DATA 201 - Techniques in Data Science

Lectures 8-9: Classification

Binh Nguyen

School of Mathematics and Statistics, Victoria University of Wellington

Adapted from “Hands-On Machine Learning with Scikit-Learn and TensorFlow” by Aurèlien Géron

Table of contents

Links

- Data 201 Course Page: `https://sms.wgtn.ac.nz/Courses/DATA201_2020T1/WebHome`
- Data 201 Class Facebook Page: `https://www.facebook.com/groups/3449204258439404`
- Google Colaboratory: `https://colab.research.google.com`
- Microsoft Azure Notebooks: `https://notebooks.azure.com`
- Try Jupyter Notebook: `https://mybinder.org/v2/gh/ipython/ipython-in-depth/master?filepath=binder/Index.ipynb`
- CoCal: `https://cocalc.com`

Setup

```
# Common imports
import numpy as np
import os

# Scikit-Learn ≥0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"

# For better representation of figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
```

MNIST Dataset

Load the Dataset

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. We will use function `fetch_openml()` in scikit-learn v0.20+ to fetch the dataset.

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1)
```

```
mnist.keys()
```

```
dict_keys(['data', 'target', 'frame', 'feature_names', 'target_nam
es', 'DESCR', 'details', 'categories', 'url'])
```

The `DESCR` key describes the dataset:

```
print(mnist.DESCR)
```

Author: Yann LeCun, Corinna Cortes, Christopher J.C. Burges
Source: [MNIST Website](http://yann.lecun.com/exdb/mnist/) - Date unknown
Please cite:

The MNIST database of handwritten digits with 784 features, raw data available at: http://yann.lecun.com/exdb/mnist/. (http://yann.lecun.com/exdb/mnist/.) It can be split in a training set of the first 60,000 examples, and a test set of 10,000 examples

It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting. The original black and white (bilevel) images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. the images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field.

With some classification methods (particularly template-based methods, such as SVM and K-nearest neighbors), the error rate improves when the digits are centered by bounding box rather than center of mass. If you do this kind of pre-processing, you should report it in your publications. The MNIST database was constructed from NIST's NIST originally designated SD-3 as their training set and SD-1 as their test set. However, SD-3 is much cleaner and easier to recognize than SD-1. The reason for this can be found on the fact that SD-3 was collected among Census Bureau employees, while SD-1 was collected among high-school students. Drawing sensible conclusions from learning experiments requires that the result be independent of the choice of training set and test among the complete set of samples. Therefore it was necessary to build a new database by mixing NIST's datasets.

The MNIST training set is composed of 30,000 patterns from SD-3 and 30,000 patterns from SD-1. Our test set was composed of 5,000 patterns from SD-3 and 5,000 patterns from SD-1. The 60,000 pattern training set contained examples from approximately 250 writers. We made sure that the sets of writers of the training set and test set were disjoint. SD-1 contains 58,527 digit images written by 500 different writers. In contrast to SD-3, where blocks of data from each writer appeared in sequence, the data in SD-1 is scrambled. Writer identities for SD-1 is available and we used this information to unscramble the writers. We then split SD-1 in two: characters written by the first 250 writers went into our new training set. The remaining 250 writers were placed in our test set. Thus we had two sets with nearly 30,000 examples each. The new training set was completed with enough examples from SD-3, starting at pattern # 0, to make a full set of 60,000 training patterns. Similarly, the new test set was completed with SD-3 examples starting at pattern # 35,000 to make a full set with 60,000 test patterns. Only a subset of 10,000 test images (5,000 from SD-1 and 5,000 from SD-3) is available on this site. The full 60,000 sample training set is available.

The `data` key contains an array with one row per instance and one column per feature. The `target` key contains an array with the *labels*.

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

```
print(X.shape, ' & ', y.shape)
```

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

There are 70,000 images, and each image has 784 features. This is because each image is 28x28 pixels, and each feature simply represents one pixel's intensity, from 0 (white) to 255 (black).

To display a digit from the dataset, we need to its feature vector, reshape the vector to a 28x28 array, and display it using Matplotlib's `imshow()` function.

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

```
def plot_digits(instances, images_per_row=10, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    images = [instance.reshape(size,size) for instance in instances]
    n_rows = (len(instances) - 1) // images_per_row + 1
    row_images = []
    n_empty = n_rows * images_per_row - len(instances)
    images.append(np.zeros((size, size * n_empty)))
    for row in range(n_rows):
        rimages = images[row * images_per_row : (row + 1) * images_per_row]
        row_images.append(np.concatenate(rimages, axis=1))
    image = np.concatenate(row_images, axis=0)
    plt.imshow(image, cmap = mpl.cm.binary, **options)
    plt.axis("off")
```

```
plot_digit(X[0])
```

```
y[0]
```

```
'5'
```

```
y = y.astype(np.uint8) # since the labels are strings, let’s cast y to integers
```

```
y = y.astype(np.uint8) # since the labels are strings, let’s cast y to integers
```

```
plt.figure(figsize=(9,9))
plot_digits(X[:100], images_per_row=10)
```

The MNIST dataset is actually already split into a training set (the first 60,000 images - this training set is already shuffled) 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:]
```

Binary Classification

We simplify the problem: to identify only 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 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)
```

Now let's select a classifier and train it. We start 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. Let's create an `SGDClassifier` and train it on the whole training set:

```
from sklearn.linear_model import SGDClassifier

# The SGDClassifier relies on randomness during training
# so to reproducible results, we should set the random_state parameter.
sgd_clf = SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

```
SGDClassifier(alpha=0.0001, average=False, class_weight=None,
              early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
              l1_ratio=0.15, learning_rate='optimal', loss='hinge',
              max_iter=1000, n_iter_no_change=5, n_jobs=None, penalty='l2',
              power_t=0.5, random_state=42, shuffle=True, tol=0.001,
              validation_fraction=0.1, verbose=0, warm_start=False)
```

Now we can use it to detect an image of the number 5, for example, X[0]:

```
sgd_clf.predict([X[0]])
```

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

Performance Measures

There are many performance measures available and evaluating a classifier is often trickier than evaluating a regressor.

We will discuss:

- Measuring accuracy using cross-validation
- Confusion matrix
- Precision, Recall, and F1-score
- Precision/Recall Trade-off
- The ROC curve

Measuring accuracy using cross-validation

```python
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 ])
```

If we want to have more control over the cross-validation process than what function `cross_val_score` provides then we can implement cross-validation as below.

```python
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3)

for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_fold = X_train[test_index]
    y_test_fold = y_train_5[test_index]

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred))
```

```
0.95035
0.96035
0.9604
```

Obtaining more than 95% of accuracy (ratio of correct predictions) on all cross-validation folds looks amazing. However, let's examine 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(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
```

Let find out this model's accuracy:

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

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

This naive model also has over 90% accuracy. This is because only about 10% of the images are 5s, so if we always guess that an image is not a 5, we will be right about 90% of the time.

So accuracy is generally not the preferred performance measure for classifiers, especially when dealing with skewed datasets.

Confusion matrix

A much better way to evaluate the performance of a classifier is to look at the *confusion matrix*. To compute the confusion matrix, we 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

# cross_val_predict() performs K-fold CV and returns the predictions made on each test fold
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_train_5, y_train_pred)
```

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

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*): 53,892 of them were correctly classified as non-5s (they are called *true negatives*), while the remaining 687 were wrongly classified as 5s (*false positives*). The second row considers the images of 5s (the *positive class*): 1,891 were wrongly classified as non-5s (*false negatives*), while the remaining 3,530 were correctly classified as 5s (*true positives*).

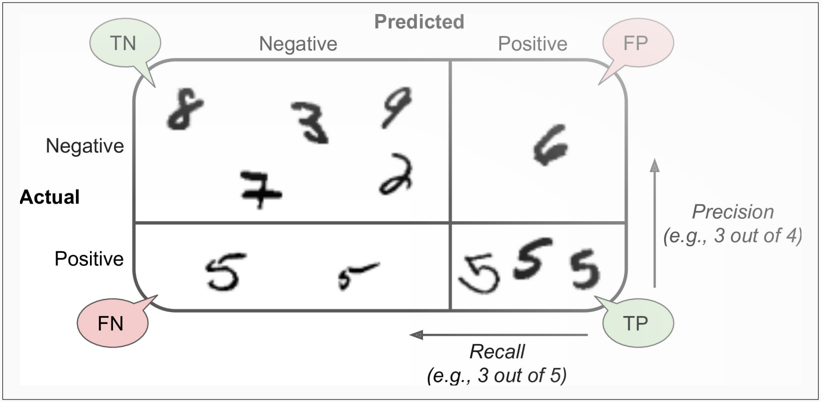
The **precision** of the classifier is the accuracy of the positive predictions:

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

Precision is typically used 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.

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

An illustrated confusion matrix shows examples of true negatives (TN), false positives (FP), false negatives (FN), and true positives (TP):



TP: hit; TN: correct rejection; FP: false alarm, Type I error; FN: miss, Type II error

$$\text{Precision} = \frac{TP}{TP + FP}$$
$$\text{Recall (Sensitivity)} = \frac{TP}{P} = \frac{TP}{TP + FN}$$
$$\text{Specificity} = \frac{TN}{N} = \frac{TN}{TN + FP}$$

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)
```

```
0.8370879772350012
```

```
3530 / (3530 + 687)
```

```
0.8370879772350012
```

```
recall_score(y_train_5, y_train_pred)
```

```
0.6511713705958311
```

```
3530 / (3530 + 1891)
```

```
0.6511713705958311
```

Although having a very high accuracy, our 5-detector is correct only 83.7% of the time when claiming an image represents a 5, and moreover, it only detects 65.1% of the 5s.

F1-score

It is often convenient to combine *precision* and *recall* into a single metric called the **F1 score**, in particular if we 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.

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

```
from sklearn.metrics import f1_score

f1_score(y_train_5, y_train_pred)
```

```
0.7325171197343846
```

```
3530 / (3530 + (687 + 1891) / 2)
```

```
0.7325171197343847
```

The **F1 score** favors classifiers that have similar *precision* and *recall*. This is not always what we want: in some contexts we mostly care about *precision*, and in other contexts we really care about *recall*. Unfortunately, we can’t have it both ways: increasing *precision* reduces *recall*, and vice versa. This is called the precision/recall tradeoff.

Precision/Recall Trade-off

Let's look at how 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.

Instead of calling the classifier’s `predict()` method, we can call its `decision_function()` method, which returns a score for each instance, and then make predictions based on those scores using any threshold we want:

```
y_scores = sgd_clf.decision_function([X[0]])
y_scores
```

```
array([2164.22030239])
```

```
threshold = 0
y0_predicted = (y_scores > threshold)
y0_predicted
```

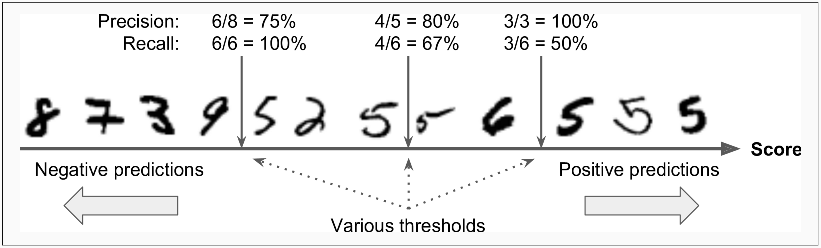
```
array([ True])
```

```
threshold = 5000
y0_predicted = (y_scores > threshold)
y0_predicted
```

```
array([False])
```

The image in X[0] actually represents a 5, and the classifier detects it when the threshold is 0, but it misses it when the threshold is increased to 5,000. This shows that raising the threshold decreases *recall*. So how do we decide which threshold to use?

Example



Precision: 6/8 = 75% 4/5 = 80% 3/3 = 100%
Recall: 6/6 = 100% 4/6 = 67% 3/6 = 50%
Score
Negative predictions
Positive predictions
Various thresholds

Suppose the *decision threshold* is positioned at the central arrow (between the two 5s): you will find 4 true positives (actual 5s) on the right of that threshold, and 1 false positive (actually a 6). Therefore, with that threshold, the precision is 80% (4 out of 5). But out of 6 actual 5s, the classifier only detects 4, so the recall is 67% (4 out of 6). If you raise the threshold (move it to the arrow on the right), the false positive (the 6) becomes a true negative, thereby increasing the precision (up to 100% in this case), but one true positive becomes a false negative, decreasing recall down to 50%. Conversely, lowering the threshold increases recall and reduces precision.

We will use the `cross_val_predict()` function to get the decision scores of all instances in the training set, then compute *precision* and *recall* for all possible thresholds using the `precision_recall_curve()` function:

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

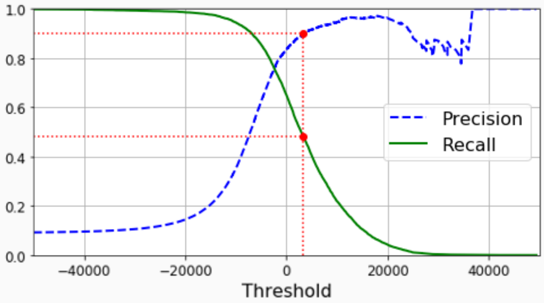
```
from sklearn.metrics import precision_recall_curve

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

Then we can plot precision and recall as functions of the threshold value using Matplotlib:

```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
    plt.legend(loc="center right", fontsize=16)
    plt.xlabel("Threshold", fontsize=16)
    plt.grid(True)
    plt.axis([-50000, 50000, 0, 1])
```

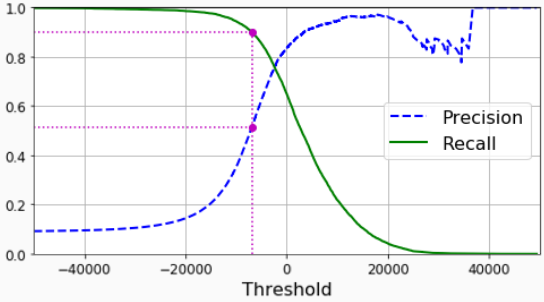
```
recall_90_precision = recalls[np.argmax(precisions >= 0.90)]
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)]
```

```
plt.figure(figsize=(8, 4))
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.plot([threshold_90_precision, threshold_90_precision], [0., 0.9], "r:")
plt.plot([-50000, threshold_90_precision], [0.9, 0.9], "r:")
plt.plot([-50000, threshold_90_precision], [recall_90_precision, recall_90_precision], "r:")
plt.plot([threshold_90_precision], [0.9], "ro")
plt.plot([threshold_90_precision], [recall_90_precision], "ro")
plt.show()
```



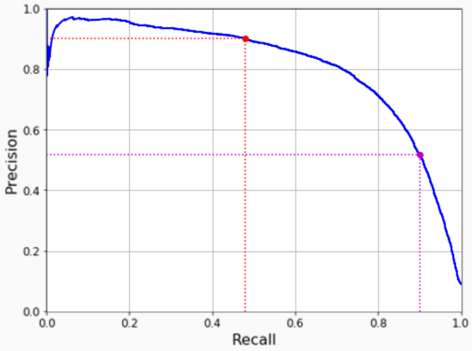
```
precision_90_recall = precisions[np.argmin(recalls >= 0.90)]
threshold_90_recall = thresholds[np.argmin(recalls >= 0.90)]
```

```
plt.figure(figsize=(8, 4))
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.plot([threshold_90_recall, threshold_90_recall], [0., 0.9], "m:")
plt.plot([-50000, threshold_90_recall], [0.9, 0.9], "m:")
plt.plot([-50000, threshold_90_recall], [precision_90_recall, precision_90_recall], "m:")
plt.plot([threshold_90_recall], [0.9], "mo")
plt.plot([threshold_90_recall], [precision_90_recall], "mo")
plt.show()
```



Note that when the threshold is increased, recall can only go down while in general precision will go up but it may sometimes go down. Why?

```python
def plot_precision_vs_recall(precisions, recalls, label=None):
    plt.plot(recalls, precisions, "b-", linewidth=2, label=label)
    plt.xlabel("Recall", fontsize=16)
    plt.ylabel("Precision", fontsize=16)
    plt.axis([0, 1, 0, 1])
    plt.grid(True)

plt.figure(figsize=(8, 6))
plot_precision_vs_recall(precisions, recalls)
plt.plot([recall_90_precision, recall_90_precision], [0., 0.9], "r:")
plt.plot([0.0, recall_90_precision], [0.9, 0.9], "r:")
plt.plot([recall_90_precision], [0.9], "ro")
plt.plot([0., 0.9], [precision_90_recall, precision_90_recall], "m:")
plt.plot([0.9, 0.9], [0.0, precision_90_recall], "m:")
plt.plot([0.9], [precision_90_recall], "mo")
plt.show()
```



So if we decide to aim for 90% precision, we will search for the lowest threshold that gives us at least 90% precision. Then we can use that threshold to make predictions instead of calling the classifier's `predict()` method.

```
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)]
y_train_pred_90 = (y_scores >= threshold_90_precision)
```

Let's check these predictions' precision and recall:

```
precision_score(y_train_5, y_train_pred_90)
```

```
0.9000345901072293
```

```
recall_score(y_train_5, y_train_pred_90)
```

```
0.4799852425751706
```

It is easy to develop a high-precision classifier (just by setting a high enough threshold) but it is not very useful if its recall is too low!

The ROC Curve

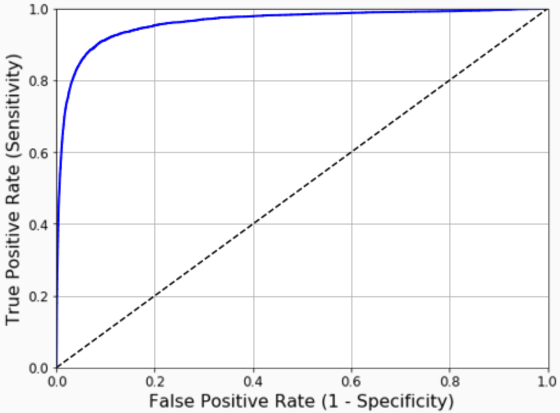
The *receiver operating characteristic (ROC) curve* is another common tool used with binary classifiers. It is very similar to the precision/recall curve, but instead of plotting *precision* versus *recall*, the ROC curve plots the **true positive rate** (another name for *recall*) against the **false positive rate** (FPR).

The FPR is the ratio of negative instances that are incorrectly classified as positive. It is equal to one minus the *true negative rate*, which is the ratio of negative instances that are correctly classified as negative. The *true negative rate* (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 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)
```

```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, "b-", linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate (1 - Specificity)', fontsize=16)
    plt.ylabel('True Positive Rate (Sensitivity)', fontsize=16)
    plt.grid(True)
plt.figure(figsize=(8, 6))
plot_roc_curve(fpr, tpr)
```



There is another tradeoff: the higher the sensitivity/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 should stay as far away from that line as possible (toward the top-left corner).

Area under the Curve (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
```

Among the ROC curve and the precision/recall (or PR) curve, we should prefer the PR curve whenever *the positive class is rare* or when we *care more about the false positives than the false negatives*. Otherwise, the ROC curve should be used.

In this example, from the ROC curve (and also the ROC AUC score), we may think that the classifier is really good. However, 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-right corner).

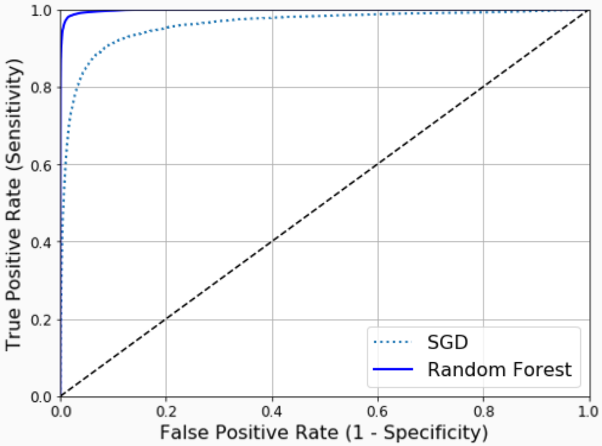
Try a `RandomForestClassifier`

Let's train a `RandomForestClassifier` and compare its ROC curve and ROC AUC score to the `SGDClassifier`. However, the `RandomForestClassifier` class does not have a `decision_function()` method. Instead it has a `predict_proba()` method which 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. We will use the positive class's probability as the *score*.

```
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.figure(figsize=(8, 6))
plt.plot(fpr, tpr, ":", linewidth=2, label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plt.legend(loc="lower right", fontsize=16)
plt.show()
```



As can be seen, 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)
```

```
0.9983436731328145
```

The precision and recall scores of the `RandomForestClassifier` are also much better:

```
y_train_pred_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3)
precision_score(y_train_5, y_train_pred_forest)
```

```
0.9905083315756169
```

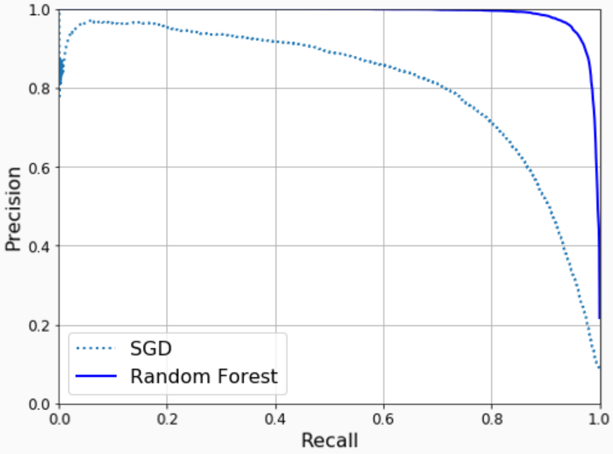
```
recall_score(y_train_5, y_train_pred_forest)
```

```
0.8662608374838591
```

And the `RandomForestClassifier` 's PR curve is much better than the `SGDClassifier` :

```
precisions_forest, recalls_forest, thresholds_forest = precision_recall_curve(y_train_5, y_scores_forest)
```

```
plt.figure(figsize=(8, 6))
plt.plot(recalls, precisions, ":", linewidth=2, label="SGD")
plot_precision_vs_recall(precisions_forest, recalls_forest, label="Random Forest")
plt.legend(loc="lower left", fontsize=16)
plt.show()
```



Summary

- Train binary classifiers
- Choose an appropriate metric for a specific task
- Evaluate the classifiers using cross-validation
- Select the precision/recall trade-off that fits the needs
- Use ROC curves (or PR curves) and ROC AUC scores to compare various models

Quiz - Question 1

In a future society, a machine is used to predict a crime before it occurs. If you were responsible for tuning this machine, what evaluation metric would you want to maximize to ensure no innocent people (people not about to commit a crime) are imprisoned (where crime is the positive label)?

a) Accuracy
b) Precision
c) Recall
d) F1-score
e) AUC-ROC

Answer: Precision

Quiz - Question 2

Consider the machine from the previous question. If you were responsible for tuning this machine, what evaluation metric would you want to maximize to ensure all criminals (people about to commit a crime) are imprisoned (where crime is the positive label)?

a) Accuracy

b) Precision

c) Recall

d) F1-score

e) AUC-ROC

Answer: Recall

To be continued...

Multiclass Classification

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

Some algorithms (for example, Random Forest) are capable of handling multiple classes directly. Others (for example, Support Vector Machines) are strictly binary classifiers. However, there are various strategies that you can use to perform *multiclass classification* using multiple binary classifiers. The two common strategies are the *one-versus-all* (OvA) strategy (also called one-versus-the-rest) and the *one-versus-one* (OvO) strategy.

- OvA/OvR: train N binary classifiers to classify N classes (e.g., a 0-detector, a 1-detector, etc.) and select the class whose classifier outputs the highest score when classifying one example.
- OvO: train $N \times (N-1)/2$ binary classifier to classify N classes (e.g., 0 vs 1, 0 vs 2, 1 vs 2, etc.) and go through all those classifiers to see which class wins the most duels when classifying one example.

Some algorithms (such as, Support Vector Machines) 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.

Let's try the SGDClassifier for a multiclass classification task.

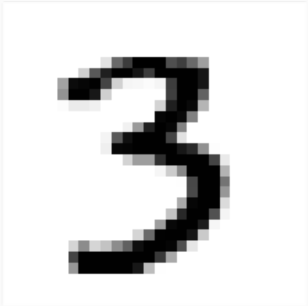
```
# classification of 10 classes, so here is y_train, not y_train_5
sgd_clf.fit(X_train, y_train)
```

```
SGDClassifier(alpha=0.0001, average=False, class_weight=None,
              early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
              l1_ratio=0.15, learning_rate='optimal', loss='hinge',
              max_iter=1000, n_iter_no_change=5, n_jobs=None, penalty='l2',
              power_t=0.5, random_state=42, shuffle=True, tol=0.001,
              validation_fraction=0.1, verbose=0, warm_start=False)
```

```
sgd_clf.intercept_
```

```
array([ -214.23804761,   -31.47820709,  -243.39283758,  -412.61204135,
        -161.56175199,   103.02146015,  -245.40214746,   -32.70213188,
       -1297.23305113,  -570.06078343])
```

```
# We will make prediction on this image
a_digit = X[10000]
plot_digit(a_digit)
```



```
# That image has label 3 (digit 3)
y[10000]
```

```
3
```

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

```
array([3], dtype=uint8)
```

We can force Scikit-Learn to use OvO or OvA by creating an instance of the `OneVsOneClassifier` or `OneVsRestClassifier` classes, respectively, then passing a binary classifier to its constructor. For example, this code creates a multiclass classifier using the OvO strategy based on a `SGDClassifier` :

```
from sklearn.multiclass import OneVsOneClassifier

ovo_clf = OneVsOneClassifier(SGDClassifier(max_iter=5, tol=-np.infty, random_state=42))
ovo_clf.fit(X_train, y_train)
```

```
OneVsOneClassifier(estimator=SGDClassifier(alpha=0.0001, average=False,
                                           class_weight=None,
                                           early_stopping=False, epsilon=0.1,
                                           eta0=0.0, fit_intercept=True,
                                           l1_ratio=0.15,
                                           learning_rate='optimal',
                                           loss='hinge', max_iter=5,
                                           n_iter_no_change=5, n_jobs=None,
                                           penalty='l2', power_t=0.5,
                                           random_state=42, shuffle=True,
                                           tol=-inf, validation_fraction=0.1,
                                           verbose=0, warm_start=False),
                   n_jobs=None)
```

```
ovo_clf.predict([a_digit])
```

```
array([3], dtype=uint8)
```

```
len(ovo_clf.estimators_)
```

```
45
```

Now we train a `RandomForestClassifier` for classification of the 10 classes.

```
forest_clf.fit(X_train, y_train)
forest_clf.predict([a_digit])
```

```
array([3], dtype=uint8)
```

Random Forest classifiers can directly classify instances into multiple classes, and we can call `predict_proba()` to get the list of probabilities that the classifier assigned to each instance for each class:

```
forest_clf.predict_proba([a_digit])
```

```
array([[0., 0., 0., 1., 0., 0., 0., 0., 0., 0.]])
```

Now we evaluate these classifiers:

```
cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
```

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

```
cross_val_score(forest_clf, X_train, y_train, cv=3, scoring="accuracy")
```

```
array([0.9646 , 0.96255, 0.9666 ])
```

```
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([0.8983, 0.891 , 0.9018])
```

```
cross_val_score(forest_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
```

```
array([0.96445, 0.96255, 0.96645])
```

Error Analysis

After having a promising model, we may want to find ways to improve it. One way to do this is to analyze the types of errors it makes.

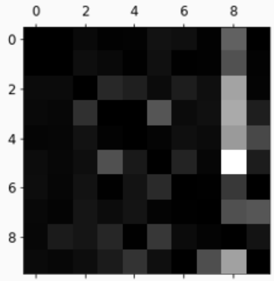
```
y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
conf_mx = confusion_matrix(y_train, y_train_pred)
conf_mx
```

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

Compare error rates instead of absolute number 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 then plot the result
np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()
```



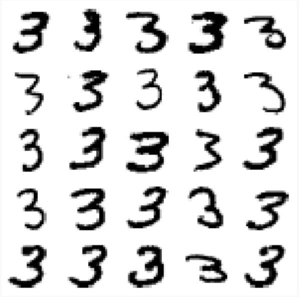
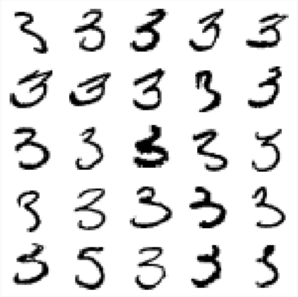
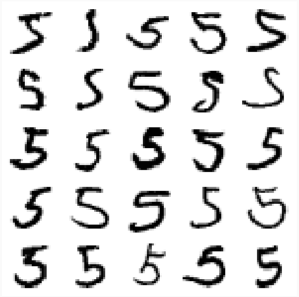
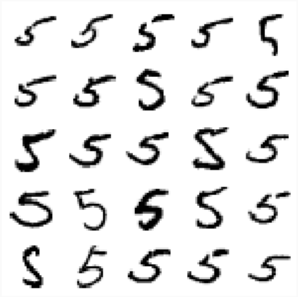
We can observe some kinds of errors that the classifier makes. Remember that rows represent actual classes, while columns represent predicted classes.

- The columns for class 8 is quite bright => many images get misclassified as 8s.
- The rows for classes 8 and 9 are also quite bright => 8s and 9s are often confused with other digits.
- Some rows are pretty dark, such as rows 0 and 1 => most 0s and 1s are classified correctly.
- The errors are not perfectly symmetrical; for example, there are more 5s misclassified as 8s than the reverse.

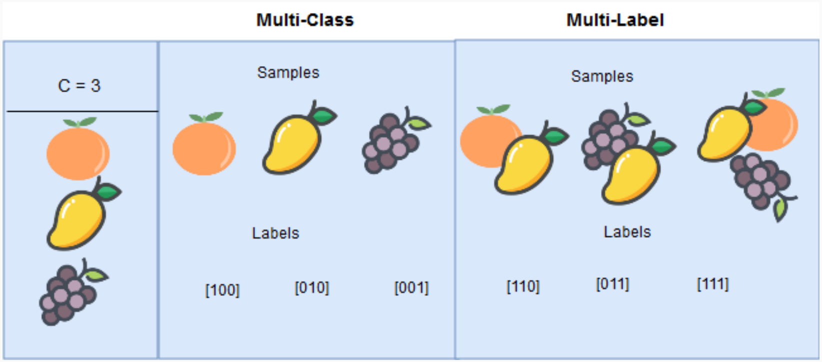
Analyzing the confusion matrix can suggest us ways to improve our classifier. In this case, we should try to reduce the false 8s. For example, we could try to collect more training data for non-8 digits looking like 8s so the classifier can learn to distinguish them from real 8s. Or we could engineer 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).

Analyzing individual errors can also be a good way to gain insights on what the classifier is doing and why it is failing, but it is more difficult and time-consuming.

```
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=(12,12))
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()
```

Multilabel Classification

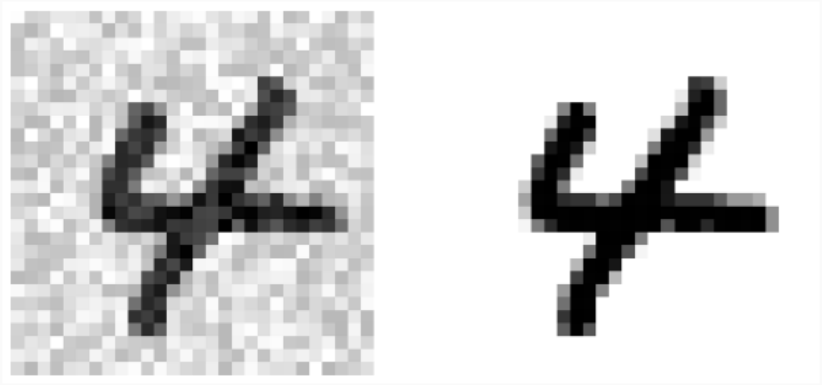


In multilabel classification, the classifier outputs multiple classes for each instance.

Multioutput Classification

- *Multioutput-multiclass classification* (or simply *multioutput classification*)
- A generalization of multilabel classification where each label can be multiclass (i.e., it can have more than 2 possible values).

Example - An image denoising system



- Input: a noisy digit image. Output: a clean digit image, represented as an array of pixel intensities
- Output is multilabel (one label per pixel) and each label can have multiple values (pixel intensity ranges from 0 to 255).

Questions?