MYONGJI UNIVERSITY

Classification(분류)
- Part2 -

Outline

- 다중 클래스 분류
- 에러 분석
- 다중 레이블 분류
- 다중 출력 분류

다중 클래스 분류

다중 클래스 분류기(multiclass classifier)

- 주어진 샘플이 3가지 이상의 클래스들 중 어느 클래스에 해당하는지를 예측
- 다항 분류기(multinomial classifier)라고도 부름

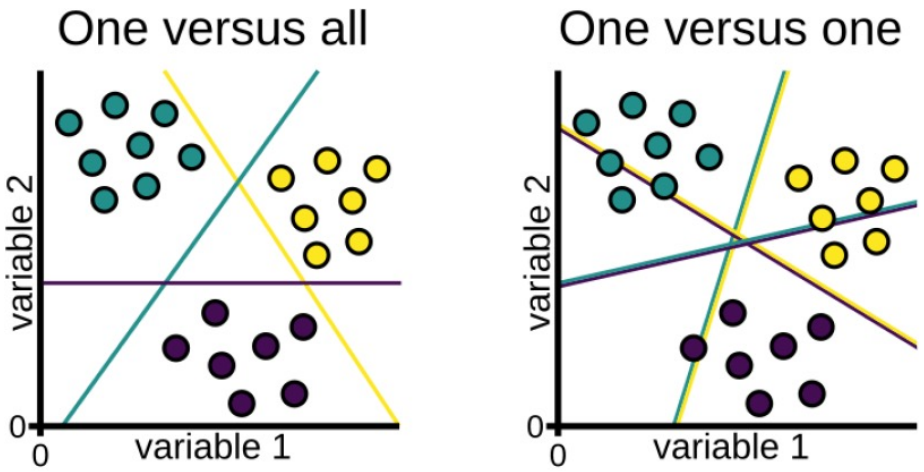
- 예: 손글씨 숫자 이미지 분류의 경우 주어진 손글씨 이미지가 0부터 9까지 10개의 클래스들 중 어느 클래스에 해당하는지를 예측해야 함

- 다중 클래스 분류를 기본으로 지원하는 사이킷런 분류기 예
  - LogisticRegression
  - RandomForestClassifier
  - GaussianNB

- 이진 분류만을 지원하는 분류기
  - SGDClassifier
  - SVC (Support Vector Machine classifier)
- 이진 분류기 여러 개를 사용하여 다중 클래스 분류를 수행할 수 있는 방법들이 존재함

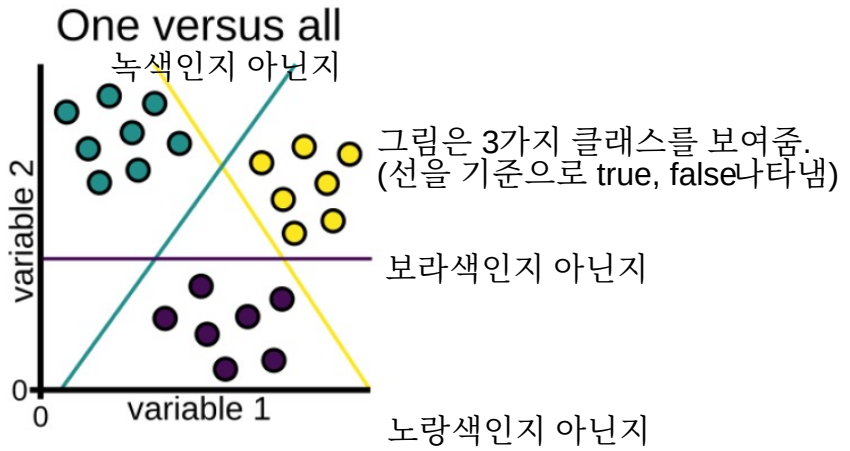
이진 분류기 여러 개를 활용한 다중 클래스 분류 전략

- 일대다: OvR(one-versus-the-rest) 또는 OvA(one-versus-all)이라고 불림
- 일대일: OvO(one-versus-one)



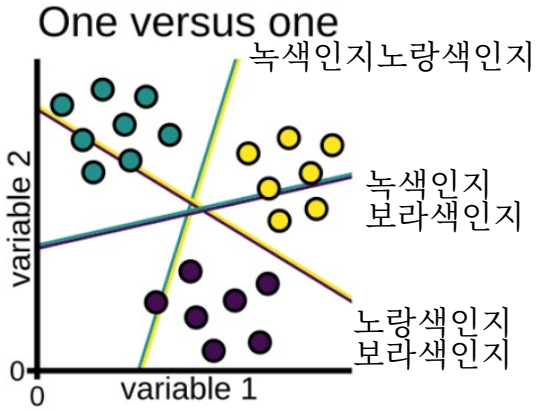
일대다 방식(OvR)을 통한 다중 클래스 분류 예

- 0~9 손글씨 숫자 이미지 분류 예
  - 5-detector 모델(5, or not)에 적용했던 이진 분류 방식을 0~9까지 각 숫자 클래스에 대해 동일하게 적용 모든 숫자에 대해 이진분류기 적용즉 각 쑤자별로 긴지 아닌지 해당하는것 찾기
  - 0-detector, 1-detector, … , 9-detector 까지 총 10개 이진 분류기를 훈련
  - 새로운 이미지 샘플이 주어지면 10개 이진 분류기 각각을 실행하여 decision score를 얻는다.
  - 가장 높은 score가 관찰된 감지기의 클래스로 예측
  - 예를 들어 주어진 이미지에 대해 2-detector(숫자2-감지기 모델)의 decision score가 가장 높게 나왔다면 숫자 2 이미지로 예측 10개 모두 돌려서 가장 점수가 높을 것으로 예측



일대일 방식(OvO)을 통한 다중 클래스 분류 예

- 모든 가능한 2가지 클래스 조합($\binom{N}{2}$, N: 클래스 수) 각각에 대해 이진 분류기를 하나씩 훈련시키는 방법 nC2인것 (콤비네이션) 즉 n(n-1)/2
- 0~9 손글씨 숫자 이미지 분류 예
  - 0-1 분류기, 0-2 분류기, …, 1-2 분류기, 1-3 분류기, …, 8-9 분류기 등, N=10 이므로 총 $\binom{10}{2}$ = 45개 이진 분류기가 필요함 0-1이든 1-0이든 순서는 상관없으므로 콤비네이션
  - 새로운 이미지 샘플이 주어지면 45개 이진 분류기 실행하여 가장 많이 예측된 클래스로 최종 예측
  - 예: 숫자 1 클래스와 관련된 총 9개 분류기(0-1, 1-2, 1-3, 1-4, 1-5, 1-6, 1-7, 1-8, 1-9 분류기) 모두 숫자 1로 예측했다면 숫자 1로 최종 예측 중복된 집합?
- 장점:
  - 각 분류기의 훈련에 전체 훈련셋 중 구별할 두 클래스에 해당하는 샘플만 필요함



따라서, 일대일이 필요한 이진분류기는 일대다보다 더 많음.
그러나 훈련셋의 사이클은 일대다는 전체를 다 해야되나 일대일은 해당 분류기에 맞는 클래스들만 (0-1분류기면 0,1이미지 샘플에 대해서만) 진행가능.

즉 3C2만큼 3만큼 필요

- SVM(Support Vector Machine) 같은 일부 머신러닝 알고리즘은 훈련셋 크기에 민감함
  - **큰 훈련셋으로 적은 수의 분류기를 훈련**시키는 것보다 **작은 훈련셋으로 많은 수의 분류기를 훈련시키는 쪽이 더 빠름**.
  - 따라서 **훈련셋 크기에 민감한 알고리즘에 대해서는 OvO 전략이 선호됨**

일대일 전략(작은크기의 훈련셋으로 많은 분류기로 분류)

- 하지만 대부분의 이진 분류 알고리즘에서는 OvR 전략을 선호함

SVM Classifier 예

- 사이킷런의 SVC 클래스를 사용하여 이진 분류기인 SVM 분류기 훈련
  - (y_train_5가 아닌) 0~9까지의 원래 타깃 특성(y_train)을 사용해 훈련
  - 처음 2000개 훈련 샘플들만으로 훈련(전체 훈련셋으로 훈련시킬 경우 시간이 많이 걸림)
  - 내부적으로 OvO 전략을 사용해 45개 이진 분류기를 훈련시킴

이전엔 6만개였으나 너무 오래걸리므로 2천으로 줄여서 진행

```
from sklearn.svm import SVC

svm_clf = SVC(random_state=42)
svm_clf.fit(X_train[:2000], y_train[:2000])
```

  - 주어진 새로운 샘플(some_digit, 숫자5에 해당)에 대해 45개 이진 분류기를 실행하여 클래스별 결정 점수를 얻는다. 점수가 가장 높은 클래스를 최종 예측값으로 결정(e.g., 숫자 5 클래스)

```
>>> some_digit_scores = svm_clf.decision_function([some_digit])
>>> some_digit_scores.round(2)
array([[ 3.79,  0.73,  6.06,  8.3 , -0.29,  9.3 ,  1.75,  2.77,  7.21,
         4.82]])
```

0~9까지의 decision score를 나타냄. 가장 높은 값이 최종 예측값일꺼임.

```
>>> class_id = some_digit_scores.argmax()
>>> class_id
5
```

- 분류기가 훈련될 때 classes_ 속성에 타깃 클래스들의 리스트를 값을 기준으로 정렬하여 저장함

```
>>> svm_clf.classes_  훈련샘플들의 속성알기
array(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'], dtype=object)
>>> svm_clf.classes_[class_id]
'5'
```

- 사이킷런에서 OvO나 OvR을 사용하도록 강제하려면 OneVsOneClassifier나 OneVsRestClassifier를 사용함
- 예: SVC 기반으로 OvR 전략을 사용하는 다중 분류기
  - 훈련

```
from sklearn.multiclass import OneVsRestClassifier

ovr_clf = OneVsRestClassifier(SVC(random_state=42))
ovr_clf.fit(X_train[:2000], y_train[:2000])
```

명시해서 바꿔줌.

  - 예측

```
>>> ovr_clf.predict([some_digit])
array(['5'], dtype='<U1')
>>> len(ovr_clf.estimators_)
10
```

Colab 확인

- SGDClassifier 사용 예
  - 다중 클래스 데이터셋으로 (이진분류기) SGDClassifier를 훈련시킬 경우 OvR 방식이 적용됨
  - 훈련 및 예측

```
>>> sgd_clf = SGDClassifier(random_state=42)
>>> sgd_clf.fit(X_train, y_train)
>>> sgd_clf.predict([some_digit])
array(['3'], dtype='<U1')
```

숫자3으로 예측하였으며
예측이 틀린 경우에 해당

  - 클래스 별 결정 점수 확인

```
>>> sgd_clf.decision_function([some_digit]).round()
array([[-31893., -34420.,  -9531.,   1824., -22320.,  -1386., -26189.,
        -16148.,  -4604., -12051.]])
```

클래스 0~9까지

교차 검증을 통한 분류기 성능 평가

- 분류기 성능 평가에는 일반적으로 교차 검증을 사용함
- 0~9 손글씨 숫자 이미지 분류 문제의 경우 **클래스 별 이미지 수가 균등하기 때문에 accuracy를 지표로 사용해도 무방함** 균형이 잘 잡혀있는 골고루 있어서 accuracy 사용

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([0.87365, 0.85835, 0.8689 ])
```

- StandardScaler() 사용하여 특성 스케일링을 적용하면 성능이 향상됨

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
>>> X_train_scaled = scaler.fit_transform(X_train.astype("float64"))
>>> cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
array([0.8983, 0.891 , 0.9018])
```

에러 분석

- 실제 머신러닝 프로젝트라면
  - 데이터 분석 및 전처리 수행
  - 여러 종류의 머신러닝 모델을 시도해봄
  - 성능이 우수한 몇가지 모델을 추린 다음 그리드 탐색, 랜덤 탐색 등을 통한 하이퍼파라미터 튜닝 수행

- 여기서는 위 과정을 통해 가장 성능이 좋은 모델을 하나 찾았다고 가정하자.
  - sgd_clf(SGDClassifier 객체)를 최종 선택된 모델로 가정한다.

- 에러 분석: 최종적으로 선택된 모델이 어떤 종류의 예측 오류를 범하는지를 분석함으로써 모델의 성능을 추가적으로 향상시킬 방법을 모색

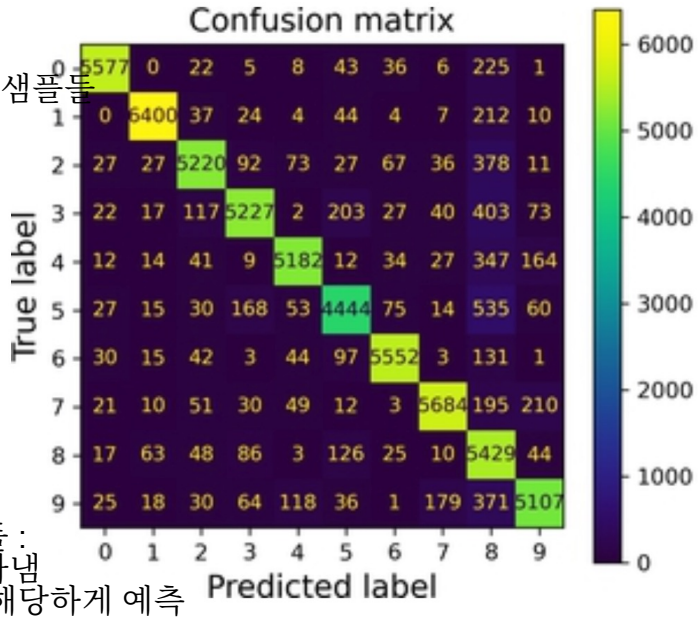
오차 행렬(Confusion Matrix)

- 오차 행렬에 관한 colored diagram 생성
  - 교차 검증을 통해 훈련 샘플들에 대한 분류기의 예측 결과를 생성(e.g., y_train_pred)
  - 분류기의 예측 값과 실제 타깃 데이터를 입력으로 오차 행렬 생성

```
from sklearn.metrics import ConfusionMatrixDisplay

y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred)
plt.show()
```

행의 대한 레이블들 :
실제 그값에 해당하는 샘플들의 예측결과 (True)

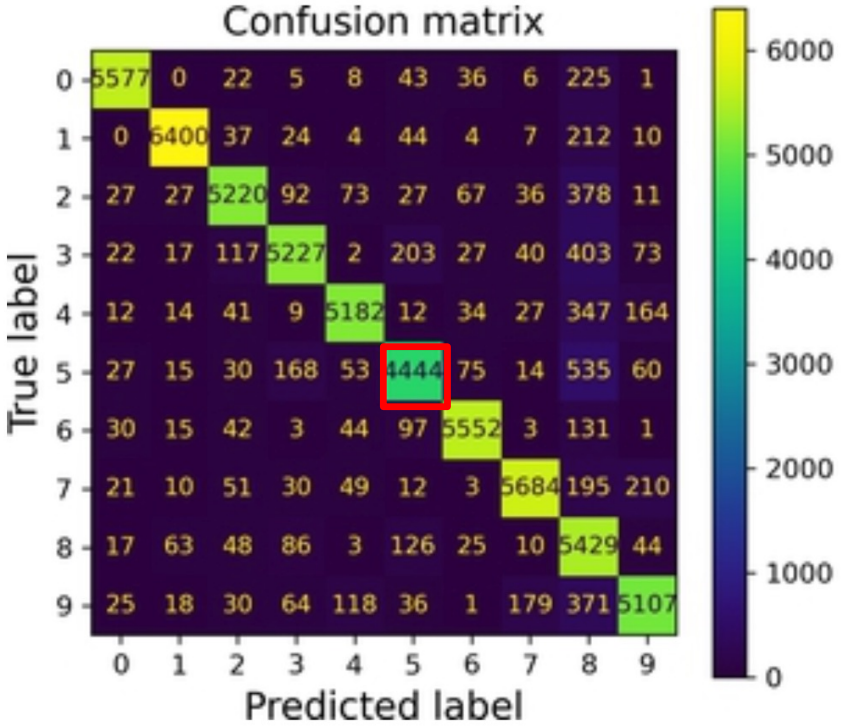


열의 대한 레이블들 :
예측한 결과를 나타냄
예측기가 그 값에 해당하게 예측한 결과

오차 행렬(Confusion Matrix)

- 오른쪽 그림: 각 행 별로 퍼센티지 합이 100이 되도록 정규화 적용

```
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred,
                                        normalize="true", values_format=".0%")
plt.show()
```



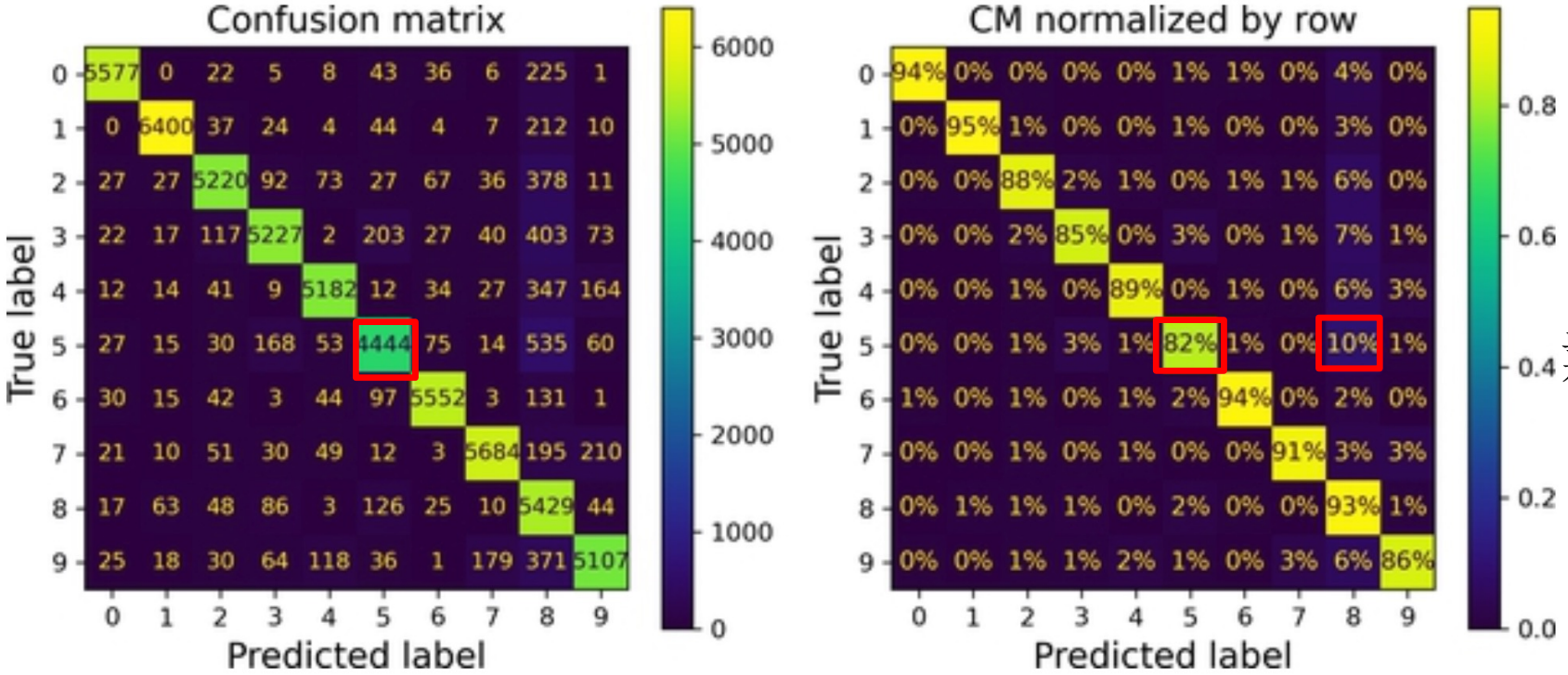
5번 행이 상대적으로 어두운 것에 대한 가능한 이유:

- 이는 숫자5 이미지에 대한 분류 정확도가 상대적으로 낮기 때문일수도 있고
- 아니면 데이터셋에 숫자5 이미지 수 자체가 적기 때문일 수도 있음.
  즉 말그대로 예측정확도가 낮거나 샘플자체수가 적거나
- 이에 대한 원인을 명확하게 파악하기 위해서는 **오차 행렬에 대한 정규화가 필요**함

오차 행렬(Confusion Matrix)

- 오른쪽 그림: 각 행 별로 퍼센티지 합이 100이 되도록 정규화 적용

```
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred,
                                        normalize="true", values_format=".0%")
plt.show()
```

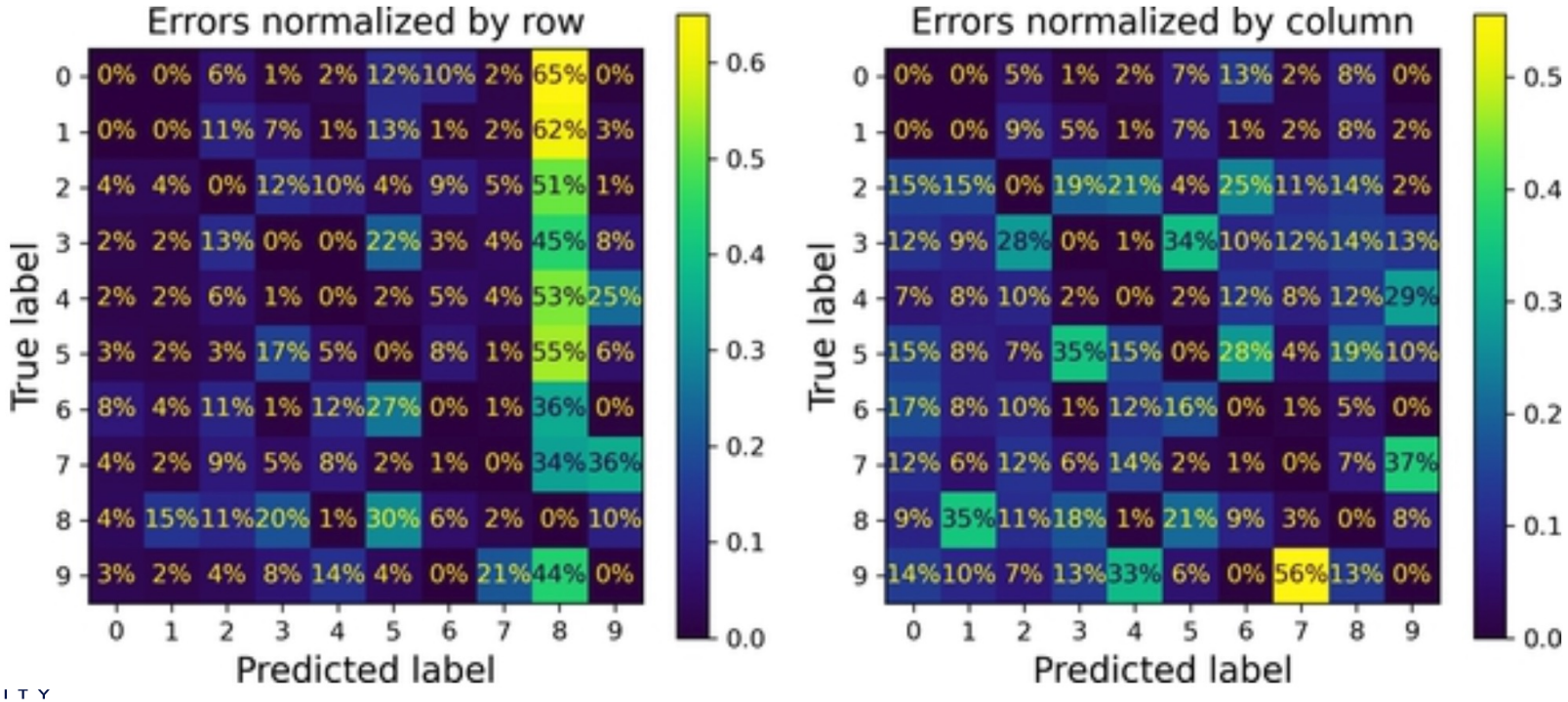


특히 5인데 8로 예측한 경
가 10퍼로 엄청 많음.

비율로 나타냈을 때 82퍼로 걍 예측성능이 낮은거 (다른 숫자들보다 성능이 떨

오차 행렬(Confusion Matrix)

- 올바른 분류 케이스들은 제외하고 잘못 분류된 케이스들만에 대한 비율 분석
  - 왼쪽: **각 행 별 퍼센티지 합이 100이 되도록 정규화** 되었음. **숫자8로 잘못 분류되는 비율이 가장 높음**. 예를 들어 실제 숫자7 이미지들 중 모델의 예측이 숫자7이 아닌, 즉 잘못 예측한 샘플들 중 34%가 8로 잘못 예측된 케이스들이었으며, 36%가 9로 잘못 예측된 케이스들이었음을 의미.
  - 오른쪽: **각 컬럼 별 퍼센티지 합이 100이 되도록 정규화** 한 다이어그램. 예를 들어 컬럼 7을 보면, 7로 잘못 분류된 케이스들 중 56%가 실제 숫자9 이미지들이었음을 의미.



비율로 봤을때 숫자8에 해당하는 부분에 분류가 가장 오류가 많음.

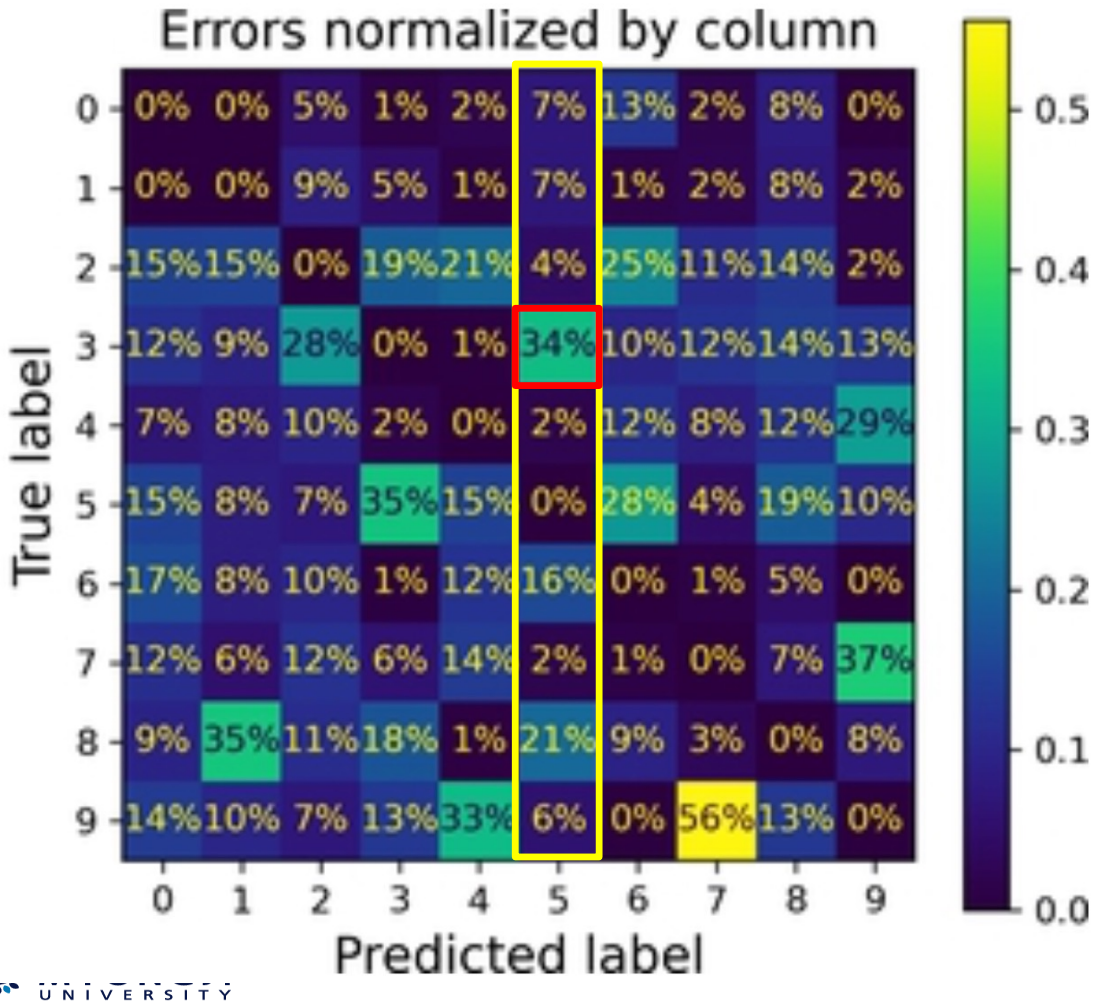
오차 행렬(Confusion Matrix)

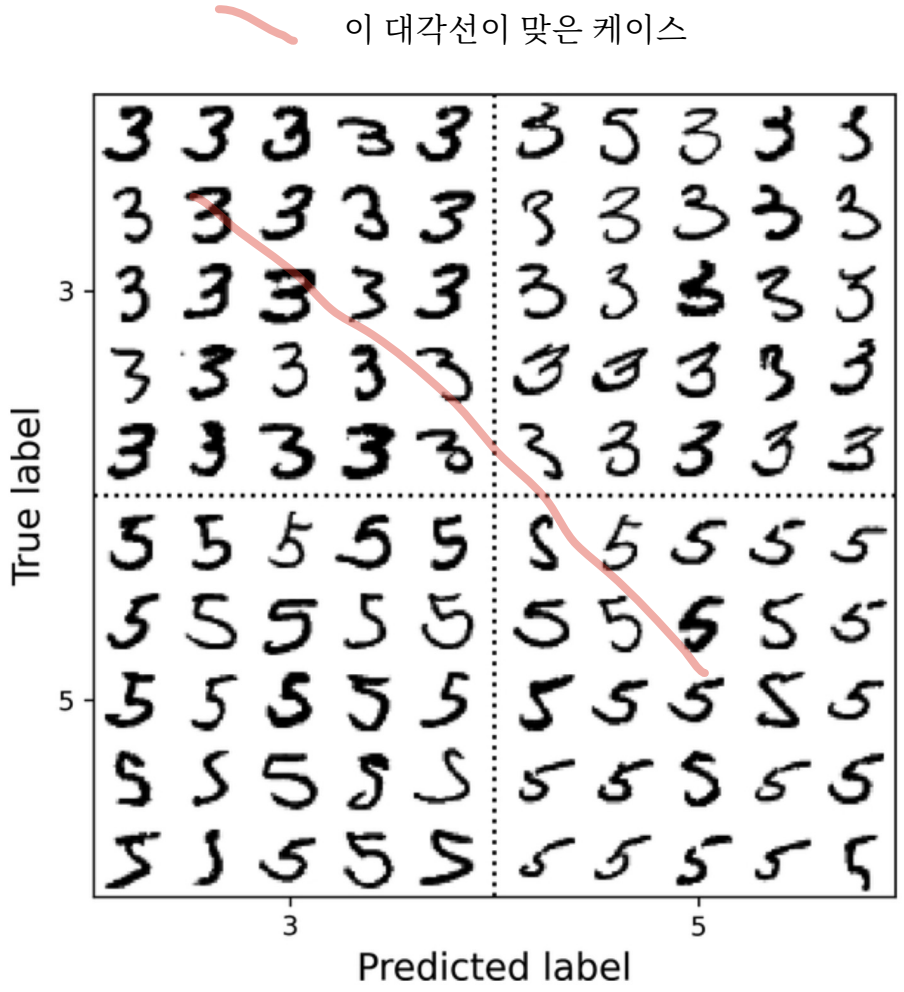
- 오차 행렬을 분석하면 분류기의 성능 향상 방안에 대한 통찰을 얻을 수 있음.
  - 분류기의 예측 오류가 주로 어떤 케이스들에 대해서 발생하는지를 확인할 수 있음

- 8로 잘못 분류되는 케이스들을 줄이도록 분류기를 개선할 필요가 있어보임
- 실제 8이 아니지만 8처럼 보이는 이미지 샘플들을 더 많이 모아서 실제 8과 구분하도록 분류기를 추가적으로 학습시킬 수 있음
- 분류기에 도움이 될 만한 특성을 찾아서 추가하는 방안도 생각해볼 수 있음
  - 예를 들어 숫자 이미지에 포함된 동심원의 수를 카운트 해서 특성으로 추가 (8은 2개, 6은 1개, 5는 0개 등)

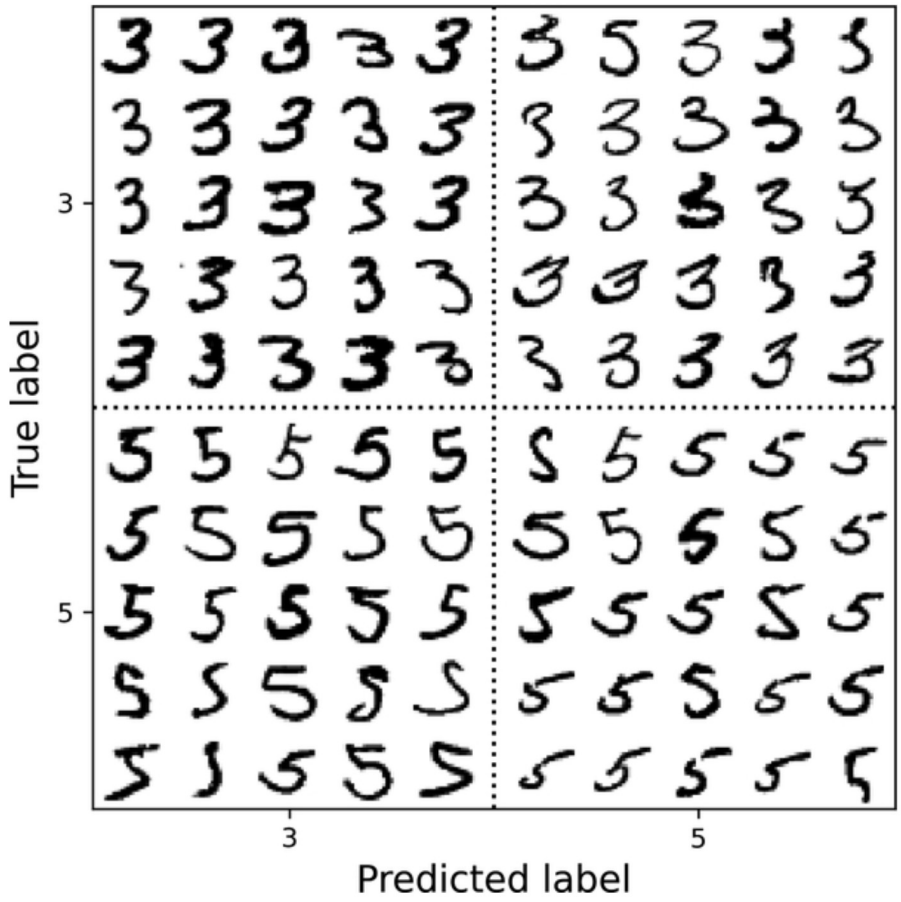
뭐가됐든 예측에 도움이 될 수 있도록 좀 더 학습시켜야 할 필요가 있음.

숫자3과 5 이미지들에 대한 개별 오류 확인





숫자3과 5 이미지들에 대한 개별 오류 확인



- 분류기가 잘못 분류한 이미지들 중 일부는 정말 잘못 쓰여 있어서 사람도 분류하기가 어려워 보임
- 하지만 대부분의 잘못 분류된 이미지들은 확실히 예측 에러인 것으로 보임
- 그 원인은 단순한 형태의 선형 모델인 SGDClassifier를 사용했기 때문으로 예측 성능이 떨어지는 것으로 추정된다.

데이터 증식(Data Augmentation)

- 보다 좋은 성능의 모델을 사용할 수도 있지만 기본적으로 **보다 많은 훈련 이미지가 필요**하다.

- 새로운 이미지를 구할 수 있으면 좋지만 일반적으로 매우 어렵다. 반면에 **기존 데이터셋의 이미지를 조금씩 회전하거나, 뒤집거나, 이동시키는 방식 등으로 생성된 이미지를 훈련셋에 추가**할 수 있음. 새로운 이미지보단 기존 것들을 변형시켜 쌀먹하자

- 이를 **데이터 증식(data augmentation)**이라고 부름

다중 클래스 분류 일반화

- 다중 레이블 분류(multilabel classification)
- 다중 출력 분류(multioutput classification)

주어진 샘플에 대한 예측결과로 두개이상의 결과를 주는 것이 다중 레이블,출력

다중 레이블 분류

- 지금까지는 각 샘플에 대한 예측 결과로 하나의 클래스만 출력되었음
  - Given an image sample x_i, output the predicted class (e.g., 0, … or 9) of x_i.

- 하지만 **하나의 샘플에 대한 예측 결과로 여러 개의 타깃/레이블들에 대한 예측 결과를 출력**해야 하는 경우도 있음
- 예: Alice, Bob, Charlie 세 사람의 얼굴을 인식하도록 훈련된 분류기를 가정
  - Alice와 Charlie 둘만 포함된 사진이 주어진다면 분류기는 [T, F, T]를 출력해야함. 즉 Alice 있음, Bob 없음, Charlie 있음을 의미 사진의 예측결과로 얼굴이 맞는지 아닌지의 여부에 따라 T,F출력

- 하나의 샘플에 대해 **여러 개의 binary tag를 출력하는 분류 시스템을 다중 레이블 분류 시스템**이라고 함
  - 즉, 각 레이블 당 가능한 출력 값은 True or False

다중 레이블 분류 예

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier

y_train_large = (y_train >= '7')
y_train_odd = (y_train.astype('int8') % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]

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

- MNIST 훈련 데이터셋의 숫자 이미지 각각에 대한 2개의 타깃 레이블을 포함하는 y_multilabel 배열 생성 총 레이블을 두개로 넣었다.
  - y_train_large: 이미지 속 숫자가 7 이상인지 여부에 대한 타깃 레이블 특성(True or False)
  - y_train_odd: 이미지 속 숫자가 홀수인지 여부에 대한 타깃 레이블 특성(True or False)
- 다중 레이블 분류를 지원하는 KNN classifier 이용
- KNN classifier를 각 샘플 당 2개의 타깃 레이블 값이 포함된 y_multilabel 배열을 이용하여 훈련

다중 레이블 분류 예

- 훈련된 KNN classifier를 이용하여 some_digit 이미지에 대한 예측 수행

```
>>> knn_clf.predict([some_digit])
array([[False,  True]])
```

some_digit: 숫자 5에 해당하는 데이터들(이미지들)

즉 7이 아니라 5이기때문에 전자는 False,후자는 홀수이기때문에 true

다중 출력 분류(Multioutput Classification)

- 다중 레이블 분류에서 **각 레이블 당 가능한 클래스가 3개 이상**이 될 수 있도록 일반화한 것
- **다중 출력 다중 클래스(Multioutput Multiclass) 분류**라고도 불림

- 예: 주어진 이미지로부터 노이즈를 제거하는 모델
  - 다중 레이블: 이미지를 구성하는 각 픽셀이 모델이 예측해야 하는 레이블 하나에 해당. 즉, 모델 실행 결과로 픽셀 수만큼의 예측 값이 출력됨 784개의 노이즈가 포함된 데이터를 넣어주면
  - 다중 클래스: 각 픽셀에 대한 레이블 값은 0~255 중 하나임. 즉, 총 256개 클래스

각각의 픽셀값에 대한 클래스는 256개

이진 분류기, 다중 레이블 분류기, 다중출력 분류기 어떻게 다른지 확인하기

다중 출력 분류 예: 이미지 노이즈 제거 모델

- MNIST 데이터셋 이미지로부터 노이즈가 추가된 이미지 샘플들을 생성하여 훈련셋 및 테스트셋 구성

```
np.random.seed(42)  # to make this code example reproducible
noise = np.random.randint(0, 100, (len(X_train), 784)) 노이즈 훈련셋 추가 (784개의 랜덤값)
X_train_mod = X_train + noise
noise = np.random.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise 노이즈가 포함된 테스트용 모델
y_train_mod = X_train 타겟 특성을 노이즈가 추가되기전 원본이미지로 잡음.
y_test_mod = X_test
```

  - 훈련셋: X_train_mod(입력 데이터, 노이즈가 추가된 훈련 이미지 샘플들), y_train_mod(타깃 데이터, 노이즈 추가 전 원본 이미지 샘플들(X_train))

  - 테스트셋: X_test_mod(입력 데이터, 노이즈가 추가된 테스트용 이미지 샘플들), y_test_mod(타깃 데이터, 노이즈 추가 전 원본 테스트셋 이미지 샘플들(X_test))

다중 출력 분류 예: 이미지 노이즈 제거 모델

- 노이즈가 추가된 이미지 및 원본 이미지 예

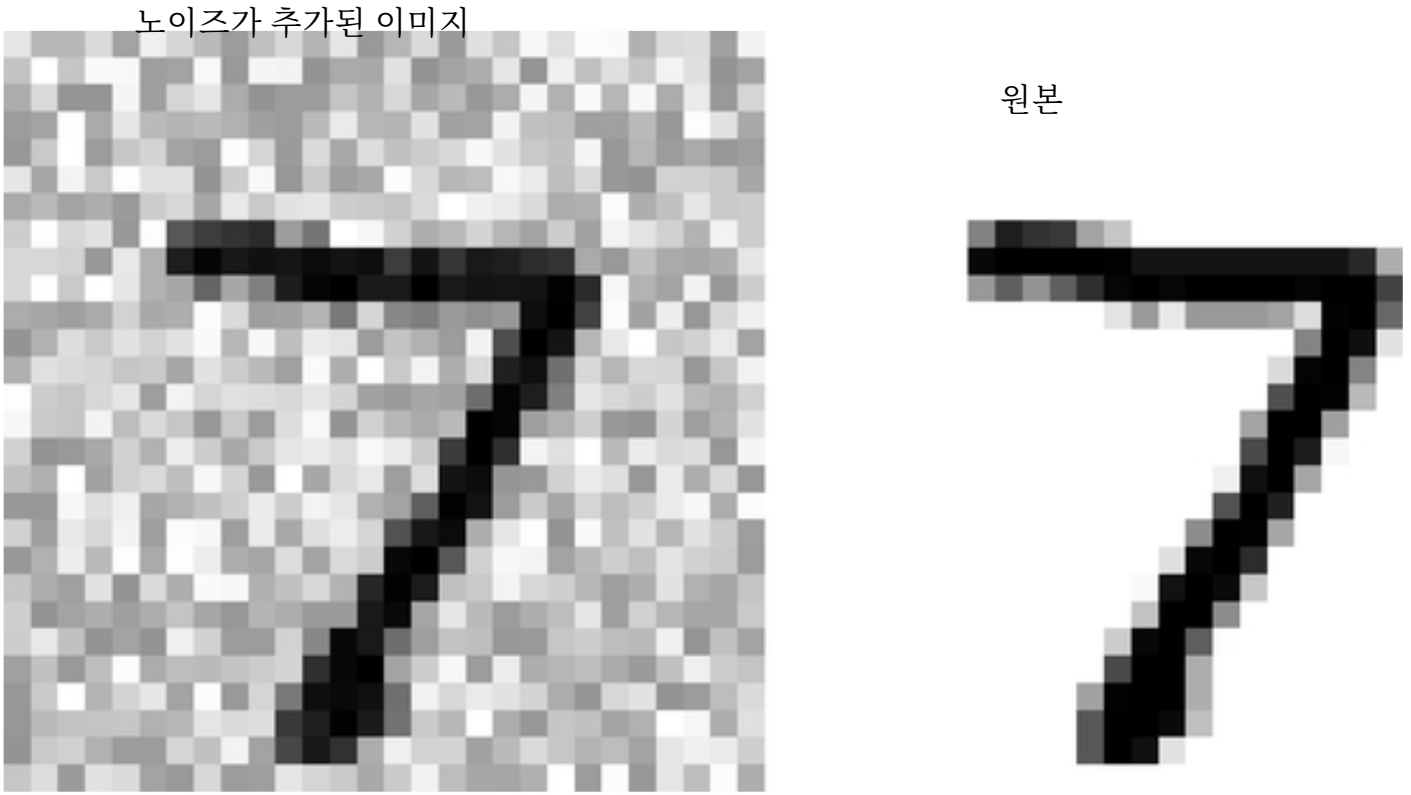


Figure 3-12. A noisy image (left) and the target clean image (right)

다중 출력 분류 예: 이미지 노이즈 제거 모델

- 노이즈 제거를 위한 KNN 분류기 모델 훈련

```
knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train_mod, y_train_mod)
```

- 노이즈가 추가된 테스트셋 이미지 샘플 X_test_mod[0]에 대해 훈련된 분류기 모델 실행 및 예측 결과 이미지 plotting

```
clean_digit = knn_clf.predict([X_test_mod[0]])
plot_digit(clean_digit)
plt.show()
```