Classification

MNIST

- **MNIST** is a dataset of 70,000 small images of digits handwritten by high school students and employees of the US Census Bureau. Each image is labeled with the digit it represents.
- **Scikit-Learn** provides many helper functions to download popular datasets. **MNIST** is one of them. The following code fetches the MNIST dataset from **OpenML.org**:

```
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', as_frame=False)
```

- The **sklearn.datasets** package contains mostly three types of functions:
- **fetch_*** functions such as **fetch_openml()** to download real-life datasets,
- **load_*** functions to load small toy datasets bundled with Scikit-Learn
- **make_*** functions to generate fake datasets, useful for tests. Generated datasets are usually returned as an (X, y) tuple containing the input data and the targets, both as NumPy arrays.
- Other datasets are returned as **sklearn.utils.Bunch** objects, which are dictionaries whose entries can also be accessed as attributes. They generally contain the following entries:
  - **"DESCR"** : A description of the dataset
  - **"data" :** The input data, usually as a 2D NumPy array
  - **"target" :** The labels, usually as a 1D NumPy array
- The **fetch_openml()** function is a bit unusual since by default it returns the inputs as a Pandas **DataFrame** and the **labels** as a Pandas Series (unless the dataset is **sparse**).
- Since **MNIST** dataset contains images, and **DataFrames** aren't ideal for that, so it's preferable to set **as_frame=False** to get the data as NumPy arrays instead.
- There are **70,000 images**, and each image has **784 features**. This is because each image is **28 × 28 pixels**, and each feature simply represents one pixel's intensity, from **0** (white) to **255** (black).
- grab an instance's feature vector, reshape it to a **28 × 28** array, and display it using Matplotlib's **imshow()** function. We use **cmap="binary"** to get a grayscale color map where 0 is white and 255 is black:

```
import matplotlib.pyplot as plt

def plot_digit(image_data):
    image = image_data.reshape(28, 28)
    plt.imshow(image, cmap="binary")
    plt.axis("off")

some_digit = X[0]
plot_digit(some_digit)
plt.show()
```

- Note that the **label** is a **string**. Most ML algorithms expect numbers, so let's cast y to integers:

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

- The **MNIST** dataset returned by **fetch_openml()** is actually already split into a **training set** (the first **60,000** images) and a **test set** (the last **10,000 images**):

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

- The **training set** is already **shuffled** for us, this guarantees that all **cross-validation** folds will be similar.
- 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.

Training a Binary Classifier

- Create a classifier that can identify one digit, First we'll create the 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')
```

- Using **stochastic gradient descent (SGD, or stochastic GD) classifier**, using Scikit-Learn's **SGDClassifier** class.
- This classifier is 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, y_train_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

- A good way to evaluate a model is to use **cross-validation**, Use the **cross_val_score()** function to evaluate our **SGDClassifier** model, using **k-fold cross-validation** with three folds.

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.95035, 0.96035, 0.9604 ])
```

- Create a **dummy classifier** that just classifies every single image in the most frequent class, which in this case is the negative class (i.e., non 5):

```
from sklearn.dummy import DummyClassifier

dummy_clf = DummyClassifier()
dummy_clf.fit(X_train, y_train_5)
print(any(dummy_clf.predict(X_train)))  # prints False: no 5s detected
```

```
>>> cross_val_score(dummy_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.90965, 0.90965, 0.90965])
```

- 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 (CM)**.
- The general idea of a confusion matrix is to count the number of times instances of class **A** are classified as class **B**, for all **A/B** pairs.

- To compute the **confusion matrix**, you first need to have a set of predictions so that they can be compared to the actual targets.
- 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.
- You can get the confusion matrix using the **confusion_matrix()** function.

```
>>> from sklearn.metrics import confusion_matrix
>>> cm = confusion_matrix(y_train_5, y_train_pred)
>>> cm
array([[53892,   687],
       [ 1891,  3530]])
```

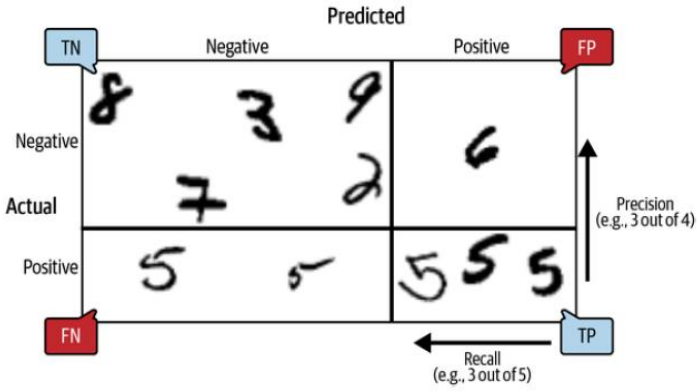
TN	FP
FN	TP

- A perfect classifier would only have **true positives** and **true negatives**.
- **Precision** is the accuracy of the positive predictions of the classifier.

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

- A trivial way to have perfect **precision** is to create a classifier that always makes **negative** predictions, except for one single positive prediction on the instance it's most confident about.
- **precision** is typically used along with another metric named **recall**, also called **sensitivity** or the **true positive rate** (**TPR**): this is the ratio of positive instances that are correctly detected by the classifier

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



Precision and recall

- Implementing Precision and Recall in scikit-learn:

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred)  # == 3530 / (687 + 3530)
0.8370879772350012
>>> recall_score(y_train_5, y_train_pred)  # == 3530 / (1891 + 3530)
0.6511713705958311
```

- It is often convenient to combine **precision** and **recall** into a single metric called the **F_1 score**, especially when you need a single metric to **compare** two classifiers. The **F_1 score** is the harmonic mean of **precision** and **recall**
- 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 **F_1 score** if both **recall,** and **precision** are high.
- To compute the **F_1 score**, simply call the **f1_score()** function:

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.7325171197343846
```

- The **F_1 score** favors classifiers that have similar **precision** and **recall**.
- This is not always what you want: in some contexts, you mostly care about **precision**, and in other contexts you really care about **recall**.

Precision/Recall Trade-off

- Increasing **precision** reduces **recall**, and vice versa. This is called the **precision/recall trade-off**.

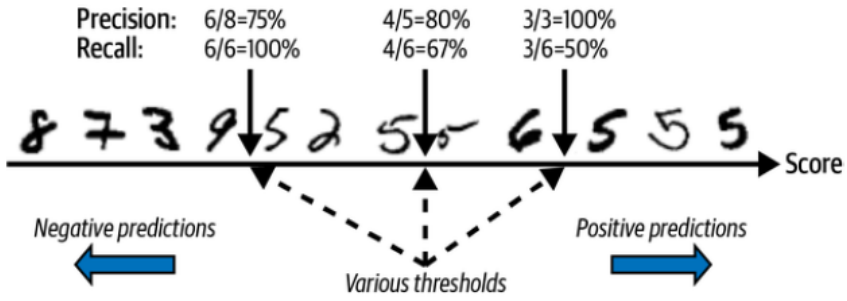


Figure 3-4. The precision/recall trade-off: images are ranked by their classifier score, and those above the chosen decision threshold are considered positive; the higher the threshold, the lower the recall, but (in general) the higher the precision

- Scikit-Learn does not let you set the threshold directly, but it does give you access to the **decision scores** that it uses to make predictions.
- you can call **decision_function()** method, which returns a **score** for each instance, and then use any threshold you want to make predictions based on those scores:

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([2164.22030239])
>>> threshold = 0

>>> y_some_digit_pred = (y_scores > threshold)
array([ True])
```

- The **SGDClassifier** uses a **threshold** equal to **0,** Let's raise the threshold:

```
>>> threshold = 3000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False])
```

- This confirms that raising the **threshold** decreases **recall**.
- How do you decide which threshold to use?
  - First, use the **cross_val_predict()** function to get the **scores** of all instances in the training set, but this time specify 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")
```

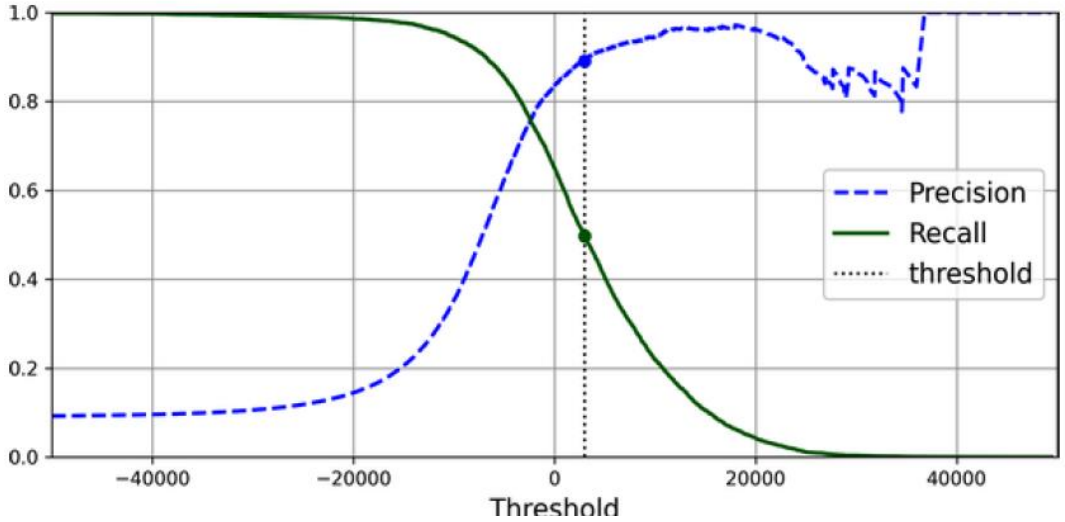
- With these scores, use the **precision_recall_curve()** function to compute precision and recall for all possible **thresholds**

```
from sklearn.metrics import precision_recall_curve

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

- use Matplotlib to plot precision and recall as functions of the threshold value

```
plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
plt.vlines(threshold, 0, 1.0, "k", "dotted", label="threshold")
```



- The **precision** curve is bumpier than the **recall** curve, The reason is that precision may sometimes go down when you raise the **threshold** On the other hand, **recall** can only go down when the **threshold** is increased, which explains why its curve looks smooth.
- Another way to select a good **precision/recall trade-off** is to plot **precision** directly against **recall**.

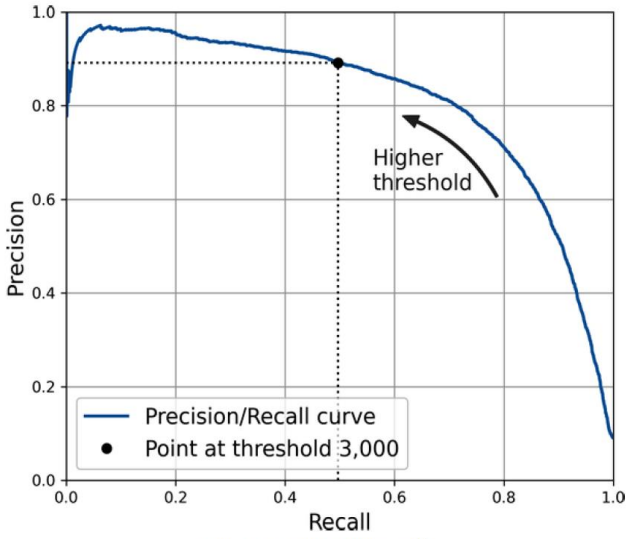


Figure 3-6. Precision versus recall

- You can see that precision really starts to fall sharply at around 80% recall. You will probably want to select a precision/recall trade-off just before that drop. But of course, the choice depends on your project.
- you can search for the lowest threshold that gives you at least 90% precision. For this, you can use the NumPy array's **argmax()** method. This returns the first index of the maximum value, which in this case means the first **True** value:

```
>>> idx_for_90_precision = (precisions >= 0.90).argmax()
>>> threshold_for_90_precision = thresholds[idx_for_90_precision]


>>> threshold_for_90_precision
3370.0194991439557
```

- To make predictions (on the training set for now), instead of calling the classifier's **predict()** method, you can run this code:

```
y_train_pred_90 = (y_scores >= threshold_for_90_precision)

>>> precision_score(y_train_5, y_train_pred_90)
0.9000345901072293
>>> recall_at_90_precision = recall_score(y_train_5, y_train_pred_90)
>>> recall_at_90_precision
0.4799852425751706
```

- a high-**precision** classifier is not very useful if its recall is too low! For many applications, 48% **recall** wouldn't be great at all.
- If someone says, "Let's reach 99% **precision**", you should ask, "At what **recall**?"

The ROC Curve

- The **receiver operating characteristic (ROC) curve** is another common tool 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** (**FPR**).
- The **FPR** (also called the **fall-out**) is the **ratio of negative instances** that are incorrectly classified as positive. It is equal to **1 – the true negative rate** (**TNR**), which is the ratio of negative instances that are correctly classified as negative.
- The **TNR** is also called **specificity**. Hence, the ROC curve plots **sensitivity** (**recall**) versus **1 – specificity**.
- To plot the **ROC curve**, you first use the **roc_curve()** function to compute the **TPR** and **FPR** for various threshold values:

```
from sklearn.metrics import roc_curve

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

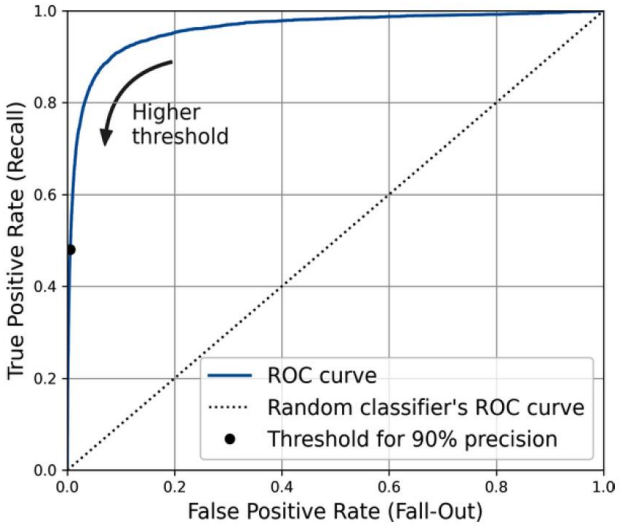


Figure 3-7. A ROC curve plotting the false positive rate against the true positive rate for all possible thresholds; the black circle highlights the chosen ratio (at 90% precision and 48% recall)

- Once again there is a trade-off: the higher the **recall** (**TPR**), the more false positives (**FPR**) the classifier produces.
- The dotted line represents the **ROC curve** of a purely random classifier; a good classifier stays as far 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**.
- Scikit-Learn provides a function to estimate the **ROC AUC**:

```
>>> from sklearn.metrics import roc_auc_score
>>> roc_auc_score(y_train_5, y_scores)
0.9604938554008616
```

- **TIP**: Since the **ROC curve** is so similar to the **precision/recall (PR) curve**, you should prefer the **PR** curve whenever the **positive** class is **rare** or when you care more about the **false positives** than the **false negatives**. Otherwise, use the **ROC curve**.
- create a **RandomForestClassifier**, whose **PR curve** and **F_1 score** we can compare to those of the **SGDClassifier**:

```
from sklearn.ensemble import RandomForestClassifier

forest_clf = RandomForestClassifier(random_state=42)
```

- The **precision_recall_curve()** function expects **labels** and **scores** for each instance, so we need to train the random forest classifier and make it assign a score to each instance. But the RandomForestClassifier class does not have a **decision_function()** method, due to the way it works
- It has a **predict_proba()** method that returns class probabilities for each instance, and we can just use the probability of the positive class as a **score**.
- We can call the **cross_val_predict()** function to train the **RandomForestClassifier** using **cross-validation** and make it predict class probabilities for every image as follows:

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

```
>>> y_probas_forest[:2]
array([[0.11, 0.89],    [0.99, 0.01]])
```

- The second column contains the estimated probabilities for the positive class, so let's pass them to the **precision_recall_curve()** function:

```
y_scores_forest = y_probas_forest[:, 1]
precisions_forest, recalls_forest, thresholds_forest = precision_recall_curve(
    y_train_5, y_scores_forest)
```

- the **RandomForestClassifier's PR curve** looks much better than the **SGDClassifier's**: it comes much closer to the top-right corner. Its **F_1 score** and **ROC AUC score** are also significantly better:

```
>>> y_train_pred_forest = y_probas_forest[:, 1] >= 0.5  # positive proba ≥ 50%
>>> f1_score(y_train_5, y_pred_forest)
0.9242275142688446
>>> roc_auc_score(y_train_5, y_scores_forest)
0.9983436731328145
```

- Try measuring the precision and recall scores: you should find about **99.1% precision** and **86.6% recall**.

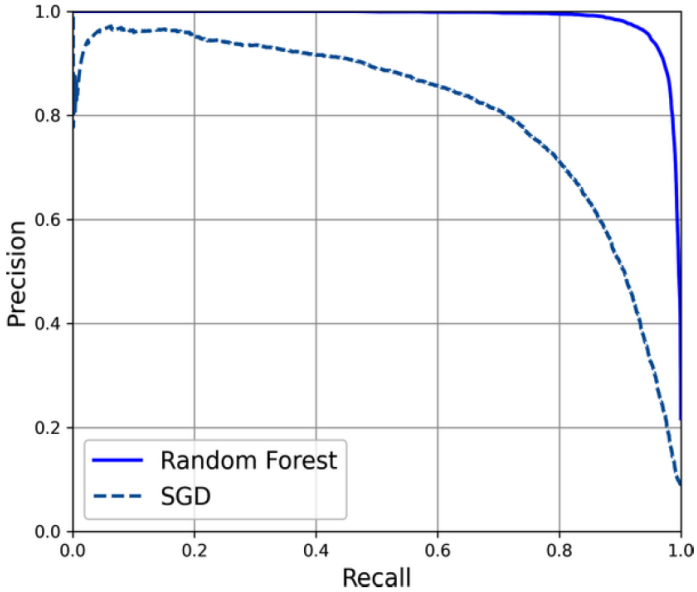


Figure 3-8. Comparing PR curves: the random forest classifier is superior to the SGD classifier because its PR curve is much closer to the top-right corner, and it has a greater AUC

- The **sklearn.calibration** package contains tools to **calibrate** the estimated probabilities and make them much closer to actual probabilities.

Multiclass Classification

- **Multiclass classifiers** (also called **multinomial** classifiers) can distinguish between more than two classes.
- Some Scikit-Learn classifiers (e.g., **LogisticRegression**, **RandomForestClassifier**, and **GaussianNB**) are capable of handling **multiple** classes natively. Others are strictly **binary** classifiers (e.g., **SGDClassifier** and **SVC**).
- However, there are various strategies that you can use to perform **multiclass** classification with multiple **binary** classifiers.
- One way to create a system that can classify the digit images into 10 classes(from 0 to 9) is to train **10 binary classifiers**, one for each digit (a 0-detector,a 1-detector, a 2-detector, and so on). Then when you want to classify an image, you get the decision score from each classifier for that image, and you select the class whose classifier outputs the **highest score**. This is called the **one-versus-the-rest (OvR) strategy**, or sometimes **one-versus-all** (**OvA**).
- Another strategy is to train 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. This is called the **one-versus-one (OvO) strategy**. If there are **N** classes, you need to train **N × (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 containing the two classes that it must distinguish. Some algorithms (such as **support vector machine** classifiers) scale poorly with the size of the training set. For these algorithms **OvO** is preferred because it is faster to train many classifiers on small training sets than to train few classifiers on large training sets.
- For most **binary** classification algorithms, however, **OvR** is preferred.
- Scikit-Learn detects when you try to use a **binary** classification algorithm for a multiclass classification task, and it automatically runs **OvR** or **OvO** depending on the algorithm.
- Create a **support vector machine** classifier using the **sklearn.svm.SVC** class:

```
from sklearn.svm import SVC

svm_clf = SVC(random_state=42)
svm_clf.fit(X_train[:2000], y_train[:2000])  # y_train, not y_train_5
```

- Scikit-Learn used the **OvO** strategy and trained **45 binary** classifiers. This code actually made **45** predictions—one per pair of classes—and it selected the class that won the most duels.
- If you call the **decision_function()** method, you will see that it returns **10** scores per instance: one per class.

```
>>> some_digit_scores = svm_clf.decision_function([some_digit])
>>> some_digit_scores.round(2)
array([[ 3.79,  0.73,  6.06,  8.3 , -0.29,  9.3 ,  1.75,  2.77,  7.21,
         4.82]])
```

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

```
>>> svm_clf.classes_
array(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'], dtype=object)
>>> svm_clf.classes_[class_id]
'5'
```

- If you want to force Scikit-Learn to use **one-versus-one** or **one-versus-the rest**, you can use the **OneVsOneClassifier** or **OneVsRestClassifier** classes.

```
from sklearn.multiclass import OneVsRestClassifier

ovr_clf = OneVsRestClassifier(SVC(random_state=42))
ovr_clf.fit(X_train[:2000], y_train[:2000])
```

- make a prediction, and check the number of trained classifiers:

```
>>> ovr_clf.predict([some_digit])
array(['5'], dtype='<U1')
>>> len(ovr_clf.estimators_)
10
```

Error Analysis

- we will assume that you have found a promising model, and you want to find ways to improve it. One way to do this is to analyze the types of errors it makes.
- First, look at the **confusion matrix**. you first need to make predictions using the **cross_val_predict()** function; then you can pass the labels and predictions to the **confusion_matrix()** function
- A colored diagram of the confusion matrix is much easier to analyze. To plot such a diagram, use the **ConfusionMatrixDisplay.from_predictions()** function like this:

```
from sklearn.metrics import ConfusionMatrixDisplay

y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred)
plt.show()
```

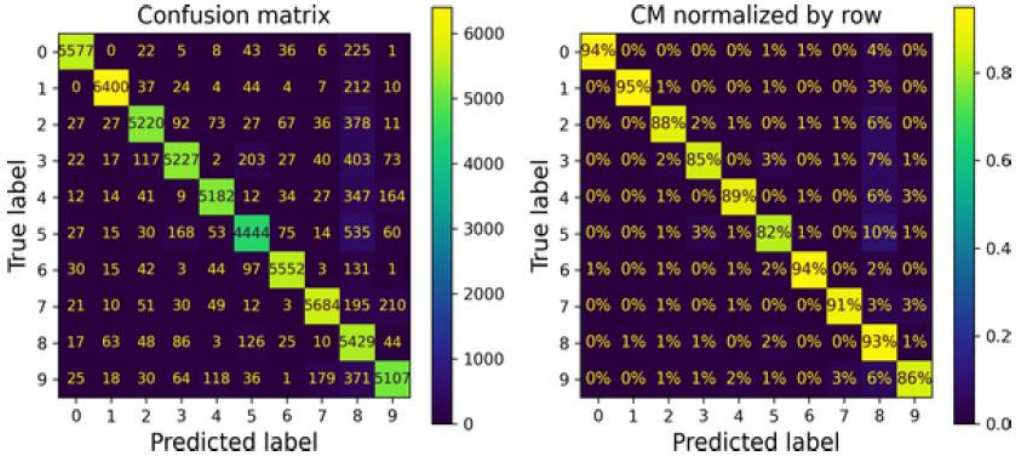


Figure 3-9. Confusion matrix (left) and the same CM normalized by row (right)

- Notice that the cell on the diagonal in **row #5** and **column #5** looks slightly **darker** than the other digits. This could be because the model made more errors on 5s, or because there are fewer 5s in the dataset than the other digits.
- That's why it's important to **normalize** the confusion matrix by dividing each value by the total number of images in the corresponding (**true**) class (i.e., divide by the **row's sum**). This can be done simply by setting **normalize="true"**. We can also specify the **values_format=".0%"** argument to show percentages with no decimals.

```
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred,
                                        normalize="true", values_format=".0%")
plt.show()
```

- Now we can easily see that only 82% of the images of 5s were classified correctly.
- The most common error the model made with images of **5s** was to misclassify them as **8s**: this happened for **10%** of all **5s**. But only **2%** of **8s** got misclassified as **5s**

- Also, many digits have been misclassified as **8s**, but this is not immediately obvious from this diagram. If you want to make the errors stand out more, you can try putting zero weight on the correct predictions.

```
sample_weight = (y_train_pred != y_train)
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred,
                                        sample_weight=sample_weight,
                                        normalize="true", values_format=".0%")
plt.show()
```

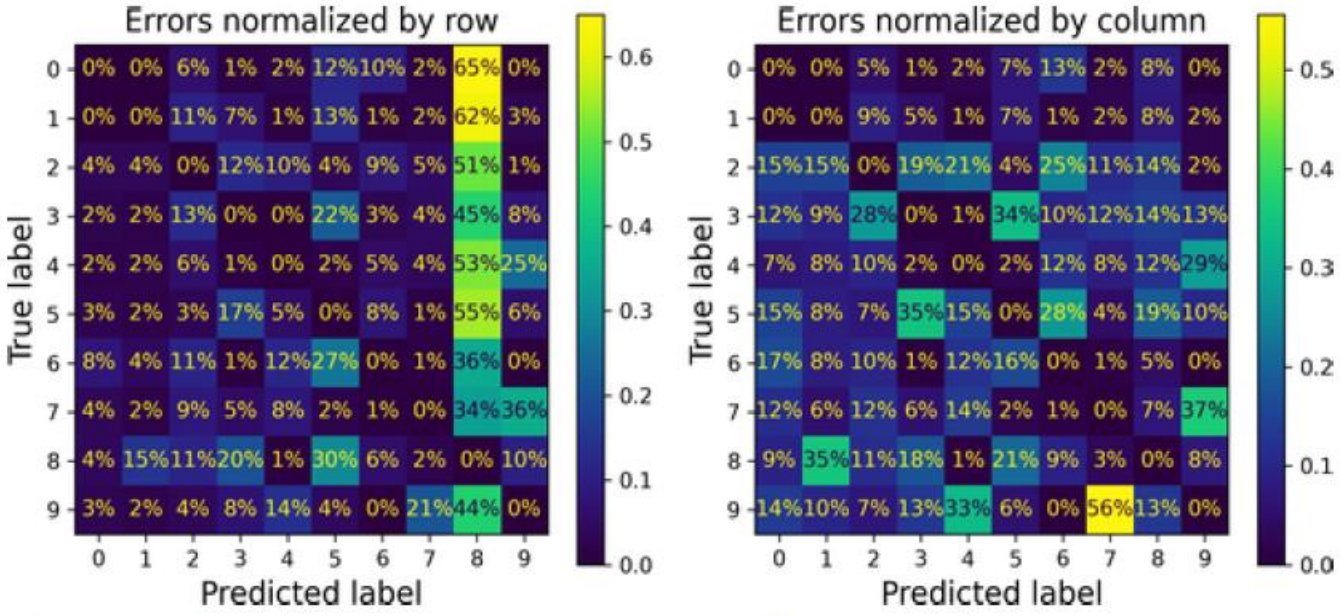


Figure 3-10. Confusion matrix with errors only, normalized by row (left) and by column (right)

- Now you can see much more clearly the kinds of errors the classifier makes. The column for **class 8** is now really bright, which confirms that many images got misclassified as 8s.
- But be careful how you interpret the percentages in this diagram: remember that we've excluded the correct predictions. For example, the 36% in row #7, column #9 does not mean that 36% of all images of 7s were misclassified as 9s. It means that 36% of the errors the model made on images of 7s were misclassifications as 9s.
- It is also possible to normalize the **confusion matrix by column** rather than by **row**: if you set **normalize="pred"**, For example, you can see that 56% of misclassified 7s are actually 9s.
- Analyzing the **confusion matrix** often gives you insights into ways to improve your classifier.
- Looking at these plots, it seems that your efforts should be spent on reducing the **false 8s**. For example, you could try to **gather** more training data for digits that look like **8s** (but are not) so that the classifier can learn to distinguish them from real **8s**. Or you could **engineer** new features that would help the classifier—for example, writing an algorithm to count the number of closed loops (e.g., 8 has two, 6 has one, 5 has none). Or you could **preprocess** the images (e.g., using **Scikit-Image**, **Pillow**, or **OpenCV**) to make some patterns, such as closed loops, stand out more.

- Analyzing individual errors can also be a good way to gain insights into what your classifier is doing and why it is failing. For example, let's plot examples of 3s and 5s in a confusion matrix style :

```
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)]
[...]  # plot all images in X_aa, X_ab, X_ba, X_bb in a confusion matrix style
```

- As you can see, some of the digits that the classifier gets wrong (i.e., in the **bottom-left** and **top-right** blocks).
- Recall that we used a simple **SGDClassifier**, which is just a **linear** model: all it does is assign a weight per class to each pixel, and when it sees a new image it just sums up the weighted pixel intensities to get a score for each class. Since **3s** and **5s** differ by only a few pixels, this model will easily confuse them.
- The main difference between **3s** and **5s** is the position of the small line that joins the top line to the bottom arc. If you draw a 3 with the junction slightly shifted to the left, the classifier might classify it as a 5, and vice versa.

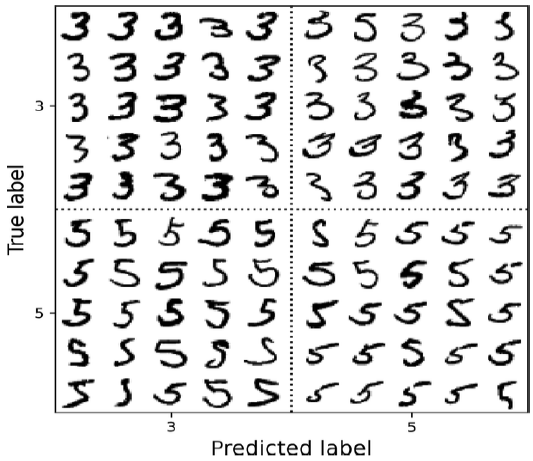


Figure 3-11. Some images of 3s and 5s organized like a confusion matrix

- In other words, this classifier is quite sensitive to image shifting and rotation. One way to reduce the **3/5** confusion is to **preprocess** the images to ensure that they are well **centered** and not too **rotated**. However, this may not be easy since it requires predicting the correct **rotation** of each image.
- A much simpler approach consists of **augmenting** the training set with slightly **shifted** and **rotated** variants of the training images. This will force the model to learn to be more tolerant to such variations. This is called **data augmentation**

Multilabel Classification

- In some cases, you may want your classifier to output **multiple classes** for each instance.
- Such a classification system that outputs multiple binary tags is called a **multilabel classification system**.

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier

y_train_large = (y_train >= '7')
y_train_odd = (y_train.astype('int8') % 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**.
- Then the code creates a **KNeighborsClassifier** instance, which supports **multilabel classification** (not all classifiers do) and trains this model 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 your project. One approach is to measure the **F_1 score** for each individual label (or any other binary classifier metric discussed earlier), then simply compute the average score.
- The following code computes the average **F_1 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")
0.976410265560605
```

- This approach assumes that all **labels** are **equally** important, which may not be the case.
- If you want to give more weight to the classifier's score; One simple option 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"** when calling the **f1_score()** function.☐
- If you wish to use a **classifier** that does not natively support **multilabel classification**, such as **SVC**, one possible strategy is to train one model per label. However, this strategy may have a hard time capturing the **dependencies** between the labels. For example, a large digit (7, 8, or 9) is twice more likely to be odd than even, but the classifier for the "odd" label does not know what the classifier for the "large" label predicted.
- To solve this issue, the models can be organized in a **chain**: when a model makes a prediction, it uses the input features plus all the predictions of the models that come before it in the chain.
- The good news is that Scikit-Learn has a class called **ChainClassifier** that does just that! By default, it will use the true labels for training, feeding each model the appropriate labels depending on their position in the chain. But if you set the **cv** hyperparameter, it will use **cross-validation** to get "clean" (out-of-sample) predictions from each trained model for every instance in the training set, and these predictions will then be used to train all the models later in the chain.
- Here's an example showing how to create and train a **ChainClassifier** using the **cross-validation** strategy.

```
from sklearn.multioutput import ClassifierChain

chain_clf = ClassifierChain(SVC(), cv=3, random_state=42)
chain_clf.fit(X_train[:2000], y_multilabel[:2000])
```

```
>>> chain_clf.predict([some_digit])
array([[0., 1.]])
```

Multioutput Classification

- **multioutput–multiclass** classification (or just **multioutput** classification). It is a generalization of **multilabel** classification where each **label** can be **multiclass** (i.e., it can have more than two possible values).
- let's build a system that removes noise from images. It will take as input a noisy digit image, and it will (hopefully) 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). This is thus an example of a **multioutput classification** system.
- Let's start by creating the **training** and **test sets** by taking the **MNIST** images and adding noise to their pixel intensities with NumPy's **randint()** function. The target images will be the original images:

```
np.random.seed(42)  # to make this code example reproducible
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 train the classifier and make it clean up this image

```
knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[0]])
plot_digit(clean_digit)
plt.show()
```

Figure 3-12. A noisy image (left) and the target clean image (right)