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

Part I

The Fundamentals of Machine Learning

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:

```
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

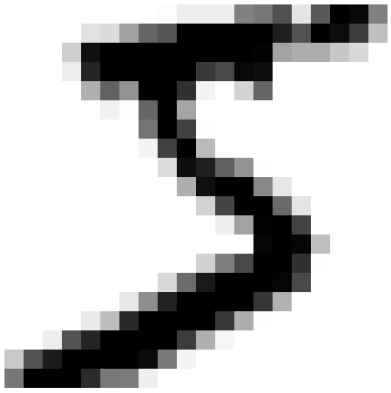

Figure 1.1: Example of an MNIST image

- `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 (Figure 1.1).

```
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)
```

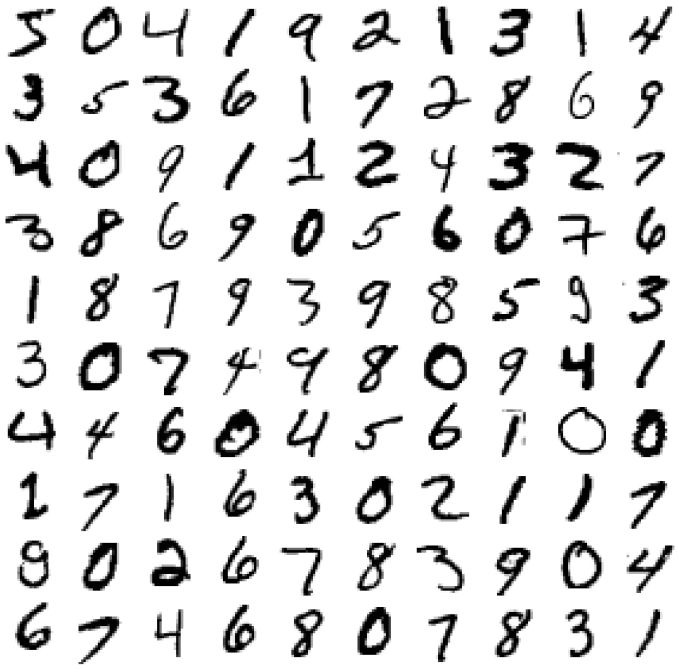

Figure 1.2: Digits from the MNIST dataset

To give you a feel for the complexity of the classification task, Figure 1.2 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:

```
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).

1.3.3 Precision and Recall

1.3.4 The Precision/Recall Trade-off

1.3.5 The ROC Curve

1.4 Multiclass Classification

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

Part II

Neural Networks and Deep Learning

Chapter 4

Training and Deploying TensorFlow Models at Scale