분류 - mnist

데이터 다운받기

In [3]:

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

Out[3]:

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

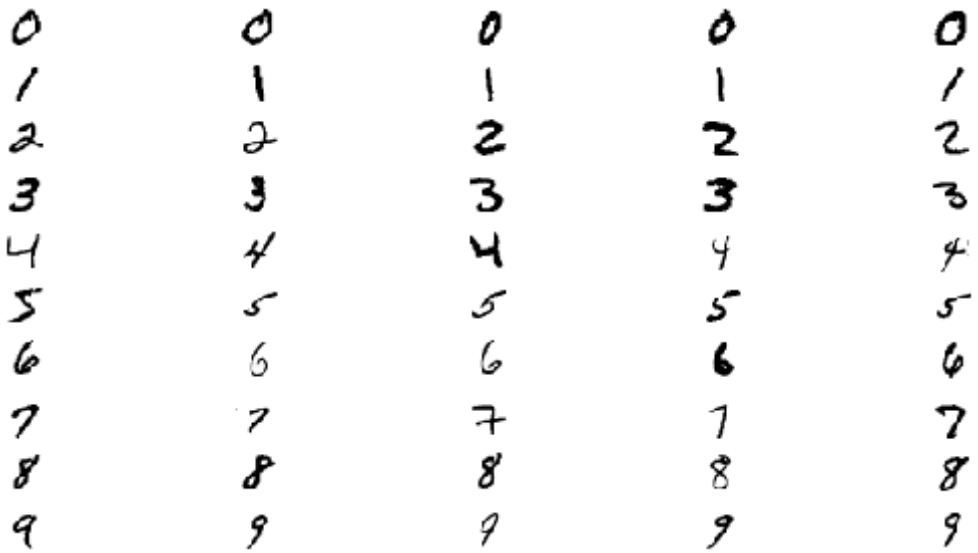
In [4]:

```
X,y = mnist["data"], mnist["target"]
print(X.shape, y.shape)
```

```
(70000, 784) (70000,)
```

In [5]:

```
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np

plt.figure(figsize=(10,5))
for i in range(10):
    idx = np.where(y==str(i))[0][:5] # 해당 숫자에 해당하는 index 위치 반환
    for th, j in enumerate(idx):
        plt.subplot(10,5,(th+1)+(i*5))
        digit = X[j].reshape(28,28)
        plt.imshow(digit, cmap="binary")
        plt.axis("off")
```



In [6]:

```
import numpy as np
```

In [7]:

```
y = y.astype(np.uint8)
```

In [8]:

```
# 앞쪽 60,000개 이미지와 뒤쪽 60,000개 이미지로 이미 나누어져 있음
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

이진 분류기 훈련

In [9]:

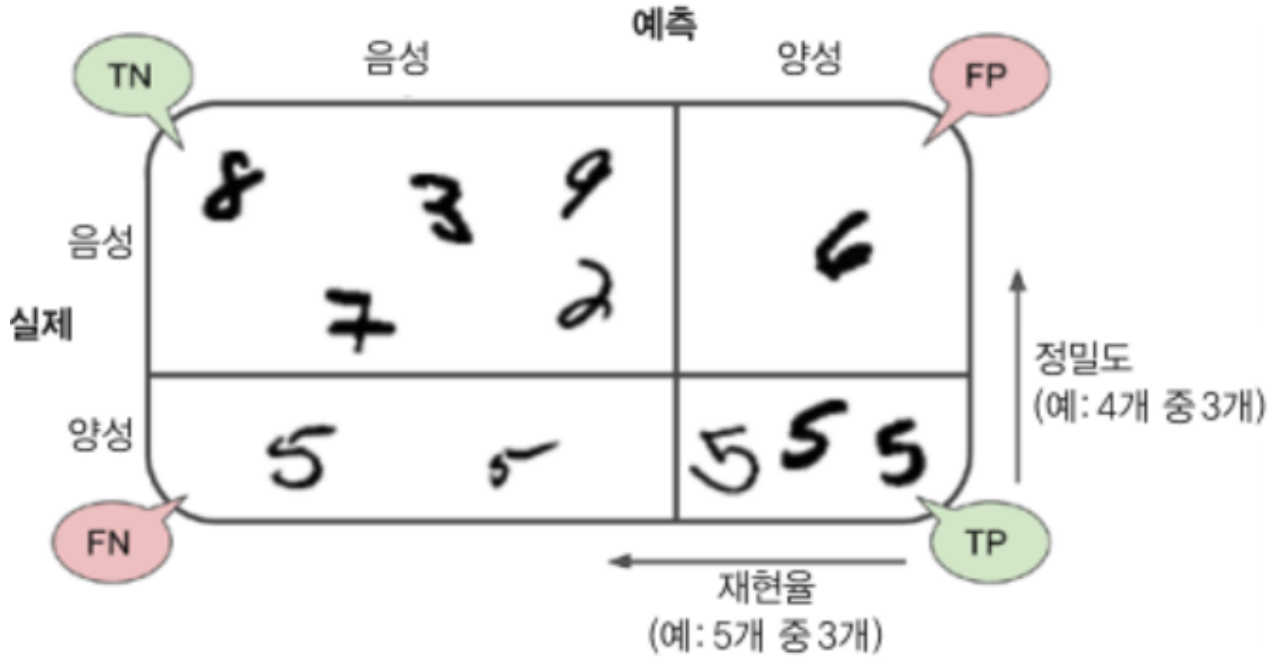
```
# 숫자 5만 식별하는 이진분류기
y_train_5 = (y_train==5)
y_test_5 = (y_test==5)
```

In [10]:

```
# SGD Classifier : 확률적 경사하강법(한 epoch 뒤에 데이터를 다시 무작위로 섞기 때문)
# 한개 한개 샘플을 독립적으로 처리
from sklearn.linear_model import SGDClassifier

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

predicted = sgd_clf.predict(X_test)
```

분류 성능측정

- 정확도 : (맞은 개수)/(전체 개수)
  - 불균형한 데이터셋의 경우 적절한 성능 측정 지표가 아님
- 오차행렬
  - 행: 실제(음, 양), 열 : 예측(음,양)
  - TN(맞음, 음성예측), FP(틀림, 양성예측)
  - FN(틀림, 음성예측), TP(맞음, 양성예측)



- 정밀도 : (양성이라고 예측한 것 중 양성인 개수)/(양성이라고 예측한 것의 개수)
  - 스팸메일
- 재현율 : (양성 예측&실제 양성)/(실제 양성인 것의 개수)
  - 의료 질병

- F1점수 : 정밀도와 재현율의 조화 평균
  - 정밀도와 재현율이 비슷할 수록 F1 점수가 높아짐.
- **정밀도/재현율 트레이드오프** : 정밀도를 올리면 재현율이 줄고, 재현율을 올리면 정밀도가 줄어듦
  - 임계값을 높이면 양성이라고 예측하는 개수가 줄어듦. -> 정밀도가 커지고, 재현율이 작아짐
  - 임계값을 낮추면 양성이라고 예측하는 개수가 많아짐. -> 정밀도가 줄어들고, 재현율이 커짐.
  - 임계값을 높이더라도 정밀도가 줄어드는 현상이 가끔 발생하기 때문에 정밀도가 재현율보다 울퉁불퉁

식 3-3 F_1 점수

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

과제에 따라 적절한 정밀도/재현율을 고르는 것이 필요

In [11]:

```
# 교차 검증 구현(cross-val-score과 거의 같은 기능 함수)
from sklearn.model_selection import StratifiedKFold
# 클래스별 비율이 유지되도록 폴드를 만들고 계층적 샘플링을 수행

from sklearn.base import clone
# 데이터를 제외하고 모델만 복사(fit 하기 전에 모델 선언하는 곳까지)하는 함수

skfolds = StratifiedKFold(n_splits = 3, random_state=42)
# test set은 맨 마지막에 완성된 모델에 대한 성능 측정을 위해서만 쓰여져야 하기 때문에 train set으로

for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf) # 모델 복사

    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_folds = X_train[test_index]
    y_test_folds = y_train_5[test_index]

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_folds)
    n_correct = sum(y_pred==y_test_folds)
    print(n_correct/len(y_pred)) # 정확도로 성능 평가
```

```
C:₩ProgramData₩Anaconda3₩lib₩site-packages₩sklearn₩model_selection₩_split.py:296: Fu
tureWarning: Setting a random_state has no effect since shuffle is False. This will
raise an error in 0.24. You should leave random_state to its default (None), or set
shuffle=True.
  FutureWarning

0.95035
0.96035
0.9604
```

In [12]:

```
from sklearn.model_selection import cross_val_score
cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

Out[12]:

```
array([0.95035, 0.96035, 0.9604 ])
```

In [13]:

```
# 모든 데이터를 5가 아닌 것으로 예측하는 분류기
from sklearn.base import BaseEstimator

class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        return self # 아무 일도 하지 않음
    def predict(self, X):
        return np.zeros(X.shape[0], dtype=bool)
```

In [14]:

```
never5Classifier = Never5Classifier()
cross_val_score(never5Classifier, X_train, y_train_5, cv=5, scoring="accuracy")
```

Out[14]:

```
array([0.91266667, 0.90866667, 0.9095    , 0.90883333, 0.90858333])
```

In [15]:

```
# cross_val_score과 비슷하지만 여기선 결과로 예측값을 반환(모든 데이터에 대해 순서에 맞춰서)
from sklearn.model_selection import cross_val_predict

from sklearn.metrics import confusion_matrix # 오차행렬

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

Out[15]:

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

In [16]:

```
# 정밀도, 재현율
from sklearn.metrics import precision_score, recall_score
print(precision_score(y_train_5, y_train_pred)) # (3530)/(687+3530) # 정밀도
print(recall_score(y_train_5, y_train_pred)) # (3530)/(3530+1891) # 재현율
```

```
0.8370879772350012
0.6511713705958311
```

In [17]:

```
# 정밀도와 재현율 트레이드 오프
y_scores = sgd_clf.decision_function(X_train) # 각 데이터의 점수 확인
y_scores
threshold = int(input()) # 임계값
y_some_digit_pred = (y_scores>threshold) # 임계값을 기준으로 예측
```

```
80
```

In [18]:

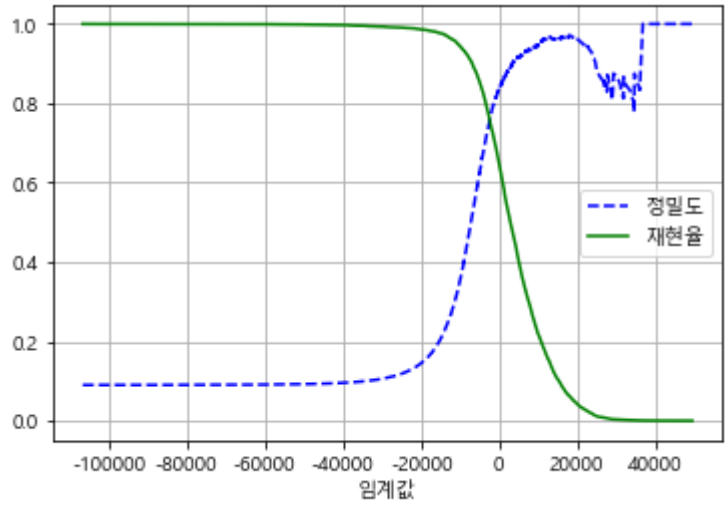
```
# cross_val_predict를 통하여 모든 데이터의 점수 확인
# 한번에 훈련 -> 한번에 예측하게 되면 예측에 대한 신뢰도가 떨어지기 때문에 cross_val_predict를 통하
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3, method="decision_function")

from sklearn.metrics import precision_recall_curve

# precision_recall_curve(실제 레이블, 각 데이터 점수)
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

In [19]:

```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    # recalls, precisions 모두 맨 마지막 값(그냥 1로 디폴트)은 제외해주어야 함
    plt.plot(thresholds, precisions[:-1], "b--", label="정밀도")
    plt.plot(thresholds, recalls[:-1], "g-", label="재현율")
    plt.xlabel("임계값")
    plt.legend()
    plt.grid()
```

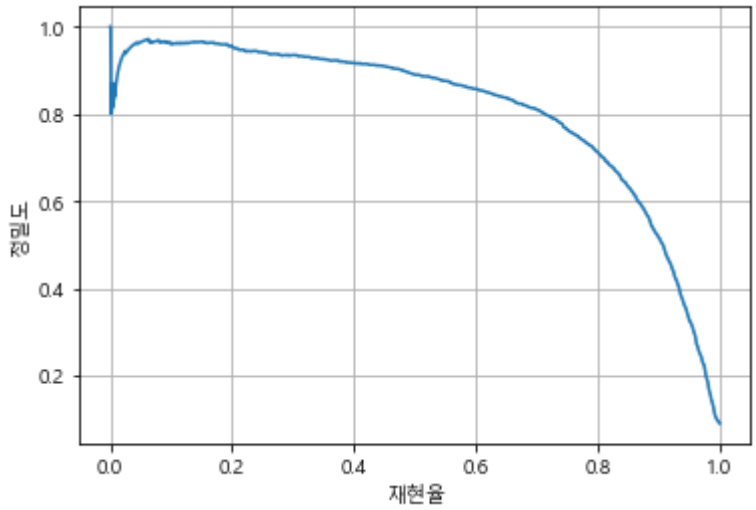
In [20]:

```
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
```



In [21]:

```
import seaborn as sns
# 재현율에대한 정밀도를 그리는 것이 좋음! (정밀도가 올라갔다 내려갔다 하는 구간이 있기 때문에)
sns.lineplot(x=recalls[:-1], y=precisions[:-1])
plt.xlabel("재현율")
plt.ylabel("정밀도")
plt.grid()
# 재현율 80% 근처에서 정밀도가 급격하게 줄어듦 -> 하강점 직전을 선택하는 것도 좋은 방법
# 이러한 선택은 프로젝트에 따라 달라짐
```



In [22]:

```
# 정밀도 90%를 달성하는 것이 목표
threshold_90_precision = thresholds[np.argmax(precisions>=0.9)]
threshold_90_precision # 정밀도 90%를 최초로 달성하게 하는 임계값

# 정밀도 90%를 달성하게 하는 임계값으로 예측
y_train_pred_90 = (y_scores>=threshold_90_precision)

# 정밀도와 재현율 확인
print("재현율", precision_score(y_train_5, y_train_pred_90))
print("정밀도", recall_score(y_train_5, y_train_pred_90))
```

재현율 0.9000345901072293
정밀도 0.4799852425751706

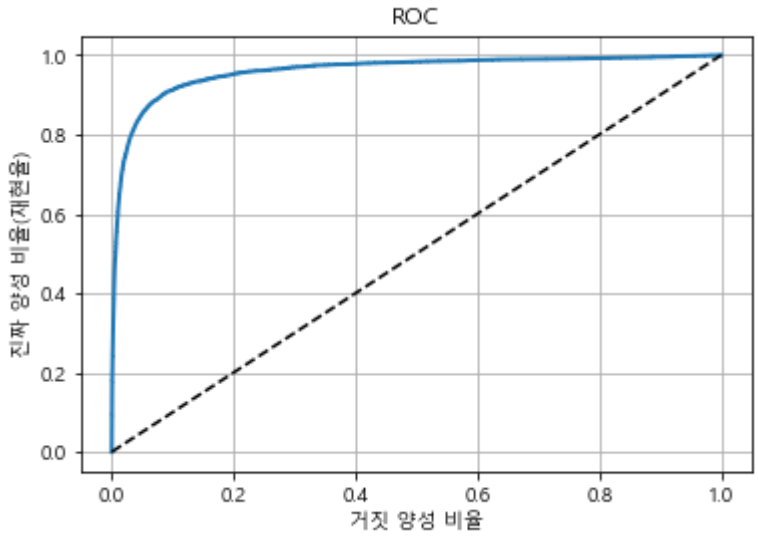
- ROC 곡선 : 거짓양성비율에 대한 진짜양성 비율, 좋은 분류기일수록 아래 면적이 1(직사각형)
  - 거짓양성비율 : 1 - 특이도
    - (양성으로 잘못 예측)/(전체 음성개수)
    - FP/(TN+FP)
  - 진짜양성비율 : 재현율
    - TP/(FN+TP)
  - 특이도 : (음성으로 잘 예측)/(전체 음성 개수)
    - TN/(TN+FP)

In [23]:

```
from sklearn.metrics import roc_curve

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

In [24]:

```
def plot_roc_curve(fpr,tpr, label=None):
    plt.plot(fpr,tpr, linewidth=2, label=label)
    plt.plot([0,1],[0,1], "k--") # 대각점선 # 완전한 랜덤 분류기
    plt.xlabel("거짓 양성 비율")
    plt.ylabel("진짜 양성 비율(재현율)")
    plt.grid()
    plt.title("ROC")

plot_roc_curve(fpr,tpr)
plt.show()
```



In [25]:

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

Out[25]:

0.9604938554008616

양성 클래스가 드물거나, 거짓 음성보다 거짓 양성이 더 중요할 때 PR 곡선을 사용

- 양성 클래스가 드물면 AUC점수가 잘나옴
- y축이 재현율이기 때문에 거짓 음성에 신경씀 -> 거짓 양성이 더 중요할 때는 PR 곡선

In [26]:

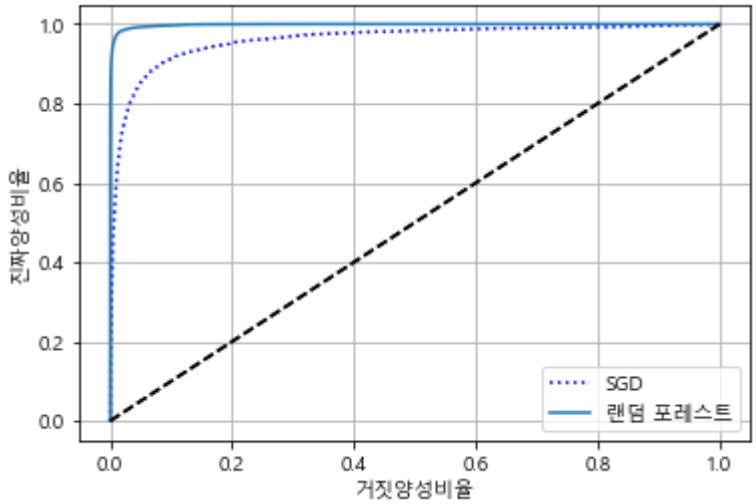
```
# randomforest를 이용한 ROC, AUC
from sklearn.ensemble import RandomForestClassifier

forest_clf = RandomForestClassifier(random_state = 42)
# 예측라벨이 아닌 예측 확률 반환, dicision_function()이 아닌 predict_proba()
# (음성일 확률, 양성일 확률) -> 2차원 배열 반환
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3, method = "predict_proba")
```

In [27]:

```
y_scores_forest = y_probas_forest[:, 1] # 양성클래스에 대한 확률을 점수로 사용
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5, y_scores_forest)
```

In [28]:

```
# SGD와 랜덤 포레스트 ROC 곡선 비교
plt.plot(fpr, tpr, "b:", label="SGD")
plt.plot(fpr_forest, tpr_forest, label = "랜덤 포레스트")
plt.legend(loc = "lower right")
plt.grid()
plt.xlabel("거짓양성비율")
plt.ylabel("진짜양성비율")
plt.plot([[0,0],[1,1]], "k--")
plt. show()

# 랜덤 포레스트의 ROC 곡선이 왼쪽 위 모서리에 더 가깝기 때문에 랜덤 포레스트가 더 좋음
```



In [29]:

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

Out[29]:

0.9983436731328145

다중분류

- 이진 분류기(SVM/logistic Regression 등은 이진분류만 가능) 를 이용하여 다중 분류 시스템을 만드는 방법
  - OvA : 특정 클래스일 확률을 구하여 가장 높은 클래스를 선택 (others)
  - OvO : 각 숫자의 두개 조합마다 이진 분류기를 학습, -> 가장 많이 분류된 클래스 선택 (SVM)
    - 훈련시에 두 클래스에 해당하는 샘플만 필요
- 다중 클래스 분류 작업에 이진 분류 알고리즘을 선택하면 사이킷런이 자동으로 OvR 또는 OvO 실행

In [30]:

```
from sklearn.svm import SVC
svm_clf = SVC()
svm_clf.fit(X_train, y_train) # 다중분류이기 때문에 y_train_5가 아닌 y_train 사용
svm_clf.predict(X_train[0])

some_digit_scores = svm_clf.decision_function(X_train[0])
some_digit_scores

label_idx = np.argmax(some_digit_scores) # 예측 label의 index값, 여기서는 label 값과도 같음
svm_clf.classes_[label_idx]

from sklearn.multiclass import OneVsRestClassifier
ovr_clf = OneVsRestClassifier(SVC(gamma="auto", random_state=42))
ovr_clf.fit(X_train[:1000], y_train[:1000])
ovr_clf.predict([some_digit])
```

Out[30]:

```
'₩nfrom sklearn.svm import SVC₩nsvm_clf = SVC()₩nsvm_clf.fit(X_train, y_train) # 다
중분류이기 때문에 y_train_5가 아닌 y_train 사용₩nsvm_clf.predict(X_train[0])₩n₩nsome
_digit_scores = svm_clf.decision_function(X_train[0])₩nsome_digit_scores₩n₩nlabel_id
x = np.argmax(some_digit_scores) # 예측 label의 index값, 여기서는 label 값과도 같음
₩nsvm_clf.classes_[label_idx]₩n'
```

에러분석

- 에러 종류를 살펴봄으로써 모델의 성능을 향상시킬 수 있음

In [31]:

```
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train, cv=3)
conf_mx = confusion_matrix(y_train, y_train_pred)
conf_mx # 다중 분류에 따른 오차행렬을 반환함
```

Out[31]:

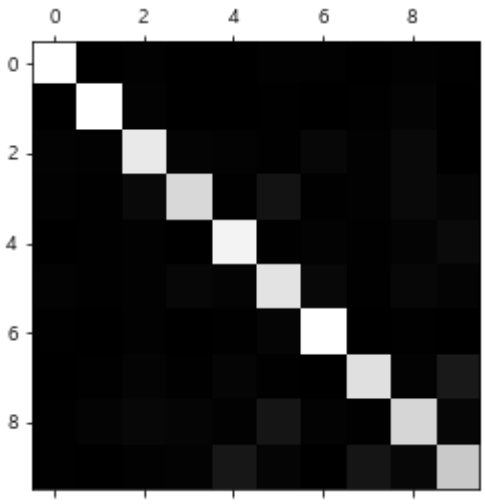
```
array([[5635,    0,   61,   10,   16,   50,   46,    7,   66,   32],
       [   3, 6393,   95,   21,   16,   47,   15,   27,  109,   16],
       [  72,   56, 5174,   89,   69,   39,  163,   66,  212,   18],
       [  58,   32,  217, 4941,   23,  441,   32,   56,  216,  115],
       [  11,   26,   46,    6, 5298,   26,   73,   32,   87,  237],
       [  68,   23,   58,  150,   83, 4606,  174,   26,  152,   81],
       [  40,   13,   56,    6,   22,  113, 5625,    5,   36,    2],
       [  23,   24,  103,   36,  124,   40,   10, 5228,   75,  602],
       [  40,  101,  158,  122,   49,  457,   77,   35, 4666,  146],
       [  33,   18,   66,   83,  515,  127,    4,  485,  166, 4452]],
      dtype=int64)
```

In [41]:

```
# 대각선이 해당 레이블에 대해 정확히 예측한 비율이기 때문에 색이 밝을 수록 더 잘 예측한 것.
# 색이 어두울 수록 잘 예측하지 못한 것에 해당
row_sums = conf_mx.sum(axis=1,keepdims=True) # 2차원을 그대로 유지한채로 더함
norm_conf_mx = conf_mx/row_sums
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
```

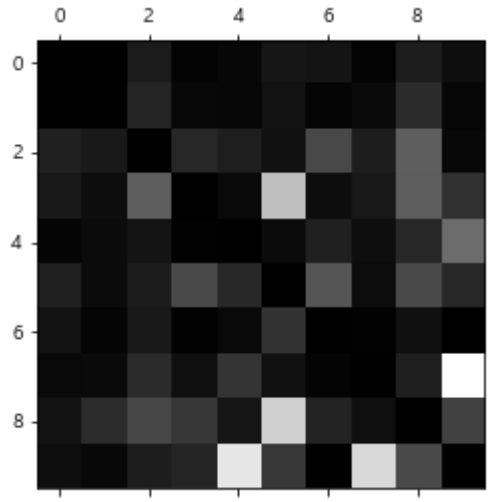
Out[41]:

<matplotlib.image.AxesImage at 0x2560b484348>



In [33]:

```
# 다른 항목은 유지하고, 주대각선만 0으로 채움.
# 어떤 레이블로 많이 잘못 예측되는지를 파악할 수 있음
# 8로 잘못예측되는 경우가 많기 때문에 8처럼 보이지만 실제로 8은 아닌 숫자의 훈련 데이터로 학습

np.fill_diagonal(norm_conf_mx, 0) # 해당 행렬의 주 대각선을 0으로 채움 -> 그래야만 다른 값들이 잘 보
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()
```



In [34]:

```
# 이미지는 위치나 회전 방향에 매우 민감.
# 따라서 이미지를 중앙에 위치시키고 회전되어 있지 않도록 전처리함으로써 성능 향상 가능
# 비선형관계를 인식할 수 있는 딥러닝으로 갈 수록 이러한 민감도가 줄어듦
```

다중 레이블 분류

- 하나의 데이터에 대해 여러 타겟값을 뱉어내야 하는 경우가 있을 수 있음
- ex) 하나의 사진에서 (앨리스가 있는지, 철수가 있는지, 마이클이 있는지)
- 평가 지표는 각 프로젝트에 따라 다를 수 있음(f1 등등)
- 지지도(타겟 레이블에 속한 샘플 수)를 가중치로 할 수도 있음
  - 해당 타겟 레이블에 속한 샘플이 많다는 것은 해당 타겟 레이블이 대표성을 띈다는 것이기 때문

In [35]:

```
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] # 열방향으로 이어붙임

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

Out[35]:

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

In [36]:

```
knn_clf.predict([X_train[0]])
```

Out[36]:

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

In [ ]:

```
y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_multilabel)
from sklearn.metrics import f1_score
# 각 레이블의 f1점수/recall/recision/accuracy를 계산 후, 가중치/평균으로 전체 평가 지표를 계산
f1_score(y_multilabel, y_train_knn_pred, average="macro") # 모든 레이블의 가중치를 같게 둔 것
f1_score(y_multilabel, y_train_knn_pred, average="weighted")
```

다중 출력 분류

- 다중 레이블 분류에서 한 레이블이 다중 클래스가 될 수 있도록 일반화한 것.
- 이 때 각각의 레이블이 회귀가 될 수도, 클래스가 될 수도 있음
- 잡음이 많은 MNIST이미지를 입력으로 받고, 깨끗한 숫자 이미지를 MNIST의 데이터처럼 출력하는 경우

In [49]:

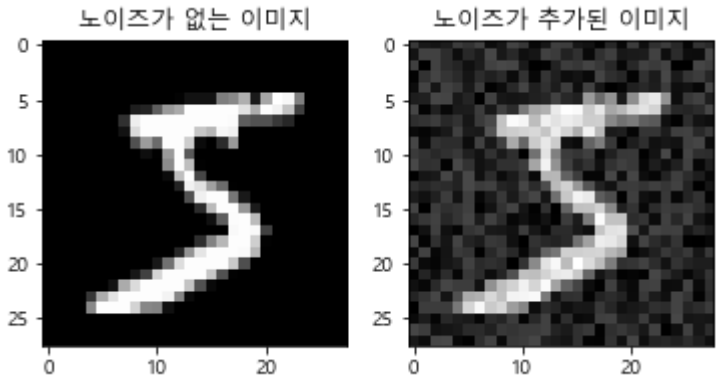
```
# MNIST 데이터에 인위적으로 noise를 첨가
noise = np.random.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise
noise = np.random.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise
y_train_mod = X_train
y_test_mod = X_test
```

In [58]:

```
num = random.randint(0,9) # 0~9의 int중 하나를 랜덤으로 선택
idx = np.argmax(y_train==num)
plt.subplot(1,2,1)
plt.imshow(X_train[idx].reshape(28,28), cmap="gray") # noise가 추가되지 않은 데이터
plt.title("노이즈가 없는 이미지")
plt.subplot(1,2,2)
plt.imshow(X_train_mod[idx].reshape(28,28), cmap="gray") # noise가 추가된 데이터
plt.title("노이즈가 추가된 이미지")
```

Out[58]:

```
Text(0.5, 1.0, '노이즈가 추가된 이미지')
```



In [59]:

```
# noise가 제거된 깨끗한 이미지가 출력됨
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[idx]])
```

In [64]:

```
plt.subplot(1,2,1)
plt.imshow(X_test_mod[idx].reshape(28,28), cmap="gray") # noise가 추가되지 않은 데이터
plt.title("X(노이즈가 있는 학습 전 데이터)")
plt.subplot(1,2,2)
plt.imshow(y_test_mod[idx].reshape(28,28), cmap="gray") # noise가 추가된 데이터
plt.title("Y(노이즈가 제거 된 학습 후 데이터)")
```

Out[64]:

```
Text(0.5, 1.0, 'Y(노이즈가 제거 된 학습 후 데이터)')
```