✉️

Chapter 3: Classification

Created	@Jun 14, 2020 6:46 PM
Type	Completed

MNIST

The MNIST dataset is a set of 70k small images of digits handwritten by highschool students and employees of the US Census Bureau. Each image is labeled with the digit it represents. This set has been studied so much that it is often called the "Hello World" of Machine Learning.

Scikit-Learn provides many helper functions to download popular datasets. The following code fetches the MNIST dataset:

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

Datasets loaded by Scikit-Learn generally have a similar dictionary structure including:

- A **DESCR key** describing the dataset.

- A **data key** containing an array with one row per instance and one column per feature.
- A **target key** containing an array with the labels.

Let's look at these arrays:

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

There are 70k images, and each image has 784 features. This is because each image is 28×28 pixels, and each feature simply represents one pixel's density, from 0 (white) to 1 (black).

Let's take a peek at one digit from the dataset:

```
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=mpl.cm.binary, interpolation="nearest")
plt.axis("off")
plt.show()
```

This looks like a 5, and indeed that's what the label tells us:

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

Note that the label is a string. We prefer numbers, so let's cast y to integers:

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

The following figure shows a few more images from the MNIST dataset:

Before inspecting the data more closely, we should first create a test set and set it aside. The MNIST dataset is already split into a training set (the first 60k images) and a test set (the last 10k images):

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

The training set is **already shuffled**, which is good as it guarantees that **all cross-validation folds will be similar** (you don't want one fold to be missing some digits). Moreover, some learning algorithms are sensitive to the order of the training instances, and they perform poorly if they get many similar instances in a row. Shuffling the dataset ensures that this won't happen.

Shuffling might be a bad idea in some cases—for example, if you are working on time series data (such as stock market prices or weather conditions).

Training a Binary Classifier

Let's simplify the problem for now and only try to identify one digit—for example, the number 5. This "5-detector" will be an example of a ***binary classifier***, capable of distinguishing between just two classes, 5 and not-5. Let's create some target vectors for this classification task:

```
y_train_5 = (y_train == 5)  # True for all 5s, False for all other digits
y_test_5 = (y_test == 5)
```

Now let's pick a classifier and train it. A good place to start is with a *Stochastic Gradient Descent* (SGD) classifier. This classifier has the advantage of being capable of handling very large datasets efficiently. This is in part because SGD deals with training instances independently, one at a time (which also makes SGD well suited for *online learning*).

```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train_5, y_train_5)
```

Now you can use it to detect images of the number 5:

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

Performance Measures

Evaluating a classifier is often **significantly trickier** than evaluating a regressor.

Measuring Accuracy Using Cross-Validation

Occasionally you will need more control over the cross-validation process than what Scikit-Learn provides off-the-shelf. In these cases, you can implement cross-validation yourself.

The following code does roughly the same thing as Scikit-Learn's cross_val_score() function, and prints the same result:

```
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_5, 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))
```

The StratifiedKFold class performs stratified sampling to produce folds that contain a representative ratio of each class. At each iteration the code creates a clone of the classifier, trains that clone on the training folds, and makes predictions on the test fold. Then it counts the number of correct predictions and outputs the ratio of correct predictions.

Now let's use the cross_val_score() function to evaluate the SGDClassifier model using K-fold cross-validation, with three folds. Remember that K-fold cross-validation means splitting the training set into K-folds (in this case, three), then making predictions and evaluating them on each fold using a model trained on the remaining folds:

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([ 0.9502 ,  0.96565,  0.96495])
```

Wow! Above *95% accuracy* (ratio of correct predictions) on all cross-validation folds? This looks amazing, but before getting too excited let's look at a very dumb classifier that just classifies every single image in the "not-5" class:

```
from sklearn.base import BaseEstimator

class Never5Classifier(BaseEstimator):
  def fit(slef, X, y=None):
    pass
  def predict(self, X):
    return np.zeros((len(X), 1), dtype=bool)  # False (0) for all predictions
```

Let's take a look at its accuracy:

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

```
array([ 0.909  ,  0.90715,  0.9128 ])
```

It has over 90% accuracy! This is simply because only about 10% of the images are 5s, so if you always guess that an image is *not* a 5, you will be right about 90% of the time.

This demonstrates why accuracy is generally not the preferred performance measure for classifiers, especially when you are dealing with *skewed datasets* (i.e., when some classes are much more frequent than others).

Confusion Matrix

A much better way to evaluate the performance of a classifier is to look at the *confusion matrix*. The general idea is to count the number of times instances of class A are classified as class B. For example, to know the number of times the classifier confused images of 5s with 3s, you would look in the 5th row and 3rd column of the confusion matrix.

To compute the confusion matrix, you first need to have a set of predictions, so they can be compared to actual targets. You could make those predictions on the test set, but let's keep it untouched now (remember to only use the test set at the very end of the project). Instead, you can use the cross_val_predict() function:

```
from sklearn.model_selection import cross_val_predict

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

Just like the cross_val_score() function, cross_val_predict() performs K-fold cross-validation, but instead of returning the evaluation scores, it returns the predictions made on each test fold. This means that you get a clean prediction for each instance in the training set ("clean" meaning that the prediction is made by a model that never saw the data during training).

Now you are ready to get the confusion matrix using the confusion_matrix() function:

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_train_5, y_train_pred)
array([[53272,  1307],
       [ 1077,  4344]])
```

Each row in a confusion matrix represents an *actual class*, while each column represents a *predicted class*. The first row of this matrix considers non-5 images (the *negative class*): 53,272 of them were correctly classified as non-5s (they are called *true negatives*), while the remaining 1,307 were wrongly classified as 5s (*false positives*). The second row considers the images of 5s (the *positive class*): 1,077 were wrongly classified as non-5s (*false negatives*), while the remaining 4,344 were correctly classified as 5s (*true positives)*.

A **perfect classifier** would have only true positives and true negatives, so its confusion matrix would have nonzero values only on its **main diagonal**:

```
>>> y_train_perfect_predictions = y_train_5  # pretend we reached perfection
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)
array([[5479,    0],
       [   0, 5421]])
```

The confusion matrix gives you a lot of information, but sometimes you may prefer a more concise metric. An interesting one to look at is the accuracy of the positive predictions; this is called the ***precision*** of the classifier.

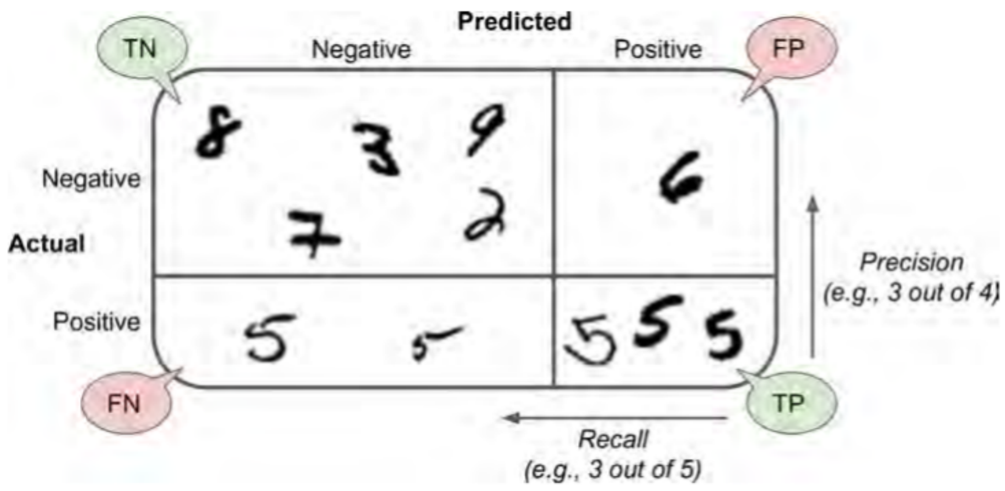
$$\text{precision} = \frac{TP}{TP + FP}$$

TP is the number of true positives, and FP is the number of false positives.

A trivial way to have perfect precision is to make one single positive prediction and ensure it is correct (precision = 1/1 = 100%). This would not be very useful since the classifier would ignore all but one positive instance. So precision is typically along with another metric named ***recall***, also called *sensitivity* or *true positive rate* (TPR): this is the ratio of positive instances that are correctly detected by the classifier.

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

The following is an ilustrated confusion matrix:



Precision and Recall

Scikit-Learn provides several functions to compute classifier metrics, including precision and recall:

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred) # == 4344 / (4344 + 1307)
0.76871350203503808
>>> recall_score(y_train_5, y_train_pred) # == 4344 / (4344 + 1077)
0.80132816823464303
```

When the classifier claims an image represents a 5, it is correct 77% of the time. Moreover, it only detects 80% of the 5s.

It is often convenient to combine precision and recall into a single metric called the ***F1 score***, in particular if you need a simple way to compare two classifiers. The F1 score is the *harmonic mean* of precision and recall. Whereas the regular mean treats all values equally, the harmonic mean gives much more weight to low values. As a result, the classifier will only get a high F1 score if both recall and precision are high.

$$F1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

To compute the F1 score, simply call the f1_score() function:

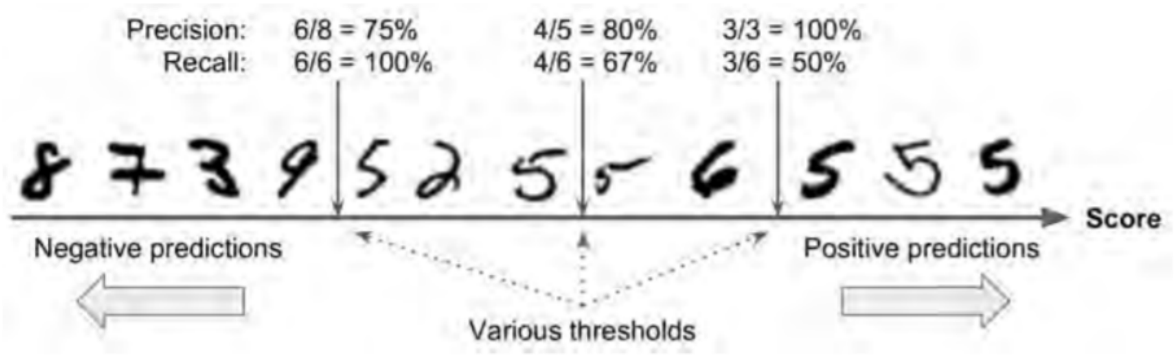
```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.78468200861626413
```

The F1 score favors classifiers that have similar precision and recall. This is not always what you want: in some contexts you moslty care about precision, and in other contexts you really care about recall. For example, if you trained a classifier to detect videos that are safe for kids, you would probably prefer a classifier that rejects many good videos (low recall) but keeps only safe ones (high precision), rather than a classifier that has a much higher recall but lets a few really bad videos show up in your product (in such cases, you may even want to add a human pipeline to check the classifier's video selection). On the other hand, suppose you train a classifier to detect shoplifters on surveillance images: it is probably fine if you classifier has only 30% precision as long as it has 99% recall (sure, the security guards will get a few false alerts, but almost all shoplifters will get caught).

Unfortunately, you can't have it both ways: increasing precision reduces recall, and vice versa. This is called the *precision/recall tradeoff*.

Precision/Recall Tradeoff

To understand this tradeoff, let's look at hwo the SGDClassifier makes its classification decisions. For each instance, it computes a score based on a *decision function*, and if that score is greater than a threshold, it assigns the instance to the positive class, or else it assigns it to the negative class.



Scikit-Learn does not let you set the threshold directly, but it does give you access to the decision scores. Instead of calling the classifier's predict() method, you can call its decision_function() method, which returns a score for each instance, and then make predictions based on those scores using any threshold you want:

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([ 161855.7486126])
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
array([ True], dtype=bool)
```

The SGDClassifier uses a threshold equal to 0, so the previous code returns the same result. Let's now raise the threshold:

```
>>> threshold = 200000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False], dtype=bool)
```

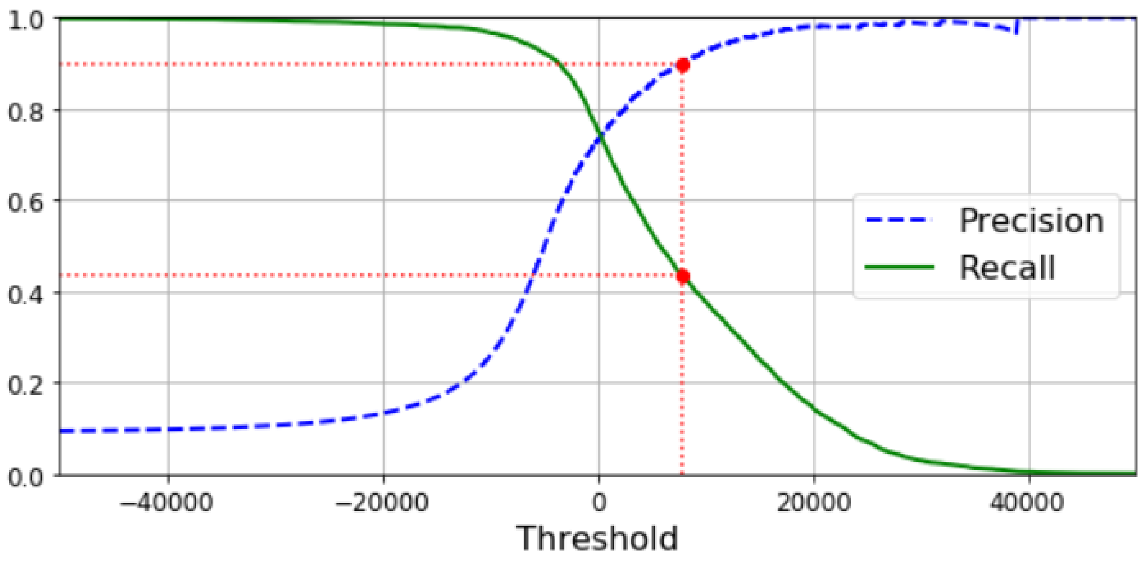
This confirms that **raising the threshold decreases recall**.

Now how do you decide which threshold to use? For this you will first need to get the scores of all instances in the training set using the cross_val_predict() function again, but this time specifying that you want to return decision scores instead of predictions:

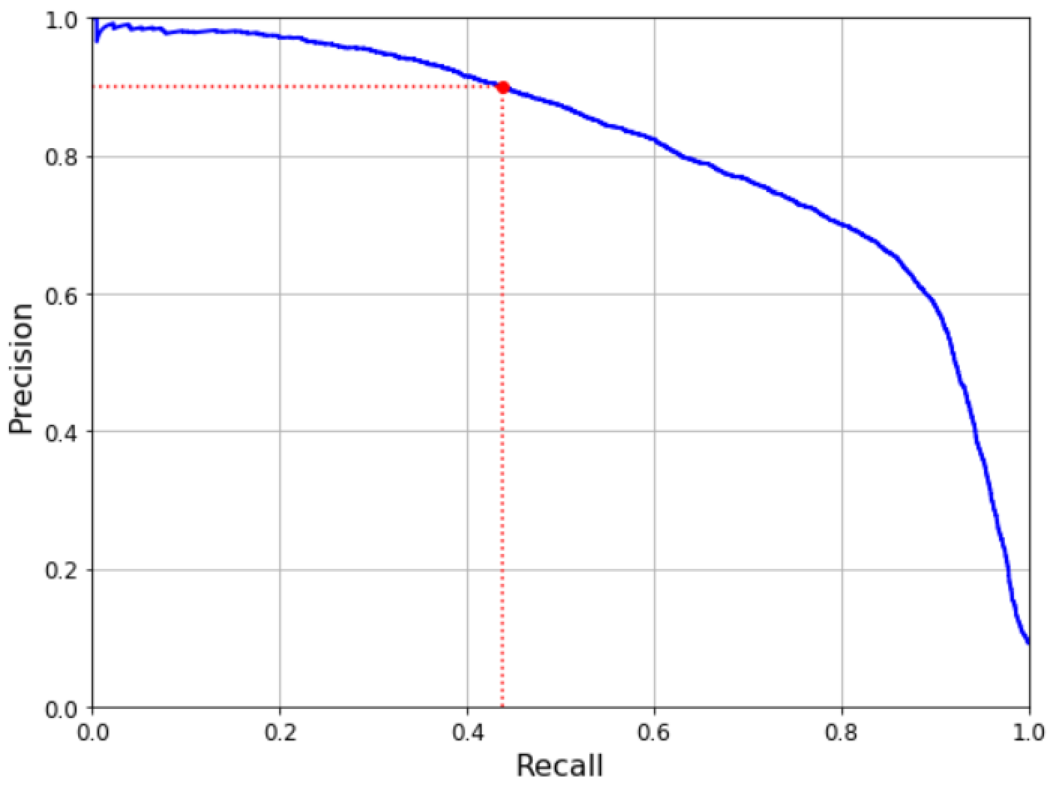
```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

Now with these scores you can compute precision and recall for all possible thresholds using the precision_recall_curve() function:

```
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, recall[:-1], "g-", label="Recall")
  plt.xlabel("Threshold")
  plt.legend(loc="center left")
  plt.ylim([0, 1])

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



Now you can simply select the threshold value that gives you the best precision/recall tradeoff for your task. Another way to select a good precision/recall tradeoff is to plot precision directly against recall as shown in the following figure:



Let's suppose you decide to aim for 90% precision. You look up the first plot and find that you need to use a threshold of about 70,000. To make predicitons (on the training set for now), instead of calling the classifier's predict() method, you can just run this code:

```
y_train_pred_90 = (y_scores > 70000)
```

Let's check these predictions' precision and recall:

```
>>> precision_score(y_train_5, y_train_pred_90)
0.86592051164915464
>>> recall_score(y_train_5, y_train_pred_90)
0.69931748916232346
```

But remember, **a high-precision classifier is not very useful if its recall is too low**.

The ROC Curve

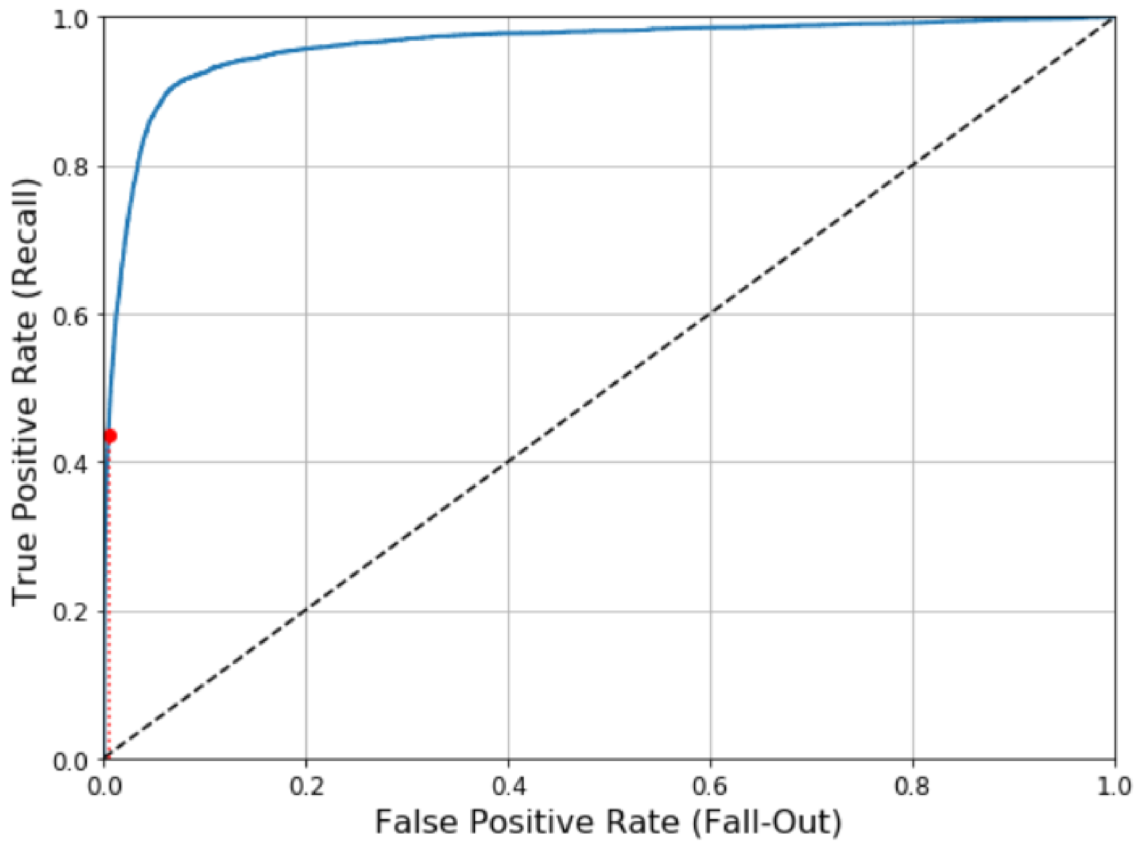
The *receiver operating characteristic* (ROC) curve is another common tools used with binary classifiers. It is very similar to the precision/recall curve, but instead of plotting precision versus recall, the ROC curve plots the *true positive rate* (another name for recall) against the *false positive rate*. The FPR is the ratio of negative instances that are incorrectly classified as positive. The TNR is also called ***specificity***. Hence the ROC curve plots *sensitivity* (recall) versus 1 - *specificity*.

To plot the ROC curve, you first need to compute the TPR and the FPR for various threshold values, using the roc_curve() function:

```
from sklearn.metrics import roc_curve

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

Then, you can plot the FPR against the TPR:

```
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()
```



There is a tradeoff to this: the higher the recall, the more false positives the classifier produces. The dotted line represents the ROC curve of a purely random classifier; a good classifier **stays away from that line as possible** (toward the top left corner).

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

```
>>> from sklearn.metrics import roc_auc_score
>>> roc_auc_score(y_train_5, y_scores)
0.9645641323158
```

Since the ROC curve is so similar to the precision/recall curve, you should prefer the P/R curve **whenever the positive class is rare** or when you **care more about the false positives** than the false negatives, and the ROC curve otherwise.

Let's train a RandomForestClassifier and compare its ROC curve and ROC AUC score to the SGDClassifier. First, you need to get scores for each instance in the training set. But due to the way it works, the RandomForestClassifier class doesn't have a decision_function() method. Instead it has a **predict_proba()**

method. Scikit-Learn classifiers generally have one or the other. The predict_proba() method returns an array containing a row per instance and a column per class, each containing the probability that the given instance belongs to the given class:

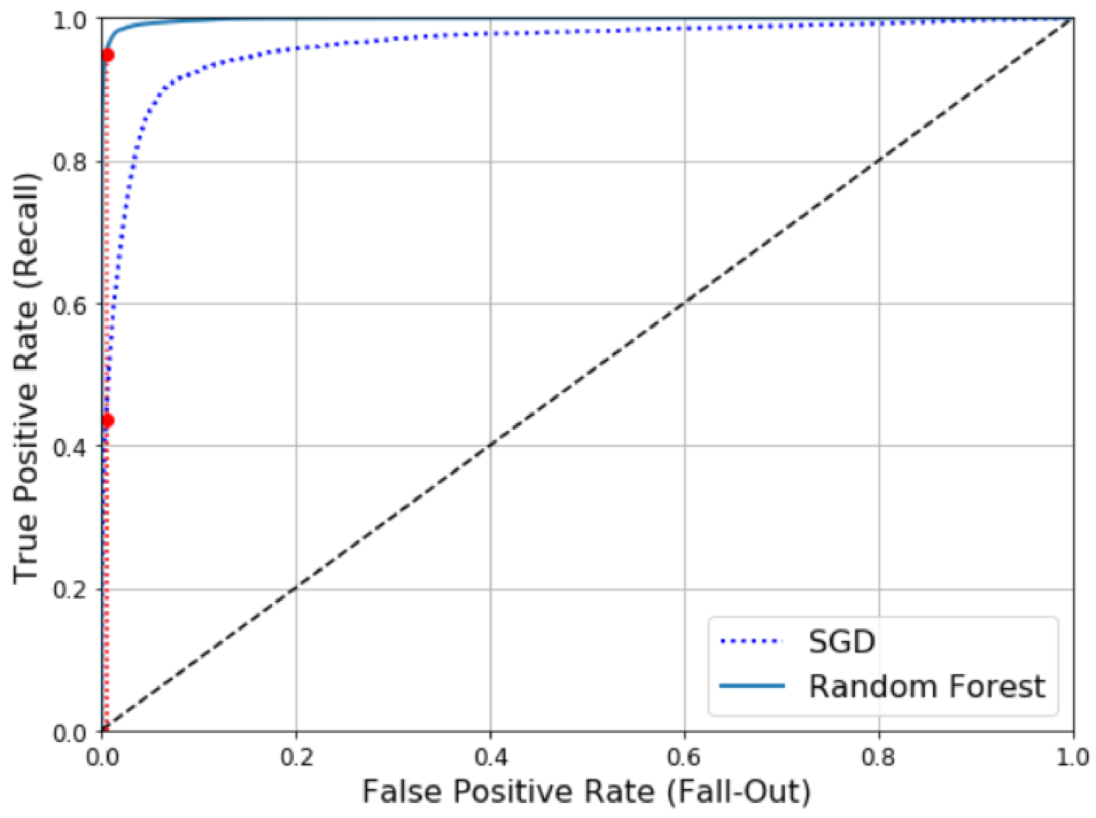
```
from sklearn.ensemble import RandomForestClassifier

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

But to plot a ROC curve, you need scores, not probabilities. A simple solution is to use the positive class's probability as the score:

```
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)
```

Now you can plot the ROC curve:

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



As you can see, the RandomForestClassifier's ROC curve looks much better than the SGDClassifier's: it comes much closer to the top-left corner. As a result, its ROC AUC score is also significantly better:

```
>>> roc_auc_score(y_train_5, y_scores_forest)
0.993124565616261
```

Comparing Binary Classifiers

$$\begin{tikzcd}
\text{Train binary classifiers} \arrow[d] \\
\text{Choose appropiate metric for the task} \arrow[d] \\
\text{Cross-validation} \arrow[d] \\
\text{Precision/Recall} \arrow[d] \\
\text{Compare models using ROC/ROC AUC}
\end{tikzcd}$$

Multiclass Classification

Whereas binary classifiers distinguish between two classes, *multiclass classifiers* (also called *multinomial classifiers*) can distinguish between more than two classes.

Some algorithms (such as Random Forest classifiers or Bayes classifiers) are capable of handling multiple classes directly. Others (such as Support Vector Machine classifiers or Linear classifiers) are strictly binary classifiers. However, there are **various strategies that you can use to perform multiclass classification using multiple binary classifiers.**

Imagine we want to have a model that can classify hand-written digits from 0 to 9.

- **OvA:** With the One Versus All approach we could train 10 binary classifiers, one for each digit. Then, when we want to classify an image, we just have to get the decision score from each classifier for that image and collect the class whose classifier outputs the highest score.
- **OvO:** Another strategy is the One Versus One approach. The method consists of training a binary classifier for every pair of digits: one to distinguish 0s and 1s, another to distinguish 0s and 2s, another for 1s and 2s, and so on. If there are N classes, you need to train $N \times \frac{(N-1)}{2}$ classifiers. The 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.

Some algorithms (such as Support Vector Machine classifiers) scale poorly with the size of the training set, so for these algorithms OvO is preferred since it is faster to train many classifiers on small training sets than training few classifiers on large training sets. **For most binary classification algorithms**, however, **OvA is preferred.**

Scikit-Learn detects when you try to use a binary classification algorithm for a multi-class classification task, and it automatically runs OvA (except for SVM classifiers for which it uses OvO).

```
>>> sgd_clf.fit(X_train, y_train)  # y_train, not y_train_5
>>> sgd_clf.predict([some_digit])
array([5], dtype=uint8)
```

Under the hood, Scikit-Learn trained 10 binary classifiers, got their decision scores for the image, and selected the class with the highest score. To see that this is indeed the case, you can call the decision_function() method. Instead of returning one score per instance, it now returns 10 scores, one per class:

```
>>> some_digit_scores = sgd_clf.decision_function([some_digit])
>>> some_digit_scores
array([[-311112.62954431, -363517.28355739, ..., -536774.6396122]], length=10)
```

The highest score is indeed the one corresponding to class 5:

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

💡 When a classifier is trained, it stores the list of target classes in its classes_ attribute, ordered by value. In this case, the index of each class in the classes_ array conveniently matches the class itself, but in general you won't be so lucky.

If you want to force Scikit-Learn to use one-versus-one or one-versus-all, you can use the OneVsOneClassifier or OneVsRestClassifier classes. Simply create an instance and pass a binary classifier to its constructor.

```
>>> 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], dtype=uint8)
>>> len(ovo_clf.estimators_)
45
```

Training a RandomForestClassifier is just as easy:

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

This time Scikit-Learn did not have to run OvA or OvO because Random Forest classifiers can directly classifiy instances into multiple classes. You can call predict_proba() to get the list of probabilities that the classifier assigned to each instance for each class:

```
>>> forest_clf.predict_proba([some_digit])
array([[ 0.1,  0. ,  0. , 0.1,  0. , 0.8,  0. ,  0.,  0.,  0., ]])
```

In this case, the classifier is very confident that the image represents a the 5th class (number 5).

Now of course we want to evaluate these classifiers. As usual, we want to use cross-validation. Let's evaluate the SGDClassifier's accuracy using the cross_val_score() function:

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([0.8465651, 0.84563132, 0.86656131])
```

It gets over 84% on all test folds. If you used a random classifier, you would get 10% accuracy, so this is not such a bad score, but we can still do much better. For example, simply **scaling the inputs** increases accuracy above 90%:

```
>>> 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([0.9132353,  0.90312646,  0.9066365])
```

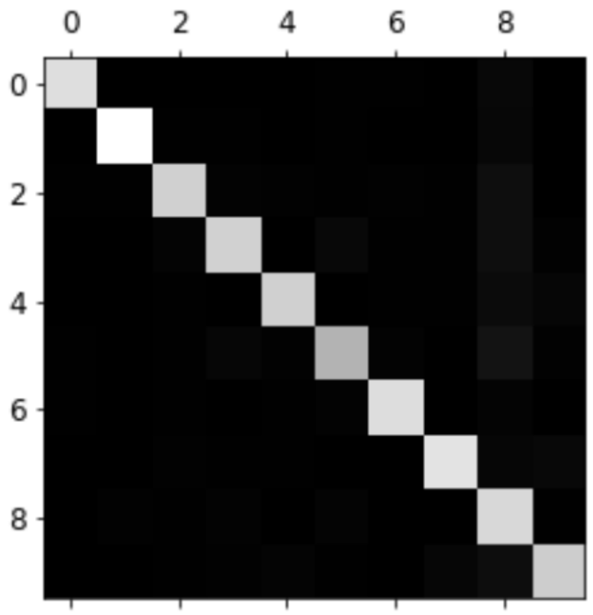
Error Analysis

From this point onwards, it is assumed that a suitable model has been found using various fine-tuning methods (exploring data preparation options, trying out multiple models, shortlisting the best ones and fine-tunning their hyperparameters using GridSearchCV, and automating as much as possible).

In order to improve the model we must analyze the types of errors it makes.

First, we must look at the **confusion matrix**. We need to make predictions using the cross_val_predict() function, then call the confusion_matrix() function.

```
y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
conf_mx = confusion_matrix(y_train, y_train_pred)
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```



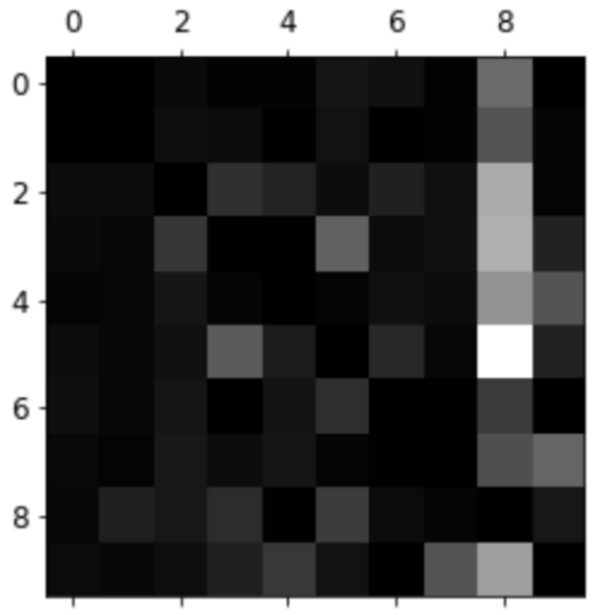
This confusion matrix looks fairly good, since most images are on the main diagonal, which means that they were classified correctly.

Let's now focus the plot on the errors. First, you need to divide each value in the confusion matrix by the number of images in the corresponding class, so you can compare error rates instead of absolute number of errors (which would make abundant classes look unfairly bad).

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

Now let's fill the diagonal with zeros to keep only the errrors, and let's plot the result:

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



Anylizing the confusion matrix can give you insights on ways to improve your classifier. For example, you could try to gather more training data for digits with high error rates. Or you could engineer new features that would help the classifier - for example, writing an algorithm to count the number of closed loops. Or you could preprocess images (e.g., using Scikit-Image, Pillow or OpenCV) to make some patterns stand out more, such as closed loops.

Multilabel Classification

In some cases you may want your classifier to output multiple classes for each instance.

```
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)
```

This code creates a y_multilabel array containing two target labels for each digit image: the first indicates whether or not the digit is large (7, 8 or 9) and the second indicates whether or not it is odd. The next lines create a KNeighborsClassifier instance (which supports multilabel classification, but not all classifiers do) and we train it using the multiple targets array.

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

There are many ways to evaluate a multilabel classifier, and selecting the right metric really depends on the nature of your project. For example, one approach is to measure the F1 score for each individual label, then simply compute de average score. This code computes the average F1 score across all labels:

```
>>> 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")
0.97765612364
```

This assumes that **all labels are equally important**, which may not be the case. One simple solution is to give each label a weight equal to its *support* (i.e., the number of instances with that target label). To do this, simply set average="weighted" in the preceding code.

Multioutput Classification

Multioutput classification is simply a generalization of multilabel classification where each label can be multiclass (i.e., it can have more than two possible values).

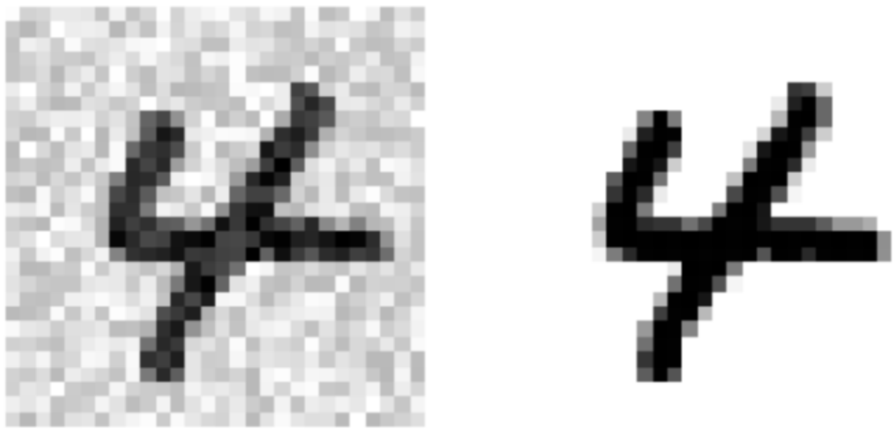
To illustrate this, let's build a system that removes noise from images. It will take as input a noisy digit image, and it will output a clean digit image, represented as an array of pixel intensities, just like the MNIST images. Notice that the classifier's output is multilabel (one label per pixel) and each label can have multiple values (pixel intensity ranges from 0 to 255). It is thus an example of multioutput classification.

💡 The line between classification and regression is sometimes blurry, such as in this example. Arguably, **predicting pixel intensity is more akin to regression than to classification.** Moreover, multioutput systems are not limited to classification tasks; you could even have a system that outputs multiple labels per instance, including both class labels and value labels.

Let's start by creating the training and test sets by taking the MNIST images and adding noise to their pixel intensities using NumPy's randint() function. The target images will be the original images:

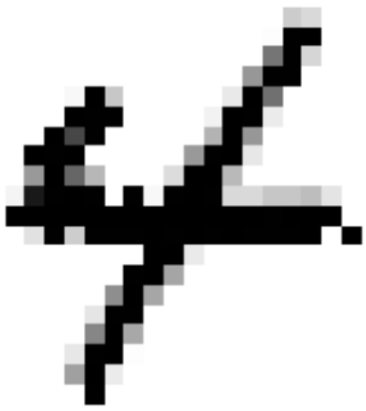
```
noise = np.random.randint(0, 100, (len(X_train)), 784)
X_train_mod = X_train + noise
noise = np.random.randint(0, 100, (len(X_test)), 784)
X_test_mod = X_test + noise
y_train_mod = X_train
y_test_mod = X_test
```

Let's peek at the data:



On the left is the noisy input image, and on the right is the clean target image. Now let's train the classifier and make it clean this image:

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



Looks close enough to the target!