机器学习导论
第三章

王小航

分类

- MNIST数据集：
  - 分类数据集
  - 这是一组由美国高中生和人口调查局员工手写的70 000个数字的图片
  - 每张图片都用其代表的数字标记
  - 这个数据集被广为使用

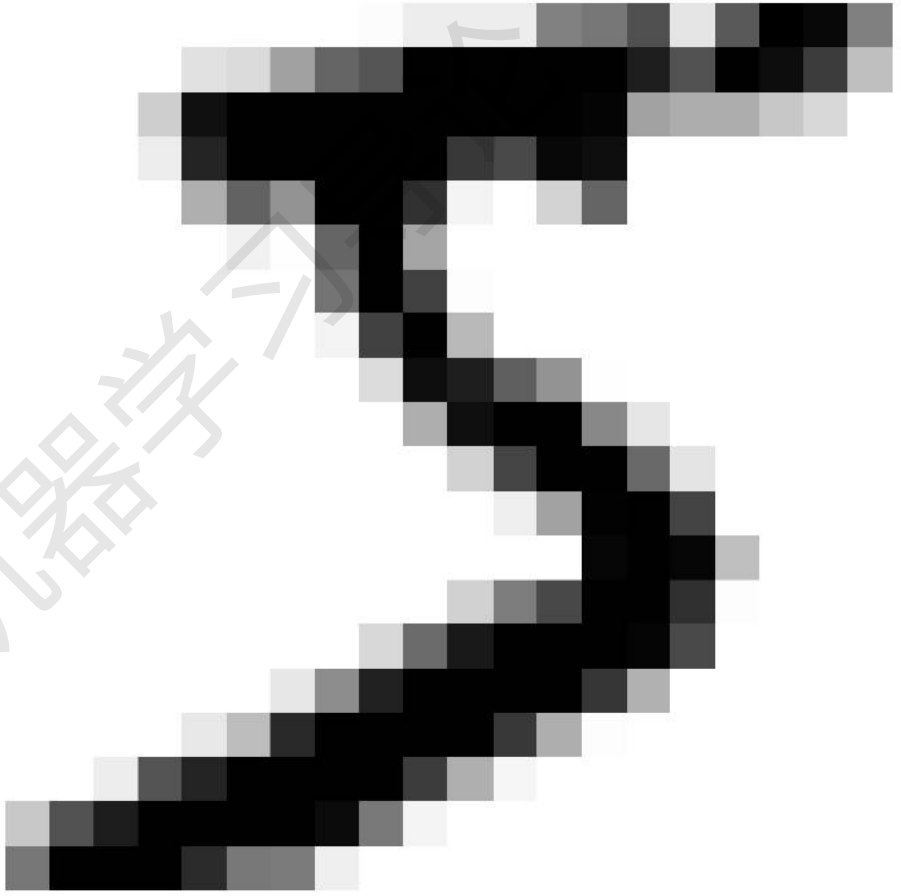
MNIST数据集

- Scikit-Learn提供了下载MNIST数据集的功能。
  - 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。

```
>>> 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'])
```

MNIST数据集

- 共有7万张图片，每张图片有784个特征。
- 因为图片是28×28像素，每个特征代表了一个像素点的强度，从0（白色）到255（黑色）。
- MNIST数据集已经分成训练集（前6万张图片）和测试集（最后1万张图片）



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

把标签和像素数据放在两个变量中

```
some_digit = X[0]
```

找一个例子

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

标签是字符，大部分机器学习算法希望是数字，让我们把y转换成整数

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

MNIST数据集已经分成训练集（前6万张图片）和测试集（最后1万张图片）

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

训练二元分类器

- 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示例，它只能区分两个类别：5和非5。
- 选择随机梯度下降（SGD）分类器（能够有效处理非常大型
- 的数据集），使用Scikit-Learn的SGDClassifier类。

```
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)
```

训练二元分类器

- 用训练好的分类器来检测数字5的图片：

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

- 分类器预测这个图像代表5（True）。预测正确
- 当更多的测试数据时，需要一些方法来评估模型的性能。

性能测量（Performance Measures）

接下来将要介绍：

- 使用交叉验证测量准确率
- 混淆矩阵
- 精度和召回率
- 精度/召回率权衡
- ROC曲线

使用交叉验证测量准确率

- 自己写代码实现交叉验证：

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3, random_state=42)   采用3-折交叉验证

for train_index, test_index in skfolds.split(X_train, y_train_5):  循环提取3次训练和测试index
    clone_clf = clone(sgd_clf)   复制SGD分类器
    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))  # prints 0.9502, 0.96565, and 0.96495
```

分配训练数据和测试数据，循环3次

对训练数据进行建模，对测试数据进行预测和计算误差

使用交叉验证测量准确率

- 直接用cross_val_score（）函数来评估SGDClassifier模型，采用K-折交叉验证法（3个折叠）：

```
>>> 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])
```

- 所有折叠交叉验证的准确率（正确预测的比率）超过93%!

使用交叉验证测量准确率

- 如果写一个简单的分类器，将每张图都分类成“非5”：

```
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)
```

- 正确率超过也90%：

```
>>> never_5_clf = Never5Classifier()
>>> cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.91125, 0.90855, 0.90915])
```

- 准确率通常无法成为分类器的首要性能指标，特别是处理有偏数据集时

混淆矩阵

- 评估分类器性能的更好方法是混淆矩阵，其总体思路就是统计A类别实例被分成为B类别的次数。

```
from sklearn.model_selection import cross_val_predict

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

cross_val_predict（）函数同样执行K-折交叉验证，但返回的不是评估分数，而是每个折叠的预测。

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

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

混淆矩阵

- 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：53 057张被正确地分为“非5”类别（真负类），1522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：1325张被错误地分为“非5”类别（假负类），4096张被正确地分在了“5”这一类别（真正类）

实际/预测	预测不是5	预测是5
实际不是5	53057 真负类TN	1522 假正类FP
实际是5	1325 假负类FN	4096 真正类TP

精度与召回率

- 正类预测的准确率也称为分类器的精度（precision）：

$$精度 = \frac{TP}{TP + FP}$$

TP是真正类的数量，FP是假正类的数量。

- 精度通常与召回率一起使用，它是分类器正确检测到的正类实例的比率：

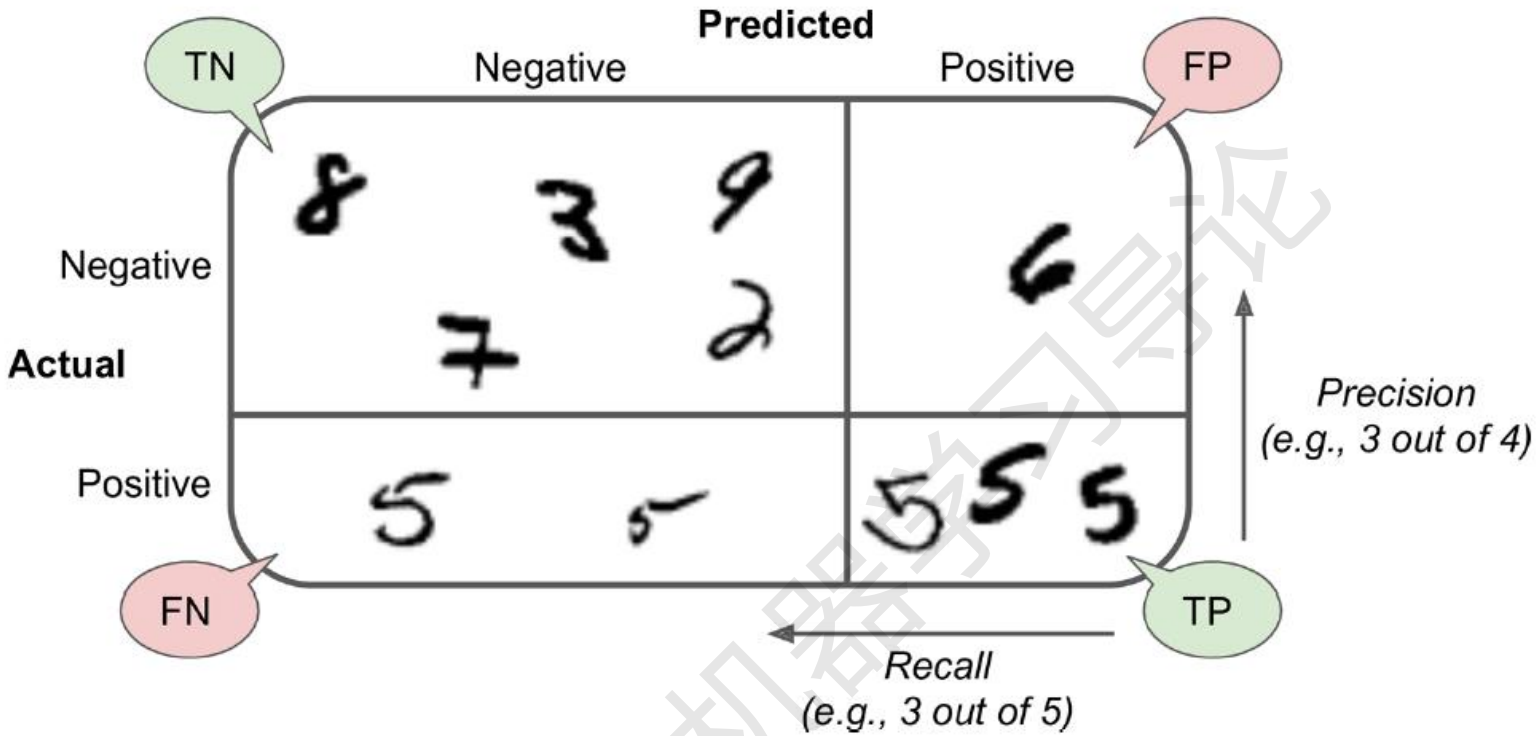
$$召回率 = \frac{TP}{TP + FN}$$

FN是假负类的数量。

实际/预测	预测不是5	预测是5
实际不是5	53057 真负类	1522 假正类
实际是5	1325 假负类	4096 真正类

精度

召回率



TN: True Negative 真负类
FP: False Positive 假正类
FN: False Negative 假负类
TP: True Positive 真正类

精度与召回率

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

```
>>> 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
```

- 从精度和召回率看，这个5-检测器看起来并不像它的准确率那么高了。
- 当它说一张图片是5时，只有72.9%的概率是准确的，并且也只有75.6%的数字5被它检测出来了。

F_1分数

- 我们可以将精度和召回率组合成一个单一的指标，称为F1分数（F1 score）。

$$F_1 = \frac{2}{\frac{1}{精度} + \frac{1}{召回率}} = 2 \times \frac{精度 \times 召回率}{精度 + 召回率} = \frac{TP}{TP + \frac{FN + FP}{2}}$$

- F1分数是精度和召回率的谐波（harmonic mean）平均值。
- 正常的平均值平等对待所有的值，而谐波平均值会给予低值更高的权重。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F1分数。

F_1分数

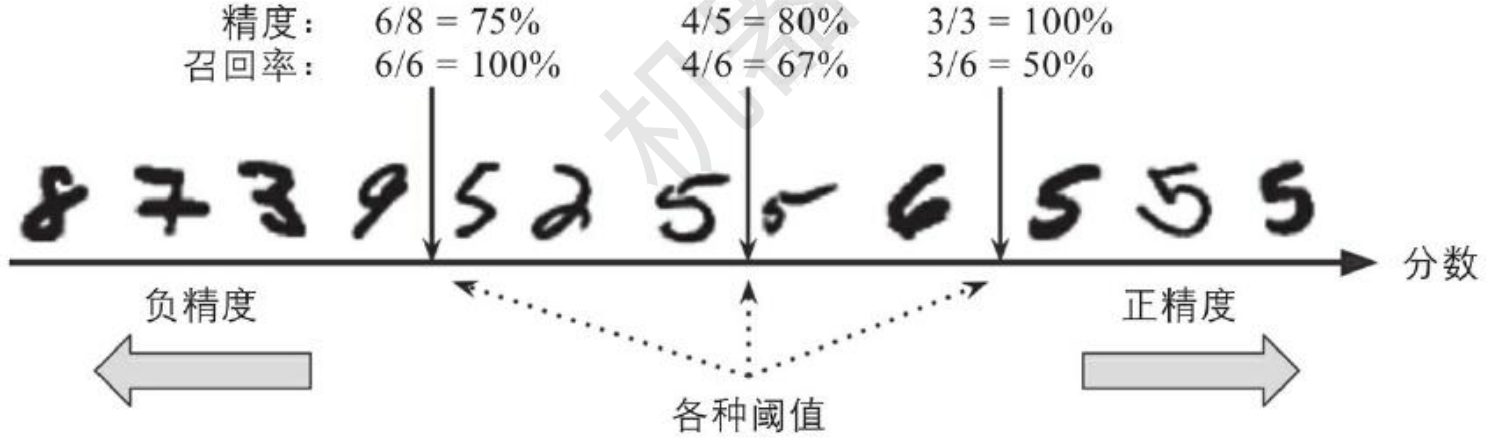
- 要计算F1分数，只需要调用f1_score（）即可：

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

- 在某些情况下，你更关心的是精度，而另一些情况下，你可能真正关心的是召回率。

精度/召回率权衡

- 对于每个实例，它会基于决策函数计算出一个分值，如果该值大于阈值（threshold），则将该实例判为正类，否则便将其判为负类。
- 图中显示了从左边最低分到右边最高分的几个数字。
- 提高阈值会增加精度、降低召回率；降低阈值则会增加召回率、降低精度。



阈值的选择

- 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：

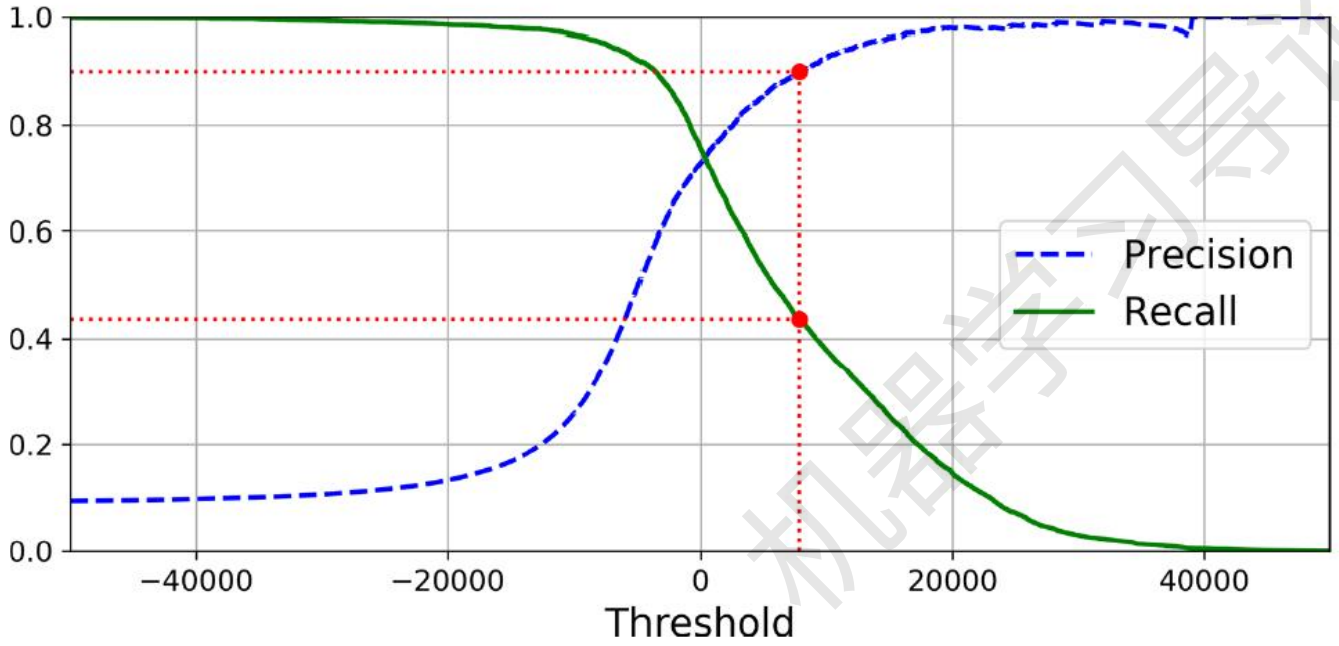
```
from sklearn.metrics import precision_recall_curve

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

阈值的选择

- 使用Matplotlib绘制精度和召回率相对于阈值的函数图

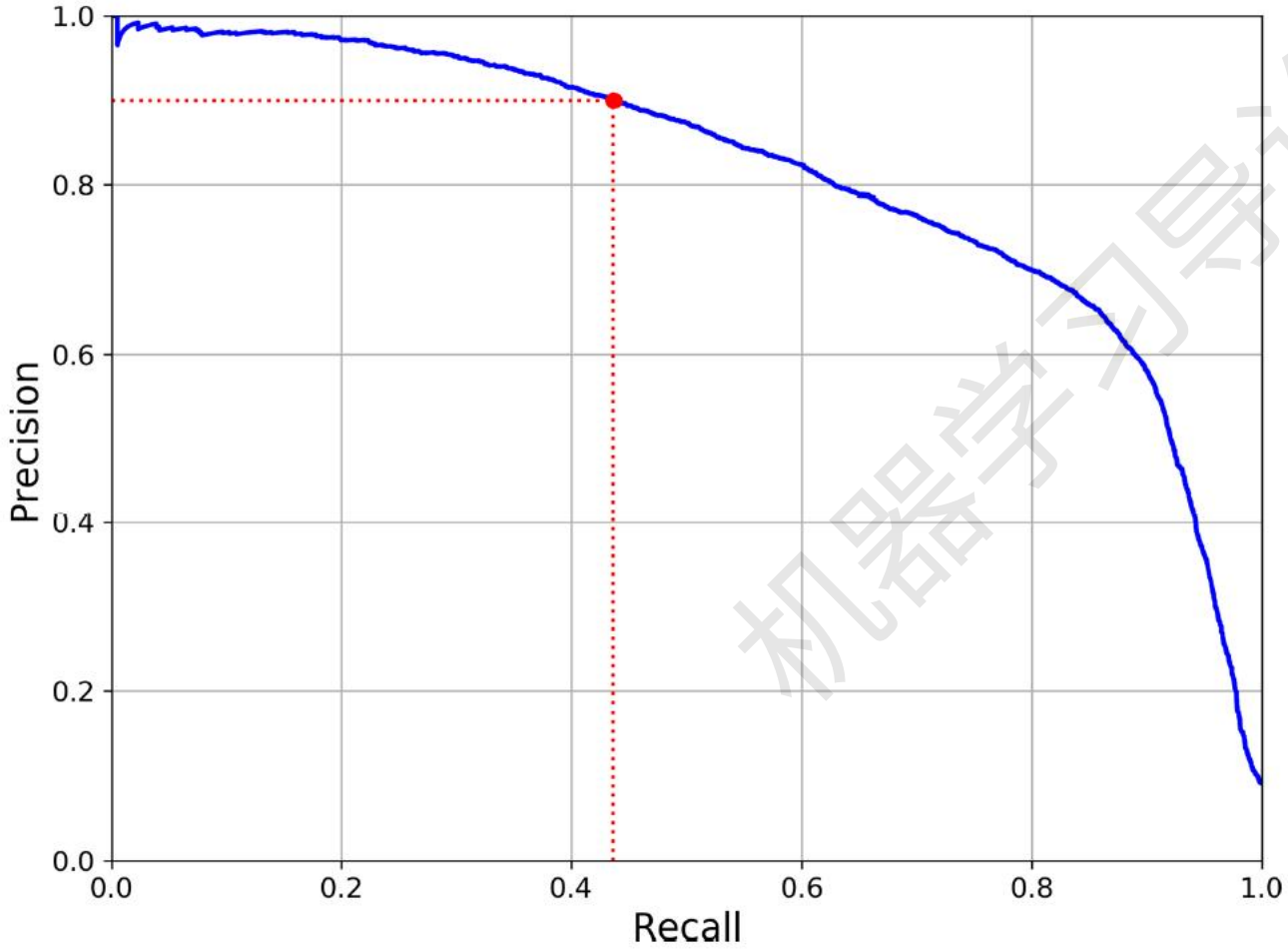
```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    [...] # highlight the threshold and add the legend, axis label, and grid

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

阈值的选择



为什么在图中精度曲线比召回率曲线要崎岖一些？原因在于，当你提高阈值时，精度有时也有可能会下降（尽管总体趋势是上升的）

阈值的选择



另一种找到好的精度/召回率权衡的方法是直接绘制精度和召回率的函数图

从图中可以看到，从80%的召回率往右，精度开始急剧下降。你可能会尽量在这个陡降之前选择一个精度/召回率权衡。

阈值的选择

假设你决定将精度设为90%，计算相应的阈值：

```
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)]
```

进行预测：

```
y_train_pred_90 = (y_scores >= threshold_90_precision)
```

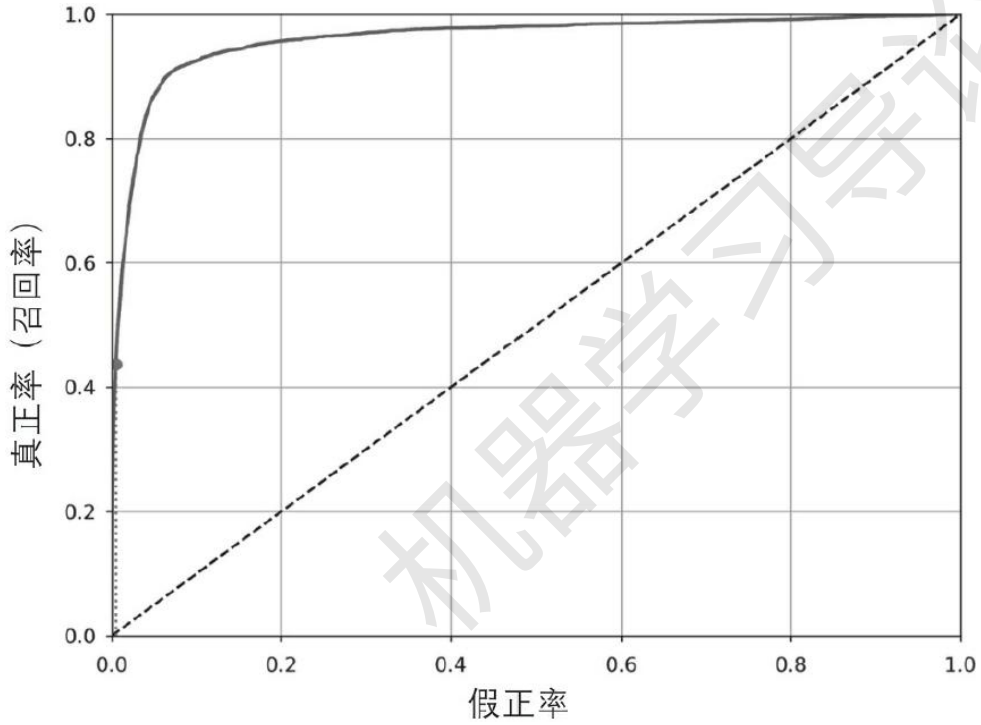
检查一下这些预测结果的精度和召回率：

```
>>> precision_score(y_train_5, y_train_pred_90)
0.9000380083618396
>>> recall_score(y_train_5, y_train_pred_90)
0.4368197749492714
```

ROC曲线

- 受试者工作特征曲线receiver operating characteristic（简称ROC）
- 绘制的是真正类率（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，也称为特异度。
- ROC曲线绘制的是灵敏度（召回率）和（1-特异度）的关系。

ROC曲线



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)
```

- 使用Matplotlib绘制FPR对TPR的曲线

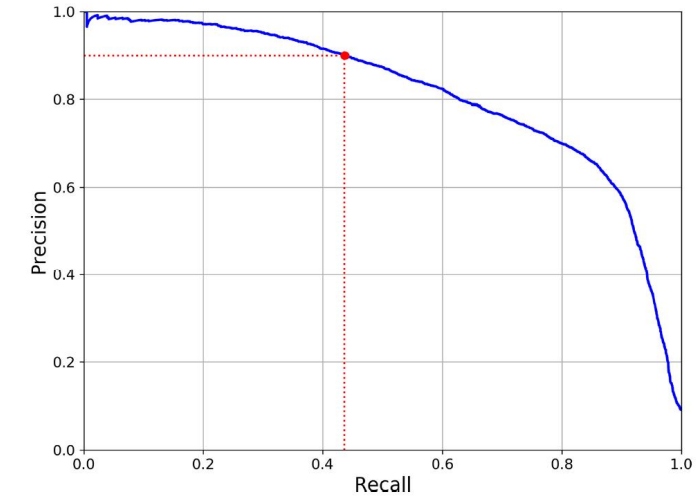
```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--') # Dashed diagonal
    [...] # Add axis labels and grid

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

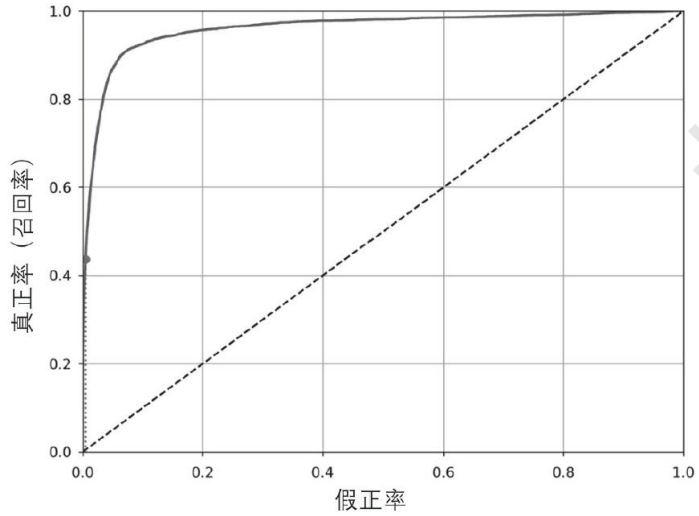
ROC曲线

- 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

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

ROC曲线





- 选择ROC曲线还是精度/召回率（PR）曲线：当正类非常少见或者你更关注假正类而不是假负类时，应该选择PR曲线，反之则是ROC曲线
- 例如，看前面的ROC曲线图（以及ROC AUC分数），你可能会觉得分类器很好。但这主要是因为跟负类（非5）相比，正类（数字5）的数量真的很少。相比之下，PR曲线清楚地说明分类器还有改进的空间

训练随机森林分类器

- 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
- 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dict_proba（）方法
- dict_proba（）方法会返回一个数组，其中每行代表一个实例，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）

训练随机森林分类器

```
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")
```

训练随机森林分类器，得到每个实例属于每个类别的概率

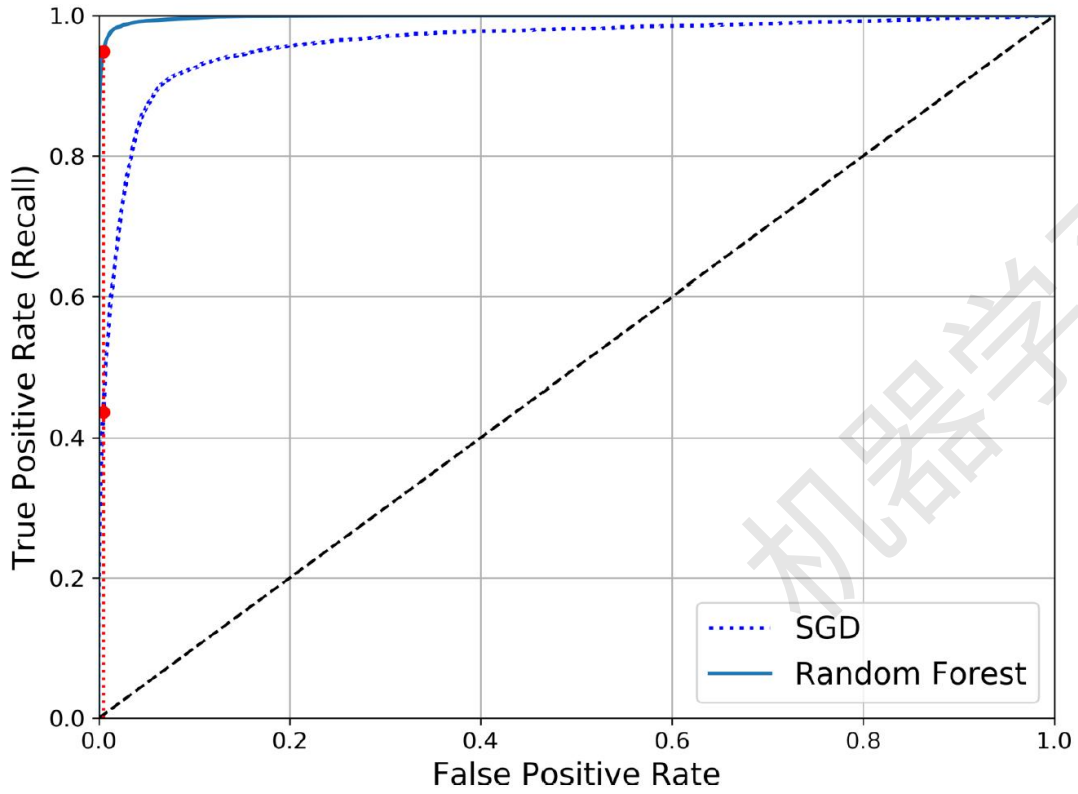
```
y_scores_forest = y_probas_forest[:, 1]   # score = proba of positive class
fpr_forest, tpr_forest, thresholds_forest =
roc_curve(y_train_5,y_scores_forest)
```

直接使用正类的概率作为分数值

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

绘制两种分类器的ROC曲线比较图

训练随机森林分类器



随机森林分类器优于SGD分类器，因为它的ROC曲线更靠近左上角

训练随机森林分类器

- 计算随机森林分类器的ROC AUC分数：

```
>>> roc_auc_score(y_train_5, y_scores_forest)
0.9983436731328145
```

- 随机森林的AUC分数比SGDClassifier更大
- 再测一测精度和召回率的分数：99.0%的精度和86.6%的召回率。比SGDClassifier更更优

多类分类器

- 二元分类器在两个类中区分，而多类分类器（也称为多项分类器）可以区分两个以上的类
- 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。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）
- 有多种策略可以让二元分类器实现多类分类的目的

一对多（OVR）策略

- 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，以此类推）
- 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。
- 这称为一对剩余（OvR）策略，也称为一对多（one-versus-all）

一对一（OVO）策略

- 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，最后看哪个类获胜最多

策略选择

- 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进行训练。
- 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。对于这类算法，OvO是一个优先的选择，因为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。
- 对大多数二元分类器来说，OvR策略还是更好的选择。

多类分类器

- Scikit-Learn可以检测到你尝试使用二元分类算法进行多类分类任务，它会根据情况自动运行OvR或者OvO
- 我们用sklearn.svm.SVC类来试试SVM分类器（见第5章）

```
>>> from sklearn.svm import SVC
>>> svm_clf = SVC()
>>> svm_clf.fit(X_train, y_train) # y_train, not y_train_5
>>> svm_clf.predict([some_digit])
array([5], dtype=uint8)
```

- 在内部，Scikit-Learn实际上训练了45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。

多类分类器

- 调用decision_function（）方法。它会返回10个分数，每个类1个，而不再是每个实例返回1个分数：

```
>>> some_digit_scores = svm_clf.decision_function([some_digit])
>>> some_digit_scores
array([[ 2.92492871,  7.02307409,  3.93648529,  0.90117363,  5.96945908,
         9.5       ,  1.90718593,  8.02755089, -0.13202708,  4.94216947]])
```

- 最高分确实是对应数字5这个类别：

```
>>> np.argmax(some_digit_scores)
5
>>> svm_clf.classes_
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)
>>> svm_clf.classes_[5]
5
```

多类分类器

- SGD分类器直接就可以将实例分为多个类：

```
>>> sgd_clf.fit(X_train, y_train)
>>> sgd_clf.predict([some_digit])
array([5], dtype=uint8)
```

- 调用decision_function（）可以获得分类器将每个实例分类为每个类的分数列表：

```
>>> sgd_clf.decision_function([some_digit])
array([[-15955.22628, -38080.96296, -13326.66695,  573.52692, -17680.68466,
         2412.53175, -25526.86498, -12290.15705, -7946.05205, -10631.35889]])
```

- 使用交叉验证，使用cross_val_score（）函数来评估SGDClassifier的准确性：

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([0.8489802 , 0.87129356, 0.86988048])
```

误差分析

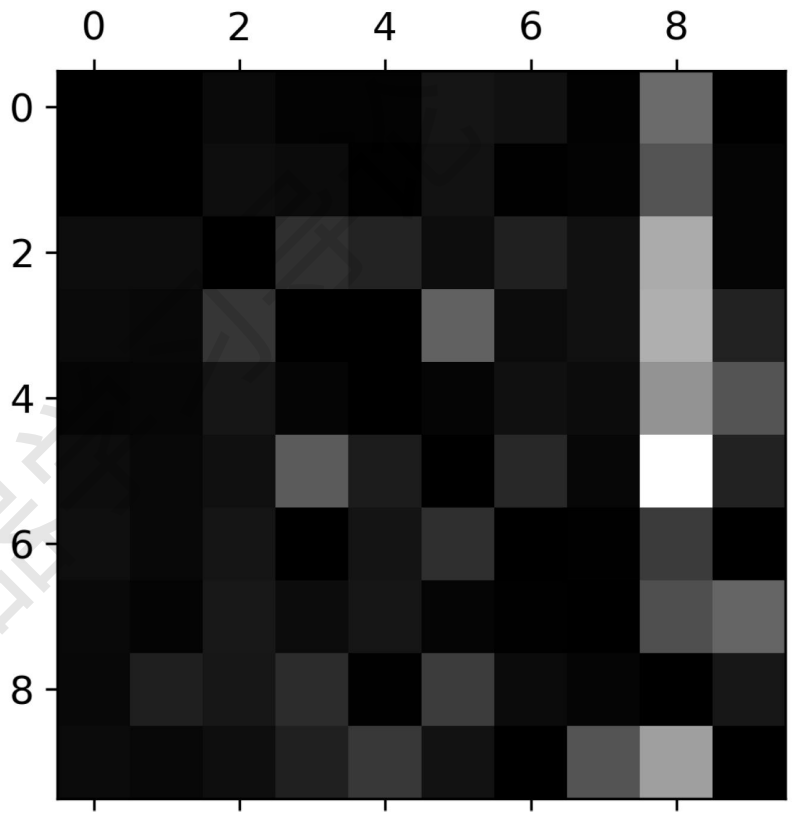
- 使用cross_val_predict（）函数进行预测，然后调用confusion_matrix（）函数查看混淆矩阵：

```
>>> 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([[5578,    0,   22,    7,    8,   45,   35,    5,  222,    1],
       [   0, 6410,   35,   26,    4,   44,    4,    8,  198,   13],
       [  28,   27, 5232,  100,   74,   27,   68,   37,  354,   11],
       [  23,   18,  115, 5254,    2,  209,   26,   38,  373,   73],
       [  11,   14,   45,   12, 5219,   11,   33,   26,  299,  172],
       [  26,   16,   31,  173,   54, 4484,   76,   14,  482,   65],
       [  31,   17,   45,    2,   42,   98, 5556,    3,  123,    1],
       [  20,   10,   53,   27,   50,   13,    3, 5696,  173,  220],
       [  17,   64,   47,   91,    3,  125,   24,   11, 5421,   48],
       [  24,   18,   29,   67,  116,   39,    1,  174,  329, 5152]])
```

误差分析

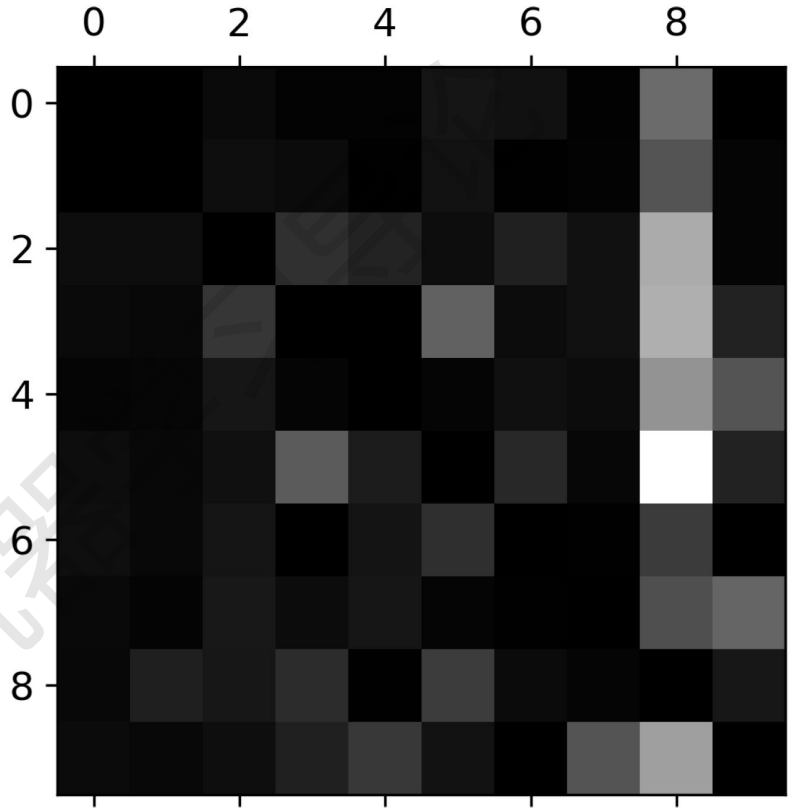
- 将混淆矩阵中的每个值除以相应类中的图片数量，这样比较的就是错误率而不是错误的绝对值
- 使用Matplotlib的matshow（）函数来查看转换后的混淆矩阵的图像（用0填充对角线，只保留错误）：

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

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

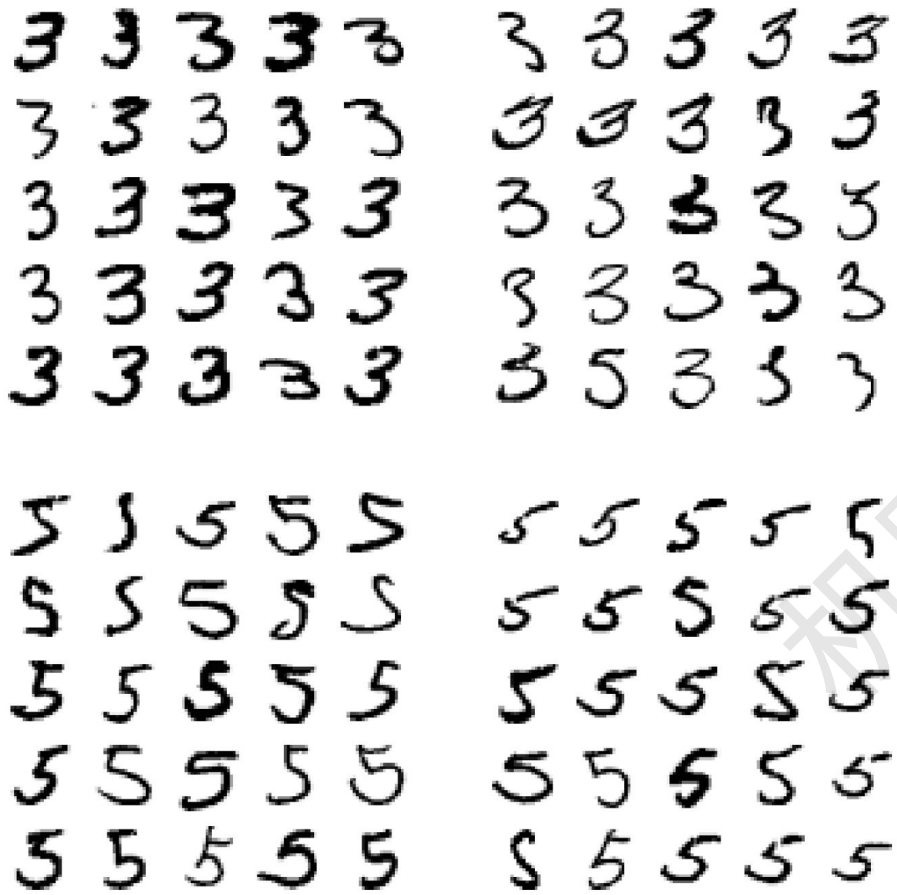


误差分析

- 每行代表实际类，而每列表示预测类
- 第8列看起来非常亮，说明有许多图片被错误地分类为数字8了
- 第8行不那么差，告诉你实际上数字8被正确分类为数字8
- 错误不是完全对称的，比如，数字3和数字5经常被混淆（在两个方向上）。



误差分析



- 左侧的两个5×5矩阵显示了被分类为数字3的图片，右侧的两个5×5矩阵显示了被分类
- 分类器弄错的数字为左下方和右上方的矩阵