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. Indeed, 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'll 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 to better understand this technique, the issues involved, and the possibility of making predictions.

© Fabio Nelli 2018
F. Nelli, *Python Data Analytics*, https://doi.org/10.1007/978-1-4842-3913-1_12

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

So even in this case you will have an *estimator* with the task of learning through a fit() function, and once it has reached a degree of predictive capability (a model sufficiently valid), it will produce a prediction with the predict() function. Then we will discuss the training set and validation set, created this time from a series of images.

Now open a new IPython Notebook session from the command line by entering the following command:

```
ipython notebook
```

Then create a new Notebook by choosing New ➤ Python 3, as shown in Figure 12-1.

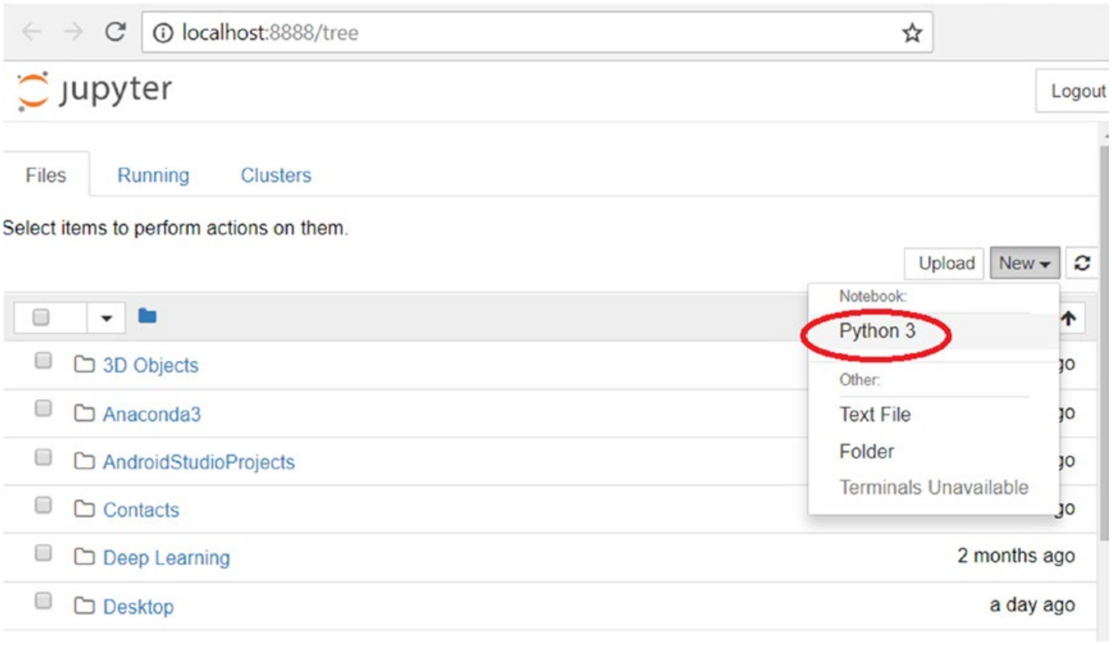


Figure 12-1. *The home page of the IPython Notebook (Jupyter)*

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

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 12-2.

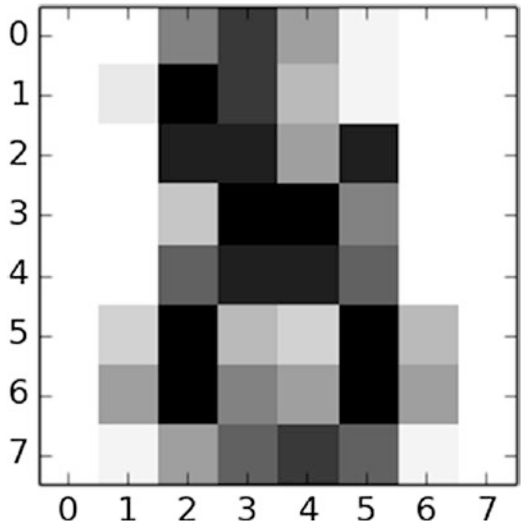


Figure 12-2. *One of 1,797 handwritten number images that make up the dataset digit*

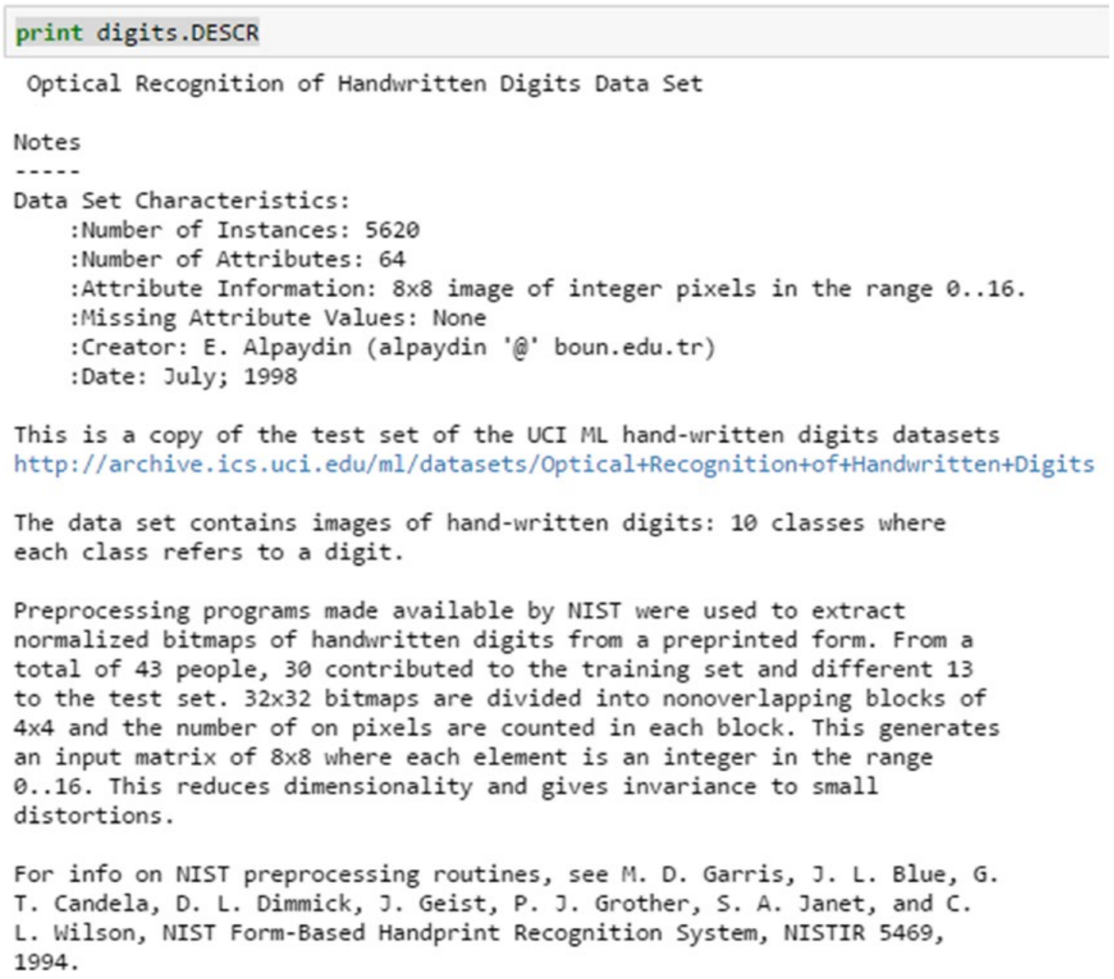
Thus, you can load the Digits dataset into your Notebook.

```
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 will appear as shown in Figure 12-3.



```
print digits.DESCR
```

```
 Optical Recognition of Handwritten Digits Data Set

Notes
-----
Data Set Characteristics:
    :Number of Instances: 5620
    :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
http://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,
1994.
```

Figure 12-3. *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 of this array is an image that is represented by an 8x8 matrix of numerical values that correspond to a grayscale from white, with a value of 0, to black, with the value 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
%matplotlib inline
plt.imshow(digits.images[0], cmap=plt.cm.gray_r, interpolation='nearest')
```

By launching this command, you will obtain the grayscale image shown in Figure 12-4.

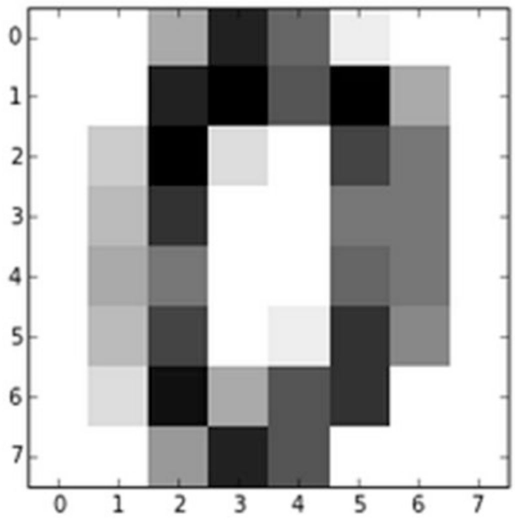


Figure 12-4. *One of the 1,797 handwritten digits*

The numerical values represented by images, i.e., the targets, are contained in the `digit.targets` array.

```
digits.target
```

You will get the following result:

```
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.size
```

This will be the result:

```
1797
```

Learning and Predicting

Now that you have loaded the Digits datasets 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 belonging class. Given the large quantity of elements contained in the Digits dataset, you will certainly obtain a very effective model, i.e., one that's capable of recognizing with good certainty the handwritten number.

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

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

```
import matplotlib.pyplot as plt
%matplotlib inline

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 12-5.

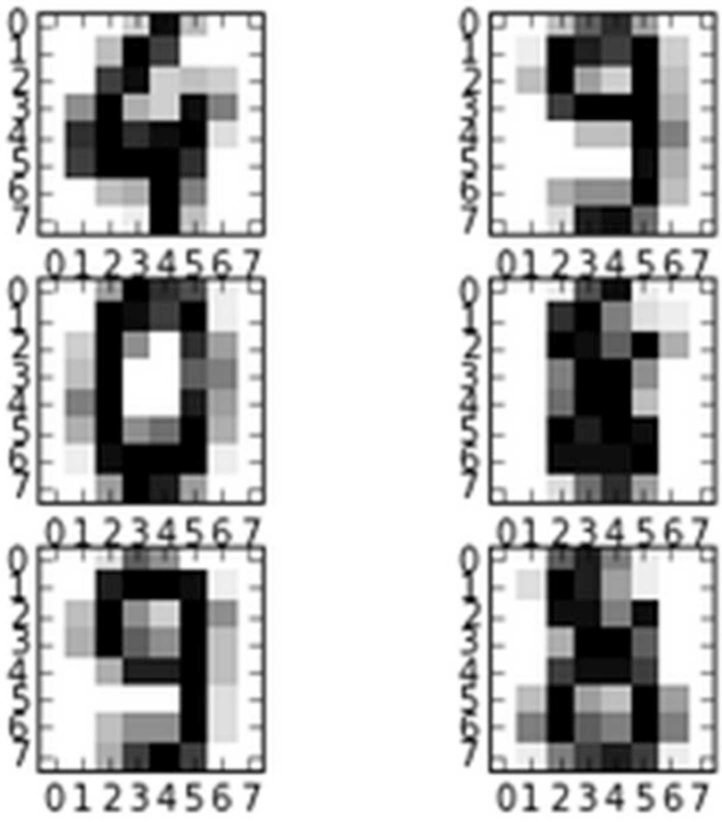


***Figure 12-5.** The six digits of the validation set*

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

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

After a short time, the trained estimator will appear with text output.

```
SVC(C=100.0, cache_size=200, class_weight=None, coef0=0.0, degree=3,
  gamma=0.001, kernel='rbf', max_iter=-1, probability=False,
  random_state=None, shrinking=True, tol=0.001, verbose=False)
```

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

```
svc.predict(digits.data[1791:1976])
```

You will obtain these results:

```
array([4, 9, 0, 8, 9, 8])
```

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

```
digits.target[1791:1976]

array([4, 9, 0, 8, 9, 8])
```

You can see that the `svc` estimator has learned correctly. It is able to recognize 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 will be applied to the deep learning techniques that we used in Chapter 9.

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

Importing the MNIST dataset into the Jupyter Notebook (in any Python session) is very simple; you just import `tensorflow.contrib.learn.python.learn.datasets.mnist` directly like any other Python package. To load the dataset in a variable, you must use the `read_data_sets()` function. Thus, an optimal form for importing the dataset is as follows.

```
from tensorflow.contrib.learn.python.learn.datasets.mnist import read_data_
sets
import numpy as np
import matplotlib.pyplot as plt
```

Since the dataset is contained in compressed files, these will automatically be downloaded to the session workspace as soon as you call the `read_data_sets` function. A good practice is to create a directory that contains them as `MNIST_data`.

```
mnist = read_data_sets ("MNIST_data/", one_hot=False)
```

In the output, the downloaded files will be shown as follows:

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

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

Given the large size of this dataset, a good practice is to break it into smaller batches, especially when it needs to be analyzed as a training set. To help you do this, TensorFlow uses the `next_batch(n)` function, which allows you to extract *n* elements from the training set. Whenever the `next_batch(n)` function is called, the *n* following elements will be extracted, until the end of the training set is reached.

To view the first 10 elements of the training set, enter the following code.

```
pixels,real_values = mnist.train.next_batch(10)
print("list of values loaded",real_values)
list of values loaded  [2 6 8 3 4 2 0 9 8 7]
```

By releasing the same code, you will get the following 10 elements of the training set, and so on.

```
pixels,real_values = mnist.train.next_batch(10)
print("list of values loaded",real_values)
list of values loaded  [6 1 8 5 0 1 8 4 7 3]
```

If you want to see the image of the handwritten digit of one of the elements contained in pixels (an array containing grayscale images), you can use matplotlib.

```
image = np.reshape(pixels[1,:],[28,28])
plt.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
plt.show()
```

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

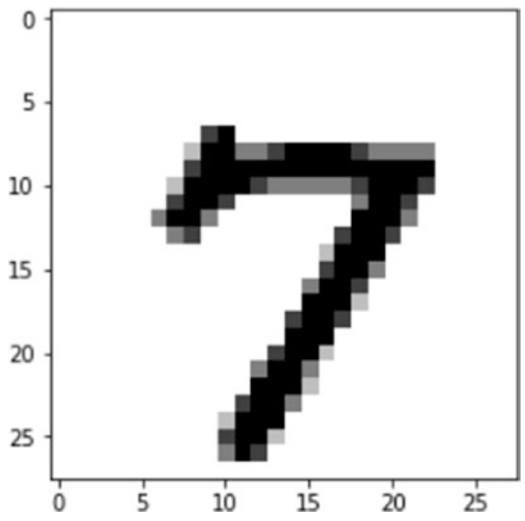


Figure 12-6. *A digit of the training set in the MNIST dataset provided by the TensorFlow library*

Learning and Predicting

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.

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

import tensorflow as tf
import matplotlib.pyplot as plt
```

```
# Parameters
learning_rate = 0.01
training_epochs = 25
batch_size = 100
display_step = 1

# tf Graph Input
x = tf.placeholder("float", [None, 784]) # mnist data image of shape
28*28=784
y = tf.placeholder("float", [None, 10]) # 0-9 digits recognition => 10 classes

# Create model

# Set model weights
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

evidence = tf.matmul(x, W) + b

# Construct model
activation = tf.nn.softmax(evidence) # Softmax

# Minimize error using cross entropy
cross_entropy = y*tf.log(activation)
cost = tf.reduce_mean(-tf.reduce_sum(cross_entropy,reduction_indices=1))

optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)

#Plot settings
avg_set = []
epoch_set=[]

# Initializing the variables
init = tf.global_variables_initializer()

# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    # Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            sess.run(optimizer, feed_dict={x: batch_xs, y: batch_ys})
            avg_cost += sess.run(cost, feed_dict={x: batch_xs, y: batch_
            ys})/total_batch
        if epoch % display_step == 0:
            print("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".
            format(avg_cost))
        avg_set.append(avg_cost)
        epoch_set.append(epoch+1)
    print("Training phase finished")

    plt.plot(epoch_set,avg_set, 'o', label='Logistic Regression Training
    phase')
    plt.ylabel('cost')
    plt.xlabel('epoch')
    plt.legend()
    plt.show()

    # Test model
    correct_prediction = tf.equal(tf.argmax(activation, 1), tf.argmax(y, 1))
    # Calculate accuracy
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    print("Model accuracy:", accuracy.eval({x: mnist.test.images, y: mnist.
    test.labels}))
```

By performing the analysis, you will obtain the cost trend during the learning phase (epoch cycle) and when the neural network will be properly instructed, the testing set `mnist.test` will be evaluated. The value of the accuracy obtained will tell you the percentage of numbers read and correctly interpreted by the neural network.

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
Epoch: 0001 cost= 1.176361134
Epoch: 0002 cost= 0.662538510
Epoch: 0003 cost= 0.550689667
Epoch: 0004 cost= 0.496738935
Epoch: 0005 cost= 0.463713668
Epoch: 0006 cost= 0.440845339
Epoch: 0007 cost= 0.423968329
Epoch: 0008 cost= 0.410662182
Epoch: 0009 cost= 0.399876185
Epoch: 0010 cost= 0.390923975
Epoch: 0011 cost= 0.383305770
Epoch: 0012 cost= 0.376747700
Epoch: 0013 cost= 0.371062683
Epoch: 0014 cost= 0.365925885
Epoch: 0015 cost= 0.361331244
Epoch: 0016 cost= 0.357197133
Epoch: 0017 cost= 0.353523670
Epoch: 0018 cost= 0.350157993
Epoch: 0019 cost= 0.347037680
Epoch: 0020 cost= 0.344143576
Epoch: 0021 cost= 0.341464736
Epoch: 0022 cost= 0.338996708
Epoch: 0023 cost= 0.336639690
Epoch: 0024 cost= 0.334515039
Epoch: 0025 cost= 0.332482831
Training phase finished

Model accuracy: 0.9143
```

From the data obtained, and observing Figure 12-7, you can see that the learning phase of the neural network has been completed and has an expected trend.
The accuracy value of 0.91 (91%) indicates that the model you chose works quite satisfactorily (not completely).

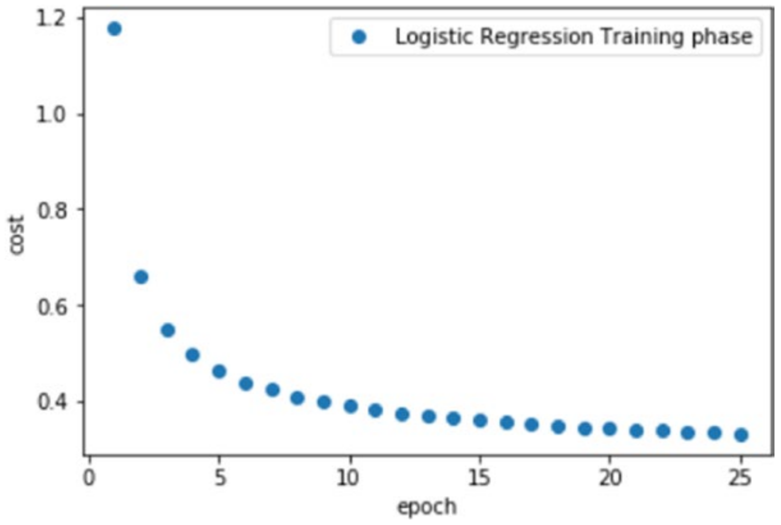


Figure 12-7. *The cost trend during the learning phase of the neural network*

Conclusions

In this short chapter, you learned how many application possibilities this data analysis process has. It is not limited only to the analysis of numerical or 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 truly optimal results thanks to machine learning and deep learning techniques, which are easily implemented thanks to the `scikit-learn` library.