3、分类

2019年1月5日　　21:10

1. MNIST

   sklearn提供了许多助手功能来帮助下载流行的数据集。MNIST就是其中之一，下面是获取MNIST数据集的代码。

```
>>> 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.])}
```

默认情况下，sklearn将下载的数据集缓存在目录HOME/scikit_learn_data中

   sklearn加载的数据集通常具有类似的字典结构，包括：

   a. DESCR：描述数据集
   b. data：包含一个数组，每个实例为一行，每个特征为一列（m,n）
   c. target：包含一个带有标记的数组（m,）

   MNIST数据集自动将数据集分为训练集和测试集（一共有70000张图片，前60000张都是训练集），但我们还需要将训练集洗牌。

```
X_train,y_train,X_test,y_test = X[:60000],y[:60000],X[60000:],y[60000:]
import numpy as np
shuffle_index = np.permutation(60000)
X_train,y_train = X_train[shuffle_index],y_train[shuffle_index]
```

2. 训练一个二分类器

   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”就是一个二分类器的例子，它只能区分两个类别：5和非5.为此分类任务创建目标向量：

```
y_train_5 = (y_train==5)
y_test_5 = (y_test==5)
```

   接着挑选一个分类器开始训练。一开始可以用sklearn的SGDClassifier类试试。

```
from sklearn.linear_model import SGDClassifier

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

   然后可以用它来检测数字5的图像了

```
sgd_clf.predict(some_digit.reshape(1,-1))
True
```

默认是一个软间隔SVM，通过改变损失函数可以切换分类器

这个SGDClassifier类一般利用随机梯度下降法求解参数的算法的解，例如SVM、LR等，而sklearn中的LR、SVM从来并非用梯度下降法求解的。

sgd_clf.predict(some_digit.reshape(1,-1))
True

3. 性能考核
   a. 使用交叉验证测量accuracy
   相比于cross_val_score()，有时候可能希望自己能控制的多一些。在这种情况下，可以自行实现交叉验证。

```
from sklearn.model_selection import StratifiedKFold   #Stratified就是分层的意思
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_corr = sum(y_pred==y_test_fold)
    print(n_corr/len(y_pred))
```

   每个折叠由stratifiedKFold执行分层抽样产生，其所包含的各个类的比例符合整体比例。现在，也可以用自带的cross_val_score()函数来评估SGDClassifier模型。

```
from sklearn.model_selection import cross_val_score
cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring='accuracy') #这个函数返回的是一个array
array([0.95455, 0.9633 , 0.9527 ])
```

   看起来折叠交叉验证的准确率很高，但是实际上如果你把每张图片都预测为“非5”，也会得到一个很高的准确率。

```
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)

never5clf = Never5Classifier()
cross_val_score(never5clf,X_train,y_train_5,cv=3,scoring='accuracy')
```

```
array([0.9088 , 0.91015, 0.91   ])
```

   这说明准确率通常无法称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（即某些类别比其他类别更为频繁）

   b. 混淆矩阵
   评估分类器性能更好的方法是混淆矩阵。总体思路就是统计A类别实例被分类成B类别的次数。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

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)
```

与cross_val_score一样，cross_val_predict同样执行K-fold交叉验证，但是返回的不是评估分数，而是每个折叠的预测（这里三个折叠的预测结果都被拼到一起了，返回的结构的shape是(60000,)）。这意味着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）。

现在就可以使用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([[53270,  1309],
       [ 1280,  4141]])
```

混淆矩阵中的行代表实际类别，列表示预测类别。本例中第一行表示所有“非5”（非负）的图片中：53270张被正确地分类为“非5”（真负类），1309张被错误地分类为“5”（假正类）；第二行表示所有5（正类）的图片中：1280张被错误地分为“非5”类别（假负类），4141张被正确地分类为“5”（真正类）。一个完美的分类器只有真正类和真负类，所以它的混淆矩阵只会在其对角线有非零值。

混淆矩阵能提供大量的信息，但有时你可能希望指标更简洁一点。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precision

precision= TP/(TP+FP)

precision通常与另一个指标一起使用，即recall，它的计算公式如下： #注意，recall也是针对正类而言的

recall = TP/(TP+FN)

sklearn提供了计算多种分类器指标的函数，precision和recall也是其中之一（这也是针对正类而言的）

```
from sklearn.metrics import precision_score,recall_score
print(precision_score(y_train_5,y_train_pred))
recall_score(y_train_5,y_train_pred)
```

```
0.7598165137614679
0.7638812027301236
```

现在再看，这个5-检测器看起来似乎并不像它的准确率那么光鲜亮眼了。当它说一张图片是5的时候，只有76%的时间是准确的，并且也只有76%的数字5被检测出来了。并且我们可以把precision和recall组合成为一个单一的指标，称为F值。当需要一个简单的方法来比较两种分类器的时候，这是个不错的指标。F值是precision和recall的调和均值。调和均值会给较低的值跟高的权重。因此，只有

recall组合成为一个单一的指标，称为F值。当需要一个简单的方法来比较两种分类器的时候，这是个不错的指标。F值是precision和recall的调和均值，调和均值会给较低的值跟高的权重。因此，只有precision和recall都很高的时候，分类器才会有较高的F值。要计算F1值，只需要调用f1_score()即可：

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

```
0.7618434366663601
```

F1值对哪些具有相近的precision和recall的分类器更有利。

c. precision/recall权衡

在分类这一任务中，对于某个example，若分类函数给出的预测结果大于某个阈值，则分类器判断为正类，但是sklearn不允许我们直接设置阈值，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，使用任意的阈值进行预测了。

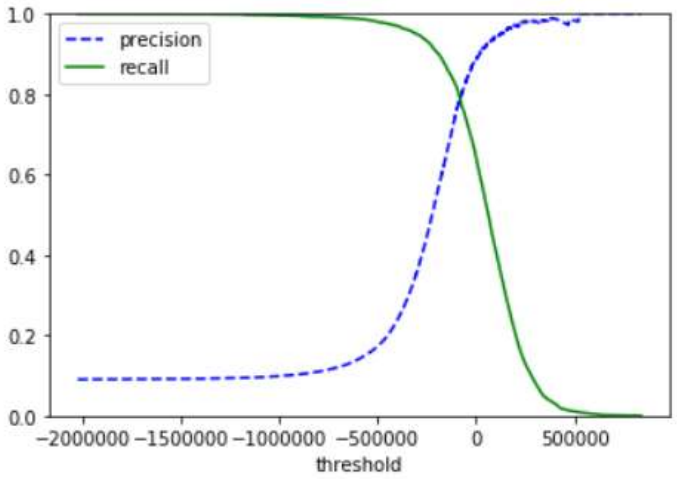
首先使用cross_val_predict()函数获取训练数据中所有实例的分数，但是这次需要它返回的是决策分数而不是预测结果

```
y_sources = 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_sources)
```

```
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()
```



现在就可以通过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。

d. ROC曲线

还有一种经常与二元分类器一起使用的工具，叫做受试者工作特征曲线。它与精度/召回率曲线非常

d. ROC曲线

还有一种经常与二元分类器一起使用的工具，叫做受试者工作特征曲线。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，也成为特异度。

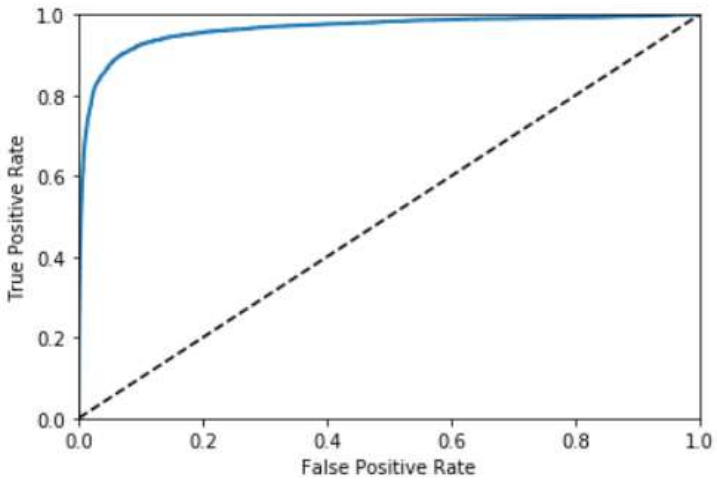
要绘制ROC曲线，首先需要使用roc_curve()函数计算多种阈值的TPR和FPR:

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

然后，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--')
    plt.axis([0,1,0,1])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')

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



同样这里在此面临一个折中权衡：召回率（TPR）越高，那么假正类就越高。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，而纯随机分类的ROC AUC等于0.5。sklearn提供计算ROC AUC的函数。

```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_train_5,y_sources)
0.9647179380917592
```

由于ROC曲线和精度/召回率（PR）曲线非常相似，因此你可能会问如何决定使用哪种曲线。有一个经验法则是，当正类非常少见或者你更关注假正类时，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），可能会觉得分类器真不错，但这主要是因为跟负类（非5）相比，正类（5）的数量很少。相比之下，PR曲线清楚地说明了分类器还有改进的空间（曲线还可以更接近右上角）。

训练一个RandomForestClassifier分类器，并比较它和SGDClassifer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，RandomForestClassifer类没有

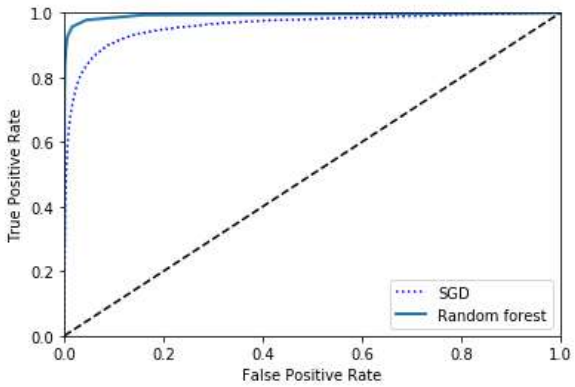
训练一个RandomForestClassifier分类器，并比较它和SGDClassifer分类器的ROC曲线和ROC AUC分数。首先，获取训练集中每个实例的分数。但是由于它的工作方式不同，RandomForestClassifer类没有decision_function方法，相反，他有的是dict_proba方法。sklearn的分类器通常都会有这两种方法中的一种。dirc_proba方法会返回一个数组，其中每行为一个实例（多列），每列代表一个类别，意思是某个给定实例属于某个给定类别的概率。

```
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]
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,y_scores_forest)
```

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



从上图可以看出来，RF的ROC曲线看起来比SGDClassifier好很多：它离左上角更接近。因此它的ROC AUC分数也高得多：

```
roc_auc_score(y_train_5,y_scores_forest)
```

```
0.9931232888526922
```

上述即为一些如何训练二分类器、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。

4. 多类别分类器

有一些算法（如随机森林分类器或者朴素贝叶斯）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。

例如，要创建一个系统将数字图片分为10类，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，。。。）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。

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1，一个用于区分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 X (N-1) / 2个分类器。对于MNIST问题，这意味需要训练45个二元分类器，当需要对一张图片进行分类时，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。

类别进行训练。

有些算法（例如SVM）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，由于在较小数据集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。但是对于大多数二元分类器来说，OVA还是更好的选择。

sklearn可以检测到你尝试使用二元分类算法进行多分类算法进行分类任务，它会自动运行OVA（SVM分类器除外，它会使用OVO）。下面用SGDClassifier试试：

```
sgd_clf.fit(X_train,y_train)
sgd_clf.predict([some_digit])
```

array([5.])

这段代码使用原始目标类别0到9在训练集上对SGDClassifier进行训练，而不是以“5”和“非5”作为目标类别。然后做出预测。而在内部，sklearn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。我们可以调用decision_function（）方法。它会返回10个分数，每个类别1个，而不是每个实例返回1个分数（是否是正类）。

```
some_digit_scores = sgd_clf.decision_function([some_digit])
```

```
some_digit_scores
```

```
array([[-258427.51899065, -436802.09299734, -320895.04818437,
         -52509.62350253, -551390.06245406,   54861.04777531,
        -779260.7669452 , -296222.69362636, -578026.66093227,
        -528513.31695052]])
```

（虽说这里的sgd_clf实质上也是svm，因为用的loss是hinge loss，但是这里用的也是OVA，所以上文的意思应该是只有严格意义上的SVM才会运行OVO吧）

如果想要强制sklearn使用一对一或者一对多策略，可以使用OneVsOne Classifier或者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，然后基于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
```

训练randomforestClassifier同样简单：

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

array([5.])

这次sklearn不必运行ova或者ovo了，因为随机森林分类器直接就可以将每个实例分为多个类别。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类别列表：

```
forest_clf.predict_proba([some_digit])
```

```
array([[0., 0., 0., 0., 0., 1., 0., 0., 0., 0.]])
```

这时，如果要评估这些分类器，也可以使用交叉验证。比如cross_val_score()

cross_val_score(sgd_clf,X_train,y_train,cv=3,scoring='accuracy')

array([0.86707658, 0.88094405, 0.81107166])

这些准确率依然可以提高，例如将输入进行简单缩放可以将准确率提升到90%以上：

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_train)
cross_val_score(sgd_clf,X_scaled,y_train,cv=3,scoring='accuracy')
```

array([0.90841832, 0.90934547, 0.91148672])

5. 错误分析

如果这是一个真正的项目，你将遵循机器学习项目清单中的步骤（附录B）：探索数据准备的选项，尝试多个模型，列出最佳模型并用GridSearch对其超参数进行微调，尽可能自动化，等等。现在假设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行进一步改进。方法之一就是分析其错误类型。

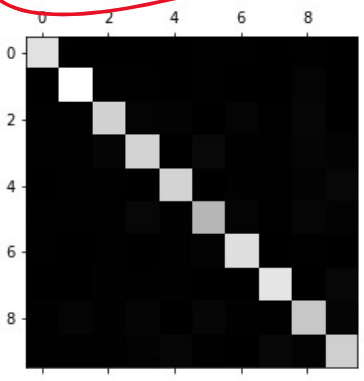
首先，看看混淆矩阵。就像之前做的，使用cross_val_predict()函数进行预测，然后调用confusion_matrix（）函数。

```
y_train_pred = cross_val_predict(sgd_clf,X_scaled,y_train,cv=3)
conf_mat = confusion_matrix(y_train,y_train_pred)
conf_mat
```

```
array([[5730,    2,   24,    9,   12,   50,   42,    8,   42,    4],
       [   1, 6475,   41,   27,    7,   36,    9,   11,  124,   11],
       [  59,   37, 5316,  105,   79,   24,  102,   50,  169,   17],
       [  49,   36,  137, 5337,    3,  234,   37,   56,  145,   97],
       [  16,   28,   41,   11, 5356,   12,   56,   28,   80,  214],
       [  70,   36,   33,  187,   71, 4617,  112,   32,  172,   91],
       [  33,   20,   45,    1,   39,   87, 5646,    7,   40,    0],
       [  23,   18,   65,   29,   56,   11,    5, 5815,   17,  226],
       [  53,  151,   68,  144,   13,  165,   62,   26, 5037,  132],
       [  41,   29,   26,   92,  156,   41,    3,  224,   81, 5256]])
```

数字有点多，使用matplotlib的matshow（）函数来查看混淆矩阵的图像表示，通常更加方便：

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



混淆矩阵看起来很不错，因为大多数图片都在主对角线上，这说明它们被正确分类。数字5看起来比其他比其他数字稍稍暗一些，这可能意味着数据集中数字5的图片较少，也可能是分类器在数字5上的执行效果不

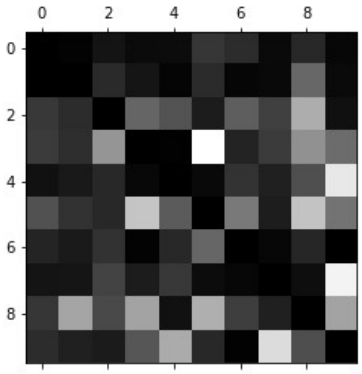
如在其他数字上好。实际上，可能两种情况都存在。（混淆矩阵中的行代表实际类别，列表示预测类别。）

让我们把焦点放在错误上。首先，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_nums = conf_mat.sum(axis=1,keepdims=True)
norm_conf_mx = conf_mat/row_nums
```

用0填充对角线，只保留错误，重新绘制结果：

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



现在可以清晰地看到分类器产生的错误种类了。每行代表实际类别，每列表示预测类别。第八列和第九列整体看起来非常亮，说明有许多图片被错误地分类为数字8或数字9了。同样，类别8和类别9的行看起来也偏亮，说明数字8和数字9经常和其他数字混淆。相反，一些行很暗，比如行1，这意味着大多数数字1都被正确分类了。注意，错误不是完全对称的，比如，数字5被错误分类为8的数量比数字8被错误分类为数字5的数量要更多。

分析混淆矩阵通常可以帮助深入了解如何改进分类器。通过上面那张图来看，可以把精力花在改进数字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，可以收集更多的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）。再比如，可以对图片进行预处理，让某些模式更为突出，比如闭环之类。

6. 多标签分类

有些情况下，希望模型为每个实例产出多个类别。比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统称为多标签分类系统。

为了阐释清楚，这里不讨论面部识别，用一个简单的例子举例：

```
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)
```

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，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他进行训练。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]])
```

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F1分数（或者是之前罗列的任何其他二元分类器指标）。下面这段代码计算所有标签的平均F1分数：

```
y_trian_knn_pred = cross_val_predict(knn_clf,X_train,y_train,cv=3)
f1_score(y_train,y_train_knn_pred,average='macro')
0.9684
```

这里假设了所有的标签同样重要，但实际上可能不是这样。如果要区分各个类别的权重，一个简单的办法是给每个标签设置一个等于其自身支持的权重（也就是具有该目标标签的实例的数量）。只需要在上面的代码中设置average="weighted"即可。

7. 多输出分类

   最后一种分类任务叫做多输出-多类别分类（简称为多输出分类）。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（比如它可能有两个以上的可能的值）。

   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（之前是每个标签只有两个值）。