에 편리한 프레임워크와 API제공
- 오랜 기간 많이 사용되어 검증 완료

#2.1 사이킷런 소개와 특징

설치방법

*anaconda를 설치하면 기본적으로 사이킷런이 설치되므로 별도 설치 필요 없음
(필요시 `conda install scikit-learn`, `pip install scikit-learn`으로 설치)



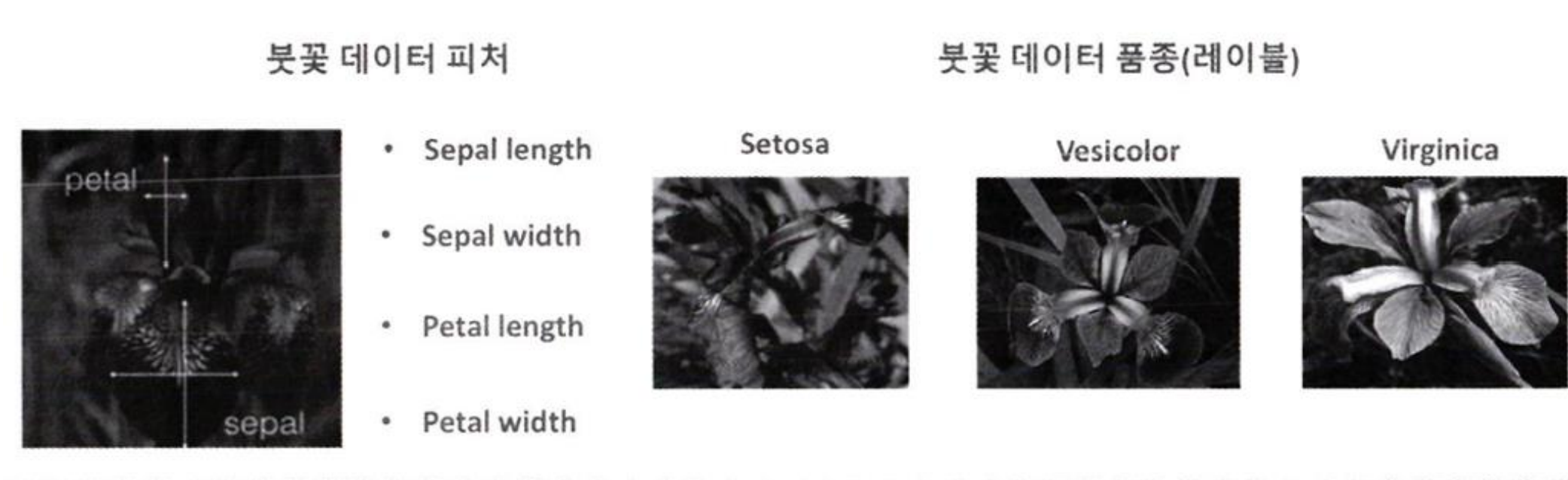
```
import sklearn
print(sklearn.__version__)
```

1.5.2

└ import로 사이킷런을 불러오고 버전을 출력

#2.2 첫번째 머신러닝 만들어 보기-붓꽃 품종 예측하기

붓꽃 품종 예측: 붓꽃 데이터 세트 속 Feature(꽃잎 길이, 너비, 꽃받침 길이, 너비)를 기반으로 붓꽃의 품종을 분류(Classification)



*Classification: 대표적인 supervised learning의 방법.

*Supervised learning: Label데이터로 모델 학습 후 별도의 test set으로 label을 예측해 모델 성능을 평가함

#2.2 첫번째 머신러닝 만들어 보기-붓꽃 품종 예측하기

Sklearn.datasets: 사이킷런에서 제공하는 데이터 셋 생성 모듈 모임

Sklearn.tree: 트리 기반 ML알고리즘을 구현한 클래스 모임

Sklearn.model_selection: 데이터를 학습,검증,예측으로 분리하거나 최적의 하이퍼 파라미터로 평가하기위한 모듈 모임

*하이퍼 파라미터: ML알고리즘 별로 최적 학습을 위해 직접 입력하는 파라미터

```
[ ] from sklearn.datasets import load_iris
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.model_selection import train_test_split
```

↳ load_iris(): 붓꽃 데이터 셋, Decision TreeClassifier 채택, train_test_split()함수로 데이터 셋 분리

#2.2 첫번째 머신러닝 만들어 보기-붓꽃 품종 예측하기

```
import pandas as pd

#붓꽃 데이터 세트 로딩
iris=load_iris()

#iris_data는 Iris 데이터 세트에서 feature만으로 된 데이터를 numpy로 가짐
iris_data=iris.data

#iris.target은 붓꽃 데이터 세트에서 label 데이터를 numpy로 가짐
iris_label=iris.target
print('iris target값:', iris_label)
print('iris target명:', iris.target_names)

#붓꽃 데이터 세트를 자세히 보기 위해 DataFrame형태로 변환
iris_df=pd.DataFrame(data=iris_data,columns=iris.feature_names)
iris_df['label']=iris.target
iris_df.head(3)
```

```
iris.target명: ['setosa' 'versicolor' 'virginica']
```

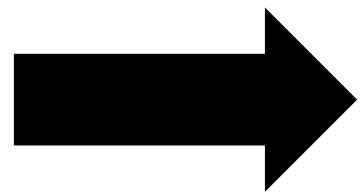
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	label
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0

#2.2 첫번째 머신러닝 만들어 보기-붓꽃 품종 예측하기

Train_test_split()을 이용해 학습 데이터와 테스트 데이터를 test_size 파라미터 입력 값 비율로 분할

```
X_train, X_test, y_train, y_test=train_test_split(iris_data,iris_label,test_size=0.2, random_state=11)
```

- └ iris_data: feature data set, iris_label: label data set
- └ test_size=0.2: 전체 데이터 중 20%가 테스트 데이터, 80%는 학습 데이터
- └ random_state: 호출 때마다 같은 학습/테스트 셋을 생성하기 위한 난수 발생값(아무 숫자나 상관 없음)



X_train: 학습용 feature data set
X_test: 테스트용 feature data set
y_train: 학습용 label data set
y_test: 테스트용 label data set

#2.2 첫번째 머신러닝 만들어 보기-붓꽃 품종 예측하기

```
#DecisionTreeClassifier 객체 생성
df_clf=DecisionTreeClassifier(random_state=11)

#학습 수행
df_clf.fit(X_train,y_train)

#학습 완료된 DecisionTreeClassifier 객체에서 test data set으로 예측 수행
pred=df_clf.predict(X_test)

#예측 성능 평가
from sklearn.metrics import accuracy_score
print('예측 정확도: {0:.4f}'.format(accuracy_score(y_test,pred)))
```

예측 정확도: 0.9333

#2.2 첫번째 머신러닝 만들어 보기-붓꽃 품종 예측하기

데이터 세트 분리: 학습 데이터와 테스트 데이터로 분리



모델 학습: 학습 데이터 기반으로 ML알고리즘 적용

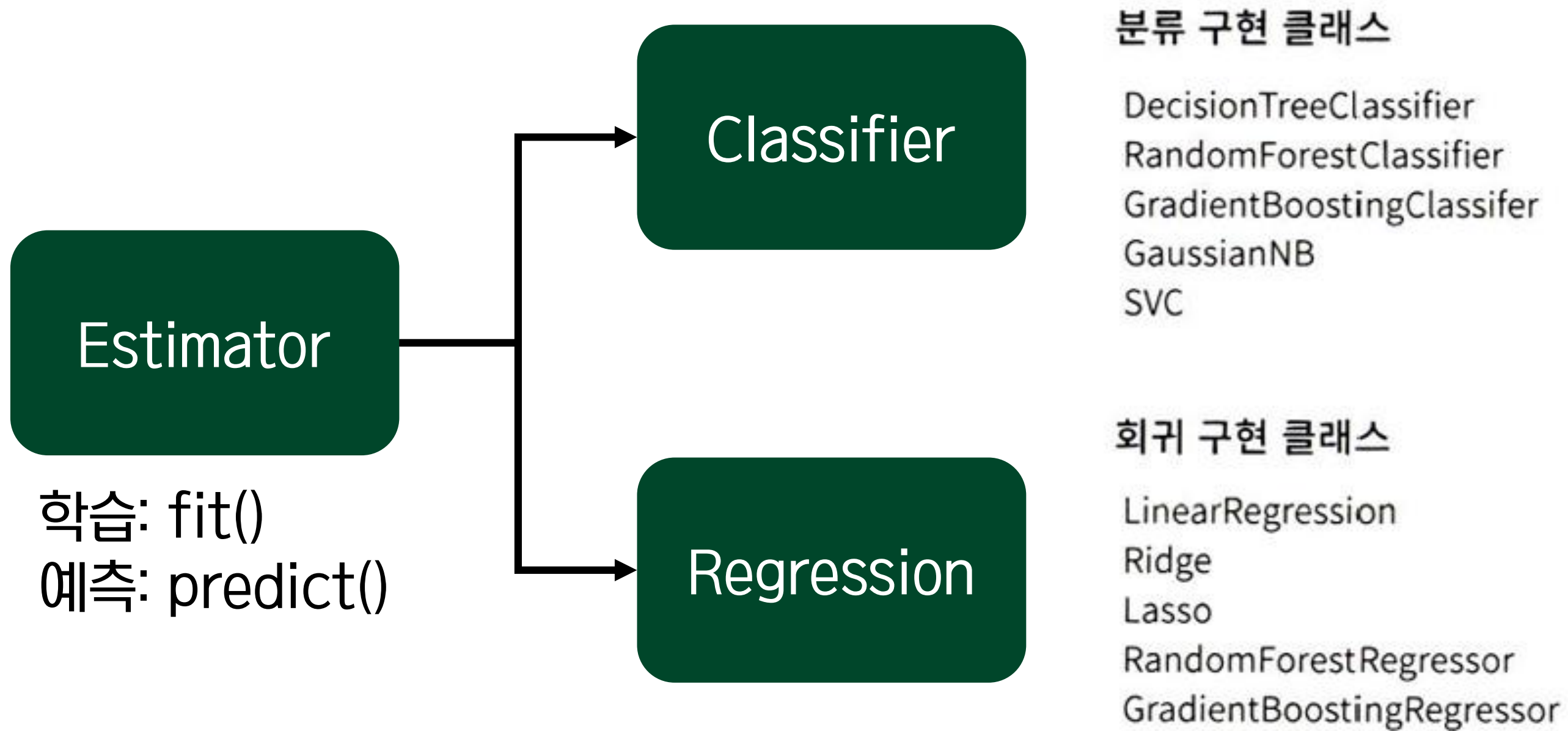


예측 수행: 학습된 ML모델로 테스트 데이터 분류



평가: 예측된 결과값과 실제 데이터 값을 비교

#2.3 사이킷런의 기반 프레임워크 익히기



#2.3 사이킷런의 기반 프레임워크 익히기

Evaluation 함수(e.g. `cross_val_score()`), 하이퍼 파라미터 튜닝 클래스(e.g. `GridSearchCV` ← `estimator`를 인자로 받음)

Unsupervised learning인 차원축소, 클러스터링, 피처 추출 ← `fit()`으로 입력 데이터 형태에 맞춰 데이터 변환 사전 구조 맞추기, `transform()`으로 실제 작업

#2.3 사이킷런의 기반 프레임워크 익히기

사이킷런의 주요 모듈

분류	모듈명	설명
예제 데이터	sklearn.datasets	예제 데이터셋
피처처리	sklearn.preprocessing	데이터 전처리
	sklearn.feature_selection	피처를 우선순위로 선택
	sklearn.featre_extraction	텍스트, 이미지 데이터의 벡터화된 피처 추출
피처 처리&차원 축소	sklearn.decomposition	차원축소
데이터 분리,검증&파라미터 튜닝	sklearn.model_selection	데이터 분리, 최적 파라미터 추출 API
평가	sklearnr.metrics	성능측정법 제공
ML 알고리즘	sklearn.ensemble	앙상블 알고리즘
	sklearn.linear_model	회귀 관련 알고리즘&SGD
	sklearn.naïve_bayes	나이브 베이즈 알고리즘
	sklearn.neighbors	최근접 이웃 알고리즘
	sklearn.svm	서포터 벡터 머신 알고리즘
	sklearn.tree	의사결정트리 알고리즘
	sklearn.cluster	비지도 클러스터링 알고리즘
유틸리티	sklearnr.pipeline	변환, 학습, 예측을 함께 묶음

#2.3 사이킷런의 기반 프레임워크 익히기

내장된 예제 데이터 셋 +) fetch 계열 명령어로 인터넷에서 내려받는 방법도 있음

분류나 회귀 연습용 예제 데이터

API 명	설명
<code>datasets.load_boston()</code>	회귀 용도이며, 미국 보스턴의 집 피쳐들과 가격에 대한 데이터 세트
<code>datasets.load_breast_cancer()</code>	분류 용도이며, 위스콘신 유방암 피쳐들과 악성/음성 레이블 데이터 세트
<code>datasets.load_diabetes()</code>	회귀 용도이며, 당뇨 데이터 세트
<code>datasets.load_digits()</code>	분류 용도이며, 0에서 9까지 숫자의 이미지 픽셀 데이터 세트
<code>datasets.load_iris()</code>	분류 용도이며, 붓꽃에 대한 피쳐를 가진 데이터 세트

표본 데이터 생성기

API 명	설명
<code>datasets.make_classifications()</code>	분류를 위한 데이터 세트를 만듭니다. 특히 높은 상관도, 불필요한 속성 등의 노이즈 효과를 위한 데이터를 무작위로 생성해 줍니다.
<code>datasets.make_blobs()</code>	클러스터링을 위한 데이터 세트를 무작위로 생성해 줍니다. 군집 지정 개수에 따라 여러 가지 클러스터링을 위한 데이터 세트를 쉽게 만들어 줍니다.

#2.3 사이킷런의 기반 프레임워크 익히기

딕셔너리 형 데이터셋의 키 값

Key	
Data	Feature의 데이터 셋
Target	분류→ label 값, 회귀→ 숫자 결과값 데이터 셋
Target_name	개별 label 이름
Feature_names	Feature 이름
DESCR	데이터 셋에 대한 설명과 각 feature 설명

```
from sklearn.datasets import load_breast_cancer
iris_data=load_iris()
print(type(iris_data))

<class 'sklearn.utils._bunch.Bunch'>

keys=iris_data.keys()
print('붓꽃 데이터 세트의 키들:',keys)

붓꽃 데이터 세트의 키들: dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names', 'filename', 'data_module'])
```


#2.3 사이킷런의 기반 프레임워크 익히기

Loda_iris()가 반환하는 붓꽃 데이터 셋 각 키가 의미하는 값

feature_names				target_names		
sepal length (cm) sepal width (cm) petal length (cm) petal width (cm)				setosa, versicolor, virginica (0 , 1 , 2)		
data	5.1	3.5	1.4	0.2	0	target
	4.9	3.0	1.4	0.2	1	
	
	4.6	3.1	1.5	0.2	2	
	5.0	3.6	1.4	0.2	0	

EWCHA
EUROPEAN

```
feature_names의 type: <class 'list'>
feature_names의 shape: 4
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']

target_names의 type: <class 'numpy.ndarray'>
target_names의 shape: 3
['setosa' 'versicolor' 'virginica']

data의 type: <class 'numpy.ndarray'>
data의 shape: (150, 4)
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
```

[illegible]

#2.4 Model Selection 모듈 소개

1) train_test_split(): 학습/테스트 데이터 셋 분리

(feature data set, label data set, test_size, train_size, shuffle, random_state)

Test_size: 테스트 데이터 세트 크기를 얼마나 샘플링할 것인가(default=0.25)

Train_size: 학습용 데이터 세트 크기를 얼마나 샘플링할 것인가(보통 사용 안함)

Shuffle: 데이터 분리 전 데이터를 미리 섞을 지 결정(default=True)

Random_state: 호출할 때마다 동일한 분리 셋 생성을 위해 주어지는 난수 값

#2.4 Model Selection 모듈 소개

```
[25] from sklearn.tree import DecisionTreeClassifier
      from sklearn.metrics import accuracy_score
      from sklearn.datasets import load_iris
      from sklearn.model_selection import train_test_split

      dt_clf=DecisionTreeClassifier()
      iris_data=load_iris()

      X_train, X_test, y_train, y_test=train_test_split(iris_data.data,iris_data.target,test_size=0.3, random_state=121)
```

```
▶ dt_clf.fit(X_train, y_train)
  pred=dt_clf.predict(X_test)
  print('예측 정확도: {0:.4f}'.format(accuracy_score(y_test,pred)))
```

```
↗ 예측 정확도: 0.9556
```

#2.4 Model Selection 모듈 소개

2) 교차 검증

:별도의 여러 세트로 구성된 학습 데이터 세트와 검증 데이터 세트에서 학습과 평가를 수행함

*과적합(overfitting)문제 해결을 위해 교차 검증을 이용해 다양한 학습과 평가 수행



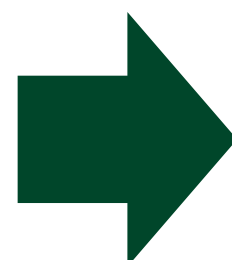
#2.4 Model Selection 모듈 소개

-k 폴드 교차 검증: k개의 데이터 폴드 세트를 만들어 k번 각 폴드 세트에 학습,검증 평가 반복

5-fold CV

DATASET

Estimation 1	Test	Train	Train	Train	Train
Estimation 2	Train	Test	Train	Train	Train
Estimation 3	Train	Train	Test	Train	Train
Estimation 4	Train	Train	Train	Test	Train
Estimation 5	Train	Train	Train	Train	Test



교차 검증 최종
평가=평균

e.g. k=5

#2.4 Model Selection 모듈 소개

```
n_iter=0

#KFold 객체의 split()를 호출하면 폴드 별 학습용, 검증용 테스트의 로우 인덱스를 array로 반환
for train_index, test_index in kfold.split(features):
    #kfold.split()으로 반환된 인덱스를 이용해 학습용, 검증용 테스트 데이터 추출
    X_train, X_test=features[train_index], features[test_index]
    y_train, y_test=label[train_index], label[test_index]
    #학습 및 예측
    dt_clf.fit(X_train, y_train)
    pred=dt_clf.predict(X_test)
    n_iter+=1
    #반복 시마다 정확도 측정
    accuracy=np.round(accuracy_score(y_test, pred), 4)
    train_size=X_train.shape[0]
    test_size=X_test.shape[0]
    print("\n#{0}교차 검증 정확도:{1}, 학습 데이터 크기:{2}, 검증 데이터 크기:{3}".format(n_iter, accuracy, train_size, test_size))
    print("#{0}검증 세일 인덱스:{1}".format(n_iter, test_index))
    cv_accuracy.append(accuracy)

#개별 iteration별 정확도를 합하여 평균 정확도 계산
print("\n## 평균 검증 정확도:", np.mean(cv_accuracy))
```

```
#1교차 검증 정확도:1.0, 학습 데이터 크기:120, 검증 데이터 크기:30
#1검증 세일 인덱스:[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29]

#2교차 검증 정확도:0.9667, 학습 데이터 크기:120, 검증 데이터 크기:30
#2검증 세일 인덱스:[30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53
54 55 56 57 58 59]

#3교차 검증 정확도:0.8667, 학습 데이터 크기:120, 검증 데이터 크기:30
#3검증 세일 인덱스:[60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
84 85 86 87 88 89]

#4교차 검증 정확도:0.9333, 학습 데이터 크기:120, 검증 데이터 크기:30
#4검증 세일 인덱스:[ 90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107
108 109 110 111 112 113 114 115 116 117 118 119]

#5교차 검증 정확도:0.7333, 학습 데이터 크기:120, 검증 데이터 크기:30
#5검증 세일 인덱스:[120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137
138 139 140 141 142 143 144 145 146 147 148 149]

## 평균 검증 정확도: 0.9
```


#2.4 Model Selection 모듈 소개

-stratified K 폴드: 불균형한 분포도를 가진 레이블 데이터 집합을 위한 K 폴드

원본 데이터의 레이블 분포를 먼저 고려한 후 이 분포와 동일하게 데이터 셋을 분배

```
kfold=KFold(n_splits=3)
n_iter=0
for train_index, test_index in kfold.split(iris_df):
    n_iter+=1
    label_train=iris_df['label'].iloc[train_index]
    label_test=iris_df['label'].iloc[test_index]
    print('## 교차 검증: {0}'.format(n_iter))
    print('학습 레이블 데이터 분포:\n', label_train.value_counts())
    print('검증 레이블 데이터 분포:\n', label_test.value_counts())
```

↳ 교차 검증마다 3개의 폴드 세트로 만들어지는 학습/검증 레이블이 완전히 다른 값으로 추출 → 검증 예측 정확도=0

```
## 교차 검증: 1
학습 레이블 데이터 분포:
label
1    50
2    50
Name: count, dtype: int64
검증 레이블 데이터 분포:
label
0    50
Name: count, dtype: int64
## 교차 검증: 2
학습 레이블 데이터 분포:
label
0    50
2    50
Name: count, dtype: int64
검증 레이블 데이터 분포:
label
1    50
Name: count, dtype: int64
## 교차 검증: 3
학습 레이블 데이터 분포:
label
0    50
1    50
Name: count, dtype: int64
검증 레이블 데이터 분포:
label
2    50
Name: count, dtype: int64
```

#2.4 Model Selection 모듈 소개

```
from sklearn.model_selection import StratifiedKFold

skf=StratifiedKFold(n_splits=3)
n_iter=0

for train_index, test_index in skf.split(iris_df, iris_df['label']):
    n_iter+=1
    label_train=iris_df['label'].iloc[train_index]
    label_test=iris_df['label'].iloc[test_index]
    print('##교차 검증:{0}'.format(n_iter))
    print("학습 레이블 데이터 분포:\n",label_train.value_counts())
    print("검증 레이블 데이터 분포:\n", label_test.value_counts())
```

↳ 학습/검증 레이블 데이터 값의 분포도가 동일하게 할당

```
##교차 검증:1
학습 레이블 데이터 분포:
label
2    34
0    33
1    33
Name: count, dtype: int64
검증 레이블 데이터 분포:
label
0    17
1    17
2    16
Name: count, dtype: int64
##교차 검증:2
학습 레이블 데이터 분포:
label
1    34
0    33
2    33
Name: count, dtype: int64
검증 레이블 데이터 분포:
label
0    17
2    17
1    16
Name: count, dtype: int64
##교차 검증:3
학습 레이블 데이터 분포:
label
0    34
1    33
2    33
Name: count, dtype: int64
검증 레이블 데이터 분포:
label
1    17
2    17
0    16
Name: count, dtype: int64
```

#2.4 Model Selection 모듈 소개

```
dt_clf = DecisionTreeClassifier(random_state=156)
```

```
skfold = StratifiedKFold(n_splits=3)
```

```
n_iter = 0
```

```
cv_accuracy = []
```

```
# StratifiedKFold의 split() 호출 시 레이블 데이터 셋도 추가 입력 필요
```

```
for train_index, test_index in skfold.split(features, label):
```

```
    # split()으로 반환된 인덱스를 이용해 학습용, 검증용 테스트 데이터 추출
```

```
    X_train, X_test = features[train_index], features[test_index]
```

```
    y_train, y_test = label[train_index], label[test_index]
```

```
    # 학습 및 예측
```

```
    dt_clf.fit(X_train, y_train)
```

```
    pred = dt_clf.predict(X_test)
```

```
    # 반복 시마다 정확도 측정
```

```
    n_iter += 1
```

```
    accuracy = np.round(accuracy_score(y_test, pred), 4)
```

```
    train_size = X_train.shape[0]
```

```
    test_size = X_test.shape[0]
```

```
    print(f'\n#{n_iter} 교차 검증 정확도: {accuracy}, 학습 데이터 크기: {train_size}, 검증 데이터 크기: {test_size}')
```

```
    print(f'#{n_iter} 검증 셋 인덱스: {test_index}')
```

```
    cv_accuracy.append(accuracy)
```

```
# 교차 검증별 정확도 및 평균 정확도 계산
```

```
print(f'\n## 교차 검증별 정확도:', np.round(cv_accuracy, 4))
```

```
print('## 평균 검증 정확도:', np.mean(cv_accuracy))
```

```
#1 교차 검증 정확도: 0.98, 학습 데이터 크기: 100, 검증 데이터 크기: 50
```

```
#1 검증 셋 인덱스: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 50
```

```
51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 100 101
```

```
102 103 104 105 106 107 108 109 110 111 112 113 114 115]
```

```
#2 교차 검증 정확도: 0.94, 학습 데이터 크기: 100, 검증 데이터 크기: 50
```

```
#2 검증 셋 인덱스: [ 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 67
```

```
68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 116 117 118
```

```
119 120 121 122 123 124 125 126 127 128 129 130 131 132]
```

```
#3 교차 검증 정확도: 0.98, 학습 데이터 크기: 100, 검증 데이터 크기: 50
```

```
#3 검증 셋 인덱스: [ 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 83 84
```

```
85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 133 134 135
```

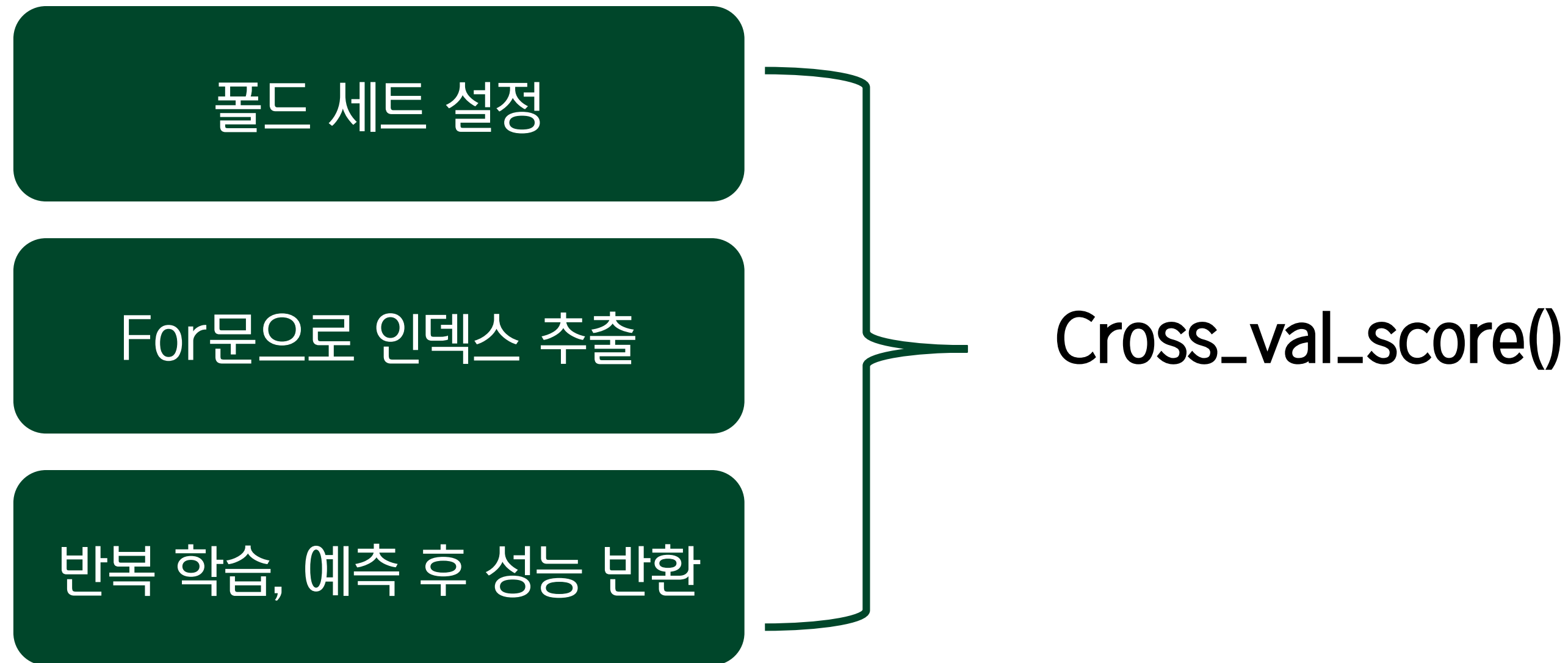
```
136 137 138 139 140 141 142 143 144 145 146 147 148 149]
```

```
## 교차 검증별 정확도: [0.98 0.94 0.98]
```

```
## 평균 검증 정확도: 0.9666666666666667
```

#2.4 Model Selection 모듈 소개

3) `cross_val_score()`: 교차 검증을 간편하게



#2.4 Model Selection 모듈 소개

`Cross_val_score(estimator, X, y=None, scoring=None, cv=None, n_jobs=1, verbose=0, fit_params=None, pre_dispatch='2*n_jobs')`

Estimator: 사이킷런의 Classifier 또는 Regressor

X: feature set

y: label data set

Scoring: 예측 성능 평가 지표

Cv: 교차 검증 폴드 수

#2.4 Model Selection 모듈 소개

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score, cross_validate
from sklearn.datasets import load_iris

iris_data = load_iris()
dt_clf = DecisionTreeClassifier(random_state=156)

data = iris_data.data
label = iris_data.target

scores = cross_val_score(dt_clf, data, label, scoring='accuracy', cv=3)
print('교차 검증별 정확도:', np.round(scores, 4))
print('평균 검증 정확도:', np.round(np.mean(scores), 4))
```

```
교차 검증별 정확도: [0.98 0.94 0.98]
평균 검증 정확도: 0.9667
```

#2.4 Model Selection 모듈 소개 –GridSearchCV

하이퍼 파라미터

머신러닝 알고리즘을 구성하는 주요 구성 요소로, 이 값을 조정해 알고리즘의 예측 성능 개선 가능.

GridSearchCV

- 교차 검증을 기반으로 하이퍼 파라미터의 최적 값을 편리하게 찾아줌.
- 수행시간이 상대적으로 오래 걸림.
- 사이킷런은 GridSearchCV API를 이용해 최적의 파라미터를 도출할 수 있는 방안 제공.

```
grid_parameters = {'max_depth': [1, 2, 3],  
                  'min_samples_split': [2, 3]}
```

순번	max_depth	min_samples_split
1	1	2
2	1	3
3	2	2
4	2	3
5	3	2
6	3	3

#2.4 Model Selection 모듈 소개 –GridSearchCV

GridSearchCV 클래스 생성자로 들어가는 주요 파라미터

- estimator: classifier, regressor, pipeline이 사용될 수 있음.
- Param_grid: key + 리스트 값을 가지는 딕셔너리가 주어짐. Estimator의 튜닝을 위해 파라미터명과 사용될 여러 파라미터 값을 지정.
- Scoring: 예측 성능을 측정할 평가 방법을 지정.
- cv: 교차 검증을 위해 분할되는 학습/테스트 세트의 개수를 지정.
- Refit: 디폴트가 True이며 True로 생성 시 가장 최적의 하이퍼 파라미터를 찾은 뒤 입력된 estimator 객체를 해당 하이퍼 파라미터로 재학습시킴.

#2.4 Model Selection 모듈 소개 –GridSearchCV

예제로 보는 GridSearchCV API 사용법

1)

```
[22] from sklearn.datasets import load_iris
      from sklearn.tree import DecisionTreeClassifier
      from sklearn.model_selection import GridSearchCV

      # 데이터를 로딩하고 학습 데이터와 테스트 데이터 분리
      iris = load_iris()
      X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
                                                          test_size=0.2, random_state=121)

      dtree = DecisionTreeClassifier()

      ### 파라미터를 딕셔너리 형태로 설정
      parameters = {'max_depth': [1, 2, 3], 'min_samples_split': [2, 3]}
```

3)

```
[24] print('GridSearchCV 최적 파라미터:', grid_dtree.best_params_)
      print('GridSearchCV 최고 정확도: {0:.4f}'.format(grid_dtree.best_score_))
```

⇒ GridSearchCV 최적 파라미터: {'max_depth': 3, 'min_samples_split': 2}
GridSearchCV 최고 정확도: 0.9750

```
[25] # GridSearchCV의 refit으로 이미 학습된 estimator 반환
      estimator = grid_dtree.best_estimator_

      # GridSearchCV의 best-estimator_는 이미 최적 학습이 됐으므로 별도 학습이 필요 없음
      pred = estimator.predict(X_test)
      print('테스트 데이터 세트 정확도: {0:.4f}'.format(accuracy_score(y_test, pred)))
```

⇒ 테스트 데이터 세트 정확도: 0.9667

2)

```
[23] import pandas as pd

      # param_grid의 하이퍼 파라미터를 3개의 train, test set fold로 나누어 테스트 수행 설정
      ### refit = True가 default임. True이면 가장 좋은 파라미터 설정으로 재학습시킴.
      grid_dtree = GridSearchCV(dtree, param_grid=parameters, cv=3, refit=True)

      # 붓꽃 학습 데이터로 param_grid의 하이퍼 파라미터를 순차적으로 학습/평가
      grid_dtree.fit(X_train, y_train)

      # GridSearchCV 결과를 추출해 DataFrame으로 변환
      scores_df = pd.DataFrame(grid_dtree.cv_results_)
      scores_df[['params', 'mean_test_score', 'rank_test_score',
                  'split0_test_score', 'split1_test_score', 'split2_test_score']]
```



	params	mean_test_score	rank_test_score	split0_test_score	split1_test_score	split2_test_score
0	{'max_depth': 1, 'min_samples_split': 2}	0.700000	5	0.700	0.7	0.70
1	{'max_depth': 1, 'min_samples_split': 3}	0.700000	5	0.700	0.7	0.70
2	{'max_depth': 2, 'min_samples_split': 2}	0.958333	3	0.925	1.0	0.95
3	{'max_depth': 2, 'min_samples_split': 3}	0.958333	3	0.925	1.0	0.95
4	{'max_depth': 3, 'min_samples_split': 2}	0.975000	1	0.975	1.0	0.95
5	{'max_depth': 3, 'min_samples_split': 3}	0.975000	1	0.975	1.0	0.95

* 주요 칼럼별 의미

- Params 칼럼: 수행할 때마다 적용된 개별 하이퍼 파라미터 값
- Rank_test_score: 하이퍼 파라미터별로 성능이 좋은 score 순위 (1순위의 파라미터가 최적의 하이퍼 파라미터)
- Mean_test_score: 개별 하이퍼 파라미터별로 CV의 폴딩 테스트 세트에 대해 총 수행한 평가 평균값

#2.5 데이터 전처리

데이터 전처리 기본 사항

- 결손값(NaN, Null)은 허용되지 X.
 - 피처 값 중 Null 값이 적은 경우: 피처의 평균값으로 대체
 - 피처 값 중 Null 값이 대부분인 경우: 해당 피처는 드롭
 - 피처 값 중 Null 값이 일정 수준 이상 되는 경우: 해당 피처의 중요도가 높고, 평균값 대체 시 예측 왜곡이 심할 수 있다면 업무 로직 등을 검토해 더 정밀한 대체 값으로 선정
- 사이킷런의 ML 알고리즘은 문자열 값을 입력 값으로 허용하지 X
 - 문자열 값(카테고리형 피처, 텍스트형 피처)은 인코딩 해서 숫자 형으로 변환.
 - 텍스트형 피처는 불필요한 피처라고 판단되면 삭제.

#2.5 데이터 전처리 – 데이터 인코딩

레이블 인코딩

- 카테고리 피처를 코드형 숫자 값으로 변환하는 것.

Ex) TV: 1, 냉장고: 2, 전자레인지:3 ※ '01', '02' (문자열 값) -> 1, 2(숫자형 값)

- 사이킷런의 레이블 인코딩은 LabelEncoder 클래스로 구현함.

```
[53] from sklearn.preprocessing import LabelEncoder

items = ['TV', '냉장고', '전자레인지', '컴퓨터', '선풍기', '선풍기', '믹서', '믹서']

# LabelEncoder를 객체로 생성한 후, fit()과 transform()으로 레이블 인코딩 수행
encoder = LabelEncoder()
encoder.fit(items)
labels = encoder.transform(items)
print('인코딩 변환값:', labels)
```

⇒ 인코딩 변환값: [0 1 4 5 3 3 2 2]

```
[54] print('인코딩 클래스:', encoder.classes_)
```

⇒ 인코딩 클래스: ['TV' '냉장고' '믹서' '선풍기' '전자레인지' '컴퓨터']

```
[55] print('디코딩 원본값:', encoder.inverse_transform([4, 5, 2, 0, 1, 1, 3, 3]))
```

⇒ 디코딩 원본값: ['전자레인지' '컴퓨터' '믹서' 'TV' '냉장고' '냉장고' '선풍기' '선풍기']

#2.5 데이터 전처리 – 데이터 인코딩

레이블 인코딩의 한계

- 레이블 인코딩이 일괄적인 숫자 값으로 변환되면서, 몇몇 ML 알고리즘에 이를 적용할 경우 예측 성능이 떨어지는 경우가 발생할 수 있음.
- 따라서 선형회귀와 같은 ML 알고리즘에는 적용할 수 X.

#2.5 데이터 전처리 – 데이터 인코딩

원-핫 인코딩(One-Hot Encoding)

- 피처 값의 유형에 따라 새로운 피처를 추가해 고유 값에 해당하는 칼럼에만 1을 표시하고 나머지 칼럼에는 0을 표시하는 방식
- 사이킷런에서 OneHotEncoder 클래스로 쉽게 변환 가능.

※ 단, LabelEncoder와 다르게 주의점이 있음.

1) OneHotEncoder로 변환하기 전 모든 문자열 값이 숫자형 값으로 변환되어야 함.

2) 입력 값으로 2차원 데이터가 필요함.

```
[57] from sklearn.preprocessing import OneHotEncoder
import numpy as np

items=['TV', '냉장고', '전자레인지', '컴퓨터', '선풍기', '선풍기', '믹서', '믹서']

# 먼저 숫자 값으로 변환을 위해 LabelEncoder로 변환합니다.
encoder = LabelEncoder()
encoder.fit(items)
labels = encoder.transform(items)
# 2차원 데이터로 변환합니다.
labels = labels.reshape(-1, 1)

# 원-핫 인코딩을 적용합니다.
oh_encoder = OneHotEncoder( )
oh_encoder.fit(labels)
oh_labels = oh_encoder.transform(labels)
print('원-핫 인코딩 데이터')
print(oh_labels.toarray())
print('원-핫 인코딩 데이터 차원')
print(oh_labels.shape)
```



원-핫 인코딩 데이터

```
[[1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]]
```

원-핫 인코딩 데이터 차원
(8, 6)

원본 데이터	원-핫 인코딩					
상품 분류	상품분류_ TV	상품분류_ 냉장고	상품분류_ 믹서	상품분류_ 선풍기	상품분류_ 전자렌지	상품분류_ 컴퓨터
TV	1	0	0	0	0	0
냉장고	0	1	0	0	0	0
전자렌지	0	0	0	0	1	0
컴퓨터	0	0	0	0	0	1
선풍기	0	0	0	1	0	0
선풍기	0	0	0	1	0	0
믹서	0	0	1	0	0	0
믹서	0	0	1	0	0	0

+ 판다스에서는 사이킷런의 OneHotEncoder와 다르게 `get_dummies()`를 이용해 문자열 카테고리 값을 숫자 형으로 변환할 필요 없이 바로 변환할 수 있음.

#2.5 데이터 전처리 – 피처 스케일링과 정규화

피처 스케일링(feature scaling)

- 서로 다른 변수의 값 범위를 일정한 수준으로 맞추는 작업.
- 대표적인 방법으로 표준화(Standardization)와 정규화(Normalization)가 있음.

* 표준화: 데이터의 피처 각각이 평균이 0이고 분산이 1인
가우시안 정규 분포를 가진 값으로 변환하는 것



$$x_{i_new} = \frac{x_i - \text{mean}(x)}{\text{stdev}(x)}$$

* 정규화: 서로 다른 피처의 크기를 통일하기 위해 크기를
변환해주는 개념.



$$x_{i_new} = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

※ 사이킷런의 전처리에서 제공하는 정규화 모듈 ≠ 일반적인 정규화

사이킷런의 정규화 모듈은 선형대수 개념의 정규화로, 개별 벡터의 크기를 맞추기 위해 변환하는 것을 의미함.

∴ 일반적 의미의 표준화와 정규화 → 피처 스케일링, 선형대수 개념의 정규화 → 벡터 정규화

#2.5 데이터 전처리 – StandardScaler

StandardScaler

- 개별 피처를 평균이 0이고, 분산이 1인 값으로 변환해줌.
- ‘서포트 벡터 머신(Support Vector Machine)’ 이나 ‘선형 회귀(Linear Regression)’, ‘로지스틱 회귀(Logistic Regression)’의 예측 성능 향상에 중요한 요소가 될 수 있음.

```
[32] from sklearn.datasets import load_iris
import pandas as pd
# 붓꽃 데이터 세트를 로딩하고 DataFrame으로 변환합니다.
iris = load_iris()
iris_data = iris.data
iris_df = pd.DataFrame(data=iris_data, columns=iris.feature_names)

print('feature 들의 평균 값')
print(iris_df.mean())
print('\nfeature 들의 분산 값')
print(iris_df.var())
```

```
⇒ feature 들의 평균 값
sepal length (cm)    5.843333
sepal width (cm)     3.057333
petal length (cm)    3.758000
petal width (cm)     1.199333
dtype: float64
```

```
feature 들의 분산 값
sepal length (cm)    0.685694
sepal width (cm)     0.189979
petal length (cm)    3.116278
petal width (cm)     0.581006
dtype: float64
```

```
▶ from sklearn.preprocessing import StandardScaler

# StandardScaler 객체 생성
scaler = StandardScaler()
# StandardScaler로 데이터 세트 변환. fit()과 transform() 호출
scaler.fit(iris_df)
iris_scaled = scaler.transform(iris_df)

# transform() 시 스케일 변환된 데이터 세트가 NumPy ndarray로 반환돼 이를 DataFrame으로 변환
iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_names)
print('feature 들의 평균 값')
print(iris_df_scaled.mean())
print('\nfeature 들의 분산 값')
print(iris_df_scaled.var())
```

```
⇒ feature 들의 평균 값
sepal length (cm)    -1.690315e-15
sepal width (cm)     -1.842970e-15
petal length (cm)    -1.698641e-15
petal width (cm)     -1.409243e-15
dtype: float64
```

```
feature 들의 분산 값
sepal length (cm)    1.006711
sepal width (cm)     1.006711
petal length (cm)    1.006711
petal width (cm)     1.006711
dtype: float64
```

#2.5 데이터 전처리 – MinMaxScaler

MinMaxScaler

- 데이터 값을 0과 1사이의 범위 값으로 변환. (음수 값이 있으면 -1에서 1값으로 변환)
- 데이터 분포가 가우시안 분포가 아닐 경우 적용 가능.

```
[34] from sklearn.preprocessing import MinMaxScaler
# MinMaxScaler 객체 생성
scaler = MinMaxScaler()
# MinMaxScaler로 데이터 세트 변환. fit()과 transform() 호출
scaler.fit(iris_df)
iris_scaled = scaler.transform(iris_df)

# transform() 시 스케일 변환된 데이터 세트가 NumPy ndarray로 반환돼 이를 DataFrame으로 변환
iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_names)
print('feature들의 최솟값')
print(iris_df_scaled.min())
print('\nfeature들의 최댓값')
print(iris_df_scaled.max())
```

↔ feature들의 최솟값

sepal length (cm)	0.0
sepal width (cm)	0.0
petal length (cm)	0.0
petal width (cm)	0.0

dtype: float64

feature들의 최댓값

sepal length (cm)	1.0
sepal width (cm)	1.0
petal length (cm)	1.0
petal width (cm)	1.0

dtype: float64

#2.5 데이터 전처리 – 학습 데이터와 테스트 데이터의 스케일링 변환 시 유의점

- fit(): 데이터 변환을 위한 기준 정보(Ex) 데이터 세트의 최댓값/최솟값 등) 설정을 적용.
- transform(): 설정된 정보를 이용해 데이터를 변환.
- fit_transform(): fit()과 transform()을 한번에 적용.

※ 학습 데이터와 테스트 데이터의 스케일링 변환 시 유의점

1. 전체 데이터의 스케일링 변환을 적용한 뒤 학습과 테스트 데이터로 분리.
2. 1이 여의치 않다면 테스트 데이터 변환 시에는 fit()이나 fit_transform()을 적용하지 않고 학습 데이터로 이미 fit() 된 Scaler 객체를 이용해 transform()으로 변환.

#2.5 데이터 전처리 – 학습 데이터와 테스트 데이터의 스케일링 변환 시 유의점

<테스트 데이터 변환 시 fit() 적용한 경우>

```
[37] # MinMaxScaler 객체에 별도의 feature_range 파라미터 값을 지정하지 않으면 0~1 값으로 변환
scaler = MinMaxScaler()

# fit()하게 되면 train_array 데이터의 최솟값이 0, 최댓값이 10으로 설정
scaler.fit(train_array)

# 1/10 scale로 train_array 데이터 변환함. 원본 10-> 1로 변환됨
train_scaled = scaler.transform(train_array)

print('원본 train_array 데이터:', np.round(train_array.reshape(-1), 2))
print('Scale된 train_array 데이터:', np.round(train_scaled.reshape(-1), 2))
```

⇒ 원본 train_array 데이터: [0 1 2 3 4 5 6 7 8 9 10]
Scale된 train_array 데이터: [0. 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.]

```
38] # MinMixscaler에 test_array를 fit()하게 되면 원본 데이터의 최솟값이 0, 최댓값이 5로 설정됨
scaler.fit(test_array)

# 1/5 scale로 test_array 데이터 변환함. 원본 5->1로 변환
test_scaled = scaler.transform(test_array)

# test_array의 scale 변환 출력
print('원본 test_array 데이터:', np.round(test_array.reshape(-1), 2))
print('Scale된 test_array 데이터:', np.round(test_scaled.reshape(-1), 2))
```

⇒ 원본 test_array 데이터: [0 1 2 3 4 5]
Scale된 test_array 데이터: [0. 0.2 0.4 0.6 0.8 1.]

<학습 데이터로 fit()을 수행한 transform()을 이용해 데이터를 변환한 경우 >

```
[39] scaler = MinMaxScaler()
scaler.fit(train_array)
train_scaled = scaler.transform(train_array)
print('원본 train_array 데이터:', np.round(train_array.reshape(-1), 2))
print('Scale된 train_array 데이터:', np.round(train_scaled.reshape(-1), 2))
```

```
# test_array에 Scale 변환을 할 때는 반드시 fit()을 호출하지 않고 transform()만으로 변환해야 함.
test_scaled = scaler.transform(test_array)
print('\n원본 test_array 데이터:', np.round(test_array.reshape(-1), 2))
print('Scale된 test_array 데이터:', np.round(test_scaled.reshape(-1), 2))
```

⇒ 원본 train_array 데이터: [0 1 2 3 4 5 6 7 8 9 10]
Scale된 train_array 데이터: [0. 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.]

원본 test_array 데이터: [0 1 2 3 4 5]
Scale된 test_array 데이터: [0. 0.1 0.2 0.3 0.4 0.5]

Chapter 03. 평가



#3.0 평가의 개요

#1 머신러닝의 단계

데이터 가공/변환 -> 모델 학습/예측 -> 평가 (Evaluation)

#2 분류에서 사용되는 성능 평가 지표

정확도
오차 행렬
정밀도
재현율
F1 Score
ROC AUC

#3.1 정확도

#1 정확도(Accuracy)란?

- 실제 데이터에서 예측 데이터가 얼마나 같은지를 판단하는 지표
- 직관적으로 모델 예측 성능을 나타냄

$$\text{정확도(Accuracy)} = \frac{\text{예측 결과가 동일한 데이터 건수}}{\text{전체 예측 데이터 건수}}$$

#2 불균형한 레이블 데이터 분포의 경우

- > 정확도 평가 지표의 맹점
- > 여러 가지 분류 지표와 함께 적용하는 것이 바람직

#3.2 오차 행렬

#1 오차 행렬 (Confusion matrix)

- 이진 분류의 예측 오류가 얼마인지, 어떠한 유형의 예측 오류가 발생하고 있는지를 나타내는 지표
- 이진 분류의 성능 지표로 잘 활용됨

#3.2 오차 행렬

		예측 클래스 (Predicted Class)	
		Negative(0)	Positive(1)
실제 클래스 (Actual Class)	Negative(0)	TN (True Negative)	FP (False Positive)
	Positive(1)	FN (False Negative)	TP (True Positive)

#1 오차 행렬 (Confusion matrix)

- **TN** : 예측값을 Negative 값인 0으로 예측했고 실제 값 역시 Negative 값인 0
- **FP** : 예측값을 Positive 값인 1로 예측했으나 실제 값은 Negative 값인 0
- **FN** : 예측값을 Negative 값인 0으로 예측했으나 실제 값은 Positive 값인 1
- **TP** : 예측값을 Positive 값인 1로 예측했고 실제 값 역시 Positive 값인 1

#3.2 오차 행렬

#3 오차 행렬 (Confusion matrix)

		예측 클래스 (Predicted Class)	
		Negative(0)	Positive(1)
실제 클래스 (Actual Class)	Negative(0)	TN (True Negative)	FP (False Positive)
	Positive(1)	FN (False Negative)	TP (True Positive)

-> TN, FP, FN, TP 값을 다양하게 결합해 분류
모델 예측 성능의 오류 발생 모습을 알 수 있음

-> 정확도의 경우

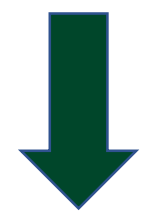
정확도 = 예측 결과와 실제 값이 동일한 건수/전체 데이터 수 = $(TN + TP) / (TN + FP + FN + TP)$

#3.2 오차 행렬

#2 confusion_matrix()

- 오차 행렬을 구하기 위해 사이킷런이 제공하는 API
- 인자: confusion_matrix(실제 결과, 예측 결과)
- 출력
 - : ndarray 형태로 출력됨
 - : array에서 TN, FP, FN, TP 는 좌측의 도표와 동일한 위치에 있음

```
from sklearn.metrics import confusion_matrix  
  
confusion_matrix(y_test, fakepred)
```



[output]

```
array([[405,  0],  
       [ 45,  0]])
```

		예측 클래스 (Predicted Class)	
		Negative(0)	Positive(1)
실제 클래스 (Actual Class)	Negative(0)	TN (True Negative)	FP (False Positive)
	Positive(1)	FN (False Negative)	TP (True Positive)

#3.3 정밀도와 재현율

#1 정밀도 [= 양성 예측도]

- $\text{정밀도} = \frac{TP}{FP+TP}$
- 예측을 Positive로 한 대상 중 예측과 실제 값이 Positive로 일치한 데이터의 비율
- precision_score()

#2 재현율 [= 민감도(Sensitivity) = TPR(True Positive Rate)]

- $\text{재현율} = \frac{TP}{FN+TP}$
- 실제 값이 Positive인 대상 중 예측과 실제 값이 Positive로 일치한 데이터의 비율
- recall_score()

#3.3 정밀도와 재현율

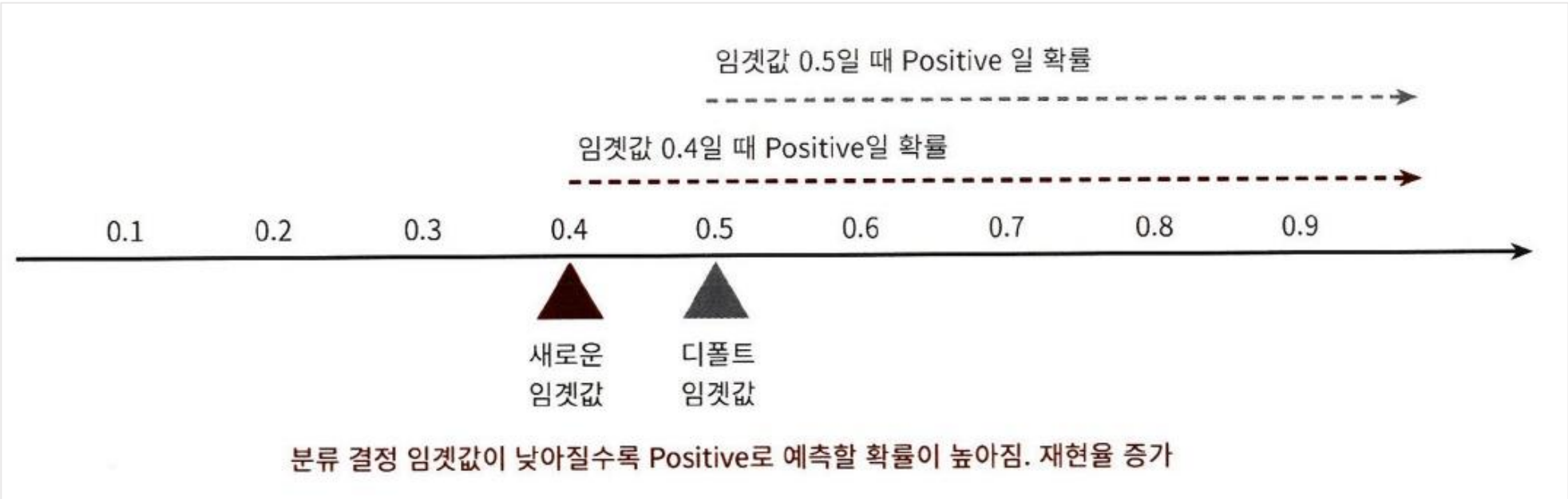
#3 정밀도/재현율 트레이드오프

- 상호 보완적인 평가 지표
- 결정 임계값(threshold)을 변화시켜 수치 조정

#4 분류 결정 임계값의 조정과 정밀도/재현율의 맹점

⇒ 임계값을 감소시킨 경우
: 재현율이 올라가고 정밀도가 떨어짐

⇒ 임계값을 증가시킬 경우

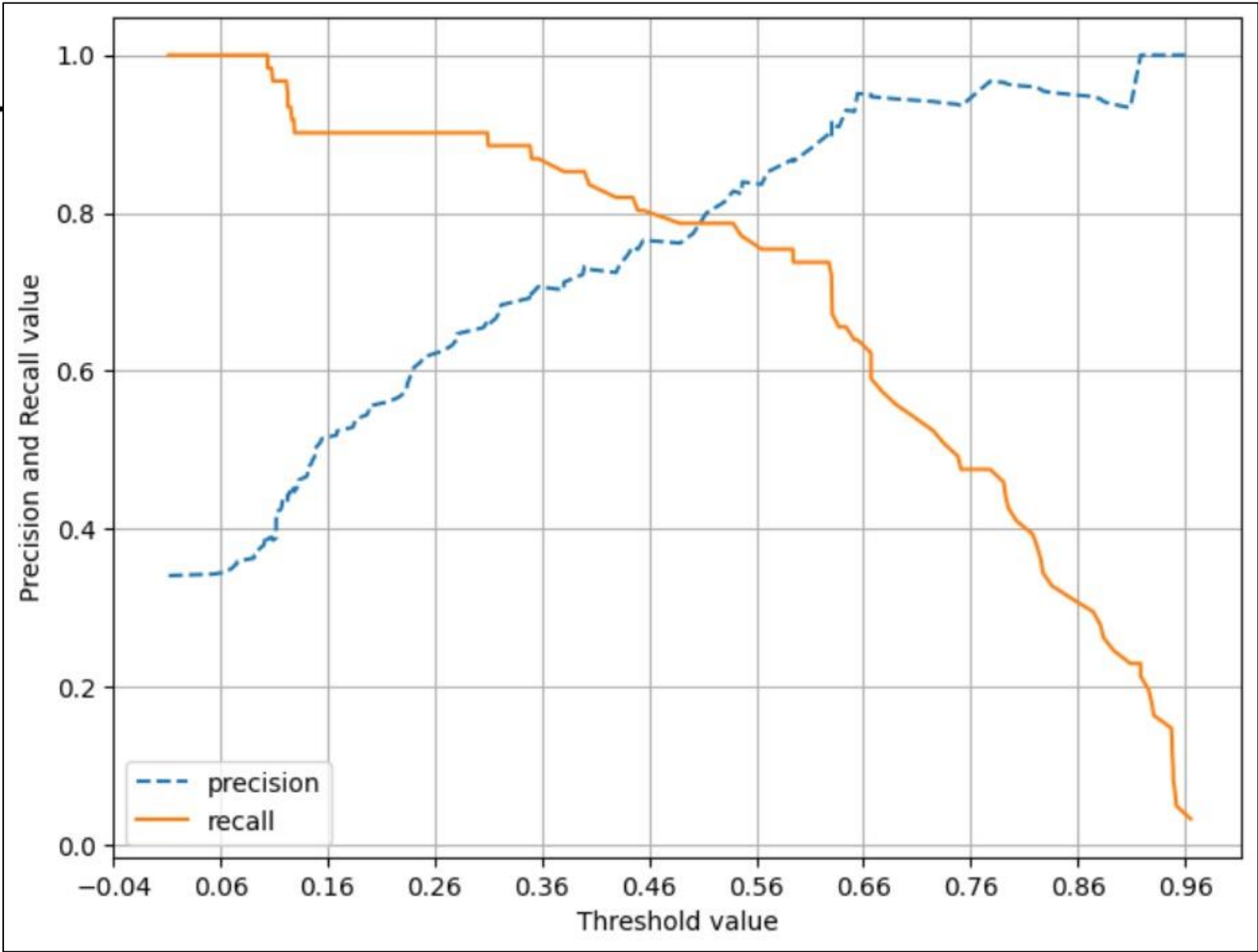


평가 지표	분류 결정 임계값				
	0.4	0.45	0.5	0.55	0.6
정확도	0.8212	0.8547	0.8659	0.8715	0.8771
정밀도	0.7042	0.7869	0.8246	0.8654	0.8980
재현율	0.8197	0.7869	0.7705	0.7377	0.7213

#3.3 정밀도와 재현율

#5 precision_recall_curve()

- 임계값에 따른 정밀도와 재현율 값을 배열로 변환
- 정밀도와 재현율의 임계값에 따른 값 변화를 곡선 형태의 그래프로 시각화 가능



입력 파라미터	y_true: 실제 클래스값 배열 (배열 크기 = [데이터 건수]) probas_pred: Positive 칼럼의 예측 확률 배열 (배열 크기 = [데이터 건수])
반환 값	정밀도: 임계값 별 정밀도 값을 배열로 변환 재현율: 임계값 별 재현율 값을 배열로 변환

#3.4 F1 Score

#1 F1 Score

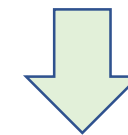
- 정밀도와 재현율을 결합한 지표
- 정밀도와 재현율이 한 쪽으로 치우치지 않는 경우 상대적으로 높은 값을 가짐

$$F1 = \frac{2}{\frac{1}{recall} + \frac{1}{precision}} = 2 * \frac{precision * recall}{precision + recall}$$

#2 f1_score()

- F1 Score를 구하기 위해 사이킷런이 제공하는 API

```
from sklearn.metrics import f1_score  
f1 = f1_score(y_test, pred)  
print('F1 스코어: {0:.4f}'.format(f1))
```

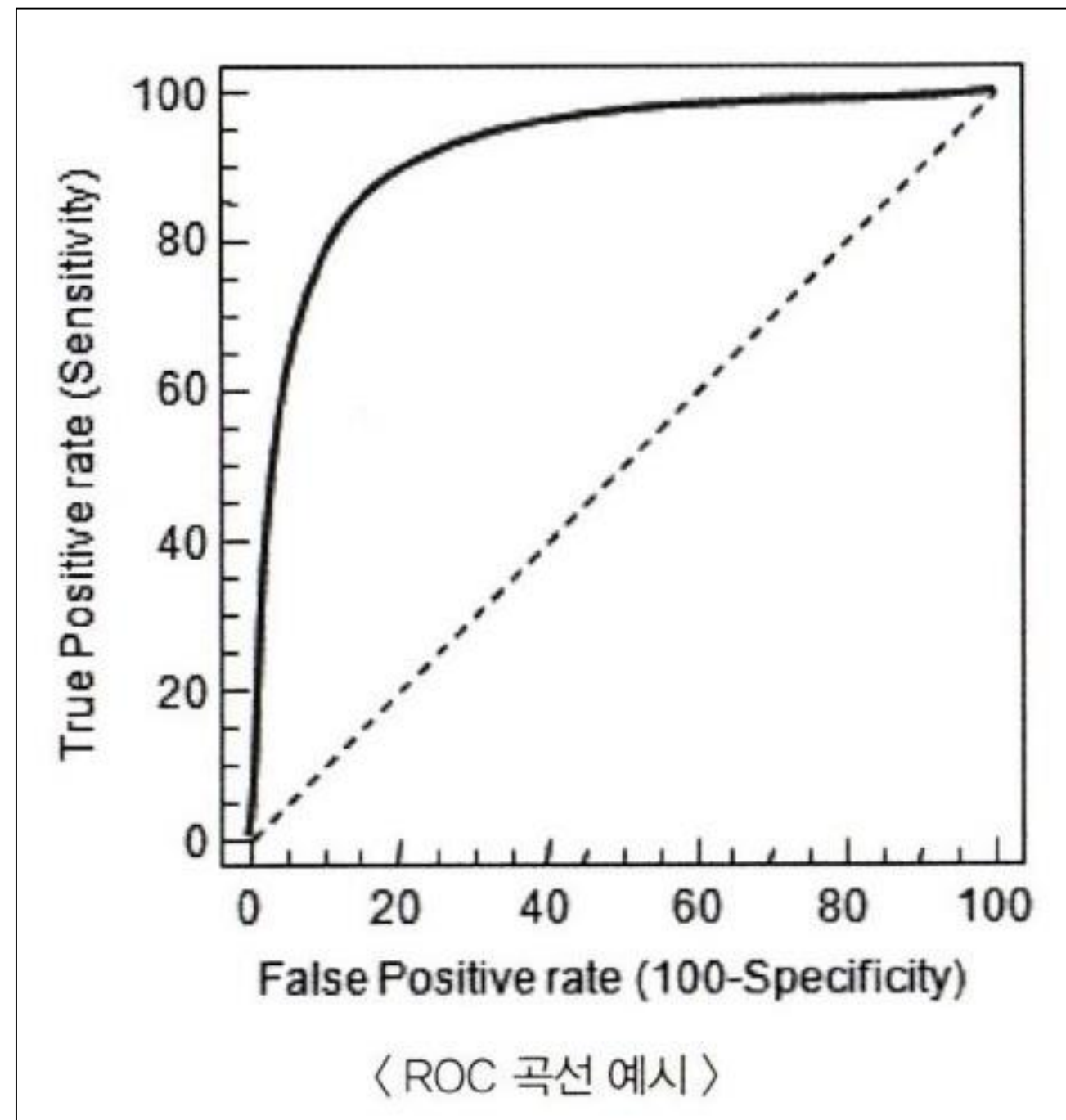


F1 스코어: 0.7966

#3.5 ROC Curve, AUC

#1 ROC Curve (Receiver Operation Characteristic Curve)

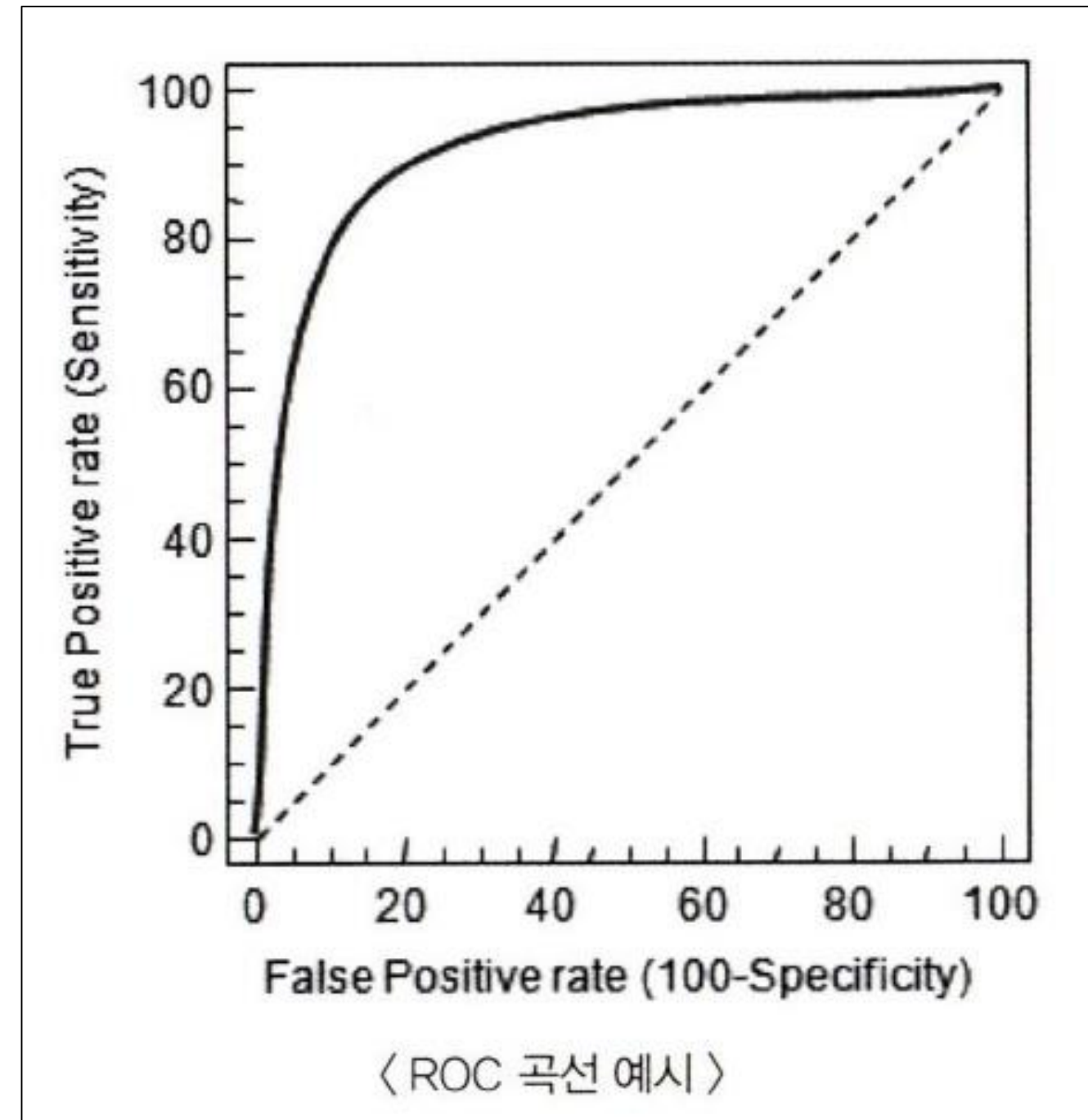
- FPR(False Positive Rate)이 변할 때 TPR(True Positive Rate)이 어떻게 변하는지 나타내는 곡선



#3.5 ROC Curve, AUC

- TPR(True Positive Rate) = 재현율 = 민감도
 - $TPR = TP / (FN + TP)$
 - 실제값 positive가 정확히 예측되어야 하는 수준
- TNR(True Negative Rate) = 특이성(Specificity)
 - $TNR = TN / (FP + TN)$
 - 실제값 negative가 정확히 예측되어야 하는 수준
- FPR(False Positive Rate)
 - $FPR = FP / (FP + TN) = 1 - TNR = 1 - \text{특이성}$
 - ROC 곡선의 X 축
- `roc_curve()`

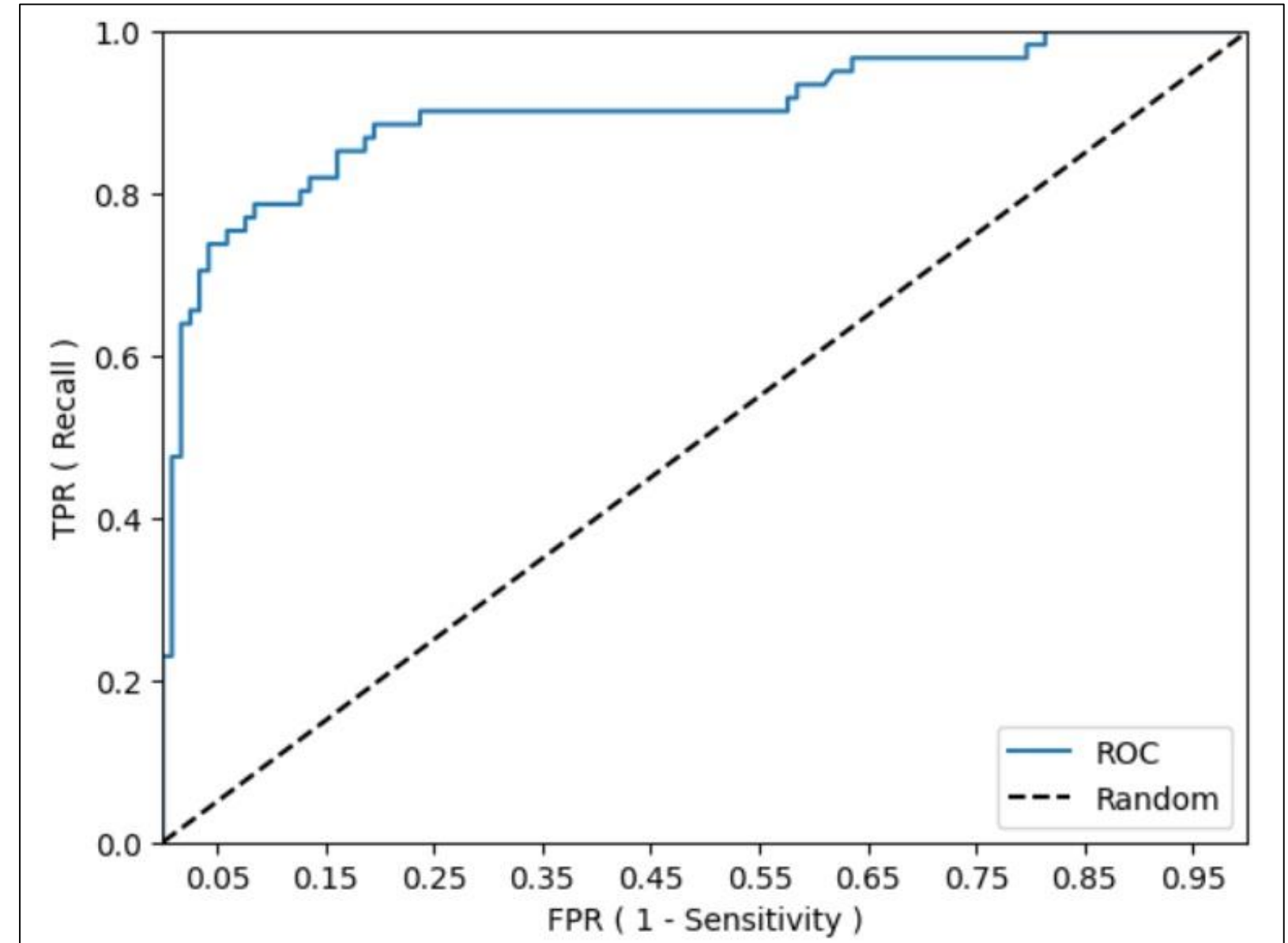
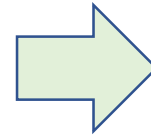
```
from sklearn.metrics import roc_curve
pred_proba_class1 = lr_clf.predict_proba(X_test)[: , 1]
fprs, tprs, thresholds = roc_curve(y_test, pred_proba_class1)
```



#3.5 ROC Curve, AUC

- `roc_curve()`
- `def roc_curve_plot():`

```
def roc_curve_plot(y_test, pred_proba_c1):  
    # 임계값에 따른 FPR, TPR 값을 반환받음  
    fprs, tprs, thresholds = roc_curve(y_test, pred_proba_c1)  
    # ROC 곡선을 그래프 곡선으로 그림  
    plt.plot(fprs, tprs, label = 'ROC')  
    # 가운데 대각선 직선을 그림  
    plt.plot([0,1], [0,1], 'k--', label = 'Random')  
  
    # FPR X 축의 Scale을 0.1 단위로 변경, X, Y축 명 설정 등  
    start, end = plt.xlim()  
    plt.xticks(np.round(np.arange(start, end, 0.1), 2))  
    plt.xlim(0,1); plt.ylim(0,1)  
    plt.xlabel('FPR ( 1 - Sensitivity )'); plt.ylabel('TPR ( Recall )')  
    plt.legend()  
  
    roc_curve_plot(y_test, pred_proba[:,1])
```



#2 AUC (Area Under Curve)

- ROC 곡선 밑의 면적을 구한 것
- 1에 가까울수록 좋은 수치
- `roc_auc_score()`

Q & A



THANK YOU

