




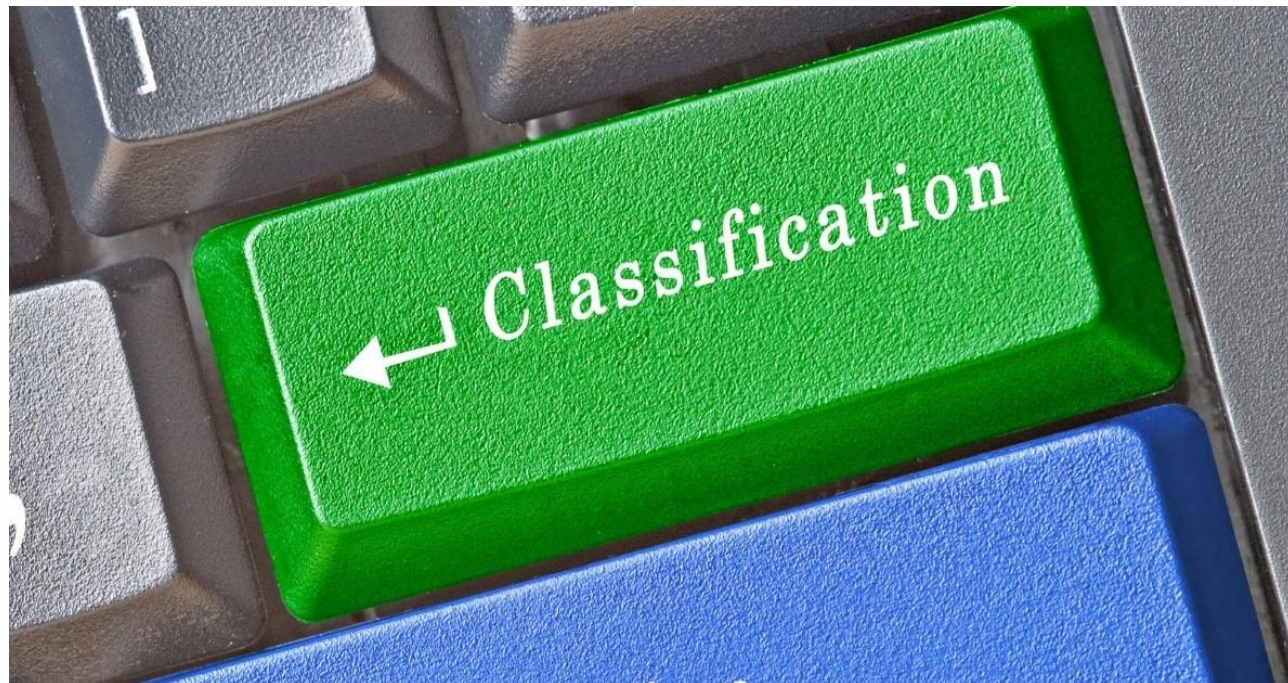
# AI in Biomedical Data

Dr. M.B. Khodabakhshi

Amir Hossein Fouladi

Alireza Javadi

 [github.com/mbkhodabakhshi/AI\\_in\\_BiomedicalData](https://github.com/mbkhodabakhshi/AI_in_BiomedicalData)



# یادگیری ماشین در زیست پزشکی

## Chapter 3. Classification

دکتر محمدباقر خدا بخشی

[mb.khodabakhshi@gmail.com](mailto:mb.khodabakhshi@gmail.com)

# Classification

As a supervised learning task, now we will turn our attention to *classification* systems.

In this chapter we will be using the **MNIST dataset**, which is a set 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.**

```
>>> 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'])

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

There are 70,000 images, and each image has 784 features. This is because each image is  $28 \times 28$  pixels, and each feature simply represents one pixel's intensity, from 0 (white) to 255 (black).

Let's take a peek at one digit from the dataset. All you need to do is grab an instance's feature vector, reshape it to a  $28 \times 28$  array, and display it using Matplotlib's `imshow()` function:

```
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="binary")
plt.axis("off")
plt.show()
```



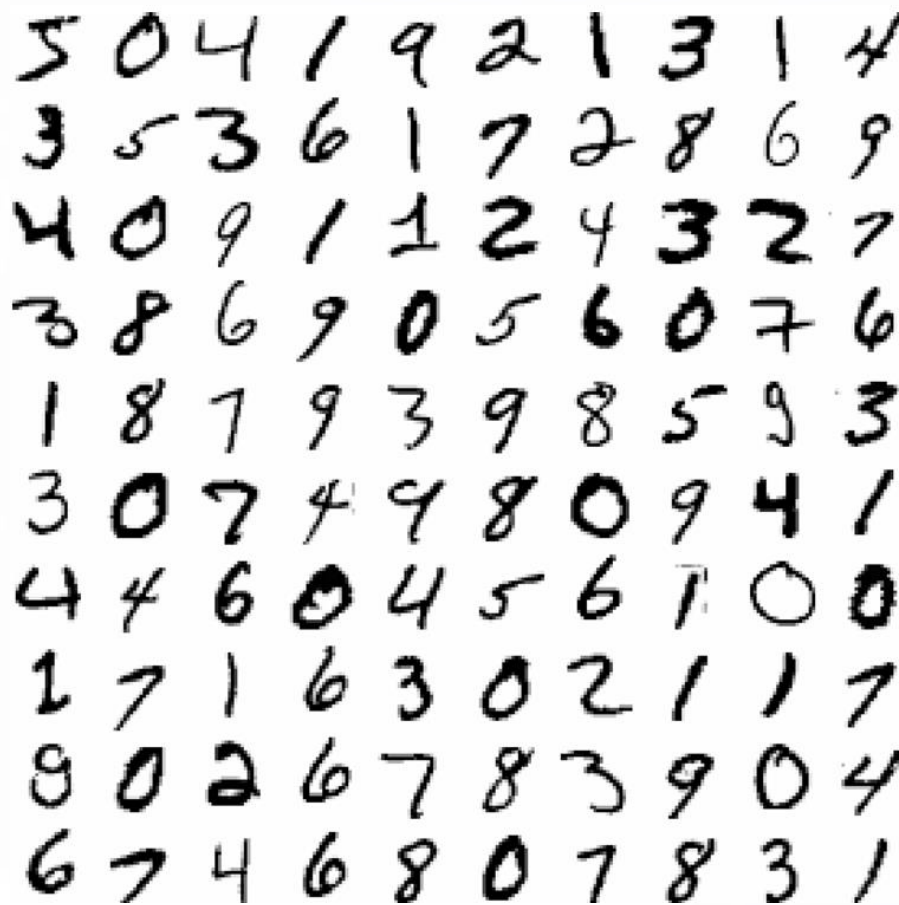
```
>>> y[0]
'5'
```



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

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

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



The MNIST dataset 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, which is good because **this 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 may be a bad idea in some contexts—for example, if you are working on time series data (such as stock market prices or weather conditions).

# Training a Binary Classifier

“5-detector” will be an example of a **binary classifier**, capable of distinguishing between just two classes, **5** and **not-5**. Let’s 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)
```

A good place to start is with a *Stochastic Gradient Descent (SGD)* classifier, using Scikit-Learn’s *SGDClassifier* class. 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**), as we will see later.

```
from sklearn.linear_model import SGDClassifier
```

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

# Measuring Accuracy Using Cross-Validation

Let's use the `cross_val_score()` function to evaluate our *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([0.96355, 0.93795, 0.95615])
```

Confusion Matrix

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**. Beats Nostradamus 😊.

This demonstrates why **accuracy is generally not the preferred performance measure for classifiers, especially when you are dealing with skewed datasets**.



# 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.

To compute the confusion matrix, you first need to have *a set of predictions so that they can be compared to the actual targets*.

```
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).

```
>>> from sklearn.metrics import confusion_matrix  
>>> confusion_matrix(y_train_5, y_train_pred)  
array([[53057, 1522],  
       [ 1325, 4096]])
```

```
>>> from sklearn.metrics import confusion_matrix  
>>> confusion_matrix(y_train_5, y_train_pred)  
array([[53057, 1522],  
       [ 1325, 4096]])
```

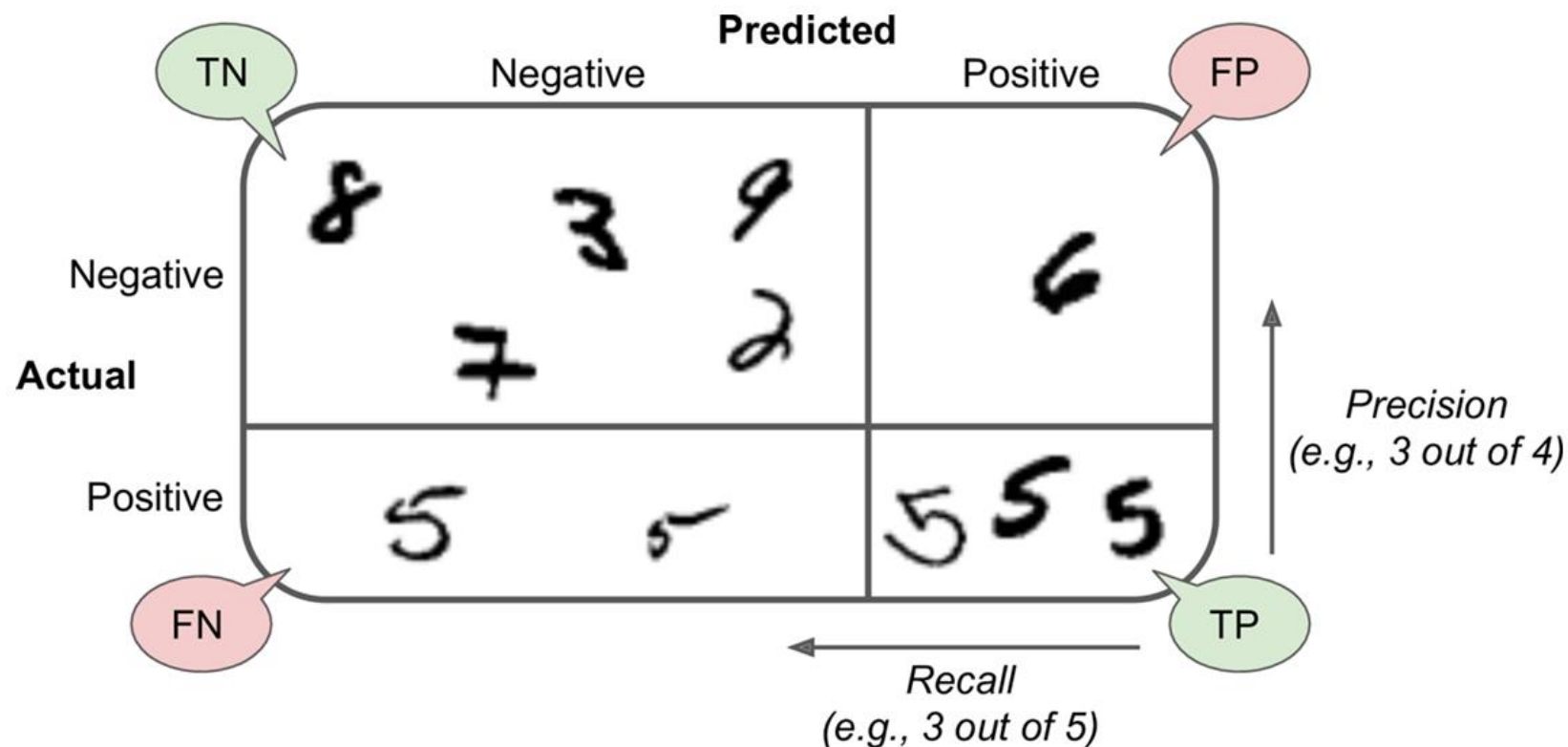
53,057 of them were correctly classified as non-5s (they are called **true negatives**), while the remaining 1,522 were wrongly classified as 5s (**false positives**). The second row considers the images of 5s (the positive class): 1,325 were wrongly classified as non-5s (**false negatives**), while the remaining 4,096 were correctly classified as 5s (**true positives**).

		Actual	
		Positive	Negative
Predicted	Positive	<b>True Positive</b>	<b>False Positive</b>
	Negative	<b>False Negative</b>	<b>True Negative</b>

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

**Precision** is typically used along with another metric named **recall**, also called sensitivity or the **true positive rate (TPR)**:

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



# Precision, Recall and F1 score

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1522)
0.7290850836596654
>>> recall_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1325)
0.7555801512636044
```

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.

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

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.7420962043663375
```

## Precision/Recall trade off

- 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.
- On the other hand, suppose you train a classifier to detect shoplifters in surveillance images: it is probably fine if your classifier has only 30% precision as long as it has 99% recall.



# The ROC Curve

The *receiver operating characteristic* (ROC) curve is another common tool used with **binary classifiers**. The ROC curve plots the *true positive rate* (another name for recall) against the *false positive rate* (FPR). The FPR is the ratio of negative instances that are incorrectly classified as positive.

**By varying the classification threshold and calculating TPR and FPR for each threshold, we can plot the ROC curve.**

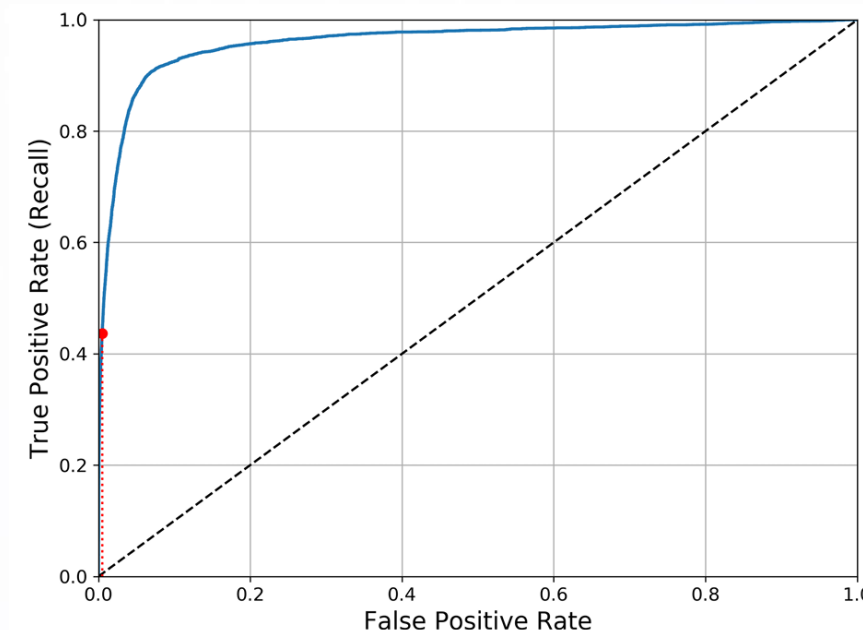
It is equal to  $1 - \text{the true negative rate}$  (TNR), which is the ratio of **negative instances that are correctly classified as negative**. The TNR is also called *specificity*. Hence, the ROC curve plots *sensitivity* (recall) versus  $1 - \text{specificity}$ .

```
from sklearn.metrics import roc_curve
```

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

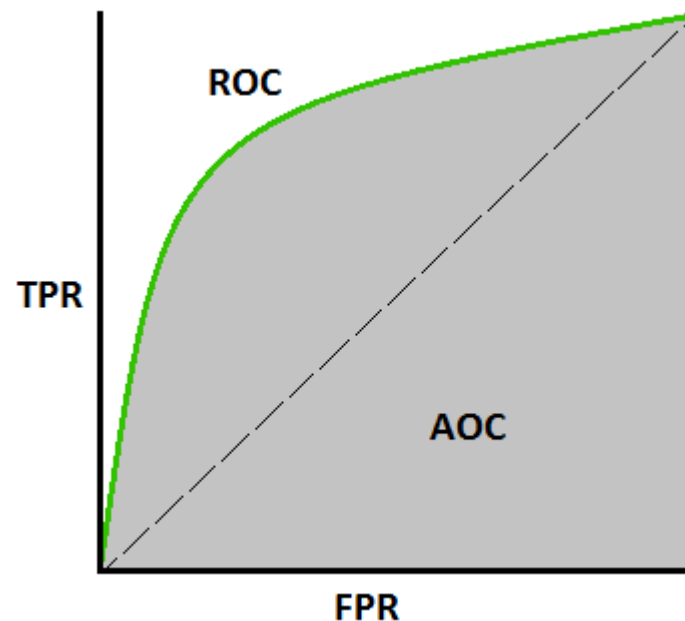
```
def plot_roc_curve(fpr, tpr, label=None):  
    plt.plot(fpr, tpr, linewidth=2, label=label)  
    plt.plot([0, 1], [0, 1], 'k--') # Dashed diagonal  
    [...] # Add axis labels and grid
```

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



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 compute the ROC AUC:

```
>>> from sklearn.metrics import roc_auc_score  
>>> roc_auc_score(y_train_5, y_scores)  
0.9611778893101814
```





Since the ROC curve is so similar to the precision/recall (PR) curve, you may wonder how to decide which one to use. As a rule of thumb, 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. For example, looking at the previous ROC curve (and the ROC AUC score), you may think that the classifier is really good. But this is mostly because there are few positives (5s) compared to the negatives (non-5s). In contrast, the PR curve makes it clear that the classifier has room for improvement (the curve could be closer to the top-left corner).



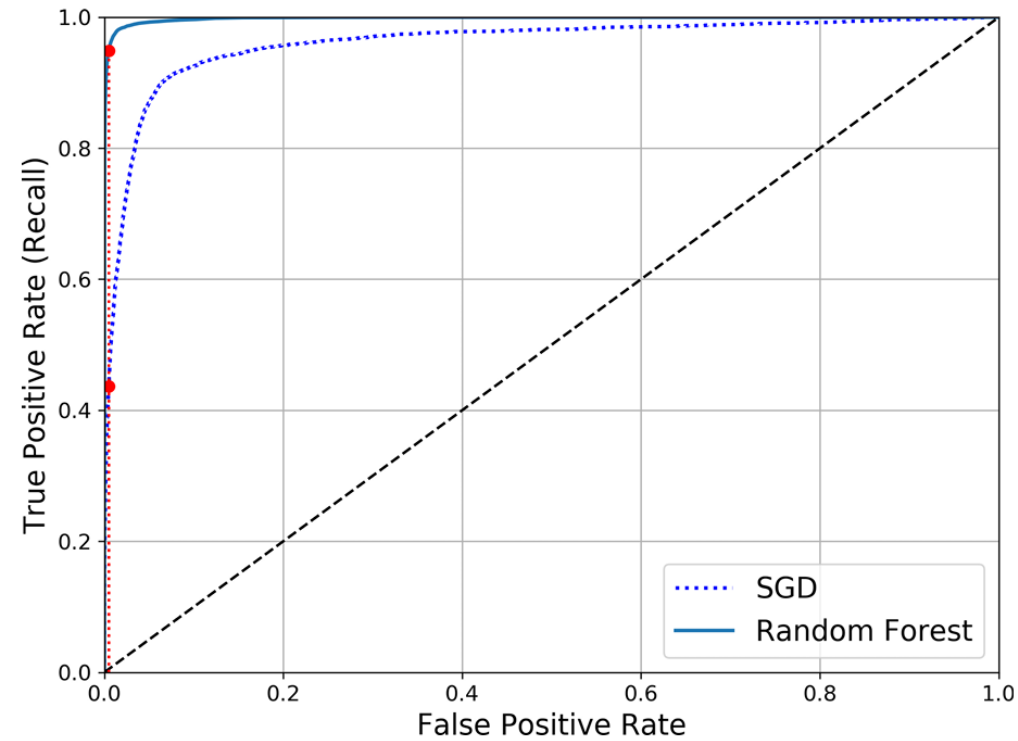
Let's now train a *RandomForestClassifier* and compare its ROC curve and ROC AUC score to those of the *SGDClassifier*. First, you need to get scores for each instance in the training set. But due to the way it works (see Chapter 7), the *RandomForestClassifier* class does not have a *decision\_function()* method. Instead, it has a *predict\_proba()* method.

```
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_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)
```

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



```
>>> roc_auc_score(y_train_5, y_scores_forest)
0.9983436731328145
```

Try measuring the precision and recall scores: you should find 99.0% precision and 86.6% recall. Not too bad!



# Multiclass Classification

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

Some algorithms (such as **SGD classifiers**, **Random Forest classifiers**, and **naive Bayes classifiers**) are capable of handling multiple classes natively.

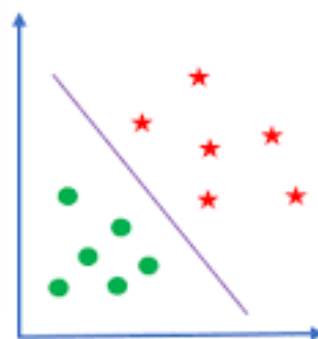
Others (such as **Logistic Regression** or **Support Vector Machine classifiers**) are strictly binary classifiers.

However, there are various strategies that you can use to perform multiclass classification with *multiple binary classifiers*:

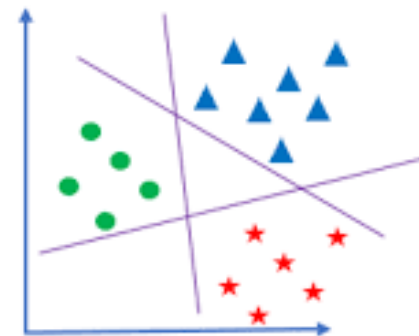
***One-versus-the-rest (OvR) strategy (also called one-versus-all):***

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 out-puts the highest score

Binary classification



Multi-class classification



### *One-versus-one (OvO ):*

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  $N \times (N - 1) / 2$  classifiers.

For most binary classification algorithms, **OvR is preferred**

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.



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.

Let's try this with a **Support Vector Machine classifier** (see Chapter 5), using the *sklearn.svm.SVC* class:

```
>>> from sklearn.svm import SVC
>>> svm_clf = SVC()
>>> svm_clf.fit(X_train, y_train) # y_train, not y_train_5
>>> svm_clf.predict([some_digit])
array([5], dtype=uint8)
```

it trained 45 binary classifiers, got their decision scores for the image, and selected the class that won the most duels

If you call the *decision\_function()* method:

```
>>> some_digit_scores = svm_clf.decision_function([some_digit])
>>> some_digit_scores
array([[ 2.92492871,  7.02307409,  3.93648529,  0.90117363,  5.96945908,
         9.5          ,  1.90718593,  8.02755089, -0.13202708,  4.94216947]])
```

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())
>>> ovr_clf.fit(X_train, y_train)

>>> ovr_clf.predict([some_digit])
array([5], dtype=uint8)
>>> len(ovr_clf.estimators_)
10
```



Training an SGDClassifier (or a RandomForestClassifier) is just as easy:

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

This time Scikit-Learn did not have to run OvR or OvO because SGD classifiers can directly classify instances into multiple classes. The `decision_function()` method now returns one value per class. Let's look at the score that the SGD classifier assigned to each class:

```
>>> sgd_clf.decision_function([some_digit])
array([[ -15955.22628,  -38080.96296, -13326.66695,    573.52692, -17680.68466,
         2412.53175, -25526.86498, -12290.15705, -7946.05205, -10631.35889]])
```



# Cross validation in Multiclass Classification

As usual, you can use cross-validation. Use the `cross_val_score()` function to evaluate the SGDClassifier's accuracy:

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")  
array([0.8489802 , 0.87129356, 0.86988048])
```

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 you can still do much better. Simply scaling the inputs (as discussed in Chapter 2) increases accuracy above 89%:

```
>>> 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([0.89707059, 0.8960948 , 0.90693604])
```

# Error Analysis

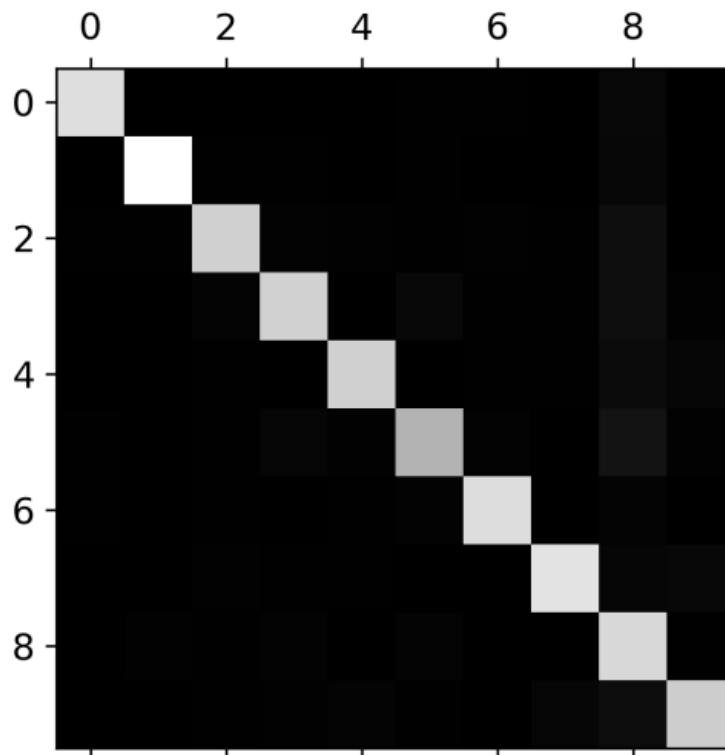
Here, we will assume that you have found a promising model and you want to find ways to improve it. One way to do this is to analyze the types of errors it makes.

First, look at the confusion matrix. You need to make predictions using the `cross_val_predict()` function, then call the `confusion_matrix()` function, just like you did earlier:

```
>>> 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
array([[5578,    0,   22,    7,    8,   45,   35,    5,  222,    1],
       [    0, 6410,   35,   26,    4,   44,    4,    8,  198,   13],
       [  28,   27, 5232,  100,   74,   27,   68,   37,  354,   11],
       [  23,   18,  115, 5254,    2,  209,   26,   38,  373,   73],
       [  11,   14,   45,   12, 5219,   11,   33,   26,  299,  172],
       [  26,   16,   31,  173,   54, 4484,   76,   14,  482,   65],
       [  31,   17,   45,    2,   42,   98, 5556,    3,  123,    1],
       [  20,   10,   53,   27,   50,   13,    3, 5696,  173,  220],
       [  17,   64,   47,   91,    3,  125,   24,   11, 5421,   48],
       [  24,   18,   29,   67,  116,   39,    1,  174,  329, 5152]])
```

That's a lot of numbers. It's often more convenient to look at an image representation of the confusion matrix, using Matplotlib's `matshow()` function:

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



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

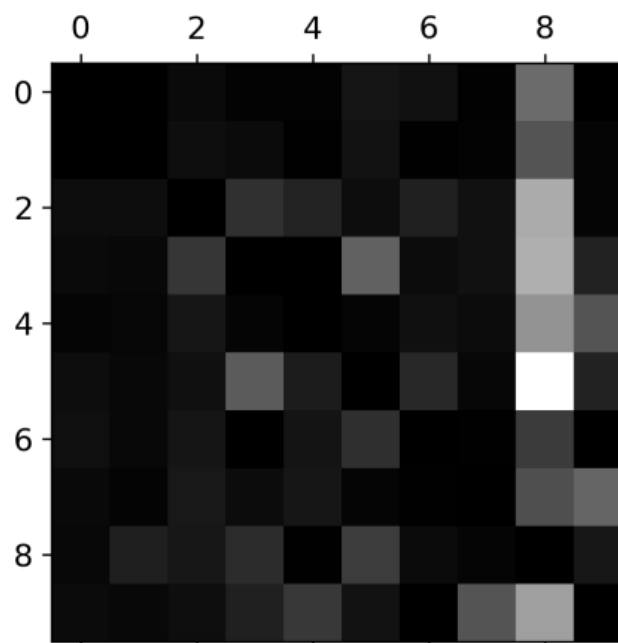
The 5s look slightly darker than the other digits, which could mean that there are fewer images of 5s in the dataset or that the classifier does not perform as well on 5s as on other digits.

Let's 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 that you can compare error rates instead of absolute numbers of errors**

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

Fill the diagonal with zeros to keep only the errors, and plot the result:

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

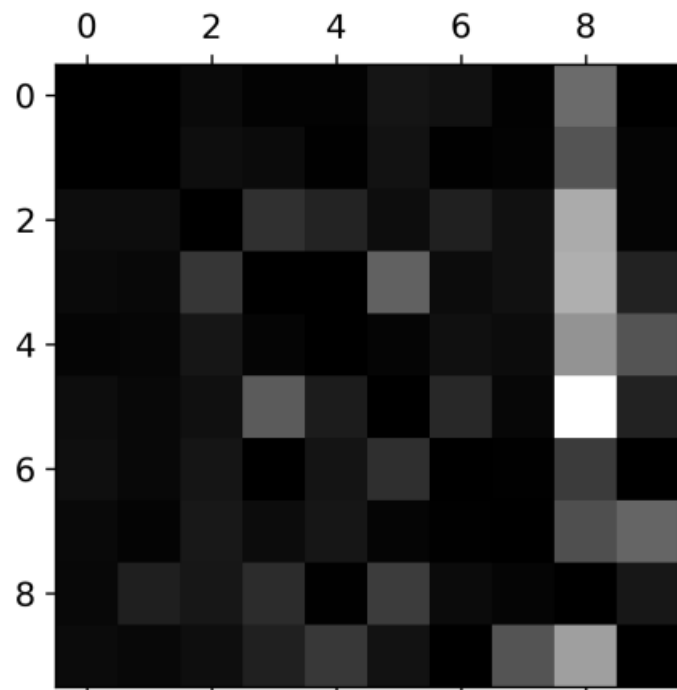


The column for class 8 is quite bright, which tells you that **many images get misclassified as 8s**

However, the row for class 8 is not that bad, telling you that **actual 8s in general get properly classified as 8s**.

As you can see, the confusion matrix is not necessarily symmetrical.

**You can also see that 3s and 5s often get confused** (in both directions)



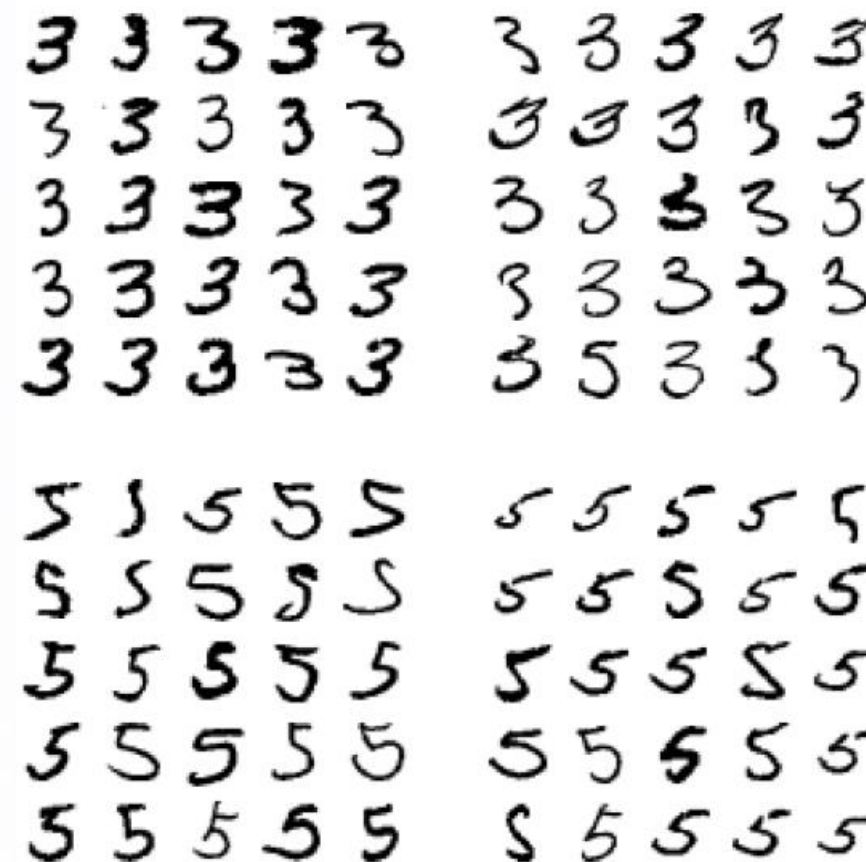
- ❖ You could try to gather more training data for digits that look like 8s (but are not) so that the classifier can learn to distinguish them from real 8s.
- ❖ Or you could engineer new features that would help the classifier—for example, writing an algorithm to count the number of closed loops (e.g., 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



let's plot examples of 3s and 5s

```
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=(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)
plt.show()
```



Classified as 3

Classified as 5

- ❑ Some of the digits that the classifier gets wrong (i.e., in the bottom-left and top-right blocks) **are so badly written that even a human would have trouble classifying them** (e.g., the 5 in the first row and second column truly looks like a badly written 3).
- ❑ However, **most misclassified images seem like obvious errors to us**, and it's hard to understand why the classifier made the mistakes it did.

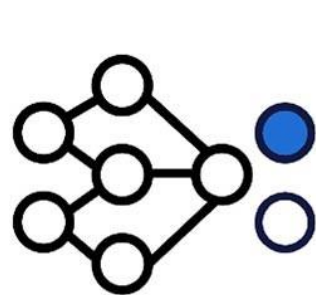
The reason is that we used a simple *SGDClassifier*, which is a **linear model**. All it does is assign a weight per class to each pixel, and when it sees a new image, it just sums up the weighted pixel intensities to get a score for each class. **So since 3s and 5s differ only by a few pixels, this 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.**

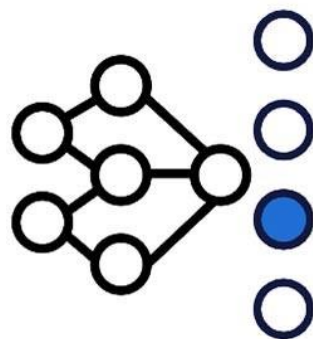
If you draw a 3 with the junction slightly shifted to the left, the classifier might classify it as a 5, and vice versa.

**In other words, this classifier is quite sensitive to image shifting and rotation.** So one way to reduce the 3/5 confusion would be to preprocess the images **to ensure that they are well centered and not too rotated.**

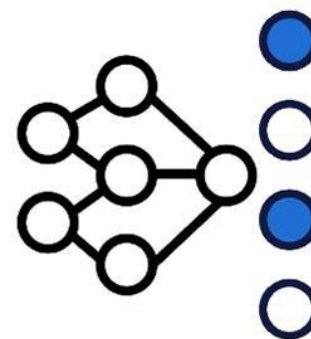
# Multilabel Classification



Binary



Multi-Class

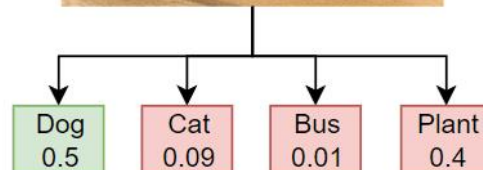


Multi-Label

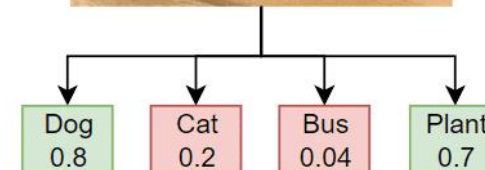
Binary Classification



Multiclass Classification



Multilabel Classification



```
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.

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

And it gets it right! The digit 5 is indeed not large (False) and odd (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. This 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
```



It 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 (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**). It is thus an example of a multioutput classification system.



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 with 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
```



```
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! This concludes our tour of classification. You should now know how to select good metrics for classification tasks, pick the appropriate precision/recall trade-off, compare classifiers, and more generally build good classification systems for a variety of tasks.