第三章

# 分類

第 1 章提到，回歸（預測值）與分類（預測類別）是最常見的監督學習任務。我們在第 2 章已經探討回歸任務、預測房地產價格、使用各種演算法，例如線性回歸、決策樹和隨機森林（後續章節會進一步解釋）了，接下來要把注意力放在分類系統上面。

## MNIST

本章將使用 MNIST 資料組，它有 70,000 張美國高中生和人口普查局員工手寫的數字圖片。每一張圖片都有一個標籤，指出它代表的數字。因為有太多研究使用這個資料組了，所以它經常被稱為機器學習的「hello world」：每當有新的分類演算法被發明出來時，大家都會用 MNIST 來看看它的效果如何，而且每一個學習機器學習的人遲早都會處理這個資料組。

Scikit-Learn 有許多協助下載熱門資料組的函式，MNIST 正是它支援的其中一種資料組。下面的程式可取得 MNIST 資料組[1]：

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

Scikit-Learn 可以載入的資料組通常有相似的目錄結構，包括：

1 在預設情況下，Scikit-Learn 會將下載的資料組放在 *$HOME/scikit_learn_data* 目錄內。

- 描述資料組的 DESCR 鍵
- 含有一個陣列的 data 鍵，陣列內每個實例有一列，每個特徵有一欄
- 一個 target 鍵，裡面有個包含標籤的陣列

我們來看一下這些陣列：

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

這裡面有 70,000 張圖像，每張圖像有 784 個特徵，因為每張圖片有 28 × 28 個像素，每一個特徵代表一個像素的顏色深淺，從 0（白色）到 255（黑色）。我們來顯示資料組裡面的一個數字，只要抓取一個實例的特徵向量，將它的形狀改成 28 × 28 陣列，並且用 Matplotlib 的 imshow() 函式顯示它即可：

```
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()
```

它看起來像 5，從標籤來看，確實也是如此：

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

這個標籤是個字串，多數的 ML 演算法都期望看到數字，所以我們要將 y 轉型成整數：

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

圖 3-1 是 MNIST 的一些其他圖片，你可以從中知道分類任務的複雜程度。

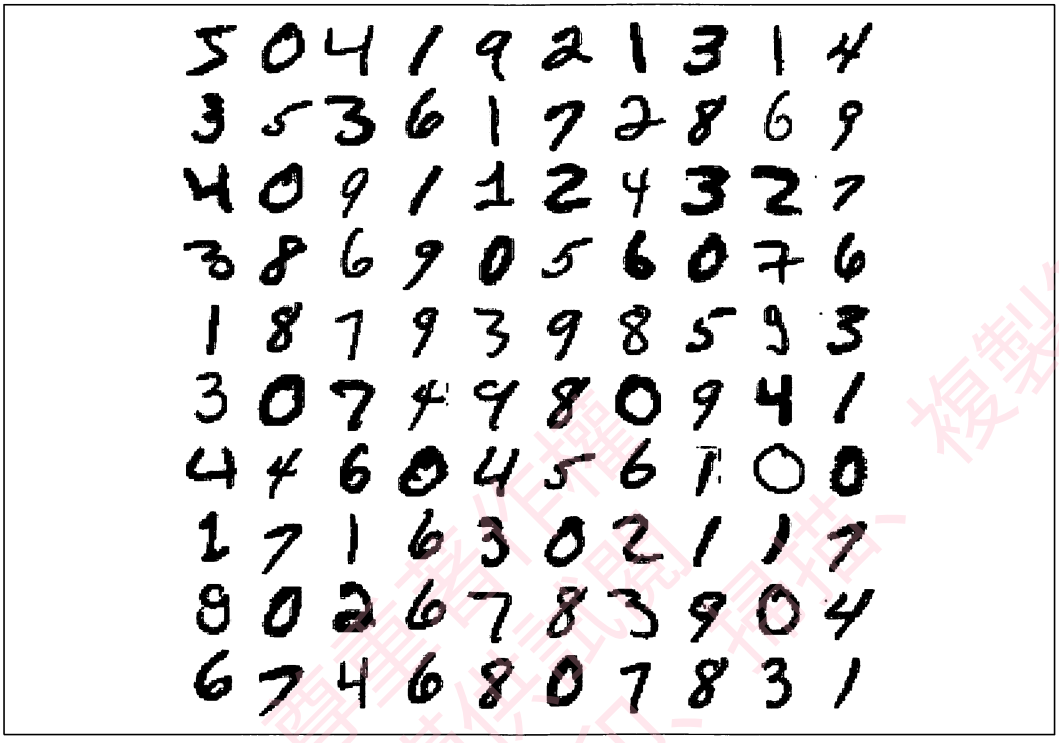


圖 3-1　MNIST 的數字

等一下！你一定要建立一個測試組，先將它放在一旁，才能仔細查看資料。其實 MNIST 資料組已經被拆成訓練組（前 60,000 張圖片）與測試組（後 10,000 張圖片）了：

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

它已經幫我們洗亂訓練組了，這是件好事，因為這可保證所有交叉驗證 fold 都是相似的（你絕對不希望有一個 fold 缺少某些數字）。此外，有些學習演算法對訓練實例的順序很敏感，如果有連續好幾個實例是類似的，它們的表現會很糟，洗亂資料組可確保這種情況不會發生[2]。

2　有時洗亂是不好的做法，例如，當你處理的是時間序列資料（例如股價或天氣狀態）時。下一章將探討這個主題。

## 訓練二元分類器

我們先簡化問題，只嘗試辨識一個數字，例如，數字 5。這個「數字 5 偵測器」是個**二元分類器**，它只能分辨兩個類別，5 和非 5。我們為這項分類任務建立目標向量：

```
y_train_5 = (y_train == 5)  # 所有的 5 都是 True，所有其他數字都是 False
y_test_5 = (y_test == 5)
```

我們接著選擇一種分類器並訓練它，**隨機梯度下降**（*Stochastic Gradient Descent*，SGD）分類器是很好的起點，即 Scikit-Learn 的 `SGDClassifier` 類別。這個分類器的優點是，它可以高效地處理非常大型的資料組，部分的原因是 SGD 以獨立的方式處理訓練實例，每次處理一個（所以 SGD 非常適合用在線上學習）。我們來建立一個 `SGDClassifier`，並且用整個訓練組訓練它：

```
from sklearn.linear_model import SGDClassifier

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

`SGDClassifier` 的訓練過程需要依靠隨機性（因此名字中有「隨機（stochastic）」）。如果你想要重現結果，就要設定 `random_state` 參數。

我們用它來偵測數字 5 的圖片：

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

分類器猜測這張圖片代表 5（`True`）。看起來它在這個案例猜對了！接著，我們來評估這個模型的績效。

## 評量績效

評估分類器通常比評估回歸器還要麻煩，所以本章會用大部分的篇幅探討這個主題。績效指標有很多種，所以先沖杯咖啡，準備學習許多新概念和縮寫吧！

### 用交叉驗證評估準確度

如第 2 章所述，使用交叉驗證來評估模型是很好的做法。

### 實作交叉驗證

有時你需要比 Scikit-Learn 提供的現成功能還要緊密地控制交叉驗證程序，此時，你可以自己實作交叉驗證。下面的程式所做的事情大致上與 Scikit-Learn 的 `cross_val_score()` 函式相同，也會印出相同的結果：

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

skfolds = StratifiedKFold(n_splits=3, random_state=42)

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))  # 印出 0.9502, 0.96565 與 0.96495
```

`StratifiedKFold` 類別執行分層抽樣（見第 2 章）來產生 fold，這些 fold 裡面有具代表性的類別比率。這段程式在每次迭代時都會建一個分類器複製品，用訓練 fold 來訓練那個複製品，再用測試 fold 進行預測，接著計算正確的預測數量，並輸出正確預測比率。

我們用 `cross_val_score()` 函式來評估 `SGDClassifier` 模型，使用 K-fold 交叉驗證、3 fold。之前說過，K-fold 交叉驗證的意思是將訓練組拆成 K 個 fold（在這個例子是 3 個），接著使用以其他 fold 訓練的模型，來對它們進行預測並評估（見第 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([0.96355, 0.93795, 0.95615])
```

哇！它處理每一個 fold 都有超過 93% 的準確度（正確預測的比率）？看起來很了不起，不是嗎？不過，在你樂昏頭之前，我們來看一個很蠢的分類器，它會將每一張圖片都分類為「不是 5」：

```
from sklearn.base import BaseEstimator

class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        return self
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
```

你猜得到這個模型的準確度嗎？我們來看答案：

```
>>> 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])
```

是的，它的準確度超過 90% ！這單純是因為只有大約 10% 的圖片是 5，所以如果你從頭到尾都猜圖片不是 5，90% 的猜測都是對的，比 Nostradamus（著名的西方預言家）更厲害！

這說明為什麼準確度通常不是評估分類器的首要指標，尤其是當你處理的資料組有偏差（*skew*）的時候（即，有些類別比其他類別更常出現）。

## 混淆矩陣

評估分類器的較佳工具是混淆矩陣（*confusion matrix*，或譯為誤差矩陣）。它的概念大致是計算類別 A 的實例被分類為類別 B 的次數。例如，要知道分類器錯誤地將 5 看成 3 的次數，你就要查看混淆矩陣的第 5 列第 3 行。

若要算出混淆矩陣，你要先算出一組預測值，才能拿它們與實際的目標做比較。你可以對測試組進行預測，但我們先不要碰它（之前提過，在專案快結束，已經有個可以發表的分類器時，你才可以使用測試組）。你可以改用 `cross_val_predict()` 函式：

```
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()` 與 `cross_val_score()` 函式一樣執行 K-fold 交叉驗證，但它並非回傳評估分數，而是回傳對著各個測試 fold 進行的預測。也就是說，你可以得到模型對每一個訓練組實例進行的乾淨預測（「乾淨」是指該結果是用沒有在訓練期看過那筆資料的模型預測出來的）。

你可以用 `confusion_matrix()` 函式取得混淆矩陣，只要將目標類別（`y_train_5`）與預測類別（`y_train_pred`）傳給它就可以了：

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

混淆矩陣的每一列都代表一個實際的類別，每一行都代表一個預測的類別。這個矩陣的第一列考慮非 5 圖片（陰性（*negative*）類別）：裡面有 53,057 張被正確地分類為非 5（稱為真陰（*true negative*）），其他的 1,522 張被錯誤地分類為 5（偽陰（*false positive*））。第二列考慮的是 5 圖片（陽性（*positive*）類別）：有 1,325 張被錯誤地分類為非 5（偽陰（*flase negative*）），其他的 4,096 張被正確地分類為 5（真陽（*true positive*））。完美的分類器只有 true positive 與 true negative，所以它的混淆矩陣的主對角線（左上到右下）才會有非零值：

```
>>> y_train_perfect_predictions = y_train_5 # 假裝我們得到完美的結果
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)
array([[54579,     0],
       [    0,  5421]])
```

雖然混淆矩陣提供許多資訊，但你可能比較喜歡更簡潔的評量標準。有一種有趣的指標是陽性預測的準確性，稱為分類器的 *precision*（精度）（公式 3-1）。

公式 *3-1*　*precision*

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

*TP* 是 true positive 的數量，*FP* 是 false positive 的數量。

取得完美 precision 最簡單的方式，就是只做一個陽性預測，並確定它是對的（precision = 1/1 = 100%）。但是這種做法不切實際，因為它讓分類器忽略一個 positive 實例之外的所有實例。所以 precision 通常會與另一個評量標準，稱為 *recall*，或稱為 *sensitivity* 或 *true positive* 率（TPR）一起使用，這個標準是分類器正確認出 positive 實例的比率（公式 3-2）。

公式 *3-2*　*Recall*

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

*FN* 是 false negative 的數量。

如果你無法理解混淆矩陣，圖 3-2 或許可以協助你。

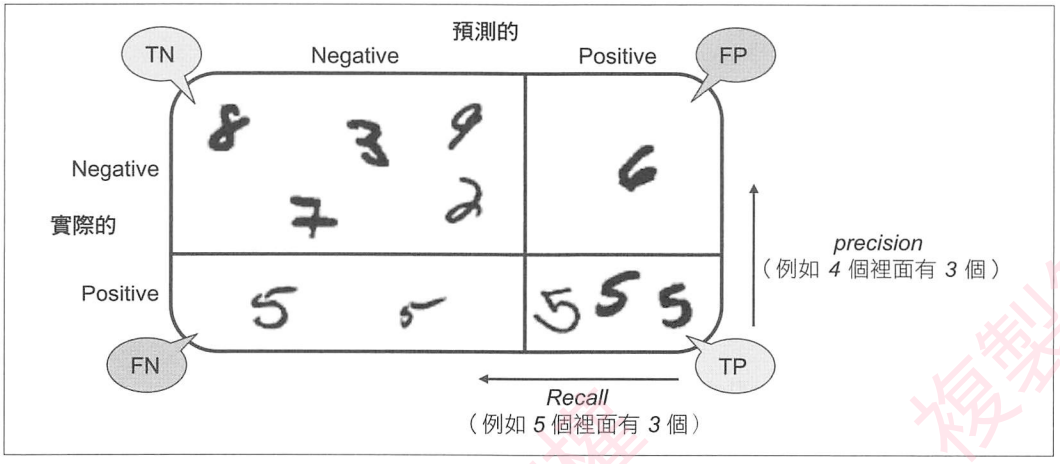



圖 3-2　這個混淆矩陣展示 true negative（左上）、false positive（右上）、false negative（左下）、true positive（右下）

## precision 與 recall

Scikit-Learn 有一些計算分類器評量標準的函式，包括 precision 與 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
```

從「數字 5 偵測器」的準確度來看，它已經不像之前那麼光彩奪目了。當它聲稱圖片是 5 時，只有 72.9% 是對的，而且，它只能認出 75.6% 的 5。

我們可以將 precision 與 recall 結合成單一評量標準，稱為 $F_1$ **分數**，這個評量標準很方便，尤其是當你需要用簡單的方式來比較兩種分類器時。$F_1$ 分數是 precision 與 recall 的**調和平均數**（公式 3-3）。我們計算平均數時，通常會平等看待每一個值，但調和平均數賦予小值更高權重，因此，唯有 recall 與 precision 的分數都很高時，分類器的 $F_1$ 分數才會高。

公式 3-3　$F_1$

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

呼叫 `f1_score()` 函式即可計算 $F_1$ 分數：

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

當分類器具有相似的 precision 與 recall 時，$F_1$ 分數較高。這個結果不一定符合你的需求：有時你最在乎 precision，有時你在乎的是 recall。例如，如果你訓練了一個分類器來偵測適合兒童觀賞的影片，或許比較喜歡一個會拒絕很多好影片（低 recall），但只會留下安全的影片（高 precision）的分類器，而不是有高很多的 recall，但是會讓產品顯示一些很不好的影片的分類器（此時，你甚至想要人為檢查分類器的選擇）。另一方面，假設你訓練了一個分類器，用它在監視影片中找出小偷，如果它只有 30% 的 precision，但只要它有 99% 的 recall，它就是個好的分類器（當然，保全會看到一些假警報，但是幾乎可以抓到所有小偷）。

可惜的是，你無法兩者兼得，提升 precision 會降低 recall，反之亦然。這稱為 *precision/recall* **取捨**。

## precision/recall 取捨

為了瞭解這項取捨，我們來看一下 `SGDClassifier` 如何做出它的分類決策。它會用**決策函數**來為每一個實例計算一個分數，如果分數大於某個**閾值**，它就會將實例指派給 positive 類別，否則指派給 negative 類別。圖 3-3 有一些數字，它們被放在左邊的最低分到右邊的最高分之間。假如**決策閾值**位於中央箭頭處（介於兩個 5 之間）：你會在那個閾值右邊看到 4 個 true positive（真的是 5），與 1 個 false positive（其實是 6）。因此，使用那個閾值時，precision 是 80%（5 個有 4 個）。但是在 6 個真的 5 裡面，分類器只偵測出 4 個，所以 recall 是 67%（6 個有 4 個）。當你將閾值提升時（將它移到右邊的箭頭），false positive（那個 6）就變成 true negative，因而提升 precision（到達 100%），但是有一個 true positive 變成 false negative，所以 recall 減為 50%。反過來，將閾值降低會增加 recall 並減少 precision。

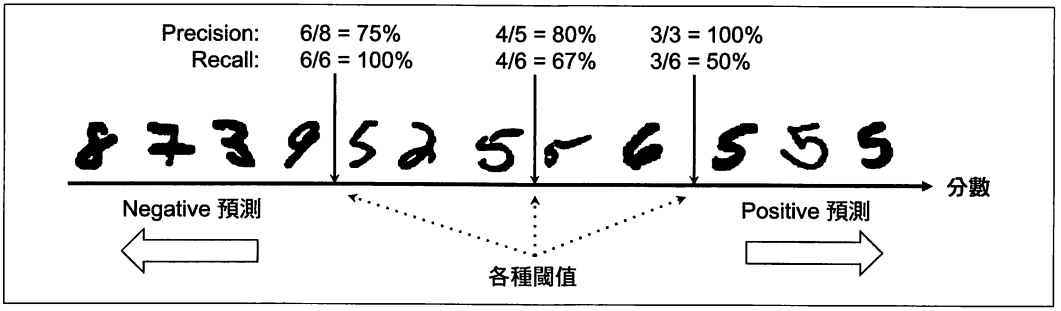



**圖 3-3　在這個 precision/recall 取捨中，圖像是用它們的分類器分數來排列的，高於決策閾值的被視為 positive。閾值越高，則 recall 越低，但（一般來說）precision 越高**

Scikit-Learn 不讓你直接設定閾值，但可讓你讀取它用來進行預測的研判分數。你可以呼叫分類器的 `decision_function()` 方法（而不是 `predict()` 方法），來取得各個實例的分數，再使用任何閾值，根據這些分數進行決策：

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([2412.53175101])
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
array([ True])
```

`SGDClassifier` 使用的閾值是 0，所以上面的程式回傳的結果與 `predict()` 方法一樣（也就是 `True`）。我們提升閾值：

```
>>> threshold = 8000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False])
```

結果證實提升 threshold 會降低 recall。圖像確實是 5，這個分類器在閾值是 0 時可以偵測到它，但是當閾值升到 8,000 時偵測不到它。

如何決定閾值？首先，使用 `cross_val_predict()` 函式來取得訓練組的所有實例的分數，但是這一次指定你想要得到 decision 分數，而不是 prediction：

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

取得這些分數後，使用 `precision_recall_curve()` 函式來計算所有可能的閾值的 precision 與 recall：

```
from sklearn.metrics import precision_recall_curve

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

最後，使用 Matplotlib 來將 precision 與 recall 畫成閾值函數（圖 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")
    [...] # 突顯閾值，並加上圖例、軸標、網格

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

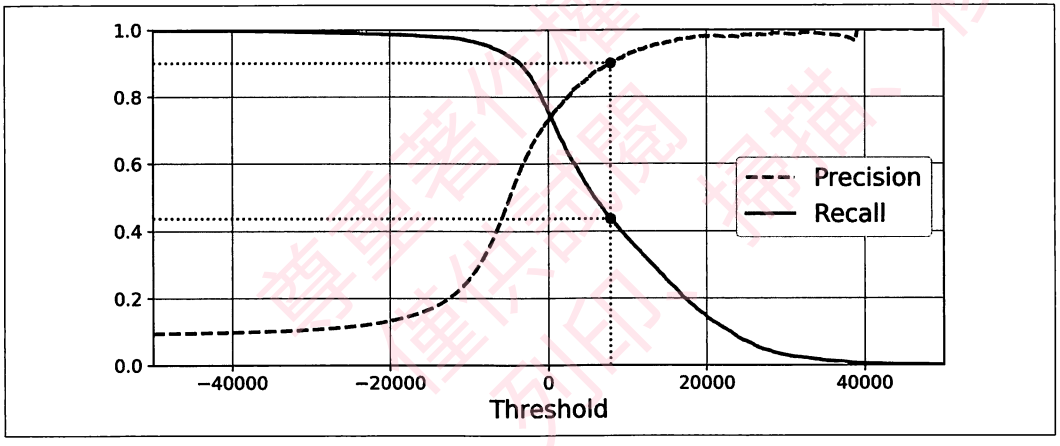



圖 3-4　precision 和 recall vs. decision 閾值

你可能想知道為什麼圖 3-4 的 precision 曲線不像 recall 曲線那麼平滑。原因是，有時提高 threshold 時，precision 可能會下降（雖然通常它會上升）。為了瞭解原因，回去看一下圖 3-3，注意當你從中央閾值開始往右移動一個數字時發生什麼事：precision 從 4/5（80%）下降為 3/4（75%）。另一方面，recall 只會在 threshold 增加時下降，這就是它的曲線看起來很平滑的原因。

要做出良好的 precision/recall 取捨，另一種方式是直接畫出 precision vs. recall 的關係圖，見圖 3-5（圖中標示與之前一樣的閾值）。

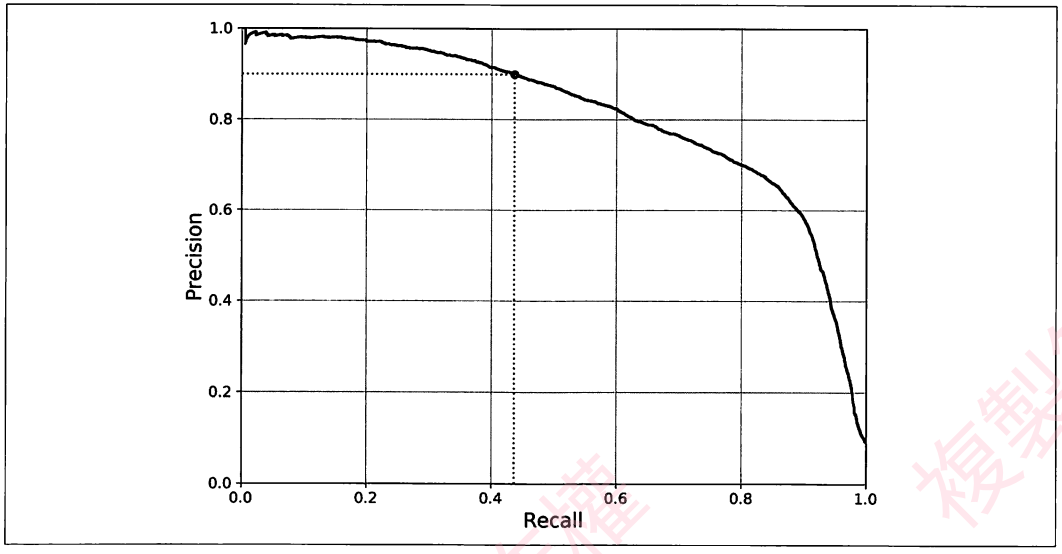



圖 3-5　precision v.s. recall

你可以看到，precision 在大約 80% recall 的地方開始急遽下降，你應該選擇下降之前的 precision/recall，例如在大約 60% recall 的地方。但是當然，你要根據專案來做選擇。

假設你的目標是 90% 的 precision，你看了第一張圖，發現你要使用 8,000 左右的閾值。為了取得更準確的值，你可以搜尋可以提供至少 90% 的 precision 的最低閾值（使用 `np.argmax()` 可取得最大值的第一個索引，在這個例子中，它代表第一個 `True` 值）：

```
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)] # ~7816
```

為了進行預測（目前是針對訓練組），你可以執行這段程式來取代呼叫分類器的 `predict()` 方法：

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

我們來看一下這些預測的 precision 與 recall：

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

太棒了，你有一個 90% precision 的分類器了！如你所見，建立一個可以提供你想要的幾乎任何 precision 的分類器很簡單，只要設定夠高的閾值就可以了，但事情還沒結束，如果高 precision 的分類器有太低的 recall，它就不太好用了！

如果有人說「我們要高達 99% 的 precision」，你應該問他「這時的 recall 是多少？」

## ROC 曲線

**接收者作業特徵**（*receiver operating characteristic*，ROC）曲線經常和二元分類器一起使用，它很像 precision/recall 曲線，但 ROC 曲線並非畫出 precision vs. recall，而是畫出 *true positive* 率（recall 的別名）vs. *false positive* 率（FPR）。FPR 是陰性實例被錯誤地歸類為陽性的比率。它等於 1 減 *true negative* 率（TNR），TNR 是陰性實例被正確地歸類為陰性的比率，也稱為 *specificity*。因此，ROC 曲線畫的是 *sensitivity*（recall）vs. 1 – *specificity*。

要畫出 ROC 曲線，你要先使用 `roc_curve()` 函式來計算各種閾值的 TPR 與 FPR：

```
from sklearn.metrics import roc_curve

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

接著你可以用 Matplotlib 畫出 FPR vs. TPR。這段程式可產生圖 3-6：

```
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
    [...] # 加上軸標與網格

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

這裡同樣需要進行取捨：recall（TPR）越高，分類器產生的 false positive（FPR）越多。虛線是純隨機的分類器的 ROC 曲線，離這條線越遠（靠左上角），分類器越好。

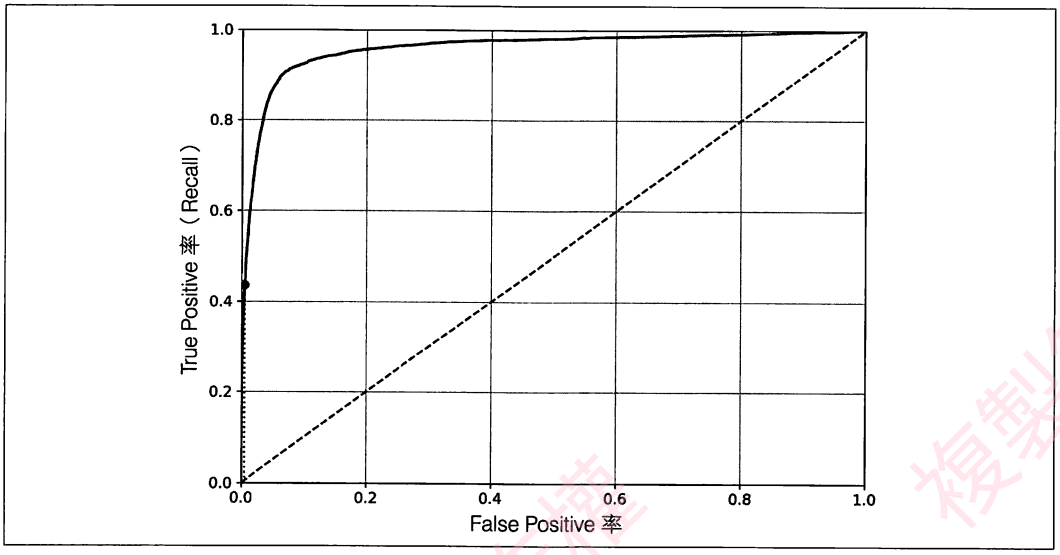



**圖 3-6　這張 ROC 曲線圖畫出所有可能的閾值的 false positive 率 vs. true positive 率；紅點是我們選擇的比率（在 43.68% recall 那裡）**

有一種比較分類器的做法是計算**曲線下方區域面積**（*area under the curve*，AUC）。完美分類器的 ROC AUC 等於 1，純隨機分類器的 ROC AUC 等於 0.5。Scikit-Learn 有個函式可計算 ROC AUC：

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

因為 ROC 曲線非常類似 precision/recall（PR）曲線，你可能想知道如何選擇該使用哪一個。根據經驗，當陽性類別很少，或當你比較在意 false positive 而非 false negative 時，你應該優先選擇 PR 曲線，否則就使用 ROC 曲線。例如，根據上面的 ROC 曲線（以及 ROC AUC 分數），或許你認為這一個分類器很好，但是它在很大程度上是因為陽性類別（5）比陰性類別（非 5）少很多。相較之下，PR 曲線可明顯地展示分類器還有改善的空間（曲線還可以更靠近左上角）。

接著我們來訓練一個 `RandomForestClassifier`，並且拿它的 ROC 曲線與 ROC AUC 分數與 `SGDClassifier` 的做比較。首先，你要取得訓練組的各個實例的分數。但是出於 `RandomForestClassifier` 的運作方式（見第 7 章），它沒有 `decision_function()` 方法，卻

有個 predict_proba() 方法。Scikit-Learn 分類器通常有這兩種方法中的一個，或同時擁有這兩個。predict_proba() 方法回傳一個陣列，在裡面，每一個實例都有一列，每一個類別都有一欄，內容是特定實例屬於特定類別的機率（例如圖片有 70% 的機率是 5）：

```
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")
```

roc_curve() 函式期望收到標籤與分數，但你可以給它類別機率，而不是分數。我們將陽性類別的機率當成分數傳入：

```
y_scores_forest = y_probas_forest[:, 1]   # 分數 = 陽性類別的機率
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,y_scores_forest)
```

接下來就可以畫出 ROC 曲線了，我們可以一起畫出第一條 ROC 曲線來比較它們（圖 3-7）：

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

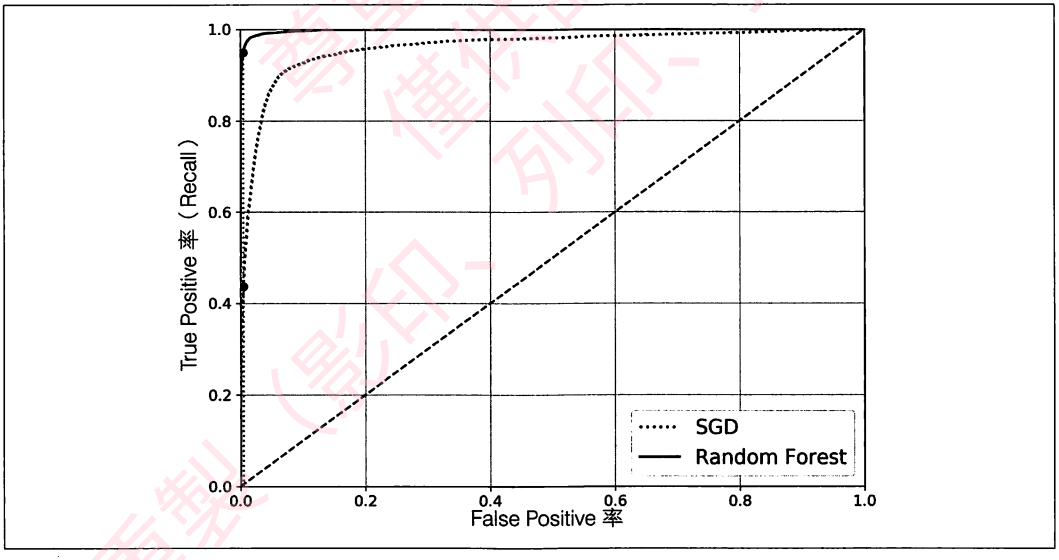



圖 3-7　比較 ROC 曲線：隨機森林分類器優於 SGD 分類器，因為它的 ROC 曲線靠近左上角許多，而且有更大的 AUC

從圖 3-7 可以知道，`RandomForestClassifier` 的 ROC 曲線看起來比 `SGDClassifier` 的好多了，它靠近左上角許多。因此，它的 ROC AUC 分數也明顯更好：

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

當你試著評測 precision 與 recall 分數時，可以得到 99.0% precision 與 86.6% recall。還不賴！

現在你已經知道如何訓練二元分類器、為你的任務選擇適當的評量標準、用交叉驗證評估分類器、選擇符合需求的 precision/recall，以及使用 ROC 曲線和 ROC AUC 分數來比較各種模型了。接著我們要試著偵測 5 之外的東西。

# 多類別分類

二元分類器可區分兩個類別，**多類別分類器**（也稱為**多項**（*multinomial*）**分類器**）則可以區分超過兩個類別。

有些演算法（例如 SGD 分類器、隨機森林分類器，以及樸素 Bayes 分類器）原本就可以處理多個類別。其他的演算法（例如 Logistic 回歸或支援向量機分類器）都是不折不扣的二元分類器。但是你可以採取各種策略，使用多個二元分類器來執行多類別分類。

建立系統來將數字圖片分成 10 個類別（從 0 到 9）的做法之一是訓練 10 個二元分類器，每一個負責一個數字（一個 0 偵測器，一個 1 偵測器，一個 2 偵測器，以此類推）。接著在分類圖片時，你可以讓各個分類器輸出該圖片的研判分數，看看哪個分類器輸出最高的分數，並選擇該類別。這種做法稱為**一對其餘**（*one-versus-the-rest*，*OvR*）策略（也稱為**一對全部**（*one-versus-all*））。

另一種做法是幫每一對數字訓練一個二元分類器：一個負責區分 0 與 1，另一個區分 0 與 2，另一個 1 與 2，以此類推。這種做法稱為**一對一**（*OvO*）策略。如果類別有 $N$ 個，你就要訓練 $N \times (N - 1) / 2$ 個分類器。對 MNIST 問題而言，這代表你要訓練 45 個二元分類器！當你想要分類一張圖片時，你就要讓全部的 45 個分類器處理那張圖片，看看哪個類別在彼此競爭中獲勝最多次。OvO 的主要優點是訓練各個分類器時，你只要用它區分的兩個類別來訓練即可。

有些演算法（例如支援向量機）無法隨著訓練組規模的增加而很好地擴展。對這些演算法而言，OvO 是首選，因為用小型的訓練組來訓練許多分類器的速度，比用大型的訓練組來訓練少量的分類器快得多。但是對多數的二元分類演算法而言，OvR 是首選。

Scikit-Learn 可偵測你試著使用二元分類演算法來處理多類別分類任務，並根據演算法自動執行 OvR 或 OvO。我們用支援向量機分類器（見第 5 章），即 `klearn.svm.SVC` 類別來嘗試這件事：

```
>>> 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)
```

做法很簡單！這段程式用訓練組和原始的目標類別 0 至 9（`y_train`）來訓練 SVC（而不是「5 對其餘」目標類別 `y_train_5`），接著進行預測（在這個例子是正確的）。在引擎蓋下，Scikit-Learn 其實使用了 OvO 策略，它訓練了 45 個二元分類器、取得它們對圖片的研判分數，接著選擇在互相競爭中勝出的類別。

當你呼叫 `decision_function()` 方法時，你會看到它為每個實例回傳 10 個分數（而不是只有 1 個），每個類別有一個分數：

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

最高分確實是對應類別 5 的那一個：

```
>>> 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
```

當分類器被訓練好之後，它會將目標類別串列存入它的 `classes_` 屬性，按值排序。在這個例子中，在 `classes_` 陣列裡面的各個類別的索引剛好是類別本身（例如索引 5 的類別剛好是類別 5），但通常你都不會這麼幸運。

如果你想要強迫 Scikit-Learn 使用一對一或一對其餘，可使用 `OneVsOneClassifier` 或 `OneVsRestClassifier` 類別。你只要建立一個實例，並且將一個分類器傳給它的建構式（它甚至不必是個二元分類器）。例如，這段程式以 SVC 建立一個 OvR 多類別分類器：

```
>>> from sklearn.multiclass import OneVsRestClassifier
>>> ovr_clf = OneVsRestClassifier(SVC())
>>> ovr_clf.fit(X_train, y_train)
>>> ovr_clf.predict([some_digit])
array([5], dtype=uint8)
>>> len(ovr_clf.estimators_)
10
```

訓練 SGDClassifier（或 RandomForestClassifier）也很簡單：

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

這一次 Scikit-Learn 不需要執行 OvR 或 OvO，因為 SGD 分類器可以直接將實例分成多個類別。decision_function() 方法現在為每個類別回傳一個值。看一下 SGD 分類器指派給各個類別的分數：

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

看起來分類器對它的預測相當有信心，幾乎所有分數都是大負數，而類別 5 則是 2412.5 分。這個模型對類別 3 有點猶豫，它得到 573.5 分。接著我們要評估這個分類器。與之前一樣，你可以使用交叉驗證，用 cross_val_score() 函式來評估 SGDClassifier 的準確度：

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

處理所有測試 fold 時，它有超過 84% 的準確度，使用隨機分類器可能得到 10% 的準確度，所以這個分數不至於太差，但可以更好。只要縮放輸入尺度（第 2 章談過）就可以將準確度提升至 89% 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([0.89707059, 0.8960948 , 0.90693604])
```

## 誤差分析

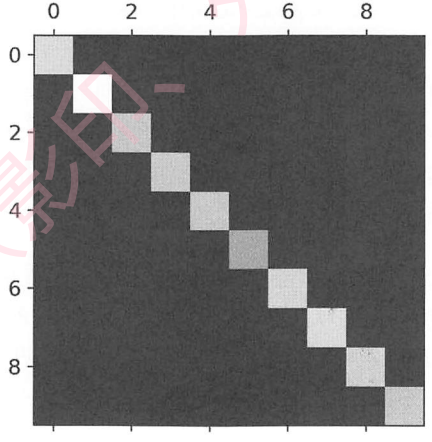
如果這是真正的專案，你現在就可以按照機器學習專案檢核表（見附錄 B）裡面的步驟進行操作。你會研究資料預備選項、嘗試多個模型（選出最好的，並使用 GridSearchCV 微調它們的超參數），並且盡可能地自動化。在此，我們假設你已經找到有希望的模型，而且你想要找到改善它的方法，有一種做法是分析它造成的誤差的類型。

先看一下混淆矩陣。跟之前的做法一樣，先用 cross_val_predict() 函式來進行預測，接著呼叫 confusion_matrix() 函式：

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

好多數字！查看混淆矩陣的圖像通常比較方便，我們使用 Matplotlib 的 matshow() 函式：

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



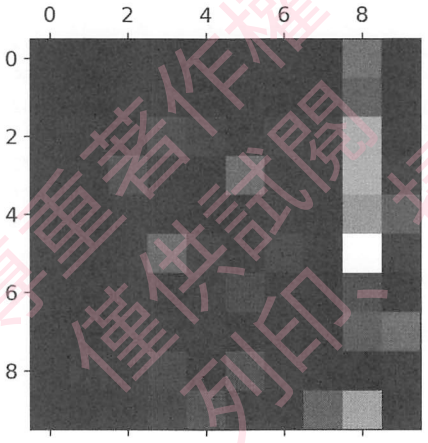
這個混淆矩陣看起來很好，因為大部分的圖片都在主對角線上，代表它們都被正確地分類。5 的顏色比其他數字深，這可能代表 5 圖片的數量在資料組裡面比較少，或分類器處理 5 的能力不像其他數字那麼好。事實上，你可以檢驗兩者。

我們先來畫誤差圖。首先，你要將混淆矩陣的每一個值除以對應類別的圖片數量，來比較錯誤率，而不是絕對錯誤數量（這個數字會讓數量豐富的類別看起來很糟）：

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

將對角線填上零，只保留錯誤率，並畫出結果：

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




你可以清楚地看到分類器犯下的錯誤種類。請記得，列代表真正的類別，行代表預測的類別。類別 8 那一欄很亮，代表有許多圖片被錯誤地分類為 8。但是，類別 8 那一列沒那麼糟，代表真正的 8 一般會正確地被歸類為 8。如你所見，混淆矩陣不一定是對稱的。你也可以看到分類器經常（雙向）混淆 3 與 5。

分析混淆矩陣通常可以找出改善分類器的做法。從這張圖看來，你應該努力防止分類器將其他數字歸類為 8。例如，你可以試著收集更多看起來像 8（但不是）的訓練資料，讓分類器學習區分它們與真正的 8。你也可以設計新特徵來協助分類器——例如，寫個演算法來計算封閉迴圈的數量（例如，8 有兩個，6 有一個，5 沒有）。你也可以預先處理圖片（例如使用 Scikit-Image、Pillow 或 OpenCV）來突顯一些圖案（例如封閉的迴圈）。

分析個別的錯誤也可以瞭解你的分類器在做什麼，以及為何它失敗了，但這種做法比較困難而且花時間。例如，我們畫出 3 與 5 的樣本（`plot_digits()` 函式只有使用 Matplotlib 的 `imshow()` 函式；詳情見本章的 Jupyter 筆記本）：

```
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)
plt.subplot(223); plot_digits(X_ba[:25], images_per_row=5)
plt.subplot(224); plot_digits(X_bb[:25], images_per_row=5)
plt.show()
```

左邊的兩個 5 × 5 區塊是被歸類為 3 的數字，右邊的兩個 5 × 5 區塊則是被歸類為 5 的圖片。有些讓分類器錯誤分類的數字（例如在左下與右上區塊裡面的數字）寫得十分潦草，就連人類都難以分辨（例如，第一列第二行的 5 看起來很像寫得很醜的 3）。但是，大部分被分錯類別的圖片對我們來說都是明顯的錯誤，我們不容易瞭解為何分類器犯下錯誤[3]，原因是我們使用了簡單的 `SGDClassifier`，它是個線性模型。它的做法只是幫每一個像素設定各個類別的權重，當它看到新圖片時，它只是把加權的像素強度總和起來，得到每個類別的分數。所以因為 3 與 5 的差異只有幾個像素，這個模型很容易分不清它們。

3 但是請記得，我們的大腦是奇妙的圖案辨識系統，在意識到任何資訊之前，視覺系統就已經做了大量、複雜的前置作業了，因此，我們認為很簡單不代表確實如此。

3 與 5 的差異主要是最上面的橫線與底下的圓弧之間的小直線。如果你寫 3 的時候稍微左偏，分類器就可能會把它歸類為 5，反之亦然。換句話說，這個分類器對圖片的偏移和旋轉相當敏感。所以為了減少 3/5 之間的混淆，你可以預先處理圖片，來確保它們都被置中而且不被過度旋轉。這或許也可以協助減少其他的錯誤。

## 多標籤分類

到目前為止，各個實例都只被指派給一個類別。有時你可能希望分類器為各個實例輸出多個類別。例如人臉辨識分類器：當它在同一張照片裡面認出很多人時，它該怎麼做？它應該為每一個被認出來的人加上一個標籤。假如分類器被訓練成可以認出三張臉：Alice、Bob 與 Charlie，當分類器看到 Alice 與 Charlie 的合照時，它應該輸出 [1, 0, 1]（代表「Alice yes, Bob no, Charlie yes」）。這種可輸出多個二元標籤的分類系統稱為**多標籤分類系統**。

我們還不打算討論人臉辨識系統，為了說明，我們先看一個比較簡單的範例：

```
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)
```

這段程式建立一個 `y_multilabel` 陣列，裡面有各個數位圖像的目標標籤：第一個代表它是不是大數字（7、8、9），第二個代表它是不是奇數。下一行建立一個 `KNeighborsClassifier` 實例（它支援多標籤分類，但並非所有分類器都如此），我們用多目標陣列來訓練它。現在你可以進行預測，注意它輸出兩個標籤：

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

這是對的！數字 5 確實不大（`False`），而是奇數（`True`）。

評估多標籤分類器的方法很多，而正確的評量標準依你的專案而定。你可以為各個單獨的標籤（或之前提到的任何其他二元分類器評量標準）評量 $F_1$ 分數，接著直接計算平均分數。這段程式計算所有標籤的平均 $F_1$ 分數：

```
>>> 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")
0.976410265560605
```

這段程式假設所有標籤的重要性一樣，但是事實不一定如此，例如，如果 Alice 的照片比 Bob 或 Charlie 的多，你可能會讓 Alice 照片的分類器分數有更多權重。有一種簡單的做法是讓每一個標籤的權重等於它的 *support*（也就是有該標籤的實例數量）。為此，你只要在上面的程式中設定 `average="weighted"` 即可[4]。

## 多輸出分類

我們要討論的最後一種分類是**多輸出多類別分類**（或簡稱**多輸出分類**）。它只不過是更廣泛的多標籤分類，各個標籤都可以有多個類別（也就是它的值可以超過兩個）。

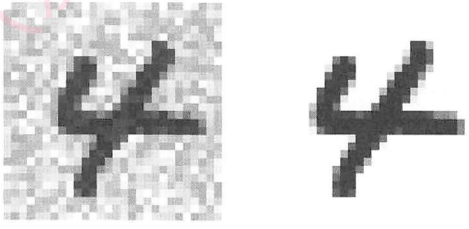
為了說明，我們來建立一個移除圖像雜訊的系統。它會接收一張有雜訊的數位圖像，並且會（希望如此）輸出一張乾淨的數位圖像，圖像是用像素強度陣列來表示的，就像 MNIST 的圖片那樣。請注意，分類器的輸出是多標籤的（每個像素一個標籤），而且各個標籤可以是多個值（像素強度的範圍是 0 到 255）。因此它是個多輸出分類系統。

分類與回歸之間的界限有時很模糊，就像這個範例。我們可以說，預測像素強度比較像回歸，而不是分類。此外，多輸出系統並非只能進行分類任務；你甚至可以讓系統為每個實例輸出多個標籤，包括類別標籤與值標籤。

我們先取得 MNIST 圖片，並且用 NumPy 的 `randint()` 函式將雜訊加到它們的像素強度。目標圖片是原始圖片：

```
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
```

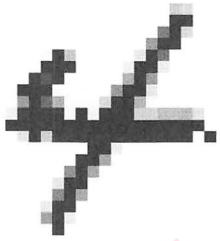
我們來看一下測試組的圖片（是的，我們正在偷窺測試資料，所以現在你應該要皺著眉頭）：



4 Scikit-Learn 還有一些其他的計算平均值的選項與多標籤分類器評量標準，詳情請參考文件。

左邊是有雜訊的輸入圖片，右邊是乾淨的目標圖片。接著我們要訓練分類器，讓它清理這張圖片：

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



看起來很接近目標！我們的分類之旅在此結束。你現在應該知道如何幫分類任務選擇正確的評量標準、選擇適當的 precision/recall、比較分類器，以及為各種任務建立良好的分類系統了。

## 習題

1. 試著為 MNIST 資料組建立一個分類器，讓它在處理測試組時，有超過 97% 的準確度。提示：`KNeighborsClassifier` 很適合這項任務，你只要找到好的超參數值就可以了（試著對 `weights` 與 `n_neighbors` 超參數進行網格搜尋）。

2. 寫一個可以將 MNIST 圖片往任何方向移動（左、右、上、下）一個像素的函式[5]。接著，為訓練組的每一張圖片建立四張移動過的複本（每個方向一張），並將它們加入訓練組。最後，用這個擴展過的訓練組來訓練你的最佳模型，並且用測試組評量它的準確度。你應該可以看到，模型的表現更好了！這項人工擴展訓練組的技術稱為**資料擴增**（*data augmentation*）或**訓練組擴展**（*training set expansion*）。

3. 處理 Titanic 資料組。Kaggle 是很棒的起點（*https://www.kaggle.com/c/titanic*）。

4. 建立垃圾郵件分類器（這個習題比較有挑戰性）：

   - 從 Apache SpamAssassin（*https://homl.info/spamassassin*）的公共資料組下載垃圾郵件與一般郵件範例。

5 你可以使用 `scipy.ndimage.interpolation` 模組的 `shift()` 函式。例如，`shift(image, [2, 1], cval=0)` 會將圖片下移兩個像素，右移一個像素。

- 將資料組解壓縮，並且熟悉資料格式。
- 將資料組拆成訓練組與測試組。
- 寫一個資料預備 pipeline 來將各個 email 轉換成特徵向量。你的預備 pipeline 要將 email 轉換成代表各個可能出現的單字是否真的出現的（稀疏）向量。例如，如果所有郵件都只有四個單字，「Hello」、「how」、「are」與「you」，那麼 email「Hello you Hello Hello you」會被轉換成向量 [1, 0, 0, 1]（代表 [ 有「Hello」，沒有「how」，沒有「are」，有「you」]），或 [3, 0, 0, 2]，如果你比較喜歡計算各個單字出現的次數的話。

  你可能要在預備 pipeline 加入超參數來控制是否刪除 email 標題、將每個 email 轉換成小寫、刪除標點符號、將所有 URL 換成單字「URL」、將所有數字換成單字「NUMBER」，甚至執行 *stemming*（也就是移除字尾，有一些 Python 程式庫可執行這項工作）。

  最後，嘗試幾個分類器，看看你能不能建立一個很棒的垃圾郵件分類器，具備高 recall 與高 precision。

你可以在 *https://github.com/ageron/handson-ml2* 的 Jupyter 筆記本找到這些習題的解答。