Classification

- Ideas and objects are recognized, differentiated and understood

分类系统

分类

- 最常见的监督式学习任务包括回归任务（预测值）和分类任务（预测类）。前面探讨了一个回归任务 — 预测住房价格，用到了线性回归、决策树以及随机森林等各种算法（我们将会在后续部分中进一步讲解这些算法）。本节我们将把注意力转向分类系统。

MNIST

- 本章将使用MNIST数据集，这是一组由美国高中生和人口调查局员工手写的70000个数字的图片。每张图像都用其代表的数字标记。这个数据集被广为使用，因此也被称作是机器学习领域的“Hello World”：但凡有人想到了一个新的分类算法，都会想看看在MNIST上的执行结果。因此只要是学习机器学习的人，早晚都要面对MNIST。
- Scikit-Learn提供了许多助手功能来帮助你下载流行的数据集。MNIST也是其中之一。下面是获取MNIST数据集的代码：

MNIST

```
>>> from sklearn.datasets import fetch_mldata
>>> mnist = fetch_mldata('MNIST original')
>>> mnist
{'COL_NAMES': ['label', 'data'],
'DESCR': 'mldata.org dataset: mnist-original',
'data': 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]], dtype=uint8),
'target': array([ 0., 0., 0., ..., 9., 9., 9.])}
```

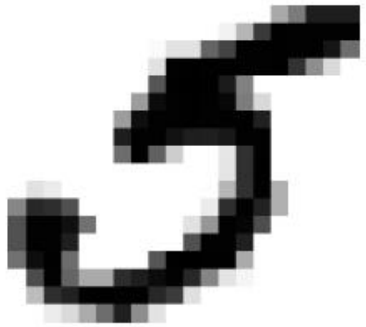
MNIST

- Scikit-Learn加载的数据集通常具有类似的字典结构，包括：
  - DESCR键，描述数据集
  - data键，包含一个数组，每个实例为一行，每个特征为一列
  - target键，包含一个带有标记的数组

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

MNIST

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
some_digit = X[36000]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap = matplotlib.cm.binary,
                    interpolation="nearest")
plt.axis("off")
plt.show()
```



```
>>> y[36000]
5.0
```

MNIST

- MNIST数据集已经分成训练集（前6万张图像）和测试集（最后1万张图像）了：

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

- 我们先将训练集数据洗牌，这样能保证交叉验证时所有的折叠都差不多（你肯定不希望某个折叠丢失一些数字）。此外，有些机器学习算法对训练实例的顺序敏感，如果连续输入许多相似的实例，可能导致执行性能不佳。给数据集洗牌正是为了确保这种情况不会发生：

```
import numpy as np
shuffle_index = np.random.permutation(60000)
X_train, y_train = X_train[shuffle_index], y_train[shuffle_index]
```

训练一个二元分类器

- 现在，先简化问题，只尝试识别一个数字——比如数字5。那么这个“数字5检测器”就是一个二元分类器的例子，它只能区分两个类别：5和非5。先为此分类任务创建目标向量：

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

Training a Binary Classifier

- 接着挑选一个分类器并开始训练。一个好的初始选择是随机梯度下降（SGD）分类器，使用Scikit-Learn的SGDClassifier类即可。这个分类器的优势是，能够有效处理非常大型的数据集。这部分是因为SGD独立处理训练实例，一次一个（这也使得SGD非常适合在线学习），稍后我们将会看到。此时先创建一个SGDClassifier并在整个训练集上进行训练：

```
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], dtype=bool)
```

性能考核

- 评估分类器比评估回归器要困难得多，我们可以用cross_val_score（）函数来评估SGDClassifier模型，采用K-fold交叉验证法（3折）。记住，K-fold交叉验证的意思是将训练集分解成K个折叠（在本例中，为3折），然后每次留其中1个折叠进行预测，剩余的折叠用来训练（参见第2章）：

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([ 0.9502 , 0.96565, 0.96495])
```

性能考核

```
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)
```

- 能猜到这个模型的准确度吗？看看：

```
>>> never_5_clf = Never5Classifier()
>>> cross_val_score(never_5_clf, X_train, y_train_5, cv=3,
scoring="accuracy")
array([ 0.909 , 0.90715, 0.9128 ])
```

性能考核

- 这说明准确率通常无法成为分类器的首要性能指标，特别是当你处理偏斜数据集（skewed dataset）的时候（即某些类比其他类更为频繁）。

混淆矩阵

- 评估分类器性能的更好方法是混淆矩阵。总体思路就是统计A类别实例被分成为B类别的次数。例如，要想知道分类器将数字3和数字5混淆多少次，只需要通过混淆矩阵的第5行第3列来查看。

混淆矩阵

- 要计算混淆矩阵，需要先有一组预测才能将其与实际目标进行比较。当然可以通过测试集来进行预测，但是现在先不要动它（测试集最好留到项目最后，准备启动分类器时再使用）。作为替代，可以使用cross_val_predict（）函数：

```
from sklearn.model_selection import cross_val_predict
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

混淆矩阵

- 与cross_val_score（）函数一样，cross_val_predict（）函数同样执行K-fold交叉验证，但返回的不是评估分数，而是每个折叠的预测。这意味着对于每个实例都可以得到一个干净的预测（“干净”的意思是模型预测时使用的数据，在其训练期间从未见过）。

混淆矩阵

- 现在，可以使用confusion_matrix（）函数来获取混淆矩阵了。只需要给出目标类别（y_train_5）和预测类别（y_train_pred）即可：

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_train_5, y_train_pred)
array([[53272, 1307],
       [ 1077, 4344]])
```

混淆矩阵

array([[53272, 1307],
[1077, 4344]])

- 混淆矩阵中的行表示实际类别，列表示预测类别。本例中第一行表示所有“非5”（负类）的图片中：53272张被正确地分为“非5”类别（真负类），1307张被错误地分类成了“5”（假正类）；第二行表示所有“5”（正类）的图片中：1077张被错误地分为“非5”类别（假负类），4344张被正确地分在了“5”这一类别（真正类）。

混淆矩阵

- 正类预测的准确率是一个有意思的指标，它也称为分类器的精度（公式3-1）：

Equation 3-1. Precision

$$precision = TP/(TP + FP)$$

- TP是真正类的数量，FP是假正类的数量。

混淆矩阵

- 做一个单独的正类预测，并确保它是正确的，就可以得到完美精度（精度＝1/1＝100%）。但这没什么意义，因为分类器会忽略这个正类实例之外的所有内容。因此，精度通常与另一个指标一起使用，这个指标就是召回率（recall），也称为灵敏度（sensitivity）或者真正类率（TPR）：它是分类器正确检测到的正类实例的比率（公式3-2）：

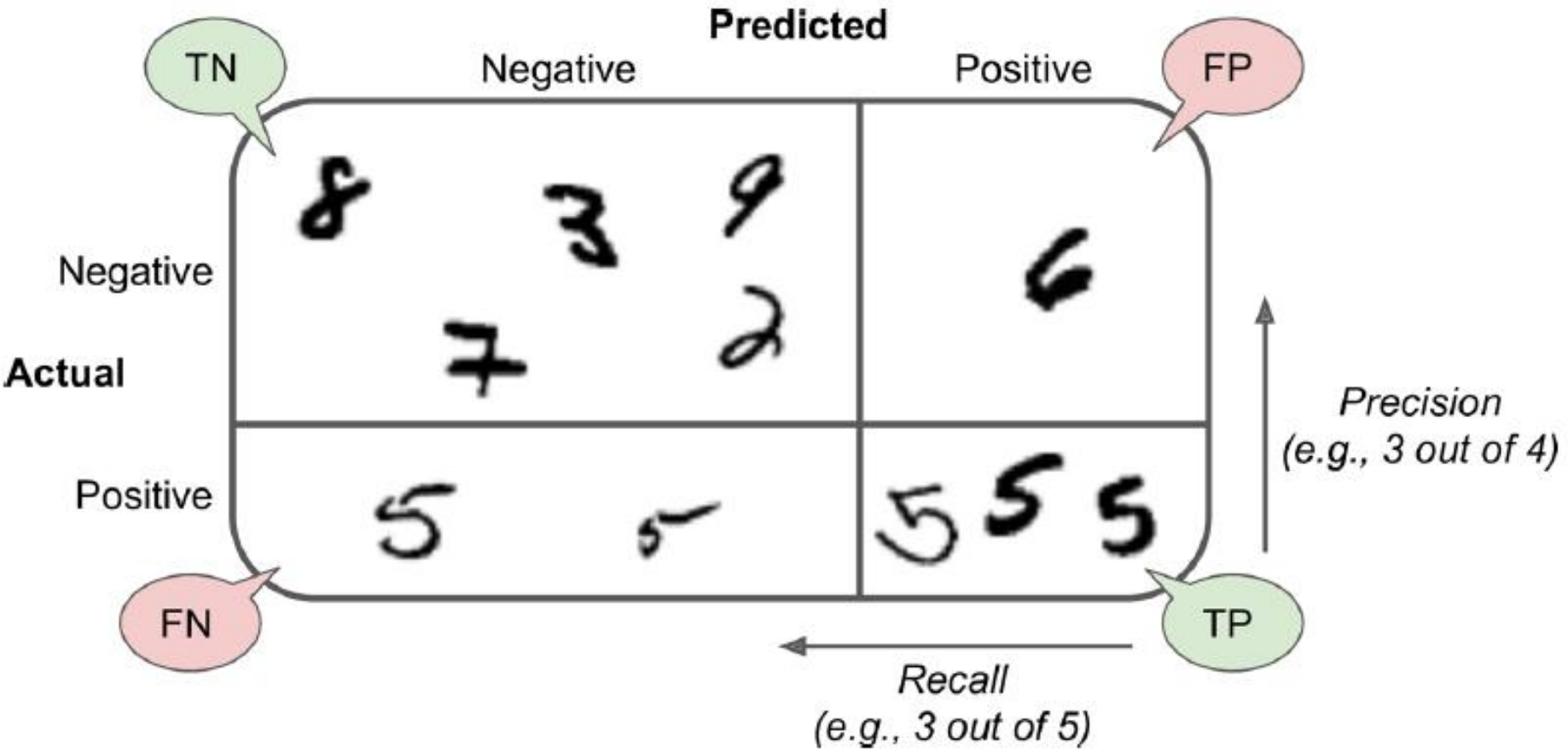
Equation 3-2. Recall

$$\text{recall} = TP/(TP + FN)$$

- FN是假负类的数量。

混淆矩阵

图解混淆矩阵


Predicted
TN
Negative
Positive
FP
Negative
Actual
Positive
FN
TP
Precision
(e.g., 3 out of 4)
Recall
(e.g., 3 out of 5)

精度/召回率

- Scikit-Learn提供了计算多种分类器指标的函数，精度和召回率也是其一：

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred)
# == 4344 / (4344 + 1307)
0.7687135020350381

>>> recall_score(y_train_5, y_train_pred)
# == 4344 / (4344 + 1077)
0.7913669064748201
```

精度/召回率

- 我们可以很方便地将精度和召回率组合成一个单一的指标，称为F_1分数。当你需要一个简单的方法来比较两种分类器时，这是个非常不错的指标。F_1分数是精度和召回率的调和平均值（见公式3-3）。正常的平均值平等对待所有的值，而调和平均值会给予较低的值更高的权重。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F_1分数。

Equation 3-3. F_1 score

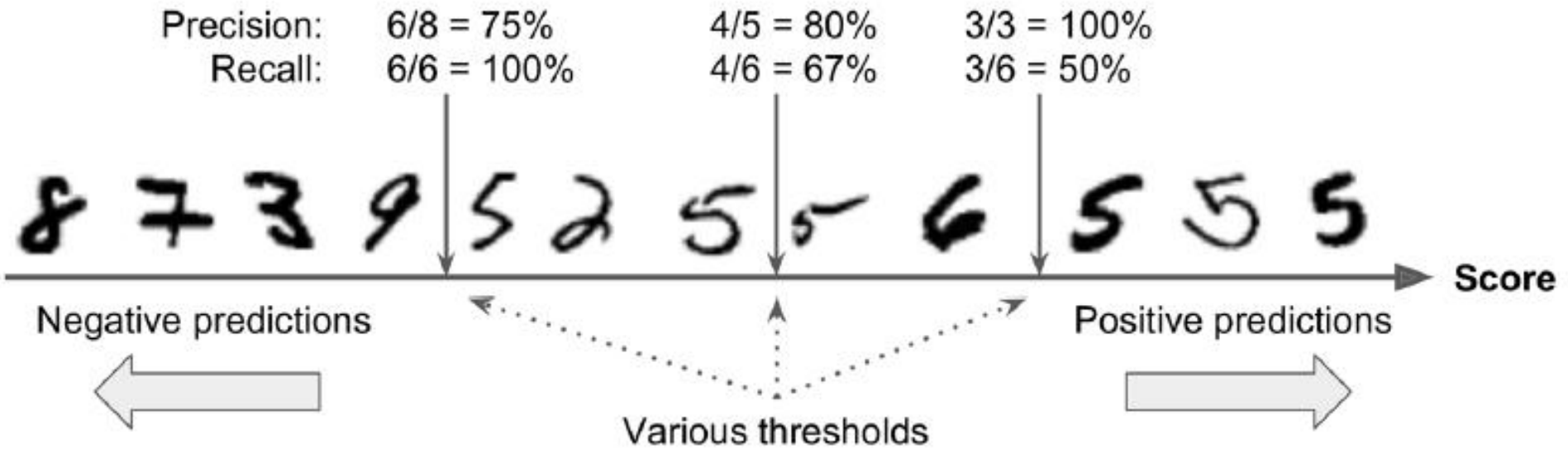
$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN + FP}{2}}$$

精度/召回率

- 要计算F_1分数，只需要调用f1_score（）即可：

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.7846820809248547
```

精度/召回率权衡


Precision:
Recall:
6/8 = 75%
6/6 = 100%
4/5 = 80%
4/6 = 67%
3/3 = 100%
3/6 = 50%
Score
Negative predictions
Positive predictions
Various thresholds

精度/召回率权衡

- Scikit-Learn不允许直接设置阈值，但是可以访问它用于预测的决策分数。不是调用分类器的predict（）方法，而是调用decision_function（）方法，这个方法返回每个实例的分数，然后就可以根据这些分数，使用任意阈值进行预测了：

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([ 161855.74572176])
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
array([ True], dtype=bool)
```

精度/召回率权衡

- SGDClassifier分类器使用的阈值是0，所以前面的代码返回结果与predict（）方法一样（也就是True）。我们来试试提升阈值：

```
>>> threshold = 200000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False], dtype=bool)
```

- 这证明了提高阈值确实可以降低召回率。这张图确实是5，当阈值为0时，分类器可以检测到该图，但是当阈值提高到200000时，就错过了这张图。

精度/召回率权衡

- 那么要如何决定使用什么阈值呢？首先，使用cross_val_predict（）函数获取训练集中所有实例的分数，但是这次需要它返回的是决策分数而不是预测结果：

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

- 有了这些分数，可以使用precision_recall_curve（）函数来计算所有可能的阈值的精度和召回率：

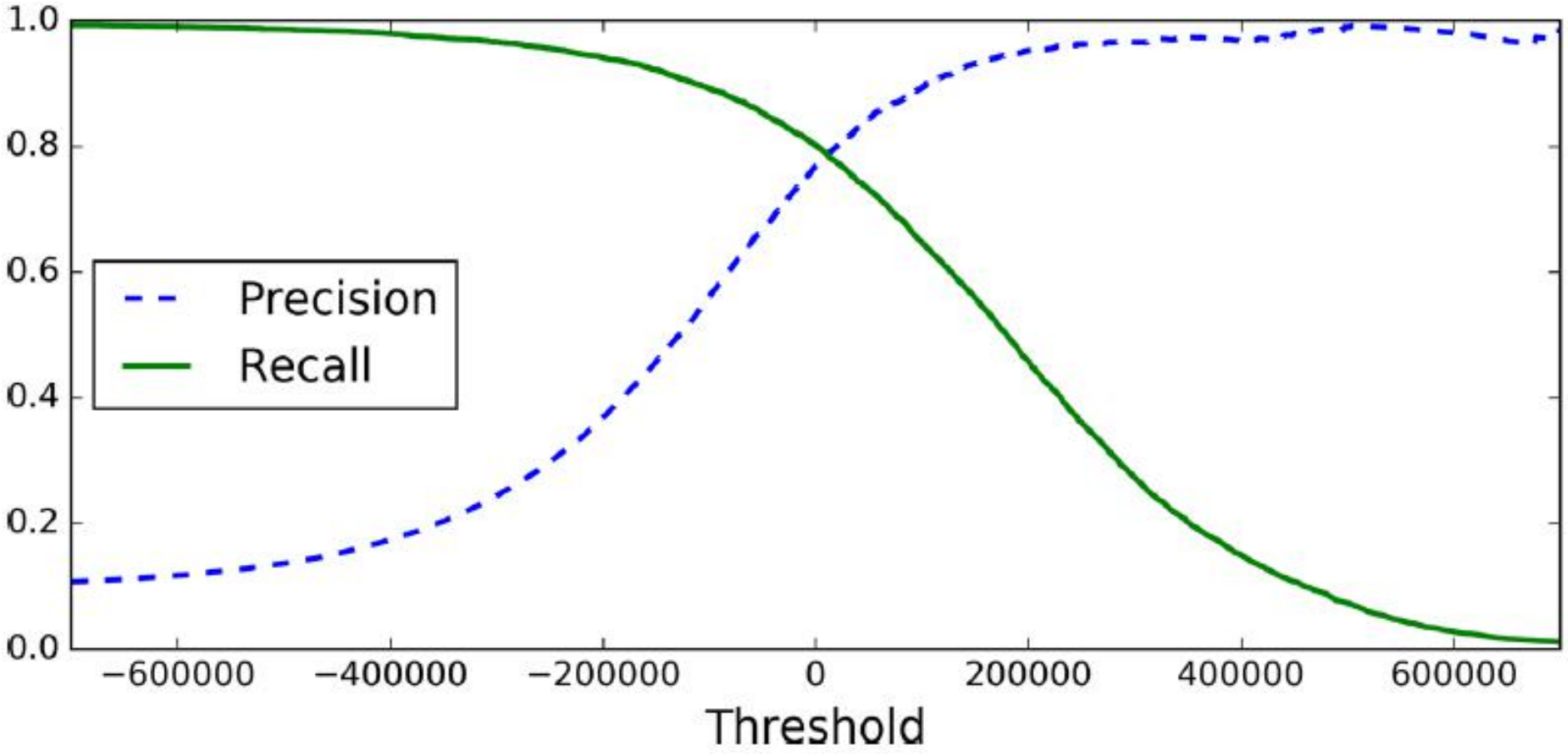
精度/召回率权衡

```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

- 最后，使用Matplotlib绘制精度和召回率相对于阈值的函数图（见图3-4）：

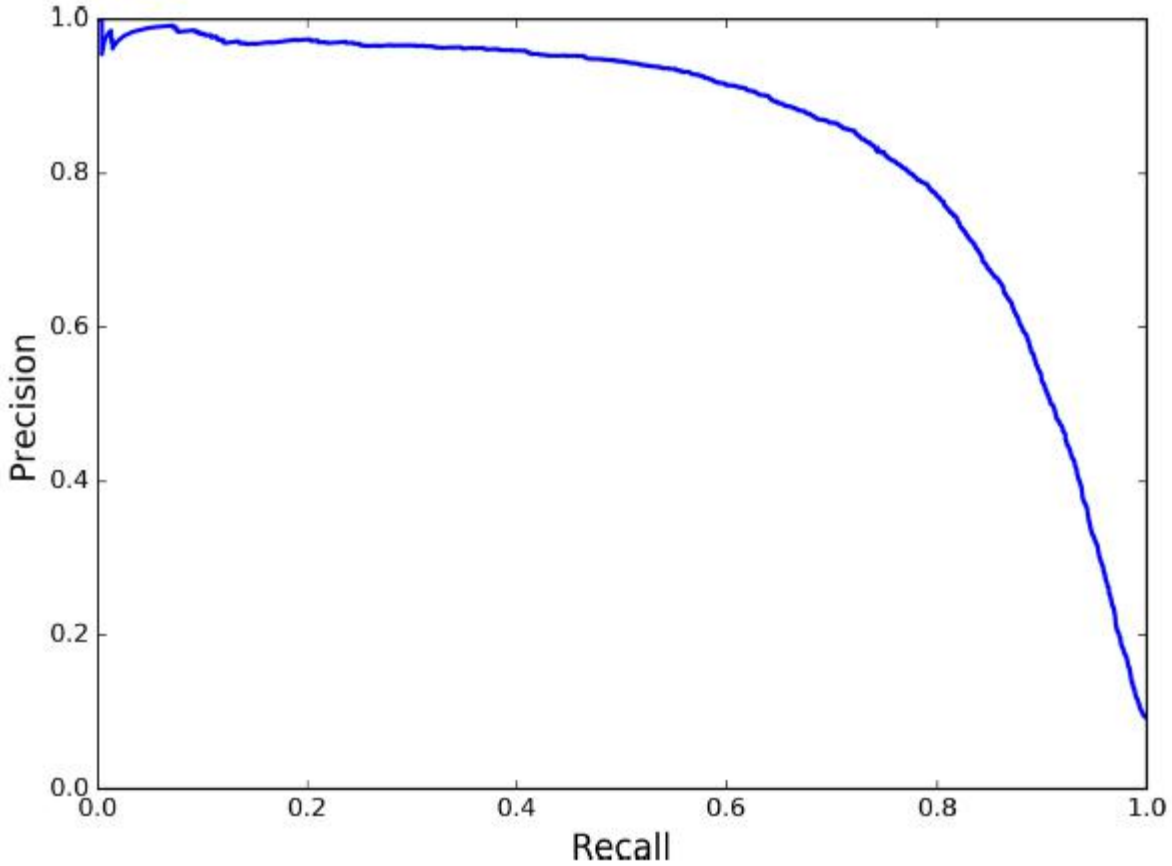
```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    plt.xlabel("Threshold")
    plt.legend(loc="upper left")
    plt.ylim([0, 1])

plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.show()
```

精度/召回率权衡


1.0
0.8
0.6
0.4
0.2
0.0
Precision
Recall
−600000
−400000
−200000
0
200000
400000
600000
Threshold

精度/召回率权衡

- 现在，就可以通过轻松选择阈值来实现最佳的精度/召回率权衡了。还有一种找到好的精度/召回率权衡的方法是直接绘制精度和召回率的函数图，如图3-5所示。



精度/召回率权衡

- 假设你决定瞄准90%的精度目标。通过绘制的第一张图（放大一点），得出需要使用的阈值大概是70000。要进行预测（现在是在训练集上），除了调用分类器的predict（）方法，也可以运行这段代码：

```
y_train_pred_90 = (y_scores > 70000)
```

- 我们检查一下这些预测结果的精度和召回率：

```
>>> precision_score(y_train_5, y_train_pred_90)
0.899870298313878 1
>>> recall_score(y_train_5, y_train_pred_90)
0.6399188341634385 3
```

ROC曲线

- 还有一种经常与二元分类器一起使用的工具，叫作受试者工作特征曲线（简称ROC）。它与精度/召回率曲线非常相似，但绘制的不是精度和召回率，而是真正类率（召回率的另一名称）和假正类率（FPR）。FPR是被错误分为正类的负类实例比率。它等于1减去真负类率（TNR），后者是被正确分类为负类的负类实例比率，也称特异度（*specificity*）。因此，ROC曲线绘制的是灵敏度 *sensitivity* 和 1-*specificity* 的关系。

ROC曲线

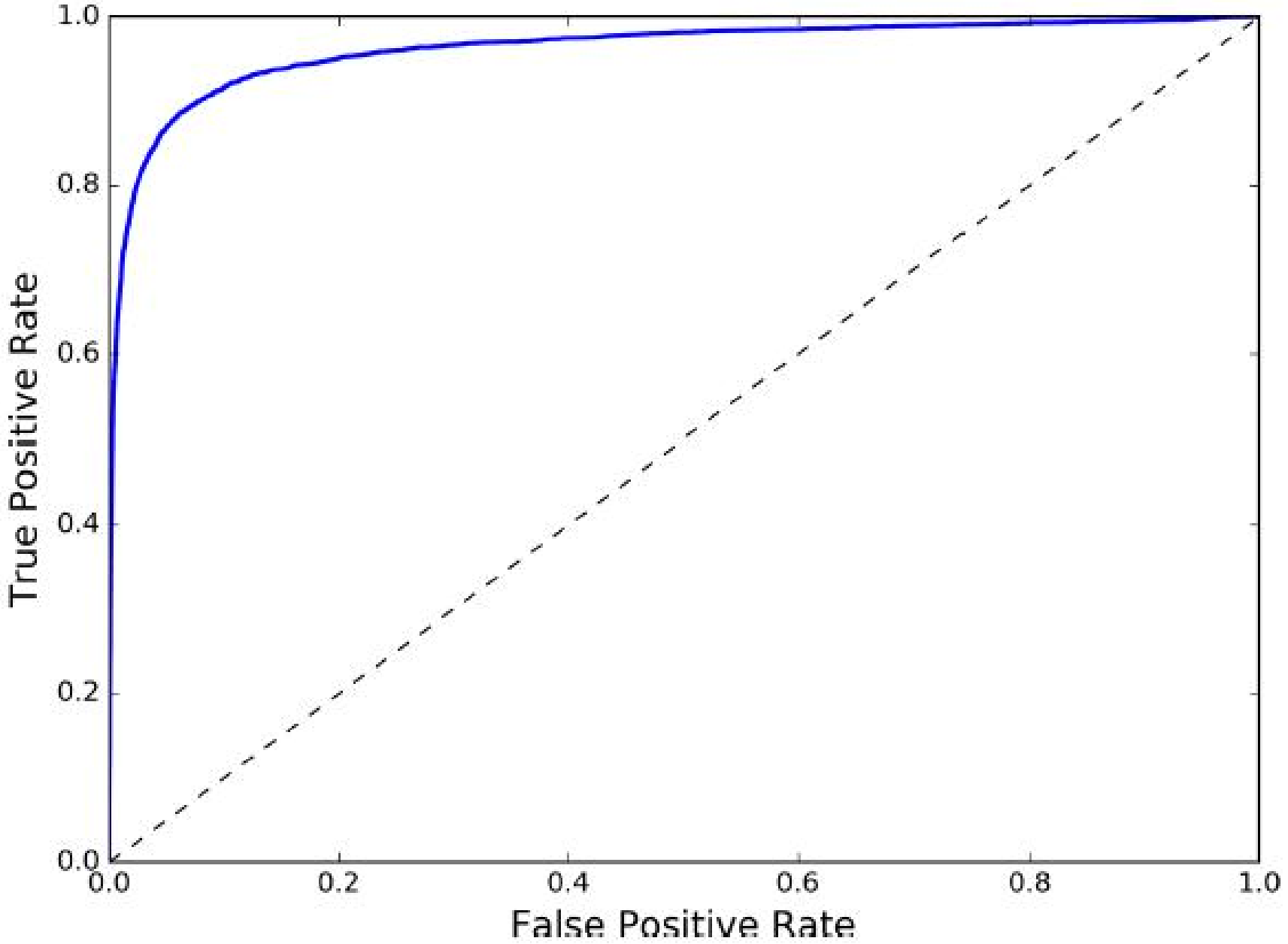
- 要绘制ROC曲线，首先需要使用roc_curve（）函数计算多种阈值的TPR和FPR：

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

- 然后，使用Matplotlib绘制FPR对TPR的曲线。下面的代码可以绘制出图3-6的曲线：

```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
plot_roc_curve(fpr, tpr)
plt.show()
```

ROC曲线


1.0
0.8
0.6
0.4
0.2
0.0
True Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate

ROC曲线

- 同样这里再次面临一个折中权衡：召回率（TPR）越高，分类器产生的假正类（FPR）就越多。虚线表示纯随机分类器的ROC曲线；一个优秀的分类器应该离这条线越远越好（向左上角）。
- 有一种比较分类器的方法是测量曲线下面积（AUC）。完美的分类器的ROC AUC等于1，而纯随机分类器的ROC AUC等于0.5。Scikit-Learn提供计算ROC AUC的函数：

```
>>> from sklearn.metrics import roc_auc_score
>>> roc_auc_score(y_train_5, y_scores)
0.9706107279717494
```

ROC曲线

- 训练一个RandomForestClassifier分类器，并比较它和SGDClassifier分类器的ROC曲线和ROC AUC分数。首先，获取训练集中每个实例的分数。但是由于它的工作方式不同（参见第7章），RandomForestClassifier类没有decision_function（）方法，相反，它有的是predict_proba（）方法。Scikit-Learn的分类器通常都会有这两种方法的其中一种。 predict_proba （）方法会返回一个数组，其中每行为一个实例，每列代表一个类别，意思是某个给定实例属于某个给定类别的概率（例如，这张图片有70%的可能是数字5）：

ROC曲线

```
from sklearn.ensemble import RandomForestClassifier
forest_clf = RandomForestClassifier(random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                    method="predict_proba")
```

- 但是要绘制ROC曲线，需要的是分数值而不是概率大小。一个简单的解决方案是：直接使用正类的概率作为分数值：

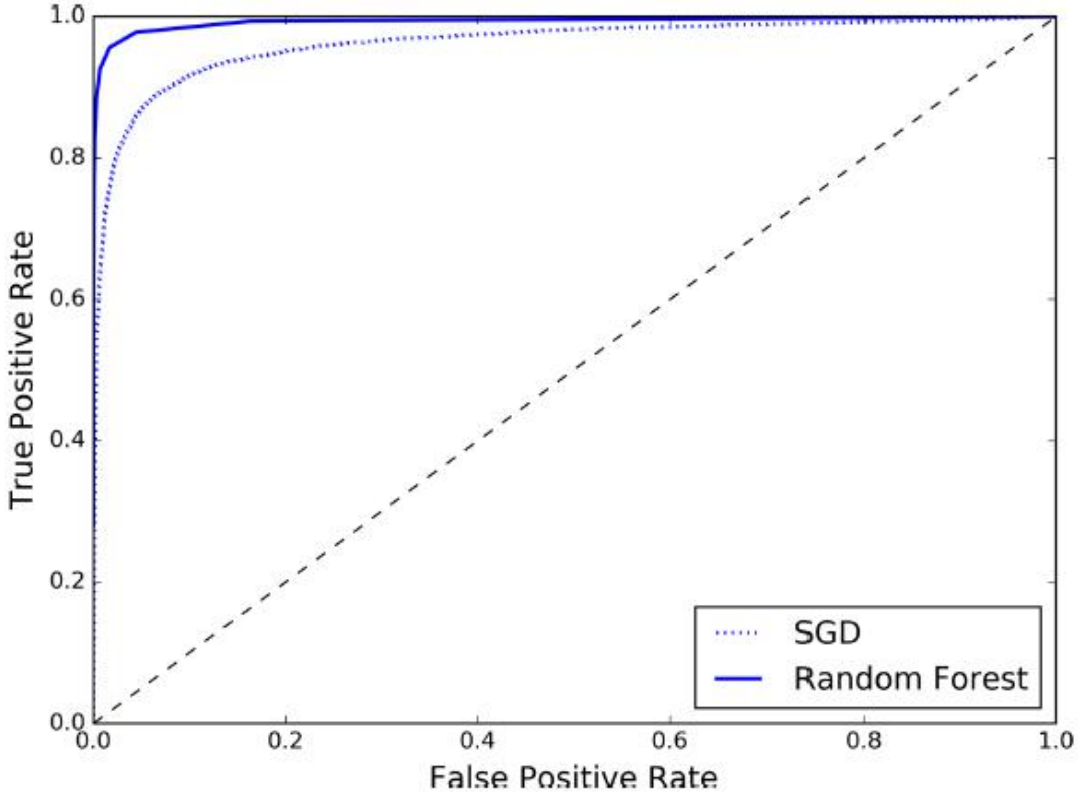
```
y_scores_forest = y_probas_forest[:, 1] # score = proba of positive class
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,y_scores_forest)
```

ROC曲线

- 现在可以绘制ROC曲线了。绘制第一条ROC曲线来看看对比结果（见图3-7）：

```
plt.plot(fpr, tpr, "b:", label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plt.legend(loc="bottom right")
plt.show()
```

```
>>> roc_auc_score(y_train_5,
                  y_scores_forest)
0.9931243660038291
```



ROC曲线

- 希望现在你已经掌握了如何训练二元分类器，如何选择合适的指标利用交叉验证来对分类器进行评估，如何选择满足需求的精度/召回率权衡，以及如何使用ROC曲线和ROC AUC分数来比较多个模型。我们再来试试对数字5之外的检测。

多类别分类器

- 二元分类器在两个类别中区分，而多类别分类器（也称为多项分类器）可以区分两个以上的类别。
- 有一些算法（如随机森林分类器或朴素贝叶斯分类器）可以直接处理多个类别。也有一些严格的二元分类器（如支持向量机分类器或线性分类器）。但是，有多种策略可以让你用几个二元分类器实现多类别分类的目的。

多类别分类器

- 例如，要创建一个系统将数字图片分为10类（从0到9），一种方法是训练10个二元分类器，每个数字一个（0-检测器、1-检测器、2-检测器，等等，以此类推）。然后，当你需要对一张图片进行检测分类时，获取每个分类器的决策分数，哪个分类器给分最高，就将其分为哪个类。这称为一对多（OvA）策略（也称为one-versus-the-rest）。

多类别分类器

- 另一种方法是，为每一对数字训练一个二元分类器：一个用于区分0和1，一个区分0和2，一个区分1和2，以此类推。这称为一对一（OvO）策略。如果存在N个类别，那么这需要训练N（N-1）/2个分类器。对于MNIST问题，这意味着要训练45个二元分类器！当需要对一张图片进行分类时，你需要运行45个分类器来对图片进行分类，最后看哪个类别获胜最多。OvO的主要优点在于，每个分类器只需要用到部分训练集对其必须区分的两个类别进行训练。

多类别分类器

- 有些算法（例如支持向量机分类器）在数据规模扩大时表现糟糕，因此对于这类算法，OvO是一个优先的选择，由于在较小训练集上分别训练多个分类器比在大型数据集上训练少数分类器要快得多。但是对大多数二元分类器来说，OvA策略还是更好的选择。

多类别分类器

- Scikit-Learn可以检测到你尝试使用二元分类算法进行多类别分类任务，它会自动运行OvA（SVM分类器除外，它会使用OvO）。我们用SGDClassifier试试：

```
>>> sgd_clf.fit(X_train, y_train) # y_train, not y_train_5
>>> sgd_clf.predict([some_digit])
array([ 5.])
```

多类别分类器

- 这段代码使用原始目标类别0到9（y_train）在训练集上对SGDClassifier进行训练，而不是以“5”和“剩余”作为目标类别（y_train_5）。然后做出预测（在本例中预测正确）。而在内部，Scikit-Learn实际上训练了10个二元分类器，获得它们对图片的决策分数，然后选择了分数最高的类别。

多类别分类器

- 想要知道是不是这样，可以调用decision_function（）方法。它会返回10个分数，每个类别1个，而不再是每个实例返回1个分数：

```
>>> some_digit_scores = sgd_clf.decision_function([some_digit])
>>> some_digit_scores
array([[-311402.62954431, -363517.28355739, -446449.5306454 ,
-183226.61023518, -414337.15339485, 161855.74572176,
-452576.39616343, -471957.14962573, -518542.33997148,
-536774.63961222]])
```

多类别分类器

```
>>> np.argmax(some_digit_scores)
5
>>> sgd_clf.classes_
array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
>>> sgd_clf.classes[5]
5.0
```

多类别分类器

- 如果想要强制Scikit-Learn使用一对一或者一对多策略，可以使用OneVsOneClassifier或OneVsRestClassifier类。只需要创建一个实例，然后将二元分类器传给其构造函数。例如，下面这段代码使用OvO策略，基于SGDClassifier创建了一个多类别分类器：

```
>>> from sklearn.multiclass import OneVsOneClassifier
>>> ovo_clf = OneVsOneClassifier(SGDClassifier(random_state=42))
>>> ovo_clf.fit(X_train, y_train)
>>> ovo_clf.predict([some_digit])
array([ 5.])
>>> len(ovo_clf.estimators_)
45
```

多类别分类器

- 训练RandomForestClassifier同样简单：

```
>>> forest_clf.fit(X_train, y_train)
>>> forest_clf.predict([some_digit])
array([ 5.])
```

- 这次Scikit-Learn不必运行OvA或者OvO了，因为随机森林分类器直接就可以将实例分为多个类别。调用predict_proba（）可以获得分类器将每个实例分类为每个类别的概率列表：

```
>>> forest_clf.predict_proba([some_digit])
array([[ 0.1, 0. , 0. , 0.1, 0. , 0.8, 0. , 0. , 0. , 0. ]])
```

多类别分类器

- 要评估这些分类器。跟之前一样，使用交叉验证。我们来试试使用cross_val_score（）函数评估一下SGDClassifier的准确率：

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([ 0.84063187, 0.84899245, 0.86652998])
```

多类别分类器

- 在所有的测试折叠上都超过了84%。如果是一个纯随机分类器，准确率大概是10%，所以这个结果不是太糟，但是依然有提升的空间。例如，将输入进行简单缩放（如第2章所述）可以将准确率提到90%以上：

```
>>> from sklearn.preprocessing import
    StandardScaler
>>> scaler = StandardScaler()
>>> X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
>>> cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
array([ 0.91011798, 0.90874544, 0.906636 ])
```

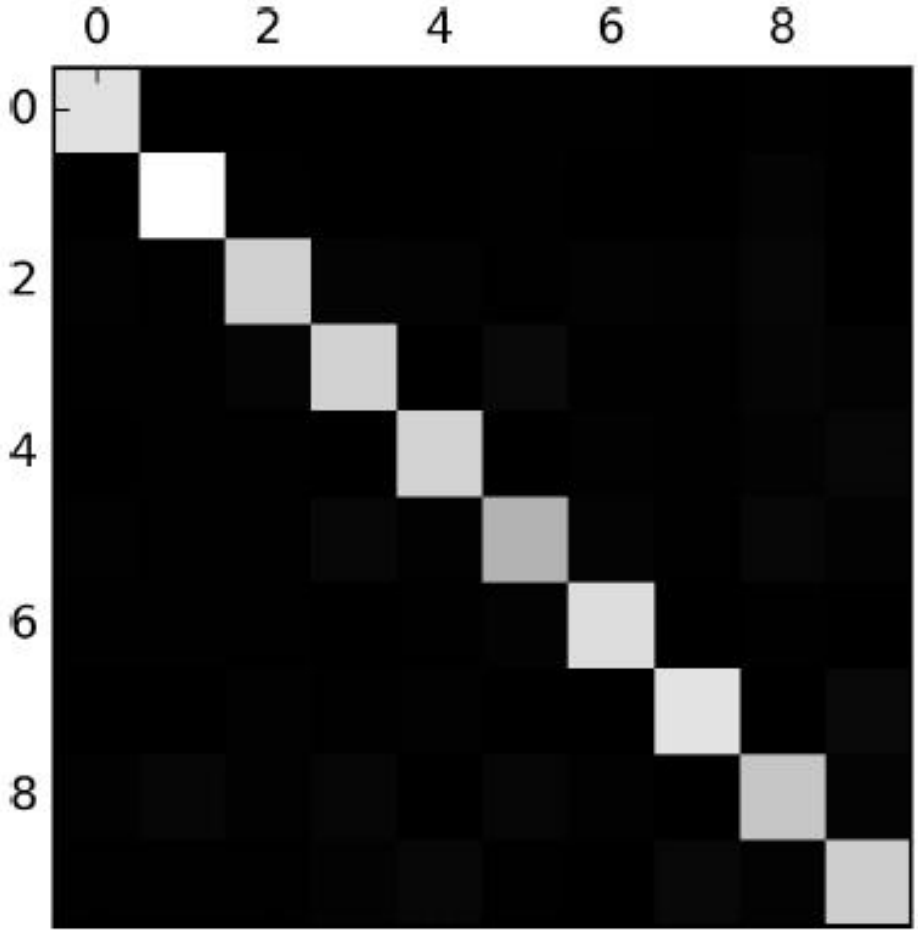
错误分析

- 假设你已经找到了一个有潜力的模型，现在你希望找到一些方法对其进一步改进。首先，看看混淆矩阵，使用cross_val_predict（）函数进行预测，然后调用confusion_matrix（）函数：

```
>>> 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([[5725, 3, 24, 9, 10, 49, 50, 10, 39, 4],
       [ 2, 6493, 43, 25, 7, 40, 5, 10, 109, 8],
       [ 51, 41, 5321, 104, 89, 26, 87, 60, 166, 13],
       [ 47, 46, 141, 5342, 1, 231, 40, 50, 141, 92],
       [ 19, 29, 41, 10, 5366, 9, 56, 37, 86, 189],
       [ 73, 45, 36, 193, 64, 4582, 111, 30, 193, 94],
       [ 29, 34, 44, 2, 42, 85, 5627, 10, 45, 0],
       [ 25, 24, 74, 32, 54, 12, 6, 5787, 15, 236],
       [ 52, 161, 73, 156, 10, 163, 61, 25, 5027, 123],
       [ 43, 35, 26, 92, 178, 28, 2, 223, 82, 5240]])
```

错误分析

```
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```



错误分析

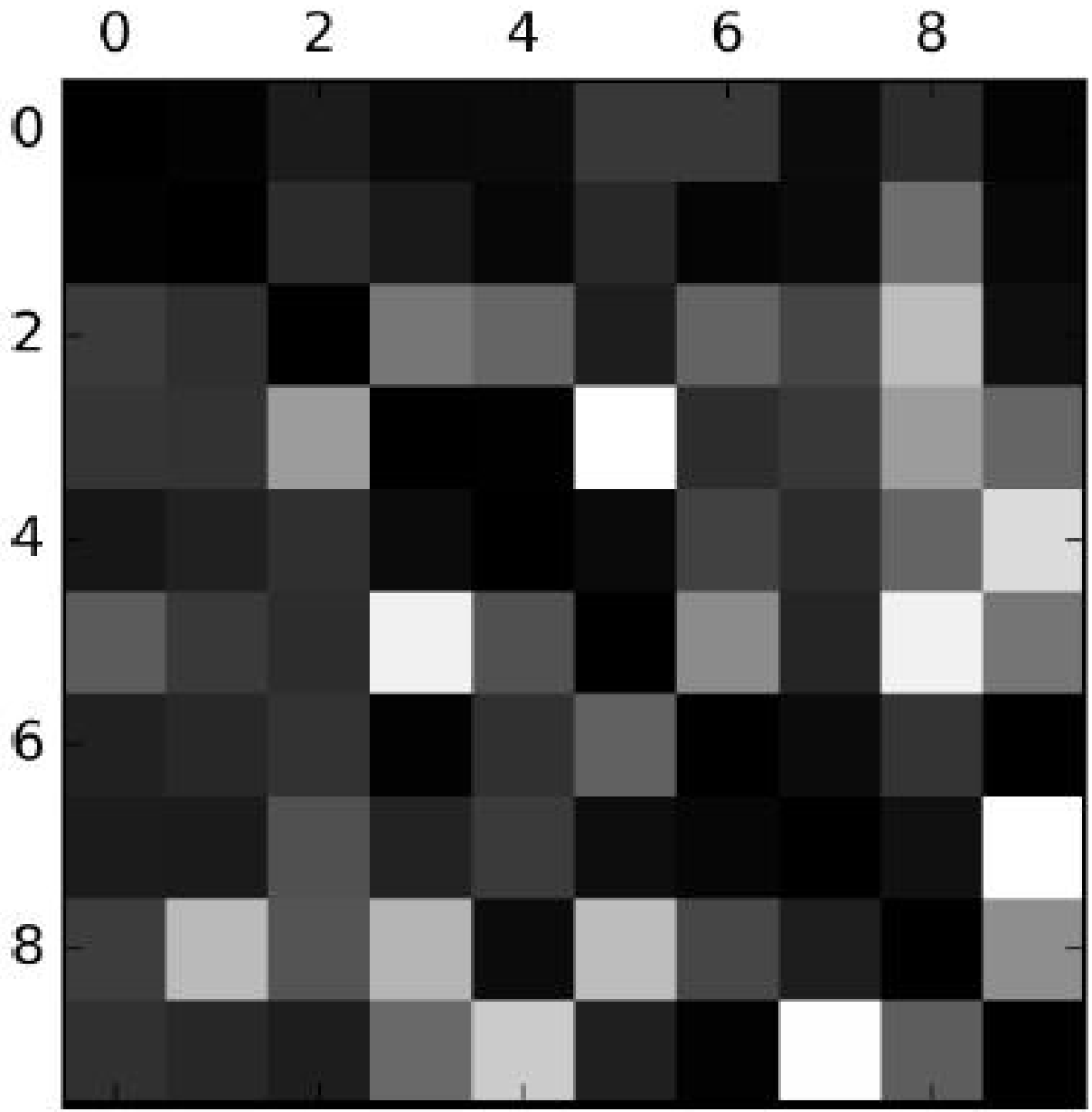
- 需要将混淆矩阵中的每个值除以相应类别中的图片数量，这样你比较的就是错误率而不是错误的绝对值（后者对图片数量较多的类别不公平）：

```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums
```

- 用0填充对角线，只保留错误分类，重新绘制：

```
np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()
```

错误分析



错误分析

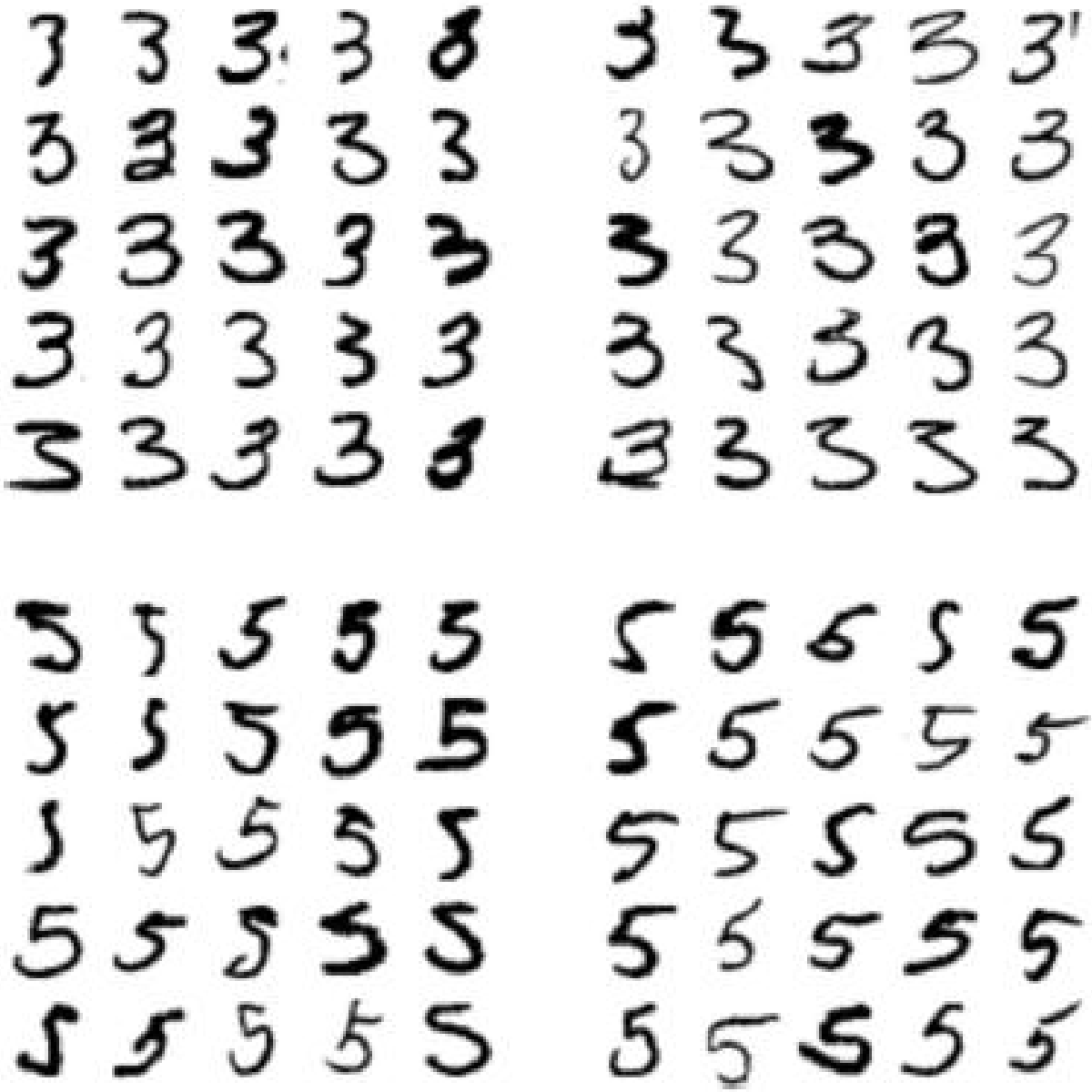
- 分析混淆矩阵通常可以帮助你深入了解如何改进分类器。通过上面那张图来看，你的精力可以花在改进数字8和数字9的分类，以及修正数字3和数字5的混淆上。例如，可以试着收集更多这些数字的训练数据。或者，也可以开发一些新特征来改进分类器——举个例子，写一个算法来计算闭环的数量（例如，数字8有两个，数字6有一个，数字5没有）。再或者，还可以对图片进行预处理（例如，使用Scikit-Image、Pillow或OpenCV）让某些模式更为突出，比如闭环之类的。

错误分析

- 分析单个的错误也可以为分类器提供洞察：它在做什么？它为什么失败？但这通常更加困难和耗时。例如，我们来看看数字3和数字5的例子：

```
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)]
plt.figure(figsize=(8,8))
plt.subplot(221); plot_digits(X_aa[:25], images_per_row=5)
plt.subplot(222); plot_digits(X_ab[:25], images_per_row=5)
```

错误分析



多标签分类

- 到目前为止，每个实例都只会被分在一个类别里。而在某些情况下，你希望分类器为每个实例产出多个类别。例如，人脸识别的分类器：如果在一张照片里识别出多个人怎么办？当然，应该为识别出来的每个人都附上一个标签。假设分类器经过训练，已经可以识别出三张脸—爱丽丝、鲍勃和查理，那么当看到一张爱丽丝和查理的照片时，它应该输出[1，0，1]（意思是“是爱丽丝，不是鲍勃，是查理”）这种输出多个二元标签的分类系统称为多标签分类系统。

多标签分类

- 让我们来看一个更为简单的例子：

```
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)
```

多标签分类

- 这段代码会创建一个y_multilabel数组，其中包含两个数字图片的目标标签：第一个表示数字是否是大数（7、8、9），第二个表示是否为奇数。下一行创建一个KNeighborsClassifier实例（它支持多标签分类，不是所有的分类器都支持），然后使用多个目标数组对它进行训练。现在用它做一个预测，注意它输出的两个标签：

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

多标签分类

- 评估多标签分类器的方法很多，如何选择正确的度量指标取决于你的项目。比如方法之一是测量每个标签的F_1分数（或者是之前讨论过的任何其他二元分类器指标），然后简单地平均。下面这段代码计算所有标签的平均F_1分数：

```
>>> y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_train, cv=3)
>>> f1_score(y_train, y_train_knn_pred, average="macro")
0.9684554018028022 1
```

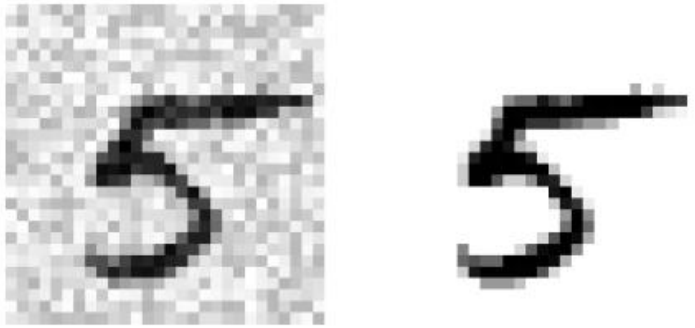
多输出分类

- 我们即将讨论的最后一种分类任务叫作多输出-多类别分类（或简单地称为多输出分类）。简单来说，它是多标签分类的泛化，其标签也可以是多种类别的（比如它可以有两个以上可能的值）。
- 为了说明这一点，构建一个系统去除图片中的噪声。给它输入一张有噪声的图片，它将（希望）输出一张干净的数字图片，跟其他MNIST图片一样，以像素强度的一个数组作为呈现方式。请注意，这个分类器的输出是多个标签（一个像素点一个标签），每个标签可以有多个值（像素强度范围为0到225）。所以这是个多输出分类器系统的例子。

多输出分类

- 先从创建训练集和测试集开始，使用NumPy的randint（）函数为MNIST图片的像素强度增加噪声。目标是将图片还原为原始图片：

```
noise = rnd.randint(0, 100, (len(X_train), 784))
noise = rnd.randint(0, 100, (len(X_test), 784))
X_train_mod = X_train + noise
X_test_mod = X_test + noise
y_train_mod = X_train
y_test_mod = X_test
```



多输出分类

```
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[some_index]])
plot_digit(clean_digit)
```

- 看起来离目标够接近了。分类器之旅到此结束。希望现在你掌握了如何为分类任务选择好的指标，如何选择适当的精度/召回率权衡，如何比较多个分类器，以及更为概括地说，如何为各种任务构建卓越的分类系统。