Intro to classification

CSCI-P 556

ZORAN TIGANJ

Reminders/Announcements

- Don't forget the quiz deadline today

Today: Intro to classification

- After regression example, now we will cover a classification example
- We will use MNIST dataset, which is a set of 70,000 small images of digits handwritten by high school students and employees of the US Census Bureau. Each image is labeled with the digit it represents.

```
>>> from sklearn.datasets import fetch_openml
>>> mnist = fetch_openml('mnist_784', version=1)
>>> mnist.keys()
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'details',
           'categories', 'url'])

>>> X, y = mnist["data"], mnist["target"]
>>> X.shape
(70000, 784)
>>> y.shape
(70000,)
```

MNIST dataset

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

some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)

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

```
>>> y[0]
'5'
```

Examples of MNIST digits

Figure 3-1. Digits from the MNIST dataset

Training a Binary Classifier

- Let's simplify the problem for now and only try to identify one digit—for example, the number 5.
- This "5-detector" will be an example of a binary classifier, capable of distinguishing between just two classes, 5 and not 5.

```
y_train_5 = (y_train == 5)  # True for all 5s, False for all other digits
y_test_5 = (y_test == 5)

from sklearn.linear_model import SGDClassifier

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

```
>>> sgd_clf.predict([some_digit])
array([ True])
```

Performance Measures

- Let's do cross-validation:

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3,
scoring="accuracy")
array([0.96355, 0.93795, 0.95615])
```

corss_val_score did K-fold cross-validation which means splitting the training set into K folds (in this case, three), then making predictions and evaluating them on each fold using a model trained on the remaining folds.

Performance Measures

- Well, before you get too excited, let's look at a very dumb classifier that just classifies every single image in the "not-5" class:

```
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)
```

```
>>> never_5_clf = Never5Classifier()
>>> cross_val_score(never_5_clf, X_train, y_train_5, cv=3,
scoring="accuracy")
```

Performance Measures

- Well, before you get too excited, let's look at a very dumb classifier that just classifies every single image in the "not-5" class:

```
from sklearn.base import BaseEstimator

class Never5Classifier(BaseEstimator):
```

This demonstrates why accuracy is generally not the preferred performance measure for classifiers, especially when you are dealing with skewed datasets (i.e., when some classes are much more frequent than others)

```
>>> cross_val_score(never_5_clf, X_train, y_train_5, cv=3,
scoring="accuracy")

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

Confusion Matrix

- A much better way to evaluate the performance of a classifier is to look at the confusion matrix.
- The general idea is to count the number of times instances of class A are classified as class B.
  - For example, to know the number of times the classifier confused images of 5s with 3s, you would look in the fifth row and third column of the confusion matrix.

```
from sklearn.model_selection import cross_val_predict

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

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_train_5, y_train_pred)
array([[53057,  1522],
       [ 1325,  4096]])
```

Confusion Matrix

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_train_5, y_train_pred)
array([[53057,  1522],
       [ 1325,  4096]])
```

- Each row in a confusion matrix represents an actual class, while each column represents a predicted class.
- The first row of this matrix considers non-5 images (the negative class): 53,057 of them were correctly classified as non-5s (they are called true negatives), while the remaining 1,522 were wrongly classified as 5s (false positives).
- The second row considers the images of 5s (the positive class): 1,325 were wrongly classified as non-5s (false negatives), while the remaining 4,096 w ere correctly classified as 5s (true positives).

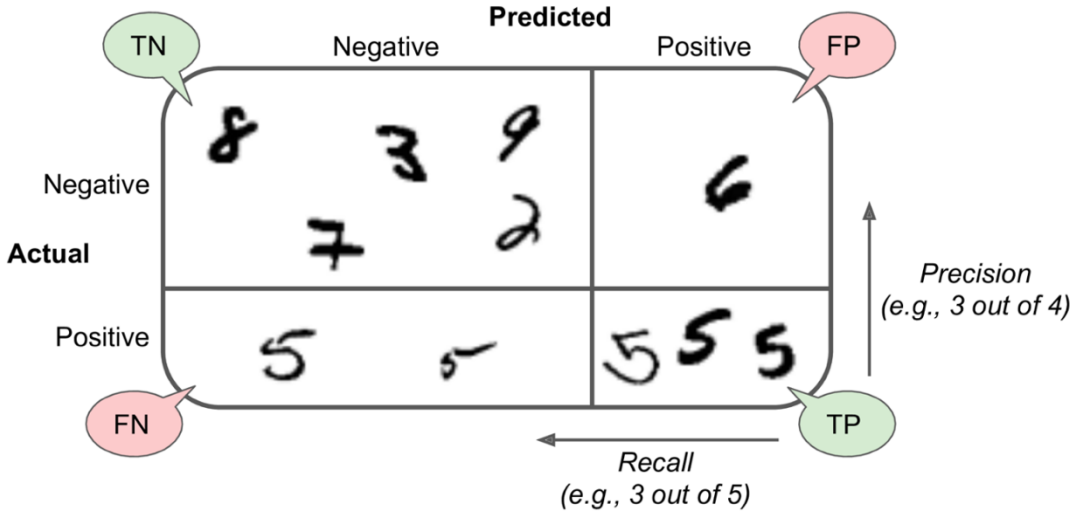


Figure 3-2. An illustrated confusion matrix shows examples of true negatives (top left), false positives (top right), false negatives (lower left), and true positives (lower right)

Confusion Matrix

```
>>> y_train_perfect_predictions = y_train_5  # pretend we reached
perfection
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)
array([[54579,     0],
       [    0,  5421]])
```

Precision of the classifier

- The confusion matrix gives you a lot of information, but sometimes you may prefer a more concise metric.
- An interesting one to look at is the accuracy of the positive predictions; this is called the **precision of the classifier**

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

  - TP is the number of true positives,
  - FP is the number of false positives.

Sensitivity

- Precision is typically used along with another metric named **recall**, also called **sensitivity** or the true positive rate (TPR): the ratio of positive instances that are correctly detected by the classifier

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

  - FN is the number of false negatives.

Confusion matrix - illustration

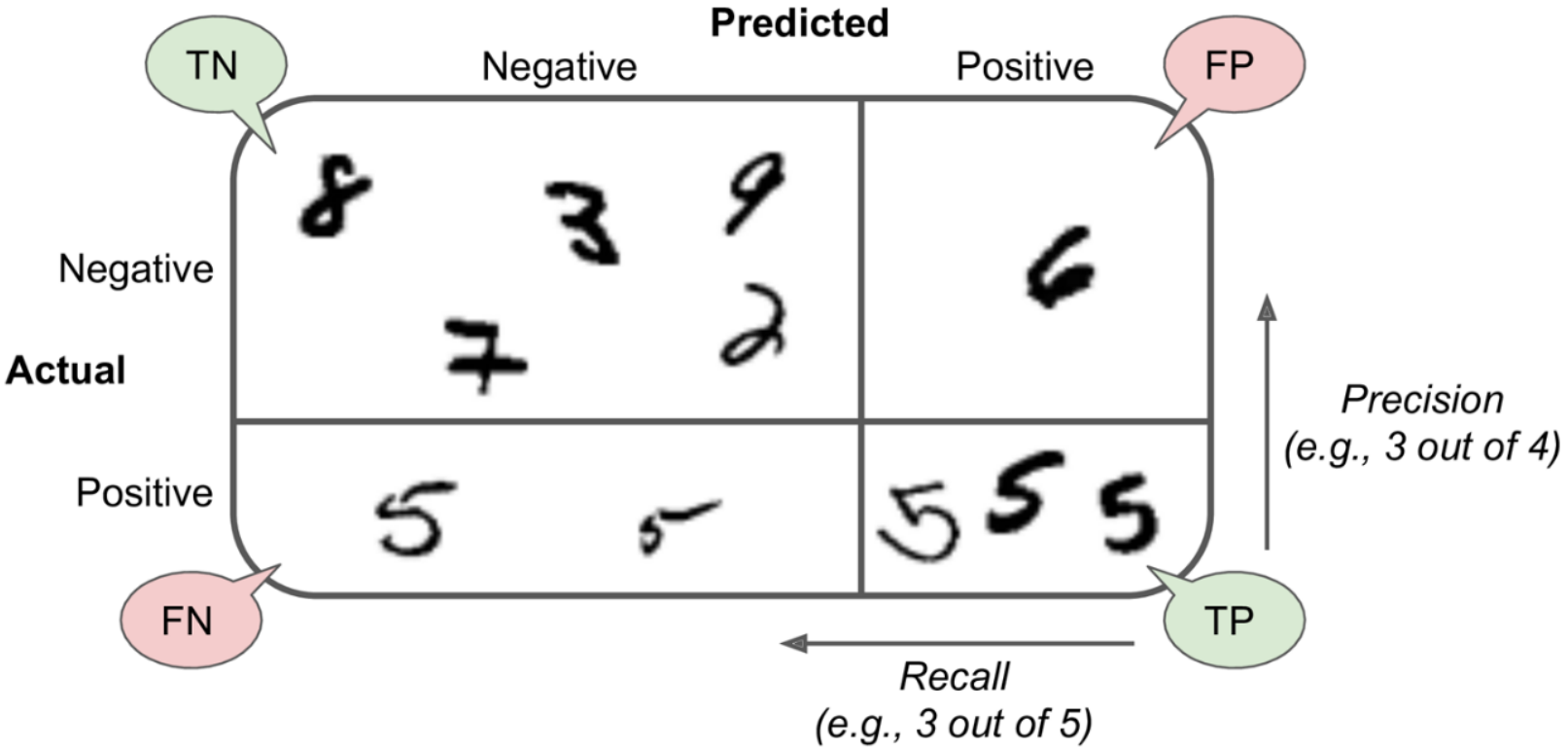


Figure 3-2. An illustrated confusion matrix shows examples of true negatives (top left), false positives (top right), false negatives (lower left), and true positives (lower right)

Precision and recall of 5-detector

- Scikit-Learn provides several functions to compute classifier metrics, including precision and recall:

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1522)
0.7290850836596654
>>> recall_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1325)
0.7555801512636044
```

F_1 score

- It is often convenient to combine precision and recall into a single metric called the F_1 score, in particular if you need a simple way to compare two classifiers.
- The F score is the harmonic mean of precision and recall. Whereas the regular mean treats all values equally, the harmonic mean gives much more weight to low values.
- As a result, the classifier will only get a high F score if both recall and precision are high.

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

F_1 score of 5-detector

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.7420962043663375
```

Precision/Recall Trade-off

- The F1 score favors classifiers that have similar precision and recall.
- This is not always what you want: in some contexts you mostly care about precision, and in other contexts you really care about recall.
  - For example, if you trained a classifier to detect videos that are safe for kids, you would probably prefer a classifier that rejects many good videos (low recall) but keeps only safe ones (high precision).
  - On the other hand, suppose you train a classifier to detect shoplifters in surveillance images: it is probably fine if your classifier has only 30% precision as long as it has 99% recall (sure, the security guards will get a few false alerts, but almost all shoplifters will get caught).
- Unfortunately, you can't have it both ways: increasing precision reduces recall, and vice versa. This is called the **precision/recall trade-off**.

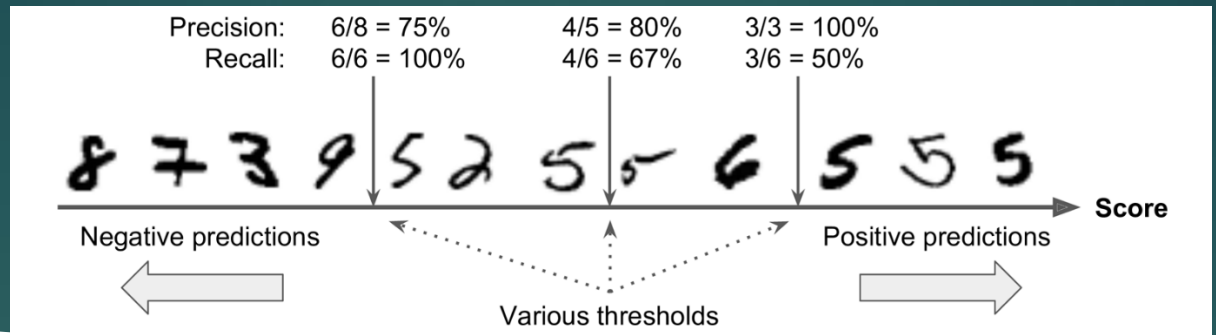


Figure 3-3. In this precision/recall trade-off, images are ranked by their classifier score, and those above the chosen decision threshold are considered positive; the higher the threshold, the lower the recall, but (in general) the higher the precision

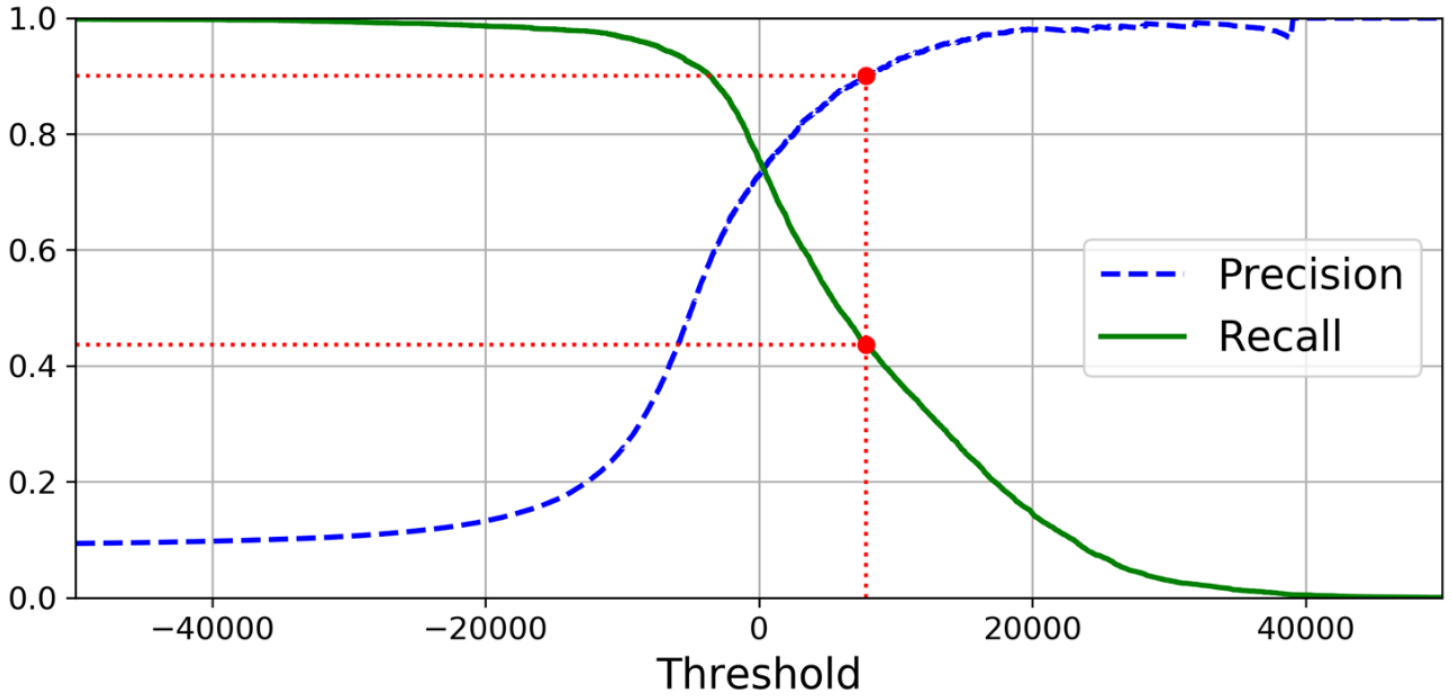


Figure 3-4. Precision and recall versus the decision threshold

The receiver operating characteristic (ROC) curve

- The receiver operating characteristic (ROC) curve is another common tool used with binary classifiers.
- It is very similar to the precision/recall curve, but instead of plotting precision versus recall, the ROC curve plots the true positive rate (another name for recall) against the false positive rate (FPR).
- The FPR is the ratio of negative instances that are incorrectly classified as positive. It is equal to 1 – the true negative rate (TNR), which is the ratio of negative instances that are correctly classified as negative.
- The TNR is also called specificity. Hence, the ROC curve plots sensitivity (recall) versus 1 – specificity .

```
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--') # Dashed diagonal
    [...] # Add axis labels and grid

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

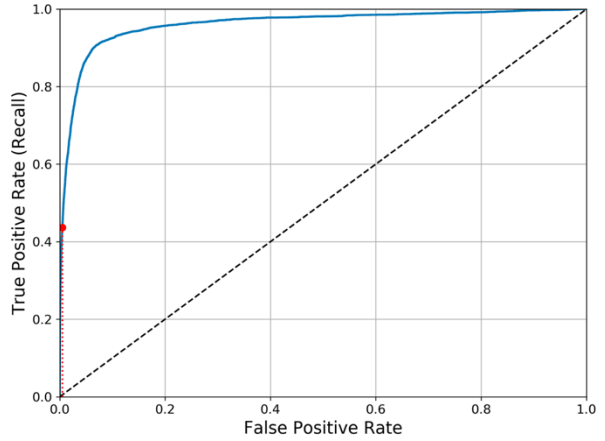


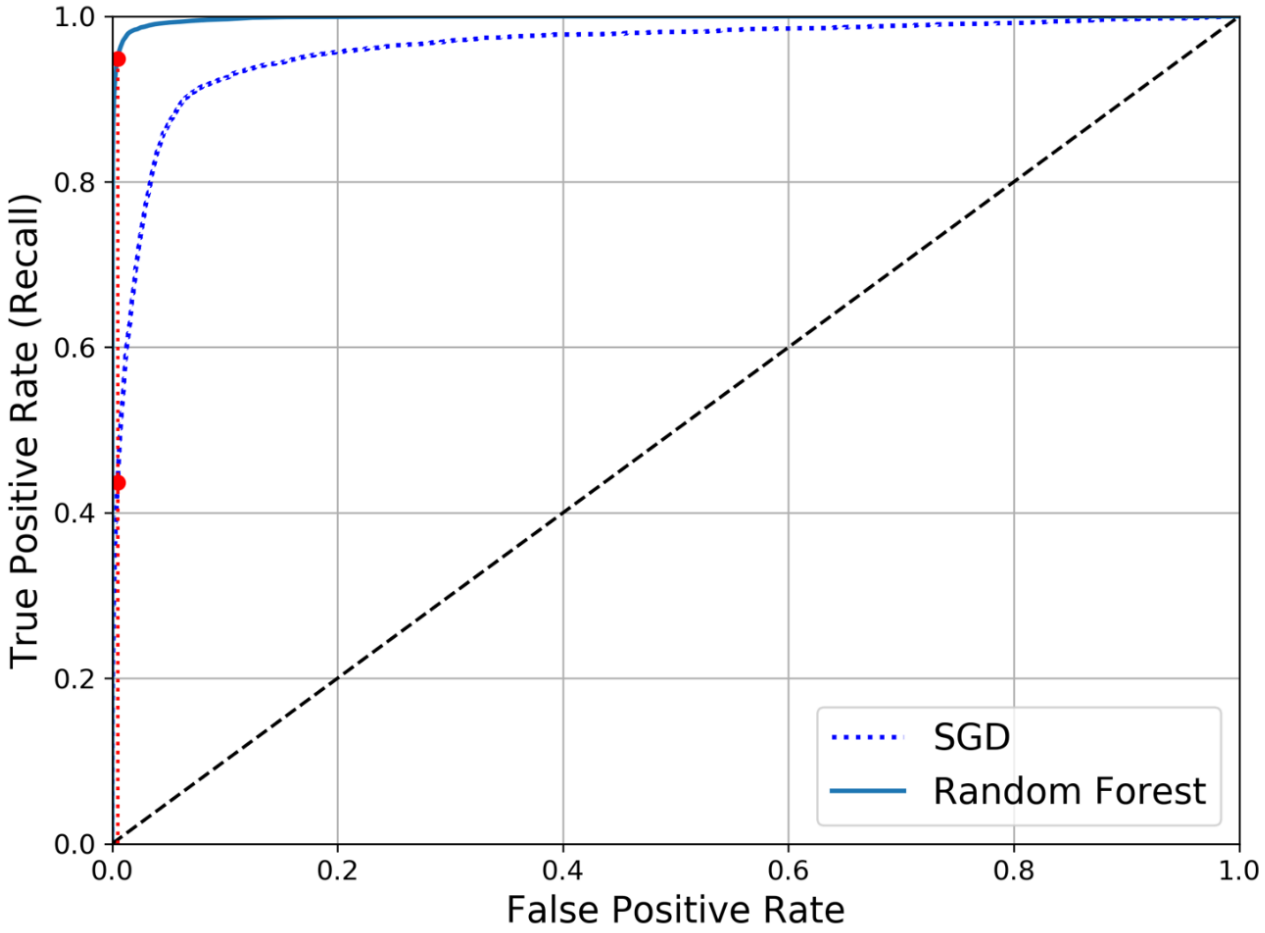
Figure 3-6. This ROC curve plots the false positive rate against the true positive rate for all possible thresholds; the red circle highlights the chosen ratio (at 43.68% recall)

Area under the curve (AUC)

- One way to compare classifiers is to measure the area under the curve (AUC).
- A perfect classifier will have a ROC AUC equal to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5.
- Scikit-Learn provides a function to compute the ROC AUC:

```
>>> from sklearn.metrics import roc_auc_score
>>> roc_auc_score(y_train_5, y_scores)
0.9611778893101814
```

Comparing different classifiers using ROC



OvR vs OvO

- Some algorithms (such as SGD classifiers, Random Forest classifiers, and naive Bayes classifiers) are capable of handling multiple classes natively.
- Others (such as Logistic Regression or Support Vector Machine classifiers) are strictly binary classifiers. However, there are various strategies that you can use to perform multiclass classification with multiple binary classifiers.
  - One way to create a system that can classify the digit images into 10 classes (from 0 to 9) is to train 10 binary classifiers, one for each digit (a 0-detector, a 1-detector, a 2-detector, and so on). This is called the one-versus-the-rest (OvR) strategy (also called one-versus-all).

OvR vs OvO

- Some algorithms (such as SGD classifiers, Random Forest classifiers, and naive Bayes classifiers) are capable of handling multiple classes natively.
- Others (such as Logistic Regression or Support Vector Machine classifiers) are strictly binary classifiers. However, there are various strategies that you can use to perform multiclass classification with multiple binary classifiers.
  - Another strategy is to train a binary classifier for every pair of digits: one to distinguish 0s and 1s, another to distinguish 0s and 2s, another for 1s and 2s, and so on. This is called the one-versus-one (OvO) strategy.
  - The main advantage of OvO is that each classifier only needs to be trained on the part of the training set for the two classes that it must distinguish.

Multilabel Classification

- Consider a face-recognition classifier: what should it do if it recognizes several people in the same picture?
  - It should attach one tag per person it recognizes. Say the classifier has been trained to recognize three faces, Alice, Bob, and Charlie.
  - Then when the classifier is shown a picture of Alice and Charlie, it should output [1, 0, 1] (meaning "Alice yes, Bob no, Charlie yes").
  - Such a classification system that outputs multiple binary tags is called a **multilabel classification system**.

```
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)
```

Next time

- Linear regression, from Chapter 4 from *Hands on machine learning textbook*