MNIST

本文的例子是比较经典的数字识别案例，使用分类器识别0到9的手写数字。test set和train set来自于MNIST，包含70000条数据。

首先，我们需要下载这份数据。代码如下：

```
from sklearn.datasets import fetch_openml
import numpy as np


def sort_by_target(mnist):
    reorder_train = np.array(sorted([(target, i) for i, target in
enumerate(mnist.target[:60000])]))[:, 1]
    reorder_test = np.array(sorted([(target, i) for i, target in
enumerate(mnist.target[60000:])]))[:, 1]
    mnist.data[:60000] = mnist.data[reorder_train]
    mnist.target[:60000] = mnist.target[reorder_train]
    mnist.data[60000:] = mnist.data[reorder_test + 60000]
    mnist.target[60000:] = mnist.target[reorder_test + 60000]


mnist = fetch_openml('mnist_784', version=1, cache=True)
mnist.target = mnist.target.astype(np.int8) # fetch_openml()
returns targets as strings
sort_by_target(mnist) # fetch_openml() returns an unsorted dataset
print("======")
print(mnist)
```

接下来看看这个数据矩阵的维数：

```
X, y = mnist["data"], mnist["target"]
print(X.shape)
print(y.shape)

'''
(70000, 784)
(70000,)
'''
```

一共有70000条数据，每条数据有784个特征，这是因为一张图片是28*28个像素，每个特征表示一个像素值，取值范围是0~255。接下来，我们就随便打印出一个数字图片来看一下：

```
%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()
```

看上边的图片，我们猜测这个数字应该是5，让我们打印下真实的值：

```
print(y[36000])

'''
5
'''
```

在训练模型之前，我们首先需要生成test set，幸运的是MNIST已经把这些工作做完了，在70000条数据中的前60000条作为train set，后10000条作为test set。

有一点需要注意，数字的排序是按照从小到大的，我们还需要把这些顺序重新打乱，因为有些算法对这种顺序的数据会比较敏感。

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000],
y[60000:]
# 有些算法对排好序的数据更加敏感，排好序说明两个行很相似
shuffle_index = np.random.permutation(60000)
X_train, y_train = X_train[shuffle_index],y_train[shuffle_index]
```

训练一个Binary Classifier

一个Binary Classifier表示，分类器能够输出的结果为 `是` 或 `不是` 。它是二元的，比如，我们想检测一个值是不是5，我们就把该分类器设置为是5或者不是5。

```
# 使用随机梯度下降算法
from sklearn.linear_model import SGDClassifier


y_train_5 = (y_train == 5)
y_test_t = (y_test == 5)

sgd_clf = SGDClassifier(random_state=42, max_iter=1000, tol=0.1)
sgd_clf.fit(X_train, y_train_5)

sgd_clf.predict([some_digit])

'''
array([ True])
'''
```

在上边的代码中，首先需要修改labels，把等于5的改成1，其他的改成0，然后用随机梯度下降分类器（Stochastic Gradient Descent (SGD)）训练，最后对some_digit做出预测, 可以看出，预测结果为True，表示some_digit是5。

性能测量

评估分类器的性能比评估回归要复杂的多，在下边的内容中，主要会讲解集中不同的方法。

使用Cross-Validation

交叉验证能够测量模型的精度，在使用该验证方法的时候，我们可以自定义验证函数：

```
# 自定义交叉验证
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))

'''
0.96395
0.96945
0.95135
'''
```

交叉验证本质上就是计算错误的数量，然后再除以总数，算出错误率。

```
from sklearn.model_selection import cross_val_score


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

'''
array([0.96395, 0.96945, 0.95135])
'''
```

从结果来看，达到了95%的精度，这看上去非常的理想，这时候就需要警惕了，我们看另一个例子：

```
# 下边演示一个对于分类器存在的问题，如果数据集是偏斜的，指包含某一类远远大于另一类，如果仅仅预测那个大的，
# 得到的精度也很高
# 下边的例子中有90%的结果都不是5，所以产生了很高的精度
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)


never_5_clf = Never5Classifier()
cross_val_score(never_5_clf, X_train, y_train_5, cv=3,
scoring="accuracy")

'''
array([0.9087 , 0.909  , 0.91125])
'''
```

上边的代码，实现了一个特别“傻”的分类器，不管输入是啥，都判断为不是5，再看结果，竟然达到了90%以上的精度，这就是**精度不能说明问题的原因**，由于是二元分类器，训练集中数据的分布（数据分布偏斜）对分类器有明显的影响。

混淆矩阵（Confusion Matrix）

Confusion Matrix的原理是分别统计计算结果为True和False的个数，然后组成矩阵，其中每一行表示真实的值，每一列表示预测值。他是通过观测向量和预测向量计算出来的。

```
from sklearn.model_selection import cross_val_predict


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

# 混淆矩阵
from sklearn.metrics import confusion_matrix


confusion_matrix(y_train_5, y_train_pred)

'''
array([[53399,  1180],
       [ 1125,  4296]])
'''
```

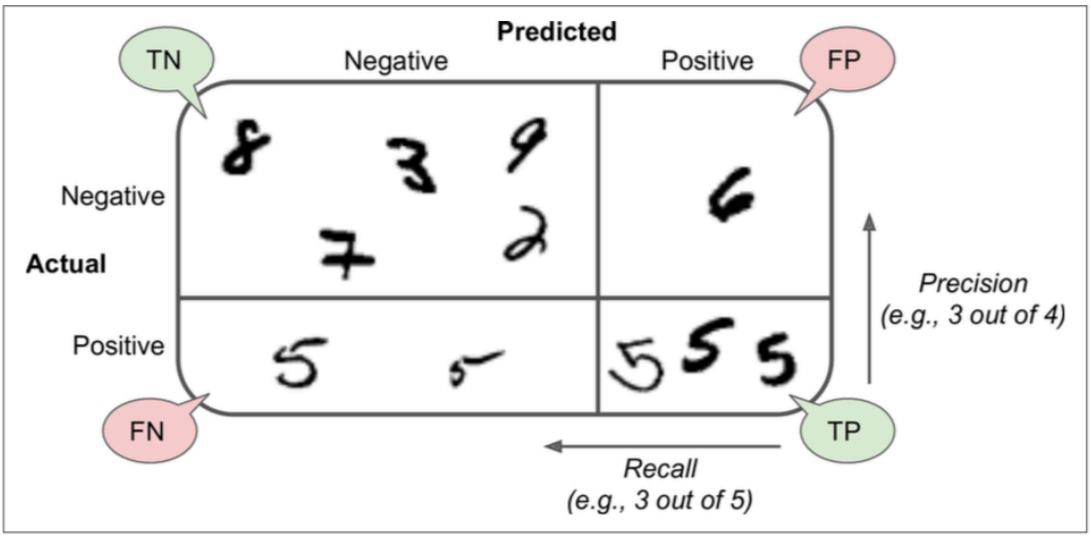
我们来分析下上边的打印结果，在矩阵的第一行，表示真实值为不是5的情况，其中，第一列表示预测正确，称为true negatives 第二列表示预测错误，称为false negatives。第二行表示真实值是5的情况，其中第一列表示预测正确，称为true positives，第二列表示预测错误，称为false positives。

基于该矩阵，我们就可以计算出比较重要的另外两个参数：precision(精确度)和recall(召回率)。

/	true negatives	false positives
/	false negatives	true positives

$$precision = \frac{TP}{TP + FP} \quad recall = \frac{TP}{TP + FN}$$

precision表示第二列的比值，recall表示第二行的比值。用一张图来看更加清晰：



Precision和Recall

sklearn中提供了计算这两个参数的方法：

```
from sklearn.metrics import precision_score, recall_score


precision_score(y_train_5, y_train_pred)

'''
0.7845142439737034
'''
```

```
recall_score(y_train_5, y_train_pred)

'''
0.7924737133370227
'''
```

从上边的打印结果可以看出，precision为78%左右，表示分类器只预测对了大概78%左右的比例，recall为79%左右表示只检测除了大概79%的为5的值。

precision和recall可以组合使用，这就是f1分数：

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

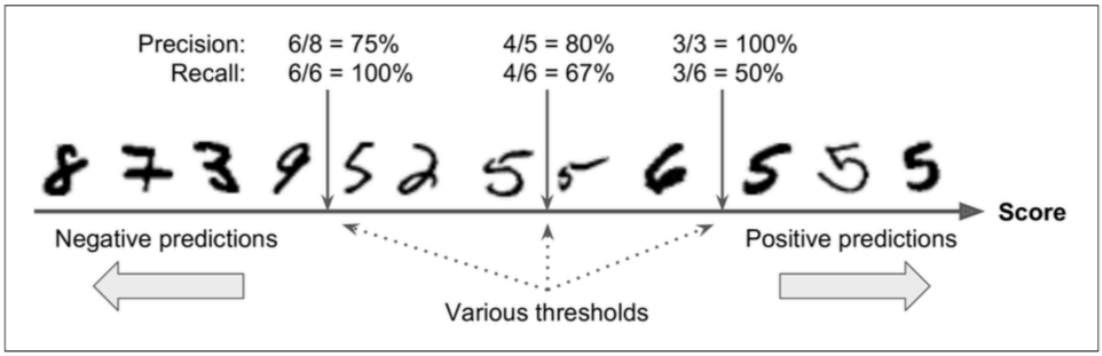
如果简单的对比两个分类器的性能，用f1分数就可以了：

```
from sklearn.metrics import f1_score


f1_score(y_train_5, y_train_pred)

'''
0.7884738918968522
'''
```

通过上边f1的公式就可以看出，f1更偏好precision和recal相似的分类器，这样的分类器能够得到更高的f1分数，但在真实开发中，我们往往更想得到高的precision或者recall。比如，如果你想做一个儿童视频分类器，你肯定想获得更高的recall，也就是牺牲掉更多的好的视频来保证不好的视频的出现，也就是提高precision。又比如，你做一个监控中识别小偷的分类器，你肯定希望获取更低的precision，宁可更多的识别出不是小偷的人，提高recall。

Precision/Recall权衡

为了明白如何权衡这两个值，我们先要弄明白SGDClassifier是如何对数据进行决策的，当它预测一个实例的时候，它内部会调用一个decision function，也就是决策函数，这个函数会返回一个分数，它内部还定义了一个threshold，当分数超过这个下限的时候，就认为是positive class，否则为negative class。



我们分析下上图，score从左到右增大，假设threshold设置为中间的箭头位置的值，看一看到这时候有4个5，也就是true positive，这时候的precision就是4/5(80%),由于一共有6个5，预测正确了4个5，因此它的recall是4/6(67%)。现在把threshol往右移动一个箭头，precision变为3/3(100%),recall变为3/6(50%)。**这说明通过调整threshold，可以改变precision和recall。**

sklearn虽然不能直接设置threshold的值，但是可以通过`decision_function()`获取到分数。

```
# 通过决策函数可以返回决策分数，通过控制这个分数的下限，就可以实现精度和召回的控制
y_scores = sgd_clf.decision_function([some_digit])
print(y_scores)

threshold = 0
y_some_digit_pred = (y_scores > threshold)
print(y_some_digit_pred)

threshold = 3000
y_some_digit_pred = (y_scores > threshold)
print(y_some_digit_pred)

'''
[108.23734896]
[ True]
[False]
'''
```

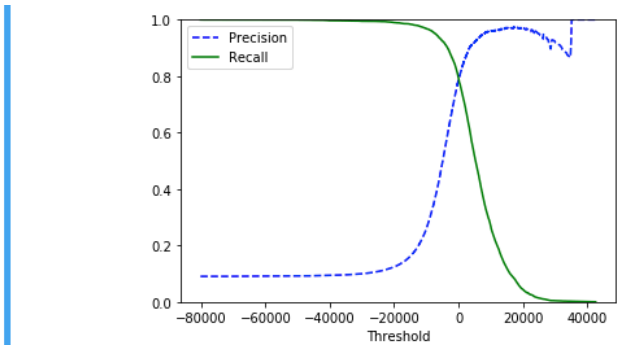
sklearn中SGDClassifier中的threshold的值为0，**那么问题来了，我如何获取threshold，才能获得理想的precision和recall呢?**

首先我们先获取所有的分数：

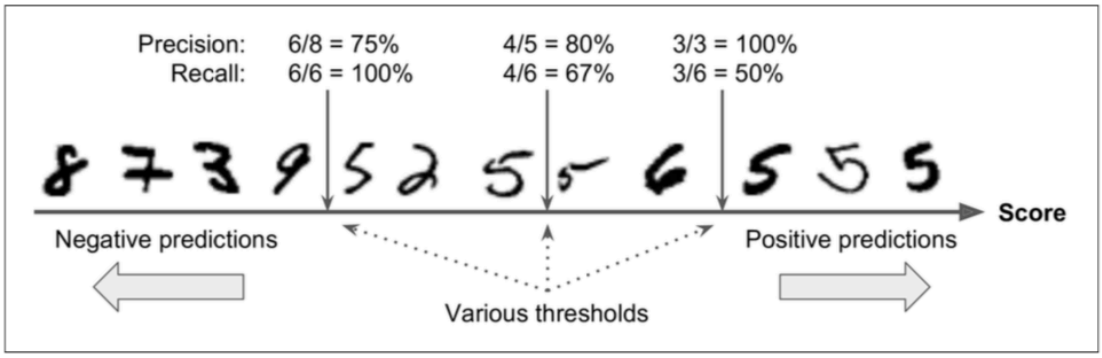
```
# 获取scores
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
method="decision_function")
print(y_scores)
print(len(y_scores))

'''
[-17186.54201291 -13902.72590167 -15972.40879221 ...
-16147.74895772
 -15201.1831391  -12556.41466713]
60000
'''
```

然后，使用 `precision_recall_curve()` 函数获取所有可能的thresholds, precisions, recalls。然后用这3个值画图。

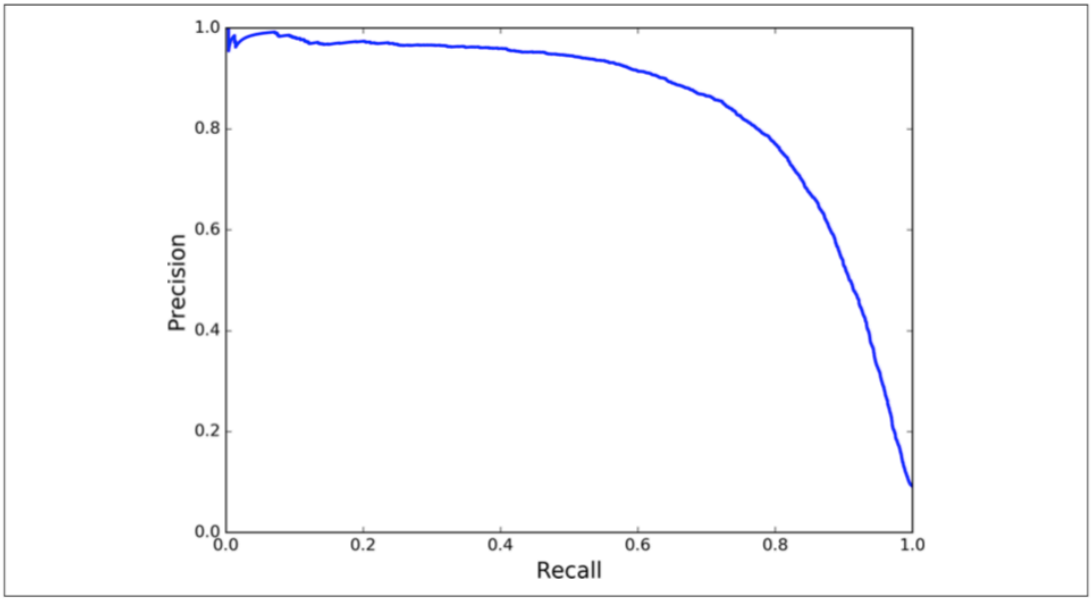
```
from sklearn.metrics import precision_recall_curve


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


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()
```



上图中有一点很奇怪，precision曲线呈现出了颠簸的性质，在右上角很明显，这是因为precision随着threshold的增大，值有可能减小。再看下边的图，如果把中间的那个threshold往右移动一个数字，这时候true positive就有3个，precision就是3/4(75%)，比没改变threshold的80%要小，这就解释了出现颠簸的原因。



通过上边的二维图基本上可以获取理想的precision和recall了，更进一步，我们把recall作为横坐标，precision作为纵坐标，作图如下：



上图更加清楚地看到这两者变化的关系。现在假设我们的目标是获取90%的precision，通过观察上图，得到如下代码：

```
# 如果确实想控制precision或者recall的话，可以这样实现
y_train_pred_90 = (y_scores > 5000)
precision_score(y_train_5, y_train_pred_90)
recall_score(y_train_5, y_train_pred_90)
```

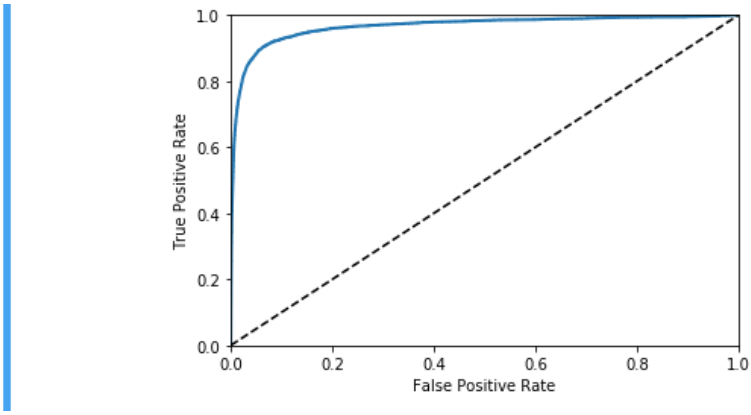
如果有人问，“我想得到99%的precision”，那么你应该问他，recall是多少?

ROC曲线

ROC是receiver operating characteristic (ROC)的简称，同样是一个测量二元分类器性能的工具。它和precision/recall很相似，知识横纵坐标的参数不太一样。

它的y坐标为true positive rate（TPR），也就是recall，它的x坐标为false positive rate（FPR），也就是所有的负值中，预测为正比上总的负值，它也等于1减去true negative rate（TNR）。TNR成为specificity特异率，因此，ROC实际画的就是敏感度（recall）/ (1 - 特异率)。

```
# 接收者操作特征曲线
from sklearn.metrics import roc_curve


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


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()
```



观察图形，y（recall）越大，x（FPR）也越大，恰恰说明了提高recall率的话，就会减低precision，也就是出现更多预测为false positive的结果。图中的虚线是分割线，分类器越好，越趋近于左上角。

可以通过计算曲线下边的面积来评估分类器的好坏，面积越靠近1越好。

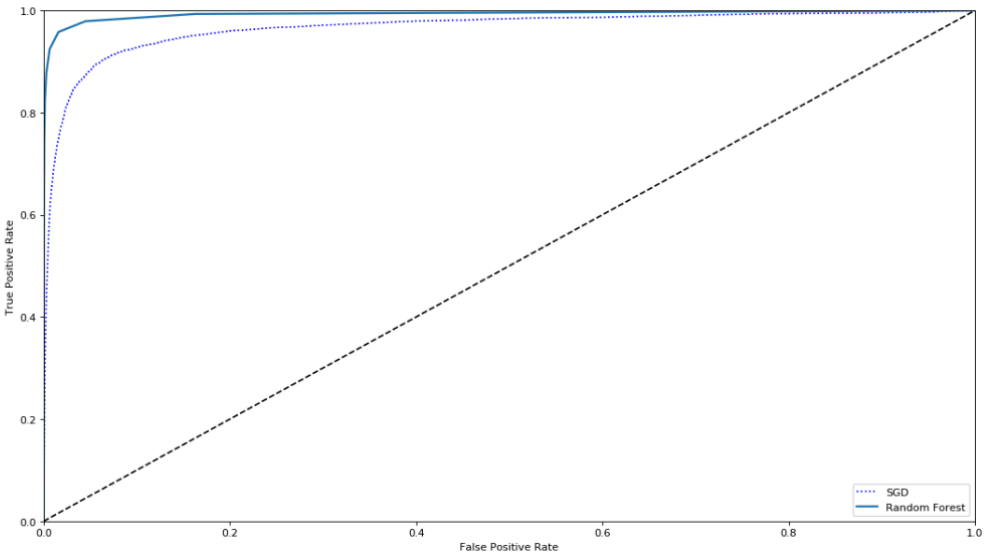
```
from sklearn.metrics import roc_auc_score


roc_auc_score(y_train_5, y_scores)

'''
0.9660644932844258
'''
```

在选择ROC或者PR时，基于其性质，选择正确的方案，当positive类比较少或者更关心false positive时使用PR，其他情况使用ROC。

接下来，我们选择另一种分类器，和之前的进行对比

```
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")
# y_probas_forest并不包含scores，因此我们用概率代替
y_scores_forest = y_probas_forest[:, 1]
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,
y_scores_forest)
```

```
from matplotlib.pyplot import figure
figure(num=None, figsize=(16, 9), dpi=80, facecolor='w',
edgecolor='k')


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




```
# 求曲线下的面积
roc_auc_score(y_train_5, y_scores_forest)

'''
0.9930611591045461
'''
```

Multiclass Classification

二元分类器只能预测True或False，如果我们预测的结果是多元的，那该怎么办呢？有一些算法本身就能很好的支持多元分类，比如Random Forest classifiers或naive Bayes classifiers。其他的算法则不行，因此需要一些策略来实现多元分类。

1. one-versus-all（OvA），比如说预测0~9，需要训练10个分类器，在对某个数字进行预测是，先求每个分类器的分数，选分数最大的分类器的结果作为预测值。
2. one-versus-one（OvO）,首先训练一个二元分类器，比如区分0和1， 1和2，等等，对于有n个类型的数据，一共需要n*(n-1)/2个分类器，这种策略需要的分类器很多，但是训练每个分类器需要的训练集很小，有些算法能够快速完成这些训练，比如svm

sklean中，如果对于多类问题，使用了二进制分类算法，则默认使用OvA，除非是svm

```
# sklean中，如果对于多类问题，使用了二进制分类算法，则默认使用OvA，除非是svm
sgd_clf.fit(X_train, y_train)
sgd_clf.predict([some_digit])

'''
array([5], dtype=int8)
'''
```

当SGDClassifier用于多元分类时，它内部会计算每个分类的分数，返回分数最大的那个分类。

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

'''
array([[-16435.99360932, -22020.2078959 , -16180.43731135,
         -2651.19394153, -11162.51700477,    108.23734896,
        -15435.22973158, -17660.76704045, -13998.69313799,
        -14196.54087665]])
'''
```

如果想使用OvO策略，可以使用sklearn的OneVsOneClassifier或OneVsRestClassifier，在创建他们是，传入一个二元分类器就可以了。

```
from sklearn.multiclass import OneVsOneClassifier


ovo_clf = OneVsOneClassifier(SGDClassifier(random_state=42,
tol=0.1))
ovo_clf.fit(X_train, y_train)
ovo_clf.predict([some_digit])
print(len(ovo_clf.estimators_))

'''
45
'''
```

上边代码的打印结果显示，其内容使用了45个分类器。随机森林有点不一样，它本身就预测了各个值的概率。

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

'''
[5]
'''
```

随机森林分类器内部计算了每个分类的概率，返回概率最高的那个值：

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

'''
[[0.  0.  0.  0.2 0.  0.8 0.  0.  0.  0. ]]
'''
```

最后，我们用交叉验证来评估分类器的性能：

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

'''
array([0.8815237 , 0.87829391, 0.87248087])
'''
```

可以看出，用SGDClassifier作为分类器只有84%左右的精度，要想提高这个精度，我们可以把缩放数据：

```
# 通过scaling参数，获取更好的结果
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([0.91276745, 0.90974549, 0.91218683])
'''
```

91%左右，这个精度还算比较理想。

Error Analysis

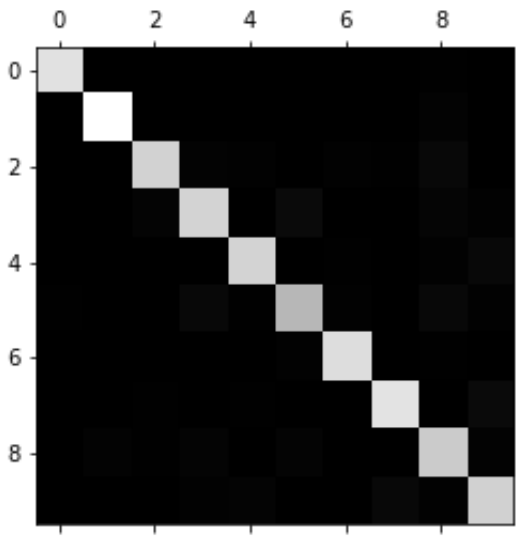
对于多元分类问题，同样可以使用confusion matrix，不过这个与二元分类还有一些不同的地方。首先我们先用代码看看这个矩阵长什么样?

```
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
```

```
Out[33]: array([[5718,    2,   22,   10,   11,   51,   44,    8,   53,    4],
                [   1, 6477,   46,   27,    6,   42,    5,   11,  116,   11],
                [  47,   34, 5329,  104,   80,   32,   79,   57,  185,   11],
                [  40,   37,  131, 5359,    1,  231,   32,   48,  157,   95],
                [  21,   24,   44,   11, 5355,    9,   53,   32,  100,  193],
                [  60,   32,   33,  183,   64, 4633,   91,   31,  208,   86],
                [  29,   25,   50,    2,   38,   96, 5607,    8,   62,    1],
                [  19,   23,   69,   31,   52,   11,    5, 5783,   21,  251],
                [  44,  127,   61,  146,    9,  142,   48,   17, 5137,  120],
                [  39,   31,   29,   91,  150,   38,    2,  180,   93, 5296]])
```

只看这些数字很难发现问题，我们使用Matplotlib的`matshow()`函数，把这些数据用图片表示：

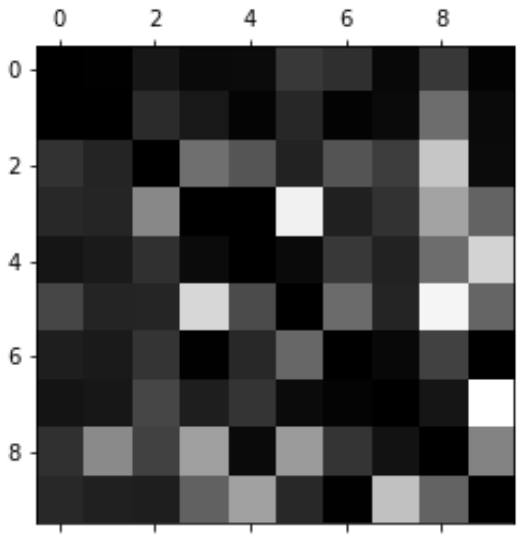
```
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```



这个矩阵看上去还不错，大部分浅色的图片都出现在了对角线上，说明这些值都预测正确了。看一看出数字5的颜色比较深一点，说明要么5在数据中的占比比较小，要么说明分类器对5的预测不够好。

现在，使用同样的方法，聚焦在错误上。对行求和后，在用矩阵除以这个数。

```
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()
```

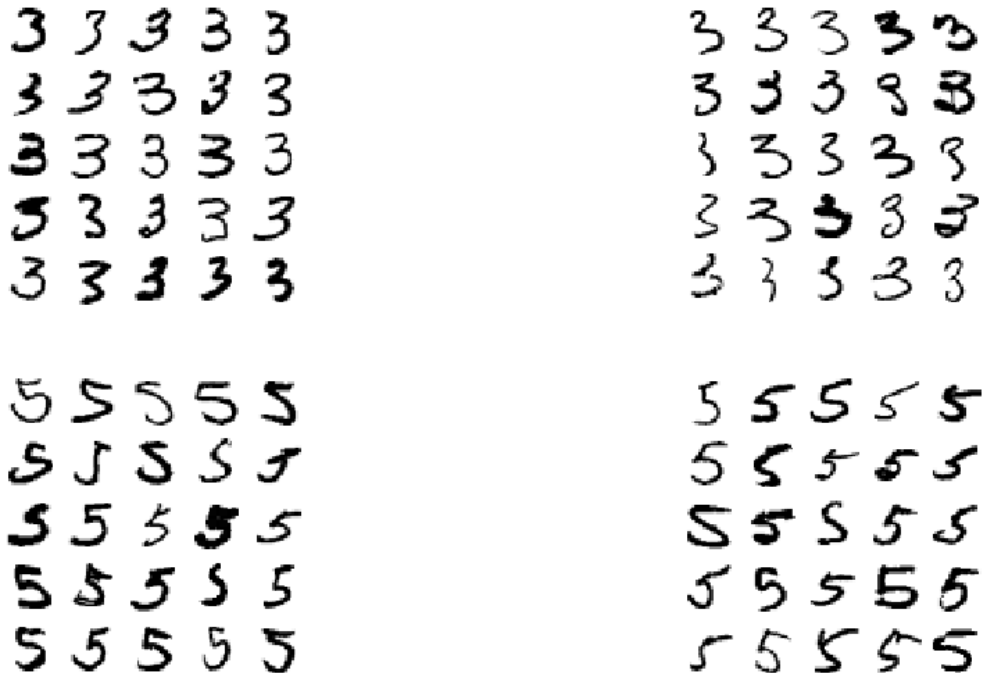


上图已经能够很清楚的看到哪里出错了，图中的每一行表示真实的值，每一列表示预测值。可以看出来第8列和第9列黑白间距比较混乱，说明有很多数字被错误的预测为8或者9了。同样第8行和第9行业比较混乱，说明8和9经常被错误的预测为别的数字了。也有一些比较理想的情况，比如第一行，基本都是很色的，说明对数字1的预测很理想，只有在8那一列会出现点问题。再比如数字5和8经常被相互预测错误。

因此，对于此案例，我们可以选择提升对8和9的预测，可以进一步处理3和5的混淆。

通过分析单独的一个数字，也能够获取提升性能的灵感：

```
def plot_digits(instances, images_per_row=10, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    images = [instance.reshape(size,size) for instance in
instances]
    n_rows = (len(instances) - 1) // images_per_row + 1
    row_images = []
    n_empty = n_rows * images_per_row - len(instances)
    images.append(np.zeros((size, size * n_empty)))
    for row in range(n_rows):
        rimages = images[row * images_per_row : (row + 1) *
images_per_row]
        row_images.append(np.concatenate(rimages, axis=1))
    image = np.concatenate(row_images, axis=0)
    plt.imshow(image, cmap = matplotlib.cm.binary, **options)
    plt.axis("off")


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=(16, 9))
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()
```



上图中的第一列是预测为3的值，第二列是预测为5的值，仔细观察这些数字，可以发现，这两个数字还是很容易被错误分类的。他们最大的不同就是最上边的那个笔画。由于SGDClassifier算法只是简单的把带有权值的像素值相加。

Multilabel Classification

我们上边讲的分类器只能返回一个值，在某些场景下，我们可能需要返回多个类别，比如人脸识别，对一张图片识别后，需要返回多个人名。

```
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)

knn_clf.predict([some_digit])

'''
array([[False,  True]])
'''
```

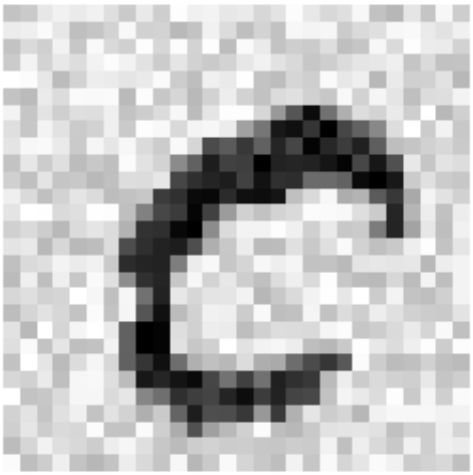
上边的代码可以输出两个类别：是否大于等于7和是否是偶数。

Multioutput Classification

multioutput classification表示输出的多个类别，每个类别又是多元的。举个例子，比方说输出一个上面中的28*28的数字，一个有784个类别，每个类别又有0~255个类别。我们先把数据中的每个实例加一些噪音：

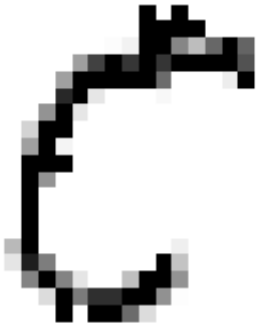
```
import numpy.random as rnd


train_noise = rnd.randint(0, 100, (len(X_train), 784))
test_noise = rnd.randint(0, 100, (len(X_test), 784))

X_train_mod = X_train + train_noise
X_test_mod = X_test + test_noise

y_train_mod = X_train
y_test_mod = X_test

print(X_test_mod)
some_index = 200
some_digit_image = X_test_mod[some_index].reshape(28, 28)
plt.imshow(some_digit_image, cmap=matplotlib.cm.binary,
interpolation="nearest")
plt.axis("off")
plt.show()
```

```
[[24. 91. 14. ... 34. 44.  8.]
 [29. 66. 50. ... 47. 48. 59.]
 [ 2. 15. 54. ... 50. 75. 31.]
 ...
 [40. 72. 82. ... 87. 87. 21.]
 [89. 56. 89. ... 90. 44. 32.]
 [ 5. 95. 67. ... 55. 83. 90.]]
```



然后训练并预测：

```
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[some_index]])
some_digit_image = clean_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap=matplotlib.cm.binary,
interpolation="nearest")
plt.axis("off")
plt.show()
```



可以看出来，过滤后的数据很不错。