CHAPTER 12

Recognizing Handwritten Digits

So far you have seen how to apply the techniques of data analysis to pandas dataframes containing numbers and strings. However, data analysis is not limited to numbers and strings, because images and sounds can also be analyzed and classified.

In this short but no-less-important chapter, you will learn about handwriting recognition.

Handwriting Recognition

Recognizing handwritten text is a problem that can be traced back to the first automatic machines that needed to recognize individual characters in handwritten documents. Think about, for example, the ZIP codes on letters at the post office and the automation needed to recognize these five digits. Perfect recognition of these codes is necessary in order to sort mail automatically and efficiently.

Included among the other applications that may come to mind is OCR (Optical Character Recognition) software. OCR software must read handwritten text, or pages of printed books, for general electronic documents in which each character is well defined.

But the problem of handwriting recognition goes farther back in time, more precisely to the early 20th century (1920s), when Emanuel Goldberg (1881–1970) began his studies regarding this issue and suggested that a statistical approach would be an optimal choice.

To address this issue in Python, the `scikit-learn` library provides a good example. This library can help you better understand this technique, the issues involved, and the possibility of making predictions.

Recognizing Handwritten Digits with scikit-learn

The `scikit-learn` library (`http://scikit-learn.org/`) enables you to approach this type of data analysis in a way that is slightly different from what you've used in the book so far. The data to be analyzed is closely related to numerical values or strings, but can also involve images and sounds.

The problem you face in this chapter involves predicting a numeric value, and then reading and interpreting an image that uses a handwritten font.

In this case, you have an *estimator* with the task of learning through a `fit()` function, and once it reaches a degree of predictive capability (the model is sufficiently valid), it will produce a prediction with the `predict()` function. The training and validation sets are created this time from a series of images.

This chapter uses Jupyter Notebook to run through the Python code examples, so open Jupyter and create a new Notebook.

An estimator that is useful in this case is `sklearn.svm.SVC`, which uses the technique of Support Vector Classification (SVC).

F. Nelli, *Python Data Analytics*, https://doi.org/10.1007/978-1-4842-9532-8_12

Thus, you have to import the `svm` module of the `scikit-learn` library. You can create an estimator of SVC type and then choose an initial setting, assigning the values `C` and `gamma` generic values. These values can then be adjusted in a different way during the course of the analysis.

```
from sklearn import svm
svc = svm.SVC(gamma=0.001, C=100.)
```

The Digits Dataset

As you saw in Chapter 8, the `scikit-learn` library provides numerous datasets that are useful for testing many problems of data analysis and prediction of the results. Also in this case there is a dataset of images called *Digits*.

This dataset consists of 1,797 images that are 8x8 pixels in size. Each image is a handwritten digit in grayscale, as shown in Figure 12-1.

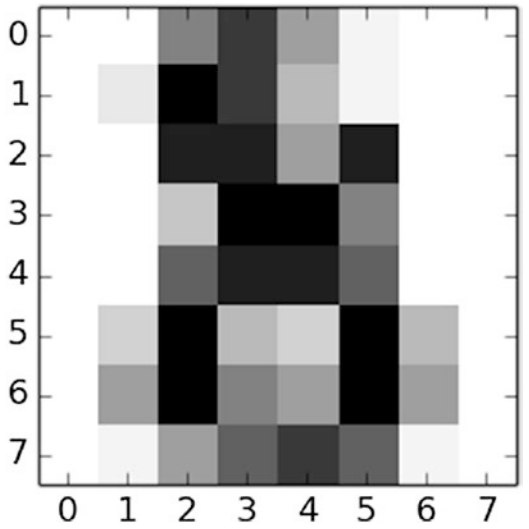

Figure 12-1. *One of 1,797 handwritten number images that make up the Digits dataset*

Thus, you can load the Digits dataset into your Notebook.

```
from sklearn import datasets
digits = datasets.load_digits()
```

After loading the dataset, you can analyze the content. First, you can read lots of information about the datasets by calling the `DESCR` attribute.

```
print(digits.DESCR)
```

For a textual description of the dataset, the authors who contributed to its creation and the references appear as shown in Figure 12-2.

```
print(digits.DESCR)
```

```
.. _digits_dataset:

Optical recognition of handwritten digits dataset
--------------------------------------------------

**Data Set Characteristics:**

    :Number of Instances: 1797
    :Number of Attributes: 64
    :Attribute Information: 8x8 image of integer pixels in the range 0..16.
    :Missing Attribute Values: None
    :Creator: E. Alpaydin (alpaydin '@' boun.edu.tr)
    :Date: July; 1998

This is a copy of the test set of the UCI ML hand-written digits datasets
https://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

The data set contains images of hand-written digits: 10 classes where
each class refers to a digit.

Preprocessing programs made available by NIST were used to extract
normalized bitmaps of handwritten digits from a preprinted form. From a
total of 43 people, 30 contributed to the training set and different 13
to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of
4x4 and the number of on pixels are counted in each block. This generates
an input matrix of 8x8 where each element is an integer in the range
0..16. This reduces dimensionality and gives invariance to small
distortions.

For info on NIST preprocessing routines, see M. D. Garris, J. L. Blue, G.
T. Candela, D. L. Dimmick, J. Geist, P. J. Grother, S. A. Janet, and C.
L. Wilson, NIST Form-Based Handprint Recognition System, NISTIR 5469,
```

Figure 12-2. *Each dataset in the scikit-learn library has a field containing all the information*

The images of the handwritten digits are contained in a `digits.images` array. Each element in this array is an image that is represented by an 8x8 matrix of numerical values that correspond to grayscale array. White has a value of 0 and black has a value of 15.

```
digits.images[0]
```

You will get the following result:

```
array([[  0.,   0.,   5.,  13.,   9.,   1.,   0.,   0.],
       [  0.,   0.,  13.,  15.,  10.,  15.,   5.,   0.],
       [  0.,   3.,  15.,   2.,   0.,  11.,   8.,   0.],
       [  0.,   4.,  12.,   0.,   0.,   8.,   8.,   0.],
       [  0.,   5.,   8.,   0.,   0.,   9.,   8.,   0.],
```

```
       [  0.,   4.,  11.,   0.,   1.,  12.,   7.,   0.],
       [  0.,   2.,  14.,   5.,  10.,  12.,   0.,   0.],
       [  0.,   0.,   6.,  13.,  10.,   0.,   0.,   0.]])
```

You can visually check the contents of this result using the `matplotlib` library.

```
import matplotlib.pyplot as plt
plt.imshow(digits.images[0], cmap=plt.cm.gray_r, interpolation='nearest')
```

When you launch this command, you obtain the grayscale image shown in Figure 12-3.

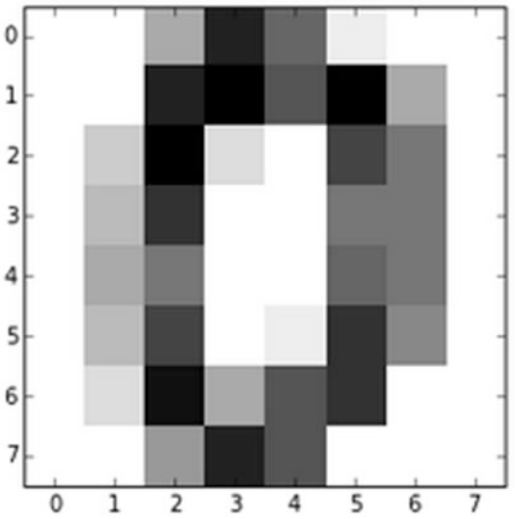

Figure 12-3. *One of the 1,797 handwritten digits*

The numerical values represented by images, that is, the targets, are contained in the `digits.target` array.

```
digits.targetOut [ ]:
array([0, 1, 2, ..., 8, 9, 8])
```

It was reported that the dataset is a training set consisting of 1,797 images. You can determine if that is true.

```
digits.target.sizeOut [ ]:
1797
```

Learning and Predicting

Now that you have loaded the Digits dataset into your Notebook and have defined an SVC estimator, you can start learning.

As you learned in Chapter 8, once you define a predictive model, you must instruct it with a training set, which is a set of data in which you already know the class. Given the large quantity of elements contained in the Digits dataset, you will certainly obtain a very effective model, that is, one that's capable of recognizing with good certainty the handwritten number.

This dataset contains 1,797 elements, so you can consider the first 1,791 as a training set and use the last 6 as a validation set.

You can see in detail these six handwritten digits by using the `matplotlib` library:

```
import matplotlib.pyplot as plt
plt.subplot(321)
plt.imshow(digits.images[1791], cmap=plt.cm.gray_r, interpolation='nearest')
plt.subplot(322)
plt.imshow(digits.images[1792], cmap=plt.cm.gray_r, interpolation='nearest')
plt.subplot(323)
plt.imshow(digits.images[1793], cmap=plt.cm.gray_r, interpolation='nearest')
plt.subplot(324)
plt.imshow(digits.images[1794], cmap=plt.cm.gray_r, interpolation='nearest')
plt.subplot(325)
plt.imshow(digits.images[1795], cmap=plt.cm.gray_r, interpolation='nearest')
plt.subplot(326)
plt.imshow(digits.images[1796], cmap=plt.cm.gray_r, interpolation='nearest')
```

This will produce an image with six digits, as shown in Figure 12-4.

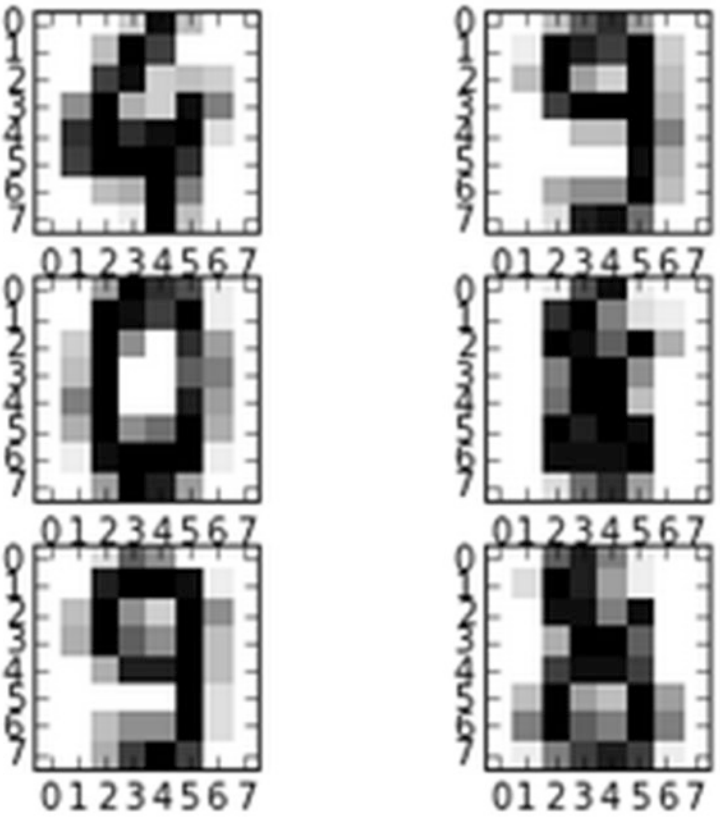

Figure 12-4. *The six digits of the validation set*

Now you can train the `svc` estimator that you defined earlier.

```
svc.fit(digits.data[1:1790], digits.target[1:1790])
```

This will produce an image as shown in Figure 12-5.

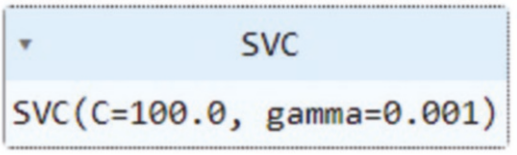

Figure 12-5. *The parameters of the SVC estimator*

Now you have to test your estimator, making it interpret the six digits of the validation set.

```
svc.predict(digits.data[1791:1976])
Out [ ]: array([4, 9, 0, 8, 9, 8])
```

If you compare them with the actual digits, as follows:

```
digits.target[1791:1976]
Out [ ]:
array([4, 9, 0, 8, 9, 8])
```

You can see that the `svc` estimator has learned correctly. It recognizes the handwritten digits, interpreting correctly all six digits of the validation set.

Recognizing Handwritten Digits with TensorFlow

You have just seen an example of how machine learning techniques can recognize handwritten numbers. Now the same problem is applied to the deep learning techniques that you used in Chapter 9. As was the case in Chapter 9, the following section regarding TensorFlow has been completely rewritten from the previous edition. In fact, here too you will use the new TensorFlow 2.x version, which is completely different from TensorFlow 1.x. The code used here is therefore not present in older editions of this book.

Given the great value of the MNIST dataset, the TensorFlow library also contains a copy of it. It will therefore be very easy to perform studies and tests on neural networks with this dataset, without having to download or import them from other data sources.

In addition to TensorFlow, install the `tensorflow-dataset` package. You can do this either using Anaconda Navigator or via the command console:

```
conda install tensorflow-dataset
```

If you don't have the Anaconda platform, you can install the package through the PyPI system.

```
pip install tensorflow-dataset
```

Importing the MNIST dataset into the Jupyter Notebook (in any Python session) is very simple. Indeed, with the new version of TensorFlow 2.x, there is no need to import test datasets like MNIST from other libraries, as they are available within Keras, which is integrated within the `tensorflow` module you already imported. You can simply import the libraries like `numpy` and `matplotlib` along with `tensorflow`, which also contains the MNIST dataset.

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
ì
```

Now load the dataset directly into your Notebook by simply writing the following line of code.

```
(x_train, y_train),(x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

Then you can load the dataset directly into your Notebook.

```
x_validation = x_train[55000:]
x_train = x_train[:55000]
y_validation = y_train[55000:]
y_train = y_train[:55000]
len(x_train)
Out [ ]:
55000

len(x_test)
Out [ ]:
10000

len(x_validation)
Out [ ]:
5000
```

The MNIST data is split into three parts: 55,000 data points of training data (`x_train`), 10,000 points of test data (`x_test`), and 5,000 points of validation data (`x_validation`).

All this data will be submitted to the model: `x` is the feature dataset and `y` is the label dataset. As you saw earlier, these are images of handwritten letters. You can look at the first image of the training dataset (features).

```
x_train[0].shape
Out [ ]:
(28, 28)

This is a square image of 28 pixels per side.

plt.imshow(x_train[0], cmap=plt.cm.gray_r, interpolation='nearest')
```

You will get the black and white image of a handwritten number, similar to the one shown in Figure 12-6.

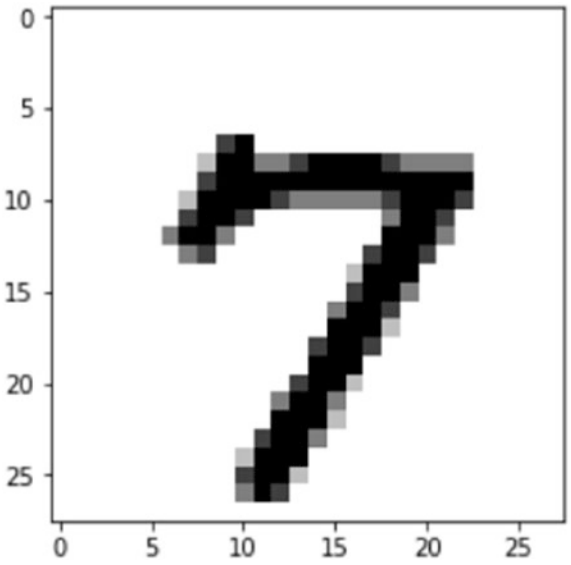

Figure 12-6. *A digit of the training set in the MNIST dataset provided by the TensorFlow library*

To give you an idea of the contents of the MNIST dataset, a better visualization is the following:

```
fig, ax = plt.subplots(10, 10)
k = 0
for i in range(10):
    for j in range(10):
        ax[i][j].imshow(x_train[k].reshape(28, 28),
                        cmap=plt.cm.gray_r,
                        interpolation='nearest',
                        aspect='auto')
        ax[i][j].set_xticks([])
        ax[i][j].set_yticks([])
        k += 1
```

Running the code, you will get a pattern of 100 numbers in the dataset, as shown in Figure 12-7.

Figure 12-7. *100 digits of the training dataset from MNIST dataset provided by the TensorFlow library*

Because these are images and therefore two-dimensional arrays, as you learned in Chapter 9, you have to use a flatten layer at the beginning of the neural network to flatten the input data and make them one-dimensional.

As for the types of values to be submitted to the neural network, you have integer values ranging from 0 to 255.

```
print(np.max(x_train))
np.min(x_train)
Out [ ]:
255
0
```

In fact, these are grayscale images which, like RGB colors, are included in a range of values between 0 and 255. You therefore also have to add a normalization layer to the neural network model.

Now convert all the arrays to tensors for use in TensorFlow.

```
train_features = tf.convert_to_tensor(x_train)
train_labels = tf.convert_to_tensor(y_train)
test_features = tf.convert_to_tensor(x_test)
test_labels =  tf.convert_to_tensor(y_test)
exp_features = tf.convert_to_tensor(x_validation)
```

Let's look at the characteristics of one of the tensors as an example.

```
train_features
Out [ ]:
<tf.Tensor: shape=(55000, 28, 28), dtype=uint8, numpy=
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]],
...
```

The results obtained are what you would expect.

Learning and Predicting with an SLP

Now that you've seen how to get the training set, the testing set, and the validation set with TensorFlow, it's time to do an analysis with a neural network, very similar to the one you used in Chapter 9. Let's start by using a Single Layer Perceptron (SLP).

First define a model with a single dense layer with ten outputs corresponding to the ten numerical digits ranging from 0 to 9, and which correspond to the ten classes of membership of the handwritten digits to be identified. To this single layer, you will add the two layers: `Normalization` and `Flatten`. The former will normalize the pixel values of the images from 0 to 255 in the range of 0 to 1, and the latter will convert the two-dimensional array of 28x28 images into a single one-dimensional array.

```
model = tf.keras.Sequential([
    tf.keras.layers.Normalization(),
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(10, activation='sigmoid')
])
```

Once the model has been defined, you can compile it, setting Adam as an optimizer and `sparse_categorical_crossentropy` as a function. Then you can start learning the model with 20 epochs.

```
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
h = model.fit(train_features, train_labels, epochs=20)
Out [ ]:
Epoch 1/20
1719/1719 [==============================] - 3s 1ms/step - loss: 10.6817 - accuracy: 0.8341
Epoch 2/20
1719/1719 [==============================] - 2s 1ms/step - loss: 6.1368 - accuracy: 0.8759
Epoch 3/20
1719/1719 [==============================] - 3s 2ms/step - loss: 5.7782 - accuracy: 0.8798
Epoch 4/20
1719/1719 [==============================] - 3s 1ms/step - loss: 5.5405 - accuracy: 0.8824
```

```
Epoch 5/20
1719/1719 [==============================] - 2s 1ms/step - loss: 5.4812 - accuracy: 0.8840
Epoch 6/20
1719/1719 [==============================] - 2s 1ms/step - loss: 5.3873 - accuracy: 0.8851
Epoch 7/20
1719/1719 [==============================] - 2s 1ms/step - loss: 5.3120 - accuracy: 0.8861
...
```

Now check the learning phase through the history by graphically monitoring the trend of the loss.

```
acc_set = h.history['loss']
epoch_set = h.epoch
plt.plot(epoch_set,acc_set, 'o', label='Training phase')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend()
```

Running the code, you get a plot like the one shown in Figure 12-8.

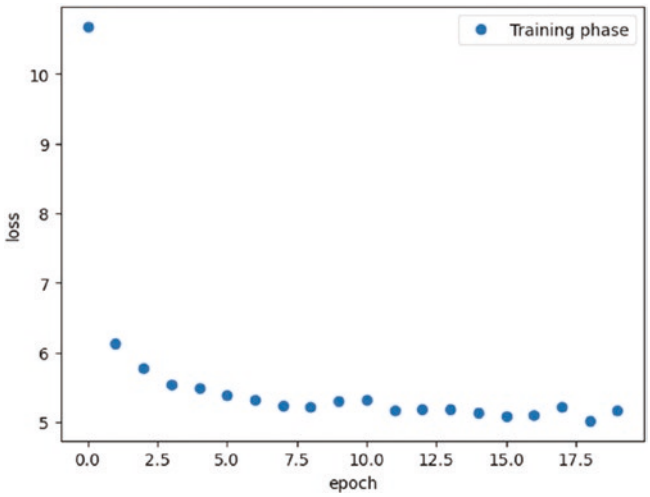

Figure 12-8. *The loss trend during the learning phase of the SLP neural network*

You can also evaluate the model numerically using the following line of code:

```
model.evaluate(test_features, test_labels)
Out [ ]:
313/313 [==============================] - 1s 1ms/step - loss: 5.8654 - accuracy: 0.8925
[5.865407466888428, 0.8924999833106995]
```

As you can see from the numerical values, an accuracy of 0.89 is not optimal and the loss value does not seem to drop too much, stabilizing at a value of 5.86.

Let's see how this model can recognize handwritten numbers that have not been used for learning or for testing. For this purpose, a third dataset has been set aside: `exp_features`. You extend the model with the Softmax layer to get the probabilities of belonging to the various classes as a result. You then let the newly educated SLP model make the predictions.

```
probability_model = tf.keras.Sequential([
    model,
    tf.keras.layers.Softmax()
])
predictions = probability_model.predict(exp_features)
Out [ ]:
157/157 [==============================] - 0s 981us/step
```

Now take the first image to be predicted, with the probabilities of recognition at each of the ten numerical digits.

```
predictions[0]
Out [ ]:
array([0.04717345, 0.12823072, 0.12823072, 0.12823072, 0.1282307 ,
       0.08905466, 0.04721416, 0.04717345, 0.12823072, 0.12823072],
      dtype=float32)
```

From the list of ten probabilities in the output, the situation is not so legible. If you use a graphical approach, representing the various probabilities in a barplot, you get better results.

```
p = plt.bar(np.arange(10),predictions[0])
plt.xticks(np.arange(10))
predicted_label = np.argmax(predictions[0])
p[predicted_label].set_color('red')
```

Running the previous code will give you a barplot similar to the one shown in Figure 12-9.

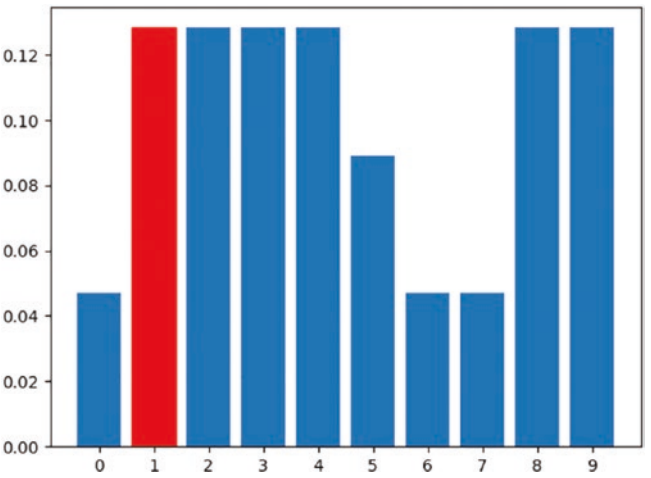

Figure 12-9. *The loss trend during the learning phase of the SLP neural network*

You can immediately see that many digits have the same probability of being the one represented in the image. Although the barplot shows the most probable figure in red, in this case there is an error since many other figures have the same probability (the graph shows only the first maximum in the case of parity of values). So the forecast was not successful. Now determine the true value of the image submitted to the model.

```
y_validation[0]
Out [ ]:
1
```

You can also look at it graphically.

```
plt.imshow(x_validation[0], cmap=plt.cm.gray_r, interpolation='nearest')
```

Executing the previous code, you obtain the number shown in Figure 12-10.

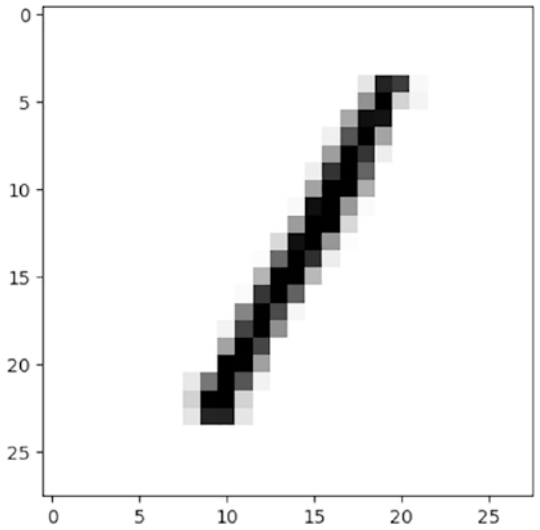

Figure 12-10. *The image shows the handwritten number 1*

As you can see, it is the number 1, which is present among the most probable results. However, there are too many probable options, so you cannot consider this a good prediction. Now take a number that is easier to recognize and see if the SLP model can recognize it correctly.

Choose the 14th number, which is easily recognizable.

```
plt.imshow(x_validation[13], cmap=plt.cm.gray_r, interpolation='nearest')
```

By running the code, you will get the image of this easily recognizable number, as shown in Figure 12-11.

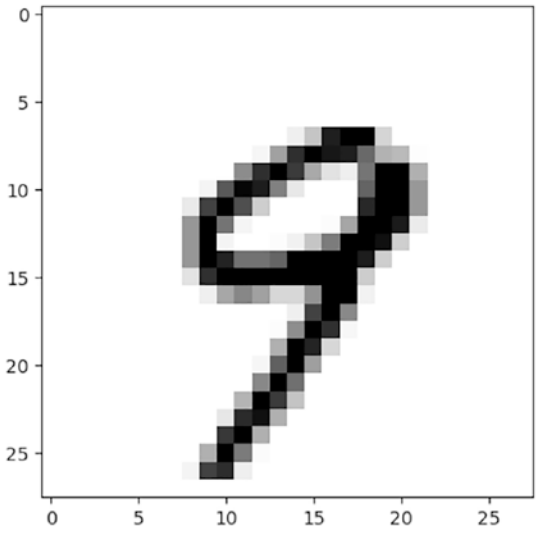

Figure 12-11. *The image shows the handwritten number 9*

The image clearly shows the number 9. Now check the corresponding label.

```
y_validation[13]
Out [ ]:
9
```

Let's see, in this very simple case, if the model recognized the number 9 clearly.

```
predictions[13]
Out [ ]:
array([0.07372559, 0.07270355, 0.07317524, 0.07424378, 0.13360192,
       0.07305884, 0.07261127, 0.13158722, 0.09798288, 0.1973097 ],
      dtype=float32)
```

You can also look at this graphically.

```
p = plt.bar(np.arange(10),predictions[13])
plt.xticks(np.arange(10))
predicted_label = np.argmax(predictions[13])
p[predicted_label].set_color('red')
```

Running the code will result in a barplot like the one shown in Figure 12-12.

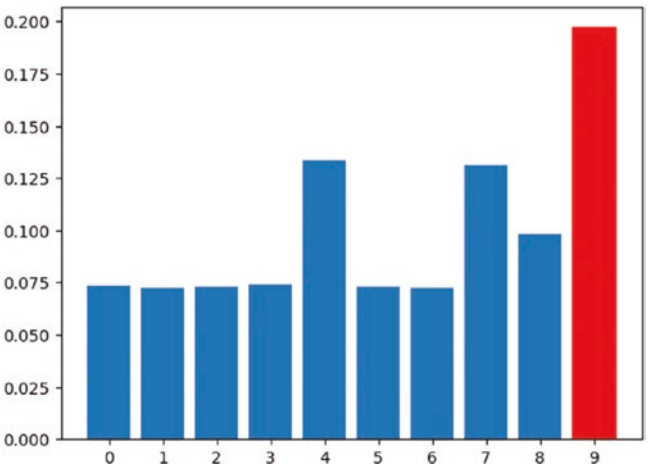

Figure 12-12. *The barplot shows the 9 digit as the most probable result (red bar)*

Although it guessed the number correctly this time, giving it a probability of around 20 percent is certainly not a good prediction. The number is easily recognizable, and it should have a much higher probability of recognition than the other digits.

Learning and Predicting with an MLP

Given the moderate success of the SLP model, this section builds a much more complex neural network, with more layers and more neurons in play. This network uses a Multiple Layer Perceptron (MLP) model with a hidden layer to predict handwritten digits. Furthermore, it brings the number of neurons of the first layer to 256, to which you will add another 128 for the hidden layer. It leaves the output layer unchanged at ten neurons (the ten digits to be classified).

```
model = tf.keras.Sequential([
    tf.keras.layers.Normalization(),
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(256, activation='sigmoid'),
    tf.keras.layers.Dense(128, activation='sigmoid'),
    tf.keras.layers.Dense(10, activation='sigmoid')
])
```

You can compile the model and train it with the same number of epochs as the previous one (20).

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
h = model.fit(train_features, train_labels, epochs=20)
Out [ ]:
Epoch 1/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.4885 - accuracy: 0.8658
Epoch 2/20
1719/1719 [==============================] - 4s 2ms/step - loss: 0.3243 - accuracy: 0.9013
```

```
Epoch 3/20
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2935 - accuracy: 0.9093
Epoch 4/20
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2630 - accuracy: 0.9171
Epoch 5/20
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2423 - accuracy: 0.9250
Epoch 6/20
1719/1719 [==============================] - 3s 2ms/step - loss: 0.2370 - accuracy: 0.9268
Epoch 7/20
...
```

You can also graphically see the trend of the loss during the learning phase of the model.

```
acc_set = h.history['loss']
epoch_set = h.epoch
plt.plot(epoch_set,acc_set, 'o', label='Training phase')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend()
```

By executing the previous code, you obtain a plot like the one shown in Figure 12-13.

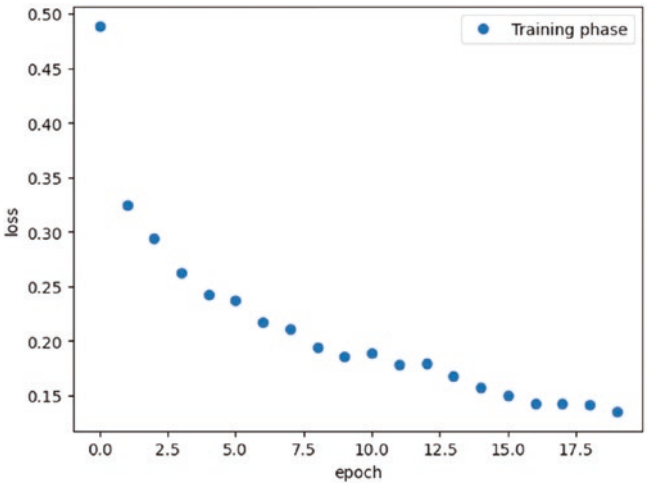

Figure 12-13. *The image shows how the value of the loss is optimized during the training phase*

You can also evaluate the model learning process numerically.

```
model.evaluate(test_features, test_labels)
Out [ ]:
313/313 [==============================] - 1s 1ms/step - loss: 0.1537 - accuracy: 0.9511
[0.1537058800458908, 0.9510999917984009]
```

As you can clearly see, this time the learning, in addition to being regular, also reaches a good accuracy value and a low loss value. You could also increase the number of epochs during the training phase to further increase the predictive power of this MLP model. However, leave things unchanged to compare the performance of this model to the previous one.

Now you can see how this model recognizes numbers using the same two images that you submitted to the SLP model.

```
probability_model = tf.keras.Sequential([
    model,
    tf.keras.layers.Softmax()
])
predictions = probability_model.predict(exp_features)
Out [ ]:
157/157 [==============================] - 0s 1ms/step
```

First you saw how the model assigns the probabilities of belonging to the ten digits of the image with the number 1.

```
predictions[0]
array([0.07896608, 0.21424502, 0.08096407, 0.07952367, 0.08464722,
       0.07919335, 0.08132026, 0.11285783, 0.10845622, 0.07982624],
      dtype=float32)
```

You can represent these graphically in a barplot.

```
p = plt.bar(np.arange(10),predictions[0])
plt.xticks(np.arange(10))
predicted_label = np.argmax(predictions[0])
p[predicted_label].set_color('red')
```

Running this code, you get the barplot shown in Figure 12-14.

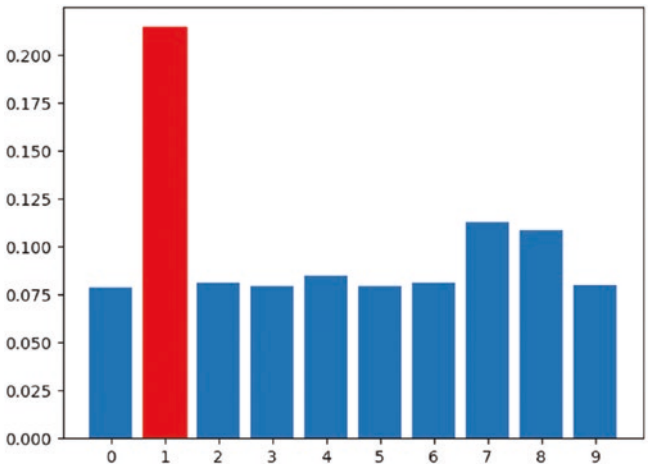

Figure 12-14. *The barplot shows the 1 digit as the most probable result (red bar)*

As you can see, this model guessed the number represented in the image, with a much higher probability than the other digits. This time, the number 1 has been clearly identified.

If you carry out the same operations with the second image (representing the number 9), you will get a similar result, as shown in the barplot in Figure 12-15.

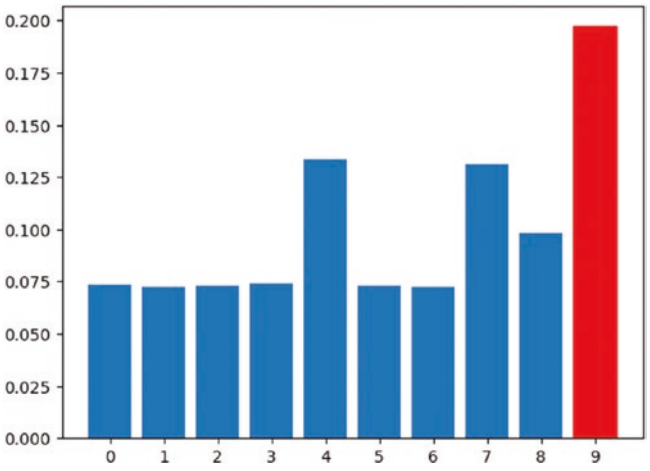

Figure 12-15. *The barplot shows the 1 digit as the most probable (red bar)*

It is therefore clear that a more complex neural network model such, as the one used in the example, is more efficient: it learns faster and is more adept at identifying the handwritten numbers in the images. But it is not always true that a more complex neural network model leads to an increase in potential. Only an adequate study of the various models, the optimizations used, the loss functions chosen, and all the other parameters used can prove the accuracy of a model. I therefore invite you to study this topic further, if you are fascinated by it.

Conclusions

In this chapter, you learned how many application possibilities this data analysis process has. It is not limited only to the analysis of numerical and textual data, but also can analyze images, such as the handwritten digits read by a camera or a scanner.

Furthermore, you have seen that predictive models can provide optimal results, thanks to machine learning and deep learning techniques, which are powerful analysis tools thanks to libraries such as TensorFlow.