Chapter 3: Classification

1. Import Data

In [ ]:
```
from sklearn.datasets import fetch_openml
import numpy as np
mnist = fetch_openml("mnist_784", version=1, cache=True, as_frame=False)
mnist.target = mnist.target.astype(np.int8)
mnist.keys()
```

Out[ ]:
```
dict_keys(['data', 'target', 'frame', 'categories', 'feature_names', 'target_name
s', 'DESCR', 'details', 'url'])
```

In [ ]:
```
X, y = mnist["data"], mnist["target"]
```

In [ ]:
```
mnist["data"], mnist["target"]
```

Out[ ]:
```
(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.]]),
 array([5, 0, 4, ..., 4, 5, 6], dtype=int8))
```

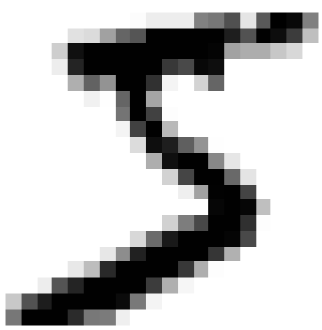
In [ ]:
```
print(X.shape)
print(y.shape)
```

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

There are 70,000 images and each image has 784 features (28 x 28 pixels)

In [ ]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

In [ ]:
```
some_digit = X[0]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap="binary")
plt.axis("off")
plt.show()
```



In [ ]:
```
y[0]
```
Out[ ]:
```
5
```

2. Train-Test-Split

In [ ]:
```
# since training set is already shuffled for us
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

3. Training a Binary Classifier

Try to only identify one digit. This is a binary classifier. Whether it is 5 or not 5

In [ ]:
```
y_train_5 = (y_train == 5) # True for all 5s and False for all othe digits
y_test_5 = (y_test == 5)
```

SGD classifier has the advantage of being capable of handling very large datasets efficiently. This is because SGD deals with training instances independently, one at a time. The SGDClassifier relies on randomness during training. If you want reproducible results, you should set the random_state parameter

In [ ]:
```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```
Out[ ]:
```
SGDClassifier(random_state=42)
```

In [ ]:
```
# detect images of number 5
sgd_clf.predict([some_digit])
```
Out[ ]:
```
array([ True])
```

4. Performance Measures

4.1 Measuring Accuracy Using Cross-Validation

Implementing CV from scratch

In [ ]:
```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3)

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))
```

```
0.95035
0.96035
0.9604
```

In [ ]:
```
from sklearn.model_selection import cross_val_score
cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

Out[ ]:
```
array([0.95035, 0.96035, 0.9604 ])
```

Compare results with a base classifier

In [ ]:
```
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)
```

In [ ]:
```
never_5_clf = Never5Classifier()
cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

Out[ ]:
```
array([0.91125, 0.90855, 0.90915])
```

Since there is only about 10% of the images that are 5s, if you always guess that an image is not a 5, you will right 90% of the time. This demonstrates why accuracy is generally not the preferred performance measure for classifiers, especially when you are dealing with a very skewed dataset.

4.2 Confusion Matrix

In [ ]:
```
from sklearn.model_selection import cross_val_predict

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

Instead of returning the evaluation scores, it returns the predictions made on each test fold

In [ ]:
```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_train_5, y_train_pred)
```

Out[ ]:
```
array([[53892,   687],
       [ 1891,  3530]], dtype=int64)
```

In [ ]:
```
y_train_pred
```

Out[ ]:
```
array([ True, False, False, ...,  True, False, False])
```

In [ ]:
```
y_train_5
```

Out[ ]:
```
array([ True, False, False, ...,  True, False, False])
```

4.3 Precision and Recall

Have to see which one you care about more. FP (precision) or FN (recall)

In [ ]:
```
from sklearn.metrics import precision_score, recall_score

print("Precision Score:", precision_score(y_train_5, y_train_pred))
print("Recall Score", recall_score(y_train_5, y_train_pred))
```

```
Precision Score: 0.8370879772350012
Recall Score 0.6511713705958311
```

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

Out[ ]:
```
0.7325171197343846
```

The higher the threshold, the lower the recall but higher the precision

In [ ]:
```
y_scores = sgd_clf.decision_function([some_digit])
y_scores
```

Out[ ]:
```
array([2164.22030239])
```

In [ ]:
```
threshold = 0
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
```

Out[ ]:
```
array([ True])
```

In [ ]:
```
threshold = 8000
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
```

Out[ ]:
```
array([False])
```

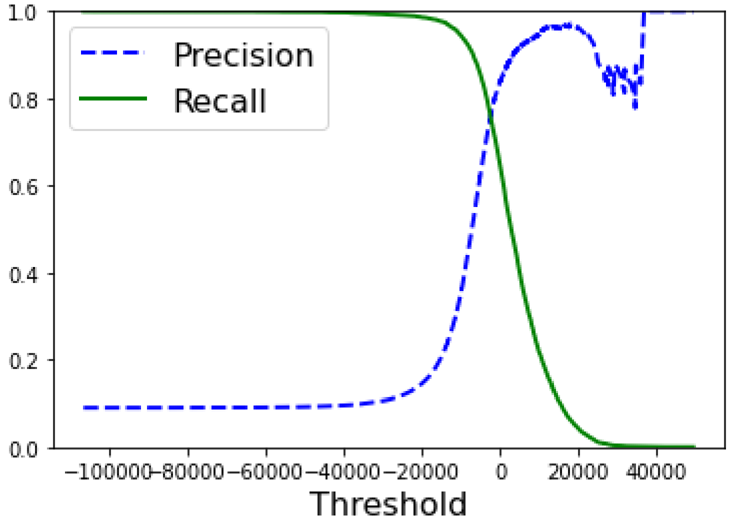
How to decide which threshold to use? Use the cross_val_predict function to get the scores of all instances in the training set

In [ ]:
```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3, method="decision_fu
```

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

In [ ]:
```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
    plt.xlabel("Threshold", fontsize=16)
    plt.legend(loc="upper left", fontsize=16)
    plt.ylim([0, 1])
```

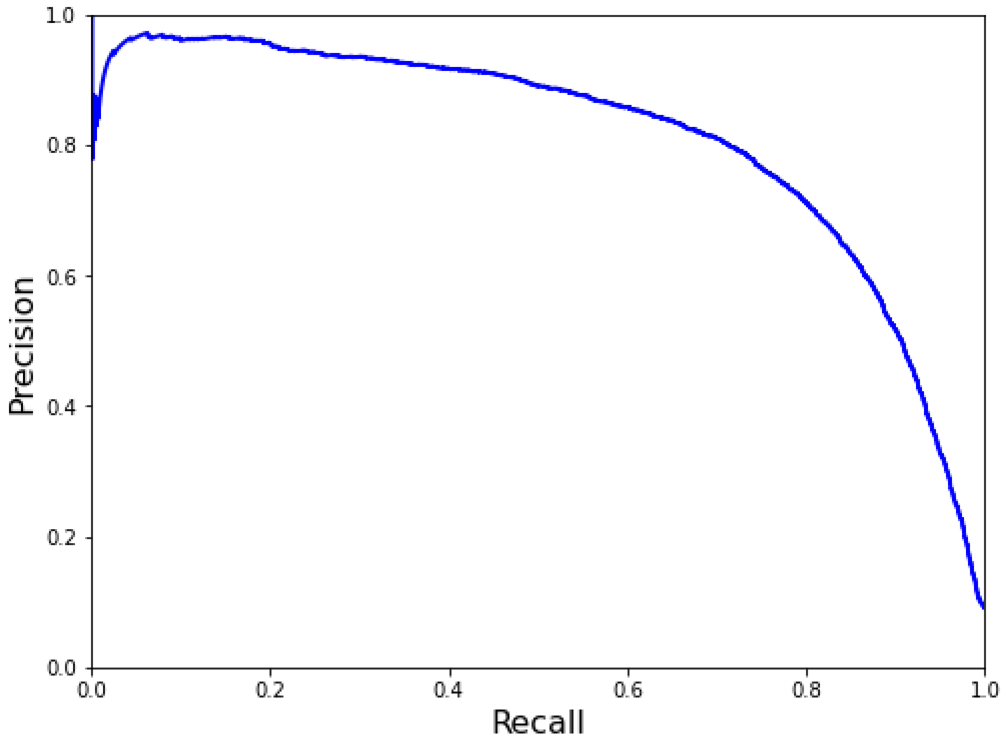
In [ ]:
```
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.show()
```



How to make predictions using threshold?

In [ ]:
```
def plot_precision_vs_recall(precisions, recalls):
    plt.plot(recalls, precisions, "b-", linewidth=2)
    plt.xlabel("Recall", fontsize=16)
    plt.ylabel("Precision", fontsize=16)
    plt.axis([0, 1, 0, 1])
```

In [ ]:
```
plt.figure(figsize=(8, 6))
plot_precision_vs_recall(precisions, recalls)
plt.show()
```



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

Out[ ]:
```
3370.0194991439557
```

In [ ]:
```
y_train_pred_90 = (y_scores >= threshold_90_precision)
```

In [ ]:
```
precision_score(y_train_5, y_train_pred_90)
```

Out[ ]: 0.9000345901072293

In [ ]:
```
recall_score(y_train_5, y_train_pred_90)
```

Out[ ]: 0.4799852425751706

However, a high-precision classifier is not very useful if its recall is too low

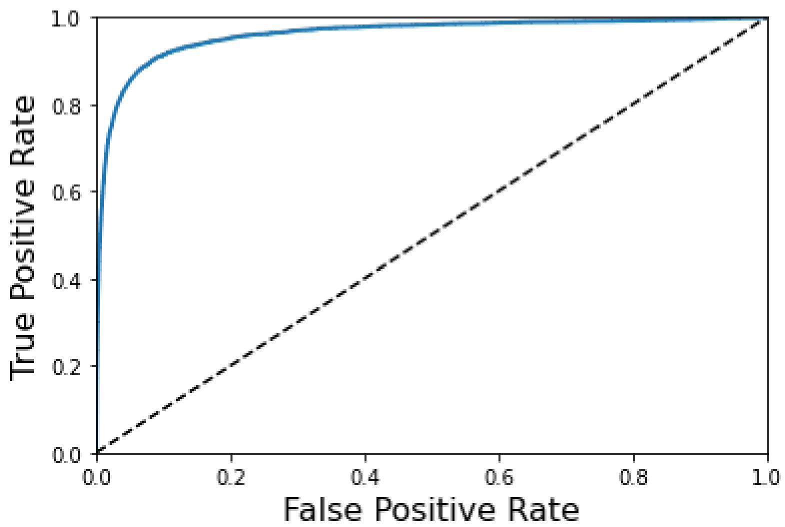
4.4 ROC Curve

ROC curve is another common tool used with binary classifiers. It is similar to precision/recall curve but instead of precision vs recall, it plots the TPR (Recall/Sensitivity) against FPR (1 - TNR = 1 - specificity)

In [ ]:
```
from sklearn.metrics import roc_curve

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

In [ ]:
```
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', fontsize=16)
    plt.ylabel('True Positive Rate', fontsize=16)
```

In [ ]:
```
plot_roc_curve(fpr, tpr)
plt.show()
```



The dotted line represents the ROC curve of a purely random classifier. A good classifier stays as far away from that line as possible (top-left corner)

One way to compare classifiers is to measure the area under the curve (AUC). A perfect classifier will have AUC=1, whereas purely random classifier will have AUC=0.5

In [ ]:
```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_train_5, y_scores)
```

Out[ ]: 0.9604938554008616

Use PR curve when the positive class is rare or when you care more about the FP than FN. Otherwise, use ROC.

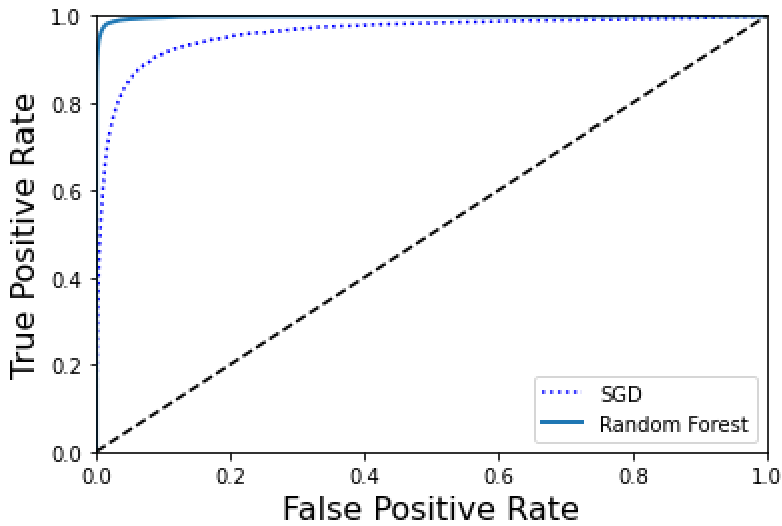
In [ ]:
```
from sklearn.ensemble import RandomForestClassifier

forest_clf = RandomForestClassifier(random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3, method="
```

In [ ]:
```
y_probas_forest
```

Out[ ]:
```
array([[0.11, 0.89],
       [0.99, 0.01],
       [0.96, 0.04],
       ...,
       [0.02, 0.98],
       [0.92, 0.08],
       [0.94, 0.06]])
```

In [ ]:
```
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)
```

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



Comparing the ROC curves, the random forest classifier is superior to the SGD classifier because its ROC curve is much closer to top-left corner and it has a greater AUC

In [ ]:
```
roc_auc_score(y_train_5, y_scores_forest)
```

Out[ ]:
```
0.9983436731328145
```

In [ ]:
```
y_train_5
```

Out[ ]:
```
array([ True, False, False, ...,  True, False, False])
```

In [ ]:
```
y_scores_forest
```

Out[ ]:
```
array([0.89, 0.01, 0.04, ..., 0.98, 0.08, 0.06])
```

In [ ]:
```
threshold = 0.5
```

```
y_scores_forest_pred = (y_scores_forest > threshold)
```

In [ ]:
```
precision_score(y_train_5, y_scores_forest_pred)
```

Out[ ]:
```
0.9905083315756169
```

In [ ]:
```
recall_score(y_train_5, y_scores_forest_pred)
```

Out[ ]:
```
0.8662608374838591
```

5. Multiclass Classification

- one-versus-rest/all (OvR): one way to create a system that can classify the digit images into 10 classes is to train 10 binary classifiers, one for each digit
- one-versus-one (OvO): train a binary classifier for every pair of digits

Main advantage of OvO is that each classifier only needs to be trained on the part of the training set for the two classes that it must distinguish

SVM scale poorly with the size of training set hence OvO is preferred because it is faster to train many classifiers on small training sets than few classifiers on large training sets. For more binary classification algorithms, OvR is preferred

Scikit-learns detects when you use binary classification algorithm for a multiclass classification task and automatically runs OvR or OvO.

In [ ]:
```
# OvO strategy used. 45 binary classifiers.
from sklearn.svm import SVC
svm_clf = SVC()
svm_clf.fit(X_train, y_train)
svm_clf.predict([some_digit])
```

Out[ ]:
```
array([5], dtype=int8)
```

It returns 10 scores per instance instead of just 1. (One score per class)

In [ ]:
```
some_digit_scores = svm_clf.decision_function([some_digit])
some_digit_scores
```

Out[ ]:
```
array([[ 1.72501977,  2.72809088,  7.2510018 ,  8.3076379 , -0.31087254,
         9.3132482 ,  1.70975103,  2.76765202,  6.23049537,  4.84771048]])
```

When a classifier is trained, it stores the list of target classes in its classes_ attribute, ordered by value.

In [ ]:
```
np.argmax(some_digit_scores)
```

Out[ ]:
```
5
```

In [ ]:
```
svm_clf.classes_
```

Out[ ]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=int8)
```

In [ ]:
```
svm_clf.classes_[np.argmax(some_digit_scores)]
```

Out[ ]:
```
5
```

If want scikit-learn to use OvO or OvR, we can use OneVsOneClassifier or OneVsRestClassifier

In [ ]:
```
from sklearn.multiclass import OneVsRestClassifier
ovr_clf = OneVsRestClassifier(SVC())
ovr_clf.fit(X_train, y_train)
```

Out[ ]:
```
OneVsRestClassifier(estimator=SVC())
```

In [ ]:
```
ovr_clf.predict([some_digit])
len(ovr_clf.estimators_)
```

Out[ ]:
```
10
```

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

Out[ ]:
```
array([3], dtype=int8)
```

In [ ]:
```
sgd_clf.decision_function([some_digit])
```

Out[ ]:
```
array([[-31893.03095419, -34419.69069632,  -9530.63950739,
          1823.73154031, -22320.14822878,  -1385.80478895,
        -26188.91070951, -16147.51323997,  -4604.35491274,
        -12050.767298  ]])
```

The classifier is confident about its prediction as almost all scores are largely negative

In [ ]:
```
cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
```

Out[ ]:
```
array([0.87365, 0.85835, 0.8689 ])
```

It gets over 84% on all test folds so its pretty good. If used a random classifier, you will get 10% accuracy. By scaling the inputs, we can get even better result.

In [ ]:
```
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')
```

Out[ ]:
```
array([0.8983, 0.891 , 0.9018])
```

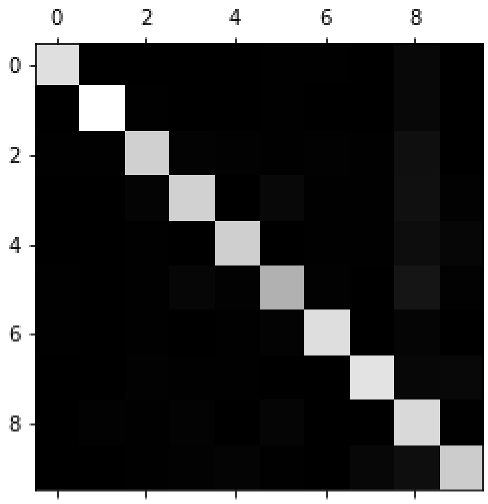
6. Error Analysis

Assuming you have found a promising model and you want to find ways to improve it, one way is to analyze the type of errors it makes.

In [ ]:
```
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[ ]:
```
array([[5577,    0,   22,    5,    8,   43,   36,    6,  225,    1],
       [   0, 6400,   37,   24,    4,   44,    4,    7,  212,   10],
       [  27,   27, 5220,   92,   73,   27,   67,   36,  378,   11],
       [  22,   17,  117, 5227,    2,  203,   27,   40,  403,   73],
       [  12,   14,   41,    9, 5182,   12,   34,   27,  347,  164],
       [  27,   15,   30,  168,   53, 4444,   75,   14,  535,   60],
       [  30,   15,   42,    3,   44,   97, 5552,    3,  131,    1],
       [  21,   10,   51,   30,   49,   12,    3, 5684,  195,  210],
       [  17,   63,   48,   86,    3,  126,   25,   10, 5429,   44],
       [  25,   18,   30,   64,  118,   36,    1,  179,  371, 5107]],
      dtype=int64)
```

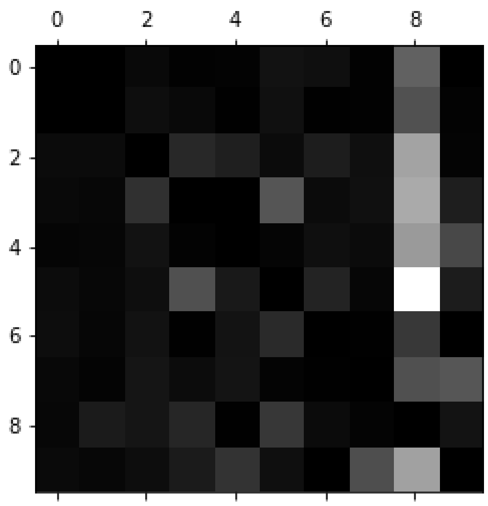
In [ ]:
```
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```



We divide each value in the confusion matrix by the number of images in the corresponding class so that you can compare error rates instead of absolute numbers. If you use absolute numbers, then those classes with many observations would look bad

In [ ]:
```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums
```

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



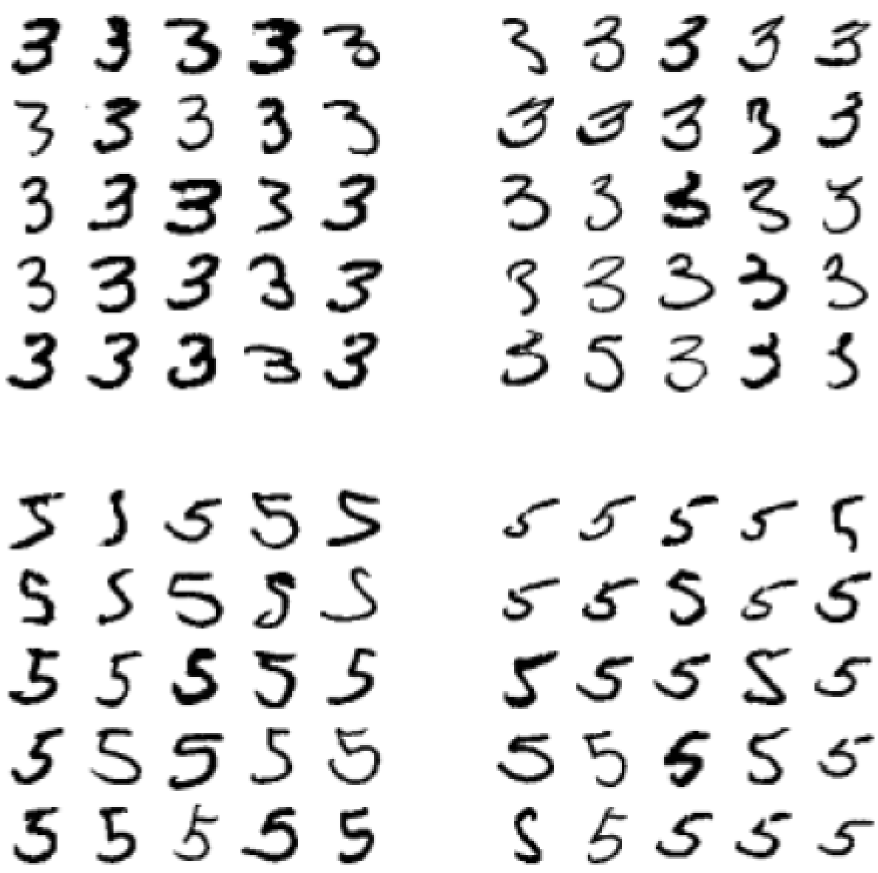
The column for class 8 is quite bright which tells us that many images get misclassified as 8s. However, the row for class 8 is not that bad, telling us that the actual 8s in general get properly classified as 8s.

From confusion matrix, efforts should be spent on reducing the false 8s. Perhaps we could try gathering more training data for digits that look like 8s (but are not) so that the classifier can learn to distinguish them for real 8s.

In [ ]:
```
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)]
```

In [ ]:
```
# EXTRA
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 = mpl.cm.binary, **options)
    plt.axis("off")
```

In [ ]:
```
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)
```



7. Multilabel Classification

The y_multilabel array contains two target labels. The first indicates whether or not the digit is large and the second indicates whether or not it is odd

In [ ]:
```
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] # array contains two target label:
```

In [ ]:
```
knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

Out[ ]:
```
KNeighborsClassifier()
```

In [ ]:
```
knn_clf.predict([some_digit]) # 5 is indeed not large and is odd
```

Out[ ]:
```
array([[False,  True]])
```

There are many ways to evaluate multilabel classifier. One way is F1 score for each individual label and then compute the average score. This assumes that all labels are equally important. If we have more pictures of Alice than of Bob or Charlie, we may want to give more weight to classifier's score on pictures of Alice.

In [ ]:
```
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")
```

Out[ ]:
```
0.976410265560605
```

8. Multioutput Classification

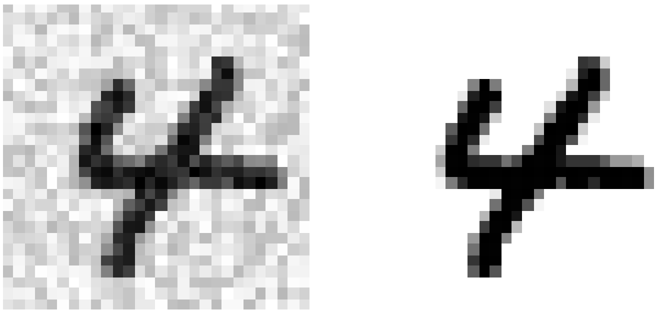
It is a generalization of multilabel classification where each label can be multiclass. An example is pixel output of image. The classifier's output is multilabel because each label forms one pixel and each label can have multiple values.

In [ ]:
```
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
```

In [ ]:
```
def plot_digit(data):
    image = data.reshape(28, 28)
    plt.imshow(image, cmap = mpl.cm.binary,
               interpolation="nearest")
    plt.axis("off")
```

In [ ]:
```
some_index = 5500
plt.subplot(121); plot_digit(X_test_mod[some_index])
```

```
plt.subplot(122); plot_digit(y_test_mod[some_index])
plt.show()
```



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