Hands-on Machine Learning

3. Classification

1.
The Dataset

MNIST Dataset

- MNIST dataset is a set of 70,000 small images of handwritten digits.

```
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', version=1)
```

- The `sklearn.datasets` package contains 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.

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

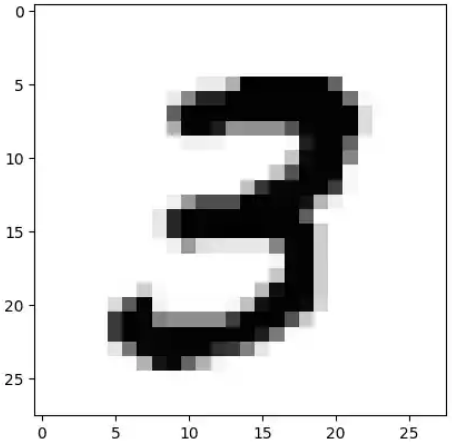
The Dataset Structure

- The “data” has 70,000 datapoints, each one consists of 784 (28×28) features which represents pixels of a 28×28-pixels image.

```
X, y = mnist.data, mnist.target
X
```

	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel10	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	p
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
69995	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
69996	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
69997	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
69998	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
69999	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	

70000 rows × 784 columns



The Dataset Structure

- The “target” is a vector of 70,000 labels (digits 0-9), each one shows what digit the datapoint represents.

```
y
```

```
0        5
1        0
2        4
3        1
4        9
        ..
69995    2
69996    3
69997    4
69998    5
69999    6
Name: class, Length: 70000, dtype: category
Categories (10, object): ['0', '1', '2', '3', ..., '6', '7', '8', '9']
```

```
y.shape
```

```
(70000,)
```

The Dataset Structure

- 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.

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

- Or use `to_numpy()` function:

```
X, y = mnist.data.to_numpy(), mnist.target.to_numpy()
X
```

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

```
y
```

```
array(['5', '0', '4', ..., '4', '5', '6'], dtype=object)
```

Datapoints

```
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()
```

Complexity of the Task

Split Data into Test and Train

- 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:
  - All cross-validation folds will be similar.
    - You don't want one fold to be missing some digits
  - Some learning algorithms are sensitive to the order of the training instances.
    - They perform poorly if they get many similar instances in a row.

Training a Binary Classifier

- Make a *binary classifier*, capable of distinguishing between just two classes, 5 and not-5:

```
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.

```
from sklearn.linear_model import SGDClassifier

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

- Make a prediction:

```
sgd_clf.predict([some_digit])
```

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

2.
Performance Measures

Measuring Accuracy Using Cross-Validation

- Use 3-fold cross-validation:

```
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 ])
```

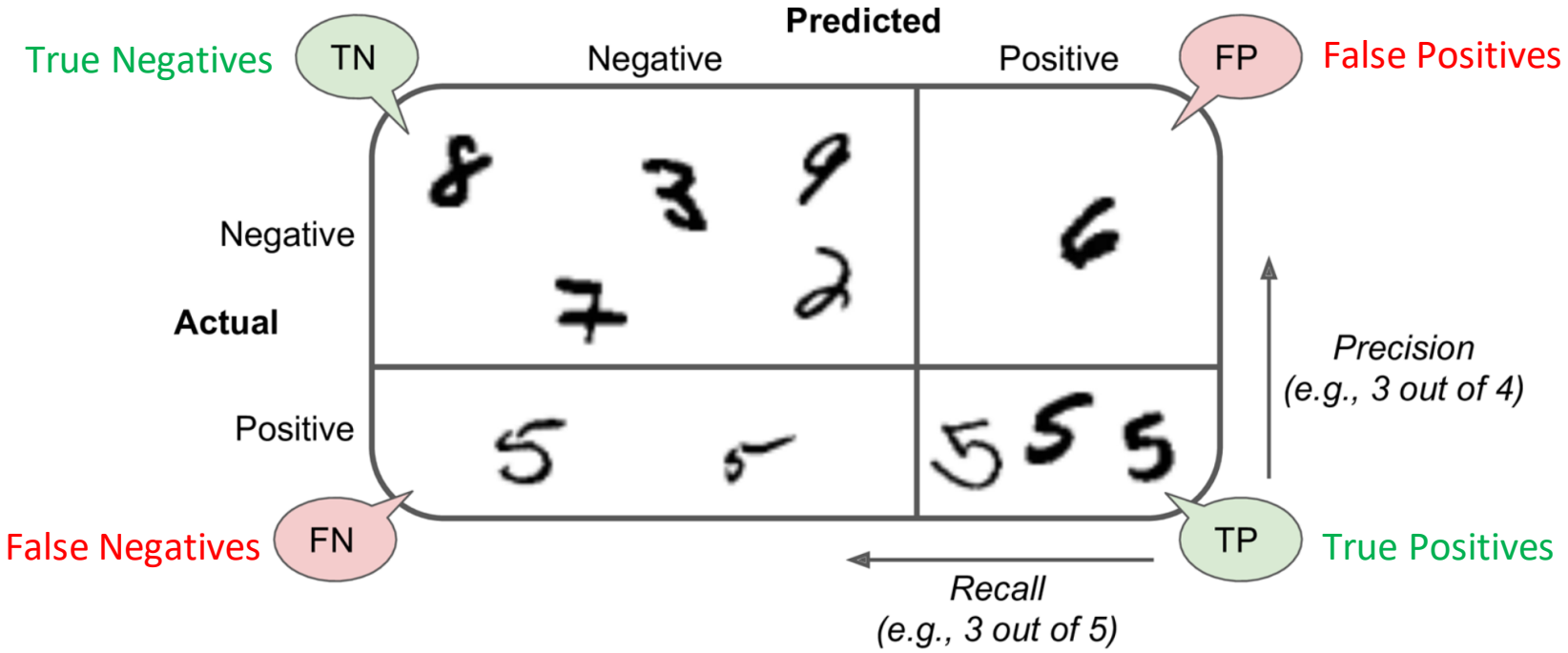
- Above 95% accuracy (ratio of correct predictions)! Compare with a dumb classifier that classifies every image in the “not-5” class.

```
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])
```

Confusion Matrix



Confusion Matrix

- 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)
```

```
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_train_5, y_train_pred)
cm
```

```
array([[53892,   687],
       [ 1891,  3530]])
```

	Negative	Positive
Negative	$TN = 53892$	$FP = 687$
Positive	$FN = 1891$	$TP = 3530$

Precision and Recall

- *Precision*: the accuracy of the positive predictions.

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

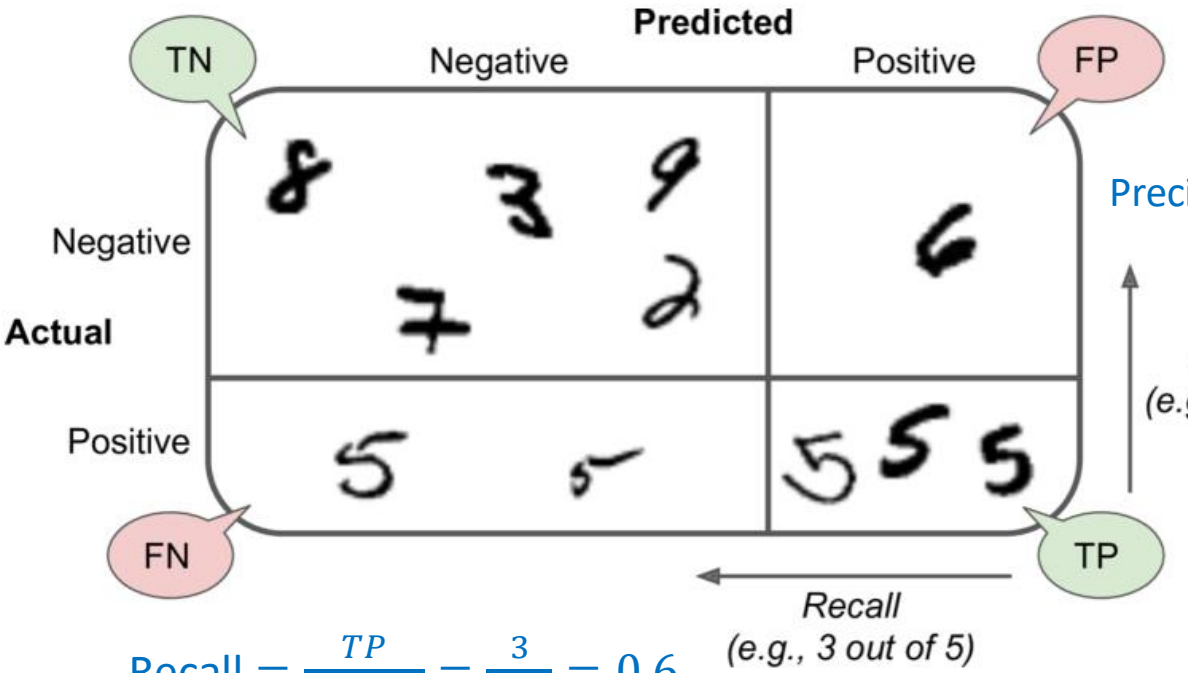
  - You can have perfect precision by making one single positive prediction and ensure it is correct!

- *Recall* (aka *sensitivity* or *true positive rate* (TPR)): the ratio of positive instances that are correctly detected by the classifier

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

  - You can have perfect recall by predicting all instances as positive!

Precision and Recall



$$\text{Precision} = \frac{TP}{TP+FP} = \frac{3}{3+1} = 0.75$$

$$\text{Recall} = \frac{TP}{TP+FN} = \frac{3}{3+2} = 0.6$$

F_1 Score

- The F_1 score is the *harmonic mean* of precision and recall:

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

```
from sklearn.metrics import 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
```

```
from sklearn.metrics import f1_score

f1_score(y_train_5, y_train_pred)
```

```
0.7325171197343846
```

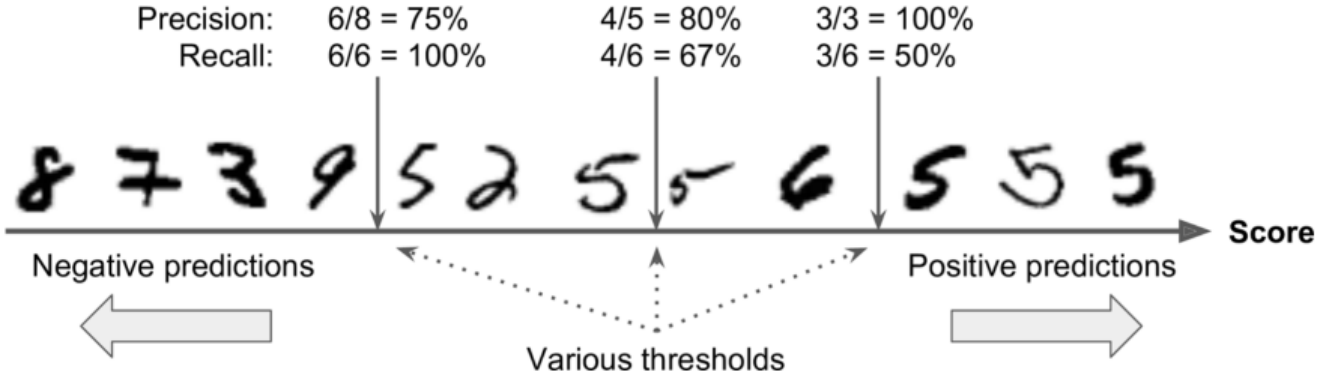
$TN = 53892$	$FP = 687$
$FN = 1891$	$TP = 3530$

Precision/Recall Trade-off

- How the `SGDClassifier` makes its classification decisions?
- It computes a score based on a *decision function*:

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

- If that score is greater than a threshold, it assigns the instance to the positive class.



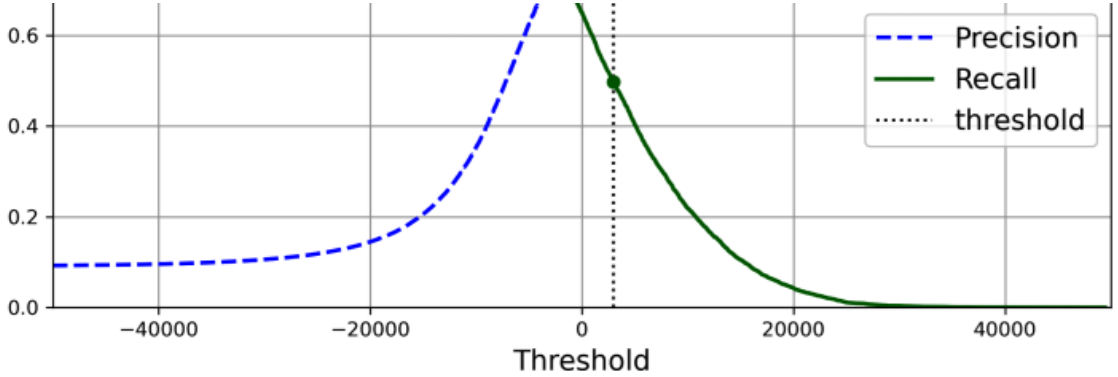
Precision/Recall Trade-off

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

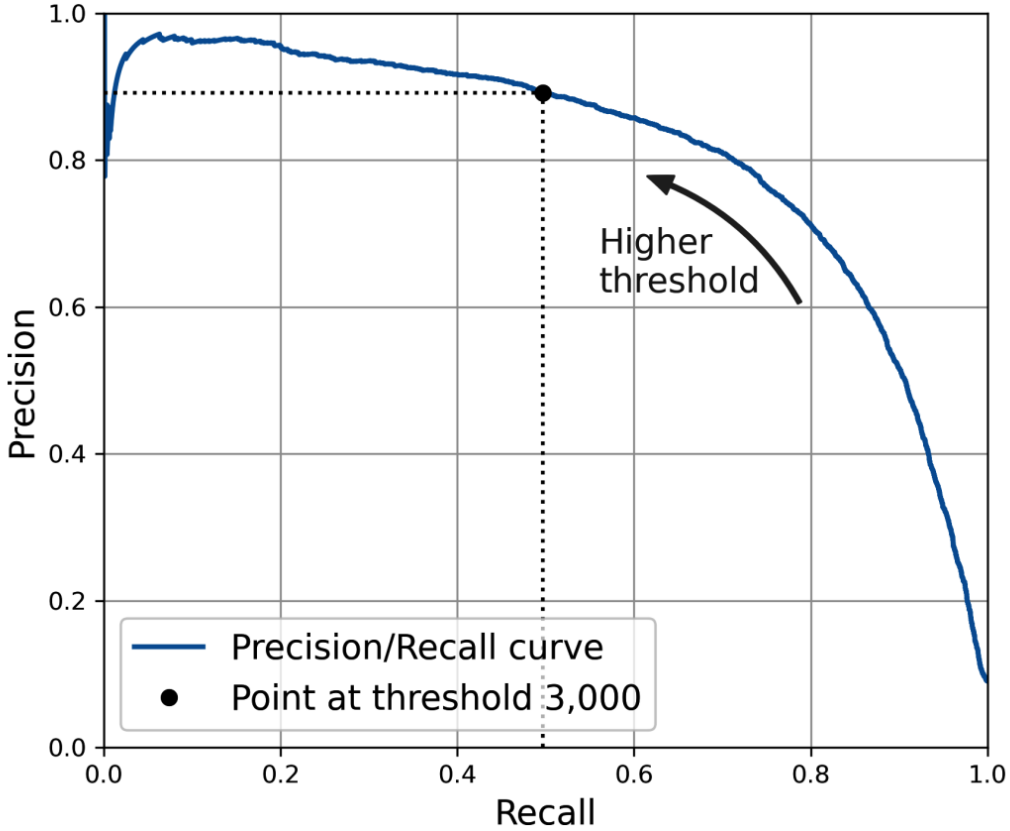
```
from sklearn.metrics import precision_recall_curve

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

```
y_scores[:5]
```

```
array([  1200.93051237, -26883.79202424, -33072.03475406, -15919.5480689 ,
       -20003.53970191])
```



Precision/Recall Trade-off



Achieve a Target Precision

- Suppose you decide to aim for 90% precision. Search for the lowest threshold that gives you at least 90% precision:

```
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, 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)
```

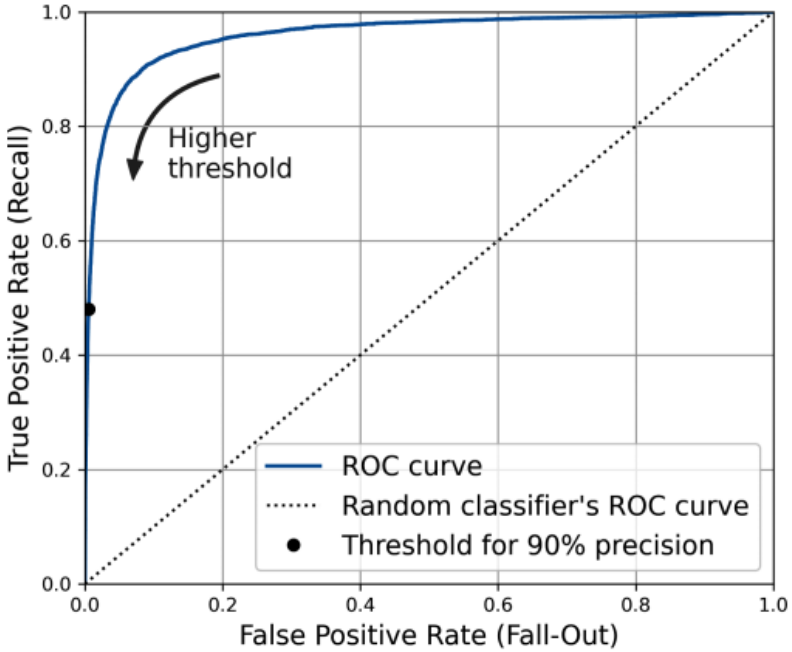
```
0.9000345901072293
```

The ROC Curve

- The *receiver operating characteristic* (ROC) curve is similar to the precision/recall curve.
- The ROC curve plots the *true positive rate* (i.e. recall) against the *false positive rate* (FPR).
- The FPR is the ratio of negative instances that are incorrectly classified as positive: $\frac{FP}{FP+TN}$
- FPR = 1 – the *true negative rate* (TNR or *specificity*), which is the ratio of negative instances that are correctly classified as negative.
- The ROC curve plots *sensitivity* (recall) versus 1 – *specificity*.

The ROC Curve

```python
from sklearn.metrics import roc_curve

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



ROC AUC

- 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)
```

```
0.9604938554008616
```

- We prefer the precision/recall (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.

Random Forest Classifier

- `RandomForestClassifier` class does not have a `decision_function()` method. Instead, it has a `predict_proba()` method.
- 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")
```

```
y_probas_forest[:5]
```

```
array([[0.11, 0.89],
       [0.99, 0.01],
       [0.96, 0.04],
       [1.  , 0.  ],
       [0.99, 0.01]])
```

Comparing Classifiers

- The `precision_recall_curve()` function expects labels and scores, but instead of scores you can give it class probabilities:

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

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

```
0.9274509803921569
```

```
roc_auc_score(y_train_5, y_scores_forest)
```

```
0.9983436731328145
```

```
precision_score(y_train_5, y_train_pred_forest)
```

```
0.9897468089558485
```

```
recall_score(y_train_5, y_train_pred_forest)
```

```
0.8725327430363402
```