😊

2장 분류(Classification)와 지도학습(Supervised Learning)

0. 사이킷런(Scikit-learn)

(1) 지도 학습

- **정의**: 입력 데이터(피처, Feature)와 정답(레이블, Label)을 함께 학습하여, 새로운 데이터의 정답을 예측하는 학습 방법.
- **특징**: 명확한 정답(Label)이 있는 데이터를 학습 후, 미지의 정답을 예측.
- **데이터 구분**
  - **학습 데이터(Train set)**: 모델 학습용
  - **테스트 데이터(Test set)**: 성능 평가용

  ***피처(feauture) = 입력값(Input variable)**
  - 즉, 우리가 모델에게 "이런 조건들을 보고 판단해줘"라고 주는 **설명 변수**
  - 예: 붓꽃(Iris) 데이터에서 꽃받침 길이, 꽃받침 너비, 꽃잎 길이, 꽃잎 너비

  ***레이블(label)**: 분류 결과 (붓꽃 품종: Setosa, Versicolor, Virginica)
  - DataFrame 변환 시 → 피처 + 레이블을 한눈에 확인 가능

(2) 사이킷런(Scikit-learn) 모듈

- **sklearn.datasets**: 데이터 세트 생성/로드 모듈
  - 예: `load_iris()` (붓꽃 데이터)
- **sklearn.tree**: 트리 기반 알고리즘 모듈
  - 예: `DecisionTreeClassifier`

    - 트리: 나무 모양의 구조로 데이터를 분류하거나 예측하는 알고리즘
- **sklearn.model_selection**: 데이터 분할 및 모델 성능 평가 모듈
    - 예: `train_test_split()`, 교차 검증, 하이퍼파라미터 튜닝 등

(3) 하이퍼 파라미터(Hyperparameter)

- **정의**: 알고리즘 학습 시 **사람이 직접 지정**해주는 값
- **역할**: 모델의 학습 방식과 성능에 큰 영향을 미침
- **예시**: 의사결정트리에서 `max_depth`, `min_samples_split` 등

💥학습 과제: 붓꽃(Iris) 데이터 분류 실습

- 절차
    1. **데이터 불러오기 (`load_iris`)**
    2. **데이터 분리 (`train_test_split`)**
    3. **모델 학습 (`DecisionTreeClassifier.fit`)**
    4. **예측 (`predict`) 및 평가**
        - iris.data: 모든 피처 데이터
        - iris.feature_names: 피처 데이터의 칼럼명
        - iris.target: 모든 레이블 데이터
        - iris.target_names: 레이블의 원래 이름

        → numpy ndarray으로 가지고 있음

---

1. 데이터 불러오기

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier from sklearn.model_selecti
on import train_test_split
```

```
import pandas as pd

#데이터 세트 로딩
iris = load_iris()

#피쳐 데이터 numpy
iris_data = iris.data

#레이블값
iris_label = iris.target
print(iris_label)
print(iris.target_names)

iris_df = pd.DataFrame(data = iris_data, columns=iris.feature_names)
iris_df['label'] = iris.target
iris_df.head(3)
```

---

2. 학습/테스트 데이터 분리

(1) 분리의 필요성

- 머신러닝 모델은 훈련(학습)과 평가(테스트)를 구분해야 함
- 이유: 학습 데이터만으로 평가하면 **과적합(Overfitting) 문제**가 발생하여 실제 성능을 알 수 없음
    - 과적합(Overfitting) 문제: 훈련 데이터의 패턴뿐 아니라 노이즈(우연한 잡음)까지 외워버려 **일반화(generalization) 능력이 약해지는 것**

(2) train_test_split() 함수

사이킷런에서 데이터 분리를 위해 제공하는 API

* 주요 파라미터

- 첫 번째 인자: 피처 데이터 (**X**) (iris.data)
- 두 번째 인자: 레이블 데이터 (**y**) (iris.target)
- test_size: 테스트 데이터 비율 (예: 0.2 → 20%)
- random_state: 난수 시드(seed). 실행마다 같은 결과를 재현하기 위해 사용

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

(3) 반환값 설명

- X_train: 학습용 피처 데이터
- X_test: 테스트용 피처 데이터
- y_train: 학습용 레이블 데이터
- y_test: 테스트용 레이블 데이터

---

3. 의사결정트리: 학습·예측·평가

(1) 모델 학습

```
# 모델 생성
dt_clf = DecisionTreeClassifier(random_state=11)
```

```
# 학습 수행
dt_clf.fit(X_train, y_train)
```

- 사이킷런의 DecisionTreeClassifier 객체 생성 후
  - fit() 메서드 → 학습
  - predict() 메서드 → 예측
- X_train, y_train: train_test_split으로 분리해 둔 학습용 피처/레이블
- fit(): 학습 데이터를 이용해 모델 파라미터를 학습

(2) 테스트 데이터로 예측

```
# 테스트 세트 예측
pred = dt_clf.predict(X_test)
```

- predict(): 학습된 모델로 보지 않은 데이터(X_test)에 대한 레이블 예측
- 예측은 반드시 학습에 쓰지 않은 데이터(보통 테스트 세트)로 수행

---

(3) 정확도(Accuracy)로 평가

```
from sklearn.metrics import accuracy_score

acc = accuracy_score(y_test, pred)
print(f"예측 정확도: {acc:.4f}")
```

- **accuracy_score(실제값, 예측값)**
- 예: 예측 정확도: 0.9333 → 약 93.33%

---

1. 데이터 세트 분리: 학습/테스트로 분리 (train_test_split)
2. 모델 학습: 의사결정트리(DecisionTreeClassifier).fit(X_train, y_train)
3. 예측 수행: predict(X_test)
4. 평가: accuracy_score(y_test, pred)로 성능 측정

4. 사이킷런 기반 프레임워크 핵심 정리

(1) Estimator와 기본 메서드

- Estimator
  - Classifier: 분류 알고리즘 클래스
  - Regressor: 회귀 알고리즘 클래스
- Estimator의 공통 메서드
  - **fit(): 학습 수행**
  - **predict(): 예측 수행**
  - **cross_val_score() & GridSearchCV**: Estimator를 인자로 받아 내부적으로 fit(), predict()를 호출
- 비지도학습(차원 축소, 클러스터링, 피처 추출)
  - fit(): 학습이 아니라, 입력 데이터 형태에 맞춰서 데이터를 변환하기 위한 사전 구조 생성
  - transform(): 입력 데이터의 차원 변환, 클러스터링 ,피처 추출 등 변환 작업 수행
  - fit_transform(): fit() + transform()을 한 번에 실행

(3) 사이킷런 주요 모듈 요약

예제 데이터

- **sklearn.datasets:** 내장 데이터 세트 제공 (iris, digits, cancer, boston 등)

피처 처리

- sklearn.preprocessing: 데이터 전처리 (인코딩, 정규화, 스케일링 등)
- sklearn.feature_selection: 중요 피처 선별 기능
- sklearn.feature_extraction: 텍스트, 이미지 등에서 피처 추출

피처처리 & 차원 축소

- sklearn.decomposition: 차원 축소 알고리즘 (PCA, NMF, Truncated SVD 등)

데이터 분리, 검증 & 파라미터 튜닝

- **sklearn.model_selection: 데이터 분리, 교차 검증, 파라미터 튜닝**

평가

- **sklearn.metrics: 다양한 성능 측정 지표 (Accuracy, Precision, Recall, ROC-AUC, RMSE 등)**

ML 알고리즘

- sklearn.ensemble: 앙상블 알고리즘 (RandomForest, AdaBoost, GradientBoosting 등)
- sklearn.linear_model: 회귀 알고리즘 (Linear, Ridge, Lasso, Logistic, SGD 등)
- sklearn.naive_bayes: 나이브 베이즈 계열
- sklearn.neighbors: 최근접 이웃 (K-NN 등)
- sklearn.svm: 서포트 벡터 머신
- **sklearn.tree: 의사결정트리**
- sklearn.cluster: 클러스터링 (K-means, DBSCAN 등)

유틸리티

- sklearn.pipeline: 전처리와 학습 과정을 묶어 실행할 수 있는 유틸리티

(4) 내장 예제 데이터 세트

분류/회귀 연습용 데이터

- 분류: datasets.load_iris, digits, breast_cancer()
- 회구: datasets.load_diabetes, boston()

표본 데이터 생성기

- datasets.make_classification(): 분류를 위한 데이터 세트

- datasets.make_blobs(): 클러스터링을 위한 데이터 세트

fetch 계열: 데이터가 커서 인터넷에 연결하여 내려 받음

- fetch_covtype() : 회귀 분석용 토지 조사 자료
- fetch_20newsgroups() : 뉴스 그룹 텍스트 자료
- fetch_olivetti_faces() : 얼굴 이미지 자료
- fetch_lfw_people() : 얼굴 이미지 자료
- fetch_lfw_pairs() : 얼굴 이미지 자료
- fetch_rcv1() : 로이터 뉴스 말뭉치
- fetch_mldata() : ML 웹사이트에서 다운로드

데이터는 딕셔너리 유사 객체(Bunch)로 제공되며, 주요 키는 다음과 같음

- **data**: 피처 데이터
- **target**: 레이블 또는 수치 결과
- **target_names**: 레이블 이름
- **feature_names**: 피처 이름
- DESCR: 데이터 세트 설명

5. Model Selection 모듈 소개 (scikit-learn)

- model_selection 모듈은 **데이터 분리, 교차 검증, 하이퍼파라미터 튜닝**을 위한 함수/클래스를 제공
- 핵심 기능
  - 데이터 분리: train_test_split
  - 교차 검증: KFold, StratifiedKFold, cross_val_score
  - 하이퍼파라미터 튜닝: GridSearchCV, RandomizedSearchCV

(1) 학습/테스트 데이터 분리: train_test_split

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

iris = load_iris()
dt_clf = DecisionTreeClassifier()
train_data = iris.data
train_label = iris.target
dt_clf.fit(train_data, train_label)

pred = dt_clf.predict(train_data)
print("예측 정확도: ", accuracy_score(train_label, pred))
```

```
예측 정확도:  1.0
```

- 학습한 세트로 테스트하여 정확도가 1.0
- train_test_split()를 사용해야 함
    - test_size : 전체 데이터에서 테스트 데이터 세트의 비율을 결정. 기본값은 0.25 (25%).
    - train_size : 전체 데이터에서 학습용 데이터 세트의 비율을 결정. 보통은 test_size를 주로 사용하므로 잘 쓰이지 않음.
    - shuffle : 분할하기 전에 데이터를 섞을지 여부. 기본값은 True. 데이터를 무작위로 섞어 학습/테스트 데이터의 편향을 줄임.
    - random_state : 난수 시드(seed). train_test_split은 호출할 때마다 무작위 분할을 하므로, 같은 random_state 값을 주면 항상 동일한 학습/테스트 세트를 생성할 수 있음. 재현성을 위해 보통 특정 숫자를 고정해 사용.

  반환값 : 튜플 형태로 반환되며, 순서는 (학습용 피처, 테스트용 피처, 학습용 레이블, 테스트용 레이블)

```
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.targe
t,
                                                    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) 교차 검증

- 고정된 하나의 학습/테스트 분할에 **과적합**되거나 **편향**될 수 있으므로, 데이터를 여러 폴드로 나눠 반복 학습·평가
  - 과적합(overfitting): 훈련 데이터의 세세한 패턴, 노이즈까지 학습해서 훈련 데이터에는 너무 잘 맞지만 일반화가 안되는 상태
  - 편향(bias): 특정 데이터 분할, 특정 방향으로 치우쳐서 잘못된 평가를 내리는 것
    ex) 반 학생 30명 중 상위권 5명만 운 나쁘게 뽑히면 평균을 너무 높게 추정
  - 따라서 고정된 한 번의 분할은 위험하고, 여러 번 데이터를 나누고 평가하여 평균 성능을 쓰면 더 안정적인 성능 평가 가능
- 일반 절차
  1. 데이터를 K개 폴드로 분할
  2. 각 반복에서 K−1개 폴드로 학습, 남은 1개 폴드로 검증
  3. K번의 점수를 평균해 일반화 성능 추정

🔎 교차검증에서 데이터가 어떻게 나뉘는지

- **KFold / StratifiedKFold**
  - 데이터를 **겹치지 않게 n등분**해서 사용.
  - 그래서 각 샘플은 **검증 세트(validation)에는 딱 한 번만** 등장.

- 하지만 train 세트에는 여러 번 등장 가능.
- 이게 바로 교차검증의 핵심 → 모든 샘플이 한 번은 검증(validation)으로 쓰이도록 보장

  예시 (데이터 6개, 3-fold), 샘플 = `[A, B, C, D, E, F]`

  - Fold 1: train = `[C, D, E, F]`, val = `[A, B]`
  - Fold 2: train = `[A, B, E, F]`, val = `[C, D]`
  - Fold 3: train = `[A, B, C, D]`, val = `[E, F]`

  👉 각 fold 안에서는 **중복 없음**

  👉 전체 교차검증 과정에서는 샘플 A~F 모두 **validation으로 딱 1번씩** 등장

KFold()

- 단순 K등분 교차 검증
- 분류 레이블 비율을 보장하지 않음(불균형 데이터에 취약)

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import KFold
import numpy  as np
iris = load_iris()
features = iris.data
label = iris.target
dt_clf = DecisionTreeClassifier(random_state=156)

# 5개의 폴드 세트로 분리하는 KFold 객체와 폴드 세트별 정확도를 담을 리스트 객체  생성
kfold = KFold(n_splits=5)
cv_accuracy = []
print('붓꽃 데이터 세트 크기:', features.shape[0])
```

```
붓꽃 데이터 크기: 150n_iter = 0
for train_index, test_index in kfold.split(features):
  X_train, X_test = features[train_index], features[test_index]
  y_train, y_test = label[train_index], label[test_index]
```

```
    dt_clf.fit(X_train, y_train)
    pred = dt_clf.predict(X_test)
    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('\n{0} 교차 검증 정확도: {1}, 학습 데이터 크기: {2}, 검증 데이터 크기: {3}'.fo
rmat(n_iter, accuracy, train_size, test_index))
    cv_accuracy.append(accuracy)

print("\n## 평균검증 정확도:", np.mean(cv_accuracy))
```

```
1 교차 검증 정확도: 1.0, 학습 데이터 크기: 120, 검증 데이터 크기: [ 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]

2 교차 검증 정확도: 0.9667, 학습 데이터 크기: 120, 검증 데이터 크기: [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]

3 교차 검증 정확도: 0.8667, 학습 데이터 크기: 120, 검증 데이터 크기: [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]

4 교차 검증 정확도: 0.9333, 학습 데이터 크기: 120, 검증 데이터 크기: [ 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]

5 교차 검증 정확도: 0.7333, 학습 데이터 크기: 120, 검증 데이터 크기: [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]

## 평균검증 정확도: 0.9
```

StratifiedKFold()

특징

- 분류 문제에서 **레이블 분포(클래스 비율)를 각 폴드에 동일하게 유지**
- 불균형 데이터 또는 클래스 수가 많은 데이터에 필수적
- `.split(X, y)` → **인덱스(index)** 를 반환하는 이유는, **데이터 자체를 직접 쪼개지 않고 "어떤 샘플을 학습/검증에 쓸지"를 지정하여 불필요한 데이터 복사를 막음**
    - 넘파이의 np.split(배열, 5)랑 다름 → 배열 반환

```
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier
import numpy as np

iris = load_iris()
X = iris.data
y = iris.target

skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=156)
scores = []

for train_idx, val_idx in skf.split(X, y):
    X_tr, X_val = X[train_idx], X[val_idx]
    y_tr, y_val = y[train_idx], y[val_idx]

    clf = DecisionTreeClassifier(random_state=156)
    clf.fit(X_tr, y_tr)
    pred = clf.predict(X_val)
    scores.append(accuracy_score(y_val, pred))

print("폴드별 정확도:", np.round(scores, 4))
print("평균 정확도:", np.round(np.mean(scores), 4))
```

```
폴드별 정확도: [0.9333 1.     0.9    0.9    0.9333]
평균 정확도: 0.9333
```

cross_val_score()로 간단 교차 검증

- KFold/StratifiedKFold 루프를 대신해 한 줄로 교차 검증 수행

- 분류 문제에서는 cv=StratifiedKFold가 자동 적용되는 경우가 많음

📌 주요 파라미터

- **estimator**: 사용할 **모델 객체** (예: `DecisionTreeClassifier()`)
- **X**: 입력 데이터(피처)
- **y**: 레이블
- **scoring**: 평가 지표 (예: `"accuracy"`, `"f1"`, `"roc_auc"`, `"r2"`)
- **cv**: 교차 검증에서 나눌 폴드 수 (예: 3 → 3등분, 5 → 5등분)

📌 반환 값

- 각 폴드에서 계산된 점수 배열 → `numpy.ndarray`
- 예: `[0.95 0.97 0.93]`

이 배열을 평균 → 교차 검증의 평균 성능

```
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier
import numpy as np

clf = DecisionTreeClassifier(random_state=156)
scores = cross_val_score(clf, X, y, scoring="accuracy", cv=5)
print("교차 검증별 정확도:", np.round(scores, 4))
print("평균 정확도:", np.round(np.mean(scores), 4))
```

```
교차 검증별 정확도: [0.9667 0.9667 0.9    0.9667 1.    ]
평균 정확도: 0.96
```

GridSearchCV로 하이퍼파라미터 튜닝

- 지정한 파라미터 그리드를 교차 검증으로 탐색해 최적 조합과 점수를 제공
  -
- 내부적으로 각 조합에 대해 fit, predict를 반복 수행
  - 각 조합마다 교차검증(cv) 수행

- 각 조합 별 나온 3번의 검증 점수를 평균화
- 평균 점수 가장 높은 하이퍼파라미터 조합을 최적으로 선택
- 하이퍼파라미터 조합 만들기

```
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier

param_grid = {
    "max_depth": [2, 3, 4, 5, None],
    "min_samples_split": [2, 4, 6, 8],
    "min_samples_leaf": [1, 2, 3]
}

clf = DecisionTreeClassifier(random_state=156)
grid = GridSearchCV(
    clf,
    param_grid=param_grid,
    scoring="accuracy",
    cv=5,
    n_jobs=-1
)

grid.fit(X, y)
print("최적 파라미터:", grid.best_params_)
print("최적 교차 검증 점수:", round(grid.best_score_, 4))

best_clf = grid.best_estimator_
```

```
최적 파라미터: {'max_depth': 3, 'min_samples_leaf': 1, 'min_samples_split': 2}
최적 교차 검증 점수: 0.9733
```

(3) 실무 체크리스트

- *데이터 분리 전에 누수 가능 전처리(스케일링, 인코딩 등) 금지 → 파이프라인 사용 권장*
- *분류 문제는 stratify=y 또는 StratifiedKFold 사용*

- *단일 스플릿 점수에 의존하지 말고 cross_val_score로 평균·분산 확인*
- *최종 평가는 오직 테스트 세트로 단 한 번 수행*
- *하이퍼파라미터는 GridSearchCV/RandomizedSearchCV로 조정하고 교차 검증 점수 기반으로 선택*

6. 데이터 전처리 (Data Preprocessing)

- 머신러닝 알고리즘만큼이나 **데이터 전처리**는 중요
- Garbage In, Garbage Out → 데이터 입력이 잘못되면 결과도 잘못됨.

결손값(Null, NaN) 처리

- **허용되지 않음** → 반드시 변환 필요
- 처리 방법:
  - 소수의 Null → 평균값, 중앙값 등으로 대체
  - Null 값이 대부분 → 해당 피처 **드롭(drop)**
  - 일정 수준 이상 Null 존재:
    - 명확한 기준은 없음
    - 중요한 피처라면 평균값 대체는 왜곡 가능 → **업무 로직 검토 후 정밀한 대체값 설정**

문자열 값 처리

- **사이킷런 알고리즘은 문자열 입력 불가**
- 모든 문자열 값 → 숫자형으로 인코딩 필요
- 문자열 피처 종류:
  - **카테고리형** → 코드 값(인코딩)으로 변환
  - **텍스트형** → 벡터화(feature vectorization) 또는 불필요하면 삭제

1) 데이터 인코딩

(1) 레이블 인코딩

사이킷런의 **레이블 인코딩**은 `LabelEncoder` 클래스로 구현

- `fit()` : 데이터 학습 (고유한 카테고리 추출 및 정렬)
- `transform()` : 문자열 → 숫자 코드 변환
- `inverse_transform()` : 숫자 코드 → 원래 문자열로 복원
- **숫자를 붙이는 기준**은 단순히 **고유값을 오름차순 정렬한 순서**

```
from sklearn.preprocessing import LabelEncoder

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

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

```
인코딩 변환값: [0 1 4 5 3 3 2 2]
```

- TV → 0
- 냉장고 → 1
- 믹서 → 2
- 선풍기 → 3
- 전자레인지 → 4
- 컴퓨터 → 5

인코딩된 클래스 확인 .classes_

```
print("인코딩 클래스:", encoder.classes_)
```

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

- `classes_` 속성은 **0번부터 순서대로 매핑된 원본 값**을 저장

  → 따라서 `0=TV, 1=냉장고, 2=믹서, 3=선풍기, 4=전자레인지, 5=컴퓨터`

디코딩 .inverse_transform

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

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

원본 데이터

상품 분류	가격
TV	1,000,000
냉장고	1,500,000
전자레인지	200,000
컴퓨터	800,000
선풍기	100,000
선풍기	100,000
믹서	50,000
믹서	50,000

상품 분류를 레이블 인코딩한 데이터

상품 분류	가격
0	1,000,000
1	1,500,000
4	200,000
5	800,000
3	100,000
3	100,000
2	50,000
2	50,000

⚠️ 레이블 인코딩의 한계

- 레이블 인코딩은 **카테고리형 데이터를 단순히 숫자로 치환**
- 숫자의 크기·순서를 **의미 있는 특징으로 잘못 해석**할 수 있음
- 특히, **선형 회귀(Linear Regression), 로지스틱 회귀, KNN, SVM, 신경망(ANN)** 등 거리나 선형 결합에 민감한 모델에서는 **예측 성능이 저하**될 수 있음
- **트리 계열(Tree-based) 알고리즘**에서는 큰 문제가 없음
  - 이유: 트리 모델은 숫자의 크기 순서가 아니라 분할 기준(조건)으로만 사용하기 때문

(2) 원-핫 인코딩 (One-Hot Encoding)

- **카테고리형 피처 → 고유 값 개수만큼 새로운 피처(column)로 분리**
- 해당 레코드의 카테고리에 해당하는 칼럼만 `1`, 나머지는 `0`으로 변환
- 이름 그대로 **"여러 속성 중 단 하나만 1"** 이라는 특징 때문에 *One-Hot* 인코딩이라 불림

원본 데이터

상품 분류
TV
냉장고
전자레인지
컴퓨터
선풍기
선풍기
믹서
믹서

원-핫 인코딩

상품분류_ TV	상품분류_ 냉장고	상품분류_ 믹서	상품분류_ 선풍기	상품분류_ 전자레인지	상품분류_ 컴퓨터
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

```
from sklearn.preprocessing import OneHotEncoder
import numpy as np

items = ['TV', '냉장고', '전자레인지', '컴퓨터', '선풍기', '선풍기', '믹서',  '믹서']
#2차원 ndarray로 변환
items = np.array(items).reshape(-1, 1)

#원핫인코딩
oh_encoder = OneHotEncoder()
oh_encoder.fit(items)
oh_labels = oh_encoder.transform(items)

#변환 결과는 희소행렬, toarray()를 이용해 밀집 행렬로 변환
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.]]

데이터 차원: (8, 6)
```

- 원본 데이터: **8개의 레코드, 1개 컬럼**
- 변환 후: **8개의 레코드, 6개 컬럼(고유값 개수)**
- 첫 번째 칼럼: `TV`, 두 번째 칼럼: `냉장고`, 세 번째: `믹서`, 네 번째: `선풍기`, 다섯 번째: `전자레인지`, 여섯 번째: `컴퓨터`

📌 `toarray()`

- `OneHotEncoder`는 기본적으로 희소행렬(sparse matrix)을 반환
- 사람이 확인하거나, 일부 연산에서는 **밀집행렬**이 필요할 수 있음
- 이때 `.toarray()`를 호출하면 → numpy ndarray 형태의 밀집행렬로 변환됨

📌 희소행렬 (Sparse Matrix)

- 대부분의 값이 `0`인 행렬
- **메모리 절약**을 위해 0은 저장하지 않고, 0이 아닌 값만 따로 저장
- `OneHotEncoder` 결과처럼 `0`이 많고 `1`이 드문 경우 효율적

예시 (원-핫 인코딩 결과, 희소 표현)

```
(0,0)   1.0
(1,2)   1.0
(2,1)   1.0
...
```

➡️ "행, 열, 값" 형태만 저장함

📌 밀집행렬 (Dense Matrix)

- 모든 값을 **그대로** 2차원 배열로 저장
- 사람이 보기에는 훨씬 직관적
- 하지만 `0`도 전부 저장하므로 메모리 낭비 발생

예시 (원-핫 인코딩 결과, 밀집 표현)

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

사이킷런의 `OneHotEncoder`는 문자열을 숫자로 먼저 변환해야 하지만, pandas는 **직접 문자열 카테고리를 처리**할 수 있음.

```
import pandas as pd

df = pd.DataFrame({
    'item': ['TV', '냉장고', '전자레인지', '컴퓨터', '선풍기', '선풍기', '믹서', '믹서']
})

one_hot = pd.get_dummies(df)
print(one_hot)
```

```
   item_TV  item_냉장고  item_믹서  item_선풍기  item_전자레인지  item_컴퓨터
0        1         0        0         0          0         0
1        0         1        0         0          0         0
2        0         0        0         0          1         0
3        0         0        0         0          0         1
4        0         0        0         1          0         0
5        0         0        0         1          0         0
6        0         0        1         0          0         0
7        0         0        1         0          0         0
```

- 별도의 숫자 인코딩 없이 문자열 그대로 원-핫 인코딩 가능
- 코드가 간결하고 실무에서 자주 사용됨

2) 피처 스케일링과 정규화 (Feature Scaling & Normalization)

머신러닝에서 서로 다른 변수의 값 범위를 일정한 수준으로 맞추는 작업을 **피처 스케일링 (feature scaling)**

대표적인 방식은 표준화(Standardization)와 정규화(Normalization)

1️⃣ 표준화 (Standardization)

- 각 피처의 값을 **평균=0, 분산=1**인 정규 분포로 변환

$$x_i\_new = \frac{x_i - mean(x)}{stdev(x)}$$

- **적용 이유**
  - 선형 회귀(Linear Regression), 로지스틱 회귀(Logistic Regression), SVM(RBF 커널), 신경망(ANN) 등은 데이터가 가우시안 분포를 따른다고 가정하기 때문에 성능 향상에 매우 중요

2️⃣ 정규화 (Normalization)

- 각 피처 값을 **최소 0 ~ 최대 1** 범위로 변환

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

- 서로 다른 단위(예: 거리 0~100km, 금액 0~10억 원)를 같은 스케일로 맞출 때 사용

3️⃣ 벡터 정규화 (Vector Normalization)

- 사이킷런 `Normalizer` 는 선형대수 개념의 정규화 적용
- 개별 벡터의 크기를 1로 맞추기 위해, 벡터의 L2 노름(길이)으로 나눔

$$x_i\_new = \frac{x_i}{\sqrt{x_i^2 + y_i^2 + z_i^2}}$$

- 주로 텍스트 벡터화(코사인 유사도 계산 등)에서 사용

(1) StandardScaler

- 표준화를 쉽게 지원하기 위한 클래스
- RBF 커널을 이용하는 서포트 벡터 머신(Support Vector Machine)이나 선형 회귀(Linear Regression), 로지스틱 회귀(Logistic Regression) 데이터의 가우시안 분포 가정 → 사전에 표준화를 적용하는 것은 **예측 성능 향상**에 중요한 요소

```
from sklearn.datasets import load_iris
import pandas as pd
#붓꽃 데이터 세트 로딩 후 DataFrame으로 변환

iris = load_iris()
iris_data = iris.data
iris_df = pd.DataFrame(data = iris_data, columns=iris.feature_names)

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

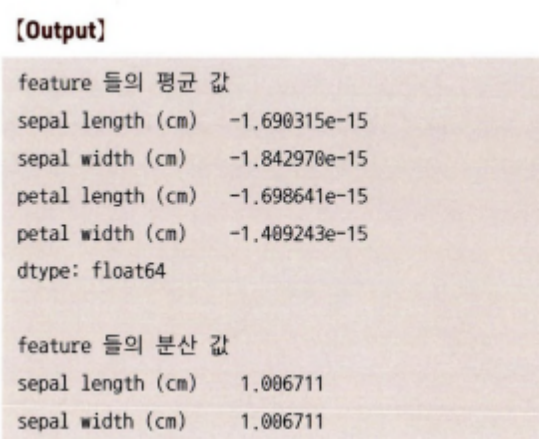
출력

```
feature 평균값
sepal length (cm)    5.843333
sepal width (cm)     3.057333
petal length (cm)    3.758000
petal width (cm)     1.199333

feature 분산값
sepal length (cm)    0.685694
sepal width (cm)     0.189979
petal length (cm)    3.116278
petal width (cm)     0.581006
```

- StandardScaler 객체 생성 후 fit() & transform() 활용하면 간단하게 변환

```
from sklearn.preprocessing import StandardScaler

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

#Numpy로 반환돼 이를 DataFrame으로 변환
iris_df_scaled = pd.DataFrame(data=iris_scaled, columns=iris.feature_nam
es)
print('feature 평균 값')
print(iris_df_scaled.mean())
print('\nfeature 분산값')
print(iris_df_scaled.var())
```

출력



👉 모든 피처가 평균 0, 분산 1로 변환됨을 확인할 수 있음.

---

(2) MinMaxScaler

`MinMaxScaler`는 데이터 값을 **0~1 범위**(또는 지정한 범위)로 변환

- 음수 값이 있으면 **1 ~ 1 범위**로 변환할 수도 있음 (`feature_range` 파라미터 조정).
- 데이터의 분포가 **가우시안 분포가 아닐 때**도 안정적으로 적용 가능.
- 분포 모양은 유지하면서 값의 범위만 동일하게 맞춤.

```
from sklearn.preprocessing import MinMaxScaler

#MinMaxScaler객체  생성
scaler = MinMaxScaler()
#MinMaxScaler로  데이터  세트  변환. fit()과  transformO 호출
scaler.fit(iris_df)
iris_scaled = scaler.transform(iris_df)

# 변환된 데이터 세트가 NumPy ndarray로 반환돼 이를 DataFrame으로 변환
iris_df_scaled = pd.DataFrame(data = iris_scaled, columns=iris.feature_nam
es)
print("feature들의 최솟값\n", iris_df_scaled.min())
print("\nfeature들의 최댓값\n", 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
```

➡️ 모든 피처가 **0~1 범위 값으로 변환**됨 ✅

⚖️ **정리**

- **표준화(Standardization)** : 평균 0, 분산 1로 맞춤 → SVM, 회귀 모델 등에 중요
- **정규화(Normalization)** : 데이터 범위를 0~1로 변환 → 단위 다른 피처 비교용
- **벡터 정규화(Normalizer)** : 벡터 크기를 1로 맞춤 → 텍스트/추천 시스템 등

(3) 학습 데이터 vs 테스트 데이터 스케일링 유의점

- **fit()** : 스케일링 기준 정보(최솟값/최댓값, 평균/표준편차 등)를 학습
- **transform()** : 학습된 기준 정보를 이용해 데이터 변환
- **fit_transform()** : `fit()` + `transform()`을 한 번에 수행

*유의점

- 반드시 **학습 데이터로만 `fit()`을 수행**해야 함
- 학습 데이터로 `fit()` + `transform()` → 테스트 데이터는 `transform()`만 사용
- **학습 데이터의 스케일링 기준 정보를 그대로 테스트 데이터 적용**
- 테스트 데이터에 따로 `fit()`을 적용하면 **스케일 기준이 달라져 성능이 왜곡**될 수 있음

```
from sklearn.preprocessing import MinMaxScaler
import numpy as np

train_array = np.arange(0, 11).reshape(-1, 1)   # 0~10
test_array  = np.arange(0, 6).reshape(-1, 1)    # 0~5

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))
```

```
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. ]
```

➡️ 학습 데이터는 `1 → 0.1`, 테스트 데이터는 `1 → 0.2`로 달라져 스케일 불일치 발생 ⚠️

올바른 사용법

```
scaler.fit(test_array)
test_scaled = scaler.transform(test_array)
```

```
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 데이터와 테스트 데이터가 **같은 기준(0~10 범위)**으로 변환됨 ✅

요약

1. 가능하다면 **전체 데이터에 스케일링 적용 후** 학습/테스트 데이터로 분리
2. 그렇지 못한다면, 반드시
    - 학습 데이터 → `fit_transform()`
    - 테스트 데이터 → `transform()` (`fit()` / `fit_transform()` 사용 금지)

👉 이 원칙은 PCA 같은 **차원 축소**, 텍스트 **피처 벡터화**에서도 동일하게 적용됨.

3장 머신러닝 모델 평가 (Evaluation)

머신러닝은 크게 **데이터 가공/변환 → 모델 학습/예측 → 성능 평가**의 프로세스로 구성

1. 성능 평가 지표(Evaluation Metric) 구분

- **회귀(Regression)**
  - 실제 값과 예측 값의 **오차 평균** 기반
  - 예:
    - MAE (Mean Absolute Error, 평균 절대 오차)
    - MSE (Mean Squared Error, 평균 제곱 오차)
    - RMSE (Root Mean Squared Error, 평균 제곱근 오차)
  - 비교적 단순
- **분류(Classification)**
  - 실제 결과와 예측 결과가 얼마나 정확하고 오류가 적은지 측정
  - **단순 정확도(Accuracy)만 보면 잘못된 평가 위험**
  - 특히 이진 분류(Binary Classification: 0/1, 긍정/부정)에서 다양한 지표 필요

2. 분류 성능 평가 지표

- **정확도 (Accuracy)**

  전체 데이터 중 예측이 맞은 비율
- **오차 행렬 (Confusion Matrix)**

  예측 결과를 **TP TN FP FN**으로 분류한 행렬
- **정밀도 (Precision)**

  "양성이라고 예측한 것" 중 실제로 맞은 비율

- **재현율 (Recall, Sensitivity, TPR)**

  "실제 양성" 중에서 얼마나 잘 맞췄는지

- **F1 스코어 (F1-Score)**

  정밀도와 재현율의 조화 평균

- **ROC AUC (Receiver Operating Characteristic - Area Under Curve)**
  - 분류 성능을 다양한 임계값(threshold)에서 평가
  - AUC 값이 1에 가까울수록 성능이 우수

이진 분류 vs 멀티 분류

- **이진 분류** (Binary Classification)
  - 두 개의 클래스만 존재 (예: 스팸메일=1, 정상메일=0)
  - 위의 지표(Accuracy, Precision, Recall, F1, ROC AUC)가 모두 중요
- **멀티 분류** (Multi-class Classification)
  - 여러 클래스 존재 (예: 숫자 0~9 이미지 분류)
  - 지표는 동일하게 적용 가능하지만, **특히 이진 분류에서 더 강조**됨

📌 정확도(Accuracy) 평가 지표의 한계

1️⃣ 정확도의 정의

Accuracy=예측 결과가 동일한 데이터 건수전체 예측 데이터 건수

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

- 직관적으로 모델 성능을 판단할 수 있는 가장 기본적인 지표

- 하지만 **데이터 분포에 따라 성능이 왜곡될 수 있음**

2️⃣ 단순 분류기의 함정 (Titanic 예제)

- 타이타닉 데이터에서 "여성 → 생존, 남성 → 사망"으로만 단순 예측해도 약 **78% 정확도** 달성 가능
- 즉, 별다른 학습 없이도 데이터 불균형/특징 편향 때문에 높은 수치가 나올 수 있음

```
BaseEstimator
```

를 상속

- `BaseEstimator`는 **scikit-learn** 라이브러리에서 제공하는 **모든 추정기(Estimator: 모델, 변환기 등)의 공통 기반 클래스,** 사용자가 직접 만든 모델이나 변환기가 scikit-learn과 호환되려면 이 부모 클래스를 상속 받는 경우가 많음

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

titanic_df = pd.read_csv('Titanic-Dataset.csv')
y_titanic_df = titanic_df['Survived']
x_titanic_df = titanic_df.drop('Survived', axis = 1)
x_titanic_df = transform_features(x_titanic_df)
x_train, x_test, y_train, y_test = train_test_split(x_titanic_df, y_titanic_df, test_
size=0.2, random_state = 0)


myclf = MyDummyClassifier()
myclf.fit(x_train, y_train)

mypredictions = myclf.predict(x_test)
print("정확도 {0:.4f}".format(accuracy_score(y_test, mypredictions)))
```

출력 예시

```
Dummy Classifier의 정확도는: 0.7877
```

➡️ 단순 규칙 기반 예측임에도 높은 정확도 → **정확도 단독 지표의 위험성**

📌 왜 `self`를 꼭 써야 할까?

: 객체 고유의 속성(attribute) 을 만들기 위해 반드시 `self.`를 붙인다. 안 그러면 호출할 방법이 없음.

ex. self를 안 쓰면...

```
class MyClass:
    def __init__(self, value):
        value = value   # self 없음 → 그냥 지역변수(local variable)일 뿐

a = MyClass(10)
b = MyClass(20)

print(a.value)  # AttributeError (속성이 없음)
```

`value = value`는 그냥 **함수 안에서만 잠깐 쓰였다가 사라짐,** 객체(`a`, `b`)에는 아무 속성도 저장되지 않음.

3️⃣ 불균형 데이터에서의 왜곡 (MNIST 변형 예제)

- MNIST(0~9) 데이터에서 "7 → 1(True), 나머지 → 0(False)"로 변환 → **10%만 1인 불균형 데이터**
- `MyFakeClassifier`: 모든 입력을 **0으로만 예측**
- `load_digits()`는 **손글씨 숫자(0~9)** 데이터셋

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.base import BaseEstimator
from sklearn.metrics import accuracy_score
import numpy as np
import pandas as pd

class MyFakeClassifier(BaseEstimator):
  def fit(self, X, y):
    pass
```

```
  def predict(self, X):
    return np.zeros((len(X), 1), dtype = bool)

digits = load_digits()

y = (digits.target ==7).astype(int)
x_train, x_test, y_train, y_test = train_test_split(digits.data, y, random_state=1
1)
#불균형한 레이블 데이터 분포도 확인

print("레이블 테스트 세트 크기:", y_test.shape)
print('테스트 세트 레이블의 0과 1 분포도:')
print(pd.Series(y_test).value_counts())

fakeclf = MyFakeClassifier()
fakeclf.fit(x_train, y_train)
fakepred = fakeclf.predict(x_test)

print('모든 예측을 0으로 했을 때 정확도는:{:.3f}'.format(accuracy_score(y_test,
fakepred)))
```

출력 예시

```
레이블 테스트 세트 크기: (450,)
테스트 세트 레이블의 0과 1 분포도:
0    405
1     45
Name: count, dtype: int64
모든 예측을 0으로 했을 때 정확도는:0.900
```

➡️ 실제로는 "양성(1)"을 전혀 못 맞추는데도 높은 정확도 수치 전혀 못 맞추는 쓸모없는 모델

🔑 왜 클래스로 만드나? → **scikit-learn 생태계와 호환**

- scikit-learn에서는 모든 모델이 `fit()`, `predict()` 인터페이스를 가져야 `cross_val_score`, `GridSearchCV`, `Pipeline` 같은 scikit-learn 도구와 함께 쓸 수 있음

4️⃣ 정리

- **정확도는 데이터 불균형 상황에서 왜곡 발생**
- 단순 규칙 기반, 무조건 다수를 예측하는 모델도 높은 수치를 가질 수 있음
- 따라서 분류 문제에서는 반드시 **정밀도(Precision), 재현율(Recall), F1, ROC AUC** 등과 함께 종합적으로 평가해야 함

📌 오차 행렬(Confusion Matrix)

1️⃣ 정의

- 이진 분류에서 **예측 결과와 실제 값의 관계**를 4분면 행렬로 나타낸 것
- 모델이 **어떤 유형의 오류를 범하고 있는지** 함께 보여줌
- 사이킷런에서는 `confusion_matrix()` API 제공

2️⃣ 오차 행렬 구조

실제 / 예측	Negative (0)	Positive (1)
Negative (0)	TN (True Negative)	FP (False Positive)
Positive (1)	FN (False Negative)	TP (True Positive)

- **TN** : 0을 0으로 맞춤
- **FP** : 실제는 0인데 1로 잘못 예측
- **FN** : 실제는 1인데 0으로 잘못 예측
- **TP** : 1을 1로 맞춤

3️⃣ 예제 (MyFakeClassifier)

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_test, fakepred)
```

출력

```
array([[405,   0],
       [ 45,   0]], dtype=int64)
```

- TN = 405
- FP = 0
- FN = 45
- TP = 0

➡️ 모든 예측을 0으로 했으므로, 실제 0인 405건만 맞고 실제 1인 45건은 모두 놓침

4️⃣ 정확도(Accuracy)와 오차 행렬의 관계

$$Accuracy = \frac{TN + TP}{TN + FP + FN + TP}$$

- 즉, 예측 결과가 맞은 건수 / 전체 데이터 수
- 불균형 데이터에서는 TN이 압도적으로 크면 **정확도가 왜곡**됨

5️⃣ 불균형 데이터에서의 문제

- 현실 문제에서는 Positive(1)가 매우 적은 경우가 많음
  - 예: 사기 거래 탐지(사기=1), 암 진단(암=1)
- 이때 모델은 Negative로 치우친 예측을 하게 되고, TN 위주로 정확도가 높게 계산됨
- 하지만 정작 중요한 Positive(사기, 암)를 잘 잡지 못함 → **정확도만으로는 모델 성능을 신뢰할 수 없음**

6️⃣ 정리

- 오차 행렬은 분류 모델 성능을 세밀하게 분석하는 기본 도구
- 정확도만 볼 경우 **불균형 데이터 문제**에서 잘못된 판단을 내릴 수 있음
- 따라서 **정밀도(Precision), 재현율(Recall), F1-score, ROC-AUC** 등을 함께 활용해야 함

📌 정밀도(Precision)와 재현율(Recall)

1️⃣ 정의와 공식

- **정밀도 (Precision)**

$$precision = \frac{TP}{TP + FP}$$

  → Positive라고 예측한 것 중 실제 Positive인 비율 (**양성 예측도**)

- **재현율 (Recall, Sensitivity, TPR)**

$$Sensitivity = \frac{TP}{TP + FN}$$

  → 실제 Positive 중에서 얼마나 맞췄는지 (**민감도**)

2️⃣ 특징

- 두 지표 모두 **TP를 높이는 것**이 핵심
- **정밀도**: FP(거짓 양성) 줄이는 데 초점
- **재현율**: FN(거짓 음성) 줄이는 데 초점
- 서로 보완 관계 → 둘 다 높이는 것이 이상적

4️⃣ 예제 코드

사이킷런의 `precision_score()`, `recall_score()` 활용

```
from sklearn.metrics import accuracy_score, precision_score, recall_score,
confusion_matrix

def get_clf_eval(y_test, pred):
  confusion = confusion_matrix(y_test, pred)
  accuracy = accuracy_score(y_test, pred)
  precision = precision_score(y_test, pred)
  recall = recall_score(y_test, pred)
  print('오차행렬')
  print(confusion)
```

```
  print('정확도: {0:.4f}, 정밀도: {1:.4f}, 재현율: {2:.4f}'.format(accuracy, precisi
on, recall))
```

5️⃣ 타이타닉 로지스틱 회귀 예제

- 로지스틱 회귀는 **특정 사건이 일어날 확률을 예측하는 모델**
- `solver` → **모델의 파라미터(w, b)를 어떻게 최적화(학습)할지 결정하는 알고리즘**

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

#원본 데이터 재로딩
titanic_df = pd.read_csv('Titanic-Dataset.csv')
y_titanic_df = titanic_df['Survived']
x_titanic_df = titanic_df.drop('Survived', axis=1)
x_titanic_df = transform_features(x_titanic_df)

x_train, x_test, y_train, y_test = train_test_split(x_titanic_df, y_titanic_df, test_
size=0.2, random_state=11)
lr_clf = LogisticRegression(solver='liblinear')
lr_clf.fit(x_train, y_train)
pred = lr_clf.predict(x_test)
get_clf_eval(y_test, pred)
```

출력

```
오차 행렬
[[108  10]
 [ 14  47]]
정확도: 0.8659, 정밀도: 0.8246, 재현율: 0.7705
```

6️⃣ 정밀도/ 재현율 트레이드오프

- threshold를 낮추면 → **재현율 ↑, 정밀도 ↓**
- threshold를 높이면 → **정밀도 ↑, 재현율 ↓**

7️⃣ 확률 기반 API

- `predict_proba()` → 각 클래스(0/1) 예측 확률 반환
- 입력 테스트 데이터 세트의 표본 개수가 100개 & 예측 클래스 값 유형이 2개(이진 분류) → 반환 값은 100 x 2 ndarray
- 이거 기반으로 predict(_) 메서드 만든 것

8️⃣ Binarizer를 이용한 threshold 조정

```
from sklearn.preprocessing import Binarizer

# Binarizer의 threshold 설정값. 분류 결정 임곗값임.
custom_threshold = 0.5
# predict_proba( ) 반환값의 두 번째 칼럼, 즉 Positive 클래스 칼럼 하나만 추출해 B
inarizer를 적용
pred_proba_1 = pred_proba[:,1].reshape(-1,1)

binarizer = Binarizer(threshold=custom_threshold).fit(pred_proba_1)
custom_predict = binarizer.transform(pred_proba_1)

get_clf_eval(y_test, custom_predict)
```

출력 (threshold=0.5)

```
오차 행렬
[[108  10]
 [ 14  47]]
정확도: 0.8659, 정밀도: 0.8246, 재현율: 0.7705
```

출력 (threshold=0.4)

```
오차 행렬
[[97 21]
 [11 50]]
정확도: 0.8212, 정밀도: 0.7042, 재현율: 0.8197
```

➡️ threshold ↓ → Positive를 더 많이 잡아내서 재현율 ↑, 대신 오탐이 늘어 정밀도 ↓

정리:

- **정밀도**: 예측 양성 중 얼마나 맞췄는가
- **재현율**: 실제 양성 중 얼마나 맞췄는가
- **트레이드오프** 존재 → threshold 조정으로 균형을 맞춤

📌 정밀도–재현율과 임곗값(Threshold)

1️⃣ 임곗값 변화에 따른 효과(예시)

- 임곗값 0.50 → 0.40
  - **TP**: 47 → 50 (↑)
  - **FN**: 14 → 11 (↓) → **재현율** 0.770 → **0.820** (↑)
  - **FP**: 10 → 21 (↑) → **정밀도** 0.825 → **0.704** (↓)
  - **정확도**: 0.866 → **0.821** (↓)

2️⃣ 임곗값별 평가 지표 계산 함수

임곗값 조금씩 증가시키기

```
# 테스트를 수행할 모든 임곗값을 리스트 객체로 저장.
thresholds = [0.4, 0.45, 0.50, 0.55, 0.60]

def get_eval_by_threshold(y_test, pred_proba_c1, threshold):
  for custom_threshold in threshold:
    binarizer = Binarizer(threshold=custom_threshold).fit(pred_proba_c1)
    custom_predict = binarizer.transform(pred_proba_c1)
    print("임계값: ", custom_threshold)
    get_clf_eval(y_test, custom_predict)

get_eval_by_threshold(y_test, pred_proba[:,1].reshape(-1,1), thresholds)
```

3️⃣ precision_recall_curve로 임곗값/정밀도/재현율 확인

.

```
import numpy as np
from sklearn.metrics import precision_recall_curve

# 레이블 1(Positive)에 대한 예측확률
# pred_proba_class1 = classifier.predict_proba(X_test)[:, 1]

precisions, recalls, thresholds = precision_recall_curve(y_test, pred_proba_
class1)
print("반환된 임곗값 배열 shape:", thresholds.shape)

# 임곗값이 많으므로 15 step 간격으로 샘플 확인
idx = np.arange(0, thresholds.shape[0], 15)
print("샘플 임곗값:", np.round(thresholds[idx], 2))
print("샘플 정밀도:", np.round(precisions[idx], 3))
print("샘플 재현율:", np.round(recalls[idx], 3))
```

특징:

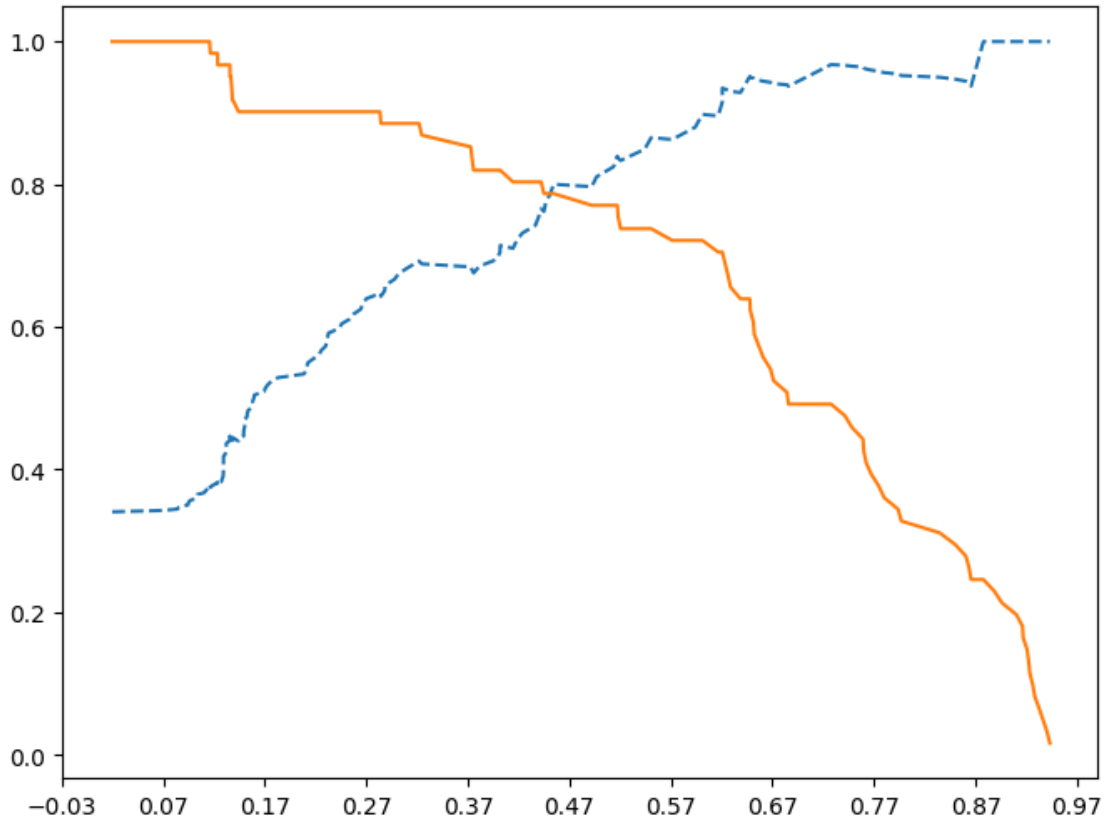
- 임곗값 ↑ → **정밀도 ↑**, **재현율 ↓**
- 임곗값 ↓ → **정밀도 ↓**, **재현율 ↑**

4️⃣ 정밀도–재현율 곡선 시각화

- `precision_recall_curve()`는 임곗값(threshold)을 변화시켰을 때 정밀도(precision)와 재현율(recall)이 어떻게 바뀌는지를 보여주는 함수

```
import matplotlib.ticker as ticker
import matplotlib.pyplot as plt
%matplotlib inline

# precision-recall curve plotting 함수
def precision_recall_curve_plot(y_test, pred_proba_c1):
    # threshold ndarray와 이 threshold에 따른 정밀도, 재현율 ndarray 추출
    precisions, recalls, thresholds = precision_recall_curve(y_test, pred_prob
a_c1)
```

```
    # Plot 그리기
    plt.figure(figsize=(8, 6))
    threshold_boundary = thresholds.shape[0]

    # 정밀도는 점선으로 표시
    plt.plot(thresholds, precisions[0:threshold_boundary], linestyle='--', label
='precision')
    plt.plot(thresholds, recalls[0:threshold_boundary], label='recall')

    # threshold 값 X축의 눈금을 0.1 단위로 변경
    start, end = plt.xlim()
    plt.xticks(np.round(np.arange(start, end, 0.1), 2))
    plt.show()

precision_recall_curve_plot(y_test, lr_clf.predict_proba(x_test)[:,1])
```



5 주의사항

- **정밀도 100%**: 아주 확실한 사례만 Positive로 판정(예측 Positive 수가 극단적으로 적어짐)
- **재현율 100%**: 모든 샘플을 Positive로 판정(오탐 대량 발생)
- 한쪽 지표만 극단적으로 높이는 것은 의미 없음 → **업무 특성에 맞춰 임곗값을 조정하며 두 지표의 균형을 모색**

📌 F1 스코어 (F1 Score)

1️⃣ 정의

- 정밀도와 재현율의 **조화 평균**

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

- 정밀도나 재현율이 한쪽으로 치우치면 F1 값이 낮아짐
  - 예) 모델 A: Precision=0.9, Recall=0.1 → **F1=0.18**
  - 예) 모델 B: Precision=0.5, Recall=0.5 → **F1=0.50**

2️⃣ 사이킷런 계산 예시

```
from sklearn.metrics import f1_score

# y_test: 실제 레이블, pred: 예측 레이블
f1 = f1_score(y_test, pred)
print("F1 스코어: {0:.4f}".format(f1))
```

```
F1 스코어: 0.7966
```

3️⃣ 평가 함수에 F1 추가

- 임곗값 0.4 ~ 0.6별 정확도, 정밀도, 재현율, F1 스코어

```
def get_clf_eval(y_test, pred):
    confusion = confusion_matrix(y_test, pred)
    accuracy = accuracy_score(y_test, pred)
    precision = precision_score(y_test, pred)
    recall = recall_score(y_test, pred)
    f1  = f1_score(y_test, pred)
    print("오차 행렬\n", confusion)
    print("정확도: {0:.4f}, 정밀도: {1:.4f}, 재현율: {2:.4f}, F1: {3:.4f}"
          .format(accuracy, precision, recall, f1))

thresholds = [0.4, 0.45, 0.50, 0.55, 0.60]
pred_proba = lr_clf.predict_proba(x_test)
get_eval_by_threshold(y_test, pred_proba[:, 1].reshape(-1,1), thresholds)
```

4️⃣ ROC 곡선과 AUC

1. ROC 곡선이란?

- **ROC (Receiver Operating Characteristic Curve)** = 이진 분류기의 성능 평가 지표

→ 임곗값 1~0에 따른 **FPR** 와 FPR 값의 변화에 따른 **TPR 값** 구하는 것

- **X축**: FPR (False Positive Rate) = FP / (FP + TN) = 1 - TNR = 1- 특이성
- **Y축**: TPR (True Positive Rate, 재현율/민감도) = TP / (TP + FN)
- 임곗값을 1 → 0으로 변화시키며 FPR vs TPR의 변화를 곡선으로 그림

*민감도(TPR): 실제값 Positive가 정확히 예측되어야 하는 수준

*특이성(TNR)은 실제값 Negative가 정확히 예측되어야 하는 수준

---

3. 임곗값 변화와 FPR/TPR

- **Threshold = 1** → Positive 예측 거의 없음 → FP=0 → FPR=0
- **Threshold = 0** → 모두 Positive 예측 → TN=0 → FPR=1

---

4. ROC & AUC

- **ROC 곡선**: FPR이 변할 때 TPR이 어떻게 변하는지 시각화
- **AUC (Area Under Curve)**: ROC 곡선 아래 면적
  - 1에 가까울수록 좋은 성능
  - 0.5 = 랜덤 분류 수준
  - 0.7~0.8: 보통, 0.8~0.9: 우수, 0.9 이상: 매우 우수

---

5. 사이킷런 API

- `roc_curve()` → FPR, TPR, Threshold 반환
- `roc_auc_score()` → AUC 값 반환

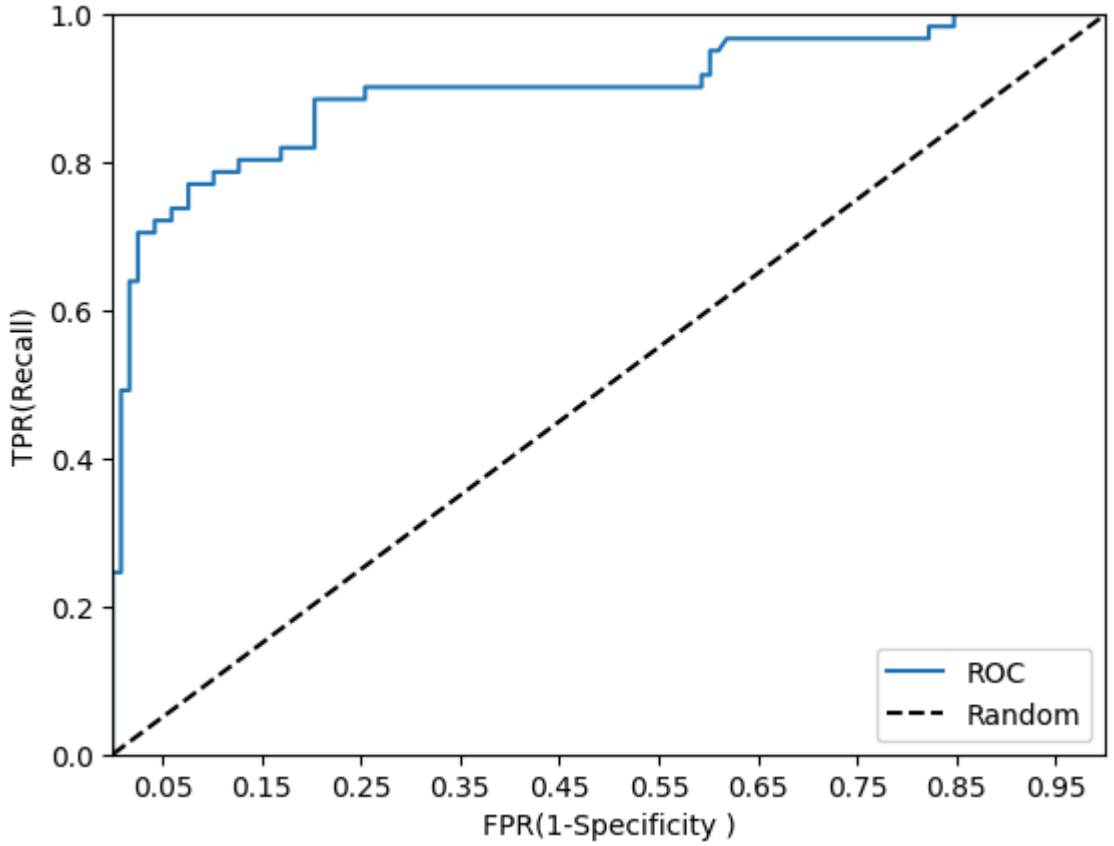
```
from sklearn.metrics import roc_curve

#레이블 값이 1일 때 예측 확률을 추출
pred_proba_class1 = lr_clf.predict_proba(x_test)[:,1]

fprs, tprs, thresholds = roc_curve(y_test, pred_proba_class1)

thr_index = np.arange(1, thresholds.shape[0], 5)
print(thr_index)
print(np.round(thresholds[thr_index], 2))
print('FPR: ', np.round(fprs[thr_index], 3))
print('TPR: ', np.round(tprs[thr_index], 3))
```

```
임계값:  0.4
오차 행렬
 [[97 21]
 [11 50]]
정확도: 0.8212, 정밀도: 0.7042, 재현율: 0.8197, F1: 0.7576
임계값:  0.45
오차 행렬
 [[105  13]
 [ 13  48]]
정확도: 0.8547, 정밀도: 0.7869, 재현율: 0.7869, F1: 0.7869
```

```
임계값:  0.5
오차 행렬
 [[108  10]
 [ 14  47]]
정확도: 0.8659, 정밀도: 0.8246, 재현율: 0.7705, F1: 0.7966
임계값:  0.55
오차 행렬
 [[111   7]
 [ 16  45]]
정확도: 0.8715, 정밀도: 0.8654, 재현율: 0.7377, F1: 0.7965
임계값:  0.6
오차 행렬
 [[113   5]
 [ 17  44]]
정확도: 0.8771, 정밀도: 0.8980, 재현율: 0.7213, F1: 0.8000
```

시각화

```
def roc_curve_plot(y_test, pred_proba_class1):
  fprs, tprs, thresholds = roc_curve(y_test, pred_proba_class1)
  plt.plot(fprs, tprs, label='ROC')
  plt.plot([0,1], [0,1], 'k--', label = 'Random')

  start, end = plt.xlim() #x값의 최대 최소 튜플로 반환
  plt.xticks(np.round(np.arange(start, end, 0.1), 2))
  plt.xlim(0, 1); plt.ylim(0, 1)
  plt.xlabel('FPR(1-Specificity )'); plt.ylabel('TPR(Recall)')
  plt.legend()
roc_curve_plot(y_test, pred_proba[:, 1])
```



6. get_clf_eval() 개선

ROC AUC까지 출력하려면 **예측 확률**을 추가 인자로 받아야 함:

```
def get_clf_eval(y_test, pred=None, pred_proba=None):
    confusion = confusion_matrix(y_test, pred)
    accuracy = accuracy_score(y_test, pred)
    precision = precision_score(y_test, pred)
    recall = recall_score(y_test, pred)
    f1 = f1_score(y_test, pred)

    roc_auc = roc_auc_score(y_test, pred_proba)

    print("오차 행렬\n", confusion)
    print("정확도: {0:.4f}, 정밀도: {1:.4f}, 재현율: {2:.4f}, "
          "F1: {3:.4f}, AUC: {4:.4f}"
          .format(accuracy, precision, recall, f1, roc_auc))
```

✅ 핵심

- ROC = FPR vs TPR 곡선
- TPR = 재현율, TNR = 특이도, FPR = 1 - TNR
- AUC = ROC 곡선 아래 면적 (클수록 좋음)
- 사이킷런 → `roc_curve()`, `roc_auc_score()`

```
import numpy as np

def get_eval_by_threshold(y_test, pred_proba_class1, thresholds):
    """
    y_test: 실제 이진 레이블(0/1)
    pred_proba_class1: 레이블 1의 예측확률 (shape: [n_samples])
    thresholds: 검사할 임곗값 리스트/ndarray
    """
    for thr in thresholds:
        pred = (pred_proba_class1 >= thr).astype(int)
        acc = accuracy_score(y_test, pred)
        pre = precision_score(y_test, pred, zero_division=0)
        rec = recall_score(y_test, pred, zero_division=0)
        f1  = f1_score(y_test, pred)
        print("임곗값: {0:.2f} → 정확도: {1:.4f}, 정밀도: {2:.4f}, 재현율: {3:.4f}, F1:
{4:.4f}"
              .format(thr, acc, pre, rec, f1))

# 사용 예시
# pred_proba = lr_clf.predict_proba(X_test)[:, 1]
# get_eval_by_threshold(y_test, pred_proba, np.arange(0.40, 0.61, 0.05))
```