Chap 3. 분류 (Classification)

Seolyoung Jeong, Ph.D.

경북대학교 IT 대학

Contents

3.1 MNIST

- MNIST : 미국에서 손으로 쓴 70,000개의 작은 숫자 이미지를 모은 데이터셋 (머신러닝 분야의 'Hello World')

- 목적 : 28x28 픽셀의 필기 숫자 이미지를 어떤 숫자인지 판별

MNIST 데이터셋 다운로드

```
In [2]: from sklearn.datasets import fetch_mldata
        mnist = fetch_mldata('MNIST original')
        mnist
```

```
Out[2]: {'DESCR': 'mldata.org dataset: mnist-original',
         'COL_NAMES': ['label', 'data'],
         'target': array([0., 0., 0., ..., 9., 9., 9.]),
         'data': array([[0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0],
                ...,
                [0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0]], dtype=uint8)}
```

다운로드 오류
- **'mnist_original.mat' 파일 다운로드**
- **Jupyter Notebook의 '/scikit_learn_data/mldata/' 위치에 업로드**

- **MNIST 딕셔너리 구조**
  - DESCR : 데이터셋에 대한 설명
  - target : 레이블 배열
  - data : 2차원 배열 [샘플][특성]
    - 샘플 : image
    - 특성 : 0~255 (pixel)

MNIST 배열 확인

◆ 배열 확인

```
In [3]: X, y = mnist["data"], mnist["target"]
        X.shape
Out[3]: (70000, 784)

In [4]: y.shape
Out[4]: (70000,)

In [5]: 28*28
Out[5]: 784
```

- data : 이미지 70,000개, 각 이미지에는 784개 특성 (28x28 픽셀)
  - 특성 : 0(흰색) ~ 255(검은색)까지의 픽셀 강도
- target : label 70,000개
  - 각 이미지별 0~9에 해당하는 정답 숫자

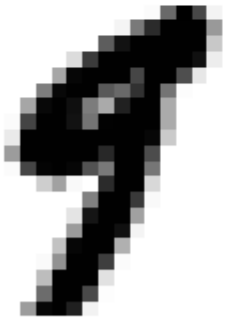
MNIST 이미지 확인

- 샘플의 특성 벡터 추출 → 28x28 배열로 크기 변경

In [8]:
```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt

some_digit = X[36000]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap = matplotlib.cm.binary,
           interpolation="nearest")
plt.axis("off")

plt.show()
```



In [9]:
```
y[36000]
```

Out[9]: 9

MNIST 숫자 이미지

테스트 세트 분리

- **MNIST 데이터셋은 이미 분리되어 있음**
  - 훈련 세트 : 앞쪽 60,000개 이미지
  - 테스트 세트 : 뒤쪽 10,000개 이미지

```
In [12]: X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

- **훈련 세트를 섞어서 모든 교차 검증 폴드가 비슷해지도록... (하나의 폴드라도 특정 숫자가 누락되면 안됨)**

```
In [13]: import numpy as np

         shuffle_index = np.random.permutation(60000)
         X_train, y_train = X_train[shuffle_index], y_train[shuffle_index]
```

  - 경우에 따라, 섞는 것이 좋지 않을 수도 있음
  - → 시계열 데이터를 다루는 경우 (예: 주식가격, 날씨 예보)

3.2 이진 분류기 훈련

- **하나의 숫자(예: 숫자 5)만 식별하는 예**
  - 5-감지기 : '5'와 '5 아님' 두 개의 클래스 구분 → 이진 분류기

In [14]:
```
y_train_5 = (y_train == 5)
y_test_5 = (y_test == 5)
```

  - 5는 True, 다른 숫자는 모두 False
- **분류 모델 선택 후 훈련**
  - 모델1) 확률적 경사 하강법 (Stochastic Gradient Descent : SGD) 분류기
    - 장점: 매우 큰 데이터셋을 효율적으로 처리
    - 한번에 하나씩 훈련 샘플을 독립적으로 처리 (온라인 학습에 적합)
    - 확률적 : 무작위성 사용

In [15]:
```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(max_iter=5, random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

    - 모델을 사용하여 숫자 5 이미지 감지

In [16]:
```
sgd_clf.predict([some_digit])
```

Out[16]:
```
array([False])
```

3.3 성능 측정

- **회귀모델 평가보다 분류기 평가에 사용할 수 있는 성능 지표가 더 많음!**

- **3.3.1 교차 검증을 사용한 정확도 측정**
  - 훈련 세트를 3개의 폴드로 나누고, 각 폴드에 대해 예측을 만든 후 평가하기 위해 나머지 폴드로 훈련시킨 모델 사용

```
In [17]: from sklearn.model_selection import cross_val_score
         cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
Out[17]: array([0.9605 , 0.95595, 0.95375])
```

  - 모든 교차 검증 폴드에 대해 정확도 95% 이상

3.3.1 교차 검증을 사용한 정확도 측정

- **모든 이미지를 '5 아님' 클래스로 분류 (임의의 더미 분류기)**

```
In [19]: from sklearn.base import BaseEstimator
         class Never5Classifier(BaseEstimator):
             def fit(self, X, y=None):
                 pass
             def predict(self, X):
                 return np.zeros((len(X), 1), dtype=bool)
```

```
In [20]: never_5_clf = Never5Classifier()
         cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

```
Out[20]: array([0.909  , 0.90745, 0.9125 ])
```

  - 정확도 : 90% 이상
  - 이미지의 10% 정도만 숫자 5이므로, 무조건 '5 아님'으로 예측하면 정확히 맞출 확률은 90%임

- **정확도는 분류기의 성능 측정 지표로 선호하지 않음 (특히, 불균형한 데이터셋을 다룰 때...)**

3.3.2 오차 행렬

- **분류기 성능 평가에 더 좋은 방법 : 오차 행렬 조사**
- **클래스 A의 샘플이 클래스 B로 분류된 횟수를 측정**
- **cross_val_predict() 함수**
  - 교차 검증 수행 후 각 테스트 폴드에서 얻은 예측값 반환 (score 아님)

```
from sklearn.model_selection import cross_val_predict

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

```
y_train_pred
```

```
array([False, False, False, ..., False, False, False])
```

  - confusion_matrix() 함수로 오차 행렬 생성
    - 타깃 클래스 : y_train_5
    - 예측 클래스 : y_train_pred

```
In [22]: from sklearn.metrics import confusion_matrix

         confusion_matrix(y_train_5, y_train_pred)

Out[22]: array([[52972,  1607],
                [  989,  4432]], dtype=int64)
```

→ '5 아님' 이미지

→ '5' 이미지

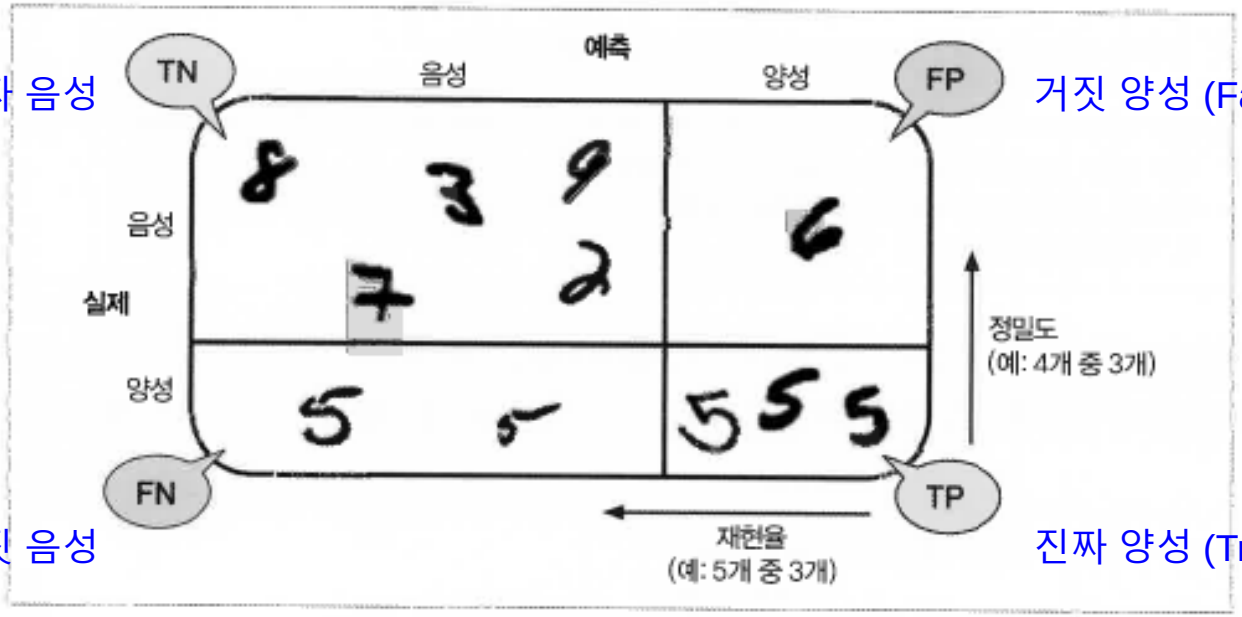
'5 아님' 분류 '5' 분류

3.3.2 오차 행렬

- 오차 행렬

(True Negative) 진짜 음성

그림 3-2 오차 행렬



거짓 양성 (False Positive)

(False Negative) 거짓 음성

진짜 양성 (True Positive)

- 완벽한 분류기라면 진짜 양성, 진짜 음성만 가지고 있음

```
In [26]: y_train_perfect_predictions = y_train_5

In [27]: confusion_matrix(y_train_5, y_train_perfect_predictions)

Out[27]: array([[54579,     0],
                [    0,  5421]], dtype=int64)
```

3.3.2 오차 행렬

- **정밀도 : 양성 예측의 정확도**
  - 분류기가 양성이라고 판단한 샘플(이미지) 중 실제 양성 샘플 수

$$\text{정밀도} = \frac{TP}{TP+FP}$$

TP : 진짜 양성 수, FP : 거짓 양성 수

- **재현율 : 분류기가 정확하게 감지한 양성 샘플의 비율**
  - 실제 양성인 샘플(이미지) 중에서 양성이라고 판단한 샘플 수
  - 민감도, 진짜 양성 비율

$$\text{재현율} = \frac{TP}{TP+FN}$$

FN : 거짓 음성 수

3.3.3 정밀도와 재현율

```
In [28]: from sklearn.metrics import precision_score, recall_score

         precision_score(y_train_5, y_train_pred)
Out[28]: 0.7338963404537175

In [30]: 4432 / (4432 + 1607)
Out[30]: 0.7338963404537175

In [31]: recall_score(y_train_5, y_train_pred)
Out[31]: 0.81756133554694 71

In [32]: 4432 / (4432 + 989)
Out[32]: 0.81756133554694 71
```

- 5로 판별된 이미지 중 73%만 정확, 전체 숫자 5에서 81%만 감지

◆ **F_1 점수 : 정밀도와 재현율의 조화평균**

정밀도와 재현율이 비슷한 분류기에서는 F_1 점수가 높다.

```
In [33]: from sklearn.metrics import f1_score
         f1_score(y_train_5, y_train_pred)
Out[33]: 0.7734729493891798
```

$$F_1 = \frac{2}{\frac{1}{\text{정밀도}} + \frac{1}{\text{재현율}}} = 2 \times \frac{\text{정밀도} \times \text{재현율}}{\text{정밀도} + \text{재현율}} = \frac{TP}{TP + \frac{FN + FP}{2}}$$

```
In [34]: 4432 / (4432 + (1607 + 989)/2)
Out[34]: 0.7734729493891798
```

3.3.3 정밀도와 재현율

- **상황에 따라 정밀도 or 재현율 중요도 다를 수 있음**

- **예) 어린 아이에게 안전한 동영상을 걸러내는 분류기**
  - (재현율 보다는) 높은 정밀도 선호
  - 나쁜 동영상 몇개 노출보다 좋은 동영상이 많이 제외되더라도 (낮은 재현율) 안전한 것들만 노출(높은 정밀도)

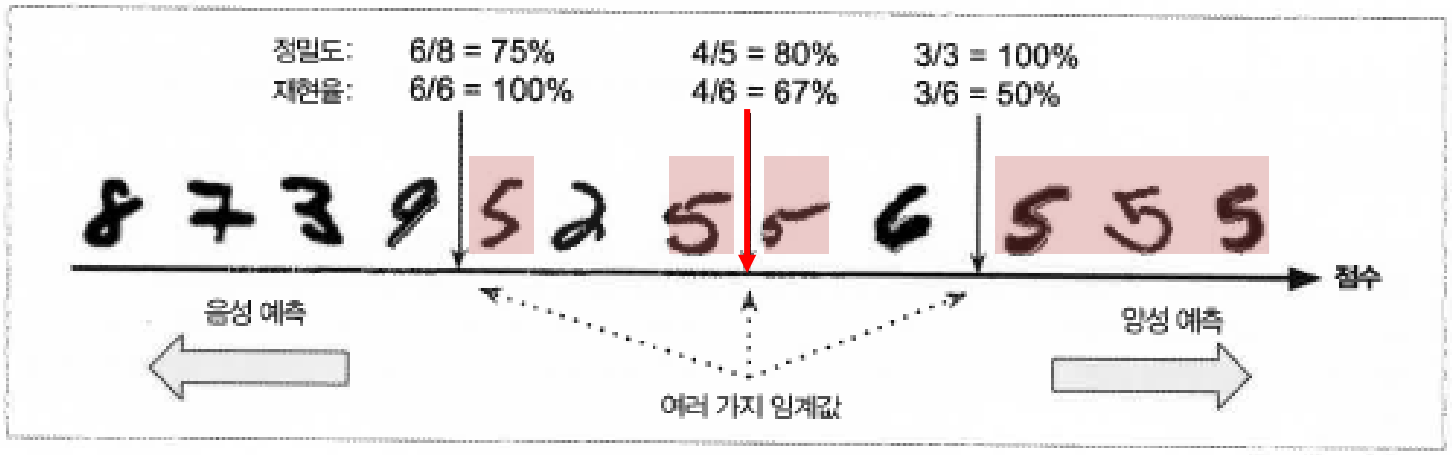
- **예) 감시 카메라를 통해 좀도둑을 잡아내는 분류기**
  - (정밀도 보다는) 높은 재현율 선호
  - 분류기의 재현율이 99%라면 정확도가 30%만 되더라도 괜찮음
  - 잘못된 알람 자주 발생하나, 거의 모든 좀도둑을 잡음

- **정밀도 / 재현율 트레이드오프 관계**
  - 정밀도를 올리면 재현율이 줄고, 그 반대도 마찬가지

3.3.4 정밀도/재현율 트레이드오프

- **현재 분류기(SGD)의 결정 점수**
  - 결정함수(decision function)로 각 샘플 점수 계산
  - 점수 > 임곗값 : 양성 클래스에 할당 (아니면 음성 클래스에 할당)

그림 3-3 결정 임곗값과 정밀도/재현율 트레이드오프



  - 결정 임곗값 : 가운데 화살표
    - 양성 예측 : 진짜 양성(숫자 5) 4개, 거짓 양성(숫자6) 1개
      - 정밀도 80% (5개 중 4개)
    - 실제 숫자 5는 6개, 분류기는 4개만 감지
      - 재현율 67% (6개 중 4개)
  - 임곗값을 높이면 : 정밀도 (3/3) = 100%, 재현율 (3/6) = 50%
  - 임곗값을 내리면 : 정밀도 (6/8) = 75%, 재현율 (6/6) = 100%

3.3.4 정밀도/재현율 트레이드오프

- **사이킷런에서 임곗값을 직접 지정할 수는 없지만, 예측에 사용한 점수 확인 가능**
  - decition_function()에 의한 샘플의 점수

```
In [35]: y_scores = sgd_clf.decision_function([some_digit])
         y_scores
Out[35]: array([-176288.21703149])
```

  - SGDClassifier 임곗값 = 0 → 결과 True

```
In [36]: threshold = 0
         y_some_digit_pred = (y_scores > threshold)
```

```
In [37]: y_some_digit_pred
```

  - SGDClassifier 임곗값 = 200000 → 결과 False

```
In [38]: threshold = 200000
         y_some_digit_pred = (y_scores > threshold)
         y_some_digit_pred
```

3.3.4 정밀도/재현율 트레이드오프

◆ 적절한 임곗값 결정

- 훈련 세트에 있는 모든 샘플의 점수 구함 (결정 점수)

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

- 이 점수를 이용하여 가능한 모든 임곗값에 대해 정밀도와 재현율 계산

```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

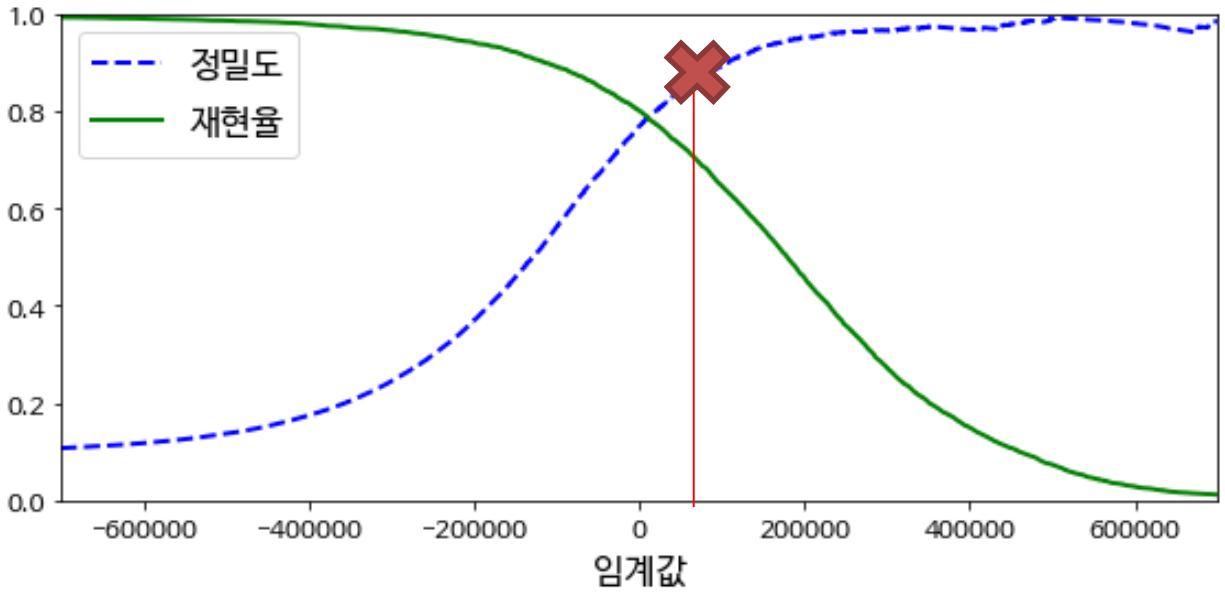
- matplotlib을 이용해 임곗값의 함수로 정밀도 재현율 그림

```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="정밀도", linewidth=2)
    plt.plot(thresholds, recalls[:-1], "g-", label="재현율", linewidth=2)
    plt.xlabel("임계값", fontsize=16)
    plt.legend(loc="upper left", fontsize=16)
    plt.ylim([0, 1])

plt.figure(figsize=(8, 4))
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.xlim([-700000, 700000])

plt.show()
```

3.3.4 정밀도/재현율 트레이드오프



- 정밀도 90% 달성 분류기

```
(y_train_pred == (y_scores > 0)).all()
```

```
True
```

```
y_train_pred_90 = (y_scores > 70000)
```

```
precision_score(y_train_5, y_train_pred_90)
```

```
0.855198572066042
```

거의 90%

```
recall_score(y_train_5, y_train_pred_90)
```

```
0.7070651171370596
```

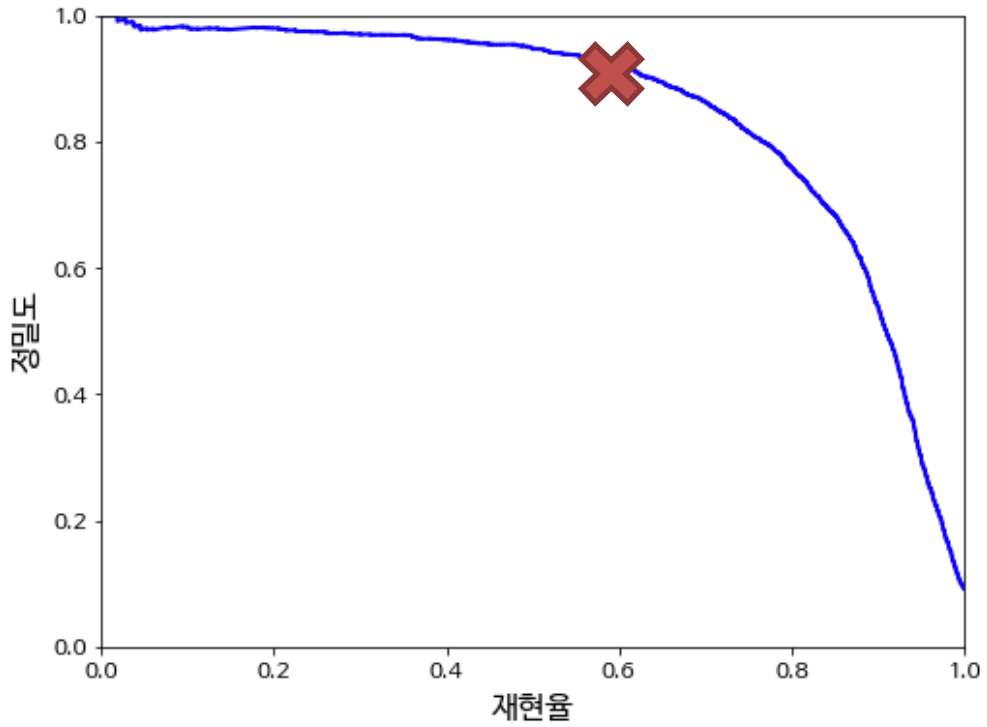
정밀도를 높게 만들더라도, 재현율이 너무 낮다면 전혀 유용하지 않음!

3.3.4 정밀도/재현율 트레이드오프

- 작업에 맞는 최선의 정밀도/재현율 트레이드오프를 만드는 임곗값 선택

```
def plot_precision_vs_recall(precisions, recalls):
    plt.plot(recalls, precisions, "b-", linewidth=2)
    plt.xlabel("재현율", fontsize=16)
    plt.ylabel("정밀도", fontsize=16)
    plt.axis([0, 1, 0, 1])

plt.figure(figsize=(8, 6))
plot_precision_vs_recall(precisions, recalls)
plt.show()
```



재현율 80% 근처에서 정밀도가 급격하게 줄어들기 시작.
이 하강점 직전을 정밀도/재현율 트레이드오프로 선택하는것이 좋다.

예) 재현율 60% 정도인 지점.

3.3.5 ROC 곡선

- **ROC (Receiver Operating Charactoeristic) 곡선**
  - 이진 분류 평가에 많이 사용됨
  - 거짓 양성 비율에 대한 진짜 양성 비율(재현율)의 곡선 (정밀도/재현율 곡선과 비슷하게 생김)
  - 거짓 양성 비율(FPR) = 1 – 진짜 음성 비율 (TNR: 특이도)

  - ROC 곡선 : 민감도(재현율)에 대한 1- 특이도 그래프
  - 여러 임곗값에서 TPR, FPR 계산 : roc_curve() 함수

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

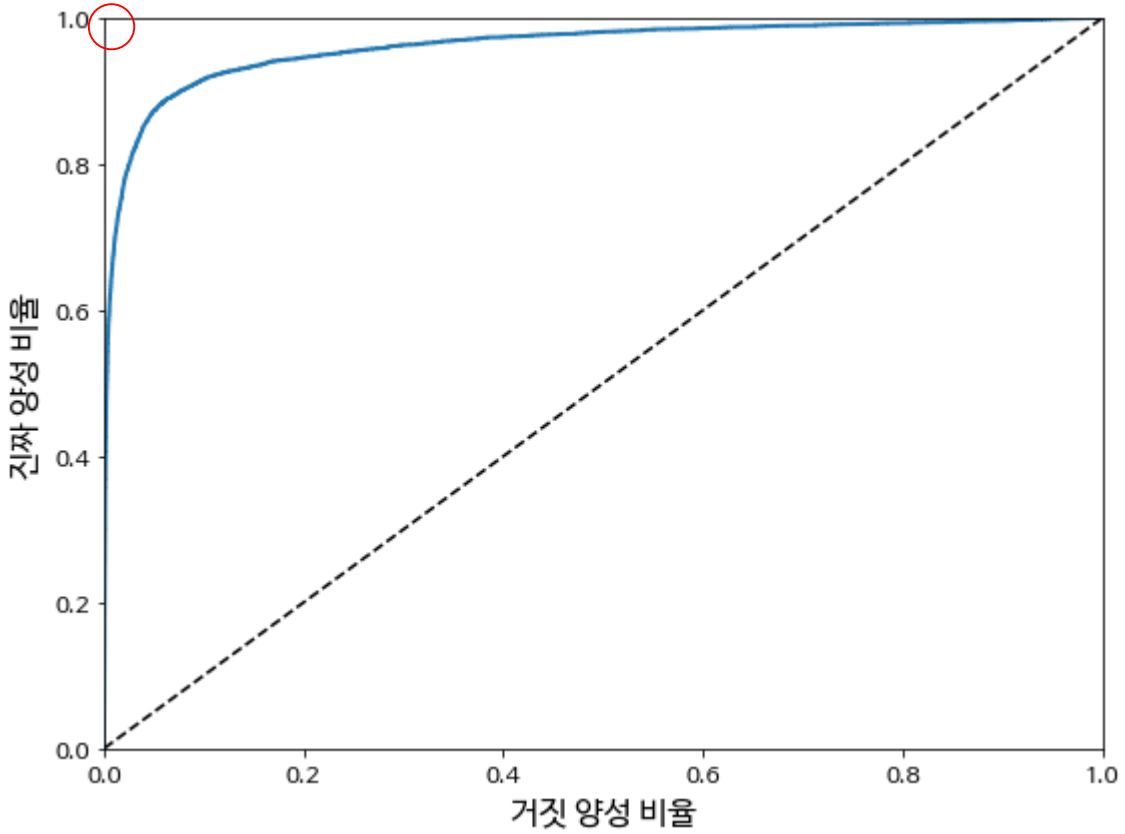
```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('거짓 양성 비율', fontsize=16)
    plt.ylabel('진짜 양성 비율', fontsize=16)

plt.figure(figsize=(8, 6))
plot_roc_curve(fpr, tpr)

plt.show()
```

3.3.5 ROC 곡선

- **재현율(TPR)이 높을수록 분류기가 만드는 거짓 양성(FPR)이 늘어남**
  - 점선 : 완전한 랜덤 분류기의 ROC 곡선
  - 좋은 분류기 : 점선에서 최대한 멀리 떨어져 있어야 (왼쪽 위 모서리)



3.3.5 ROC 곡선

- **ROC 곡선 아래 면적 : Area Under the Curve (AUC)**
  - 완벽한 분류기는 ROC의 AUC가 1
  - 완전한 랜덤 분류기는 0.5

```
from sklearn.metrics import roc_auc_score

roc_auc_score(y_train_5, y_scores)
```

```
0.9614189997126434
```

- **예) RandomForestClassifier vs. SGDClassifier 비교**
  - 훈련 세트의 샘플에 대한 점수

```
from sklearn.ensemble import RandomForestClassifier
forest_clf = RandomForestClassifier(n_estimators=10, random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                    method="predict_proba")
```

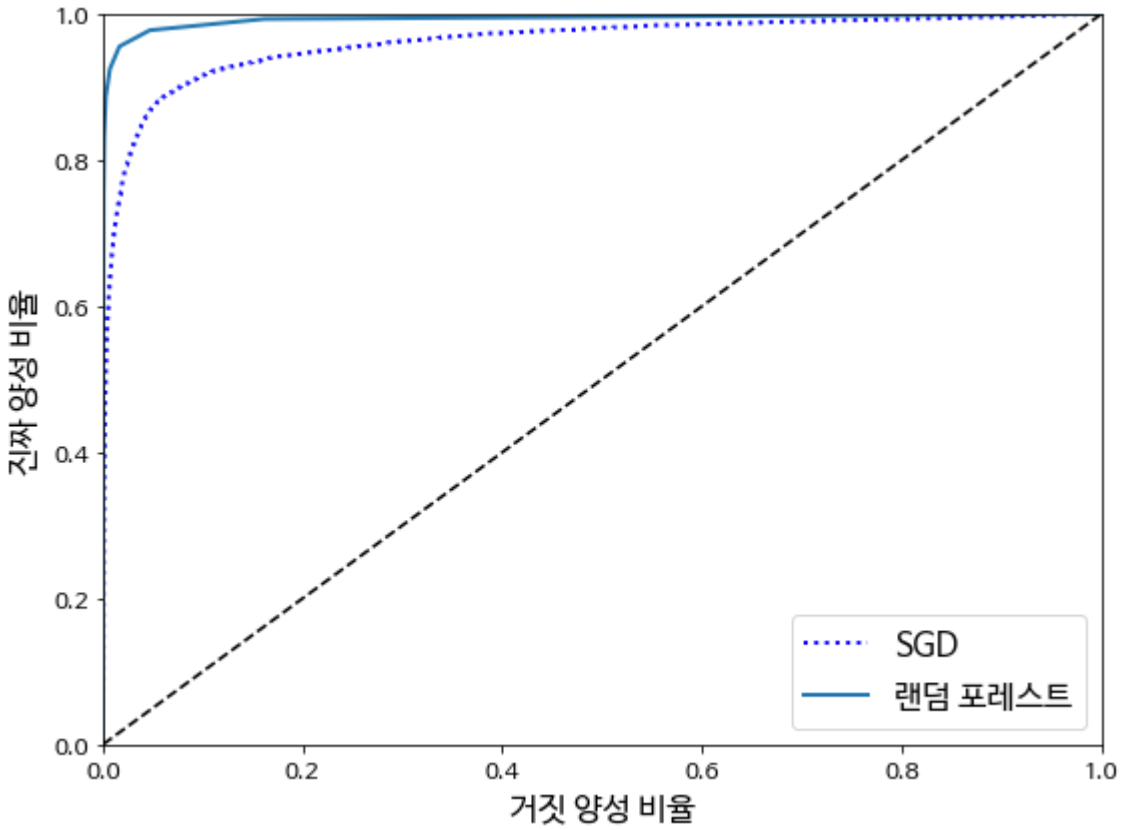
  - 확률이 아니라, 점수 필요 → 양성 클래스 확률을 점수로 사용

```
y_scores_forest = y_probas_forest[:, 1] # 점수는 양성 클래스의 확률입니다
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,y_scores_forest)
```

3.3.5 ROC 곡선

```
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, "b:", linewidth=2, label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "랜덤 포레스트")
plt.legend(loc="lower right", fontsize=16)

plt.show()
```




```
roc_auc_score(y_train_5, y_scores_forest)
```

```
0.9928250745111685
```

3.4 다중 분류

- **다중 분류기 (다항 분류기) : 둘 이상의 클래스 구별**
- **여러 개 클래스 직접 처리 가능 : RandomForest, NaiveBayes**
- **이진분류만 가능 : SVM, 선형분류기**

- **이진분류기를 이용한 다중분류 구성 방법**
  - 특정 숫자 하나만 구분하는 숫자별 이진분류기 10개 (0~9)를 훈련시켜, 클래스가 10개인 숫자 이미지 분류 시스템 구성
    - 이미지 분류 시 각 분류기의 결정 점수 중 가장 높은 것을 클래스로 선택
    - 일대다(OvA) 전략
  - 0과 1 구별, 0과 2 구별, 1과 2구별 등 각 숫자 조합마다 이진 분류기 훈련
    - 일대일(OvO) 전략
    - 클래스가 N개라면 분류기는 $N \times (N-1) / 2$개 필요
    - MNIST 문제 : 45개 분류기 훈련 필요

3.4 다중 분류

- **다중 클래스 분류 작업에 이진 분류 알고리즘을 선택하면 사이킷런이 자동으로 감지해 OvA (SVM 분류기일 때는 OvO) 적용**
  - 0~9까지의 원래 타깃 클래스(y_train)을 사용하여 SGDClassifier를 훈련시키고 예측 생성
  - 내부에서는 사이킷런이 실제로 10개의 이진 분류기를 훈련시키고 각각의 결정 점수를 얻어 점수가 가장 높은 클래스를 선택
  - decision_function() 함수 : 클래스마다 하나씩, 총 10개의 점수 반환
  - → 가장 높은 점수 : 클래스 5

```
sgd_clf.fit(X_train, y_train)
sgd_clf.predict([some_digit])
```

```
array([5.])
```

```
some_digit_scores = sgd_clf.decision_function([some_digit])
some_digit_scores
```

```
array([[-311402.62954431, -363517.28355739, -446449.53064539,
        -183226.61023518, -414337.15339485,  161855.74572176,
        -452576.39616343, -471957.14962573, -518542.33997148,
        -536774.63961222]])
```

3.4 다중 분류

- OvO 혹은 OvA를 강제로 지정하기 위해 OneVsOneClassifier, OneVsRestClassifier를 사용
- SGDClassifier 기반으로 OvO 사용

```
from sklearn.multiclass import OneVsOneClassifier
ovo_clf = OneVsOneClassifier(SGDClassifier(max_iter=5, random_state=42))
ovo_clf.fit(X_train, y_train)
ovo_clf.predict([some_digit])
```

```
array([5.])
```

```
len(ovo_clf.estimators_)
```

```
45
```

- RandomForestClassifier 기반 훈련
  - 직접 샘플을 다중 클래스로 분류
  - 5를 80%확률로 추측

```
forest_clf.fit(X_train, y_train)
forest_clf.predict([some_digit])
```

```
array([5.])
```

```
forest_clf.predict_proba([some_digit])
```

```
array([[0.1, 0. , 0. , 0.1, 0. , 0.8, 0. , 0. , 0. , 0. ]])
```

3.4 다중 분류

- 분류기 평가 : 교차 검증 사용

- 모든 테스트 폴트에서 84% 이상

```
cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
```

```
array([0.84063187, 0.84899245, 0.86652998])
```

- 입력의 스케일을 조정하면 정확도 90%이상

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
```

```
array([0.91011798, 0.90874544, 0.906636  ])
```

3.5 에러 분석

- **실제 프로젝트라면...**
  - 여러 모델을 시도하고,
  - 가장 좋은 몇 개를 골라,
  - GridSearchCV를 사용해 하이퍼파라미터를 세밀하게 튜닝하고,
  - 가능한 한 자동화

- **가능성이 높은 모델을 하나 선정, 모델의 성능 향상**
  - 만들어진 에러 종류 분석 → 오차행렬 분석

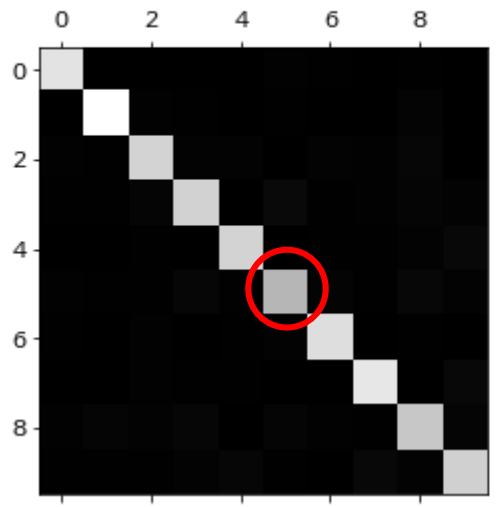
```
y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
conf_mx = confusion_matrix(y_train, y_train_pred)
conf_mx
```

```
array([[5730,    2,   22,    9,   11,   52,   47,    8,   39,    3],
       [   1, 6459,   54,   29,    6,   43,    6,   11,  123,   10],
       [  53,   34, 5371,   90,   80,   23,   79,   57,  157,   14],
       [  48,   37,  147, 5315,    3,  248,   36,   58,  143,   96],
       [  20,   25,   45,   10, 5338,   10,   49,   36,   87,  222],
       [  72,   40,   38,  170,   72, 4616,  108,   28,  186,   91],
       [  35,   24,   56,    1,   39,   86, 5622,    6,   49,    0],
       [  22,   20,   73,   28,   48,   12,    4, 5832,   18,  208],
       [  53,  146,   84,  152,   14,  147,   55,   27, 5037,  136],
       [  44,   33,   29,   89,  164,   39,    3,  211,   77, 5260]],
      dtype=int64)
```

3.5 에러 분석

- **이미지화 →**
  - 배열에서 가장 큰 값 흰색, 가장 작은 값 검은색으로 정규화
- **대부분의 이미지가 올바르게 분류되었음**
  - 숫자 5의 색상이 다른 색에 비해 조금 어두움
  - 데이터셋에 숫자 5의 이미지가 적거나, 분류기가 숫자 5를 다른 숫자만큼 잘 분류하지 못함

```
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```



3.5 에러 분석

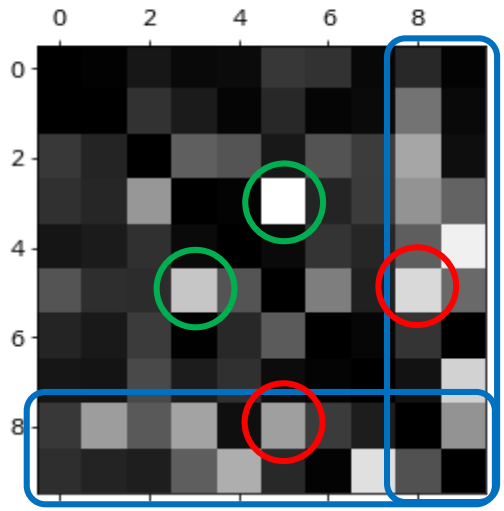
- 오차 행렬의 각 값을 대응되는 클래스의 이미지 개수로 나누어 에러 비율 비교

```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums
```

- 다른 항목은 그대로 유지, 주대각선만 0으로 채워서 그래프로 그림

```
np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)

plt.show()
```



행 : 실제 클래스
열 : 예측한 클래스

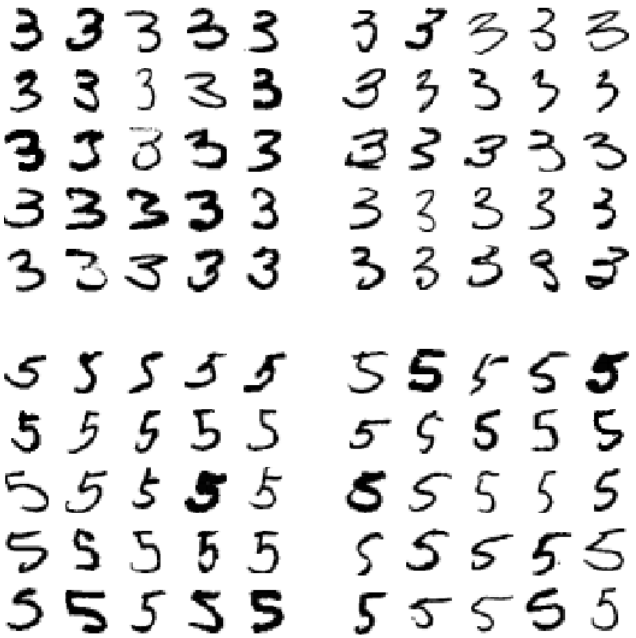
- **8과 9의 열이 상당히 밝음**
  → 많은 이미지가 8과 9로 잘못 분류되었음
- **8과 9의 행도 밝음**
  → 숫자 8과 9가 다른 숫자들과 혼돈이 자주 됨
- **클래스 1의 열은 매우 어두움**
  → 대부분의 숫자 1이 정확하게 분류되었음
- **8→5로 잘못 분류된 경우 > 5→8로 잘못 분류된 경우**
- **3→5, 5→3 잘못 분류되는 경우 많음**

3.5 에러 분석

◆ 3과 5의 샘플 그림

```
cl_a, cl_b = 3, 5
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)]
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)]
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)]

plt.figure(figsize=(8,8))
plt.subplot(221); plot_digits(X_aa[:25], images_per_row=5)
plt.subplot(222); plot_digits(X_ab[:25], images_per_row=5)
plt.subplot(223); plot_digits(X_ba[:25], images_per_row=5)
plt.subplot(224); plot_digits(X_bb[:25], images_per_row=5)
save_fig("error_analysis_digits_plot")
plt.show()
```



왼쪽 블록 두 개 : 3으로 분류된 이미지
오른쪽 블록 두개 : 5로 분류된 이미지

3과 5는 몇 개의 픽셀만 다름
SGDClassifier를 사용하는 경우, 픽셀에 가중치를 할당하고, 픽셀 강도의 가중치 합을 클래스 점수로 계산
→ 3과 5 쉽게 혼동

분류기는 이미지의 위치나 회전 방향에 매우 민감
3과 5의 에러를 줄이는 방법 예)
이미지를 중앙에 위치시키고, 회전되어 있지 않도록 전처리

3.6 다중 레이블 분류

- **샘플마다 여러 개의 클래스를 출력해야 하는 경우**

- **예) 얼굴 인식 분류기**
  - 같은 사진에 여러 사람이 등장
  - 인식된 사람마다 레이블을 하나씩 할당
  - 분류기 인식 얼굴 : [앨리스, 밥, 찰리] 인 경우
  - 한 사진에 앨리스&찰리 → 분류기는 [1,0,1] 출력

- **다중 레이블 분류: 여러 개의 이진 레이블을 출력하는 분류 시스템**

3.6 다중 레이블 분류

- 각 숫자 이미지에 두 개의 타깃 레이블이 담긴 y-multilabel 배열 생성
  - 7 이상
  - 홀수 여부

```
from sklearn.neighbors import KNeighborsClassifier

y_train_large = (y_train >= 7)
y_train_odd = (y_train % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=5, p=2,
           weights='uniform')
```

```
knn_clf.predict([some_digit])
```

```
array([[False,  True]])
```

3.6 다중 레이블 분류

- **다중 레이블 분류기 평가**
  - 예) 각 레이블의 F_1 점수를 구하고, 간단하게 평균 점수 계산

```
import time
start_time = time.time()
y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_multilabel, cv=3
f1_score(y_multilabel, y_train_knn_pred, average="macro")
print("start_time", start_time) #출력해보면, 시간형식이 사람이 읽기 힘든 일련번호형
print("--- %s seconds ---" %(time.time() - start_time))
```

```
start_time 1538413119.2766328
--- 3582.56902384758 seconds ---
```

  - 주의!!! 실행하는데 시간이 매우 오래 걸림
    (하드웨어에 따라 몇 시간씩 걸리기도...)

Any Questions...
Just Ask!