

한눈에 보는 머신 러닝 3장 - 분류

3.1 MNIST

- 고등학생과 미국 인구조사국 직원이 손으로 쓴 70,000개의 작은 숫자 이미지



3.1 MNIST

- MNIST 데이터셋 내려 받는 코드

```
from sklearn.datasets import fetch_openml  
mnist = fetch_openml('mnist_784', version=1)
```

✓ 43.7s

```
mnist.keys()
```

✓ 0.0s

```
>>> dict_keys(['data', 'target', 'frame', 'categories', 'feature_names', 'target_names', 'DESCR', 'details', 'url'])
```

- DESCR : 데이터셋 설명하는 키
- data 키 : 샘플이 하나의 행, 특성이 하나의 열로 구성된 배열
- target 키 : 레이블 배열

3.1 MNIST

```
[13] X, y = mnist["data"], mnist["target"]
      ✓ 0.0s Python

[14] X.shape
      ✓ 0.0s Python
... (70000, 784)

[15] y.shape
      ✓ 0.0s Python
... (70000,)
```

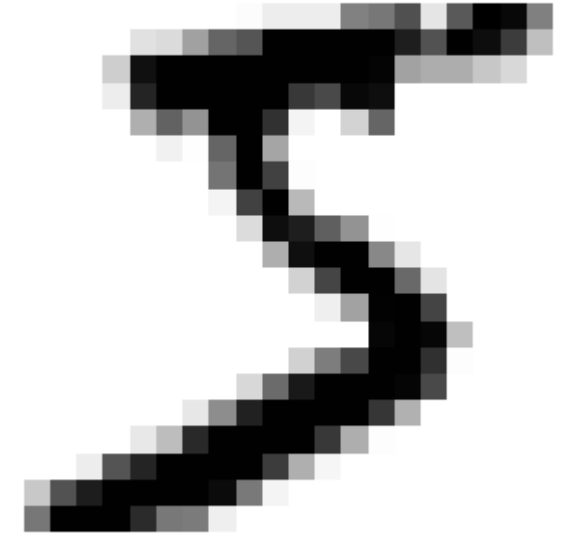
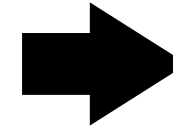
X (샘플) : 이미지가 70000개, 각 이미지마다 784개의 특성($28 * 28$ 픽셀)
y (레이블) : 개개의 특성 0(흰색) ~ 255(검은색)까지의 픽셀 강도를 나타냄.

3.1 MNIST

```
some_digit = X[0]
#reshape() 함수 이용 -> 28 * 28 모양의 어레이로 변환
some_digit_image = some_digit.reshape(28,28)

plt.imshow(some_digit_image, cmap="binary")
plt.axis("off")
plt.show()
```

✓ 0.1s



- mnist 데이터 셋에서 이미지 데이터 : 1차원 어레이였으므로 다시 이미지의 공간적인 구조 정보 유지하기 위해 reshape() 함수를 이용하여 2차원 어레이로 변환.

3.1 MNIST

```
[44] ✓ 0.0s
... '5'

▶ y = y.astype(np.uint8)
[45] ✓ 0.0s
```

```
y[0]
✓ 0.0s
5
```

- 정수형으로 변환

실제 레이블을 확인하면 '5'
하지만 레이블이 문자형으로 저장되어 있음 -> 모델 훈련을 위해선 정수형으로 변환.
8바이트를 사용하여 양의 정수를 표현하는 uint8 자료형 사용

3.1 MNIST

정수형으로 변환하는 이유?

- 정수형으로 입력값 변환 -> 모델 훈련이 더 빠르게 수행, 메모리를 덜 사용함.
- 전처리 과정에서 필요한 다양한 변환 작업이 더 쉬워짐.
- 머신 러닝 모델의 출력값이 정수형인 경우가 더 많음.
-> 출력값, 정답값 모두 정수형으로 일관성 유지

3.1 MNIST

```
▶ x_train, x_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

MNIST 데이터 셋을 훈련 데이터 셋과 테스트 데이터 셋으로 나누는 과정
-> 모델이 학습하지 않은 데이터로 모델의 성능을 평가,
모델의 일반화 능력 더 잘 평가할 수 있다.

3.2 이진 분류기 (binary classifier) 훈련

```
▶ y_train_5 = (y_train == 5) # 5는 true (1) 다른 숫자는 모두 false(0)  
y_test_5 = (y_test == 5)
```

이미지가 숫자 5를 표현하는지 여부 판단 (이진 분류기 훈련)

레이블을 0 또는 1로 변경

- 0 (false) : 숫자 5 아님
- 1 (true) : 숫자 5 맞음

3.2 이진 분류기 훈련

```
from sklearn.linear_model import SGDClassifier

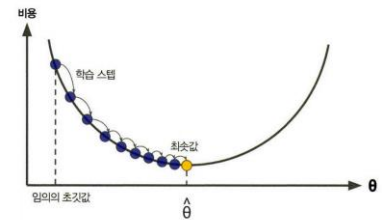
sgd_clf = SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

[52] ✓ 19.8s

SGDClassifier
SGDClassifier(random_state=42)

SGDClassifier(SGD 분류기)

- 확률적 경사 하강법(stochastic gradient descent)
- 한번에 하나씩 훈련 샘플 처리 후 파라미터 조정
- 매우 큰 데이터셋 처리에 효율적, 온라인 학습에도 적합



3.2 이진 분류기 훈련

```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

[52] ✓ 19.8s

... **SGDClassifier**
SGDClassifier(random_state=42)

확률적 경사 하강법 분류기를 훈련시키기 위해 fit() 메서드 호출

- max_iter=1000 : 훈련 데이터를 최대 1000번 반복해서 학습에 사용
- tol=1e-3: 전체 훈련 세트를 한 번 학습할 때마다 성능 향상이 연속적으로 지정된 값 이상으로 이루어지지 않을 때 학습을 멈추도록 함
5번 이상 성능 향상이 10의 -3승 이상 이루어지지 않으면 학습을 강제로 멈춤

3.2 이진 분류기 훈련

```
▶ ▾ sgd_clf.predict([some_digit])  
[53] ✓ 0.0s  
... array([ True])
```

숫자 5의 이미지 감지 -> array([True]) 분류기가 이미지가 5를 나타낸다고 추측함.

3.3 성능 측정

- ① 교차 검증을 사용한 정확도 측정
- ② 정밀도 / 재현율 조율
- ③ AUC 측정

3.3.1 교차 검증을 사용한 정확도 측정

```
• ✓ from sklearn.model_selection import StratifiedKFold
    from sklearn.base import clone

    # 계층별 샘플링 진행: 3개의 폴드 지정
    # 주의사항: shuffle=False가 기본값이기 때문에 random_state를 삭제하던지
    # shuffle=True로 지정하라는 경고가 발생한다.
    # 0.24버전부터는 에러가 발생할 예정이므로 향후 버전을 위해 shuffle=True를 지정한다.
    skfolds = StratifiedKFold(n_splits=3, random_state=42, shuffle=True)

    # 3개의 폴드를 대상으로 두 개씩 돌아가면 훈련 및 검증 진행
    ✓ for train_index, test_index in skfolds.split(X_train, y_train_5):
        # 기존 모델에 영향을 주지 않도록 모델을 복제해서 사용한다.
        clone_clf = clone(sgd_clf)
        X_train_folds = X_train[train_index]
        y_train_folds = y_train_5[train_index]
        X_test_fold = X_train[test_index]
        y_test_fold = y_train_5[test_index]

        clone_clf.fit(X_train_folds, y_train_folds)
        y_pred = clone_clf.predict(X_test_fold)
        n_correct = sum(y_pred == y_test_fold)
        print(n_correct / len(y_pred))
```

- 교차 검증 모델 직접 구현

-> cross_val_score() 함수가 제공하지 않는 기능 이용하려면 직접 교차 검증 구현.

3겹 교차 검증을 위해 계층별 샘플링 ~ 세 번의 훈련 및 검증 직접 진행

3.3.1 교차 검증을 사용한 정확도 측정

```
▶ ▾  
from sklearn.model_selection import cross_val_score  
cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")  
[56] ✓ 28.6s  
... array([0.95035, 0.96035, 0.9604 ])
```

- 3겹 교차 검증 실행 (결과 : 정확도 95% 이상)
- cv=3 : 폴더 3개 사용
- scoring="accuracy" : 정확도 검증

3.3.1 교차 검증을 사용한 정확도 측정 - 문제점

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

[57] ✓ 0.0s

▶ ▾ never_5_clf = Never5Classifier()
    cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")

[58] ✓ 0.8s

... array([0.91125, 0.90855, 0.90915])
```

- 무조건 False 라고 예측하는 분류기
무조건 5가 아니라고 찍는 분류기도 90%의 정확도를 보여줌.
이미지의 10% 정도만 숫자 5이기 때문에 무조건 '5 아님' 이라고 예측하면 정확히 맞출 확률
→ 90%

3.3.2 오차 행렬

- 오차행렬의 행 : 실제 클래스
- 오차행렬의 열 : 예측한 클래스
- 클래스 A의 샘플이 클래스 B의 샘플로 분류된 횟수를 알고자 하면 A행 B열의 값 확인

3.3.2 오차 행렬

```
▶ ~  
from sklearn.model_selection import cross_val_predict  
  
y_trained_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)  
[27] ✓ 29.7s Python
```

- 오차 행렬을 만들기 위해선 실제 타깃과 비교할 수 있도록 예측값을 만들어야 함.
- `cross_val_predict()` k-겹 교차 검증 수행 -> 평가 점수 반환 x, 각 테스트 폴드에서 얻은 예측 반환함 (훈련 세트의 모든 샘플에 대해 깨끗한 예측 얻음.)

3.3.2 오차 행렬



```
from sklearn.metrics import confusion_matrix  
confusion_matrix(y_train_5, y_train_pred)
```

[27] ✓ 0.0s

```
... array([[53892,   687],  
          [ 1891, 3530]], dtype=int64)
```

- 첫번째 행 : '5 아님' 이미지(negative class) 53892개로 정확히 분류 (true negative), 나머지 687개 '5' 라고 잘못 분류 (false positive)

3.3.2 오차 행렬

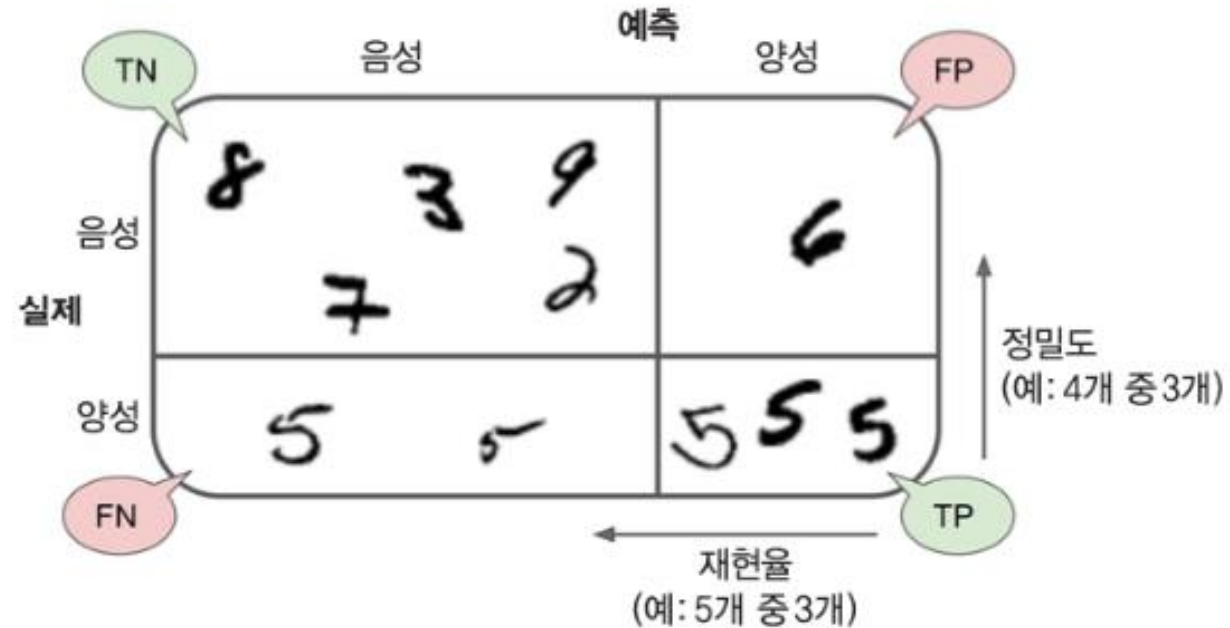
```
▶ ▼ from sklearn.metrics import confusion_matrix
      confusion_matrix(y_train_5, y_train_pred)

[27] ✓ 0.0s

... array([[53892,   687],
           [ 1891, 3530]], dtype=int64)
```

- 두번째 행: '5' 이미지(positive class)에 대한 것
1891개를 '5 아님'으로 잘못 분류 (false negative)
나머지 3530개를 정확히 '5'라고 분류(true positive)

3.3.2 오차 행렬



- TN(True negative) : 참 음성 개수, 5가 아닌 이미지를 5가 아니라고 정확하게 예측한 경우의 수
- FP(False positive) : 거짓 양성 개수, 5가 아닌 이미지를 5라고 잘못 예측한 경우의 수
- FN(False negative) : 거짓 음성 개수, 5를 나타내는 이미지를 5가 아니라고 잘못 예측한 경우의 수
- TP(True positive) : 참 양성 개수, 5를 나타내는 이미지를 5라고 정확하게 예측한 경우의 수

3.3.2 오차 행렬

```
▶ ✓ from sklearn.metrics import confusion_matrix
    confusion_matrix(y_train_5, y_train_pred)

[27] ✓ 0.0s

... array([[53892,   687],
          [ 1891, 3530]], dtype=int64)
```

- TN(True negative) : 참 음성 개수, 5가 아닌 이미지를 5가 아니라고 정확하게 예측한 경우의 수 (53892)
- FP(False positive) : 거짓 양성 개수, 5가 아닌 이미지를 5라고 잘못 예측한 경우의 수 (687)
- FN(False negative) : 거짓 음성 개수, 5를 나타내는 이미지를 5가 아니라고 잘못 예측한 경우의 수 (1891)
- TP(True positive) : 참 양성 개수, 5를 나타내는 이미지를 5라고 정확하게 예측한 경우의 수 (3530)

3.3.3 정밀도와 재현율

정밀도 (precision)

- 양성 예측의 정확도

```
... array([[53892, 687],  
          [ 1891, 3530]], dtype=int64)
```

```
▶ from sklearn.metrics import precision_score, recall_score  
  precision_score(y_train_5, y_train_pred)
```

[29] ✓ 0.0s

```
... 0.8370879772350012
```

$$\text{precision} = \frac{TP}{TP + FP} = \frac{3530}{3530 + 687} = 0.837$$

3.3.3 정밀도와 재현율

재현율(recall)

- 정밀도 하나만으로 분류기 성능 평가 불가능.
(숫자 5를 가리키는 이미지 중 숫자 5라고 판명한 비율인 재현율 고려 x)
- 분류기가 정확히 감지한 양성 샘플의 비율
- 민감도(sensitivity), 진짜 양성 비율(true positive rate(TPR))

```
[29] ✓ 0.0s  
... 0.6511713705958311
```

$$\text{recall} = \frac{TP}{TP + FN}$$

$$\frac{3530}{3530 + 1891} = 0.651$$

3.3.3 정밀도와 재현율

F1 점수

- 정밀도와 재현율의 조화 평균(harmonic mean)
- F1 점수가 높을 수록 분류기의 성능을 좋게 평가하지만 경우에 따라 재현율과 정밀도 둘 중의 하나에 높은 가중치를 두어야 할 때가 있음.

$$\text{조화평균} = \frac{1}{\frac{\frac{1}{a} + \frac{1}{b}}{2}} = \frac{2ab}{a+b}$$

```
▶ from sklearn.metrics import f1_score
  f1_score(y_train_5, y_train_pred)

[30] ✓ 0.0s

... 0.7325171197343846
```



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

3.3.3 정밀도와 재현율

정밀도 vs 재현율

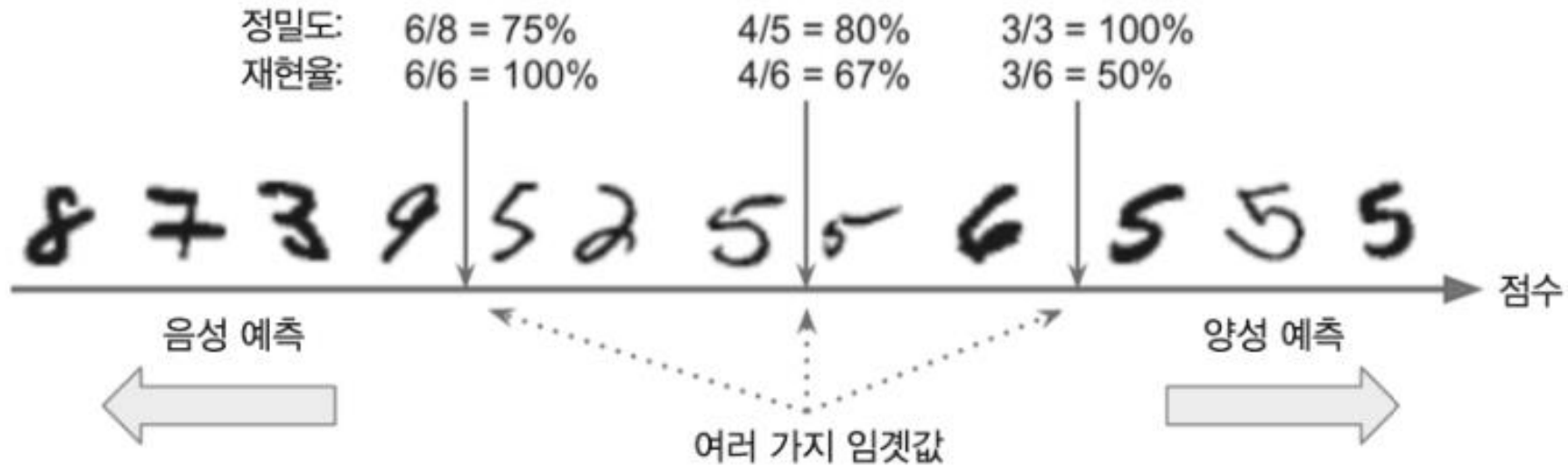
- 모델 사용 목적에 따라 정밀도와 재현율 중요도가 달라질 수 있음.
- 재현율이 더 중요한 경우 : 암 진단 기준
 - 정밀도 : 양성으로 진단된 경우 실제로도 양성인 경우의 비율
 - 재현율 : 실제로 양성인 경우 중에서 양성으로 진단하는 경우의 비율
- 정밀도가 더 중요한 경우 : 어린아이에게 안전한 동영상 걸러내는 분류기
 - 정밀도 : 안전하다고 판단된 동영상 중 실제로 안전한 동영상 비율
 - 재현율 : 실제로 동영상 중 좋은 동영상이라고 판단되는 동영상 비율

3.3.4 정밀도/재현율 트레이드오프

- 정밀도  재현율  (상호 반비례 관계)
- 정밀도와 재현율 사이의 적절한 비율을 유지하는 적절한 결정 임계값 지정해야함.
- 결정함수(decision function) 사용하여 각 샘플 점수 계산
점수가 임계값보다 크면 샘플을 양성 클래스에 할당, 아니면 음성 클래스에 할당

3.3.4 정밀도/재현율 트레이드오프

- 정밀도 \uparrow 재현율 \downarrow (상호 반비례 관계)
- 정밀도와 재현율 사이의 적절한 비율을 유지하는 적절한 결정 임계값 지정해야함.



3.3.4 정밀도/재현율 트레이드오프

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

[31] ✓ 0.0s

```
... array([2164.22030239])
```

```
threshold = 0
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
```

[33] ✓ 0.0s

```
... array([ True])
```

```
threshold = 8000
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
```

[34] ✓ 0.0s

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

- Decision_function() 이용
-> 예측한 점수를 각 샘플에 대해 계산

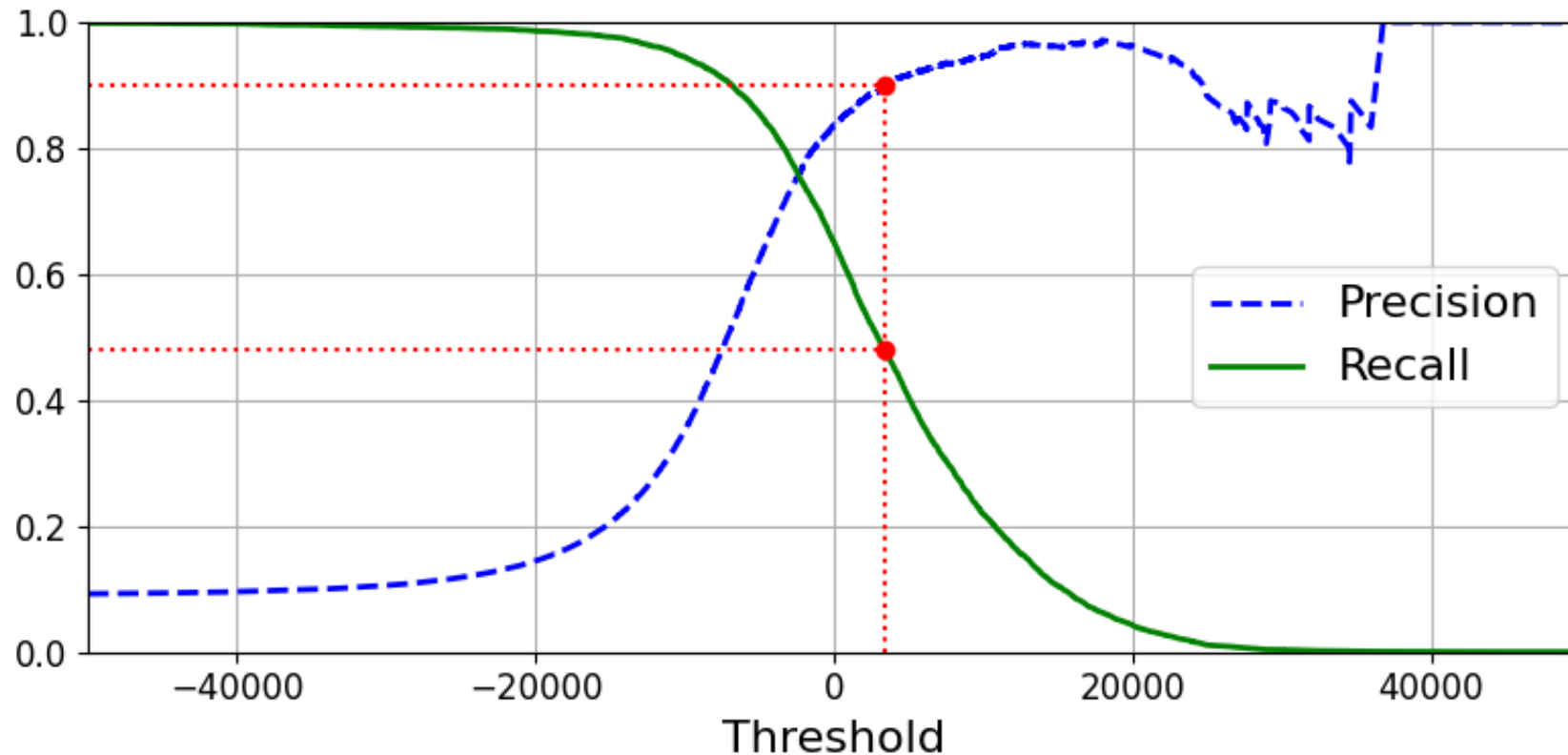
3.3.4 정밀도/재현율 트레이드오프

```
[34] y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,  
                                method="decision_function")  
✓ 31.0s
```

```
▶ from sklearn.metrics import precision_recall_curve  
  precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)  
[35] ✓ 0.0s
```

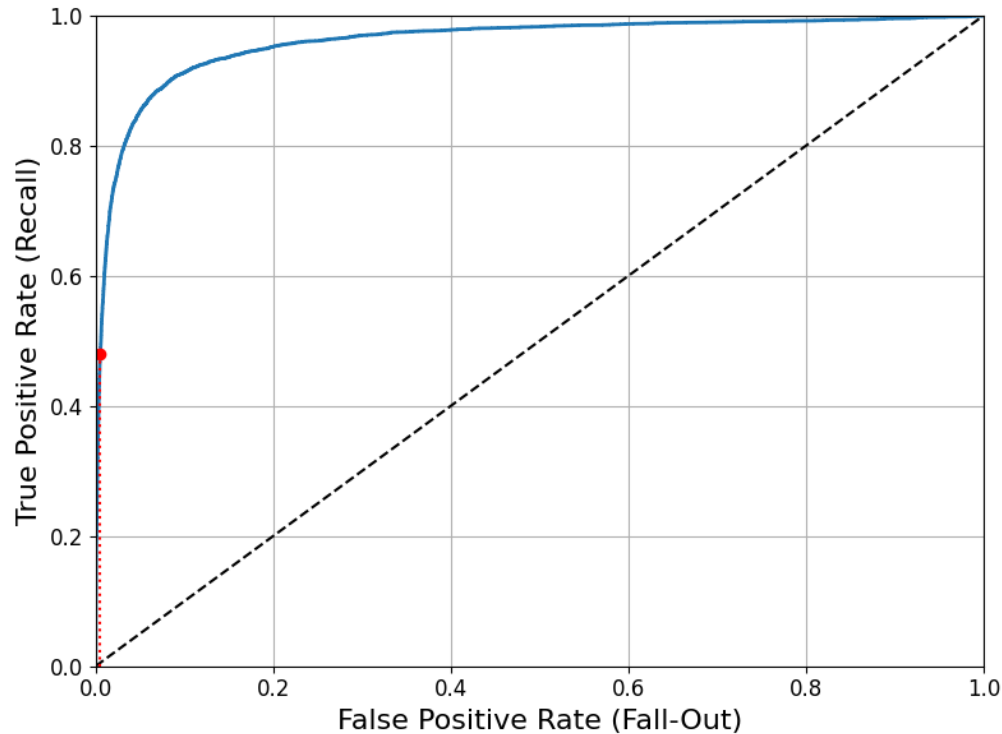
- 교차검증 이용하여 훈련 샘플에 대한 예측점수 확인.
 - Precision_recall_curve() 함수
- > 가능한 모든 임계값에 대한 정밀도와 재현율을 계산한 값 반환

3.3.4 정밀도/재현율 트레이드오프



- 빨강 점들과 연결된 빨강 점선 : 정밀도가 90%가 되는 지점의 정밀도의 재현율

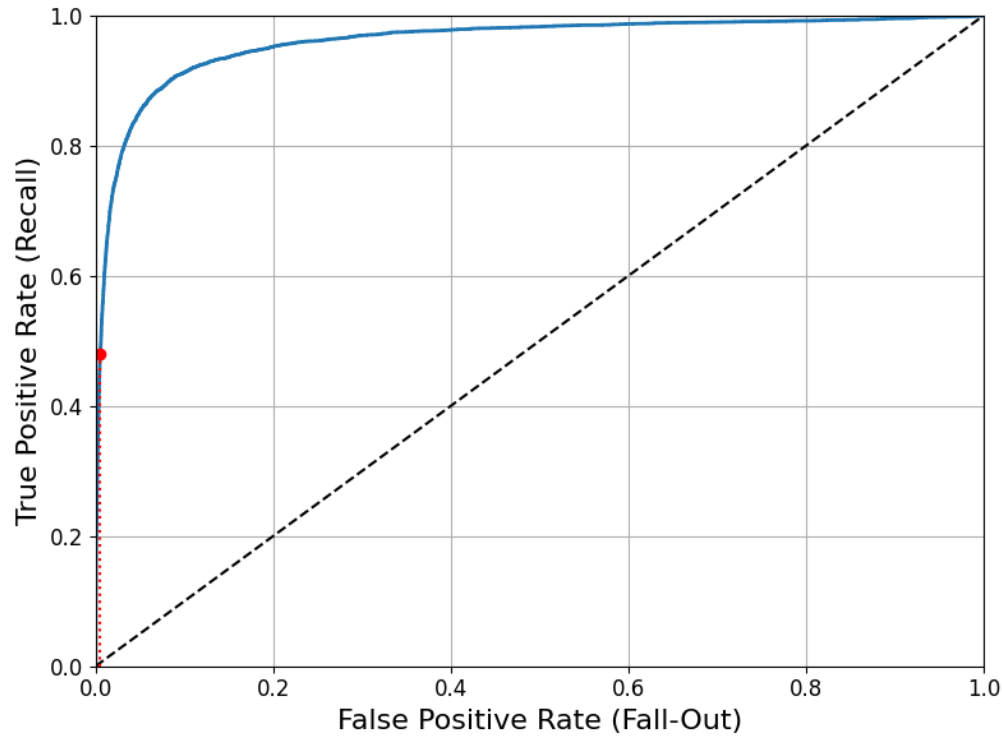
3.3.5 ROC 곡선



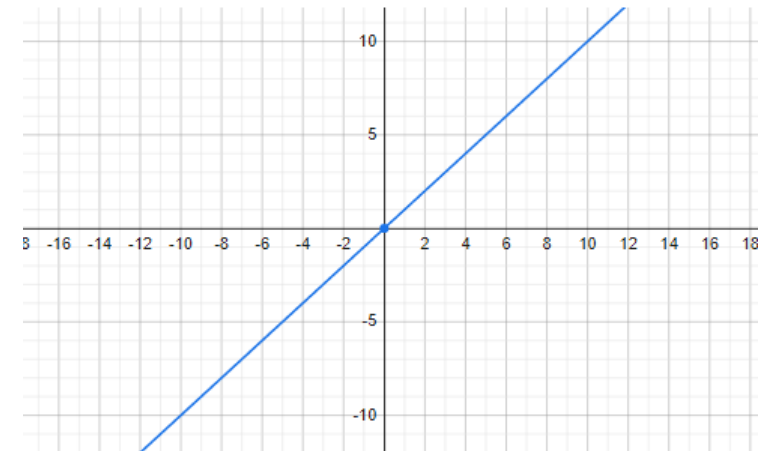
- 거짓 양성 비율(false positive rate(FPR))에 대한 진짜 양성 비율(true positive rate(TPR))의 곡선
- FPR : 양성으로 잘못 분류된 음성 샘플비율
- 1-TNR(음성으로 정확하게 분류한 음성샘플 비율 (True Negative Rate) 뺀 값)
- 민감도(또는 재현율)에 대한 (1-특이도)그래프

- $TNR(\text{True Negative Rate}) = \text{특이도(Specificity)}$

3.3.5 ROC 곡선



- 재현율(TPR) ↑ , 거짓 양성 비율(FPR) ↑
- 점선 : 완전한 랜덤 분류기의 ROC 곡선
 - 점선에서 최대한 멀리 떨어질수록 좋은 분류기



3.3.5 ROC 곡선

- 곡선 아래의 면적(area under the curve(AUC)) 측정 – 분류기 비교
- 완벽한 분류기 : ROC의 AUC = 1, 완전한 랜덤 분류기 0.5

```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_train_5,y_scores)
```

✓ 0.0s

0.9604938554008616

- 일반적으로 양성 class 드물거나, False Negative(FN)보다 False Positive(FP)가 더 중요할 때(악성 종양 분류 등)는 PR 곡선(정밀도-재현율 곡선), 그렇지않으면 ROC곡선 사용

3. 4 다중 분류

- 다중 분류기(multiclass classifier), 다항 분류기(multinomial classifier)
- 둘 이상의 클래스 구별
- SGD 분류기, 랜덤 포레스트 분류기, 나이브 베이즈(naive Bayes 분류기)
- 이진 분류기(Binary classifier)
- 두 개의 클래스 구별
- 로지스틱 회귀, 서포트 벡터 머신 등...
- 이진분류기 여러 개 사용해서 다중 클래스를 분류할 수도 있음

3.4 다중 분류

- OvR(one-versus-the-rest) , OvA(one-versus-all) 전략
- 숫자 5 예측하기에서 사용했던 이진 분류 방식을 숫자 10개에 훈련하여 클래스가 10개인 숫자 이미지 분류 시스템 만들기
- 이미지를 분류할 때 각 분류기의 결정 점수 중 가장 높은 것을 클래스로 선택

3.4 다중 분류

- OvO(one-versus-one) 방식
- 0과 1 구별, 0과 2 구별, 1과 2 구별 같이 각 숫자의 조합마다 이진 분류기 훈련
- 각 분류기의 훈련에 전체 훈련 세트 중 구별할 두 클래스에 해당하는 샘플만 필요함.
- 일부 알고리즘은 훈련세트 크기에 민감. -> 작은 훈련세트에서 많은 분류기를 훈련시키는 것이 빠름 -> OvO 선호
- $N(N-1)/2$
- 대부분의 이진 분류 알고리즘에서는 OvR 선호

3.4 다중 분류

```
from sklearn.svm import SVC

svm_clf = SVC(gamma="auto", random_state=42)
svm_clf.fit(X_train[:1000], y_train[:1000]) # 이제부터는 y_train_5 대신에 y_train 사용

SVC(gamma='auto', random_state=42)
```

첫째 훈련 샘플에 대해 숫자 5로 정확하게 예측한다.

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

array([5], dtype=uint8)
```

- 5를 구별할 타깃 클래스(y_train_5) 대신 0~9까지의 원래 타깃 클래스(y_train)을 사용해 SVC 훈련. -> 예측 만듬
- 내부에서는 사이킷런이 OvO전략 사용하여 10개의 이진 분류기 훈련 후 각각의 결정 점수를 얻어 점수가 가장 높은 클래스를 선택함

3.4 다중 분류

```
[ ] svm_clf.classes_  
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)
```

```
[ ] svm_clf.classes_[5]  
5
```

- 분류기가 훈련될 때 `classes_` 속성에 타깃 클래스의 리스트를 값으로 정렬하여 저장
- 5번 인덱스에 해당하는 클래스는 5

3.5 에러 분석

- 가능성이 높은 모델을 찾은 후에는 에러 분석을 해야함
- 머신러닝 모델 성능 향상, 모델의 동작 이해 등을 위해
- 신뢰성 있고 유의미한 예측 모델 구축 가능

3.5 에러 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

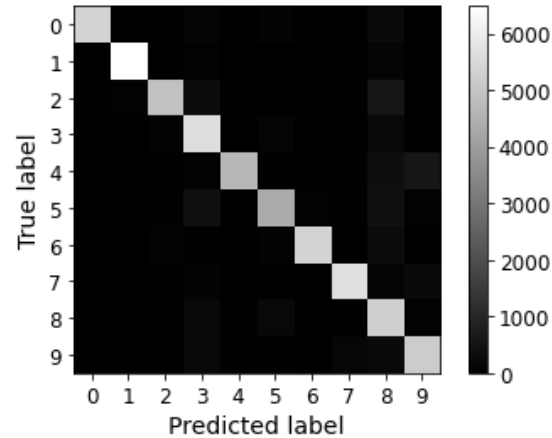
```
array([[5577,  0,  22,  5,  8,  43,  36,  6, 225,  1],
       [  0, 6400,  37, 24,  4,  44,  4,  7, 212, 10],
       [ 27,  27, 5220, 92, 73,  27, 67, 36, 378, 11],
       [ 22,  17, 117, 5227,  2, 203, 27, 40, 403, 73],
       [ 12,  14,  41,  9, 5182, 12, 34, 27, 347, 164],
       [ 27,  15,  30, 168, 53, 4444, 75, 14, 535, 60],
       [ 30,  15,  42,  3,  44,  97, 5552,  3, 131,  1],
       [ 21,  10,  51, 30, 49, 12,  3, 5684, 195, 210],
       [ 17,  63,  48, 86,  3, 126, 25, 10, 5429,  44],
       [ 25,  18,  30, 64, 118, 36,  1, 179, 371, 5107]])
```

- `cross_val_predict()` : 예측 만듦
- `Confusion_matrix()` : 오차행렬 생성

3.5 에러 분석

```
[ ] from sklearn.metrics import plot_confusion_matrix  
  
plot_confusion_matrix(sgd_clf, X_train, y_train, include_values=False, cmap='gray')
```

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f96fa9b6040>



- Matshow() : 흑백 이미지로 표현 가능 , 사이킷런 0.22버전부터 plot_confusion_matrix() 사용하여 흑백이미지 표현, 가능
- 오차행렬에 사용된 숫자가 클수록 해당 칸의 색은 상대적으로 밝아짐
- 숫자 5 이미지가 상대적으로 적거나 다른 숫자 5에 대한 분류기의 정확도 떨어짐 : 어두움

3.5 에러 분석

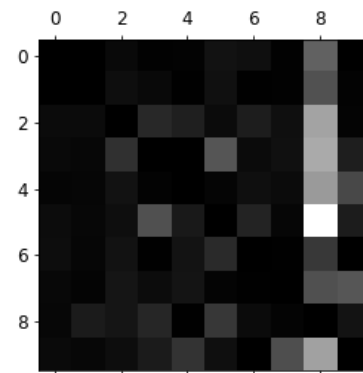
```
[ ] row_sums = conf_mx.sum(axis=1, keepdims=True) # 숫자별 총 이미지 개수
    norm_conf_mx = conf_mx / row_sums             # 숫자별 오차율 행렬
```

각 숫자별로 이미지 개수 확인한 다음에
숫자 별 오차율로 이루어진 행렬 생성
오차행렬 / 숫자별 이미지 총 개수

```
[ ] np.fill_diagonal(norm_conf_mx, 0) # 대각선을 0으로 채우기
    plt.matshow(norm_conf_mx, cmap=plt.cm.gray)

    save_fig("confusion_matrix_errors_plot", tight_layout=False)
    plt.show()
```

그림 저장: confusion_matrix_errors_plot



- 숫자별 오차율을 계산하여 이미지로 표현 : 정확한 이유를 알 수 있음
- 대각선 0으로 채움 -> 오류가 있는 칸이 보다 선명하게 드러남
- 클래스 8의 열 : 상당히 밝음, 많은 이미지가 8로 잘못 분류됨
- 클래스 8의 행 : 실제 8이 적절히 8로 분류됨

3.5 에러 분석

```
cl_a, cl_b = 3, 5
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)] # 3을 3으로 예측한 샘플들
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)] # 3을 5로 예측한 샘플들
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)] # 5를 3으로 예측한 샘플들
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)] # 5를 5로 예측한 샘플들

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



- 개개의 에러 분석 시 분류기가 무슨 일을 하고 왜 잘못 되었는지 알 수 있음
- 더 어렵고 시간 오래걸림
- 왼쪽 5*5 블록 2개 = 3으로 분류된 이미지, 오른쪽 5*5 두 개 = 5로 분류된 이미지
- SGDClassifier 사용 -> 3과 5는 몇 개의 픽셀만 달라 모델이 쉽게 혼동 가능

3.5 에러 분석



- 3과 5의 에러를 줄이는 법
- 3과 5는 위쪽 선과 아래쪽 호를 이어주는 작은 직선의 위치로 나뉨
- 이미지를 중앙에 위치시키고 회전되어 있지 않도록 전처리함

3.6 다중 레이블 분류(multilabel classification)

- 분류기가 샘플마다 여러 개의 클래스 출력해야 할 상황
- Ex) 얼굴 인식 분류기 : 같은 사진에 여러 사람이 등장하면 사람마다 꼬리표(tag)를 붙여야 함.
 - 엘리스, 밥, 찰리 세 얼굴 인식하도록 훈련. 분류기가 엘리스와 찰리가 있는 사진을 보면 [1,0,1] 출력 (엘리스 o, 밥 x, 찰리 o)
- 이진 꼬리표를 출력하는 분류 시스템

3.6 다중 레이블 분류(multilabel classification)

```
▶ from sklearn.neighbors import KNeighborsClassifier

y_train_large = (y_train >= 7)    7보다 큰 값인지 확인
y_train_odd = (y_train % 2 == 1)   홀수인지 확인
y_multilabel = np.c_[y_train_large, y_train_odd]

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

knn_clf.predict([some_digit])

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

- y_multilabel : 각 숫자 이미지에 두 개의 타깃 레이블이 담김
- KNeighborsClassifier 인스턴스 생성 후 다중 타깃 배열 사용하여 훈련

3.6 다중 레이블 분류(multilabel classification)

```
▶ from sklearn.neighbors import KNeighborsClassifier

y_train_large = (y_train >= 7)    7보다 큰 값인지 확인
y_train_odd = (y_train % 2 == 1)   홀수인지 확인
y_multilabel = np.c_[y_train_large, y_train_odd]

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

knn_clf.predict([some_digit])

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

```
knn_clf.predict([some_digit])

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

- 숫자 5 확인해보면 false, true 값을 볼 수 있음($5 < 7$, 홀수)

- y_multilabel : 각 숫자 이미지에 두 개의 타깃 레이블이 담김
- KNeighborsClassifier 인스턴스 생성 후 다중 타깃 배열 사용하여 훈련

3.6 다중 출력 분류(multioutput classification)

- 다중 레이블 분류에서 한 레이블이 다중 클래스가 될 수 있도록 일반화

3.6 다중 출력 분류(multiclass classification)

```
[ ] # MNIST 훈련 세트의 모든 샘플에 잡음 추가
noise = np.random.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise

# MNIST 테스트 세트의 모든 샘플에 잡음 추가
noise = np.random.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise

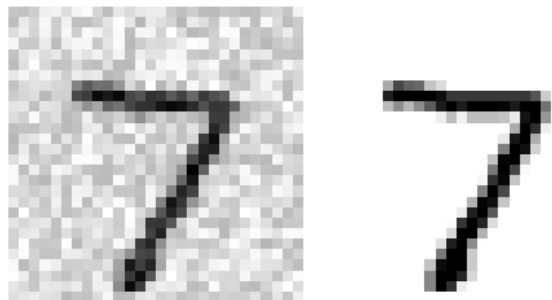
# 레이블은 사진 원본
y_train_mod = X_train
y_test_mod = X_test
```

```
some_index = 0 # 0번 인덱스

plt.subplot(121); plot_digit(X_test_mod[some_index]) # 잡음 추가된 이미지
plt.subplot(122); plot_digit(y_test_mod[some_index]) # 원본 이미지

save_fig("noisy_digit_example_plot")
plt.show()
```

그림 저장: noisy_digit_example_plot



- 원: 잡음 추가한 이미지 오: 깨끗한 타깃 이미지
IN 모델을 이용하여 다중 출력 모델을 훈련시킨다.

```
] knn_clf.fit(X_train_mod, y_train_mod)

KNeighborsClassifier()
```

- MNIST 이미지에서 추출한 훈련세트와 테스트 세트에 넘파이의 randint() 함수 사용해서 픽셀 강도에 잡음을 추가함

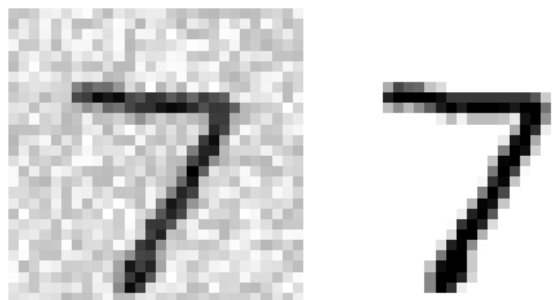
3.6 다중 출력 분류(multioutput classification)

```
some_index = 0 # 0번 인덱스

plt.subplot(121): plot_digit(X_test_mod[some_index]) # 잡음 추가된 이미지
plt.subplot(122): plot_digit(y_test_mod[some_index]) # 원본 이미지

save_fig("noisy_digit_example_plot")
plt.show()
```

그림 저장: noisy_digit_example_plot



IN 모델을 이용하여 다중 출력 모델을 훈련시킨다.

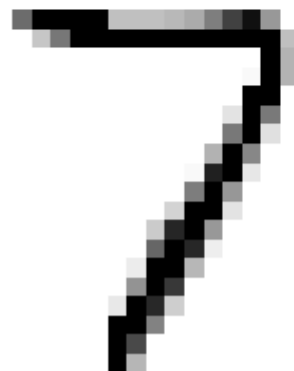
```
] knn_clf.fit(X_train_mod, y_train_mod)

KNeighborsClassifier()
```

```
clean_digit = knn_clf.predict([X_test_mod[some_index]])
```

```
plot_digit(clean_digit)
save_fig("cleaned_digit_example_plot")
```

그림 저장: cleaned_digit_example_plot



- 분류기 훈련시켜 이미지를 깨끗하게 만듦.