Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow

Concepts, Tools, and Techniques to Build Intelligent Systems

Stephen CUI[1]

October 30, 2022

[1] cuixuanStephen@gmail.com

Contents

Chapter 1

Classification

Now we will turn our attention to classification systems.

1.1 MNIST

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. This set has been studied so much that it is often called the “hello world” of machine learning: whenever people come up with a new classification algorithm they are curious to see how it will perform on MNIST, and anyone who learns machine learning tackles this dataset sooner or later.

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

```python
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 (so they don’t need to be downloaded over the internet);
- `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

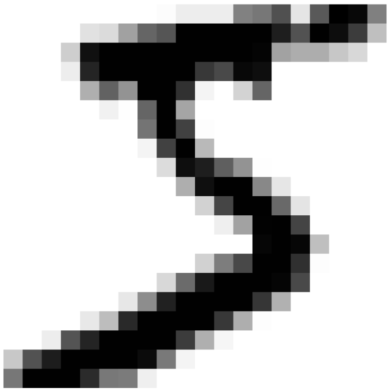

Figure 1.1: Example of an MNIST image

- `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). But the 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. Let's look at these arrays:

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

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). Let's take a peek at one digit from the dataset (**??**).

```
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)
```

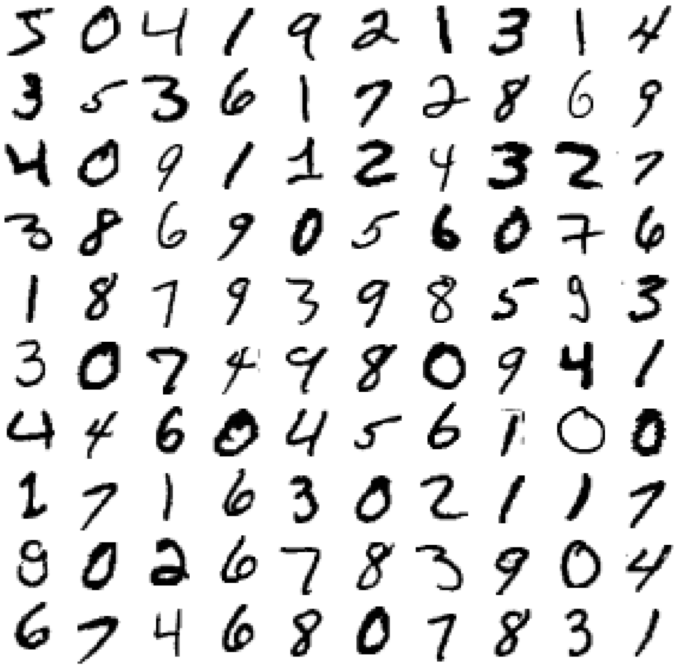

Figure 1.2: Digits from the MNIST dataset

To give you a feel for the complexity of the classification task, **??** shows a few more images from the MNIST dataset.

在你进一步了解数据的时候，你应该将数据切分为训练集和测试集。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)[1]:

```
    X_train, X_test, y_train, y_test = X[:60_000], X[60_000:], y[:60_000], y[60_000:]
```

1.2 Training a Binary Classifier

Let’s simplify the problem for now and only try to identify one digit—for example, the number 5.

[1] Datasets returned by `fetch_openml()` are not always shuffled or split.

```
y_train_5 = (y_train == '5')
y_test_5 = (y_test == '5')
```

Now let's pick a classifier and train it. A good place to start is with a *stochastic gradient descent* (SGD, 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, as you will see later.

```
from sklearn.linear_model import SGDClassifier

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

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

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

1.3 Performance Measures

Evaluating a classifier is often significantly trickier than evaluating a regressor, so we will spend a large part of this chapter on this topic. There are many performance measures available, so grab another coffee and get ready to learn a bunch of new concepts and acronyms!

1.3.1 Measuring Accuracy Using Cross-Validation

A good way to evaluate a model is to use cross-validation, just as you did in **?? ??**.

```
from sklearn.model_selection import cross_val_score

cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring='accuracy')
```

Wow! Above 95% accuracy (ratio of correct predictions) on all cross-validation folds? This looks amazing, doesn't it? Well, before you get too excited, let's look at **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)))
```

```
cross_val_score(dummy_clf, X_train, y_train_5, cv=3, scoring='accuracy')
```

That's right, 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). A much better way to evaluate the performance of a classifier is to look at the *confusion matrix* (CM). 当数据是不平衡时，使用准确率accuracy来评价模型时不合理的。

Implementing Cross-Validation

Occasionally you will need more control over the cross-validation process than what Scikit-Learn provides off the shelf. In these cases, you can implement cross-validation yourself. The following code does roughly the same thing as Scikit-Learn's `cross_val_score()` function, and it prints the same result. The StratifiedKFold class performs stratified sampling (分层抽样) to produce folds that contain a representative ratio of each class. At each iteration the code creates a clone of the classifier, trains that clone on the training folds, and makes predictions on the test fold. Then it counts the number of correct predictions and outputs the ratio of correct predictions.

```
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))
```

1.3.2 Confusion Matrices

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 could make predictions on the test set, but it's best to keep that untouched for now. Instead, you can use the `cross_val_predict()` function:

```python
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 (by “clean” I mean “out-of-sample” : the model makes predictions on data that it never saw during training).

Now you are ready to get the confusion matrix using the `confusion_matrix()` function. Just pass it the target classes (`y_train_5`) and the predicted classes (`y_train_pred`):

```python
from sklearn.metrics import confusion_matrix

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

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*, TN), while the remaining 687 were wrongly classified as 5s (*false positives*, FP, also called *type I errors*). The second row considers the images of 5s (the *positive class*): 1,891 were wrongly classified as non-5s (*false negatives*, FN, also called *type II errors*), while the remaining 3,530 were correctly classified as 5s (*true positives*, TP).

Sometimes you may prefer a more concise metric. An interesting one to look at is the accuracy of the positive predictions; this is called the *precision* of the classifier (**??**).

$$precision = \frac{TP}{TP+FP} \tag{1.1}$$

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 (**??**).

$$recall = \frac{TP}{TP+FN} \tag{1.2}$$

1.3.3 Precision and Recall

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

```
from sklearn.metrics import precision_score, recall_score

print(precision_score(y_train_5, y_train_pred))
print(recall_score(y_train_5, y_train_pred))
```

Now our 5-detector does not look as shiny as it did when we looked at its accuracy. When it claims an image represents a 5, it is correct only 83.7% of the time. Moreover, it only detects 65.1% of the 5s.

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(调和平均数，倒数平均数的倒数) of precision and recall (**??**). Whereas the regular mean treats all values equally, the harmonic mean gives much more weight to low values.

$$F_1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = 2 \times \frac{precision \times recall}{precision + recall} = \frac{TP}{TP + \frac{FN+FP}{2}} \tag{1.3}$$

To compute the $F_1 score$, simply call the `f1_score()` function:

```
from sklearn.metrics import f1_score

print(f1_score(y_train_5, y_train_pred))
```

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.

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

1.3.4 The Precision/Recall Trade-off

To understand this 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. If that score is greater than a threshold, it assigns the instance to the positive class; otherwise it assigns it to the negative class.

Scikit-Learn does not let you set the threshold directly, but it does give you access to the decision scores that it uses to make predictions. Instead of calling the classifier's `predict()` method, you can call its `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])
print(y_scores)
threshold = 0
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
```

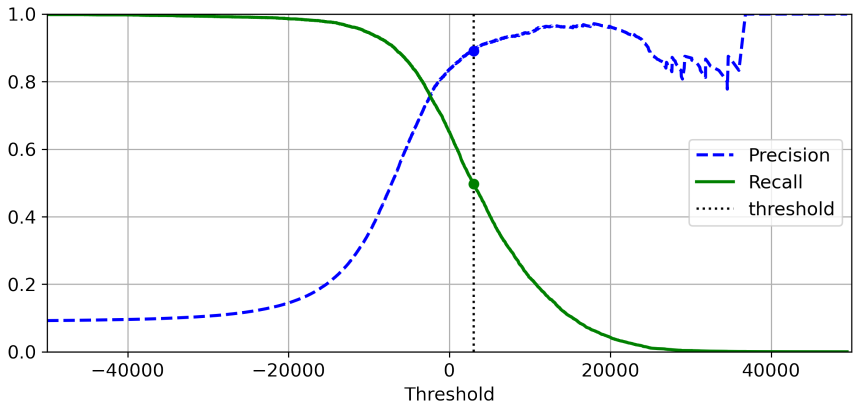

Figure 1.3: Precision and recall versus the decision threshold

The SGDClassifier uses a threshold equal to 0, so the preceding code returns the same result as the `predict()` method (i.e., True).(有些样本的得分确实会小于0) Let's raise the threshold:

```
threshold = 3_000
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
```

This confirms that raising the threshold decreases recall.

那么如何确定阈值呢？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 (the function adds a last precision of 0 and a last recall of 1, corresponding to an infinite threshold):

```
from sklearn.metrics import precision_recall_curve

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

Finally, use Matplotlib to plot precision and recall as functions of the threshold value **??**.

Another way to select a good precision/recall trade-off is to plot precision directly against recall, as shown in **??** (the same threshold is shown):

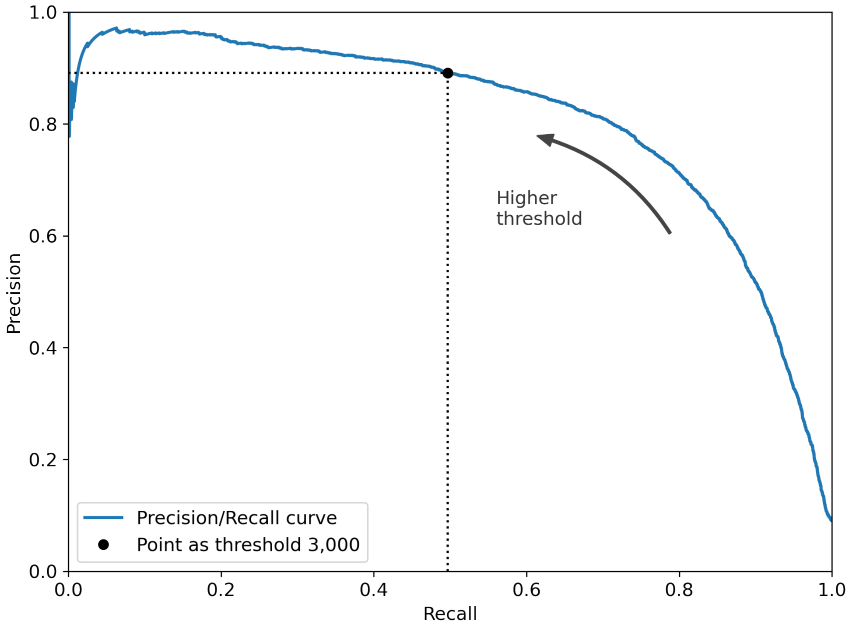

Figure 1.4: Precision versus recall

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—for example, at around 60% recall. But of course, the choice depends on your project.

Suppose you decide to aim for 90% precision. You could use the first plot to find the threshold you need to use, but that's not very precise. Alternatively, 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 >= .9).argmax()
threshold_for_90_precision = thresholds[idx_for_90_precision]
threshold_for_90_precision # 3370.0194991439594
```

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
```

As you can see, it is fairly easy to create a classifier with virtually any precision you want: just set a high enough threshold, and you're done. But wait, not so fast–a high-precision classifier is not very useful if its recall is too low!

Suggestions

If someone says, "Let's reach 99% precision", you should ask, "At what recall?"

1.3.5 The ROC Curve

1.4 Multiclass Classification

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). 整个系统是由多个单类分类器组成，每个分类器可以判断一种class，然后给出概率或者得分，最高值最为结果。

1.4.1

1.4.2

1.5

1.5.1

1.5.2

1.6

1.7

1.8

Chapter 2

Decision Trees

Chapter 3

Ensemble Learning and Random Forests

Chapter 4

Training and Deploying TensorFlow Models at Scale