

Chapter 3 分类

Setup

```
In [1]: import sys

        assert sys.version_info >= (3, 7)
```

```
In [2]: from packaging import version
        import sklearn

        assert version.parse(sklearn.__version__) >= version.parse("1.0.1")
```

```
In [3]: import matplotlib.pyplot as plt

        plt.rc('font', size=14)
        plt.rc('axes', labelsz=14, titlesz=14)
        plt.rc('legend', fontsize=14)
        plt.rc('xtick', labelsz=10)
        plt.rc('ytick', labelsz=10)
```

```
In [4]: from pathlib import Path

        IMAGES_PATH = Path() / "images" / "classification"
        IMAGES_PATH.mkdir(parents=True, exist_ok=True)

        def save_fig(fig_id, tight_layout=True, fig_extension="png", resolution=300):
            path = IMAGES_PATH / f"{fig_id}.{fig_extension}"
            if tight_layout:
                plt.tight_layout()
            plt.savefig(path, format=fig_extension, dpi=resolution)
```

1. MNIST

在本章中，我们将使用MNIST数据集，这是一组由高中生和美国人口普查局的雇员手写的7万个数字的小图像。每幅图像都用它所代表的数字进行标记。这组已经研究了很多，它通常被称为机器学习的“Hello World”：每当人们想出一个新的分类算法，他们好奇它在MNIST上的执行情况，任何学习机器学习迟早处理这个数据集。

Scikit-Learn提供了许多辅助功能来下载流行的数据集。MNIST就是其中之一。下面的代码从 OpenML.org 获取MNIST数据集

```
In [5]: from sklearn.datasets import fetch_openml

        mnist = fetch_openml('mnist_784', as_frame=False)
```

sklearn.datasets 包主要包含三种类型的函数：

1. **fetch_*** 函数，如 **fetch_openml()** 下载真实数据集
2. **load_*** 函数，加载与Scikit-Learn捆绑的小玩具数据集（所以它们就不需要在互联网上下载）
3. **make_*** 函数,生成假数据集，这对测试很有用。

生成的数据集通常作为一个包含输入数据和标签的 **(X, y)** 元组返回，两者都作为 NumPy 数组返回。其他数据集作为 **sklearn.utils.Bunch objects** 返回，这是字典，其条目也可以作为属性访问。它们通常包含以下条目：

1. **DESCR**: 对数据集的描述
2. **data**: 输入数据，通常作为一个2D NumPy数组
3. **target**: 标签，通常是一个1D NumPy数组

```
In [6]: # extra code - it's a bit too long
print(mnist.DESCR)
```

```
**Author**: Yann LeCun, Corinna Cortes, Christopher J.C. Burges
**Source**: [MNIST Website] (http://yann.lecun.com/exdb/mnist/) - Date unknown
**Please cite**:
```

```
The MNIST database of handwritten digits with 784 features, raw data available at: http://yann.lecun.com/exdb/mnist/. It can be split in a training set of the first 60,000 examples, and a test set of 10,000 examples
```

```
It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting. The original black and white (bilevel) images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. the images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field.
```

```
With some classification methods (particularly template-based methods, such as SVM and K-nearest neighbors), the error rate improves when the digits are centered by bounding box rather than center of mass. If you do this kind of pre-processing, you should report it in your publications. The MNIST database was constructed from NIST's NIST originally designated SD-3 as their training set and SD-1 as their test set. However, SD-3 is much cleaner and easier to recognize than SD-1. The reason for this can be found on the fact that SD-3 was collected among Census Bureau employees, while SD-1 was collected among high-school students. Drawing sensible conclusions from learning experiments requires that the result be independent of the choice of training set and test among the complete set of samples. Therefore it was necessary to build a new database by mixing NIST's datasets.
```

```
The MNIST training set is composed of 30,000 patterns from SD-3 and 30,000 patterns from SD-1. Our test set was composed of 5,000 patterns from SD-3 and 5,000 patterns from SD-1. The 60,000 pattern training set contained examples from approximately 250 writers. We made sure that the sets of writers of the training set and test set were disjoint. SD-1 contains 58,527 digit images written by 500 different writers. In contrast to SD-3, where blocks of data from each writer appeared in sequence, the data in SD-1 is scrambled. Writer identities for SD-1 is available and we used this information to unscramble the writers. We then split SD-1 in two: characters written by the first 250 writers went into our new training set. The remaining 250 writers were placed in our test set. Thus we had two sets with nearly 30,000 examples each. The new training set was completed with enough examples from SD-3, starting at pattern # 0, to make a full set of 60,000 training patterns. Similarly, the new test set was completed with SD-3 examples starting at pattern # 35,000 to make a full set with 60,000 test patterns. Only a subset of 10,000 test images (5,000 from SD-1 and 5,000 from SD-3) is available on this site. The full 60,000 sample training set is available.
```

Downloaded from openml.org.

fetch_openml() 函数有点不寻常，因为默认情况下以 Pandas DataFrame 返回输入，将标签作为 Pandas Series 返回（除非数据集是稀疏的）。但是 MNIST 数据集包含图像，而且 DataFrame 此时并不理想，所以最好设置 **as_frame=False** 来获取数据为 NumPy 数组。让我们来看看这些数组：

```
In [7]: mnist.keys() # extra code - we only use data and target in this notebook
```

```
Out[7]: dict_keys(['data', 'target', 'frame', 'categories', 'feature_names', 'target_names', 'DESCR', 'details', 'url'])
```

```
In [8]: X, y = mnist.data, mnist.target
```

```
X
```

```
Out[8]: 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.]])
```

```
In [9]: X.shape
```

```
(70000, 784)
```

```
In [10]: y
```

```
Out[10]: array(['5', '0', '4', ..., '4', '5', '6'], dtype=object)
```

```
In [11]: y.shape
```

```
Out[11]: (70000,)
```

有7万张图片，每张图片有784个特征。这是因为每张图片是28×28像素，每个特征只是代表一个像素的强度，从0（白色）到255（黑色）。让我们观察数据集中的一位数字。我们所需要的就是获取一个实例的特征向量，将其重塑为一个28×28数组，并使用 Matplotlib 的 **imshow()** 函数来显示它。我们使用 **cmap="binary"** 得到一个灰度图，其中0是白色，255是黑色：

```
In [12]: import matplotlib.pyplot as plt

def plot_digit(image_data):
    image = image_data.reshape(28, 28)
    plt.imshow(image, cmap="binary")
    plt.axis("off")

some_digit = X[0]
plot_digit(some_digit)
save_fig("some_digit_plot") # extra code
plt.show()
```



这看起来像一个5，确实这就是标签告诉我们的：

```
In [13]: y[0]
```

```
Out[13]: '5'
```

```
In [14]: # extra code - this cell generates and saves Figure
plt.figure(figsize=(9, 9))

for idx, image_data in enumerate(X[:100]):
    plt.subplot(10, 10, idx + 1)
    plot_digit(image_data)

plt.subplots_adjust(wspace=0, hspace=0)

save_fig("more_digits_plot", tight_layout=False)

plt.show()
```



但是等等！在仔细检查数据之前，应该创建一个测试集并将其放在一边。`fetch_openml()` 返回的MNIST数据集实际上已经被分为一个训练集（前60000张图像）和一个测试集（最后10000张图像）：

```
In [15]: X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

训练集对我们来说已经被打乱了，这很好，因为这保证了所有的交叉验证折叠都是相似的（我们不希望一个折叠丢失一些数字）。此外，一些学习算法对训练实例的顺序很敏感，如果它们连续获得许多相似的实例，其表现就会很差。打乱数据集可以确保这种情况不会发生。

2. 训练一个二元分类器（Training a Binary Classifier）

让我们暂时简化这个问题，并且只尝试识别一个数字。例如，数字5。这个“5-detector”将是一个二元分类器的一个例子，能够只区分两个类，5和非5。首先，我们将为这个分类任务创建目标向量：

```
In [16]: y_train_5 = (y_train == '5') # True for all 5s, False for all other digits
         y_test_5 = (y_test == '5')
```

现在让我们选择一个分类器并训练它。一个很好的开始是使用 随机梯度下降（**stochastic gradient descent, SGD**）分类器，使用Scikit-Learn的 **SGDClassifier** 类。这个分类器能够有效地处理非常大的数据集。这在一定程度上是因为SGD可以独立地处理训练实例，一次处理一个，这也使得SGD非常适合在线学习，稍后您将在后面看到。让我们创建一个SGD分类器，并在整个训练集上对其进行训练：

```
In [17]: from sklearn.linear_model import SGDClassifier

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

```
Out[17]: SGDClassifier
```

SGDClassifier(random_state=42)

现在我们可以用它来检测数字5的图像：

```
In [18]: sgd_clf.predict([some_digit])
```

```
Out[18]: array([ True])
```

分类器猜测此图像代表一个5（True）。看起来在这个特殊的情况下猜对了！现在，让我们来评估一下这个模型的性能。

3. 性能度量（Performance Measures）

评估分类器通常比评估回归器要复杂得多，所以我们将在本章的大部分时间来讨论这个主题。有许多性能指标可用，所以再喝一杯咖啡，准备学习一堆新概念和缩略词！

3.1 使用交叉验证测量精度（Measuring Accuracy Using Cross-Validation）

评估模型的一个好方法是使用交叉验证，就像您在第二章中所做的那样。让我们使用 **cross_val_score()** 函数来评估我们的 **SGDClassifier** 模型，使用k折交叉验证。请记住，k折交叉验证意味着将训练集分割成k个折叠（在这种情况下，是三个），然后训练模型k次，每次都保持一个不同的折叠进行评估（见第2章）：

```
In [19]: from sklearn.model_selection import cross_val_score

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

```
Out[19]: array([0.95035, 0.96035, 0.9604 ])
```

哇！所有交叉验证折叠的准确率（正确预测的比率）超过 95%？这看起来很神奇，不是吗？好吧，在你太兴奋之前，让我们看看一个虚拟分类器，它只对最频繁出现的类别中的每个图像进行分类，在这种情况下是负类（即非5）：

```
In [20]: from sklearn.dummy import DummyClassifier

dummy_clf = DummyClassifier()
dummy_clf.fit(X_train, y_train_5)
print(any(dummy_clf.predict(X_train)))
```

False

没错，它的准确率超过90%！这仅仅是因为只有大约 10% 的图像是 5，所以如果您总是猜测图像不是 5，那么您在 90% 的情况下都是正确的。击败诺查丹玛斯（一位预言家）。

这说明了为什么准确性通常不是分类器的首选性能度量，特别是当您在处理倾斜的数据集时(例如，当一些类比其他类要频繁得多)。评估分类器性能的一个更好的方法是查看 **混淆矩阵 (confusion matrix, CM)**。

实施交叉验证：有时，您需要对交叉验证过程进行比Scikit-Learn提供的更多的控制。在这些情况下，您可以自己实现交叉验证。下面的代码与Scikit-Learn的 **cross_val_score()** 函数大致相同，它打印相同的结果：

```
In [21]: from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3) # add shuffle=True if the dataset is not
                                     # already shuffled
for train_index, test_index in skfolds.split(X_train, y_train_5):
    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))

cross_val_score(dummy_clf, X_train, y_train_5, cv=3, scoring="accuracy")

0.95035
0.96035
0.9604
array([0.90965, 0.90965, 0.90965])

Out[21]:
```

StratifiedKFold 类执行分层抽样（如第2章所述），以产生包含每个类的代表性比例的折叠。在每次迭代中，代码创建一个分类器的克隆，在训练折叠上进行克隆的训练，并在测试折叠上进行预测。然后，它计算正确预测的数量，并输出正确预测的比率。

3.2 混淆矩阵 (Confusion Matrices)

混淆矩阵的一般思想是计算所有A/B对的A类实例被归类为B类的次数。例如，要知道分类器将8的图像与0混淆的次数，您可以查看混淆矩阵的第8行，第0列。

要计算混淆矩阵，首先需要有一组预测，以便将它们与实际目标进行比较。您可以对测试集做出预测，但最好暂时保持不变（记住，一旦有了准备启动的分类器，您只想在项目的最后使用测试集）。相反，您可以使用 **cross_val_predict()** 功能：

```
In [22]: 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([ True, False, False, ...,  True, False, False])

Out[22]:
```

就像 **cross_val_score()** 函数一样，**cross_val_predict()** 执行k折交叉验证，但它不是返回评估分数，而是返回对每个测试折叠所做的预测。这意味着您可以对训练集中的每个实例进行一个干净的预测（“干净”我指的是“样本外”：模型对在训练中从未见过的数据进行预测）。

现在您准备使用 **confusion_matrix()** 函数得到混淆矩阵。只需将传递目标类（y_train_5）和预测类（y_train_pred）：

```
In [23]: from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_train_5, y_train_pred)

cm
```

```
Out[23]: array([[53892,   687],
        [ 1891,  3530]], dtype=int64)
```

混淆矩阵中的每一行代表一个实际的类，而每一列则代表一个预测的类。该矩阵的第一行考虑了非5图像（负类）：其中53892个被正确地归类为非5（它们被称为真阴性），而其余687个被错误地归类为5（假阳性，也称为I型错误）。第二行考虑了5的图像（阳性类）：1891个被错误地归类为非5s（假阴性，也称为II型错误），而其余的3530个被正确地归类为5s（真阳性）。一个完美的分类器只有真正和真负，所以它的混淆矩阵只有在其主对角线（从左上到右下）上有非零值：

```
In [24]: y_train_perfect_predictions = y_train_5 # pretend we reached perfection

confusion_matrix(y_train_5, y_train_perfect_predictions)
```

```
Out[24]: array([[54579,    0],
        [    0,  5421]], dtype=int64)
```

混淆矩阵提供了很多信息，但有时您可能更喜欢更简洁的度量。一个有趣的度量是正预测的准确性；这被称为分类器的 **精度（precision）**。

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

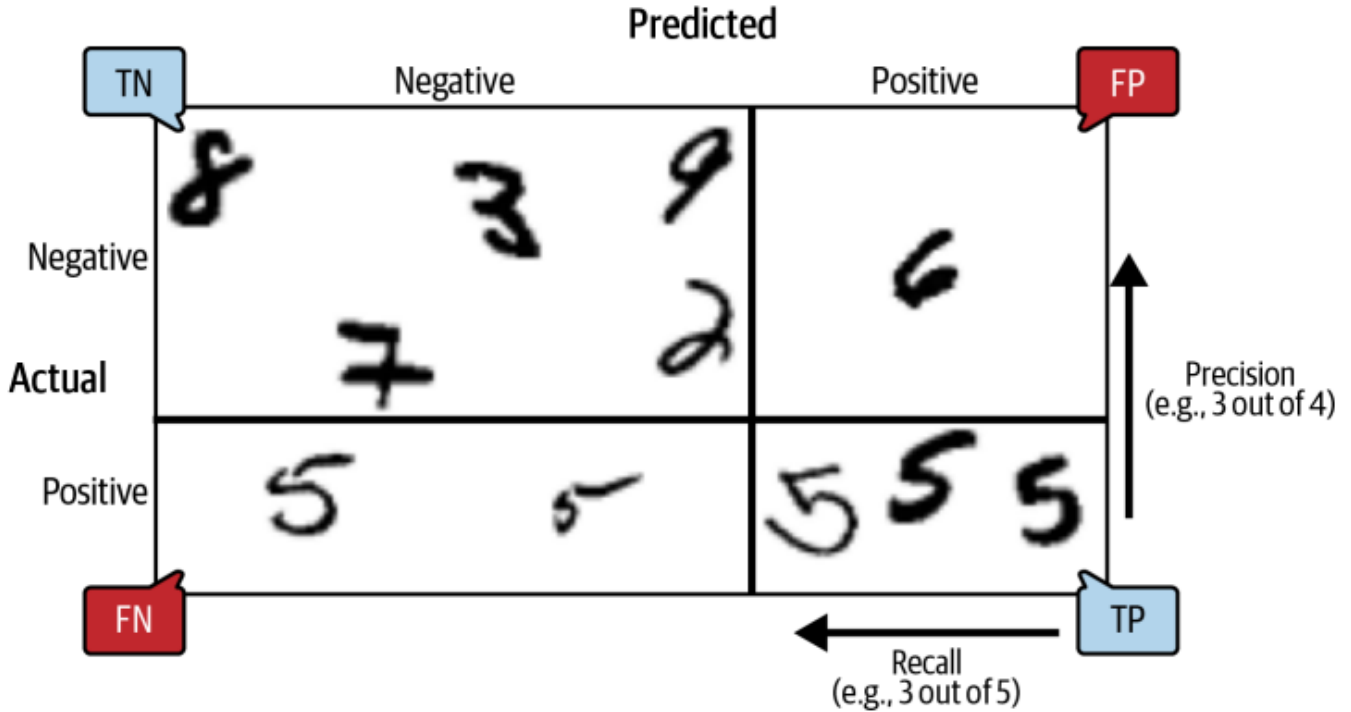
TP 为真阳性数量， FP 为假阳性数量。

获得完美精度的一种简单方法是创建一个总是做出负面预测的分类器，除了对它最有信心的实例的一个单一正面预测。如果这个预测是正确的，那么分类器就有 100% 的精度（精度 = $1/1 = 100\%$ ）。显然，这样的分类器不会很有用，因为它会忽略除一个正实例之外的所有实例。因此，精度通常与另一个名为召回率的指标一起使用，召回率（**recall**）也称为 **灵敏度（sensitivity）** 或 **真阳性率(true positive rate, TPR)**：这是分类器正确检测到的阳性实例的比率。

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

当然， FN 是假阴性的数量。

混淆矩阵：



3.3 精度和召回率（Precision and Recall）

Scikit-Learn提供了几个功能来计算分类器指标，包括精度和召回率：

```
In [25]: from sklearn.metrics import precision_score, recall_score

precision_score(y_train_5, y_train_pred)  # == 3530 / (687 + 3530)
```

Out[25]: 0.8370879772350012

```
In [26]: # extra code - this cell also computes the precision: TP / (FP + TP)
cm[1, 1] / (cm[0, 1] + cm[1, 1])
```

Out[26]: 0.8370879772350012

```
In [27]: recall_score(y_train_5, y_train_pred)  # == 3530 / (1891 + 3530)
```

Out[27]: 0.6511713705958311

```
In [28]: # extra code - this cell also computes the recall: TP / (FN + TP)
cm[1, 1] / (cm[1, 0] + cm[1, 1])
```

Out[28]: 0.6511713705958311

现在我们的 5-detector 看起来不像我们看它的准确性时那么闪亮了。当它声称一个图像代表5时，只有83.7%的情况是正确的。此外，它只能检测到65.1%的5。

将精度和召回率合并到一个称为 **F1** 的单一度量中通常很方便，特别是当您需要一个单一的度量来比较两个分类器时。F1是 精度和召回率的调和平均值。而正则平均值平均对待所有的值，调和平均值给低值更多的权重。因此，只有当召回率和精度都很高时，分类器才能得到较高的F1分数。

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

要计算 F1 得分，只需调用 `f1_score()` 函数：

```
In [29]: from sklearn.metrics import f1_score  
  
f1_score(y_train_5, y_train_pred)
```

```
Out[29]: 0.7325171197343846
```

```
In [30]: # extra code - this cell also computes the f1 score  
cm[1, 1] / (cm[1, 1] + (cm[1, 0] + cm[0, 1]) / 2)
```

```
Out[30]: 0.7325171197343847
```

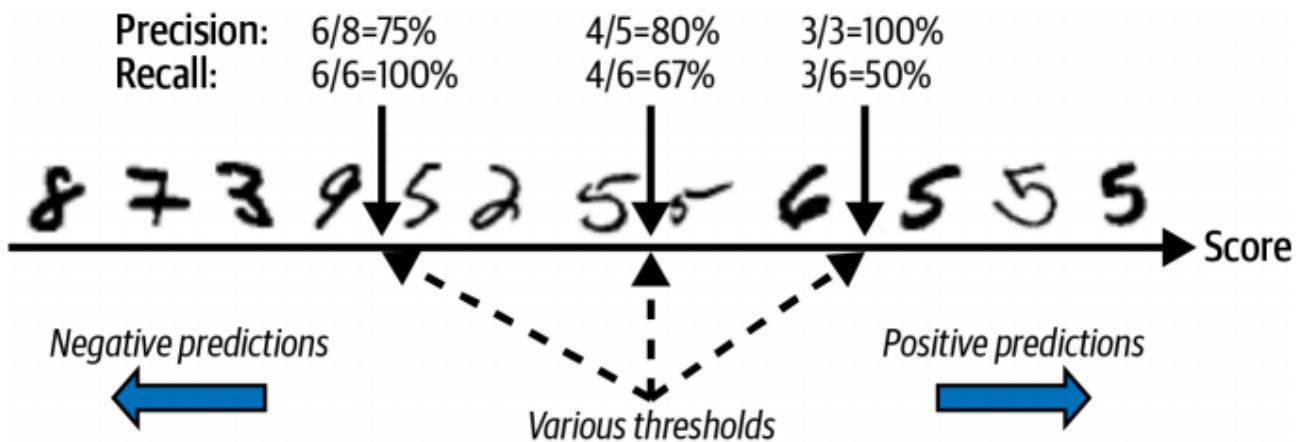
F1得分更倾向于具有相似精度和召回率的分类器。这并不总是你想要的：在某些情况下，你主要关心精确度，而在其他情况下，你才真正关心召回率。

例如，如果你训练一个分类器来检测视频对孩子是安全的，你可能会喜欢一个分类器拒绝许多好视频（低召回）但只保持安全的（高精度），而不是一个分类器高召回但让一些非常糟糕的视频出现在你的产品（在这种情况下，你甚至可能想添加一个人类管道来检查分类器的视频选择）。另一方面，假设你训练一个分类器在监控图像中的商店扒手：如果你的分类器只有30%的准确率，只要召回99%，这可能是可以的（当然，保安会得到一些错误警报，但几乎所有的商店扒手都会被抓住）。

不幸的是，你不能两者兼得：提高精度会降低召回率，反之亦然。这被称为精度/召回率的权衡。

3.4 精度/召回率权衡（The Precision/Recall Trade-of）

为了理解这种权衡，让我们来看看SGD分类器是如何做出其分类决策的。对于每个实例，它都会根据一个决策函数来计算一个分数。如果该分数大于阈值，它将实例分配给正类；否则，它将其分配给负类。



如图显示了从左边的最低分数到右边的最高分数的几位数字。假设决策阈值位于中心箭头处（在两个秒之间）：您将会在该阈值的右侧找到4个真阳性（实际值为5），以及1个假阳性（实际上为6）。因此，在这个阈值下，精度为80%（5分之4分）。但在6个实际的5中，分类器只检测到4个，所以召回率为67%（6个中的4个）。如果提高阈值（将其移动到右边的箭头），假阳性将变成真阴性，从而提高精度（在这种情况下高达100%），但一个真阳性将变成假阴性，将召回率降低到50%。相反，降低阈值会增加查全率，降低精度。

Scikit-Learn 不允许您直接设置阈值，但它确实可以让您访问它用于进行预测的决策分数。不调用分类器的 `predict()` 方法，而是可以调用它的 `decision_function()` 方法，该方法为每个实例返回一个分数，然后使用你想要的任何阈值来基于这些分数进行预测：

```
In [31]: y_scores = sgd_clf.decision_function([some_digit])
```

```
y_scores
```

```
Out[31]: array([2164.22030239])
```

```
In [32]: threshold = 0

y_some_digit_pred = (y_scores > threshold)
```

```
In [33]: y_some_digit_pred
```

```
Out[33]: array([ True])
```

SGDClassifier 使用一个等于0的阈值，因此前面的代码返回与 **predict()** 方法相同的结果（即True）。让我们来提高这个阈值：

```
In [34]: threshold = 3000

y_some_digit_pred = (y_scores > threshold)

y_some_digit_pred
```

```
Out[34]: array([False])
```

这证实了提高阈值会降低召回率。图像实际上代表了一个5，当阈值为0时，分类器会检测到它，但当阈值增加到3000时，它会错过它。

如何决定使用哪个阈值？首先，使用 **cross_val_predict()** 函数来获得训练集中所有实例的分数，但这一次指定您想要返回的决策分数，而不是预测：

```
In [35]: y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                                     method="decision_function")

y_scores
```

```
Out[35]: array([ 1200.93051237, -26883.79202424, -33072.03475406, ...,
                13272.12718981, -7258.47203373, -16877.50840447])
```

有了这些分数，使用 **precision_recall_curve()** 函数计算所有可能阈值的精度和召回率（该函数增加最后一个精度为0，最后一个召回率为1，对应于一个无限阈值）：

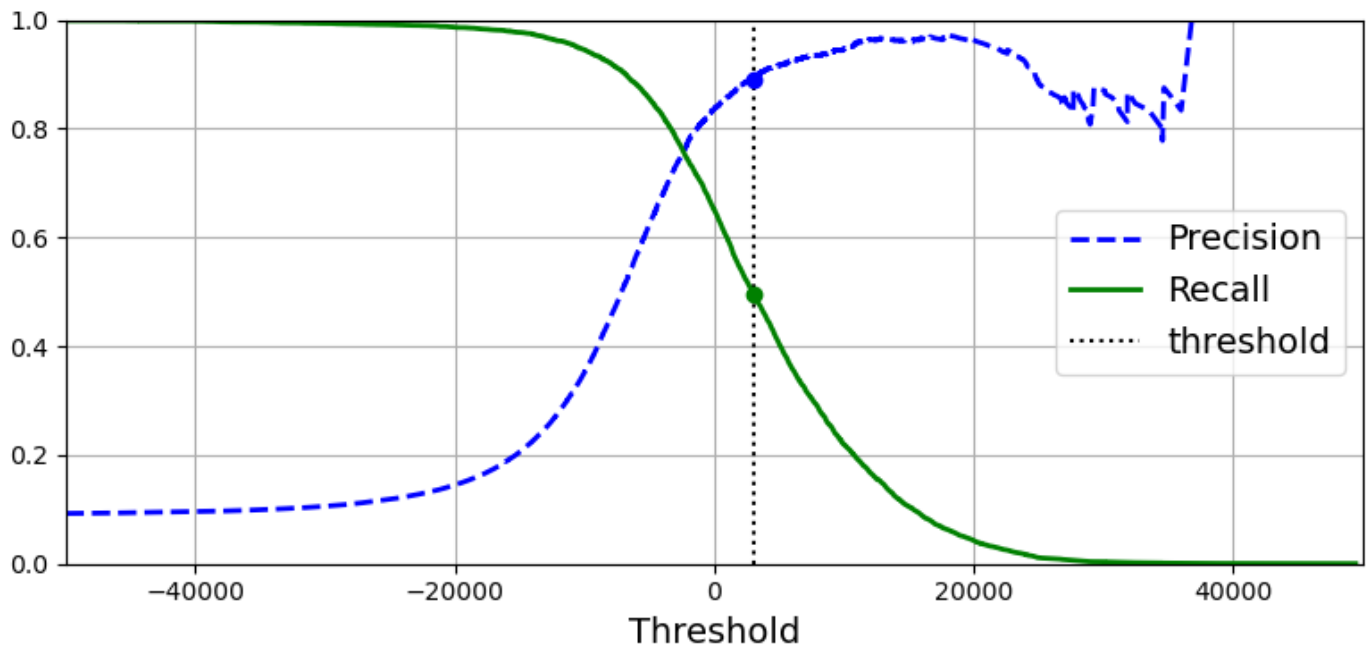
```
In [36]: from sklearn.metrics import precision_recall_curve

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

```
In [37]: plt.figure(figsize=(8, 4)) # extra code - it's not needed, just formatting
plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
plt.vlines(threshold, 0, 1.0, "k", "dotted", label="threshold")

# extra code - this section just beautifies and saves Figure
idx = (thresholds >= threshold).argmax() # first index >= threshold
plt.plot(thresholds[idx], precisions[idx], "bo")
plt.plot(thresholds[idx], recalls[idx], "go")
plt.axis([-50000, 50000, 0, 1])
plt.grid()
plt.xlabel("Threshold")
plt.legend(loc="center right")
save_fig("precision_recall_vs_threshold_plot")
```

```
plt.show()
```



您可能想知道为什么精度曲线比召回率曲线更颠簸。原因是当你提高阈值时，精度有时会下降（尽管通常它会上升）。要理解原因，请回顾之前，注意当您从中央阈值开始并向右移动一位时会发生什么：精度从 4/5 (80%) 下降到 3/4 (75%)。另一方面，召回率只有在阈值增加时才会下降，这也解释了为什么它的曲线看起来很平滑。

在这个阈值下，准确率接近90%，召回率约为50%。另一种选择良好精度/召回率权衡的方法是直接绘制精度和召回率，如图所示（显示相同的阈值）：

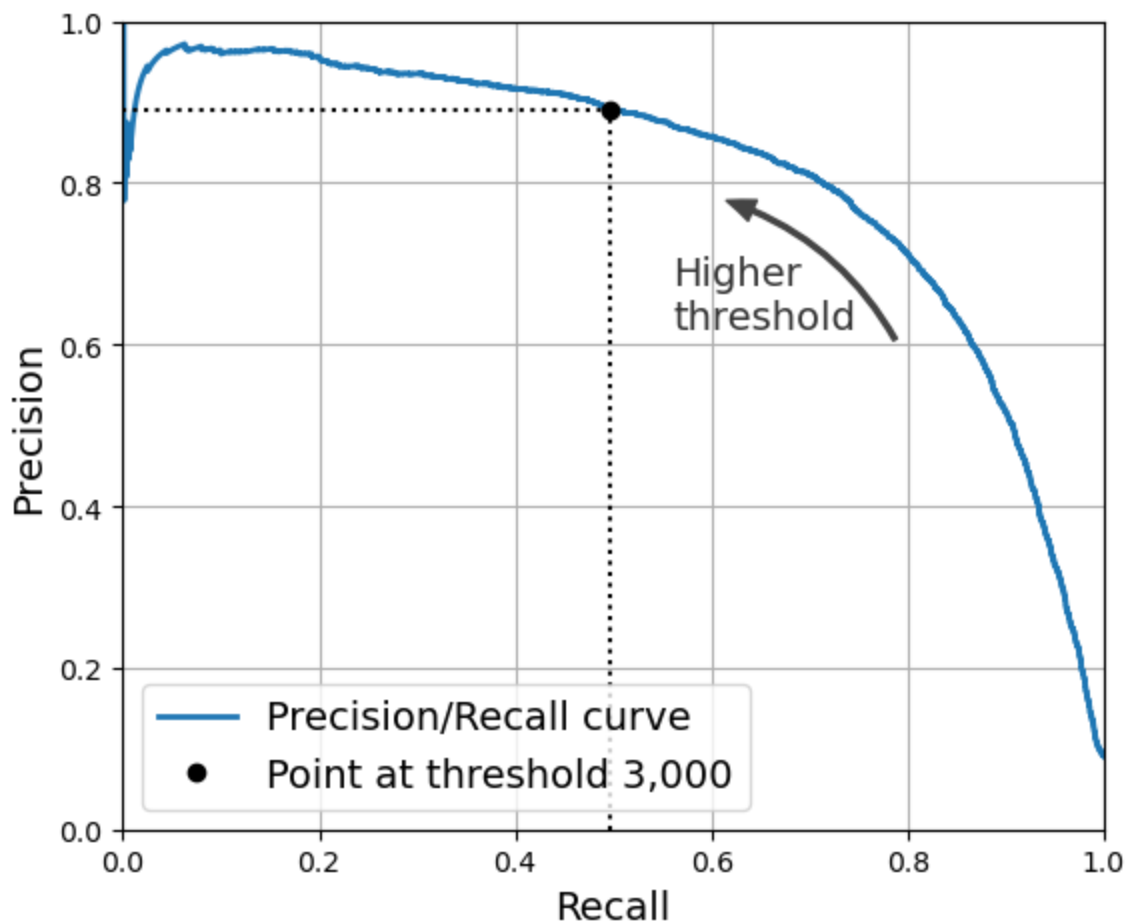
```
In [38]: import matplotlib.patches as patches # extra code - for the curved arrow

plt.figure(figsize=(6, 5)) # extra code - not needed, just formatting

plt.plot(recalls, precisions, linewidth=2, label="Precision/Recall curve")

# extra code - just beautifies and saves Figure 3-6
plt.plot([recalls[idx], recalls[idx]], [0., precisions[idx]], "k:")
plt.plot([0.0, recalls[idx]], [precisions[idx], precisions[idx]], "k:")
plt.plot([recalls[idx], [precisions[idx]], "ko",
         label="Point at threshold 3,000")
plt.gca().add_patch(patches.FancyArrowPatch(
    (0.79, 0.60), (0.61, 0.78),
    connectionstyle="arc3,rad=.2",
    arrowstyle="Simple, tail_width=1.5, head_width=8, head_length=10",
    color="#444444"))
plt.text(0.56, 0.62, "Higher\ nthreshold", color="#333333")
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.axis([0, 1, 0, 1])
plt.grid()
plt.legend(loc="lower left")
save_fig("precision_vs_recall_plot")

plt.show()
```



你可以看到，在80%左右的精度开始急剧下降的召回率。你可能会想要在下落之前选择一个精度/召回率的权衡——例如，在大约60%的召回率下。但当然，选择取决于你的项目。

假设你决定瞄准90%的精度。您可以使用第一个图来找到您需要使用的阈值，但这不是很精确。或者，您可以搜索至少90%精度的最低阈值。为此，您可以使用NumPy数组的 **argmax()** 方法。这将返回最大值的第一个索引，在这种情况下表示第一个 True 值：

```
In [39]: idx_for_90_precision = (precisions >= 0.90).argmax()
threshold_for_90_precision = thresholds[idx_for_90_precision]
threshold_for_90_precision
```

```
Out[39]: 3370.0194991439557
```

要进行预测（目前在训练集上），您可以运行以下代码，而不是调用分类器的 **predict()** 方法：

```
In [40]: y_train_pred_90 = (y_scores >= threshold_for_90_precision)
```

让我们来看看这些预测的精度和召回率：

```
In [41]: precision_score(y_train_5, y_train_pred_90)
```

```
Out[41]: 0.9000345901072293
```

```
In [42]: recall_score(y_train_5, y_train_pred_90)
```

```
Out[42]: 0.4799852425751706
```

很好，你有一个90%的精度分类器！正如您所看到的，创建一个几乎具有任何您想要的精度的分类器是相当容易的：只需设置一个足够高的阈值，然后您就完成了。但是等等，别那么快——如果一个高精度分类器的

召回率太低，它就不是很有用！对于许多应用程序来说，48%的召回率根本不算太好。

3.5 ROC 曲线（The ROC Curve）

接受者操作特征 (**receiver operating characteristic, ROC**) 曲线 是二元分类器使用的另一种常用工具。它与精度/召回率曲线非常相似，但不是绘制精确率与召回率。ROC 曲线绘制的是真阳性率（召回率的另一个名称）与假阳性率 (FPR) 的关系。FPR（也称为 **fall-out**）是被错误分类为正例的负例的比率。它等于 $1 - \text{真阴性率 (TNR)}$ ，即被正确分类为阴性的阴性实例的比率。TNR 也称为特异性。因此，ROC 曲线绘制了灵敏度（召回率）与 $1 - \text{特异性}$ 。

要绘制 ROC 曲线，首先使用 `roc_curve()` 函数计算各种阈值的 TPR 和 FPR：

```
In [43]: from sklearn.metrics import roc_curve

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

thresholds
```

```
Out[43]: array([ 49442.43765905,  49441.43765905,  36801.60697028, ...,
        -105763.22240074, -106527.45300471, -146348.56726174])
```

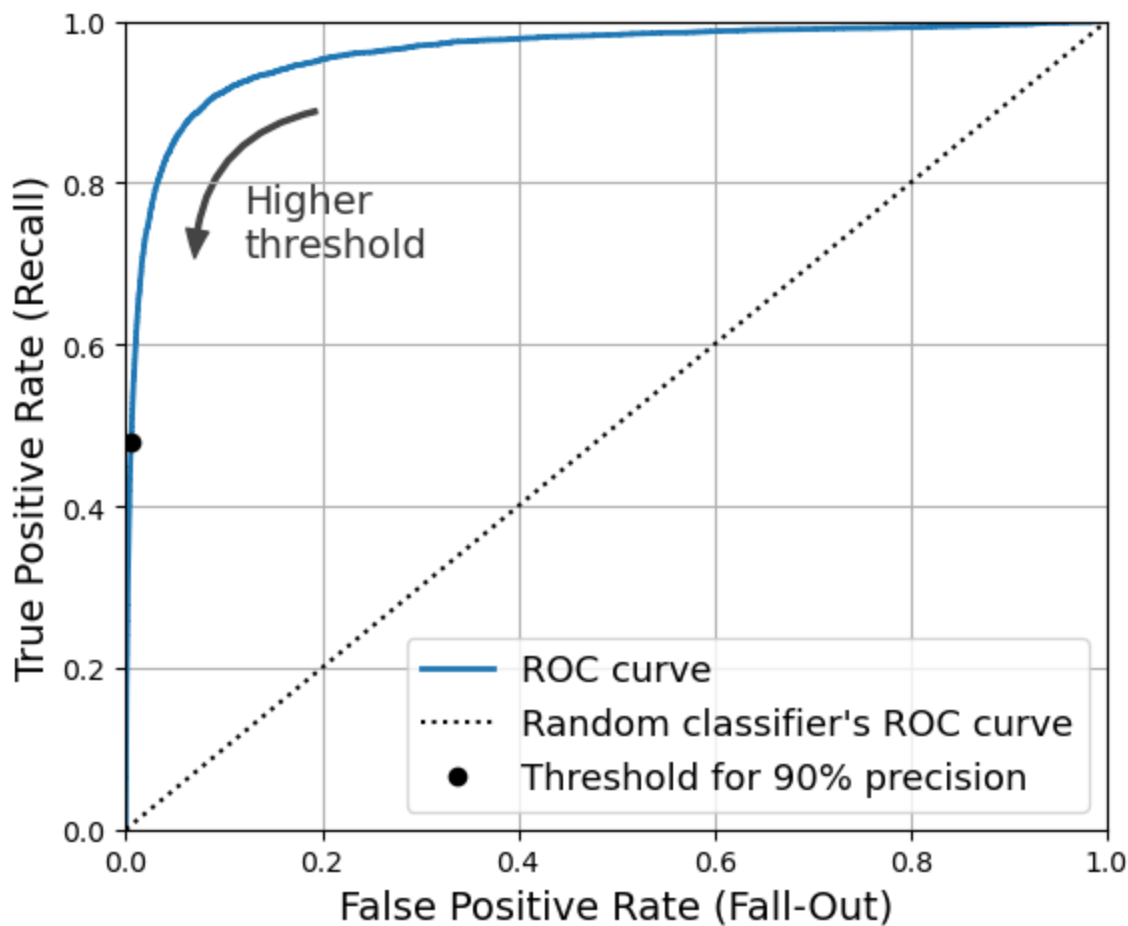
然后你可以使用 Matplotlib 绘制针对 TPR 的 FPR。为了找到对应于90%精度的点，我们需要寻找期望阈值的索引。由于在这种情况下，阈值是按递减顺序列出的，所以我们在第一行使用 `<=` 而不是 `>=`：

```
In [44]: idx_for_threshold_at_90 = (thresholds <= threshold_for_90_precision).argmax()
tpr_90, fpr_90 = tpr[idx_for_threshold_at_90], fpr[idx_for_threshold_at_90]

plt.figure(figsize=(6, 5)) # extra code - not needed, just formatting
plt.plot(fpr, tpr, linewidth=2, label="ROC curve")
plt.plot([0, 1], [0, 1], 'k:', label="Random classifier's ROC curve")
plt.plot([fpr_90], [tpr_90], "ko", label="Threshold for 90% precision")

# extra code - just beautifies and saves Figure 3-7
plt.gca().add_patch(patches.FancyArrowPatch(
    (0.20, 0.89), (0.07, 0.70),
    connectionstyle="arc3,rad=.4",
    arrowstyle="Simple, tail_width=1.5, head_width=8, head_length=10",
    color="#444444"))
plt.text(0.12, 0.71, "Higher\ntthreshold", color="#333333")
plt.xlabel('False Positive Rate (Fall-Out)')
plt.ylabel('True Positive Rate (Recall)')
plt.grid()
plt.axis([0, 1, 0, 1])
plt.legend(loc="lower right", fontsize=13)
save_fig("roc_curve_plot")

plt.show()
```



再一次存在一种权衡：召回率（TPR）越高，分类器产生的假阳性（FPR）就越多。虚线表示一个纯随机分类器的ROC曲线；一个好的分类器会尽可能地远离这条线（朝向左上角）。

比较分类器的一种方法是测量 曲线下的面积（**area under the curve, AUC**）。一个完美的分类器的ROC AUC等于1，而一个纯随机的分类器的ROC AUC等于0.5。Scikit-Learn提供了一个功能来估计ROC的AUC：

```
In [45]: from sklearn.metrics import roc_auc_score

roc_auc_score(y_train_5, y_scores)
```

```
Out[45]: 0.9604938554008616
```

由于ROC曲线与精度/召回率（precision/recall, PR）曲线如此相似，你可能会想知道如何决定使用哪一种曲线。根据经验法则，当阳性类很少见时，或者当你更关心假阳性而不是假阴性时，你应该更喜欢PR曲线。否则，请使用ROC曲线。例如，查看前面的ROC曲线（以及ROC AUC评分），您可能认为该分类器真的很好。但这主要是因为与消极的（非5秒）相比，积极的（5秒）很少。相比之下，PR曲线清楚地表明，分类器有改进的空间：曲线确实可以更接近右上角。

现在让我们创建一个 **RandomForestClassifier**，它的 PR 曲线和 F1 分数我们可以与SGD分类器进行比较：

```
In [46]: from sklearn.ensemble import RandomForestClassifier

forest_clf = RandomForestClassifier(random_state=42)
```

precision_recall_curve() 函数对每个实例期望标签和分数，因此我们需要训练随机森林分类器，并使它为每个实例分配一个分数。但是 **RandomForestClassifier** 类没有一个 **decision_function()** 方法，这是由于它的工作方式（我们将在第7章中讨论这个方法）。幸运的是，它有一个 **predict_proba()** 方法，该方法为每个实例返回类概率，我们可以只使用正类的概率作为分数，因此它将正常工作。我们可以调用

`cross_val_predict()` 函数来使用交叉验证来训练 `RandomForestClassifier`，并使它预测每幅图像的类概率如下：

```
In [47]: y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                             method="predict_proba")

y_probas_forest

Out[47]: array([[0.11, 0.89],
                [0.99, 0.01],
                [0.96, 0.04],
                ...,
                [0.02, 0.98],
                [0.92, 0.08],
                [0.94, 0.06]])
```

该模型预测第一幅图像为正的的概率为89%，预测第二幅图像为负的概率为99%。由于每张图像要么是正的，要么是负的，所以每一行的概率加起来是100%。

这些是估计概率，而不是实际概率。例如，如果您查看模型以 50% 到 60% 的估计概率分类为阳性的所有图像，其中大约 94% 实际上是阳性的。因此，在这种情况下，模型的估计概率太低了——但模型也可能过于自信。`sklearn.calibration` 包包含用于校准估计概率并使它们更接近实际概率的工具。有关更多详细信息，请参阅本章笔记本中的额外材料部分。

这些是估计的概率。在模型以 50% 到 60% 的概率分类为正的图像中，实际上大约有 94% 的正面图像：

```
In [48]: # Not in the code
idx_50_to_60 = (y_probas_forest[:, 1] > 0.50) & (y_probas_forest[:, 1] < 0.60)
print(f"{(y_train_5[idx_50_to_60]).sum() / idx_50_to_60.sum():.1%}")

94.0%
```

第二列包含了正类的估计概率，所以让我们将它们传递给 `precision_recall_curve()` 函数：

```
In [49]: y_scores_forest = y_probas_forest[:, 1]
precisions_forest, recalls_forest, thresholds_forest = precision_recall_curve(
    y_train_5, y_scores_forest)
```

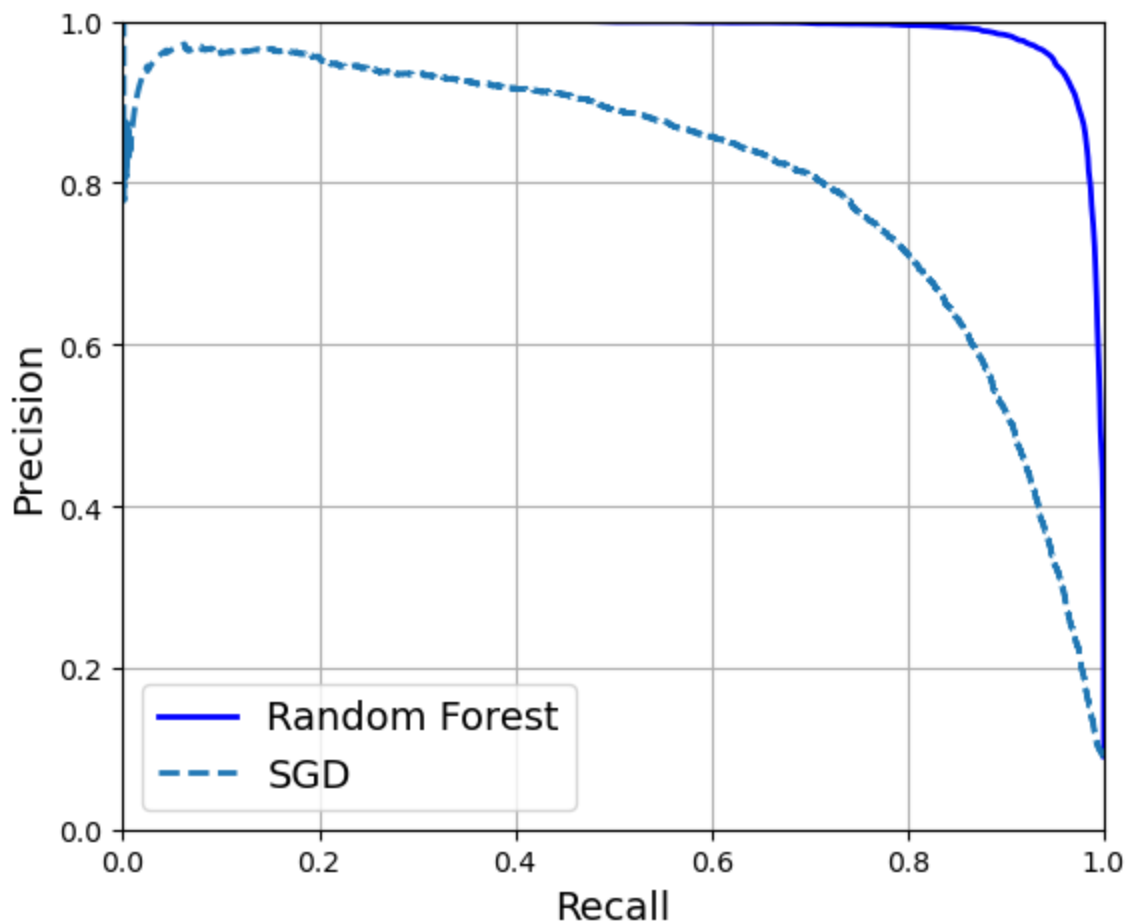
现在我们已经准备好绘制 PR 曲线了。

```
In [50]: plt.figure(figsize=(6, 5)) # extra code - not needed, just formatting

plt.plot(recalls_forest, precisions_forest, "b-", linewidth=2,
         label="Random Forest")
plt.plot(recalls, precisions, "--", linewidth=2, label="SGD")

# extra code - just beautifies and saves Figure 3-8
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.axis([0, 1, 0, 1])
plt.grid()
plt.legend(loc="lower left")
save_fig("pr_curve_comparison_plot")

plt.show()
```

正如你在图中所看到的，随机森林分类器的PR曲线看起来比SGD分类器的要好得多：它更接近右上角。其F1评分和ROC AUC评分也明显更好：

```
In [51]: y_train_pred_forest = y_probas_forest[:, 1] >= 0.5 # positive proba ≥ 50%
         f1_score(y_train_5, y_train_pred_forest)
```

```
Out[51]: 0.9274509803921569
```

```
In [52]: roc_auc_score(y_train_5, y_scores_forest)
```

```
Out[52]: 0.9983436731328145
```

试着测量准确率和召回率得分：你应该会发现大约99.1%的准确率和86.6%的召回率。还不错！

```
In [53]: precision_score(y_train_5, y_train_pred_forest)
```

```
Out[53]: 0.9897468089558485
```

```
In [54]: recall_score(y_train_5, y_train_pred_forest)
```

```
Out[54]: 0.8725327430363402
```

您现在知道如何训练二元分类器，为您的任务选择合适的度量，使用交叉验证评估分类器，选择适合您的需求的精度/召回率权衡，并使用几个度量和曲线来比较各种模型。你已经准备好尝试检测的不仅仅是5的分类器。

4. 多元分类（Multiclass Classification）

二值分类器可以区分两个类，而多类分类器（也称为多项式分类器）可以区分两个以上的类。

一些 Scikit-Learn 分类器（例如，**LogisticRegression**、**RandomForestClassifier** 和 **GaussianNB**）能够自然地处理多个类。其他的是严格的二进制分类器（例如，**SGDClassifier** 和 **SVC**）。但是，您可以使用各种策略来使用多个二进制分类器执行多类分类。

创建一个系统，将数字图像分为10类（从0到9）的一种方法是训练10个二进制分类器，每个数字一个（0检测器，1检测器，2检测器，等等）。然后，当您想对一幅图像进行分类时，您可以从该图像的每个分类器中获得决策分数，并选择其分类器输出的分数最高的类。这被称为 一对其余（**one-versus-the-rest, OvR**）策略，有时也称为 一对全（**one-versus-all, OvA**）。

另一种策略是为每一对数字训练一个二元分类器：一个区分0和1，另一个区分0和2，另一个区分1和2，以此类推。这被称为 一对一（**OvO**）策略。如果有N个类，则需要训练 $\frac{N(N-1)}{2}$ 个分类器。对于MNIST问题，这意味着训练45个二元分类器！当您想要对一个图像进行分类时，您必须在所有45个分类器中运行该图像，并看看哪个类别赢得了最多的决斗。OvO的主要优点是，每个分类器只需要在包含它必须区分的两个类的训练集中进行训练。

一些算法（如支持向量机分类器）随着训练集的大小而扩展。对于这些算法，OvO是首选的，因为它在小的训练集上训练许多分类器更快，而不是在大的训练集上训练很少的分类器。然而，对于大多数二值分类算法，OvR是首选的。

Scikit-Learn 检测到当你尝试对多类分类任务使用二值分类算法时，它会根据算法自动运行 OvR 或 OvO。让我们用 **sklearn.svm.SVC** 使用支持向量机分类器（见第5章）。我们只会训练前2000张图片，否则会需要很长时间：

```
In [63]: from sklearn.svm import SVC

svm_clf = SVC(random_state=42)

svm_clf.fit(X_train[:2000], y_train[:2000]) # y_train, not y_train_5
```

```
Out[63]: SVC
SVC(random_state=42)
```

这很容易！我们使用从0到9的原始目标类（y_train）来训练SVC，而不是使用5和其余的目标类（y_train_5）。由于有10个类（即超过2个），Scikit-Learn使用了OvO策略，并训练了45个二进制分类器。现在，让我们对一个图像做出一个预测：

```
In [64]: svm_clf.predict([some_digit])
```

```
Out[64]: array(['5'], dtype=object)
```

没错！这段代码实际上做了 45 次预测——每对类一个——它选择了赢得最多决斗的类。如果您调用 **decision_function()** 方法，您会看到它为每个实例返回 10 个分数：每个类一个。根据分类器分数，每个类的分数等于赢得决斗的次数加上或减去打破平局的小调整（最大±0.33）：

```
In [65]: some_digit_scores = svm_clf.decision_function([some_digit])

some_digit_scores.round(2)
```

```
Out[65]: array([[ 3.79,  0.73,  6.06,  8.3 , -0.29,  9.3 ,  1.75,  2.77,  7.21,
  4.82]])
```

最高分是9.3分，它确实对应于第5类：

```
In [66]: class_id = some_digit_scores.argmax()

class_id
```

```
Out[66]: 5
```

当一个分类器被训练时，它将目标类的列表存储在其 **classes_** 属性中，按值排序。在MNIST的情况下，**classes_** 数组中的每个类的索引方便地匹配类本身（例如，索引5处的类恰好是类'5'），但通常你不会那么幸运；你需要像这样查找类标签

```
In [67]: svm_clf.classes_
```

```
Out[67]: array(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'], dtype=object)
```

```
In [68]: svm_clf.classes_[class_id]
```

```
Out[68]: '5'
```

```
In [69]: # extra code - shows how to get all 45 OvO scores if needed
svm_clf.decision_function_shape = "ovo"
some_digit_scores_ovo = svm_clf.decision_function([some_digit])
some_digit_scores_ovo.round(2)
```

```
Out[69]: array([[ 0.11, -0.21, -0.97,  0.51, -1.01,  0.19,  0.09, -0.31, -0.04,
        -0.45, -1.28,  0.25, -1.01, -0.13, -0.32, -0.9 , -0.36, -0.93,
         0.79, -1.  ,  0.45,  0.24, -0.24,  0.25,  1.54, -0.77,  1.11,
         1.13,  1.04,  1.2 , -1.42, -0.53, -0.45, -0.99, -0.95,  1.21,
         1.  ,  1.  ,  1.08, -0.02, -0.67, -0.14, -0.3 , -0.13,  0.25]])
```

如果你想强制 Scikit-Learn 使用一对一（OVO）或一对其余（OVR），你可以使用 **OneVsOneClassifier** 或 **OneVsRestClassifier**。简单地创建一个实例，并将一个分类器传递给它的构造函数（它甚至不需要是一个二进制分类器）。

例如，该代码基于 **SVC**，使用 **OvR** 策略创建了一个多分类器：

```
In [70]: from sklearn.multiclass import OneVsRestClassifier

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

```
Out[70]: ▸ OneVsRestClassifier
          ▸ estimator: SVC
              ▸ SVC
```

```
In [71]: ovr_clf.predict([some_digit])
```

```
Out[71]: array(['5'], dtype='<U1')
```

```
In [72]: len(ovr_clf.estimators_)
```

```
Out[72]: 10
```

在一个多类数据集上训练一个 **SGDClassifier** 并使用它来进行预测也同样简单：

```
Out[73]: array(['3'], dtype='<U1')
```

预测错误确实会发生！这一次，Scikit-Learn在底层使用了OvR策略：由于有10个类，所以它训练了10个二值分类器。**decision_function()** 方法现在为每个类返回一个值。让我们来看看 SGD 分类器分配给每个类的分数：

```
In [74]: sgd_clf.decision_function([some_digit]).round()
```

```
Out[74]: array([[ -31893.,  -34420.,  -9531.,   1824.,  -22320.,  -1386.,  -26189.,
                -16148.,  -4604., -12051.]])
```

你可以看到分类器对它的预测不是很有信心：几乎所有的分数都非常负，而类3是1824，类5的分数为-1386。当然，您会希望在多个图像上评估这个分类器。由于每个类中的图像数量大致相同，所以精度度量很好。与往常一样，您可以使用 `cross_val_score()` 函数来评估模型：

```
In [75]: cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
```

```
Out[75]: array([0.87365, 0.85835, 0.8689 ])
```

它在所有的测试折叠中都得到了超过**85.8%**。如果你使用一个随机分类器，你将得到**10%**的准确率，所以这并不是一个很糟糕的分数，但你仍然可以做得更好。简单地缩放输入将提高**89.1%**以上的精度：

```
In [76]: 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")
```

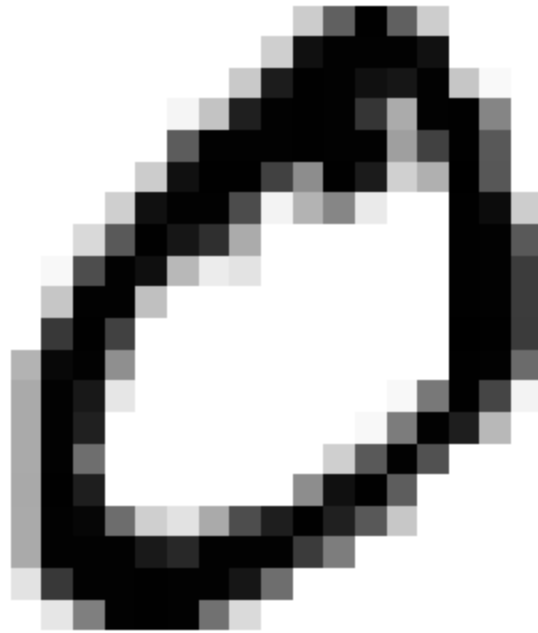
```
Out[76]: array([0.8983, 0.891 , 0.9018])
```

```
In [82]: X_train[1]
```

```
Out[82]: 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.,  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.,  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.,  0.,  0.,
  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
  48., 238., 252., 252., 252., 237.,  0.,  0.,  0.,  0.,  0.,
  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
  0.,  0.,  0.,  0.,  0.,  54., 227., 253., 252., 239., 233.,
 252.,  57.,  6.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0., 10., 60.,
 224., 252., 253., 252., 202., 84., 252., 253., 122.,  0.,  0.,
  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
  0.,  0.,  0.,  0., 163., 252., 252., 252., 253., 252., 252.,
 96., 189., 253., 167.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
```

[illegible]

```
In [83]: some_digit = X_train[1]
          plot_digit(some_digit)
          plt.show()
```

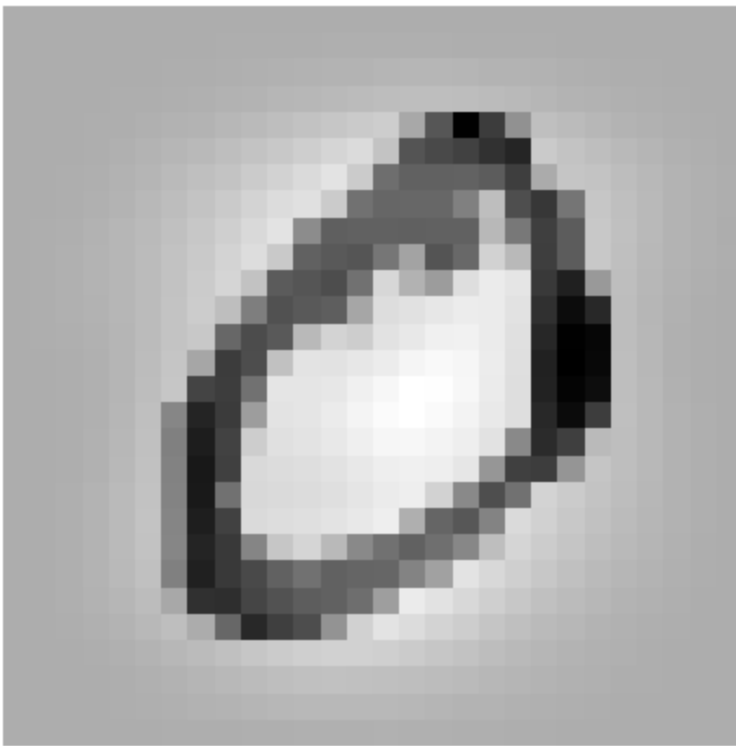


In [84]: X_train_scaled[1]

Out[84]: array([0. , 0. , 0. , 0. , 0. ,
 0. , 0. , 0. , 0. , 0. ,
 0. , 0. , -0.00441808, -0.00575482, -0.00408252,
-0.00408252, 0. , 0. , 0. , 0. , 0. ,
 0. , 0. , 0. , 0. , 0. ,
 0. , 0. , -0.00408252, -0.00470969, -0.00879935,
-0.01159056, -0.01475898, -0.01928485, -0.0246718 , -0.02907103,
-0.03059266, -0.03116401, -0.03196287, -0.03020254, -0.03131022,
-0.0283834 , -0.02311919, -0.01916663, -0.0167723 , -0.01099636,
-0.00832486, -0.00438069, 0. , 0. , 0. ,
 0. , 0. , 0. , -0.00408252, -0.00539535,
-0.00852241, -0.01198504, -0.01765348, -0.0277109 , -0.03702009,
-0.05215128, -0.0670362 , -0.08301705, -0.0993793 , -0.11518413,
-0.12913326, -0.13839468, -0.13888363, -0.13184344, -0.12042952,
-0.10189079, -0.0786049 , -0.05699561, -0.03965768, -0.02372839,
-0.01408835, -0.00783084, 0. , 0. , 0. ,
 0. , -0.00536838, -0.00887061, -0.01407082, -0.02214681,
-0.03518014, -0.05502368, -0.07909613, -0.10764901, -0.13716994,
-0.16710576, -0.19600876, -0.22449111, -0.24865599, -0.26211797,
-0.2625969 , -0.2478559 , -0.221947 , -0.1872114 , -0.14764013,
-0.10816436, -0.07681211, -0.0470753 , -0.03020197, -0.01524124,
-0.00528372, 0. , 0. , -0.00408252, -0.00776342,
-0.01489325, -0.02396275, -0.0503409 , -0.07876747, -0.11618154,
-0.16124756, -0.21196164, -0.26689873, -0.32567801, -0.39024155,
-0.45207638, -0.49939798, 0.0664791 , 1.33857348, 2.62752387,
 1.72417535, 0.42135556, -0.28368365, -0.21683666, -0.15852438,
-0.10878458, -0.07092253, -0.03895348, -0.01425239, -0.0057705 ,
 0. , 0. , -0.01190174, -0.02205016, -0.05183807,
-0.0906597 , -0.13966711, -0.19708212, -0.26438473, -0.34073044,
-0.42253575, -0.51540865, -0.61603533, -0.7159833 , -0.34621732,
 1.35857324, 1.51146014, 1.6346726 , 1.88069283, 2.09129319,
-0.44764565, -0.34762461, -0.26084091, -0.19030605, -0.13108013,
-0.07484604, -0.03193013, -0.00820892, 0. , -0.00557015,

-0.01566193, -0.0366566, -0.07902431, -0.13328909, -0.19915441,
-0.27542969, -0.36333595, -0.46308053, -0.57484519, -0.69880387,
-0.82877368, -0.47048776, 0.96721437, 1.14748071, 1.15986538,
1.13210557, 1.24594939, 1.70460494, 0.03183492, -0.37533175,
-0.33932343, -0.24893037, -0.1729397, -0.10410424, -0.04831762,
-0.01470802, -0.00408252, -0.01323101, -0.02624382, -0.05915859,
-0.11300485, -0.17844153, -0.25800481, -0.35091181, -0.45909822,
-0.58414556, -0.72495421, -0.7825882, -0.46910962, 0.89667886,
1.07977519, 1.06792169, 1.07653979, 0.68058505, -0.2662337,
1.45680752, 1.81305479, 0.85531048, -0.39754744, -0.28768128,
-0.19700703, -0.12122894, -0.05680861, -0.01539067, -0.00502549,
-0.01732052, -0.04308054, -0.0852489, -0.13945837, -0.21088331,
-0.30001798, -0.40728153, -0.53288848, -0.68062943, -0.8419885,
0.49381321, 1.19310597, 1.15478312, 1.15986991, 1.17922259,
1.16706785, 1.16966782, -0.18190196, 0.80207873, 1.67958985,
1.23168345, -0.423671, -0.29819621, -0.19768614, -0.12068382,
-0.05393563, -0.01350554, -0.00590571, -0.02110967, -0.05298904,
-0.09307849, -0.15039873, -0.22619102, -0.32364534, -0.44213122,
-0.58424748, -0.74857559, -0.43676275, 1.14312113, 1.27734539,
1.34819098, 0.83462653, 0.1343626, 1.35981076, 1.05950034,
-0.57657873, -0.18318894, 1.70169394, 1.24160583, -0.4201719,
-0.2901746, -0.18148235, -0.10493977, -0.04721317, -0.01413087,
-0.00577281, -0.02263228, -0.05280317, -0.09220581, -0.1474517,
-0.22728459, -0.33242832, -0.46269616, -0.61903352, -0.33289011,
1.24845772, 1.3435973, 1.46430749, 0.92287792, -0.65485941,
-0.09055217, 0.25260008, -0.76021703, -0.96213107, -0.86347326,
1.76822814, 2.16531034, 0.29985176, -0.27122264, -0.15932446,
-0.08125841, -0.03695087, -0.0118734, -0.00705517, -0.02077118,
-0.04720356, -0.08200384, -0.13745728, -0.22518196, -0.33849238,
-0.482935, -0.26390293, 0.72233994, 1.39958284, 1.2693843,
1.21389713, 0.1064436, -0.74095056, -0.80221047, -0.90126281,
-0.97772293, -0.95554794, -0.8249029, 1.89950656, 2.46178885,
2.03677827, -0.25846441, -0.14969571, -0.06111136, -0.02801157,
-0.0091251, -0.00530435, -0.0162384, -0.03637861, -0.06858201,
-0.1273685, -0.22614336, -0.35220291, -0.42604405, 1.074062,
1.4842561, 1.29025763, -0.18562982, -0.58381279, -0.49849034,
-0.82034372, -0.91768349, -1.02909594, -1.058279, -0.96567404,
-0.79447676, 2.0465914, 2.62578927, 2.56195635, -0.25540754,
-0.15382451, -0.05245428, -0.02150587, -0.00992198, -0.00408252,
-0.01093439, -0.0255448, -0.05707308, -0.12466202, -0.23849817,
-0.3732589, 0.10550164, 1.73384532, 1.46460089, -0.30782135,
-0.84101427, -0.81679096, -0.89192404, -0.97855056, -1.11080343,
-1.18062996, -1.12885776, -0.97607041, -0.7751954, 2.12882421,
2.66099888, 2.53621867, -0.26354918, -0.16450369, -0.05951303,
-0.02336867, -0.00740432, -0.00408252, -0.00798835, -0.01829645,
-0.05133943, -0.1293464, -0.25752139, -0.39501133, 1.60730921,
1.71913447, 0.91070888, -0.86785315, -0.86074372, -0.90403568,
-1.03470353, -1.14543618, -1.27420789, -1.24819354, -1.13977509,
-0.96328345, -0.76301132, 2.11689833, 2.59453505, 2.45887168,
-0.27332532, -0.17338663, -0.06894272, -0.02464466, -0.0069849,
-0.00477028, -0.00418943, -0.01864826, -0.05327212, -0.14042052,
-0.27718062, 0.59874893, 2.10730908, 1.74898439, 0.24284222,
-0.84194084, -0.86976698, -0.95866435, -1.09410266, -1.20489455,
-1.27242541, -1.18363406, -1.07835249, -0.92258453, -0.75248127,
2.03723548, 2.48680284, 1.70852054, -0.27829889, -0.17549155,
-0.07702614, -0.0296263, -0.00911024, -0.00408252, -0.00670728,
-0.02160143, -0.06070769, -0.15702241, -0.29716317, 0.68032474,
2.20777944, 1.6269673, -0.50332332, -0.78837956, -0.83396729,
-0.9194134, -1.02595562, -1.12277551, -1.14506316, -1.0794597,
-0.99988631, -0.81440807, 0.55412261, 1.99735007, 1.68846422,
-0.20932801, -0.27351765, -0.17353912, -0.08405015, -0.03416244,
-0.00893965, 0, -0.00893209, -0.02539776, -0.07390018,
-0.17786005, -0.31338343, 0.66111555, 2.26384481, 1.66888328,
-0.68268924, -0.72347014, -0.76555898, -0.82507042, -0.91939024,
-1.00626013, -1.03281505, -1.01067269, -0.88778656, 0.35440302,
1.67434794, 1.71250959, 0.35893907, -0.36804198, -0.26078182,

```
some_digit = X_train_scaled[1]
plot_digit(some_digit)
plt.show()
```

5. 误差分析（Error Analysis）

如果这是一个真正的项目，那么您现在将按照机器学习项目检查清单中的步骤进行操作（见附录A）。您可以探索数据准备选项，尝试多个模型，筛选出最佳模型，使用 **GridSearchCV** 微调它们的超参数，并尽可能多地实现自动化。在这里，我们将假设您已经找到了一个很有前途的模型，并且您希望找到改进它的方法。其中一种方法是分析它所犯的错误的类型。

首先，看看 **混淆矩阵**。为此，您首先需要使用 **cross_val_predict()** 函数进行预测；然后，您可以将标签和预测传递给 **confusion_matrix()** 函数，就像您之前所做的那样。但是，由于现在有10个类而不是2个类，因此混淆矩阵将包含相当多的数字，而且它可能很难阅读。

混淆矩阵的彩色图更容易分析。要绘制这样的图表，请使用这样的

ConfusionMatrixDisplay.from_predictions() 函数：

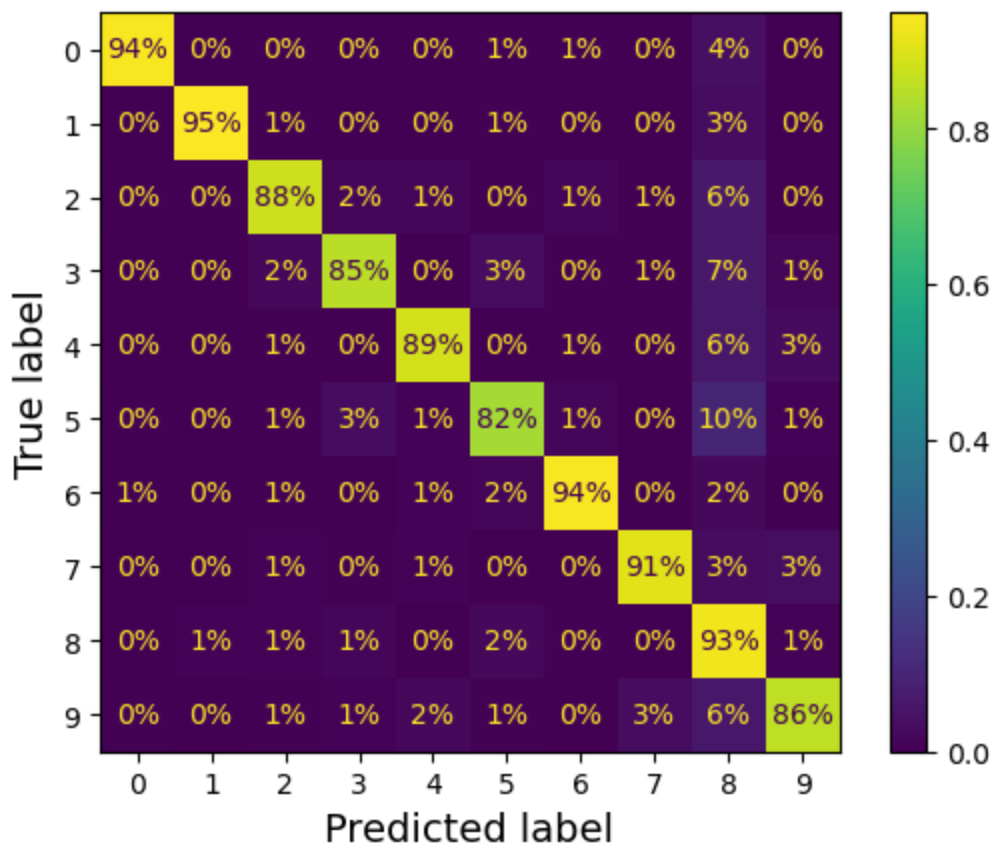
```
In [86]: from sklearn.metrics import ConfusionMatrixDisplay

y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)

plt.rc('font', size=9) # extra code - make the text smaller

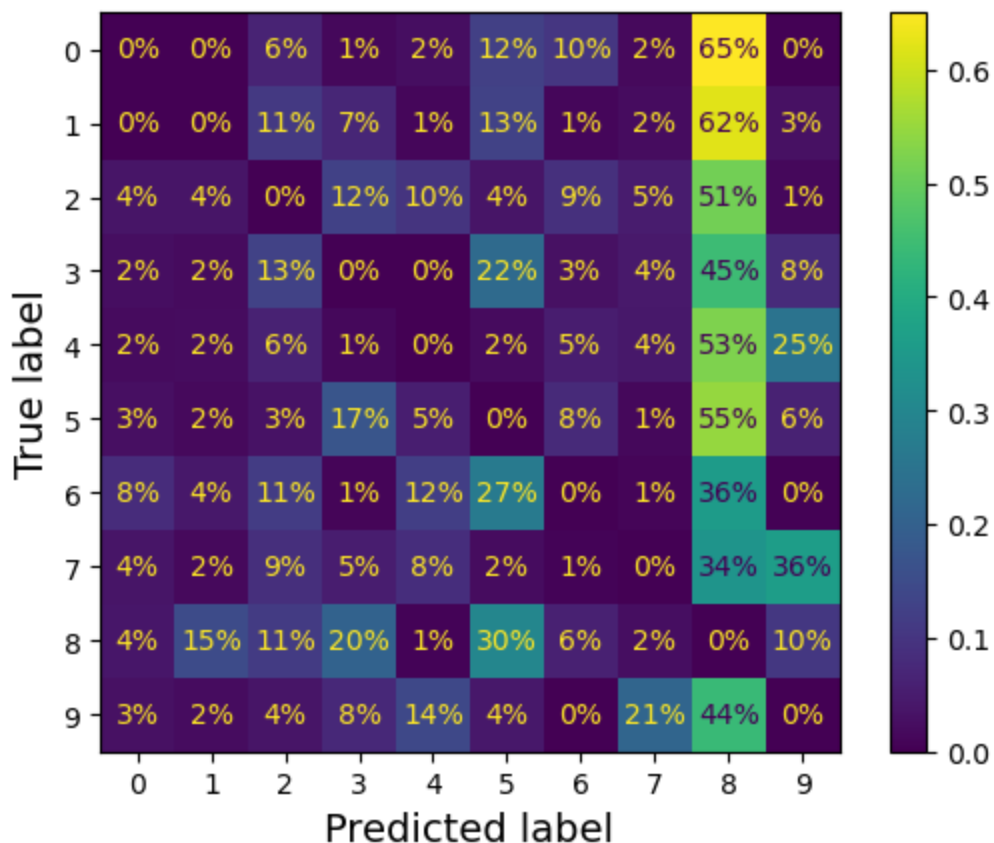
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred)

plt.show()
```

现在我们可以很容易地看到，只有82%的5图像被正确分类。该模型对5图像最常见的错误是将它们错误地分类为8：这发生在所有5图像中的10%上。但是只有2%的8被误分类为5；混淆矩阵通常是不对称的！如果你仔细看，你会注意到许多数字被错误地分类为8，但这从这个图中并不明显。如果你想让这些错误更突出，你可以尝试对正确的预测施加零权重。

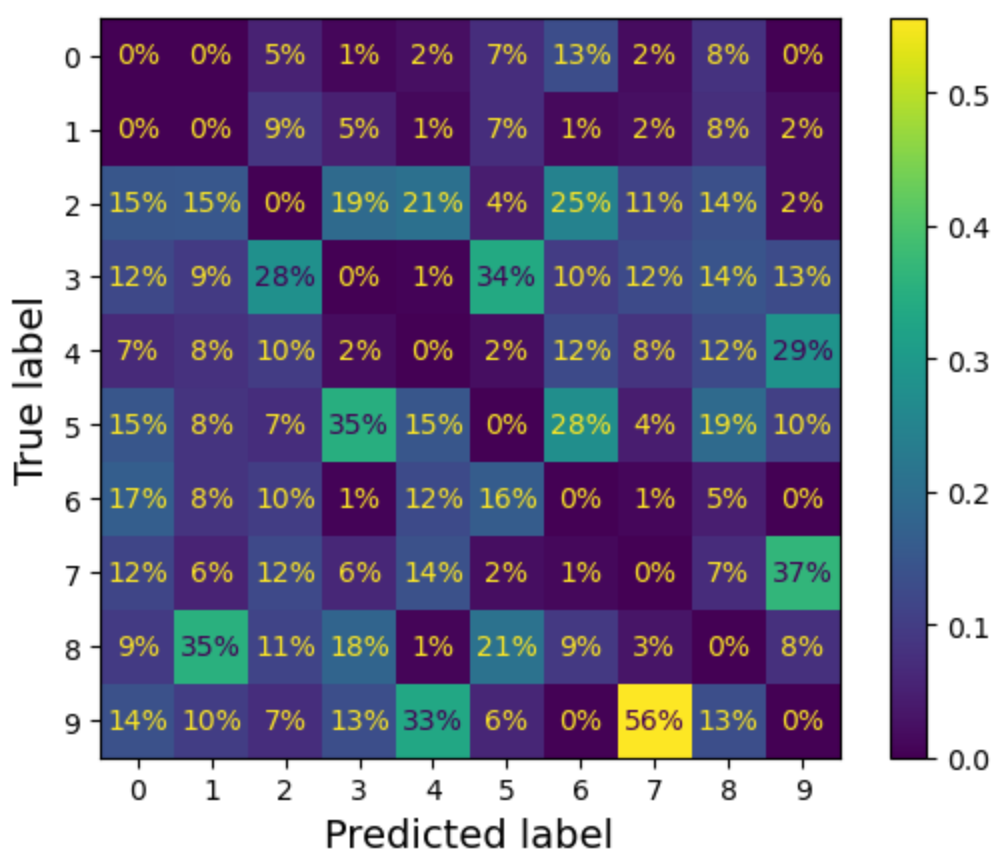
[illegible]



现在您可以更清楚地看到分类器所犯的错误类型。第8级的列现在非常明亮，这证实了许多图像被误定义为8。事实上，这是几乎所有类中最常见的错误分类。但是要小心你如何解释这个图表中的百分比：记住，我们已经排除了正确的预测。例如，第7行，第9列中的36%并不意味着7的所有图像中的36%被误诊为9。这意味着模型在7图像上所犯的错误中有36%是错误分类为9。实际上，只有3%的7图像被误分类为9。

也可以按列而不是按行对混淆矩阵进行规范化：只需设置 **normalize="pred"**。例如，你可以看到56%的错误分类的7s实际上是9s。

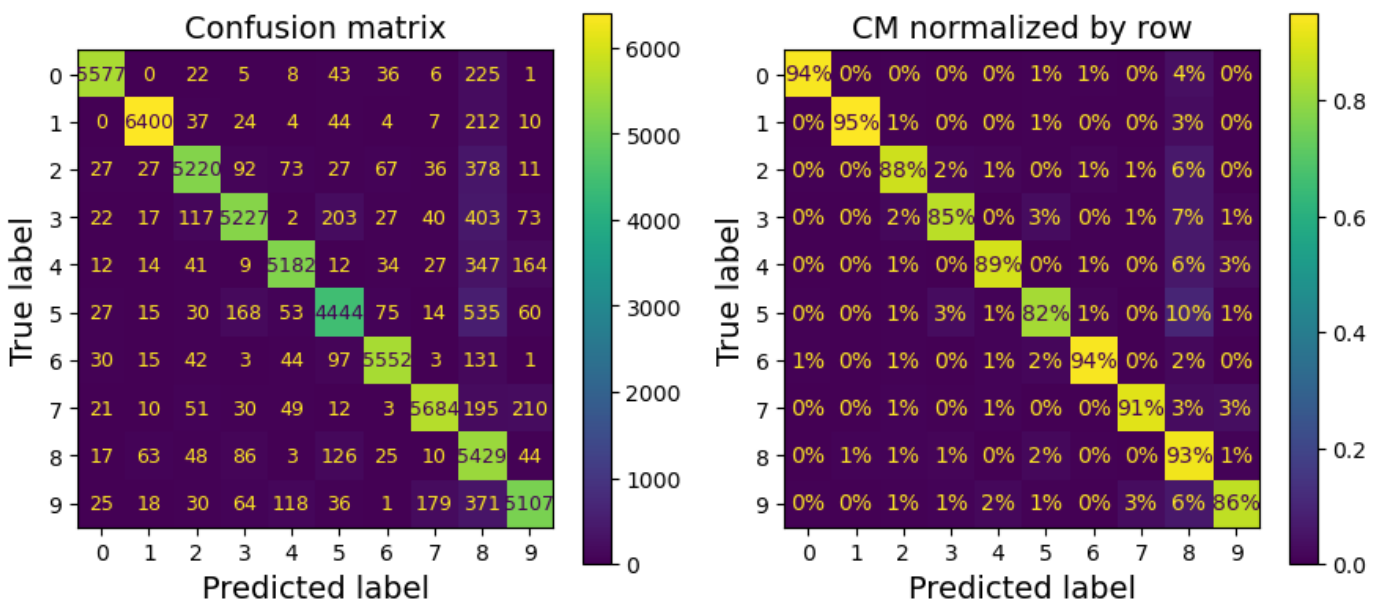
[illegible]



```
In [93]: # extra code - this cell generates and saves Figure 3-9
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(9, 4))
plt.rc('font', size=9)

ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred, ax=axes[0])
axes[0].set_title("Confusion matrix")
plt.rc('font', size=10)

ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred, ax=axes[1],
                                       normalize="true", values_format=".0%")
axes[1].set_title("CM normalized by row")
save_fig("confusion_matrix_plot_1")
plt.show()
```



```
In [94]: # extra code - this cell generates and saves Figure 3-10
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(9, 4))
plt.rc('font', size=10)
```

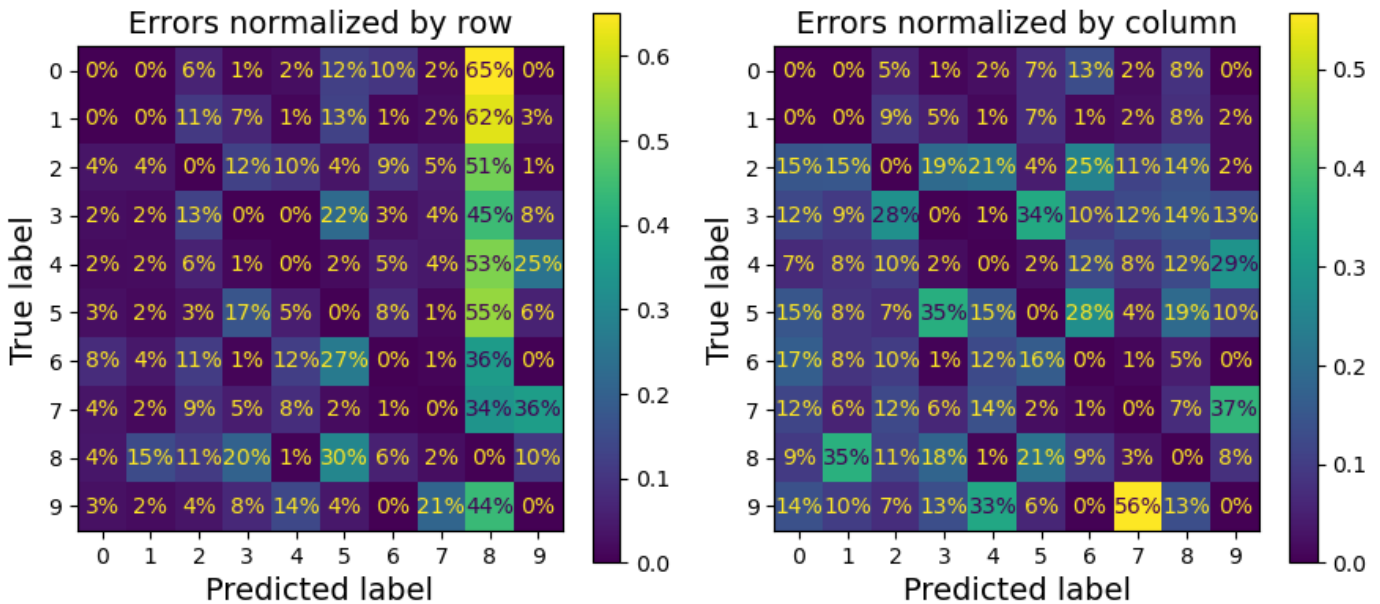
```

ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred, ax=axes[0],
                                       sample_weight=sample_weight,
                                       normalize="true", values_format=".0%")
axes[0].set_title("Errors normalized by row")

ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred, ax=axes[1],
                                       sample_weight=sample_weight,
                                       normalize="pred", values_format=".0%")
axes[1].set_title("Errors normalized by column")

save_fig("confusion_matrix_plot_2")
plt.show()
plt.rc('font', size=14) # make fonts great again

```



分析混淆矩阵通常会让您深入了解改进分类器的方法。似乎您的努力应该花在减少8的错误上。例如，您可以尝试为看起来像8（但不是）的数字收集更多的训练数据，以便分类器可以学习将它们与真实的8区分开来。或者您可以设计一些新的特性来帮助分类器——例如，编写一个算法来计算圈的数量（例如，8有2个，6有1个，5有没有圈）。或者您可以对图像进行预处理（例如，使用 **Scikit-Image**、**Pillow** 或 **OpenCV**）来制作一些模式，比如圈，更突出。

分析单个错误也是深入了解分类器在做什么以及为什么会失败的好方法。例如，让我们以混淆矩阵的风格绘制3和5的例子

```

In [95]: 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)]

In [96]: # extra code - this cell generates and saves Figure
size = 5
pad = 0.2
plt.figure(figsize=(size, size))

for images, (label_col, label_row) in [(X_ba, (0, 0)), (X_bb, (1, 0)),
                                       (X_aa, (0, 1)), (X_ab, (1, 1))]:
    for idx, image_data in enumerate(images[:size*size]):
        x = idx % size + label_col * (size + pad)
        y = idx // size + label_row * (size + pad)
        plt.imshow(image_data.reshape(28, 28),
                   cmap="binary",

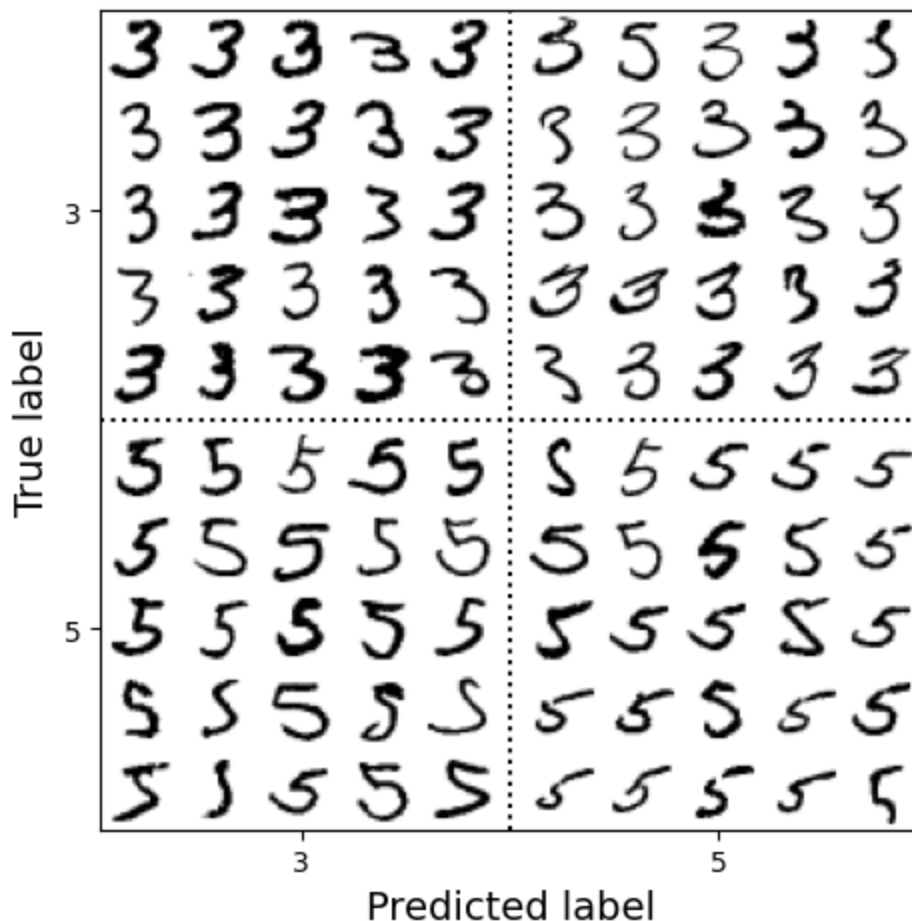
```

```

        extent=(x, x + 1, y, y + 1))
    plt.xticks([size / 2, size + pad + size / 2], [str(cl_a), str(cl_b)])

plt.yticks([size / 2, size + pad + size / 2], [str(cl_b), str(cl_a)])
plt.plot([size + pad / 2, size + pad / 2], [0, 2 * size + pad], "k:")
plt.plot([0, 2 * size + pad], [size + pad / 2, size + pad / 2], "k:")
plt.axis([0, 2 * size + pad, 0, 2 * size + pad])
plt.xlabel("Predicted label")
plt.ylabel("True label")
save_fig("error_analysis_digits_plot")
plt.show()

```



正如您所看到的，分类器出错的一些数字（即，在左下角和右上角的块中）写得非常糟糕，甚至连人类也无法对它们进行分类。然而，在我们看来，大多数错误分类的图像似乎都是明显的错误。也许很难理解为什么分类器会犯这样的错误，但请记住，人类的大脑是一个奇妙的模式识别系统，我们的视觉系统在任何信息到达我们的意识之前做了很多复杂的预处理。所以，这个任务感觉简单的事实并不意味着它是。回想一下，我们使用了一个简单的SGD分类器，它只是一个线性模型：它所做的就是给每个像素分配一个权重，当它看到一个新图像时，它只是总结加权像素强度，以得到每个类的分数。由于3和5的值相差只有几个像素，这个模型很容易混淆它们。

3和5之间的主要区别是连接顶部线到底部弧线的小线的位置。如果你绘制一个3，并且连接点轻微地向左移动，分类器可能会将其归类为一个5，反之亦然。换句话说，该分类器对图像的移动和旋转非常敏感。减少3/5混淆的一种方法是对图像进行预处理，以确保它们的中心良好，而不是过于旋转。然而，这可能并不容易，因为它需要预测每个图像的正确旋转。一种更简单的方法是用训练图像的稍微移动和旋转的变量来增强训练集。这将迫使模型学习对这种变化更加容忍。这被称为数据增强（我们将在第14章中介绍它；也请参见本章末尾的练习2）。

6. 多标签分类（Multilabel Classification）

到目前为止，每个实例总是只被分配给一个类。但在某些情况下，您可能希望分类器为每个实例输出多个类。考虑一个人脸识别分类器：如果它能识别同一张图片中的几个人，它应该怎么做？它应该给每个它能识别出的人贴上一个标签。假设分类器已经被训练成能够识别三张脸：爱丽丝、鲍勃和查理。然后，当分类器看到一张爱丽丝和查理的相片时，它应该输出 [真, 假, 真]（意思是“爱丽丝是，鲍勃不是，查理是”）。这种输出多个二进制标签的分类系统称为多标签分类系统。

我们还会讨论人脸识别，但让我们来看看一个更简单的例子，只是为了说明：

```
In [98]: 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)
```

```
Out[98]: ▼ KNeighborsClassifier
KNeighborsClassifier()
```

这段代码为每个数字图像创建一个包含每个数字图像的两个目标标签的 **y_multilabel** 数组：第一个表示该数字是否较大（7、8或9），第二个表示它是否为奇数。然后代码创建一个 **KNeighborsClassifier** 实例，它支持多标签分类（不是所有的分类器都支持），并使用多目标数组训练这个模型。现在你可以做出一个预测，并注意到它输出了两个标签：

```
In [100]: knn_clf.predict([some_digit])
```

```
Out[100]: array([[False,  True]])
```

而且它是对的！数字5确实不大（False）并且是奇数（True）。

有很多方法来评估多标签分类器，选择正确的度量实际取决于您的项目。一种方法是测量每个单独标签的分数（F1，或前面讨论的任何其他二进制分类器度量），然后简单地计算平均分数。下面的代码计算了所有标签的平均F1分数：

```
In [103]: 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")
```

```
Out[103]: 0.976410265560605
```

这种方法假设所有的标签都同样重要，但情况可能并非如此。特别是，如果你有爱丽丝的照片比鲍勃或查理的多的话，你可能想要更重视分类器对爱丽丝的照片的评分。一个简单的选择是给每个标签一个等于其支持度的权重（即，具有该目标标签的实例的数量）。要做到这一点，在调用 **f1_score()** 函数时，只需设置 **average="weighted"**。

```
In [104]: # extra code - shows that we get a negligible performance improvement when we
#           set average="weighted" because the classes are already pretty
#           well balanced.

f1_score(y_multilabel, y_train_knn_pred, average="weighted")
```

```
Out[104]: 0.9778357403921755
```


如果您希望使用一个本地不支持多标签分类的分类器，例如 **SVC**，一种可能的策略是为每个标签训练一个模型。然而，这种策略可能很难捕获标签之间的依赖关系。例如，一个大数字（7、8或9）是奇数的可能性是偶数的两倍，但是“奇数”标签的分类器不知道“大”标签的分类器预测了什么。为了解决这个问题，模型可以组织在一个链中：当一个模型做出预测时，它使用输入特征加上链中它之前的模型的所有预测。

好消息是，Scikit-Learn有一个叫做 **ChainClassifier** 的类，它就可以这样做！默认情况下，它将使用真正的标签进行训练，根据它们在链中的位置向每个模型提供适当的标签。但如果你设置了 **cv** 超参数，它将使用交叉验证从训练集中的每个实例的每个训练模型中获得“干净”（样本外）预测，然后这些预测将用于训练链中的所有模型。

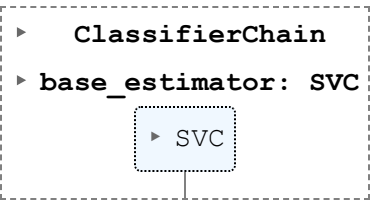
这里有一个例子，展示了如何使用交叉验证策略来创建和训练 **ChainClassifier**。与之前一样，我们将只使用训练集中的前2000张图像来加快速度：

```
In [105]: from sklearn.multioutput import ClassifierChain

chain_clf = ClassifierChain(SVC(), cv=3, random_state=42)

chain_clf.fit(X_train[:2000], y_multilabel[:2000])
```

```
Out[105]:
```



```
In [106]: chain_clf.predict([some_digit])
```

```
Out[106]: array([[0., 1.]])
```

7. 多输出分类（Multioutput Classification）

我们将在这里讨论的最后一种分类任务称为 **多输出-多类分类**（或只是**多输出分类**）。它是多标签分类的一种推广，其中每个标签都可以是多类的（即，它可以有两个以上的可能的值）。

为了说明这一点，让我们建立一个可以消除图像中的噪声的系统。它将以一个嘈杂的数字图像作为输入，并且它将（希望）输出一个干净的数字图像，表示为像素强度数组，就像MNIST图像一样。请注意，分类器的输出是多标签（每个像素一个标签），每个标签可以有多个值（像素强度范围从0到255）。因此，这是一个多输出分类系统的一个例子。

注意：分类和回归之间的界限有时是模糊的，如在本例中。可以说，预测像素强度更类似于回归，而不是分类。此外，多输出系统并不局限于分类任务；您甚至可以有一个系统，它为每个实例输出多个标签，包括类标签和值标签。

让我们首先创建训练和测试集，获取MNIST图像，并使用NumPy的 **randint()** 函数添加噪声。目标图像将是原始图像：

```
In [107]: np.random.seed(42) # to make this code example reproducible

noise = np.random.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise

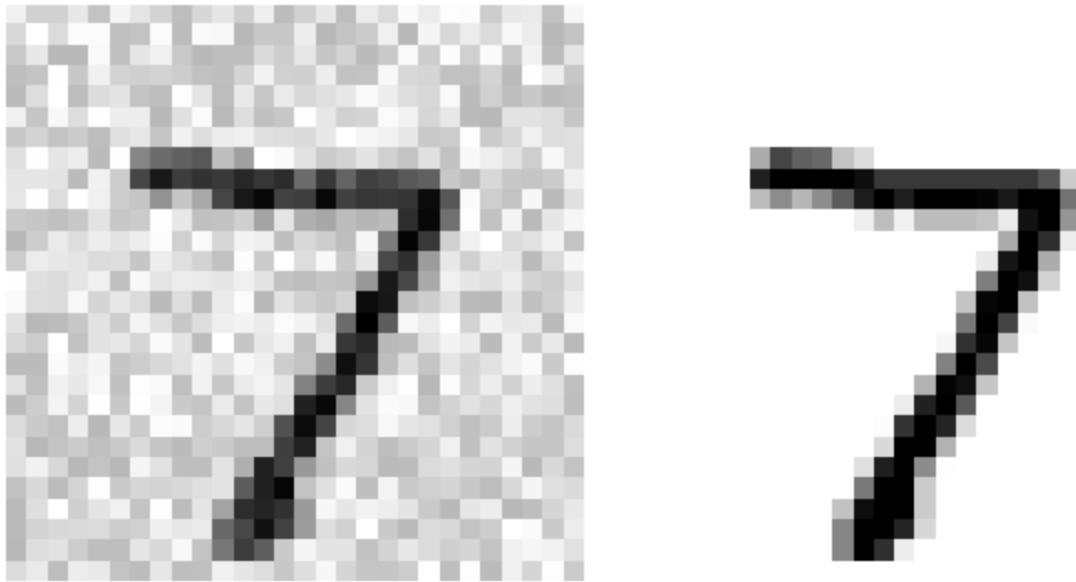
noise = np.random.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise
```

```
y_train_mod = X_train
y_test_mod = X_test
```

让我们来看看测试集中的第一张图像。是的，我们正在窥探测试数据，所以你现在应该皱眉。

```
In [108... # extra code - this cell generates and saves Figure 3-12
plt.subplot(121); plot_digit(X_test_mod[0])
plt.subplot(122); plot_digit(y_test_mod[0])

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

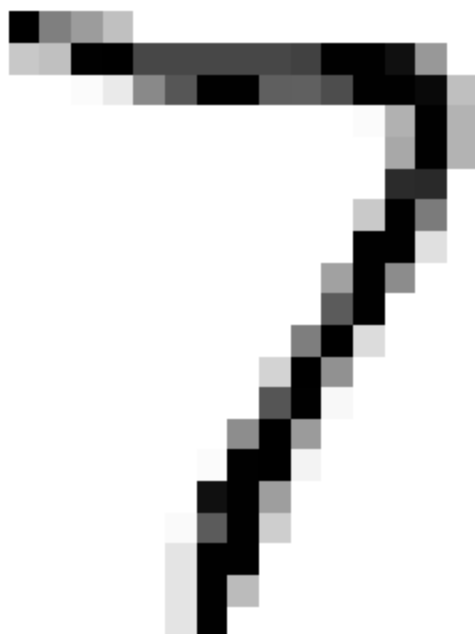


左边是有噪声的输入图像，右边是干净的目标图像。现在让我们训练分类器，让它清理这个图像：

```
In [110... knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train_mod, y_train_mod)

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

save_fig("cleaned_digit_example_plot") # extra code - saves Figure 3-13
plt.show()
```



看起来已经足够接近目标了！这就是我们的分类之旅的结束。您现在知道如何为分类任务选择良好的度量标准，选择适当的精度/召回率权衡，比较分类器，以及更普遍地为各种任务构建良好的分类系统。在下一章中，您将了解您所使用的所有这些机器学习模型是如何实际工作的。