Running code in parallel with multiprocessing

It's common to need to execute more than one thing in parallel in a modern software system. Machine learning programs and scientific simulations benefit from using the multiple cores available in a modern processor, dividing their work up between concurrent threads operating on parallel hardware. Graphical user interfaces and network servers do their work in the background, leaving a thread available to respond to user events or new requests.

As a simple example, suppose your program had to execute three steps: **A**, **B**, and **C**. These steps are not dependent on each other, meaning they can be completed in any order. Usually, you would simply execute them in order, as follows:

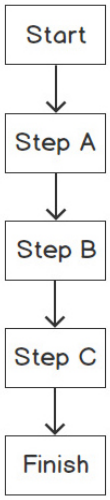


Figure 9.14 – Processing with a single thread

However, what if you could do all of these steps at the same time, rather than waiting for one to complete before moving on to the next one? Our workflow would look as follows:

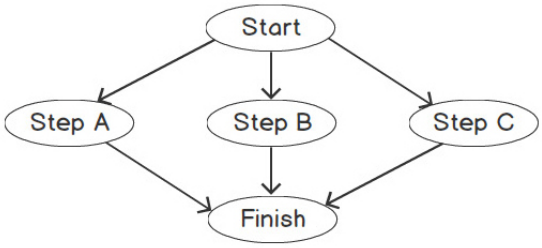


Figure 9.15 – Multithreaded processing

This has the potential to be a lot faster if you have the infrastructure to execute these steps at the same time. That is, each step will need to be executed by a different thread.

Python itself uses multiple threads to do some work internally, which puts some limits on the ways in which a Python program can do multiprocessing. The three safest ways to work are as follows:

- Find a library that solves your problem and handles multiprocessing for you (which has been carefully tested)
- Launch a new Python interpreter by running another copy of your script as a completely separate process
- Create a new thread within the existing interpreter to do some work concurrently

The first of these is the easiest and the most likely to be a success. The second is fairly simple and imposes the most overhead on your computer, as the operating system is now running two independent Python scripts. The third is very complicated, easy to get wrong, and still creates a lot of overhead, as Python maintains a **Global Interpreter Lock** (**GIL**), which means that only one thread at a time can interpret a Python instruction. A quick rule of thumb to choose between the three approaches is to always pick the first one. If a library doesn't exist to address your needs, then pick the second. If you absolutely need to share memory between the concurrent processes, or if your concurrent work is related to handling I/O, then you can choose the third carefully.

Multiprocessing with execnet

It's possible to launch a new Python interpreter with the standard library's `subprocess` module. However, doing so leaves a lot of work up to you about what code to run and how to share data between the "parent" and "child" Python scripts.

An easier interface is the `execnet` library. `execnet` makes it very easy to launch a new Python interpreter running some given code, including versions such as Jython and IronPython, which integrate with the Java virtual machine and .NET common language runtime, respectively. It exposes an asynchronous communication channel between the parent and child Python scripts, so the parent can send data that the child works on and get on with its own thing until it's ready to receive the result. If the parent is ready before the child is finished, then the parent waits.

Exercise 122 – working with execnet to execute a simple Python squaring program

In this exercise, you'll create a `squaring` process that receives `x` over an `execnet` channel and responds with `x**2`. This is much too small a task to warrant multiprocessing, but it does demonstrate how to use the library.

This exercise will be performed on a Jupyter notebook:

1. First, install `execnet` using the `pip` package manager:

```
$ pip install execnet
```

2. Now, write the `square` function, which receives numbers on a channel and returns their square:

```
import execnet
def square(channel):
    while not channel.isclosed():
        number = channel.receive()
        number_squared = number**2
        channel.send(number_squared)
```

> **Note**
> Due to the way `execnet` works, you need to type the following examples into a Jupyter notebook. You cannot type them into the interactive `>>>` prompt.

The `while not channel.isclosed()` statement ensures that we only proceed with the calculation if there is an open channel between the parent and child Python processes. `number = channel.receive()` takes the input from the parent process that you want to use `square` on. It is then squared in the `number_squared = number**2` code line. Lastly, you send the squared number back to the parent process with `channel.send(number_squared)`.

3. Now, set up a `gateway` channel to a remote Python interpreter running that function:

```
gateway = execnet.makegateway()
channel = gateway.remote_exec(square)
```

A `gateway` channel manages the communication between the parent and child Python processes. The channel is used to actually send and receive data between the processes.

4. Now, send some integers from our parent process to the child process, as shown in the following code snippet:

```
for i in range(10):
    channel.send(i)
    i_squared = channel.receive()
    print(f"{i} squared is {i_squared}")
```

You will get the following output:

```
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
```

Figure 9.16 – The results passed back from the child Python processes

Here, you loop through 10 integers, send them through the `square` channel, and then receive the result using the `channel.receive()` function.

5. When you are done with the remote Python interpreter, close the `gateway` channel to cause it to quit:

```
gateway.exit()
```

In this exercise, you learned how to use `execnet` to pass instructions between Python processes. In the following section, you will be looking at multiprocessing with the `multiprocessing` package.

Multiprocessing with the multiprocessing package

The `multiprocessing` module is built into Python's standard library. Similar to `execnet`, it allows you to launch new Python processes. However, it provides an API that is lower level than `execnet`. This means that it's harder to use than `execnet`, but affords more flexibility. An `execnet` channel can be simulated by using a pair of multiprocessing queues.

Exercise 123 – using the multiprocessing package to execute a simple Python program

In this exercise, you will use the `multiprocessing` module to complete the same task as in *Exercise 122 – working with execnet to execute a simple Python squaring program*:

1. Create a new text file called `multi_processing.py`.
2. Now, use `import` for the `multiprocessing` package:

```
import multiprocessing
```

3. Create a `square_mp` function that will continuously monitor the queue for numbers, and when it sees a number, it will take it, square it, and place it in the outbound queue:

```
def square_mp(in_queue, out_queue):
    while(True):
        n = in_queue.get()
        n_squared = n**2
        out_queue.put(n_squared)
```

4. Finally, add the following block of code to `multi_processing.py`:

```
if __name__ == '__main__':
    in_queue = multiprocessing.Queue()
    out_queue = multiprocessing.Queue()
    process = multiprocessing.Process(target=square_mp,
args=(in_queue, out_queue))
    process.start()
    for i in range(10):
        in_queue.put(i)
        i_squared = out_queue.get()
        print(f"{i} squared is {i_squared}")
    process.terminate()
```

Recall that the `if name == '__main__'` line simply avoids executing this section of code if the module is being imported elsewhere in your project. In comparison, `in_queue` and `out_queue` are both queue objects through which data can be sent between the parent and child processes. Within the following loop, you can see that you add integers to `in_queue` and get the results from `out_queue`. If you look at the preceding `square_mp` function, you can see how the child process will get its values from the `in_queue` object and pass the result back into the `out_queue` object.

5. Execute your program from the command line as follows:

```
python multi_processing.py
```

The output will be as follows:

```
(base) C:\Users\andrew.bird\Python-In-Demand\Lesson09>python multi_processing.py
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81

(base) C:\Users\andrew.bird\Python-In-Demand\Lesson09>
```

Figure 9.17 – Running our multiprocessing script

In this exercise, you learned how to pass tasks between our parent and child Python processes using the `multiprocessing` package and you found the square of a set of numbers.

Multiprocessing with the threading package

Whereas `multiprocessing` and `execnet` create a new Python process to run your asynchronous code, threading simply creates a new thread within the current process. Therefore, it uses fewer operating resources than other alternatives. Your new thread shares all its memory, including global variables, with the creating thread. The two threads are not truly concurrent, because the GIL means only one Python instruction can be running at once across all threads in a Python process.

Finally, you cannot terminate a thread, so unless you plan to exit your whole Python process, you must provide the `thread` function with a way to exit. In the following exercise, you'll use a special signal value sent to a queue to exit the thread.

Exercise 124 – using the threading package

In this exercise, you will use the `threading` module to complete the same task of squaring numbers as in *Exercise 122 – working with execnet to execute a simple Python squaring program*:

1. In a Jupyter notebook, use `import` for the `threading` and `queue` modules:

```
import threading
import queue
```

2. Create two new queues to handle the communication between our processes, as shown in the following code snippet:

```
in_queue = queue.Queue()
out_queue = queue.Queue()
```

3. Create the function that will watch the queue for new numbers and return squared numbers. The `if n == 'STOP'` line allows you to terminate the thread by passing `STOP` into the `in_queue` object:

```
def square_threading():
    while True:
        n = in_queue.get()
        if n == 'STOP':
            return
        n_squared = n**2
        out_queue.put(n_squared)
```

4. Now, create and start a new thread:

```
thread = threading.Thread(target=square_threading)
thread.start()
```

5. Loop through `10` numbers, pass them into the `in_queue` object, and receive them from the `out_queue` object as the expected output:

```
for i in range(10):
    in_queue.put(i)
    i_squared = out_queue.get()
    print(f"{i} squared is {i_squared}")
in_queue.put('STOP')
thread.join()
```

The output will be as follows:

```
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
```

Figure 9.18 – Output from the threading loop

In this exercise, you learned how to pass tasks between our parent and child Python processes using the `threading` package. In the following section, you will look at parsing command-line arguments in scripts.

Parsing command-line arguments in scripts

Scripts often need input from their user in order to make certain choices about what the script does or how it runs. For instance, consider a script to train a deep learning network used for image classification. A user of this script will want to tell it where the training images are and what the labels are, and may want to choose what model to use, the learning rate, where to save the trained model configuration, and other features.

It's conventional to use command-line arguments; that is, values that the user supplies from their shell or their own script when running your script. Using command-line arguments makes it easy to automate using the script in different ways and will be familiar to users who have experience using the Unix or Windows command shells.

Python's standard library module for interpreting command-line arguments, `argparse`, supplies a host of features, making it easy to add argument handling to scripts in a fashion that is consistent with other tools. You can make arguments required or optional, have the user supply values for certain arguments, or define default values. `argparse` creates usage text, which the user can read using the `--help argument`, and checks the user-supplied arguments for validity.

Using `argparse` is a four-step process. First, you create a `parser` object. Second, you add arguments your program accepts to the `parser` object. Third, tell the `parser` object to parse your script's `argv` (short for argument vector – the list of arguments that were supplied to the script on launch); it checks them for consistency and stores the values. Finally, use the object returned from the `parser` object in your script to access the values supplied in the arguments.

To run all of the exercises in this section, later on, you will need to type the Python code into the `.py` files and run them from your operating system's command line, not from a Jupyter notebook.

Exercise 125 – introducing argparse to accept input from the user

In this exercise, you'll create a program that uses `argparse` to take a single input from the user called `flag`. If the `flag` input is not provided by the user, its value is `False`. If it is provided, its value is `True`. This exercise will be performed in a Python terminal:

1. Create a new Python file called `argparse_demo.py`.
2. Import the `argparse` library:

```
import argparse
```

3. Create a new `parser` object, as shown in the following code snippet:

```
parser = argparse.ArgumentParser(description="Interpret a
Boolean flag.")
```

4. Add an argument that will allow the user to pass through the `–flag` argument when they execute the program:

```
parser.add_argument('--flag', dest='flag', action='store_
true', help='Set the flag value to True.')
```

The `store_true` action means that the parser will set the value of the argument to `True` if the `flag` input is present. Otherwise, it will set the value to `False`. The exact opposite can be achieved using the `store_false` action.

5. Now, call the `parse_args()` method, which executes the actual processing of the arguments:

```
arguments = parser.parse_args()
```

6. Now, print the value of the argument to see whether it worked:

```
print(f"The flag's value is {arguments.flag}")
```

7. Execute the file with no arguments supplied; the value of `arguments.flag` should be `False`:

```
python argparse_example.py
```

The output will be as follows:

```
(base) C:\Users\andrew.bird\Python-In-Demand\Lesson09>python argparse_demo.py
The flag's value is False
```

Figure 9.19 – Running argparse_demo with no arguments

8. Run the script again with the `--flag` argument to set it to `True`:

```
python argparse_demo.py -flag
```

The output will be as follows:

```
(base) C:\Users\andrew.bird\Python-In-Demand\Lesson09>python argparse_demo.py --flag
The flag's value is True

(base) C:\Users\andrew.bird\Python-In-Demand\Lesson09>
```

Figure 9.20 – Running argparse_demo with the --flag argument

9. Now, enter the following code and see the `help` text that `argparse` extracted from the description and the `help` text you supplied:

```
python argparse_demo.py -help
```

You will get the following output:

```
(base) C:\Users\andrew.bird\Python-In-Demand\Lesson09>python argparse_demo.py --help
usage: argparse_demo.py [-h] [--flag]

Interpret a Boolean flag.

optional arguments:
  -h, --help  show this help message and exit
  --flag      Set the flag value to True.
```

Figure 9.21 – Viewing the help text of argparse_demo

You have successfully created a script that allows an argument to be specified when it is executed. You can probably imagine how useful this can often be.

Positional arguments

Some scripts have arguments that are fundamental to their operation. For example, a script that copies a file always needs to know the **source** and **destination** files. It would be inefficient to repetitively type out the names of the arguments, for instance, `python copyfile.py --source infile --destination outfile`, every time you used the script.

You can use positional arguments to define arguments that the user does not name but are always provided in a particular order. The difference between a positional and a named argument is that a named argument starts with a hyphen (`-`), such as `--flag` in *Exercise 125 – introducing argparse to accept input from the user*. A positional argument does **not** start with a hyphen.

Exercise 126 – using positional arguments to accept source and destination inputs from a user

In this exercise, you will create a program that uses `argparse` to take two inputs from the user: `source` and `destination`.

This exercise will be performed in a Python terminal:

1. Create a new Python file called `positional_args.py`.
2. Import the `argparse` library:

```
import argparse
```

3. Create a new `parser` object:

```
parser = argparse.ArgumentParser(description="Interpret
positional arguments.")
```

4. Add two arguments for the `source` and `destination` values:

```
parser.add_argument('source', action='store', help='The
source of an operation.')
parser.add_argument('dest', action='store', help='The
destination of the operation.')
```

5. Call the `parse_args()` method, which executes the actual processing of `arguments`:

```
arguments = parser.parse_args()
```

6. Now, print the value of `arguments` so that you can see whether it worked:

```
print(f"Picasso will cycle from {arguments.source} to
{arguments.dest}")
```

7. Now, execute the file while using this script with no arguments, which causes an error because it expects two `positional` arguments:

```
python positional_args.py
```

The output now will be as follows:

```
(base) C:\Users\andrew.bird\Python-In-Demand\Lesson09>python positional_args.py
usage: positional_args.py [-h] source dest
positional_args.py: error: the following arguments are required: source, dest
```

Figure 9.22 – Running the script with no arguments specified

8. Try running the script and specifying two locations as the source and destination positional arguments:

> **Note**
> The arguments are supplied on the command line with no names or leading hyphens.

```
$ python positional_args.py Chichester Battersea
```

The output is as follows:

```
(base) C:\Users\andrew.bird\Python-In-Demand\Lesson09>python positional_args.py Chichester Battersea
Picasso will cycle from Chichester to Battersea
```

Figure 9.23 – Successfully specifying two positional arguments

In this exercise, you learned how to parameterize your scripts by accepting positional arguments using the `argparse` Python package.

Performance and profiling

Python is not often thought of as a high-performance language, although it really should be. The simplicity of the language and the power of its standard library mean that the time from idea to result can be much shorter than in other languages with better runtime performance.

However, we have to be honest. Python is not among the fastest-running programming languages in the world and sometimes, that's important. For instance, if you're writing a web server application, you need to be able to handle as many network requests as are being made, and with timeliness that satisfies the users making the requests.

Alternatively, if you're writing a scientific simulation or a deep learning inference engine, then the simulation or training time can completely dwarf the programmer time (which is your time) spent writing the code. In any situation, reducing the time spent running your application can decrease the cost, whether measured in dollars on your cloud hosting bill or in milliamp-hours on your laptop battery.

Changing your Python environment

You'll learn how to use some of Python's timing and profiling tools later on in this section. Before that, you can consider whether you even need to do that. Taligent, an object-oriented software company in the 1990s, had a performance saying: "*There is no code faster than no code*." You can generalize that idea as follows:

There is no work that can be done faster than doing no work.

The fastest way to speed up your Python program can often be to simply use a different Python interpreter. You saw earlier in this chapter that multithreaded Python is slowed down by `GIL`, which means that only one Python thread can execute a Python instruction at any time in a given process. The `Jython` and `IronPython` environments, targeting the Java virtual machine and .NET common language runtime, do not have `GIL`, so they may be faster for multithreaded programs, but there are also two Python implementations that are specifically designed to perform better, so we'll look to those for assistance in later sections.

PyPy

We will now look at another Python environment in more detail. It's called `pypy` and Guido van Rossum (Python's creator) has said, "*If you want your code to run faster, you should probably just use PyPy*."

PyPy's secret is **just-in-time** (**JIT**) compilation, which compiles the Python program to a machine language, such as Cython, but does it while the program is running rather than once on the developer's machine, as with **ahead-of-time** (**AOT**) compilation. For a long-running process, a JIT compiler can try different strategies to compile the same code and find the ones that work best in the program's environment. The program will quickly get faster until the best version the compiler can find is running. Take a look at PyPy in the following exercise.

Exercise 127 – using PyPy to find the time to get a list of prime numbers

In this exercise, you will be executing a Python program to get a list of prime numbers using milliamp-hours, but remember that you are more interested in checking the amount of time needed to execute the program using `pypy`.

This exercise will be performed in a Python terminal:

1. First, run the `pypy3` command, as shown in the following code snippet:

```
pypy3
Python 3.6.1 (dab365a465140aa79a5f3ba4db784c4af4d5c195,
Feb 18 2019, 10:53:27)
[PyPy 7.0.0-alpha0 with GCC 4.2.1 Compatible Apple LLVM
10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
And now for something completely different: ''release 1.2
upcoming''
>>>>
```

Note that you may find it easier to navigate to the folder with the `pypy3.exe` file and run the preceding command, instead of following the installation instructions to create a symlink.

2. Press *Ctrl* + *D* to exit `pypy`.

   You're going to use the program from *Chapter 7*, *Becoming Pythonic*, again, which finds prime numbers using the *Sieve of Eratosthenes method*. There are two changes that you will introduce here: firstly, find prime numbers up to 1,000 to give the program more work to do; secondly, instrument it with Python's `timeit` module so that you can see how long it takes to run. `timeit` runs a Python statement multiple times and records how long it takes. Tell `timeit` to run your Sieve of Eratosthenes 10,000 times (the default is 100,000 times, which takes a very long time).

3. Create an `eratosthenes.py` file and enter the following code:

```
import timeit
class PrimesBelow:
    def __init__(self, bound):
        self.candidate_numbers = list(range(2,bound))
    def __iter__(self):
        return self
    def __next__(self):
        if len(self.candidate_numbers) == 0:
            raise StopIteration
        next_prime = self.candidate_numbers[0]
        self.candidate_numbers = [x for x in self.
candidate_numbers if x % next_prime != 0]
        return next_prime
print(timeit.timeit('list(PrimesBelow(1000))',
setup='from __main__ import PrimesBelow', number=10000))
```

4. Run the file with the regular Python interpreter:

```
python eratosthenes.py
```

   You will get the following output:

```
(base) C:\Users\andrew.bird\Python-In-Demand\Lesson09>python eratosthenes.py
17.597791835
```

Figure 9.24 – Executing with the regular Python interpreter

The number will be different on your computer, but that's `17.6` seconds to execute the `list(PrimesBelow(1000))` statement 10,000 times, or 1,760 µs per iteration. Now, run the same program using `pypy` instead of CPython:

```
$ pypy3 eratosthenes.py
```

You will get the following output:

```
4.81645076300083
```

Here, it is 482 µs per iteration.

In this exercise, you will have noticed that it only takes 30% of the time to run our code in `pypy` as it took in Python. You really can get a significant performance boost with very little effort, just by switching to `pypy`.

Cython

A Python module can be compiled to C with a wrapper, which means it is still accessible from other Python code. Compiling code simply means it is taken from one language and put into another. In this case, the compiler takes Python code and expresses it in the C programming language. The tool that does this is called `Cython` and it often generates modules with lower memory use and execution time than if they're left as Python.

> **Note**
> The standard Python interpreter, the one you've almost certainly been using to complete the exercises and activities in this course, is sometimes called **CPython**. This is confusingly similar to **Cython**, but the two are actually different projects.

Exercise 128 – adopting Cython to find the time taken to get a list of prime numbers

In this exercise, you will install `Cython`, and, as mentioned in the previous exercise, you will find a list of prime numbers, although you are more interested in knowing the amount of time it takes to execute the code using Cython.

This exercise will be performed on the command line:

1. Firstly, install `cython`, as shown in the following code snippet:

```
$ pip install cython
```

2. Now, go back to the code you wrote for *Exercise 8 – displaying strings*, and extract the class for iterating over primes using the Sieve of Eratosthenes into a file, `sieve_module.py`:

```
class PrimesBelow:
    def __init__(self, bound):
        self.candidate_numbers = list(range(2,bound))
    def __iter__(self):
        return self
    def __next__(self):
        if len(self.candidate_numbers) == 0:
            raise StopIteration
        next_prime = self.candidate_numbers[0]
        self.candidate_numbers = [x for x in self.
candidate_numbers if x % next_prime != 0]
        return next_prime
```

3. Compile that into a C module using `Cython`. Create a file called `setup.py` with the following contents:

```
from distutils.core import setup
from Cython.Build import cythonize
setup(
    ext_modules = cythonize("sieve_module.py")
)
```

4. Now, on the command line, run `setup.py` to build the module, as shown in the following code snippet:

```
$ python setup.py build_ext --inplace
```

The output will look different if you're on Linux or Windows, but you should see no errors:

```
running build_ext
building 'sieve_module' extension
creating build
creating build/temp.macosx-10.7-x86_64-3.7
gcc -Wno-unused-result -Wsign-compare -Wunreachable-code
-DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes -I/
Users/leeg/anaconda3/include -arch x86_64 -I/Users/leeg/
anaconda3/include -arch x86_64 -I/Users/leeg/anaconda3/
include/python3.7m -c sieve_module.c -o build/temp.
macosx-10.7-x86_64-3.7/sieve_module.o
```

```
gcc -bundle -undefined dynamic_lookup -L/Users/leeg/
anaconda3/lib -arch x86_64 -L/Users/leeg/anaconda3/
lib -arch x86_64 -arch x86_64 build/temp.macosx-
10.7-x86_64-3.7/sieve_module.o -o /Users/leeg/Nextcloud/
Documents/Python Book/Lesson_9/sieve_module.cpython-37m-
darwin.so
```

5. Now, import the `timeit` module and use it in a script called `cython_sieve.py`:

```
import timeit
print(timeit.timeit('list(PrimesBelow(1000))',
setup='from sieve_module import PrimesBelow',
number=10000))
```

6. Run this program to see the timing:

```
$ python cython_sieve.py
```

The output will be as follows:

```
3.830873068
```

Here, it is 3.83 seconds, so 383 µs per iteration. That's a little over 40% of the time taken by the CPython version, but the `pypy` Python was still able to run the code faster. The advantage of using Cython is that you are able to make a module that is compatible with CPython, so you can make your module code faster without needing to make everybody else switch to a different Python interpreter to reap the benefits.

Profiling code

Having exhausted the minimum-effort options for improving your code's performance, it's time to actually put some work in if you need to go faster. There's no recipe to follow to write fast code: if there were, we could have taught you that in the previous chapter and there wouldn't need to be a section on performance now. Of course, speed also isn't the only performance goal – you might want to reduce memory use or increase the number of simultaneous operations that can be in flight – but programmers often use "performance" as a synonym for "reducing time to completion," and that's what you'll investigate here.

Improving performance is a scientific process: you observe how your code behaves, hypothesize about a potential improvement, make the change, and then observe it again and check that you really did improve things. Good tool support exists for the observation steps in this process and you'll look at one of these tools now: `cProfile`.

`cProfile` is a module that builds an execution profile of your code. Every time your Python program enters or exits a function or other callable, `cProfile` records what it is and how long it takes. It's then up to you to work out how you could spend less time doing that. Remember to compare a profile recorded before your change with one recorded after to make sure you improved things! As you'll see

in the next exercise, not all "optimizations" actually make your code faster, and careful measurement and thought are needed to decide whether the optimization is worth pursuing and retaining. In practice, `cProfile` is often used when trying to understand why code is taking longer than expected to execute. For example, you might write an iterative calculation that suddenly takes 10 minutes to compute after scaling to 1,000 iterations. With `cProfile`, you might discover that this is due to some inefficient function in the `pandas` library, which you could potentially avoid to speed up your code.

Profiling with cProfile

The goal of this example is to learn how to diagnose code performance using `cProfile`. In particular, to understand which parts of your code are taking the most time to execute.

This is a pretty long example, and the point is not to make sure that you type in and understand the code but to understand the process of profiling, consider changes, and observe the effects those changes have on the profile. This example will be performed on the command line:

1. Start with the code you wrote in *Chapter 7*, *Becoming Pythonic*, to generate an infinite series of prime numbers:

```
class Primes:
    def __init__(self):
        self.current = 2
    def __iter__(self):
        return self
    def __next__(self):
        while True:
            current = self.current
            square_root = int(current ** 0.5)
            is_prime = True
            if square_root >= 2:
                for i in range(2, square_root + 1):
                    if current % i == 0:
                        is_prime = False
                        break
            self.current += 1
            if is_prime:
                return current
```

2. You'll remember that you had to use `itertools.takewhile()` to turn this into a finite sequence. Do so to generate a large list of primes and use `cProfile` to investigate its performance:

```
import cProfile
import itertools
cProfile.run('[p for p in itertools.takewhile(lambda x:
x<10000, Primes())]')
```

You will get the following output:

```
      2466 function calls in 0.021 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.000    0.000 <ipython-input-1-5aedc56b5f71>:2(__init__)
     1    0.000    0.000    0.000    0.000 <ipython-input-1-5aedc56b5f71>:4(__iter__)
  1230    0.020    0.000    0.020    0.000 <ipython-input-1-5aedc56b5f71>:6(__next__)
  1230    0.000    0.000    0.000    0.000 <string>:1(<lambda>)
     1    0.001    0.001    0.021    0.021 <string>:1(<listcomp>)
     1    0.000    0.000    0.021    0.021 <string>:1(<module>)
     1    0.000    0.000    0.021    0.021 {built-in method builtins.exec}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Figure 9.25 – Investigating performance with cProfile

The `__next__()` function is called most often, which is not surprising, as it is the iterative part of the iteration. It also takes up most of the execution time in the profile. So, is there a way to make it faster?

One hypothesis is that the method does a lot of redundant divisions. Imagine that the number 101 is being tested as a prime number. This implementation tests whether it is divisible by 2 (no), then 3 (no), and then 4, but 4 is a multiple of 2, so you know it isn't divisible by 2.

3. As a hypothesis, change the `__next__()` method so that it only searches the list of known prime numbers. You know that if the number being tested is divisible by any smaller numbers, at least one of those numbers is itself prime:

```
class Primes2:
    def __init__(self):
        self.known_primes=[]
        self.current=2
    def __iter__(self):
        return self
    def __next__(self):
        while True:
            current = self.current
```

```
            prime_factors = [p for p in self.known_primes
if current % p == 0]
            self.current += 1
            if len(prime_factors) == 0:
                self.known_primes.append(current)
                return current
cProfile.run('[p for p in itertools.takewhile(lambda x:
x<10000, Primes2())]')
```

The output will be as follows:

```
   23708 function calls in 0.468 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 10006    0.455    0.000    0.455    0.000 <ipython-input-2-c6ffd796f813>:10(<listcomp>)
     1    0.000    0.000    0.000    0.000 <ipython-input-2-c6ffd796f813>:2(__init__)
     1    0.000    0.000    0.000    0.000 <ipython-input-2-c6ffd796f813>:5(__iter__)
  1230    0.011    0.000    0.466    0.000 <ipython-input-2-c6ffd796f813>:7(__next__)
  1230    0.000    0.000    0.000    0.000 <string>:1(<lambda>)
     1    0.001    0.001    0.468    0.468 <string>:1(<listcomp>)
     1    0.000    0.000    0.468    0.468 <string>:1(<module>)
     1    0.000    0.000    0.468    0.468 {built-in method builtins.exec}
 10006    0.001    0.000    0.001    0.000 {built-in method builtins.len}
  1230    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Figure 9.26 – It took longer this time!

Now, `__next()__` isn't the most frequently called function in the profile, but that's not a good thing. Instead, you've introduced a list comprehension that gets called even more times, and the whole process takes 30 times longer than it used to.

4. One thing that changed in the switch from testing a range of factors to the list of known primes is that the upper bound of tested numbers is no longer the square root of the candidate prime. Going back to thinking about testing whether 101 is prime, the first implementation tested all numbers between 2 and 10. The new one tests all primes from 2 to 97 and is therefore doing more work. Reintroduce the square root upper limit using `takewhile` to filter the list of primes:

```
class Primes3:
    def __init__(self):
        self.known_primes=[]
        self.current=2
    def __iter__(self):
        return self
    def __next__(self):
```

```
            while True:
                current = self.current
                sqrt_current = int(current**0.5)
                potential_factors = itertools.
takewhile(lambda x: x < sqrt_current, self.known_primes)
                prime_factors = [p for p in potential_factors
if current % p == 0]
                self.current += 1
                if len(prime_factors) == 0:
                    self.known_primes.append(current)
                    return current
cProfile.run('[p for p in itertools.takewhile(lambda x:
x<10000, Primes3())]')
```

The output will be as follows:

```
291158 function calls in 0.102 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
267345    0.023    0.000    0.023    0.000 <ipython-input-3-10d4133c7618>:11(<lambda>)
 10006    0.058    0.000    0.081    0.000 <ipython-input-3-10d4133c7618>:12(<listcomp>)
     1    0.000    0.000    0.000    0.000 <ipython-input-3-10d4133c7618>:2(__init__)
     1    0.000    0.000    0.000    0.000 <ipython-input-3-10d4133c7618>:5(__iter__)
  1265    0.018    0.000    0.100    0.000 <ipython-input-3-10d4133c7618>:7(__next__)
  1265    0.000    0.000    0.000    0.000 <string>:1(<lambda>)
     1    0.001    0.001    0.102    0.102 <string>:1(<listcomp>)
     1    0.000    0.000    0.102    0.102 <string>:1(<module>)
     1    0.000    0.000    0.102    0.102 {built-in method builtins.exec}
 10006    0.001    0.000    0.001    0.000 {built-in method builtins.len}
  1265    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Figure 9.27 – Getting faster this time

5. Much better. Well, much better than `Primes2` anyway. This still takes seven times longer than the original algorithm. There's still one trick to try. The biggest contribution to the execution time is the list comprehension as highlighted in the previous code. By turning that into a `for` loop, it's possible to break the loop early by exiting as soon as a prime factor for the candidate prime is found:

```
class Primes4:
    def __init__(self):
        self.known_primes=[]
        self.current=2
```

```
    def __iter__(self):
        return self
    def __next__(self):
        while True:
            current = self.current
            sqrt_current = int(current**0.5)
            potential_factors = itertools.
takewhile(lambda x: x < sqrt_ current, self.known_primes)
            is_prime = True
            for p in potential_factors:
                if current % p == 0:
                    is_prime = False
                    break
            self.current += 1
            if is_prime == True:
                self.known_primes.append(current)
                return current
cProfile.run('[p for p in itertools.takewhile(lambda x:
x<10000, Primes4())]')
```

The output will be as follows:

```
     64802 function calls in 0.033 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 61001    0.007    0.000    0.007    0.000 <ipython-input-4-4f9e19e7ebde>:11(<lambda>)
     1    0.000    0.000    0.000    0.000 <ipython-input-4-4f9e19e7ebde>:2(__init__)
     1    0.000    0.000    0.000    0.000 <ipython-input-4-4f9e19e7ebde>:5(__iter__)
  1265    0.024    0.000    0.032    0.000 <ipython-input-4-4f9e19e7ebde>:7(__next__)
  1265    0.000    0.000    0.000    0.000 <string>:1(<lambda>)
     1    0.001    0.001    0.033    0.033 <string>:1(<listcomp>)
     1    0.000    0.000    0.033    0.033 <string>:1(<module>)
     1    0.000    0.000    0.033    0.033 {built-in method builtins.exec}
  1265    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Figure 9.28 – An even faster output

Once again, the result is better than the previous attempt, but it is still not as good as the "naive" algorithm. This time, the biggest contribution to the runtime is the lambda expression on line 11. This tests whether one of the previously found primes is smaller than the square root of the candidate number. There's no way to remove that test from this version of the algorithm. In

other words, surprisingly, in this case, doing too much work to find a prime number is faster than finding the minimum work necessary and doing just that.

6. In fact, the good news is that our effort has not been wasted. I don't recommend running this yourself unless the instructor says it's time for a coffee break, but if you increase the number of primes your iterator searches for, there will come a point where the "optimized" algorithm will outpace the "naive" implementation:

```
cProfile.run('[p for p in itertools.takewhile(lambda x:
x<10000000, Primes())]')
```

You will get the following output:

```
         1329166 function calls in 147.528 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <ipython-input-1-5aedc56b5f71>:2(__init__)
        1    0.000    0.000    0.000    0.000 <ipython-input-1-5aedc56b5f71>:4(__iter__)
   664580  146.901    0.000  146.901    0.000 <ipython-input-1-5aedc56b5f71>:6(__next__)
   664580    0.101    0.000    0.101    0.000 <string>:1(<lambda>)
        1    0.514    0.514  147.516  147.516 <string>:1(<listcomp>)
        1    0.011    0.011  147.528  147.528 <string>:1(<module>)
        1    0.000    0.000  147.528  147.528 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Figure 9.29 – The result of the naive implementation

Now, we can run the same with the alternative implementation:

```
cProfile.run('[p for p in itertools.takewhile(lambda x:
x<10000000, Primes4())]')
```

You will get the following output:

```
         317503134 function calls in 106.236 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
315507795   24.815    0.000   24.815    0.000 <ipython-input-4-4f9e19e7ebde>:11(<lambda>)
        1    0.000    0.000    0.000    0.000 <ipython-input-4-4f9e19e7ebde>:2(__init__)
        1    0.000    0.000    0.000    0.000 <ipython-input-4-4f9e19e7ebde>:5(__iter__)
   665111   80.611    0.000  105.523    0.000 <ipython-input-4-4f9e19e7ebde>:7(__next__)
   665111    0.114    0.000    0.114    0.000 <string>:1(<lambda>)
        1    0.583    0.583  106.221  106.221 <string>:1(<listcomp>)
        1    0.015    0.015  106.236  106.236 <string>:1(<module>)
        1    0.000    0.000  106.236  106.236 {built-in method builtins.exec}
   665111    0.097    0.000    0.097    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Figure 9.30 – The result of the optimized implementation

By the end of this example, you were able to find the best-optimized method to run the code. This decision was made possible by observing the amount of time needed to run the code, allowing us to tweak the code to address inefficiencies. In the following activity, you will put all of these concepts together.

Activity 23 – generating a list of random numbers in a Python virtual environment

You work for a sports betting website and want to simulate random events in a particular betting market. In order to do so, your goal will be to create a program that is able to generate long lists of random numbers using multiprocessing.

In this activity, the aim is to create a new Python environment, install the relevant packages, and write a function using the `threading` library to generate a list of random numbers.

Here are the steps:

1. Create a new `conda` environment called `my_env`.
2. Activate the `conda` environment.
3. Install `numpy` in your new environment.
4. Install and run a Jupyter notebook from within your virtual environment.
5. Create a new Jupyter notebook and import libraries such as `numpy`, `cProfile`, `itertools`, and `threading`.
6. Create a function that uses the `numpy` and `threading` libraries to generate an array of random numbers. Recall that when threading, we need to be able to send a signal for the `while` statement to terminate. The function should monitor the queue for an integer that represents the number of random numbers it should return. For example, if the number `10` was passed into the queue, it should return an array of `10` random numbers.
7. Next, add a function that will start a thread and put integers into the `in_queue` object. You can optionally print the output by setting the `show_output` argument to `True`. Make this function loop through the integers `0` to `n`, where `n` can be specified when the function is called. For each integer between `0` and `n`, it will pass the integer into the queue, and receive the array of random numbers.
8. Run the numbers on a small number of iterations to test and see the output.

You will get the following output:

```
[]
[0.78155881]
[0.61671875 0.96379795]
[0.52748128 0.69182391 0.11764897]
[0.89243527 0.75566451 0.88089298 0.15782374]
[0.1140009  0.25980504 0.88632411 0.08730527 0.17493792]
[0.41370041 0.01167654 0.60758276 0.73804504 0.73648781 0.29094613
[0.8317736  0.57914287 0.01291246 0.61011878 0.91729392 0.50898183
 0.24640681]
[0.4475645  0.94036652 0.69823962 0.37459892 0.15512432 0.15115215
 0.65882522 0.77908825]
[0.42420881 0.7135031  0.22843178 0.20624473 0.32533328 0.86108686
 0.46407033 0.81794371 0.98958707]
```

Figure 9.31 – The expected sample output

9. Rerun the numbers with a large number of iterations and use `cProfile` to view a breakdown of what is taking time to execute.

> **Note**
> The solution for this activity can be found in *Appendix* on GitHub.

With this, we conclude this chapter on advanced topics for Python.

Summary

In this chapter, you have seen some of the tools and skills needed to transition from being a Python programmer to a Python software engineer. You have learned how to collaborate with other programmers using `Git` and GitHub, how to manage dependencies and virtual environments with `conda`, and how to deploy Python applications using Docker. We have explored multiprocessing and investigated tools and techniques used for improving the performance of your Python code. These new skills leave you better equipped to handle the messy real world of collaborative teams working on large problems in production environments. These skills are not just academic, but are essential tools for any aspiring Python developer to familiarize themselves with.

The next chapter will introduce the part of the book dedicated to using Python for data science. You will learn about popular libraries for working with numerical data and techniques for importing, exploring, cleaning up, and analyzing real-world data.

10
Data Analytics with pandas and NumPy

Overview

By the end of this chapter, you will be able to use NumPy to perform statistics and speed up matrix computations; use pandas to view, create, analyze, and modify DataFrames; organize and modify data using `read`, `transpose`, `loc`, `iloc`, and `concatenate`; clean data by deleting or manipulating NaN values and coercing column types; visualize data by constructing, modifying, and interpreting histograms and scatter plots; generate and interpret statistical models using pandas and statsmodels, and solve real-world problems using data analytics techniques.

Introduction

In *Chapter 9*, *Practical Python – Advanced Topics*, you learned how to use GitHub to collaborate with team members. You also used `conda` to document and set up the dependencies for Python programs and `docker` to create reproducible Python environments to run our code.

We will now shift gears to data science. Data science is booming like never before. Data scientists have become among the most sought-after practitioners in the world today. Most leading corporations have data scientists to analyze and explain their data.

Data analytics focuses on the analysis of big data. As each day goes by, there is more data than ever before – far too much for any human to analyze by sight. Leading Python developers such as Wes McKinney and Travis Oliphant addressed this gap by creating specialized Python libraries – in particular, pandas and NumPy – to handle big data.

Taken together, pandas and NumPy are masterful at handling big data. They are built for speed, efficiency, readability, and ease of use.

pandas provides you with a unique framework to view and modify data. It handles all data-related tasks such as creating DataFrames, importing data, scraping data from the web, merging data, pivoting, concatenating, and more.

NumPy, short for Numerical Python, is more focused on computation. NumPy interprets the rows and columns of pandas DataFrames as matrices in the form of NumPy arrays. When computing descriptive statistics such as the mean, median, mode, and quartiles, NumPy is blazingly fast.

Another key player in data analysis is Matplotlib, a graphing library that handles scatter plots, histograms, regression lines, and more, all of which you were introduced to in *Chapter 4*, *Extending Python, Files, Errors, and Graphs*. The importance of data graphs cannot be overstated since most non-technical professionals use them to interpret results.

We will be looking at the following topics in this chapter:

- NumPy and basic stats
- Matrices
- The pandas library
- Working with big data
- Null values
- Creating statistical graphs

Let's start!

Technical requirements

The code files for this chapter are available on GitHub at `https://github.com/PacktPublishing/The-Python-Workshop-Second-Edition/tree/main/Chapter10`.

NumPy and basic stats

NumPy is designed to handle big data swiftly. It includes the following essential components according to the NumPy documentation:

- A powerful n-dimensional array object
- Sophisticated (broadcasting) functions
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform, and random number capabilities

You will be using NumPy going forward. Instead of using lists, you will use NumPy arrays, which are basic elements of the NumPy package. NumPy arrays are designed to handle arrays of any dimension.

Numpy arrays can be indexed easily and can have many types of data, such as `float`, `int`, `string`, and `object`, but the types must be consistent to improve speed.

Exercise 129 – converting lists into NumPy arrays

In this exercise, you will convert a list into a `numpy` array. The following steps will enable you to complete this exercise:

1. Open a new Jupyter Notebook.
2. Then, you need to import `numpy`:

```
import numpy as np
```

3. Now, you must create a list for `test_scores` and confirm the type of data:

```
test_scores = [70,65,95,88]
type(test_scores)
```

The output will be as follows:

```
list
```

> **Note**
> Now that `numpy` has been imported, you can access all `numpy` methods, such as `numpy` arrays. Type `np.` and then press *Tab* on your keyboard to see the breadth of options. You are looking for an array.

4. Now, you must convert the list of marks into a `numpy` array and check the array's `type`. Enter the code shown in the following code snippet:

```
scores = np.array(test_scores)
type(scores)
```

The output will be as follows:

```
numpy.ndarray
```

In this exercise, you converted a list of test score marks into a NumPy array. You will find the mean using these values within the NumPy array in the following exercise.

One of the most common statistical measures is the mean. Traditionally thought of as the average, the mean of a list is the sum of each entry divided by the number of entries. In NumPy, the mean may be computed using the `.mean` method.

Exercise 130 – calculating the mean of the test score

In this exercise, you will use the `numpy` array you created to store our test scores from the previous exercise, and you will calculate the mean of `scores`. The following steps will enable you to complete this exercise:

1. Continue working in the same Jupyter Notebook from the previous exercise.
2. Now, to find the "average" of `scores`, you can use the `mean` method, as shown here:

   ```
   scores.mean()
   ```

   The output will be as follows:

   ```
   79.5
   ```

> **Note**
> The word "average" is in quotation marks. This is not an accident. The mean is only one kind of average. Another kind of average is the median.

Given our test scores of 70, 65, 95, and 88, the "average" is 79.5, which is the expected output. In this exercise, you were able to use the `mean` function of NumPy and find the average of `scores`. In the following exercise, you will find the median using NumPy.

The median is the number in the middle. Although not necessarily the best measure of test averages, it's an excellent measure of income average because it is robust to outliers, unlike the mean, as you will see in the next exercise.

Exercise 131 – finding the median from a collection of income data

In this exercise, you will be finding the median from a collection of income data for a neighborhood and help a millionaire decide whether he should build his dream house in the neighborhood based on the income data. The `median` function here is a method of `numpy`.

The following steps will enable you to complete this exercise:

1. Open a new Jupyter Notebook.
2. Now, you need to import the `numpy` package as `np`, then create a `numpy` array and assign various pieces of `income` data, as shown in the following code snippet:

   ```
   import numpy as np
   income = np.array([75000, 55000, 88000, 125000, 64000,
   97000])
   ```

3. Next, you must find the mean of the income data:

```
income.mean()
```

The output will be as follows:

```
84000
```

So far, so good. `84000` is the average `income` on your block.

4. Now, say the millionaire decides to build his dream house on the vacant corner lot. He adds a salary of 12 million dollars. Append the value of 12 million dollars to the current array and find the new mean:

```
income = np.append(income, 12000000)
income.mean()
```

The output will be as follows:

```
1786285.7142857143
```

The new average income is 1.7 million dollars. Okay. Nobody makes close to 1.7 million dollars on the block, so it's not a representative average. This is where the median comes into play.

> **Note**
> Here, the median is not a method of `np.array`, but a method of `numpy` (the mean may be computed in the same way, as a method of `numpy`).

5. Now, to find the `median` function from the `income` values, you can use the following code:

```
np.median(income)
```

The output will be as follows:

```
88000
```

This result says that half of the neighborhood residents make more than 88,000, and half of the blocks make less. This would give the millionaire a fair idea of the neighborhood. In this particular case, the median is a much better estimation of average income than the mean.

In the next section, you will be covering skewed data and outliers.

Skewed data and outliers

Something about the 12 million salary does not sit right. It's nowhere near anyone else's income. In statistics, there is an official terminology for this: you say that the data is skewed by an outlier of 12,000,000. In particular, the data is right-skewed since 12,000,000 is far to the right of every other data point.

Right-skewed data pulls the mean away from the median. If the mean greatly exceeds the median, this is clear evidence of right-skewed data. Similarly, if the mean is much less than the median, this is clear evidence of left-skewed data.

Unfortunately, there is no universal way to compute individual outliers. There are some general methods, including box plots, which you will check out later in *Exercise 146 – creating box plots*. For now, just keep in mind that outliers are far removed from other data points, and they skew the data.

Standard deviation

The standard deviation is a precise statistical measure of how spread out data points are. In the following exercise, you will find the standard deviation.

Exercise 132 – finding the standard deviation from income data

In this exercise, you will use the income data from *Exercise 131 – finding the median from a collection of income data*, to find the standard deviation of the dataset.

The following steps will enable you to complete this exercise:

1. Continue with the previous Jupyter Notebook.
2. Now, check the standard deviation using the `std()` method, as shown in the following code snippet:

   ```
   income.std()
   ```

   The output will be as follows:

   ```
   4169786.007331644
   ```

   As you can see, the standard deviation here is a huge number, which is 4 million. Although the standard deviation generally represents how far data points are expected to be from one another on average, the incomes are not 4 million away from each other.

   Now, try to find the standard deviation of the `test_scores` data from *Exercise 129 – converting lists into NumPy arrays*.

3. Assign the `test_scores` list value once again:

   ```
   test_scores = [70,65,95,88]
   ```

4. Now, convert this list into a `numpy` array:

   ```
   scores = np.array(test_scores)
   ```

5. Now, find the standard deviation of `test_scores` using the `std()` method:

```
scores.std()
```

The output will be as follows:

```
12.379418403139947
```

In this exercise, you observed that the income data is so skewed that the standard deviation of 4 million is practically meaningless. However, the 12.4 standard deviation of the test scores is meaningful; the mean test score of 79.5 with a standard deviation of 12.4 means that you can expect the scores to be about 12 points away from the mean on average.

Finding the min, max, and sum

What if you need to find the maximum, minimum, or sum of the `numpy` arrays?

You can find the maximum of an `np.array` array using the `max()` method, the minimum using the `min()` method, and the sum using the `sum()` method, as shown in the following example.

To find the maximum, enter the following code:

```
test_scores = [70,65,95,88]
np_scores = np.array(test_scores)
scores.max()
```

The output will be as follows:

```
95
```

To find the minimum, enter the following code:

```
scores.min()
```

The output will be as follows:

```
65
```

To find the sum, enter the following code:

```
scores.sum()
```

The output will be as follows:

```
318
```

In this example, you learned how to compute the max, min, and sum of a NumPy array. Although the range is not provided as a method, you can compute the range by taking the max minus the min.

Now, let's see how NumPy arrays can work together in matrices.

Matrices

Data is generally composed of rows, and each row contains the same number of columns. Data is often represented as a two-dimensional grid containing lots of numbers. It can also be interpreted as a list of lists, or a NumPy array of NumPy arrays.

In mathematics, a matrix is a rectangular array of numbers defined by the number of rows and columns. It is standard to list rows first, and columns second. For instance, a 2 x 3 matrix consists of 2 rows and 3 columns, whereas a 3 x 2 matrix consists of 3 rows and 2 columns.

Here is a 4 x 4 matrix:

$$\begin{bmatrix} 9 & 13 & 5 & 2 \\ 1 & 11 & 7 & 6 \\ 3 & 7 & 4 & 1 \\ 6 & 0 & 7 & 10 \end{bmatrix}$$

Figure 10.1 – Matrix representation of a 4 x 4 matrix

Exercise 133 – working with matrices

NumPy has several methods for creating matrices or n-dimensional arrays. One option is to place random numbers between 0 and 1 into each entry.

In this exercise, you will implement various `numpy` matrix methods and observe the outputs (recall that `random.seed` will allow us to reproduce the same numbers, but it's okay if you want to generate your own).

The following steps will enable you to complete this exercise:

1. Begin with a new Jupyter Notebook.
2. Now, generate a random 5 x 5 matrix, as shown in the following code snippet:

```
import numpy as np
np.random.seed(seed=60)
random_square = np.random.rand(5,5)
random_square
```

The output will be as follows:

```
array([[0.30087333, 0.18694582, 0.32318268, 0.66574957, 0.5669708 ],
       [0.39825396, 0.37941492, 0.01058154, 0.1703656 , 0.12339337],
       [0.69240128, 0.87444156, 0.3373969 , 0.99245923, 0.13154007],
       [0.50032984, 0.28662051, 0.22058485, 0.50208555, 0.63606254],
       [0.63567694, 0.08043309, 0.58143375, 0.83919086, 0.29301825]])
```

Figure 10.2 – A random 5 x 5 matrix being generated

In the preceding code, you used `random.seed`. Every time you run the script with `random.seed(seed=60)`, you will get the same sequence of values.

This matrix is very similar in composition to the DataFrames that you will be working with to analyze big data in this and the next two chapters.

Now, find the rows and columns of the generated matrix using indexing. The standard syntax is `random_square[row, column]`. If you omit the column entry, all columns will be selected.

3. Find the first row and first column of the matrix:

```
random_square[0]
```

The output will be as follows. It consists of all the columns and the first row:

```
array([0.30087333, 0.18694582, 0.32318268, 0.66574957,
0.5669708 ])
```

4. Now, to find the values of all the rows, and the first column of the matrix, enter the following code snippet:

```
random_square[:,0]
```

The output is as follows. It consists of all the rows and the first column only:

```
array([0.30087333, 0.18694582, 0.32318268, 0.66574957,
0.5669708 ])
```

5. Now, you must find individual entries by specifying the value of the matrix using the `random_square[row, column]` syntax.

Find the first entry in the matrix by entering the following code:

```
random_square[0,0]
```

The output, which shows the entry in the first row and first column, will be as follows:

```
0.30087333004661876
```

We can find the first entry in another way:

```
random_square[0][0]
```

The output will be as follows:

```
0.30087333004661876
```

Find the entry in the second row, third column:

```
random_square[2,3]
```

The output will be as follows:

```
0.9924592256795676
```

6. Now, to find the mean values of the matrix, you must find the mean of the entire matrix, individual rows, and columns using the `square.mean()` method, as shown in the following code.

   Here is the `mean` entry of the matrix:

   ```
   random_square.mean()
   ```

   The output will be as follows:

   ```
   0.42917627159618377
   ```

   Here is the `mean` entry of the first row:

   ```
   random_square[0].mean()
   ```

   The output will be as follows:

   ```
   0.4087444389228477
   ```

   Here is the `mean` entry of the last column:

   ```
   random_square[:,-1].mean()
   ```

   The output will be as follows:

   ```
   0.35019700684996913
   ```

In this exercise, you created a random 5 x 5 matrix, accessed rows, columns, and entries, and found various means of the matrix.

Computation time for large matrices

Let's see how long it takes to generate a matrix with 10 million entries and compute the mean:

```
%%time
np.random.seed(seed=60)
big_matrix = np.random.rand(100000, 100)
big_matrix.mean()
```

The output will be as follows:

```
CPU times: user 75.3 ms, sys: 8.14 ms, total: 83.5 ms
Wall time: 81.4 ms

0.5001355519953301
```

Figure 10.3 – Computation time for a matrix with 10 million entries

Your time will be different than ours, but it should be in the order of milliseconds. It takes much less than a second to generate a matrix of 10 million entries and compute its mean.

In the next exercise, you will use various NumPy arrays, including `ndarray` and `numpy.ndarray`, a (usually fixed-size) multidimensional array container of items of the same type and size.

Exercise 134 – creating an array to implement NumPy computations

In this exercise, you will generate a new matrix and perform mathematical operations on it, which will be covered later in this exercise. Unlike traditional lists, NumPy arrays allow each member of the list to be manipulated with ease. The following steps will enable you to complete this exercise:

1. Open a new Jupyter Notebook.
2. Now, import `numpy` and create an `ndarray` containing all values between `1` and `100` using `arange`:

```
import numpy as np
np.arange(1, 101)
```

   The output will be as follows:

```
array([  1,   2,   3,   4,   5,   6,   7,   8,   9,  10,  11,  12,  13,
        14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,  26,
        27,  28,  29,  30,  31,  32,  33,  34,  35,  36,  37,  38,  39,
        40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  50,  51,  52,
        53,  54,  55,  56,  57,  58,  59,  60,  61,  62,  63,  64,  65,
        66,  67,  68,  69,  70,  71,  72,  73,  74,  75,  76,  77,  78,
        79,  80,  81,  82,  83,  84,  85,  86,  87,  88,  89,  90,  91,
        92,  93,  94,  95,  96,  97,  98,  99, 100])
```

Figure 10.4 – Showing ndarray with values between 1 to 100

3. Reshape the array to `20` rows and `5` columns:

```
np.arange(1, 101).reshape(20,5)
```

The output will be as follows:

```
array([[  1,   2,   3,   4,   5],
       [  6,   7,   8,   9,  10],
       [ 11,  12,  13,  14,  15],
       [ 16,  17,  18,  19,  20],
       [ 21,  22,  23,  24,  25],
       [ 26,  27,  28,  29,  30],
       [ 31,  32,  33,  34,  35],
       [ 36,  37,  38,  39,  40],
       [ 41,  42,  43,  44,  45],
       [ 46,  47,  48,  49,  50],
       [ 51,  52,  53,  54,  55],
       [ 56,  57,  58,  59,  60],
       [ 61,  62,  63,  64,  65],
       [ 66,  67,  68,  69,  70],
       [ 71,  72,  73,  74,  75],
       [ 76,  77,  78,  79,  80],
       [ 81,  82,  83,  84,  85],
       [ 86,  87,  88,  89,  90],
       [ 91,  92,  93,  94,  95],
       [ 96,  97,  98,  99, 100]])
```

Figure 10.5 – Output with the reshaped array of 20 rows and 5 columns

4. Now, define `mat1` as a `20` x `5` array between `1` and `100` and then subtract `50` from `mat1`, as shown in the following code snippet:

```
mat1 = np.arange(1, 101).reshape(20,5)
mat1 - 50
```

The output will be as follows:

```
array([[-49, -48, -47, -46, -45],
       [-44, -43, -42, -41, -40],
       [-39, -38, -37, -36, -35],
       [-34, -33, -32, -31, -30],
       [-29, -28, -27, -26, -25],
       [-24, -23, -22, -21, -20],
       [-19, -18, -17, -16, -15],
       [-14, -13, -12, -11, -10],
       [ -9,  -8,  -7,  -6,  -5],
       [ -4,  -3,  -2,  -1,   0],
       [  1,   2,   3,   4,   5],
       [  6,   7,   8,   9,  10],
       [ 11,  12,  13,  14,  15],
       [ 16,  17,  18,  19,  20],
       [ 21,  22,  23,  24,  25],
       [ 26,  27,  28,  29,  30],
       [ 31,  32,  33,  34,  35],
       [ 36,  37,  38,  39,  40],
       [ 41,  42,  43,  44,  45],
       [ 46,  47,  48,  49,  50]])
```

Figure 10.6 – Output of subtracting values from the array

5. Now, multiply `mat1` by `10` and observe the change in the output:

```
mat1 * 10
```

The output will be as follows:

```
array([[  10,   20,   30,   40,   50],
       [  60,   70,   80,   90,  100],
       [ 110,  120,  130,  140,  150],
       [ 160,  170,  180,  190,  200],
       [ 210,  220,  230,  240,  250],
       [ 260,  270,  280,  290,  300],
       [ 310,  320,  330,  340,  350],
       [ 360,  370,  380,  390,  400],
       [ 410,  420,  430,  440,  450],
       [ 460,  470,  480,  490,  500],
       [ 510,  520,  530,  540,  550],
       [ 560,  570,  580,  590,  600],
       [ 610,  620,  630,  640,  650],
       [ 660,  670,  680,  690,  700],
       [ 710,  720,  730,  740,  750],
       [ 760,  770,  780,  790,  800],
       [ 810,  820,  830,  840,  850],
       [ 860,  870,  880,  890,  900],
       [ 910,  920,  930,  940,  950],
       [ 960,  970,  980,  990, 1000]])
```

Figure 10.7 – Output when you multiply mat1 by 10

6. Now, add `mat1` to itself, as shown in the following code snippet:

```
mat1 + mat1
```

The output will be as follows:

```
array([[  2,   4,   6,   8,  10],
       [ 12,  14,  16,  18,  20],
       [ 22,  24,  26,  28,  30],
       [ 32,  34,  36,  38,  40],
       [ 42,  44,  46,  48,  50],
       [ 52,  54,  56,  58,  60],
       [ 62,  64,  66,  68,  70],
       [ 72,  74,  76,  78,  80],
       [ 82,  84,  86,  88,  90],
       [ 92,  94,  96,  98, 100],
       [102, 104, 106, 108, 110],
       [112, 114, 116, 118, 120],
       [122, 124, 126, 128, 130],
       [132, 134, 136, 138, 140],
       [142, 144, 146, 148, 150],
       [152, 154, 156, 158, 160],
       [162, 164, 166, 168, 170],
       [172, 174, 176, 178, 180],
       [182, 184, 186, 188, 190],
       [192, 194, 196, 198, 200]])
```

Figure 10.8 – Output of adding mat1 to itself

7. Now, multiply each entry in `mat1` by itself:

```
mat1*mat1
```

The output will be as follows:

```
array([[    1,     4,     9,    16,    25],
       [   36,    49,    64,    81,   100],
       [  121,   144,   169,   196,   225],
       [  256,   289,   324,   361,   400],
       [  441,   484,   529,   576,   625],
       [  676,   729,   784,   841,   900],
       [  961,  1024,  1089,  1156,  1225],
       [ 1296,  1369,  1444,  1521,  1600],
       [ 1681,  1764,  1849,  1936,  2025],
       [ 2116,  2209,  2304,  2401,  2500],
       [ 2601,  2704,  2809,  2916,  3025],
       [ 3136,  3249,  3364,  3481,  3600],
       [ 3721,  3844,  3969,  4096,  4225],
       [ 4356,  4489,  4624,  4761,  4900],
       [ 5041,  5184,  5329,  5476,  5625],
       [ 5776,  5929,  6084,  6241,  6400],
       [ 6561,  6724,  6889,  7056,  7225],
       [ 7396,  7569,  7744,  7921,  8100],
       [ 8281,  8464,  8649,  8836,  9025],
       [ 9216,  9409,  9604,  9801, 10000]])
```

Figure 10.9 – Output of multiplying mat1 by itself

8. Now, take the `dot` product of `mat1` and `mat1.T`, which is the equivalent of matrix multiplication:

```
np.dot(mat1, mat1.T)
```

The output will be as follows:

```
array([[   55,   130,   205,   280,   355,   430,   505,   580,   655,
          730,   805,   880,   955,  1030,  1105,  1180,  1255,  1330,
         1405,  1480],
       [  130,   330,   530,   730,   930,  1130,  1330,  1530,  1730,
         1930,  2130,  2330,  2530,  2730,  2930,  3130,  3330,  3530,
         3730,  3930],
       [  205,   530,   855,  1180,  1505,  1830,  2155,  2480,  2805,
         3130,  3455,  3780,  4105,  4430,  4755,  5080,  5405,  5730,
         6055,  6380],
       [  280,   730,  1180,  1630,  2080,  2530,  2980,  3430,  3880,
         4330,  4780,  5230,  5680,  6130,  6580,  7030,  7480,  7930,
         8380,  8830],
       [  355,   930,  1505,  2080,  2655,  3230,  3805,  4380,  4955,
         5530,  6105,  6680,  7255,  7830,  8405,  8980,  9555, 10130,
        10705, 11280],
       [  430,  1130,  1830,  2530,  3230,  3930,  4630,  5330,  6030,
         6730,  7430,  8130,  8830,  9530, 10230, 10930, 11630, 12330,
        13030, 13730],
       [  505,  1330,  2155,  2980,  3805,  4630,  5455,  6280,  7105,
         7930,  8755,  9580, 10405, 11230, 12055, 12880, 13705, 14530,
        15355, 16180],
       [  580,  1530,  2480,  3430,  4380,  5330,  6280,  7230,  8180,
         9130, 10080, 11030, 11980, 12930, 13880, 14830, 15780, 16730,
        17680, 18630],
       [  655,  1730,  2805,  3880,  4955,  6030,  7105,  8180,  9255,
        10330, 11405, 12480, 13555, 14630, 15705, 16780, 17855, 18930,
        20005, 21080],
       [  730,  1930,  3130,  4330,  5530,  6730,  7930,  9130, 10330,
        11530, 12730, 13930, 15130, 16330, 17530, 18730, 19930, 21130,
        22330, 23530],
       [  805,  2130,  3455,  4780,  6105,  7430,  8755, 10080, 11405,
        12730, 14055, 15380, 16705, 18030, 19355, 20680, 22005, 23330,
        24655, 25980],
       [  880,  2330,  3780,  5230,  6680,  8130,  9580, 11030, 12480,
        13930, 15380, 16830, 18280, 19730, 21180, 22630, 24080, 25530,
        26980, 28430],
       [  955,  2530,  4105,  5680,  7255,  8830, 10405, 11980, 13555,
        15130, 16705, 18280, 19855, 21430, 23005, 24580, 26155, 27730,
        29305, 30880],
```

Figure 10.10 – Truncated output of the dot product of mat1 and mat1

In this exercise, you computed and added values to an n-dimensional array, after which you implemented different NumPy computations.

When it comes to data analysis, NumPy will make your life easier. The ease with which NumPy arrays may be mathematically combined, manipulated, and used to compute standard statistical measures such as the mean, median, and standard deviation makes them far superior to Python lists. They handle big data exceptionally well, and it's hard to imagine the world of data science without them. In the next section, we will be covering pandas, Python's state-of-the-art library for storing, retrieving, analyzing, and manipulating big data.

The pandas library

pandas is the Python library that handles data on all fronts. pandas can import data, read data, and display data in an object called a **DataFrame**. A DataFrame consists of rows and columns. It's designed to look good and perform fast computations to make sense of big data.

In the IT industry, pandas is widely used for data manipulation. It is also used for stock prediction, data storage and retrieval, statistical analysis, cleaning data, and general data science.

In the following exercises, you will begin working with DataFrames by creating them, accessing them, viewing them, and performing different computations on them.

Exercise 135 – using DataFrames to manipulate stored student test score data

In this exercise, you will create a **dictionary**, which is one way to create a `pandas` DataFrame. You will then manipulate this data as required. To use pandas, you must import `pandas`, which is universally imported as `pd`. The following steps will enable you to complete this exercise:

1. Begin by importing `pandas` as `pd`:

```
import pandas as pd
```

   Now that you have imported `pandas`, you must create a DataFrame.

2. Create a dictionary of test scores called `test_dict`:

```
# Create dictionary of test scores
test_dict = {'Scotty':[63,75,88], 'Joy':[48,98,92],
'Kamala': [87, 86, 85]}
```

3. Next, you must place `test_dict` into the DataFrame using the `DataFrame` method:

```
# Create DataFrame
df = pd.DataFrame(test_dict)
```

4. Now, you can display the Dataframe:

```
# Display DataFrame
df
```

The output will be as follows:

	Scotty	Joy	Kamala
0	63	48	87
1	75	98	86
2	88	92	85

Figure 10.11 – Output with the values added to the DataFrame

You can inspect the DataFrame visually. First, each dictionary key is listed as a column. Second, the rows are labeled with indices starting with `0` by default. Third, the visual layout is clear and legible.

Each column and row of the DataFrame is officially represented as a pandas **series**. A series is a one-dimensional `ndarray`.

5. Now, you must rotate the DataFrame, which is also known as a **transpose**, a standard `pandas` method. A transpose turns rows into columns and columns into rows. Copy the code shown in the following code snippet to perform a transpose on the DataFrame:

```
# Transpose DataFrame
df = df.T
df
```

The output will be as follows:

	0	1	2
Scotty	63	75	88
Joy	48	98	92
Kamala	87	86	85

Figure 10.12 – The output of the transpose on the DataFrame

In this exercise, you created a DataFrame that holds the values of `testscores`, and to finish, you transposed this DataFrame to get a rotated output. In the next exercise, you will rename column names and select data from the DataFrame, which is essential to working with pandas.

Exercise 136 – DataFrame computations with the student test score data

In this exercise, you will rename the columns of the DataFrame, and you will select some data to display. The steps are as follows:

1. Open a new Jupyter Notebook.
2. Import `pandas` as `pd` and enter the student value, as shown in *Exercise 135 – using DataFrames to manipulate stored student test score data*. After this, convert it into a DataFrame and transpose it:

```
import pandas as pd
# Create dictionary of test scores
test_dict = {'Scotty':[63,75,88], 'Joy':[48,98,92],
'Kamala': [87, 86, 85]}
# Create DataFrame
df = pd.DataFrame(test_dict)
df = df.T
```

3. Now, rename the columns to something more precise. You can use `.columns` on the DataFrame to rename the column names:

```
# Rename Columns
df.columns = ['Quiz_1', 'Quiz_2', 'Quiz_3']
df
```

The output will be as follows:

	Quiz_1	Quiz_2	Quiz_3
Scotty	63	75	88
Joy	48	98	92
Kamala	87	86	85

Figure 10.13 – Output with changed column names

4. Now, select a range of values from specific rows and columns. You will be using `.iloc` with the index number, which is a pandas method that uses the same `[rows, columns]` syntax as in NumPy arrays. This is shown in the following code as you select the first row:

```
# Access first row by index number
df.iloc[0]
```

The output will be as follows:

```
Quiz_1    63
Quiz_2    75
Quiz_3    88
Name: Scotty, dtype: int64
```

Figure 10.14 – Output displaying the first row of data as a pandas series

5. Now, select a column using its name, as shown in the following code snippet.

   You can access columns by putting the column name in quotes, inside brackets:

```
# Access first column by name
df['Quiz_1']
```

   The output will be as follows:

```
Scotty    63
Joy       48
Kamala    87
Name: Quiz_1, dtype: int64
```

Figure 10.15 – Output displaying the first column of data as a pandas series

6. Now, select a column using the dot (.) notation:

```
# Access first column using dot notation
df.Quiz_1
```

   The output will be as follows:

```
Scotty    63
Joy       48
Kamala    87
Name: Quiz_1, dtype: int64
```

Figure 10.16 – The same output after selecting a column using dot notation

Note
There are limitations to using dot notation, so bracket quotations are often preferable.

In this exercise, you implemented and changed the column names of the DataFrame, and then used `.iloc`, column names, and dot notation to select columns and rows of data from the DataFrame.

In the next exercise, you will implement different computations on DataFrames.

Exercise 137 – more computations on DataFrames

In this exercise, you will use the same `testscore` data and perform more computations on the DataFrame. The following steps will enable you to complete this exercise:

1. Open a new Jupyter Notebook.
2. Import `pandas` as `pd` and enter the student value, as shown in *Exercise 136 – DataFrame computations with the student test score data*. After this, convert it into a DataFrame:

```
import pandas as pd
# Create dictionary of test scores
test_dict = {'Scotty':[63,75,88], 'Joy':[48,98,92],
'Kamala': [87, 86, 85]}
# Create DataFrame
df = pd.DataFrame(test_dict)
```

3. Now, begin by arranging the rows of the DataFrame, as shown in the following code snippet.

   You can use the same bracket notation, `[]`, for rows as for lists and strings:

```
# Limit DataFrame to first 2 rows
df[0:2]
```

   The output will be as follows:

	Scotty	Joy	Kamala
0	63	48	87
1	75	98	86

Figure 10.17 – Output showing the DataFrame's first two rows only

4. Transpose the DataFrame and rename the columns `Quiz_1`, `Quiz_2`, and `Quiz_3`, as covered in *Exercise 136 – DataFrame computations with the student test score data*:

```
df = df.T
df
# Rename Columns
```

```
df.columns = ['Quiz_1', 'Quiz_2', 'Quiz_3']
df
```

The output will be as follows:

	Quiz_1	Quiz_2	Quiz_3
Scotty	63	75	88
Joy	48	98	92
Kamala	87	86	85

Figure 10.18 – DataFrame of quiz scores with column names updated

5. Now, define a new DataFrame from the first two rows and the last two columns only.

   You can choose the rows and columns by name first using the `.loc` notation, as shown in the following code snippet:

```
# Define new DataFrame - first 2 rows,  last 2 columns
rows = ['Scotty', 'Joy']
cols = ['Quiz_2', 'Quiz_3']
df_spring = df.loc[rows, cols]
df_spring
```

   The output will be as follows:

	Quiz_2	Quiz_3
Scotty	75	88
Joy	98	92

Figure 10.19 – Output of the new DataFrame showing two columns and two rows only by name

Note

When selecting rows and columns by index, use the `.iloc` notation. When selecting rows and columns by name, use the `.loc` notation.

6. Now, select the first two rows and the last two columns using index numbers.

   You can use `.iloc` to select rows and columns by index, as shown in the following code snippet:

```
# Select first 2 rows and last 2 columns using index
numbers
df.iloc[[0,1], [1,2]]
```

   The output will be as follows:

	Quiz_2	Quiz_3
Scotty	75	88
Joy	98	92

Figure 10.20 – Same output of selecting the first two rows and last two columns using index numbers

   Now, add a new column to find the quiz average of our students.

   You can generate new columns in a variety of ways. One way is to use available methods such as the mean. In pandas, it's important to specify the axis. An axis of 0 represents the column, while an axis of 1 represents the rows.

7. Now, create a new column as the mean, as shown in the following code snippet:

```
# Define new column as mean of other columns
df['Quiz_Avg'] = df.mean(axis=1)
df
```

   The output will be as follows:

	Quiz_1	Quiz_2	Quiz_3	Quiz_Avg
Scotty	63	75	88	75.333333
Joy	48	98	92	79.333333
Kamala	87	86	85	86.000000

Figure 10.21 – Adding a new Quiz_Avg column to the output

   A new column can also be added as a list by choosing the rows and columns by name first.

8. Create a new column as a list, as shown in the following code snippet:

```
df['Quiz_4'] = [92, 95, 88]
df
```

The output will be as follows:

	Quiz_1	Quiz_2	Quiz_3	Quiz_Avg	Quiz_4
Scotty	63	75	88	75.333333	92
Joy	48	98	92	79.333333	95
Kamala	87	86	85	86.000000	88

Figure 10.22 – Output with a newly added column using lists

What if you need to delete the column you created? You can do so by using the `del` function. It's easy to delete columns in pandas using `del`.

9. Now, delete the `Quiz_Avg` column as it is not needed anymore:

```
del df['Quiz_Avg']
df
```

The output will be as follows:

	Quiz_1	Quiz_2	Quiz_3	Quiz_4
Scotty	63	75	88	92
Joy	48	98	92	95
Kamala	87	86	85	88

Figure 10.23 – Output after deleting the Quiz_Avg column

In this exercise, you implemented different ways to add and remove columns as per your requirements. In the next section, you will be looking at new rows and `NaN`, an official `numpy` designation that often appears in data analytics.

New rows and NaN

Say you have a new student who joins the class for the fourth quiz. What values should you put for the other three quizzes? The answer is NaN. This stands for **Not a Number**.

`NaN` is an official NumPy term. It can be accessed using `np.NaN`. It is case-sensitive, so the first and last *N*s must be capitalized and the middle *a* must be lowercase. In later exercises, you will look at different strategies for changing NaN.

Exercise 138 – concatenating and finding the mean with null values for our test score data

In this exercise, you will be concatenating and finding the mean with null values for the student `testscore` data you created in *Exercise 137 – more computations on DataFrames*, with four quiz scores. The following steps will enable you to complete this exercise:

1. Open a new Jupyter Notebook.
2. Import `pandas` and `numpy` and create a dictionary containing the `testscore` data to be transformed into a DataFrame, as shown in *Exercise 135 – using DataFrames to manipulate stored student test score data*:

```
import pandas as pd
# Create dictionary of test scores
test_dict = {'Scotty':[63,75,88], 'Joy':[48,98,92],
'Kamala': [87, 86, 85]}
# Create DataFrame
df = pd.DataFrame(test_dict)
# Transpose the DataFrame
df = df.T
df
# Rename Columns
df.columns = ['Quiz_1', 'Quiz_2', 'Quiz_3']
# Add Quiz 4
df['Quiz_4'] = [92, 95, 88]
df
```

The output will be as follows:

	Quiz_1	Quiz_2	Quiz_3	Quiz_4
Scotty	63	75	88	92
Joy	48	98	92	95
Kamala	87	86	85	88

Figure 10.24 – DataFrame output

3. Now, add a new row using the `.loc` notation by setting the index to `Adrian`, along with null values for the first three quizzes, but a score of 71 for the fourth quiz, as shown in the following code snippet:

```
import numpy as np
df.loc['Adrian']=[np.NaN, np.NaN, np.NaN, 71]
df
```

The output will be as follows:

	Quiz_1	Quiz_2	Quiz_3	Quiz_4
Scotty	63.0	75.0	88.0	92.0
Joy	48.0	98.0	92.0	95.0
Kamala	87.0	86.0	85.0	88.0
Adrian	NaN	NaN	NaN	71.0

Figure 10.25 – Output with a new row added to the DataFrame

You can now compute the new mean, but you must skip the `NaN` values; otherwise, there will be no mean score for `Adrian`.

4. Find the `mean` value while ignoring NaN and use these values to create a new column named `Quiz_Avg`, as shown in the following code snippet:

```
df['Quiz_Avg'] = df.mean(axis=1, skipna=True)
df
```

The output will be as follows:

	Quiz_1	Quiz_2	Quiz_3	Quiz_4	Quiz_Avg
Scotty	63.0	75.0	88.0	92	79.50
Joy	48.0	98.0	92.0	95	83.25
Kamala	87.0	86.0	85.0	88	86.50
Adrian	NaN	NaN	NaN	71	71.00

Figure 10.26 – Output with the mean, which skips over null values

Notice that all values are floats (NaN is a float!). You can use `df.dtypes` to check the data types of the columns of the DataFrame.

Casting column types

Cast all the floats in `Quiz_4` that you used in *Exercise 138 – concatenating and finding the mean with null values for our test score data*, as ints using the following code snippet:

```
df.Quiz_4.astype(int)
```

```
Scotty    92
Joy       95
Kamala    88
Adrian    71
Name: Quiz_4, dtype: int64
```

Note that to change the DataFrame itself, you must use `df['Quiz_4']=df.Quiz_4.astype(int)`. Now, let's move on to the next topic, which is working with big data.

Working with big data

Now that you have been introduced to NumPy and pandas, you will use them to analyze real data of a much larger size. The phrase big data does not have an unambiguous meaning. Generally speaking, you can think of big data as data that is far too large to analyze by sight. It could contain tens of thousands, millions, billions, trillions, or even more rows of data.

Data scientists analyze data that exists in the cloud or online. One strategy to analyze real data is to download the data directly to your computer.

> **Note**
>
> It is recommended to create a new folder called `Data` to store all of the data that you will download for analysis. You can open your Jupyter Notebook in this same folder.

Downloading data

Data comes in many formats, and pandas is equipped to handle most of them. In general, when looking for data to analyze, it's worth searching for the keyword "dataset." A dataset is a collection of raw data that has been stored for others to access. Online, "data" is everywhere, whereas datasets are limited to data in its raw format.

You will start by examining the famous Boston Housing dataset from 1980, which is available in this book's GitHub repository.

This dataset can be found at `https://github.com/PacktPublishing/The-Python-Workshop-Second-Edition/tree/main/Datasets`.

If you have downloaded all of the GitHub files for this book to your computer, you already have the dataset in your files. Otherwise, you will need to download the dataset onto your local computer, as described in the next section.

Downloading the Boston Housing data from GitHub

Here are the steps to download the Boston Housing data:

1. Head to this book's GitHub repository at `https://github.com/PacktPublishing/The-Python-Workshop-Second-Edition` and download the dataset onto your local computer.
2. Move the downloaded dataset file into your data folder. It's useful to have a data folder to store all of your data.
3. Open a Jupyter Notebook in the same folder. Alternatively, if you open a Jupyter Notebook in your home directory, you can scroll to the `Data` folder on your Jupyter home page and create a new notebook from there:

Figure 10.27 – Jupyter Notebook filesystem with the new Data folder included

Reading data

Now that the data has been downloaded, and the Jupyter Notebook is open, you are ready to read the file. The most important part of reading a file is the extension. Our file is a `.csv` file. Therefore, you need a method for reading `.csv` files.

CSV stands for **Comma-Separated Values**. CSV files are a popular way of storing and retrieving data, and pandas handles them very well.

Here is a list of standard data files that pandas will read, along with the code for reading data:

type of file	code
csv files:	pd.read_csv('file_name')
excel files:	pd.read_excel('file_name')
feather files:	pd.read_feather('file_name')
html files:	pd.read_html('file_name')
json files:	pd.read_json('file_name')
sql database:	pd.read_sql('file_name')

Figure 10.28 – Standard data files that pandas can read

If the files are clean, pandas will read them properly. Sometimes, files are not clean, so you may need to change the function parameters. It's advisable to copy any errors and search for solutions online.

A further point of consideration is that the data should be read into a DataFrame. Pandas will convert the data into a DataFrame upon reading it, but you need to save the DataFrame as a variable.

> **Note**
> `df` is often used to store DataFrames, but it's not universal since you may be dealing with many DataFrames.

In the next exercise, you will be using the Boston Housing dataset and performing basic actions on the data.

Exercise 139 – reading and viewing the Boston Housing dataset

In this exercise, your goal is to read and view the Boston Housing dataset in your Jupyter Notebook. The following steps will enable you to complete this exercise:

1. Open a new Jupyter Notebook.
2. Import `pandas` as `pd`:

```
import pandas as pd
```

3. Now, choose a variable for storing the DataFrame and place the `HousingData.csv` file in the folder for this exercise. Then, run the following command:

```
df = pd.read_csv('HousingData.csv')
```

If no errors arise, the file has been read properly. Now, you can examine and view the file.

Note

If you are unable to access the data, you can use `housing_df = pd.read_csv('https://raw.githubusercontent.com/PacktWorkshops/The-Python-Workshop/master/Datasets/HousingData.csv')`. This is a special access point for raw CSV files that works in limited cases, with raw GitHub files being one of them.

4. Now, view the file by entering the following command:

```
df.head()
```

`.head` displays the first five rows of the DataFrame. You may view more rows by placing the number of your choice in parentheses.

The output will be as follows:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	NaN	36.2

Figure 10.29 – df.head() displaying the first five rows of the dataset

Before you perform operations on this dataset, you may be wondering what column names such as `CRIM` and `ZN` mean.

Here is a list of the columns, along with their meanings:

```
CRIM     per capita crime rate by town
ZN       proportion of residential land zoned for lots over 25,000 sq. ft.
INDUS    proportion of non-retail business acres per town
CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
NOX      nitric oxide concentration (parts per 10 million)
RM       average number of rooms per dwelling
AGE      proportion of owner-occupied units built prior to 1940
DIS      weighted distances to five Boston employment centers
RAD      index of accessibility to radial highways
TAX      full-value property-tax rate per $10,000
PTRATIO pupil-teacher ratio by town
LSTAT    % lower status of the population
MEDV     median value of owner-occupied homes in $1,000s
```

Figure 10.30 – Representation of the column values of the dataset

Now that you know what the columns in the dataset mean, you will perform operations on the DataFrame in the following exercise.

Exercise 140 – gaining data insights on the Boston Housing dataset

In this exercise, you will be performing some more advanced operations and using pandas methods to understand the dataset and get the desired insights. The following steps will enable you to complete this exercise:

1. Open a new Jupyter Notebook where your Boston Housing data is stored.
2. Import pandas, choose a variable where you will store the DataFrame, and read in the `HousingData.csv` file:

```
import pandas as pd
df = pd.read_csv('HousingData.csv')
```

3. Now, use the `describe()` method to display the key statistical measures of each column, including the mean, median, and quartiles, as shown in the following code snippet:

```
df.describe()
```

The truncated output will be as follows:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B
count	486.000000	486.000000	486.000000	486.000000	506.000000	506.000000	486.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.611874	11.211934	11.083992	0.069959	0.554695	6.284634	68.518519	3.795043	9.549407	408.237154	18.455534	356.674032
std	8.720192	23.388876	6.835896	0.255340	0.115878	0.702617	27.999513	2.105710	8.707259	168.537116	2.164946	91.294864
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000	0.320000
25%	0.081900	0.000000	5.190000	0.000000	0.449000	5.885500	45.175000	2.100175	4.000000	279.000000	17.400000	375.377500
50%	0.253715	0.000000	9.690000	0.000000	0.538000	6.208500	76.800000	3.207450	5.000000	330.000000	19.050000	391.440000
75%	3.560263	12.500000	18.100000	0.000000	0.624000	6.623500	93.975000	5.188425	24.000000	666.000000	20.200000	396.225000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000	396.900000

Figure 10.31 – The truncated output of descriptive statistics with df.describe()

In this output, you must review the meaning of each row:

- `count`: The number of rows with actual values.
- `mean`: The sum of each entry divided by the number of entries. It is often a good estimate of the average.
- `std`: The number of unit entries that are expected to deviate from the mean. It is a good measure of spread.
- `min`: The smallest entry in each column.

- `25%`: The first quartile. 25% of the data has a value less than this number.
- `50%`: The median. The halfway marker of the data. This is another good estimate of the average.
- `75%`: The third quartile. 75% of the data has a value less than this number.
- `max`: The largest entry in each column.

4. Now, use the `info()` method to deliver a full list of columns, along with their types and the number of null values.

   `info()` is especially valuable when you have hundreds of columns, and it takes a long time to horizontally scroll through each one:

   ```
   df.info()
   ```

   The output will be as follows:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   CRIM     486 non-null    float64
 1   ZN       486 non-null    float64
 2   INDUS    486 non-null    float64
 3   CHAS     486 non-null    float64
 4   NOX      506 non-null    float64
 5   RM       506 non-null    float64
 6   AGE      486 non-null    float64
 7   DIS      506 non-null    float64
 8   RAD      506 non-null    int64
 9   TAX      506 non-null    int64
 10  PTRATIO  506 non-null    float64
 11  B        506 non-null    float64
 12  LSTAT    486 non-null    float64
 13  MEDV     506 non-null    float64
dtypes: float64(12), int64(2)
memory usage: 55.5 KB
```

Figure 10.32 – Output of df.info()

As you can see, `df.info()` reveals the count of non-null values in each column, along with the column type. Since some columns have less than 506 non-null values, you can infer that the other values are null.

In this dataset, there's a total of 506 rows and 14 columns. You can use the `.shape` attribute to obtain this information directly.

Now, confirm the number of rows and columns in the dataset:

```
df.shape
```

The output will be as follows:

```
(506, 14)
```

This confirms that you have 506 rows and 14 columns. Notice that `shape` does not have any parentheses after it. This is because it's technically an attribute and pre-computed.

In this exercise, you performed key operations on the entire dataset, such as finding the descriptive statistics, finding columns with null values, and finding the number of rows and columns.

In the next section, we will cover null values.

Null values

You need to do something about the null values. They will break machine learning algorithms (see *Chapter 11*, *Machine Learning*) that rely on numerical values as input. There are several popular choices when dealing with null values:

- Eliminate the rows. This is a respectable approach if null values are a very small percentage – that is, around 1% of the total dataset.
- Replace the null value with a significant value, such as the median or the mean. This is a great approach if the rows are valuable, and the column itself is reasonably balanced.
- Replace the null value with the most likely value, perhaps a 0 or 1. This is preferable to averages when the median or mean might be unrealistic based on other factors.

> **Note**
> **Mode** is the official term for the value that occurs the greatest number of times.

As you can see, which option you choose depends on the data. That's a general theme that rings true for data science: no one method fits all; your choice of action will ultimately depend on the data at hand.

Exercise 141 – viewing null values

In this exercise, you will view null values in the DataFrame. Follow these steps:

1. Open a new Jupyter Notebook where your Boston Housing data is stored.

2. Import `pandas` and choose a variable for storing the DataFrame to read in the `HousingData.csv` file:

```
import pandas as pd
df = pd.read_csv('HousingData.csv')
```

3. Now, find the values and columns in the dataset with `null` values, as shown in the following code snippet:

```
df.isnull().any()
```

The output will be as follows:

```
CRIM        True
ZN          True
INDUS       True
CHAS        True
NOX        False
RM         False
AGE         True
DIS        False
RAD        False
TAX        False
PTRATIO    False
B          False
LSTAT       True
MEDV       False
dtype: bool
```

Figure 10.33 – Output of the columns with null values

The `.isnull()` method will display an entire DataFrame of `True`/`False` values, depending on the `Null` value. Feel free to give it a try.

4. The `.any()` method returns the individual columns, as shown in *Figure 10.33*. Now, using the DataFrame, find the null columns.

You can use `.loc` to find the location of particular rows. Select the first five rows and all of the columns that have null values, as shown in the following code snippet:

```
df.loc[:5, df.isnull().any()]
```

The output will be as follows:

	CRIM	ZN	INDUS	CHAS	AGE	LSTAT
0	0.00632	18.0	2.31	0.0	65.2	4.98
1	0.02731	0.0	7.07	0.0	78.9	9.14
2	0.02729	0.0	7.07	0.0	61.1	4.03
3	0.03237	0.0	2.18	0.0	45.8	2.94
4	0.06905	0.0	2.18	0.0	54.2	NaN
5	0.02985	0.0	2.18	0.0	58.7	5.21

Figure 10.34 – DataFrame of columns with null values

5. Now, for the final step. Use the `.describe()` method on the null columns of the dataset:

```
df.loc[:, df.isnull().any()].describe()
```

This code can be broken down as follows:

- `df` is the DataFrame
- `.loc` allows you to specify rows and columns
- `:` selects all rows
- `df.isnull().any()` selects only columns with null values
- `.describe()` pulls up the statistics

The output will be as follows:

	CRIM	ZN	INDUS	CHAS	AGE	LSTAT
count	486.000000	486.000000	486.000000	486.000000	486.000000	486.000000
mean	3.611874	11.211934	11.083992	0.069959	68.518519	12.715432
std	8.720192	23.388876	6.835896	0.255340	27.999513	7.155871
min	0.006320	0.000000	0.460000	0.000000	2.900000	1.730000
25%	0.081900	0.000000	5.190000	0.000000	45.175000	7.125000
50%	0.253715	0.000000	9.690000	0.000000	76.800000	11.430000
75%	3.560263	12.500000	18.100000	0.000000	93.975000	16.955000
max	88.976200	100.000000	27.740000	1.000000	100.000000	37.970000

Figure 10.35 – Descriptive statistics of the columns with null values

Consider the first column, `CRIM`. The mean is way more than the median (`50%`). This indicates that the data is very right-skewed with some outliers since outliers pull the mean away from the median. Indeed, you can see that the maximum of `88.97` is much larger than the `3.56` value of the 75th percentile. This makes the mean a poor replacement candidate for this column.

It turns out that the median is a good candidate for replacing null values in all columns shown. Although the median is not better than the mean in some cases, there are a few cases where the mean is worse (`CRIM`, `ZN`, and `CHAS`).

The choice for replacing null values depends on what you ultimately want to do with the data. If the goal is straightforward data analysis, eliminating the rows with null values is worth considering. However, if the goal is to use machine learning to predict data, then perhaps more is to be gained by changing the null values into suitable replacements.

A more thorough examination could be warranted, depending on the data. For instance, when analyzing new medical drugs, it would be worth putting more time and energy into appropriately dealing with null values. You may want to perform more analysis to determine whether a value is 0 or 1, depending on other factors.

In this particular case, replacing all the null values with the median is warranted. For the sake of practice, however, let's replace the null values with various values in the following section.

Replacing null values

pandas includes a nice method, `fillna`, which can be used to replace null values. It works for individual columns and entire DataFrames. You will use three approaches: replacing the null values of a column with the mean, replacing the null values of a column with another value, and replacing all the null values in the entire dataset with the median.

Open the same Jupyter Notebook that you used in *Exercise 141 – viewing null values*. Ensure that all cells have been run since you opened the notebook.

Here are the steps to transform the null values:

1. Replace the null values in the `AGE` column with `mean`:

```
df['AGE'] =df['AGE'].fillna(df.mean())
```

2. Replace the null values in the `CHAS` column with `0`:

```
df['CHAS'] = df['CHAS'].fillna(0)
```

3. Replace all remaining null values with `median` for the respective columns:

```
df = df.fillna(df.median())
```

4. Finally, check that all null values have been replaced:

```
df.info()
```

The output will be as follows:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   CRIM     506 non-null    float64
 1   ZN       506 non-null    float64
 2   INDUS    506 non-null    float64
 3   CHAS     506 non-null    float64
 4   NOX      506 non-null    float64
 5   RM       506 non-null    float64
 6   AGE      506 non-null    float64
 7   DIS      506 non-null    float64
 8   RAD      506 non-null    int64
 9   TAX      506 non-null    int64
 10  PTRATIO  506 non-null    float64
 11  B        506 non-null    float64
 12  LSTAT    506 non-null    float64
 13  MEDV     506 non-null    float64
dtypes: float64(12), int64(2)
memory usage: 55.5 KB
```

Figure 10.36 – df.info() revealing no null values

Since all columns contain 506 non-null values, which is the total number of rows, you can infer that all null values have been eliminated. After eliminating all null values, the dataset is much cleaner. There may still be problematic outliers that may lead to poor predictions. These can sometimes be detected through visual analysis, which we will cover in the next section.

Creating statistical graphs

Most people interpret data visually. They prefer to view colorful, meaningful graphs to make sense of the data. As a data science practitioner, it's your job to create and interpret these graphs for others.

In *Chapter 4*, *Extending Python, Files, Errors, and Graphs*, you were introduced to `matplotlib` and many different kinds of graphs. In this section, you will expand upon your knowledge by learning about new techniques to enhance the outputs and information displayed in your histograms and scatterplots. Additionally, you will see how box plots can be used to visualize statistical distributions, and how heat maps can provide nice visual representations of correlations.

In this section, you will use Python – in particular, `matplotlib` and `seaborn` – to create these graphs. Although software packages such as Tableau are rather popular, they are essentially drag-and-drop. Since Python is an all-purpose programming language, the limitations are only what you know and are capable of doing. In other words, Python's graphing libraries give you the capacity to generate any output that you desire.

Histograms

As you have seen, creating a histogram is rather simple. You choose a column and place it inside of `plt.hist()`. The general idea behind a histogram is that it groups an `x` value into various bins. The height of the bin is determined by the number of values that fall into that particular range. By default, `matplotlib` selects 10 bins, but that number may be changed.

In the interest of generating professional graphs, we will use `seaborn` as a backdrop, and you will export the graphs as PNGs using the dots-per-inch of your choice. You must also provide clear labels for readability and choose an optimal size for the graph.

Exercise 142 – creating a histogram using the Boston Housing dataset

In this exercise, you will use `MEDV`, the median value of the Boston Housing dataset, as a future target column for machine learning. The following steps will enable you to complete this exercise:

1. Open a new Jupyter Notebook where your Boston Housing data is stored.
2. Import `pandas` as `pd` and choose a variable for storing the DataFrame while reading in the `HousingData.csv` file:

```
import pandas as pd
df = pd.read_csv('HousingData.csv')
```

3. Import `matplotlib` and `seaborn`, then use `sns.set()` to create a nice gray background with white grid lines for your histogram:

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
```

4. Store a title for your histogram as a variable for the display title, and for saving the output as a PNG:

```
title = 'Median Boston Housing Prices'
```

5. Adjust the figure size of your histogram using (horizontal, vertical) dimensions, as follows:

```
plt.figure(figsize=(14,8))
```

6. Now, create the histogram itself, using a green color since we are talking about money, and set the transparency, or the alpha, at the desired percentage to lessen the brightness and add a 3D effect:

```
plt.hist(df['MEDV'], color='green', alpha=0.6)
```

7. Now, add a title and labels, increasing the font size of your title as follows:

```
plt.title(title, fontsize=15)
plt.xlabel('1980 Median Value in Thousands')
plt.ylabel('Count')
```

8. Finally, export your histogram, making sure to set `dpi`, or dots per inch, to the desired value, and show the histogram itself:

```
plt.savefig(title, dpi=300)
plt.show()
```

Here is a screenshot of the Jupyter Notebook output, which is not the saved figure:

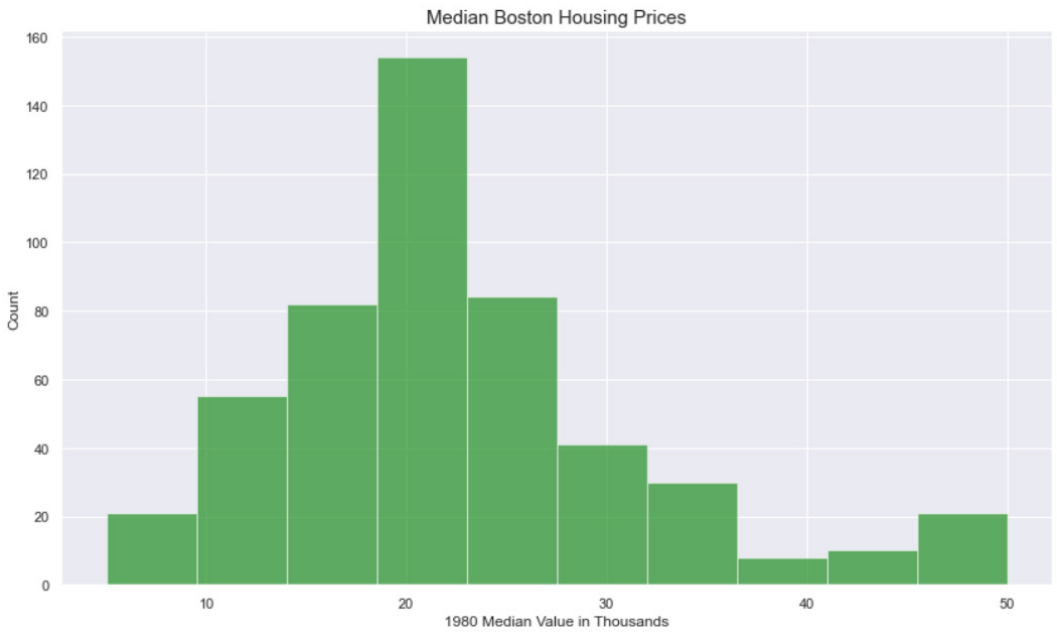


Figure 10.37 – Output of the Jupyter Notebook histogram

To access the saved figure, look in the folder where you created this notebook; it should be there. Just double-click to open it. The saved image will look far better than the Jupyter Notebook output on account of the `dpi` value. Although our input will not capture color and has been degraded since we inserted it into this editor, by printing it on the page, it should look crisper than the previous screenshot:

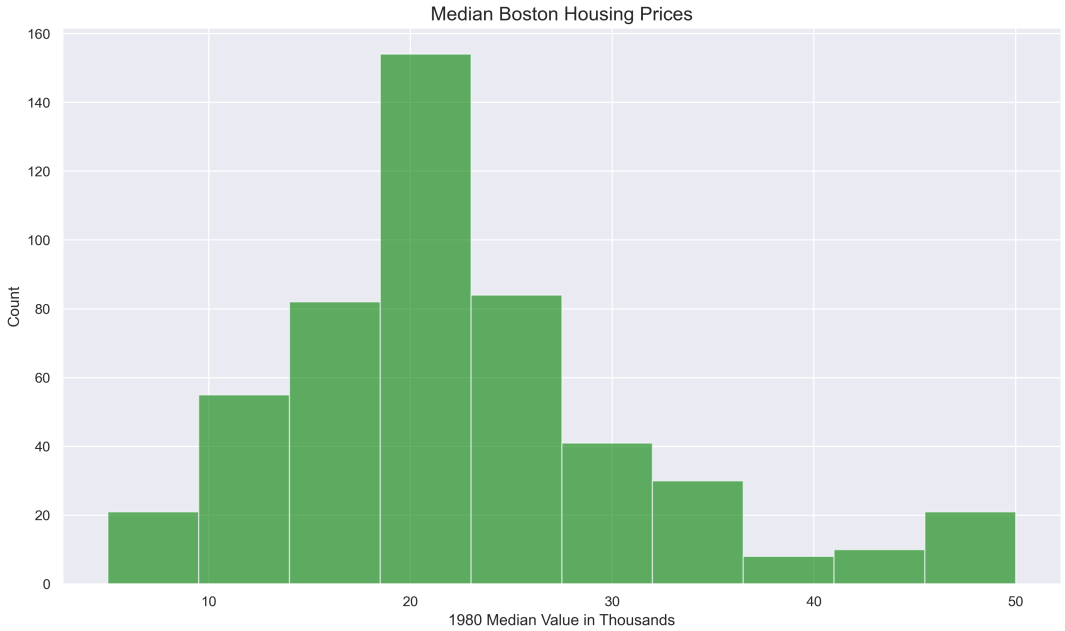


Figure 10.38 – Output of the exported PNG histogram with a 300 dpi value – it looks much stronger on screen!

Now, say you want to create another histogram. Should you keep copying the same code? Copying code repeatedly is never a good idea. It's better to write functions.

We'll create new histograms by implementing a histogram function in the following exercise.

Exercise 143 – creating histogram functions

Creating functions to display graphs is a little different from creating other functions for several reasons:

- Since you want to display the graph, it usually returns nothing
- It's not always clear what pieces of code should change
- There are customization options with the `matplotlib` graphs inside of functions

Instead of creating the most robust functions possible, we will include core concepts to make nice, repeatable histograms. You are encouraged to add customization options as desired.

Here are the steps to create professional-looking histograms using a histogram function:

1. Open the same Jupyter Notebook that you used in *Exercise 142 – creating a histogram using the Boston Housing dataset*.
2. Define your histogram function, as shown in the following code snippet:

```
def my_hist(column, title, xlab, ylab=' Count', color= '
green ', alpha=0.6, bins=10):
    plt.figure(figsize=(14,8))
    plt.hist(column, color=color, alpha=alpha)
    plt.title(title, fontsize=15)
    plt.xlabel(xlab)
    plt.ylabel(ylab)
    plt.savefig(title, dpi=300)
    plt.show()
```

   It's not easy to create functions with `matplotlib`, so let's go over the parameters carefully. `figsize` allows you to establish the size of the figure. `column` is the essential parameter. – it's what you are going to be graphing. Many possible column customizations may be included as parameters (see the official documentation at `https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.hist.html`). We have included `color` and `alpha` here. Next, you have `title`, followed by the labels for the *x* - and *y* - axes.

   Finally, you save the figure and show the plot. Inside this function is the same code that you ran previously.

3. Call the histogram function while using the `RM` column as input:

```
 my_hist(housing_df['RM'], 'Average Number of Rooms
in Boston Households', 'Average Number of Rooms',
color='royalblue')
```

   The output will be as follows:

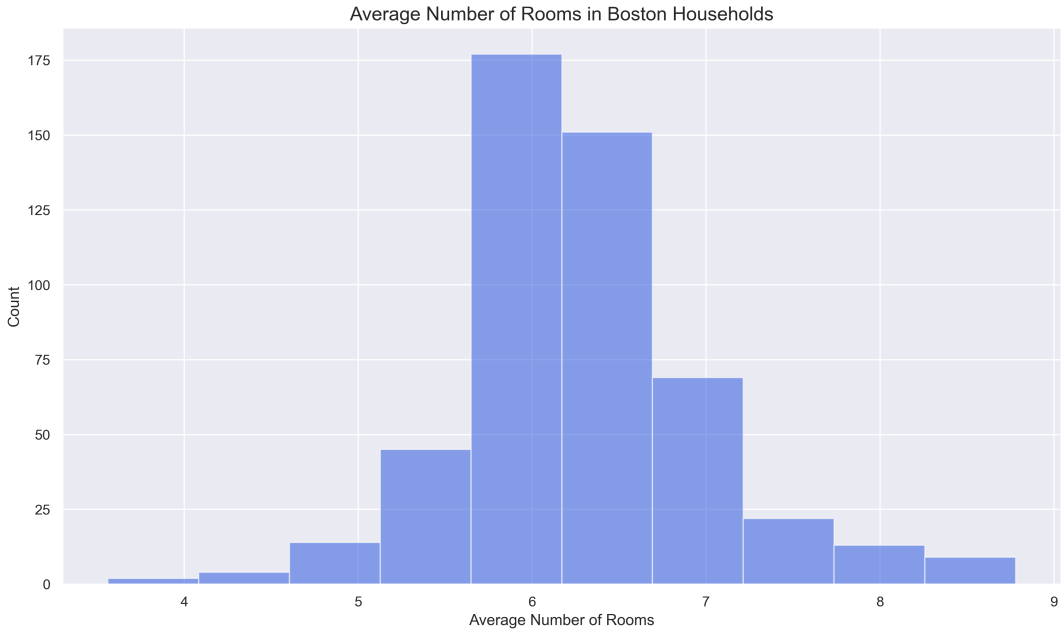


Figure 10.39 – Screenshot output of the histogram function

The output looks solid, but there's one glaring issue: it's the distribution of bins. It seems most rooms have an average of 6, but how many of those are closer to 7? Our graph could be improved if each histogram was clearly between two numbers on the plot. As `df.describe()` previously revealed, the range of rooms is between 3 and 9.

The strategy is to change both the number of bins and the range of the bins so that they fall exactly between the minimum and the maximum.

4. It's not uncommon to modify graphing functions to add customizations. Let's make these adjustments to nicely show the bins falling precisely between two numbers:

```
 def my_hist(column, title, xlab, ylab='Count',
color='green', alpha=0.6, bins=10, range=None):
    plt.figure(figsize=(14,8))
    plt.hist(column, color=color, alpha=alpha, bins=bins,
range=range)
    plt.title(title, fontsize=15)
    plt.xlabel(xlab)
    plt.ylabel(ylab)
    plt.savefig(title, dpi=300)
    plt.show()
```

5. Now, call the improved histogram function:

```
 my_hist(housing_df['RM'], 'Average Number of Rooms
in Boston Households', 'Average Number of Rooms',
color='skyblue', bins=6, range=(3,9))
```

The output will be as follows:

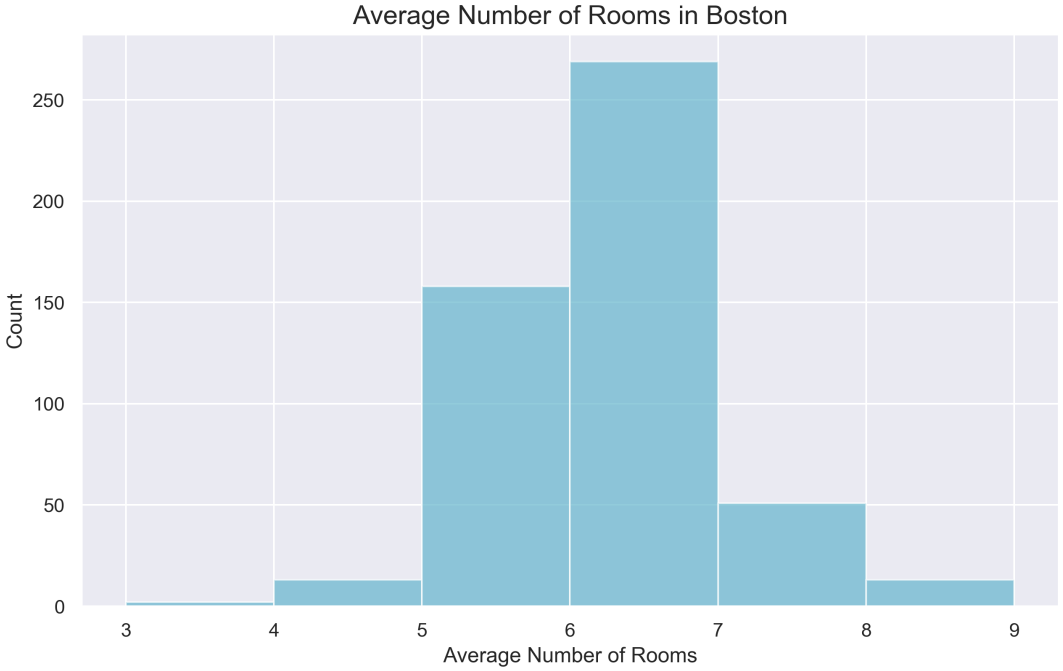


Figure 10.40 – Output of the improved histogram function clearly between the bins

As we can see, the highest average is between 6 and 7 rooms.

Now that you have understood how to create professional histograms using your own functions and customizations, let's shift from one-dimensional to two-dimensional data with scatter plots.

Scatter plots

Scatter plots are often used to compare the relationship between two variables of data. In this section, you will expand upon what you learned in *Chapter 4*, *Extending Python, Files, Errors, and Graphs*. You will create more advanced scatter plots in `seaborn` that include color and size variables to add more information.

Exercise 144 – creating a scatter plot for the Boston Housing dataset

In this exercise, you will create a `seaborn` scatter plot for our Boston Housing dataset that includes color and size variation.

The following steps will enable you to complete this exercise:

1. Open a new Jupyter Notebook where your Boston Housing data is stored.
2. Import `pandas` as `pd` and choose a variable for storing the DataFrame while reading in the `HousingData.csv` file:

```
import pandas as pd
housing_df = pd.read_csv('HousingData.csv')
```

3. Import `matplotlib` and `seaborn`, and set the gray seaborn background with the white grid:

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
```

4. Set the figure size and the title, as shown in the following code snippet:

```
plt.figure(figsize=(16,10))
my_title='Boston Housing Scatterplot'
plt.title(my_title, size=15)
```

Create a `seaborn` scatter plot with the `x` value set to the crime rate and the `y` value set to the median value of the house. Include color variation by setting the `hue` parameter equal to another column, and size variation by setting the `size` parameter equal to an additional column. Adjust the size of the dots as a tuple using the `sizes` parameter, and the color palette using the `palette` parameter. All may be executed as shown in the following code snippet:

```
sns.scatterplot(x=df['CRIM'], y=df['MEDV'],
                hue=df['RM'], size=df['AGE'],
                sizes=(20, 400),
                palette='Greens')
```

5. Save your figure and display the graph using the following code:

```
plt.savefig(my_title, dpi=225)
plt.show()
```

A screenshot of the output is as follows:

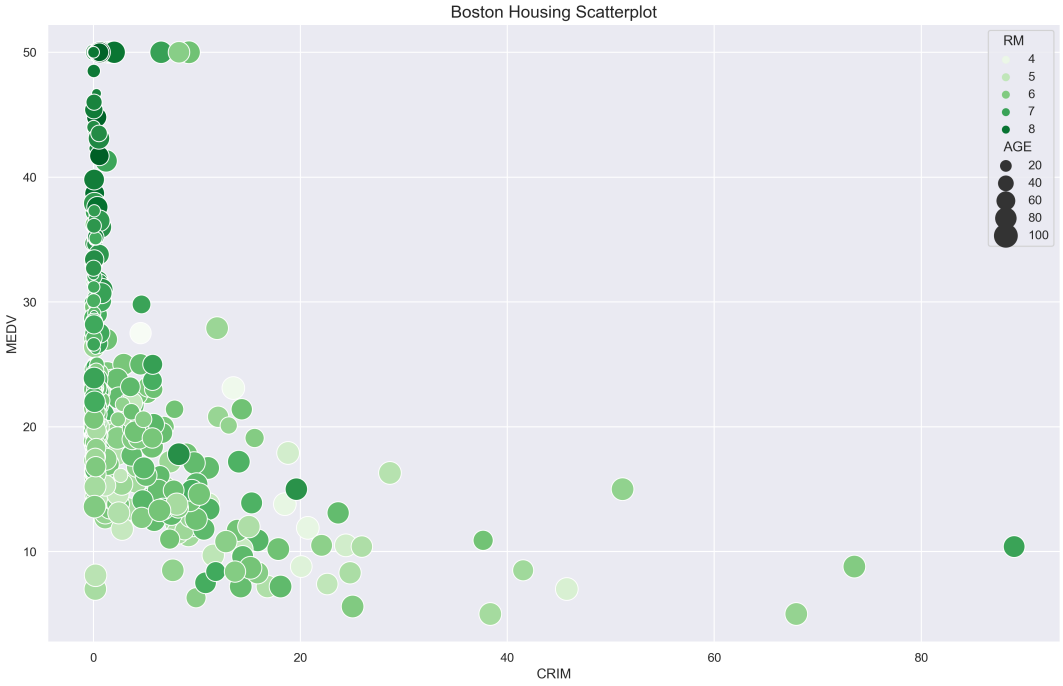


Figure 10.41 – Scatter plot output showing how crime, number of rooms, and age affect the median house value

It's your job as a practicing data scientist to help interpret your graphs. There's a clear negative relationship, as expected, between crime and the median house value. As crime goes up, the median house value goes down. We can also see from the color variation that as the number of rooms goes up, according to the key on the right, the median house values go up. Finally, there does not seem to be much of a relationship between age, according to the key on the right, and the median house value.

`seaborn` comes with many nice default options, as shown in this graph. They automatically include a key, also called a legend, along with the labels of the columns. `seaborn` and `matplotlib` work well together, and can more or less be used interchangeably.

For more information on creating cool Seaborn graphs, click on any of the links in the Seaborn gallery: `https://seaborn.pydata.org/examples/index.html`.

The question of negative and positive association, as referenced earlier, can be more accurately determined by an important statistical concept called **correlation**, which we will examine next.

Correlation

Correlation is a statistical measure between -1 and +1 that indicates how closely two variables are related. A correlation of -1 or +1 means that the variables are dependent, and they fall in a perfectly straight line. A correlation of 0 indicates that an increase in one variable gives no information whatsoever about the other variable. Visually, this would involve points being all over the place. Correlations usually fall somewhere in the middle. For instance, a correlation of 0.75 represents a fairly strong linear relationship, whereas a correlation of 0.25 is a reasonably weak linear relationship. Positive correlations go up (meaning as *x* goes up, *y* goes up), and negative correlations go down.

Here is a great image from Wikimedia Commons that shows a range of scatter plots, along with their correlations:

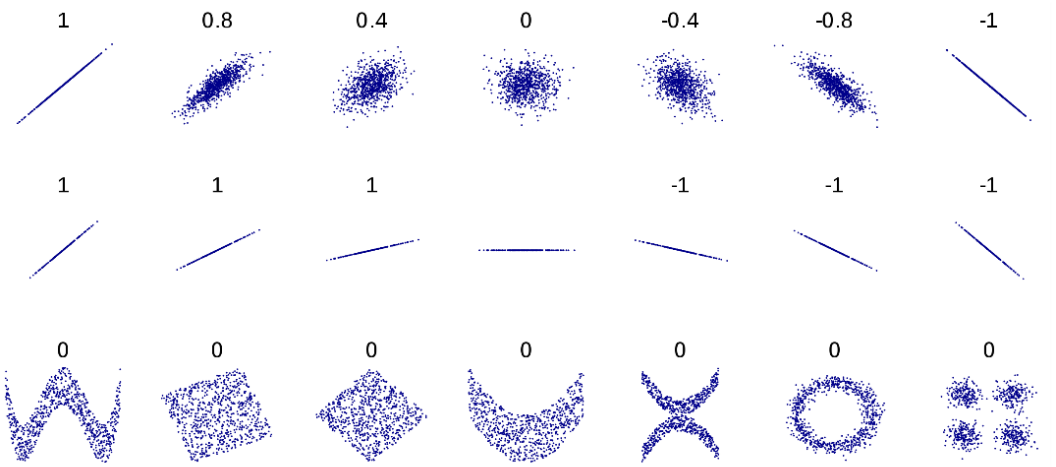


Figure 10.42 – Scatter plots and their respective correlations from Wikimedia Commons

In the following exercise, you will find the correlation values from the Boston Housing dataset and display them using a heatmap.

Exercise 145 – correlation values from the dataset

In this exercise, you will find the correlation values from the Boston Housing dataset and display them using a heat map. The following steps will enable you to complete this exercise:

1. Open a new Jupyter Notebook where your Boston Housing data is stored.

2. Import `pandas` as `pd` and choose a variable for storing the DataFrame while reading in the `HousingData.csv` file:

```
import pandas as pd
df = pd.read_csv('HousingData.csv')
```

3. Now, find the correlation value of the dataset, as shown in the following code snippet:

```
df.corr()
```

The output will be as follows:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
CRIM	1.000000	-0.191178	0.401863	-0.054355	0.417130	-0.219150	0.354342	-0.374166	0.624765	0.580595	0.281110	-0.381411	0.444943	-0.391363
ZN	-0.191178	1.000000	-0.531871	-0.037229	-0.513704	0.320800	-0.563801	0.656739	-0.310919	-0.312371	-0.414046	0.171303	-0.414193	0.373136
INDUS	0.401863	-0.531871	1.000000	0.059859	0.764866	-0.390234	0.638431	-0.711709	0.604533	0.731055	0.390954	-0.360532	0.590690	-0.481772
CHAS	-0.054355	-0.037229	0.059859	1.000000	0.075097	0.104885	0.078831	-0.093971	0.001468	-0.032304	-0.111304	0.051264	-0.047424	0.181391
NOX	0.417130	-0.513704	0.764866	0.075097	1.000000	-0.302188	0.731548	-0.769230	0.611441	0.668023	0.188933	-0.380051	0.582641	-0.427321
RM	-0.219150	0.320800	-0.390234	0.104885	-0.302188	1.000000	-0.247337	0.205246	-0.209847	-0.292048	-0.355501	0.128069	-0.614339	0.695360
AGE	0.354342	-0.563801	0.638431	0.078831	0.731548	-0.247337	1.000000	-0.744844	0.458349	0.509114	0.269226	-0.275303	0.602891	-0.394656
DIS	-0.374166	0.656739	-0.711709	-0.093971	-0.769230	0.205246	-0.744844	1.000000	-0.494588	-0.534432	-0.232471	0.291512	-0.493328	0.249929
RAD	0.624765	-0.310919	0.604533	0.001468	0.611441	-0.209847	0.458349	-0.494588	1.000000	0.910228	0.464741	-0.444413	0.479541	-0.381626
TAX	0.580595	-0.312371	0.731055	-0.032304	0.668023	-0.292048	0.509114	-0.534432	0.910228	1.000000	0.460853	-0.441808	0.536110	-0.468536
PTRATIO	0.281110	-0.414046	0.390954	-0.111304	0.188933	-0.355501	0.269226	-0.232471	0.464741	0.460853	1.000000	-0.177383	0.375966	-0.507787
B	-0.381411	0.171303	-0.360532	0.051264	-0.380051	0.128069	-0.275303	0.291512	-0.444413	-0.441808	-0.177383	1.000000	-0.369889	0.333461
LSTAT	0.444943	-0.414193	0.590690	-0.047424	0.582641	-0.614339	0.602891	-0.493328	0.479541	0.536110	0.375966	-0.369889	1.000000	-0.735822
MEDV	-0.391363	0.373136	-0.481772	0.181391	-0.427321	0.695360	-0.394656	0.249929	-0.381626	-0.468536	-0.507787	0.333461	-0.735822	1.000000

Figure 10.43 – Correlation values displayed as a DataFrame

This tells us the exact correlation values. For instance, to see what variables are the most correlated with `Median Value Home`, you can examine the values under the `MEDV` column. There, you will find that `RM` is the largest at `0.695360`. But you also see a value of `-0.735822` for `LSTAT`, which is the percentage of the lower status of the population. This is a very strong negative correlation.

4. Seaborn provides a nice way to view correlations inside of a heatmap. Begin by importing `maplotlib` and `seaborn`:

```
import matplotlib.pyplot as plt
import seaborn as snssns.set()
```

5. Now, display the heatmap, as shown in the following code snippet:

```
corr = df.corr()
plt.figure(figsize=(14,10))
sns.heatmap(corr, xticklabels=corr.columns.values,
```

```
yticklabels=corr.columns.values, cmap="Reds",
linewidths=1.25)
plt.show()
```

A screenshot of the output is as follows:

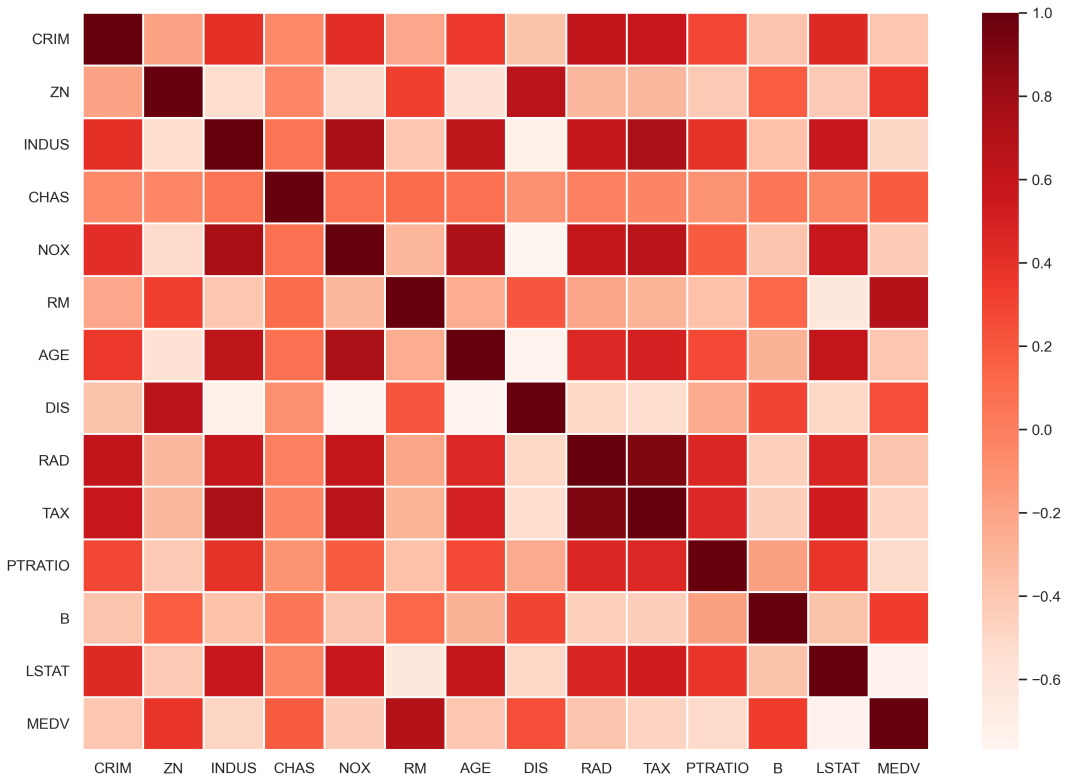


Figure 10.44 – Heatmap for the correlation values

The darker the squares, the higher the correlation, and the lighter the squares, the lower the correlation. Now, when examining the `MEDV` column, it's much easier to find the darkest square, `RM`, and the lightest square, `LSTAT`. You may have noticed that, technically, the `MEDV` square is the darkest. This has to be true because `MEDV` is perfectly correlated with itself. The same holds for each column along the diagonal.

In this exercise, you were able to work with correlation values from the dataset and get a visual aid for the data output.

In the next section, you will learn about regression.

Regression

Perhaps the most important addition to a scatter plot is the regression line. The idea of regression came from Sir Francis Galton, who measured the heights of the offspring of very tall and very short parents. The offspring were not taller or shorter than their parents on average, but rather closer to the mean height of all people. Sir Francis Galton used the term "regression to the mean," meaning that the heights of the offspring were closer to the mean of their very tall or very short parents. The name stuck.

In statistics, a regression line is a line that tries to fit the values of a scatter plot as closely as possible. Generally speaking, half of the points are above the line, and half of the points are below. The most popular regression line method is ordinary least squares, which minimizes the sum of the square of the distance from each point to the line.

There are a variety of methods to compute and display regression lines using Python.

Plotting a regression line

To create a regression line of our Boston Housing dataset, the following code steps need to be followed:

```
x = df['RM']
y = df['MEDV']
plt.figure(figsize=(14, 10))
sns.regplot(x,y)
plt.show()
```

The output will be as follows:

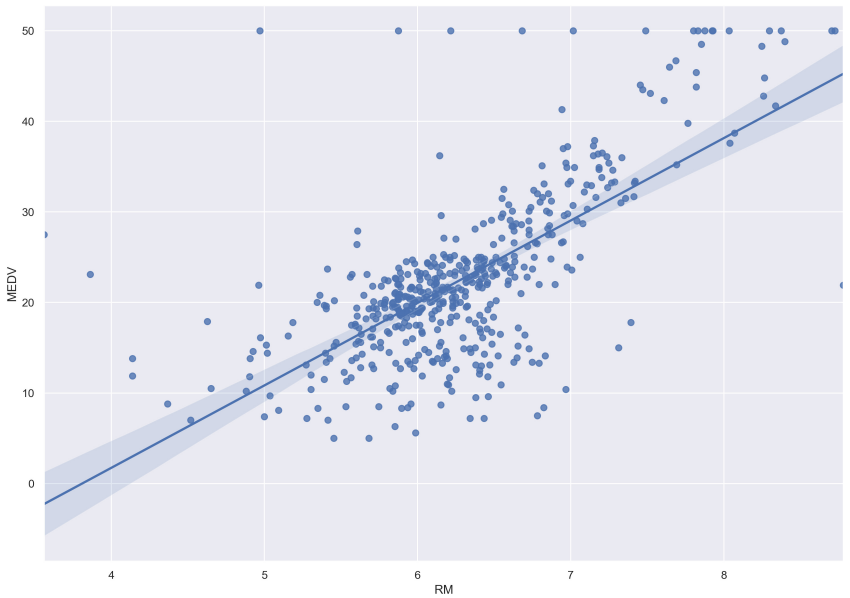


Figure 10.45 – Regression line predicting MEDV from RM with the 95% confidence interval shaded in

You may be wondering about the shaded part of the line. It represents a 95% confidence interval, meaning that Python is 95% confident that the actual regression line falls within that range. Since the shaded area is fairly small concerning the plot, this means that the regression line is reasonably accurate. Note that Seaborn provides a 95% confidence interval by default, and this number can be adjusted if desired.

The general idea behind regression lines is that they can be used to predict new *y* values from new *x* values. For instance, if there is an eight-room house, you can use regression to get an estimate of its value. You will use this general philosophy in a more sophisticated manner in *Chapter 11*, *Machine Learning*, using the machine learning version of linear regression.

Although this is not a course on statistics, if you want to go a little deeper, check out the next section, which explains how to obtain all of the key parameters of the ordinary least squares regression line. As a practicing data scientist, it's important to be able to obtain this information if needed.

StatsModel regression output

Import `StatsModel` and use its methods to print out a summary of the regression line:

```
import statsmodels.api as sm
X = sm.add_constant(x)
model = sm.OLS(y, X)
est = model.fit()
print(est.summary())
```

The strangest part of the code is adding the constant. This is the *y*-intercept. When the constant is not added, the *y*-intercept is 0. In our case, it makes sense that the *y*-intercept would be 0; if there are 0 rooms, the house should have no value. In general, however, it's a good idea to keep a *y*-intercept, and it's the default choice of the preceding Seaborn graph. It's a good idea to try both methods and compare the results of the data. A comparative analysis will improve your background in statistics. Finally, note that OLS stands for **ordinary least squares**, as described in the preceding section.

The expected output is as follows:

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   MEDV   R-squared:                       0.484
Model:                            OLS   Adj. R-squared:                  0.483
Method:                 Least Squares   F-statistic:                     471.8
Date:                Sat, 03 Sep 2022   Prob (F-statistic):           2.49e-74
Time:                        19:37:20   Log-Likelihood:                -1673.1
No. Observations:                 506   AIC:                             3350.
Df Residuals:                     504   BIC:                             3359.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const        -34.6706      2.650    -13.084      0.000     -39.877     -29.465
RM             9.1021      0.419     21.722      0.000       8.279       9.925
==============================================================================
Omnibus:                      102.585   Durbin-Watson:                   0.684
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              612.449
Skew:                           0.726   Prob(JB):                    1.02e-133
Kurtosis:                       8.190   Cond. No.                         58.4
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```

Figure 10.46 – Summary of the regression line

There's a lot of important information in this table. The first is the value of `R^2` at `0.484`. This suggests that 48% of the data can be explained by the regression line. The second is the coefficient constant of `-34.6706`. This is the *y*-intercept. The third is the RM coefficient of `9.1021`. This suggests that for every one-bedroom increase, the value of the house increased by 9,102 (keep in mind that this dataset is from 1980).

The standard error suggests how far off the actual values are from the line on average, and the numbers underneath the `[0.025 0.975]` column give the 95% **confidence interval** of the value, meaning `statsmodel` is 95% confident that the true increase in the value of the average house for every one-bedroom increase is between 8,279 and 9,925.

Box plots and violin plots

There's a great deal of data analysis in Python – far more than you can adequately cover in an introductory text. In this chapter, you covered histograms and scatter plots in considerable detail, including regression lines and heat maps. In *Chapter 4*, *Extending Python, Files, Errors, and Graphs*, you also covered line charts, bar charts, pie charts, density plots, contour plots, and 3D plots.

Two additional types of plots – box plots and violin plots – will be briefly highlighted before we move on to machine learning.

Exercise 146 – creating box plots

A box plot provides a nice visual of the mean, median, quartiles, and outliers of a given column of data.

In this exercise, you will create box plots using the Boston Housing dataset. The following steps will enable you to complete this exercise:

1. Open a new Jupyter Notebook where your Boston Housing data is stored.
2. Import `pandas` as `pd` and choose a variable for storing the DataFrame while reading in the `HousingData.csv` file:

```
import pandas as pd
housing_df = pd.read_csv('HousingData.csv')
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
```

3. Now, enter the following code to create a box plot:

```
plt.figure(figsize=(14, 10))
title='Box Plot of Boston Median House Values'
plt.title(title, size=15)
sns.boxplot(x = df['MEDV'])
plt.savefig(title, dpi=300)
plt.show()
```

Here is the output of the saved PNG file:

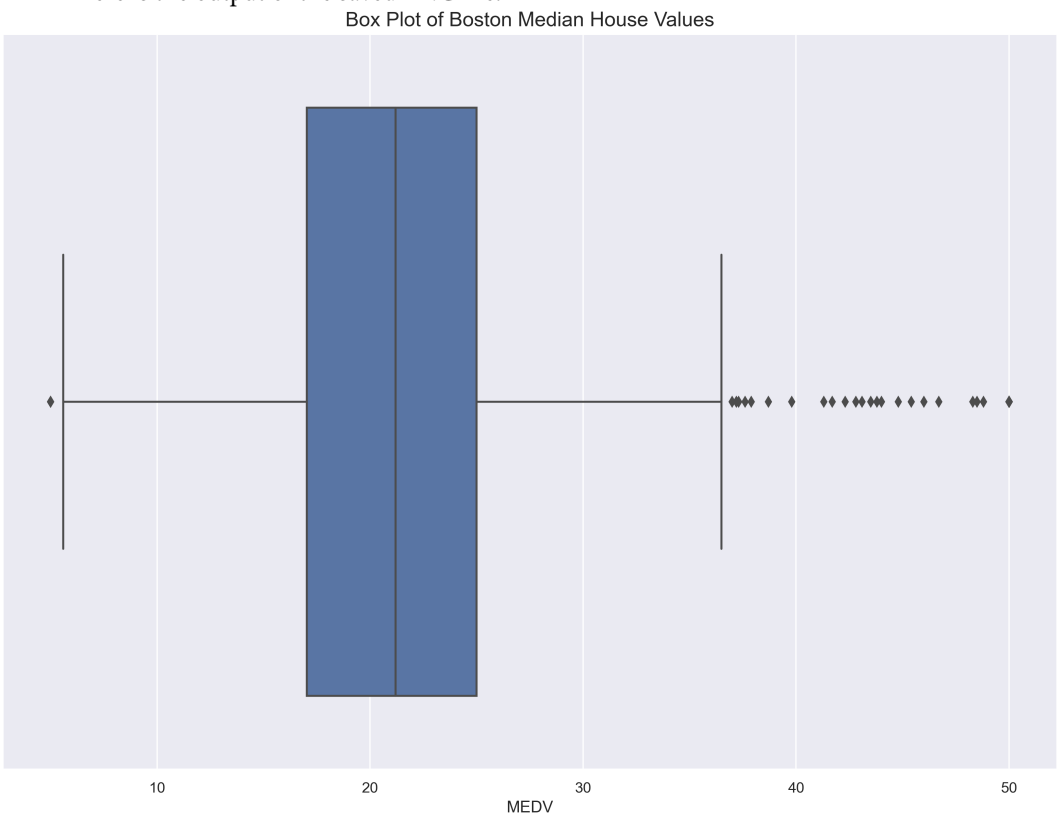


Figure 10.47 – Seaborn box plot output

Note that the small diamonds are considered outliers. The line in the middle is the median, while the bars at the end of the dark box are the 25th and 75th percentiles, or the first and third quartiles. The end bars represent the quartiles plus or minus 1.5 times the interquartile range. The value of 1.5 times the interquartile range is a standard limit in statistics that's used to define outliers, but it does not have universal acceptance since outliers must be judged, depending on the data at hand.

In this exercise, you created a box plot graph to visually represent a column of data.

Exercise 147 – creating violin plots

A violin plot is a different type of plot that conveys similar information as a box plot. In this exercise, you will create a violin plot by performing the following steps:

1. Continue with the same Jupyter Notebook as in the previous exercise.

2. Enter the following code to create the violin plot:

```
plt.figure(figsize=(14, 10))
sns.violinplot(x = df['MEDV'])
plt.show()
```

The output will be as follows:

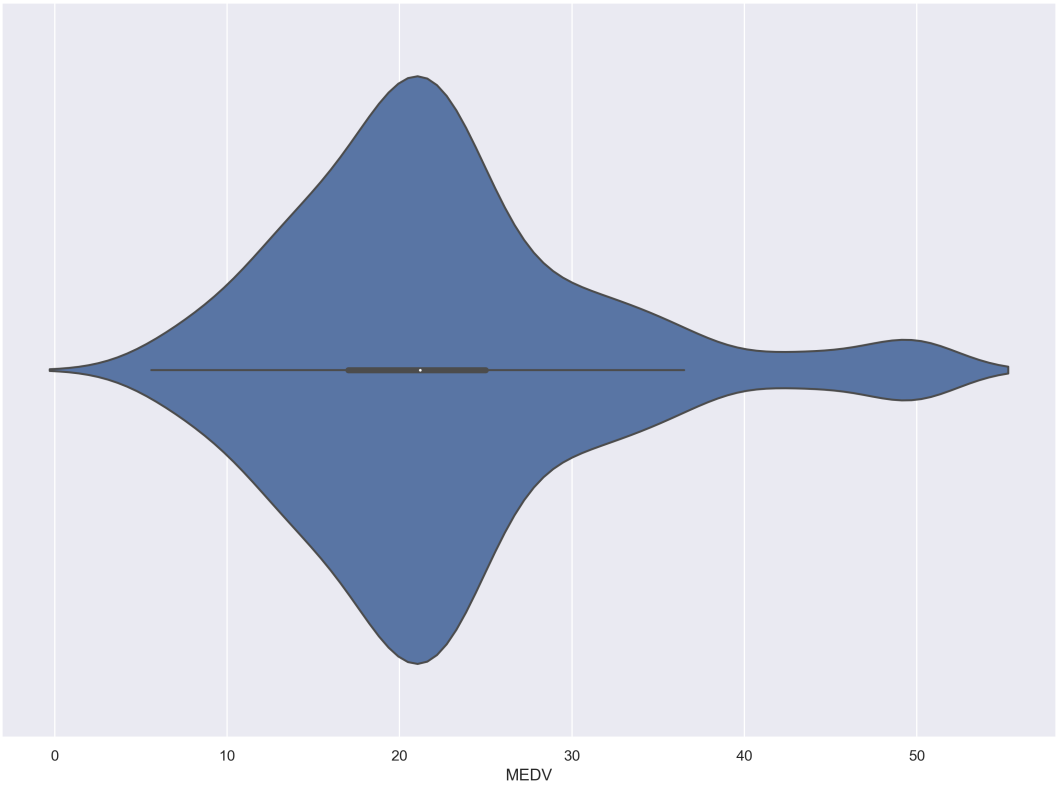


Figure 10.48 – Violin plot output

In the violin plot, the right and left edges define the minimum and maximum values, and the width of the plot indicates how many rows contain that particular value. The difference between the violin plot and the box plot is that the violin plot shows the overall distribution of the data as one continuous graph, whereas the box plot is partitioned.

That concludes our survey of visual graphs.

Now, you will complete an activity to see whether you can implement the concepts covered in this chapter.

Activity 24 – performing data analysis to find the outliers in pay versus the salary report in the UK statistics dataset

You are working as a data scientist, and you come across a government dataset that seems interesting concerning payments. But since the dataset values are cluttered, you need to use visual data analysis to study the data and determine whether any outliers need to be removed.

In this activity, you will be performing visual data analysis using histograms, scatter plots, regression lines, and box plots to arrive at your conclusion.

Follow these steps to complete this activity:

1. First, you need to copy the `UKStatistics.csv` dataset file into a specific folder.
2. Next, in a new Jupyter Notebook, import the necessary data visualization packages.
3. View the dataset file, then find the information about each column and the descriptive statistics.
4. Plot the histogram for `Actual Pay Floor (£)`.
5. Plot the scatter plot while using `x` as `Salary Cost of Reports (£)` and `y` as `Actual Pay Floor (£)`.
6. Now, get the box plot for the `x` and `y` values, as shown in *step 5*.

> **Note**
>
> `UKStatistics.csv` can be downloaded on GitHub at `https://github.com/PacktPublishing/The-Python-Workshop-Second-Edition/tree/main/Chapter10/`.
>
> More information on the `UKStatistics` dataset can be found at `https://packt.live/2BzBwqF`.

Here is the expected output with the outliers in one of the box plots:

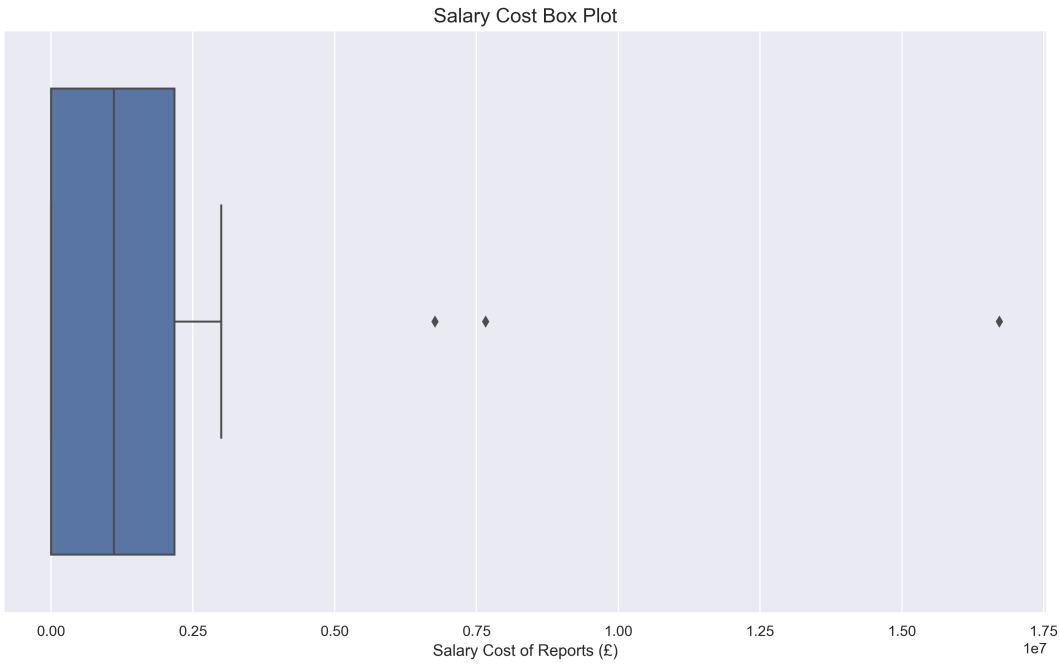


Figure 10.49 – The expected output for the box plot of the Salary Cost of Reports revealing outliers

Note

The solution for this activity can be found in the *Appendix* on GitHub.

Summary

We began our introduction to data analysis with NumPy, Python's incredibly fast library for handling massive matrix computations. Next, you learned about the fundamentals of pandas, Python's library for handling DataFrames. Taken together, you used NumPy and pandas to analyze the Boston Housing dataset by correcting null values and interpreting descriptive statistics, including the mean, standard deviation, median, quartiles, correlation, skewed data, and outliers. You also learned about advanced methods for creating clean, clearly labeled, publishable graphs, including histograms, scatter plots with variation in size and color, regression lines, box plots, and violin plots. You now have the fundamental skills to load, clean, analyze, and plot big data for technical and general audiences.

In *Chapter 11*, *Machine Learning*, you will make predictions from big data using some of the best machine learning algorithms in the world today.

11
Machine Learning

Overview

By the end of this chapter, you will be able to apply **machine learning** (**ML**) algorithms to solve different problems; compare, contrast, and apply different types of ML algorithms, including linear regression, logistic regression, decision trees, random forests, Naive Bayes, Adaptive Boosting (AdaBoost), and Extreme Gradient Boosting (XGBoost); analyze overfitting and implement regularization; work with `GridSearchCV` and `RandomizedSearchCV` to adjust hyperparameters; evaluate algorithms using a confusion matrix and cross-validation, and solve real-world problems using the ML algorithms outlined here.

Introduction

Computer algorithms enable machines to learn from data. The more data an algorithm receives, the more capable the algorithm is of detecting underlying patterns within the data. In *Chapter 10*, *Data Analytics with pandas and NumPy*, you learned how to view and analyze big data with `pandas` and `NumPy`. In this chapter, we will now extend these concepts to algorithms that learn from data.

Consider how a child learns to identify a cat. Generally speaking, a child learns by having someone point out "That's a cat", "No, that's a dog", and so on. After enough cats and non-cats have been pointed out, the child knows how to identify a cat.

ML implements the same general approach. A **convolutional neural network** (**CNN**) is an ML algorithm that distinguishes between images. Upon receiving images labeled cats and non-cats, the algorithm looks for underlying patterns within the pixels by adjusting the parameters of an equation until it finds an equation that minimizes the error.

After the algorithm has selected the best possible equation, given the data it has received, this equation is used to predict future data. When a new image arrives, the new image is placed within the algorithm to determine whether the image is a cat or not.

In this chapter on ML, you will learn how to construct linear regression, logistic regression, decision tree, random forest, Naive Bayes, AdaBoost, and XGBoost algorithms. These algorithms can be used to solve a wide range of problems, from predicting rainfall to detecting credit card fraud and identifying diseases.

Then, you will learn about Ridge and Lasso, two regularized ML algorithms that are variations of linear regression. You will learn about using regularization and cross-validation to obtain accurate results with data that the algorithm has never seen before.

After learning how to build an ML model in `scikit-learn` through an extended example with linear regression, you will take a similar approach to build models based on k-nearest neighbors, (KNN) decision trees, and random forests. You will learn how to extend these models with hyperparameter turning, a way of fine-tuning models to meet the specifications of the data at hand.

Next, you will move on to classification problems, where the ML model is used to determine whether an email is spam and whether a celestial object is a planet. All classification problems can be tackled with logistic regression, an ML algorithm that you will learn about here. In addition, you will solve classification problems with Naive Bayes, random forests, and other types of algorithms. Classification results can be interpreted with a confusion matrix and a classification report, both of which we will explore in depth.

Finally, you will learn how to implement boosting methods that transform weak learners into strong learners. In particular, you will learn how to implement AdaBoost, one of the most successful ML algorithms in history, and XGBoost, one of the best ML algorithms today.

To sum up, after completing this chapter, you will be able to apply multiple ML algorithms to solve classification and regression problems. You will be capable of using advanced tools such as a confusion matrix and a classification report to interpret results. You will also be able to refine your models using regularization and hyperparameter tuning. In short, you will have the tools to use ML to solve real-world problems, including predicting cost and classifying objects.

Here's a quick overview of the topics covered:

- Introduction to linear regression
- Testing data with cross-validation
- Regularization—Ridge and Lasso
- K-nearest neighbors, decision trees, and random forests
- Classification models
- Boosting algorithms

As for CNNs, you will learn how to build one when you conclude your ML and data science journey in Python at the end of the following chapter, *Chapter 12*, *Deep Learning with Python*.

Technical requirements

You can find the code files for this chapter on GitHub at `https://github.com/PacktPublishing/The-Python-Workshop-Second-Edition/tree/main/Chapter11`.

Introduction to linear regression

ML is the ability of computers to learn from data. The power of ML comes from making future predictions based on the data received. Today, ML is used all over the world to predict the weather, stock prices, profits, errors, clicks, purchases, words to complete a sentence, recommend movies, recognize faces and many more things.

The unparalleled success of ML has led to a paradigm shift in the way businesses make decisions. In the past, businesses made decisions based on who had the most influence, but now, the new idea is to make decisions based on data. Decisions are constantly being made about the future, and ML is the best tool at our disposal to convert raw data into actionable decisions.

The first step in building an ML algorithm is deciding what you want to predict. When looking at a DataFrame, the idea is to choose one column as the **target** column. The target column, by definition, is what the algorithm will be trained to predict.

Recall the Boston Housing dataset introduced in *Chapter 10*, *Data Analytics with pandas and NumPy*. The median value of a home is a desirable target column since real-estate agents, buyers, and sellers often want to know how much a house is worth. People usually determine this information based on the size of the house, the location, the number of bedrooms, and many other factors.

Here is the Boston Housing DataFrame from *Chapter 10*, *Data Analytics with pandas and NumPy*. Each column includes features about houses in the neighborhood, such as crime, the average age of the house, and notably, in the last column, the median value:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	NaN	36.2

Figure 11.1 – Sample from the Boston Housing dataset

The meaning of the columns from *Chapter 10*, *Data Analytics with pandas and NumPy*, is displayed again for your reference:

```
CRIM    per capita crime rate by town
ZN      proportion of residential land zoned for lots over 25,000 sq. ft.
INDUS   proportion of non-retail business acres per town
CHAS    Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
NOX     nitric oxide concentration (parts per 10 million)
RM      average number of rooms per dwelling
AGE     proportion of owner-occupied units built prior to 1940
DIS     weighted distances to five Boston employment centers
RAD     index of accessibility to radial highways
TAX     full-value property-tax rate per $10,000
PTRATIO pupil-teacher ratio by town
LSTAT   % lower status of the population
MEDV    median value of owner-occupied homes in $1,000s
```

Figure 11.2 – Dataset value representation

We want to come up with an equation that uses every other column to predict the last column, which will be our target column. What kind of equation should we use? Before we answer this question, let's have a look at a simplified solution.

Simplifying the problem

It's often helpful to simplify a problem. What if we take just one column, such as the number of bedrooms, and use it to predict the median house value?

It's clear that the more bedrooms a house has, the more valuable it will be. As the number of bedrooms goes up, so does the house value. A standard way to represent this positive association is with a straight line.

In *Chapter 10*, *Data Analytics with pandas and NumPy*, we modeled the relationship between the number of bedrooms and the median house value with the linear regression line, as shown here:

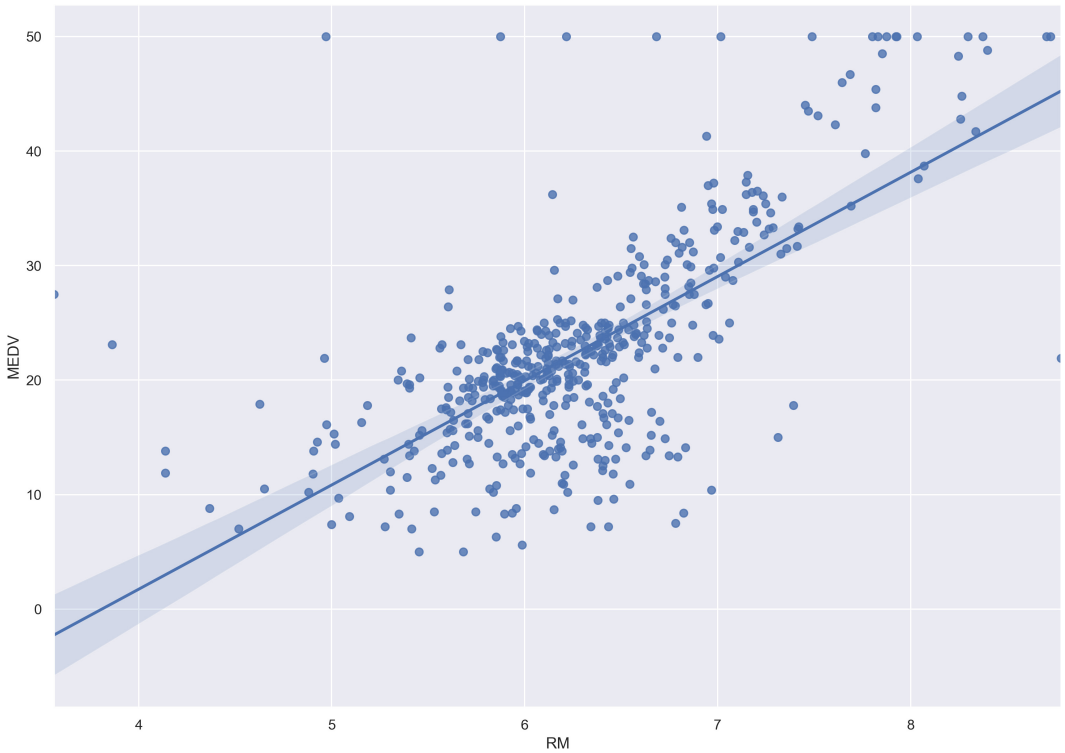


Figure 11.3 – Linear regression line for the median value and the number of bedrooms

It turns out that linear regression is a very popular ML algorithm. Linear regression is often worth trying when the target column takes on continuous values, as in this dataset. The value of a home is generally considered to be continuous. There is technically no limit to how high the cost of a home may be. It could take any value between two numbers, despite often rounding up.

By contrast, if we predict whether a house will sell after 1 month on the market, the possible answers are yes and no. In this case, the target column is not continuous but discrete.

From one to N-dimensions

Dimensionality is an important concept in ML. In math, it's common to work with two dimensions, *x* and *y*, in the coordinate plane. In physics, it's common to work with three dimensions, the *x*, *y*, and *z* axes. When it comes to spatial dimensions, three is the limit because we live in a three-dimensional universe. In mathematics, however, there is no restriction on the number of dimensions we can use theoretically. In superstring theory, 12 or 13 dimensions are often used. In ML, however, the number of dimensions is often the number of **predictor** columns.

There is no need to limit ourselves to one dimension with linear regression. Additional dimensions—in this case, additional columns—will give us more information about the median house value and make our model more valuable.

In one-dimensional linear regression, the slope-intercept equation is $y = mx + b$, where y is the target column, x is the input, m is the slope, and b is the y-intercept. This equation is now extended to an arbitrary number of dimensions using $Y = MX + B$, where Y, M, and X are vectors of arbitrary length. Instead of the slope, M is referred to as the weight.

> **Note**
> It's not essential to comprehend the linear algebra behind vector mathematics to run ML algorithms; however, it is essential to comprehend the underlying ideas. The underlying idea here is that linear regression can be extended to an arbitrary number of dimensions.

In the Boston Housing dataset, a linear regression model will select weights, which are numerical coefficients, for each of the columns. In order to predict the median house value for each row (our target column), the weights will be multiplied by the column entries and then summed, with the goal of getting as close as possible to the target value.

We will have a look at how this works in practice.

The linear regression algorithm

Before implementing the algorithm, let's take a brief look at the libraries that we will import and use in our programs:

- `pandas`: You learned how to use `pandas` in *Chapter 10*, *Data Analytics with pandas and NumPy*. When it comes to ML, in this chapter, all data will be handled through `pandas`. Loading data, reading data, viewing data, cleaning data, and manipulating data all require `pandas`, so `pandas` will always be our first import.
- `NumPy`: This was introduced in *Chapter 10*, *Data Analytics with pandas and NumPy*, as well and will often be used for mathematical computations on datasets. It's often a good idea to import `NumPy` when performing ML so that it's available when needed.
- `LinearRegression`: The `LinearRegression` library should be implemented every time linear regression is used. This library will allow you to build linear regression models and test them in very few steps. ML libraries do the heavy lifting for you. In this case, `LinearRegression` will place weights on each of the columns and adjust them until it finds an optimal solution to predict the target column, which in our case would be the median house value.

- `mean_squared_error`: In order to find optimal values, the algorithm needs a measure to test how well it's doing. Measuring how far the model's predicted value is from the target value is a standard place to start. In order to avoid negatives canceling out positives, we can use `mean_squared_error`. To compute the `mean_squared_error` value, the prediction of each row is subtracted from the target column or actual value, and the result is squared. Each result is summed, and the mean is computed. Finally, taking the square root keeps the units the same.
- `train_test_split`: Python provides `train_test_split` to split data into a **training** set and a **test** set. Splitting the data into a training set and test set is essential because it allows users to test the model right away. Testing the model on data the machine has never seen before is the most important part of building the model because it shows how well the model will perform in the real world.

Most of the data is included in the training set because more data leads to a more robust model. A smaller portion—around **20%**—is held back for the test set. An **80**-**20** split is the default, though you may adjust it as you see fit. The model is optimized on the training set, and after completion, it is scored against the test set.

These libraries are a part of `scikit-learn`, also known as `sklearn`. `scikit-learn` has a wealth of excellent online resources for beginners. See `https://scikit-learn.org/stable/` for more information.

Exercise 148 – using linear regression to predict the accuracy of the median values of our dataset

The goal of this exercise is to build an ML model using linear regression. Your model will predict the median value of Boston houses and, based on this, we will come to a conclusion about whether the value is optimal or not.

This exercise will be performed on a Jupyter Notebook with the following steps:

> **Note**
> To proceed with the exercises in the chapter, you will need the `scikit-learn` library installed that is mentioned in the *Preface* section. It should be available with any Anaconda distribution.

1. Open a new notebook file in the same folder as your `Data` folder.
2. Now, import all the necessary libraries, as shown in the following code snippet:

```
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
```

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
```

Now that we have imported the libraries, we will load the data.

3. Load the dataset and view the DataFrames to look at the first five rows:

```
# load data
housing_df = pd.read_csv('HousingData.csv')
housing_df.head()
```

Recall that, as mentioned in *Chapter 10*, *Data Analytics with pandas and NumPy*, `housing_df = pd.read_cs('HousingData.csv')` will read the CSV file in parentheses and store it in a DataFrame called `housing_df`. Then, `housing_df.head()` will display the first five rows of the `housing_df` DataFrame by default.

You should get the following output:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	NaN	36.2

Figure 11.4 – First five rows of the Boston Housing dataset

4. Next, enter the following code to clean the dataset of `null` values using `.dropna()`:

```
# drop null values
housing_df = housing_df.dropna()
```

In *Chapter 10*, *Data Analytics with pandas and NumPy*, we cleared the `null` values by counting them and comparing them to measures of central tendency. In this chapter, however, we will use a swifter approach in order to expedite testing for ML. The `housing_df.dropna()` code will drop all `null` values from the `housing_df` DataFrame.

Now that the data is clean, it's time to prepare our `X` and `y` values.

5. Now, declare `X` and `y` variables, where you use `X` for the **predictor** columns and `y` for the **target** column:

```
# declare X and y
X = housing_df.iloc[:,:-1]
y = housing_df.iloc[:, -1]
```

The target column is `MEDV`, which is the median value of Boston house prices. The predictor columns include every other column. The standard notation is to use `X` for the predictor columns and `y` for the target column.

Since the last column is the target column, which is `y`, it should be eliminated from the predictor column—that is, `X`. We can achieve this split by indexing as already shown.

Before building the regression model, we are going to use `train_test_split()` to split `X` and `y`, the predictor and target columns, into training and test sets. The model will be built using the training set. Let's split the data in the following step.

6. Split `X` and `y` into training and test sets, as follows:

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=0)
```

`test_size=0.2` reflects the percentage of rows held back for the test set. This is the default setting and does not need to be added explicitly. It is presented so that you know how to change it. The `random_state=0` parameter is also not required, but it will ensure that you have the same split as we do, later resulting in the same ML score.

7. We now build the actual linear regression model. Although many ML models are incredibly sophisticated, they can be built using very few lines of code.

   Create an empty `LinearRegression()` model, as shown in the following code snippet:

```
reg = LinearRegression()
```

   Finally, fit the model to the data using the `.fit()` method:

```
reg.fit(X_train, y_train)
```

   The parameters are `X_train` and `y_train`, which is the training set that we have defined. `reg.fit(X_train, y_train)` is where ML actually happens. In this line, the `LinearRegression()` model adjusts itself to the training data. The model keeps changing weights, according to the ML algorithm, until the weights minimize the error.

   The Jupyter Notebook shows the following output:

```
LinearRegression()
```

   At this point, `reg` is an ML model with specified weights. There is one weight for each `X` column. These weights are multiplied by the entry in each row to get as close as possible to the target column, `y`, which is the median house value.

8. Now, find how accurate the model is. Here, we can test it on unseen data:

```
# Predict on the test data: y_pred
y_pred = reg.predict(X_test)
```

To make a prediction, we implement a method, `.predict()`. This method takes specified rows of data as the input and produces the corresponding predicted values as the output. The input is `X_test`, the `X` values that were held back for our test set. The output is the predicted `y` values.

9. We can now test the prediction by comparing the predicted `y` values (`y_pred`) to the actual `y` values (`y_test`), as shown in the following code snippet:

```
# Compute and print RMSE
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
 print(f'RMSE: {rmse}')
```

The error—the difference between the two `np.array` instances—may be computed as `mean_squared_error`. We take the square root of the mean squared error to keep the same units as the target column.

The output is as follows:

```
RMSE: 5.371207757773587
```

Note that there are other errors to choose from. The square root of `mean_squared_error` is a standard choice with linear regression. `rmse`, short for root mean squared error, will give us the error of the model on the test set.

A root mean squared error of `5.37` means that, on average, the ML model predicts values approximately `5.37` units away from the target value, which is not bad in terms of accuracy given the range of column values of 45 and standard deviation of 9.1 (see `df['MEDV'].describe()`). Since the median value (from 1980) is in the thousands, the predictions are about 5.37 thousand off. Lower errors are always better, so we will see if can improve the error going forward.

In this very first exercise, we were able to load our dataset, clean it, and build a linear regression model to make predictions and check its accuracy.

Linear regression function

After building your first ML model, you may wonder what happens if you run it multiple times as a function. Will you get different results?

Let's do this, as shown in the following example, using the same Boston Housing dataset, this time without setting a random seed.

Let's put all the ML code, including the train-test split, in a function and run it again:

```
def regression_model(model):
  # Create training and test sets
  X_train, X_test, y_train, y_test = train_test_split(X, y,
```

```
test_size=0.2)
  # Create the regressor: reg_all
  reg_all = model
  # Fit the regressor to the training data
  reg_all.fit(X_train, y_train)
  # Predict on the test data: y_pred
  y_pred = reg_all.predict(X_test)
  # Compute and print RMSE
  rmse = np.sqrt(mean_squared_error(y_test, y_pred))
  print("RMSE: {}".format(rmse))
```

Now, run the function multiple times to see the results:

```
regression_model(LinearRegression())
```

Here are several sample outputs that we obtained:

```
RMSE: 4.085279539934423
RMSE:: 4.317496624587608
RMSE:: 4.7884343211684435
```

This is troublesome, right? The score is always different. Your scores are also likely to differ from ours.

The scores are different because we are splitting the data into a different training set and test set each time, and the model is based on different training sets. Furthermore, it's being scored against a different test set.

In order for ML scores to be meaningful, we want to minimize fluctuation, and ensure that our results are representative of reality. We will see how to do this in the next section.

Testing data with cross-validation

In cross-validation, also known as CV, the training data is split into five folds (any number will do, but **five is standard**). The ML algorithm is fit on one fold at a time and tested on the remaining data. The result is five different training and test sets that are all representative of the same data. The mean of the scores is usually taken as the accuracy of the model.

> **Note**
>
> For cross-validation, 5 folds is only one suggestion. Any natural number may be used, with 3 and 10 also being fairly common.

Cross-validation is a core tool for ML. Mean test scores on different folds are more reliable than one mean test score on the entire set, which we performed in the first exercise. When examining one test score, there is no way of knowing whether it is low or high. Five test scores give a better picture of the true accuracy of the model.

Cross-validation can be implemented in a variety of ways. A standard approach is to use `cross_val_score`, which returns an array of scores for each fold; `cross_val_score` breaks `X` and `y` into training and test sets for you.

Let's modify our regression ML function to include `cross_val_score` in the following exercise.

Exercise 149 – using the cross_val_score function to get accurate results on the dataset

The goal of this exercise is to use cross-validation to obtain more accurate ML results from the dataset compared to the previous exercise. The steps are as follows:

1. Continue using the same Jupyter Notebook from *Exercise 148 – using linear regression to predict the accuracy of the median values of our dataset*.
2. Now, import `cross_val_score`:

```
from sklearn.model_selection import cross_val_score
```

3. Define the `regression_model_cv` function, which takes a fitted model as one parameter. The `k=5` hyperparameter gives the number of folds. Note that `cross_val_score` does not need a random seed because it splits the data the same way every time. Enter the code shown in the following snippet:

```
def regression_model_cv(model, k=5):
    scores = cross_val_score(model, X, y, scoring='neg_
mean_squared_ error', cv=k)
    rmse = np.sqrt(-scores)
    print('Reg rmse:', rmse)
    print('Reg mean:', rmse.mean())
```

In `sklearn`, the scoring options are sometimes limited. Since `mean_squared_error` is not an option for `cross_val_score`, we choose `neg_mean_squared_error`. `cross_val_score` takes the highest value by default, and the highest negative mean squared error is 0.

4. Use the `regression_model_cv` function on the `LinearRegression()` model defined in the previous exercise:

```
regression_model_cv(LinearRegression())
```

The output is as follows:

```
Reg rmse: [3.26123843 4.42712448 5.66151114 8.09493087
5.24453989]
Reg mean: 5.337868962878373
```

5. Use the `regression_model_cv` function on the `LinearRegression()` model with `3` folds and then `6` folds, as shown in the following code snippet, for `3` folds:

```
regression_model_cv(LinearRegression(), k=3)
```

You may get something similar to the following output:

```
Reg rmse: [ 3.72504914 6.01655701 23.20863933]
Reg mean: 10.983415161090695
```

6. Now, test the values for `6` folds:

```
regression_model_cv(LinearRegression(), k=6)
```

You may get something similar to the following output:

```
Reg rmse: [3.23879491 3.97041949 5.58329663 3.92861033
9.88399671 3.91442679]
Reg mean: 5.08659081080109
```

There is a significant discrepancy between the RMSE in the different folds. One reason is that we have a reasonably small dataset to begin with. Another reason is that some outliers may be causing problems in some of the folds. Going forward, we will keep five folds as our standard.

Regularization – Ridge and Lasso

Regularization is an important concept in ML; it's used to counteract overfitting. In the world of big data, it's easy to overfit data to the training set. When this happens, the model will often perform badly on the test set, as indicated by `mean_squared_error` or some other error.

You may wonder why a test set is kept aside at all. Wouldn't the most accurate ML model come from fitting the algorithm on all the data?

The answer, generally accepted by the ML community after research and experimentation, is no.

There are two main problems with fitting an ML model on all the data:

- There is no way to test the model on unseen data. ML models are powerful when they make good predictions on new data. Models are trained on known results, but they perform in the real world on data that has never been seen before. It's not vital to see how well a model fits known results (the training set), but it's absolutely crucial to see how well it performs on unseen data (the test set).

- The model may overfit the data. Models exist that may fit any set of data points perfectly. Consider the nine points in the following diagram. An eighth-degree polynomial exists that fits these points perfectly, but it's a poor predictor of the new data because it fails to pick up on general patterns:

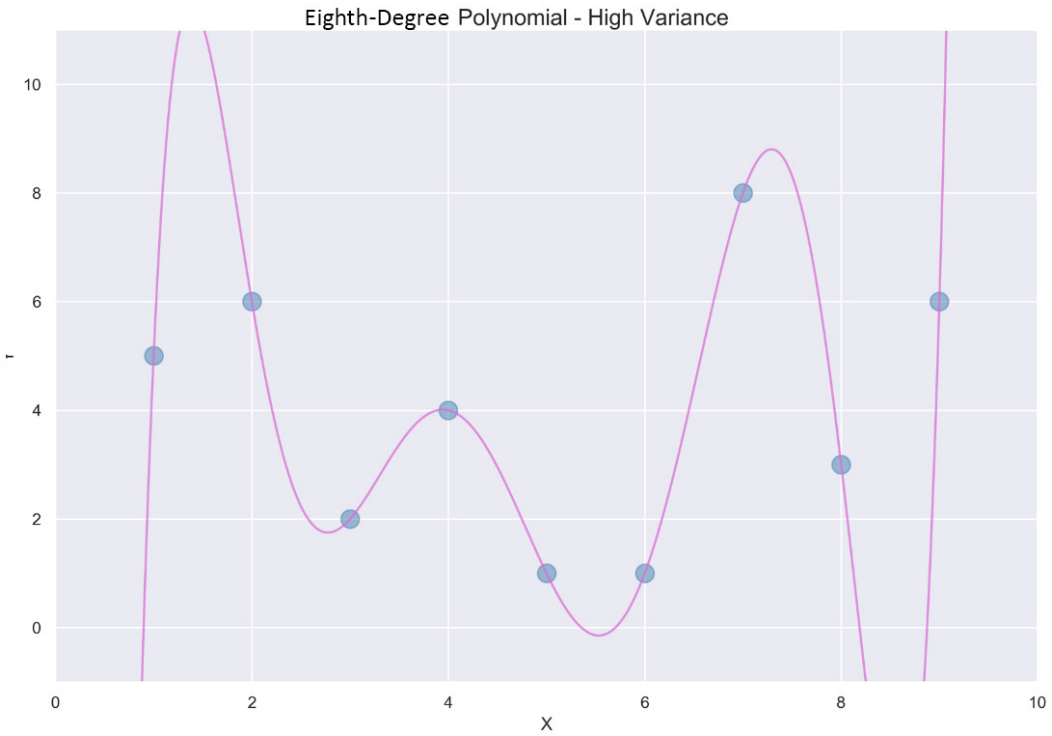


Figure 11.5 – Eighth-degree polynomial overfitting nine data points

There are many models and approaches to counteract overfitting. Let's go over a couple of linear models now:

- **Ridge** is a simple alternative to linear regression, designed to counteract overfitting. Ridge includes an L2 penalty term (L2 is based on Euclidean distance) that shrinks the linear coefficients based on their size. The coefficients are the weights—numbers that determine how influential each column is on the output. Larger weights carry greater penalties in Ridge.
- **Lasso** is another regularized alternative to linear regression. Lasso adds a penalty equal to the absolute value of the magnitude of coefficients. This L1 regularization (L1 is taxicab distance) can eliminate some column influence, but it's less widely used than Ridge on account of the L1 distance metric being less common than L2.

Let's look at an example to check how Ridge and Lasso perform on our Boston Housing dataset.

In this example, we perform regularization on the dataset using Ridge and Lasso to counteract overfitting. You can continue on the notebook from *Exercise 149 – using the cross_val_score function to get accurate results on the dataset*, to work on this example.

We begin by setting `Ridge()` as a parameter for `regression_model_cv`, as shown in the following code snippet:

```
from sklearn.linear_model import Ridge
regression_model_cv(Ridge())
```

The output is as follows:

```
Reg rmse: [3.52479283 4.72296032 5.54622438 8.00759231
5.26861171]
Reg mean: 5.414036309884279
```

Ridge has a comparable score to linear regression. This is not surprising because both algorithms use Euclidean distance, and the linear regression model is not overfitting the data by a significant amount.

Another basis of comparison is the worst score of the five. In Ridge, we obtained `8.00759` as the worst score. In linear regression, we obtained `23.20863933` as the worst score. This suggests that `23.20863933` is badly overfitting the training data. In Ridge, this overfitting is compensated.

Now, set `Lasso()` as the parameter for `regression_model_cv`:

```
from sklearn.linear_model import Lasso
regression_model_cv(Lasso())
```

You should get the following output:

```
Reg rmse: [4.712548 5.83933857 8.02996117 7.89925202
4.38674414]
Reg mean: 6.173568778640692
```

Whenever you're trying linear regression, it's often worth trying Lasso and Ridge as well since overfitting the data is common, and they only actually take a few lines of code to test. Lasso does not perform as well here because the L1 distance metric, taxicab distance, was not used in our model.

Regularization is an essential tool when implementing ML algorithms. Whenever you choose a particular model, be sure to research regularization methods to improve your results, as in the preceding example.

Now, let's get to know a developer's doubt. Although we have focused on overfitting the data, underfitting the data is also possible, right? Underfitting can occur if the model is a straight line, but a higher degree polynomial will fit the data better. By trying multiple models, you are more likely to find the optimal results.

So far, you have learned how to implement linear regression as an ML model. You have learned how to perform cross-validation to get more accurate results, and you have learned about using two additional models, Ridge and Lasso, to counteract overfitting.

Now that you understand how to build ML models using `scikit-learn`, let's take a look at some different kinds of models that will also work on regression but that will not underfit the data. In fact, some of these models are so good at picking up on nuances that they can overfit the data badly if you're not careful.

K-nearest neighbors, decision trees, and random forests

Are there other ML algorithms, besides `LinearRegression()`, that are suitable for the Boston Housing dataset? Absolutely. There are many regressors in the `scikit-learn` library that may be used. Regressors are a class of ML algorithms that are suitable for continuous target values. In addition to linear regression, Ridge, and Lasso, we can try k-nearest neighbors, decision trees, and random forests. These models perform well on a wide range of datasets. Let's try them out and analyze them individually.

K-nearest neighbors

The idea behind **k-nearest neighbors** (**KNN**) is straightforward. When choosing the output of a row with an unknown label, the prediction is the same as the output of its k-nearest neighbors, where *k* may be any whole number.

For instance, let's say that *k=3*. Given an unknown label, we take *n* columns for this row and place them in *n*-dimensional space. Then, we look for the three closest points. These points already have labels. We take the average of the three points for our new point; the value of the new point is determined by its three nearest neighbors.

KNN is commonly used for classification since classification is based on grouping values, but it can be applied to regression as well. When determining the value of a home—for instance, in our Boston Housing dataset—it makes sense to compare the values of homes in a similar location, with a similar number of bedrooms, a similar amount of square footage, and so on.

You can always choose the number of neighbors for the algorithm and adjust it accordingly. The number of neighbors denoted here is *k*, which is also called a **hyperparameter**. In ML, the model parameters are derived during training, whereas the hyperparameters are chosen in advance.

Fine-tuning hyperparameters is an essential task to master when building ML models. Learning the ins and outs of hyperparameter tuning takes time, practice, and experimentation. You will gain essential practice later in this chapter.

Exercise 150 – using k-nearest neighbors to find the median value of the dataset

The goal of this exercise is to use k-nearest neighbors to predict the optimal median value of homes in Boston. We will use the same function, `regression_model_cv`, with an input of `KNeighborsRegressor()`. Proceed as follows:

1. Continue with the same Jupyter Notebook from the previous exercise, *Exercise 149 – using the cross_val_score function to get accurate results on the dataset*.
2. Set and import `KNeighborsRegressor()` as the parameter on the `regression_model_cv` function:

```
from sklearn.neighbors import KNeighborsRegressor
regression_model_cv(KNeighborsRegressor())
```

The output is as follows:

```
Reg rmse: [ 8.24568226 8.81322798 10.58043836 8.85643441
5.98100069]
Reg mean: 8.495356738515685
```

K-nearest neighbors did not perform as well as `LinearRegression()`, but it performed respectably. Recall that `rmse` stands for root mean squared error. So, the mean error is about `8.50` (or 85,000 since the units are tens of thousands of dollars).

We can change the number of neighbors to see whether we can get better results. The default number of neighbors is `5`. Let's change the number of neighbors to `4`, `7`, and `10`.

3. Now, change the `n_neighbors` hyperparameter to `4`, `7`, and `10`. For `4` neighbors, enter the following code:

```
regression_model_cv(KNeighborsRegressor(n_neighbors=4))
```

The output is as follows:

```
Reg rmse: [ 8.44659788 8.99814547 10.97170231 8.86647969
5.72114135]
Reg mean: 8.600813339223432
```

Change `n_neighbors` to `7`:

```
regression_model_cv(KNeighborsRegressor(n_neighbors=7))
```

The output is as follows:

```
Reg rmse: [ 7.99710601 8.68309183 10.66332898 8.90261573
5.51032355]
Reg mean: 8.351293217401393
```

Change `n_neighbors` to `10`:

```
regression_model_cv(KNeighborsRegressor(n_neighbors=10))
```

The output is as follows:

```
Reg rmse: [ 7.47549287 8.62914556 10.69543822 8.91330686
6.52982222]
Reg mean: 8.448641147609868
```

The best results so far come from `7` neighbors. But how do we know whether `7` neighbors give us the best results? How many different scenarios do we have to check?

`scikit-learn` provides a nice option to check a wide range of hyperparameters: `GridSearchCV`. The idea behind `GridSearchCV` is to use cross-validation to check all the possible values in a grid. The value in the grid that gives the best result is then accepted as a hyperparameter.

Exercise 151 – K-nearest neighbors with GridSearchCV to find the optimal number of neighbors

The goal of this exercise is to use `GridSearchCV` to find the optimal number of neighbors for k-nearest neighbors to predict the median housing value in Boston. In the previous exercise, if you recall, we used only three neighbor values. Here, we will increase the number using `GridSearchCV`. Proceed as follows:

1. Continue with the Jupyter Notebook from the previous exercise.
2. Import `GridSearchCV`, as shown in the following code snippet:

   ```
   from sklearn.model_selection import GridSearchCV
   ```

3. Now, choose a grid. A grid is a range of numbers—in this case, neighbors—that will be checked. Set up a hyperparameter grid for between `1` and `20` neighbors:

   ```
   neighbors = np.linspace(1, 20, 20)
   ```

   We achieve this with `np.linspace(1, 20, 20)`, where `1` is the first number, the first `20` is the last number, and the second `20` in the brackets is the number of intervals to count.

4. Convert floats to `int` (required by `knn`):

   ```
   k = neighbors.astype(int)
   ```

5. Now, place the grid in a dictionary, as shown in the following code snippet:

   ```
   param_grid = {'n_neighbors': k}
   ```

6. Build the model for each neighbor:

```
knn = KNeighborsRegressor()
```

7. Instantiate the `GridSearchCV` object, `knn_tuned`:

```
knn_tuned = GridSearchCV(knn, param_grid, cv=5,
scoring='neg_mean_squared_error')
```

8. Fit `knn_tuned` to the data using `.fit`:

```
knn_tuned.fit(X, y)
```

9. Finally, you print the best parameter results, as shown in the following code snippet:

```
k = knn_tuned.best_params_
print("Best n_neighbors: {}".format(k))
score = knn_tuned.best_score_
rsm = np.sqrt(-score)
print("Best score: {}".format(rsm))
```

The output is as follows:

```
Best n_neighbors: {'n_neighbors': 7}
Best score: 8.516767055977628
```

Figure 11.6 – Output showing the best score using n_neighbors after GridSearchCV

It appears that `7` neighbors gave the best results after all.

Now, moving on, let's see whether we can improve our results by using tree-based algorithms.

Decision trees and random forests

You may be familiar with the game *Twenty Questions*. It's a game in which someone is asked to think of something or someone that the other person will try to guess. The questioner asks binary *yes* or *no* questions, gradually narrowing down the search in order to determine exactly who or what the other person was thinking of.

Twenty Questions is a decision tree. Every time a question is asked, there are two possible branches that the tree may take depending upon the answer. For every new question, new branching occurs, until the branches end at a prediction, called a leaf.

Here is a mini-decision tree that predicts whether someone makes over 50K:

DECISION TREE – IMAGE

Census Dataset - max_depth=2

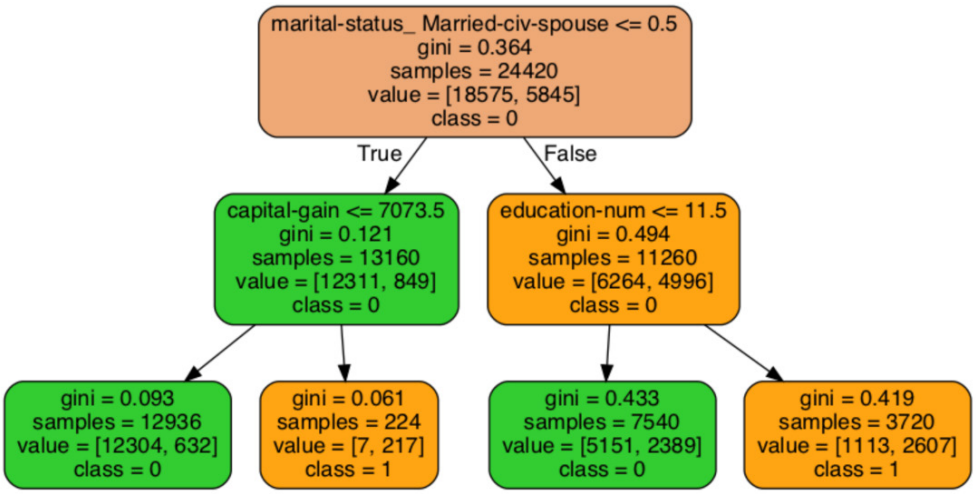


Figure 11.7 – Decision tree sample on the Titanic incident

This decision tree starts by determining whether the person is married. If the value is 0, presumably not married, the condition at the top of the decision tree is met, and you follow the `True` branch on the left in *Figure 11.7*. The next question is about the person's capital gain. If the person does not make less than 7073.5, the `False` branch on the right is followed and you end up at a leaf, where the given class is 1 with a value of 7 people who make less than 50K and 217 who make more than 50K.

Decision trees are very good ML algorithms, but they are prone to overfitting. A random forest is an ensemble of decision trees. Random forests consistently outperform decision trees because their predictions generalize to data much better. A random forest may consist of hundreds of decision trees.

A random forest is a great ML algorithm to try on almost any dataset. Random forests work well with both regression and classification, and they often perform well out of the box.

Let's try decision trees and random forests on our data.

Exercise 152 – building decision trees and random forests

The goal of this exercise is to use decision trees and random forests to predict median house values in Boston. Let's look at the steps for this exercise:

1. Continue with the same Jupyter Notebook from the previous exercise.
2. Use `DecisionTreeRegressor()` as the input for `regression_model_cv`. Include a random state set equal to `0` since decision trees have some randomness built in:

```
from sklearn import tree
regression_model_cv(tree.DecisionTreeRegressor(random_
state=0))
```

   The output is as follows:

```
Reg rmse: [3.7647936  7.26184759 7.78346186 6.48142428
4.79234165]
Reg mean: 6.016773796161434
```

3. Use `RandomForestRegressor()` as the input for `regression_model_cv`:

```
from sklearn.ensemble import RandomForestRegressor
regression_model_cv(RandomForestRegressor(random_
state=0))
```

   The output is as follows:

```
Reg rmse: [3.21859405 3.76199072 4.96431026 6.55950671
3.7700697 ]
Reg mean: 4.454894289804201
```

As you can see, the random forest regressor gives the best results yet. Let's see whether we can improve these results by examining random forest hyperparameters.

Random forest hyperparameters

Random forests have a lot of hyperparameters. Instead of going over them all, we will highlight the most important ones:

- `n_jobs(default=None)`: The number of jobs has to do with internal processing. `None` means 1. It's ideal to set `n_jobs = -1` to permit the use of all processors. Although this does not improve the accuracy of the model, it may improve the speed.
- `n_estimators(default=100)`: The number of trees in the forest. The more trees, the better. The more trees, the more RAM is required. It's worth increasing this number until the algorithm moves too slowly. Although 1,000,000 trees may give better results than 1,000, the gain might be small enough to be negligible. A good starting point is 100, and 500 if time permits.

- `max_depth(default=None)`: The maximum depth of the trees in the forest. The deeper the trees, the more information is captured about the data, but the more prone the trees are to overfitting. When set to the default `max_depth` value of `None`, there are no limitations, and each tree goes as deep as necessary. The max depth may be reduced to a smaller number of branches.
- `min_samples_split(default=2)`: This is the minimum number of samples required for a new branch or split to occur. This number can be increased to constrain the trees as they require more samples to make a decision.
- `min_samples_leaf(default=1)`: This is the same as `min_samples_split`, except it's the minimum number of samples at the leaves or the base of the tree. By increasing this number, the branch will stop splitting when it reaches this parameter.
- `max_features(default="auto")`: The number of features to consider when looking for the best split. The default for regression is to consider the total number of columns. For the classification of random forests, `sqrt` is recommended.

We could try `GridSearchCV` on a range of these hyperparameters to see whether we can find a better combination than the defaults, but checking every possible combination of hyperparameters could reach the order of thousands or millions and take way too long to build.

`sklearn` provides `RandomizedSearchCV` to check a wide range of hyperparameters. Instead of exhaustively going through a list, `RandomizedSearchCV` will check 10 random combinations and return the best results.

Exercise 153 – tuning a random forest using RandomizedSearchCV

The goal of this exercise is to tune a random forest to improve the median house value predictions for Boston. This will be done with the following steps:

1. Continue with the same Jupyter Notebook from *Exercise 152 – building decision trees and random forests*.
2. Use `RandomizedSearchCV` to look for a better combination of random forest hyperparameters than the defaults:

```
from sklearn.model_selection import RandomizedSearchCV
```

3. Set up the hyperparameter grid using `max_depth`, as shown in the following code snippet:

```
param_grid = {'max_depth': [None, 1, 2, 3, 4, 5, 6, 8,
10, 15, 20],
              'min_samples_split': [2, 3, 4, 5, 6],
```

```
                'min_samples_leaf': [1, 2, 3, 4, 6, 8],
                'max_features': [1.0, 0.9, 0.8, 0.7, 0.6,
0.5, 0.4]}
```

4. Initialize the random forest regressor:

```
reg = RandomForestRegressor(n_jobs=-1, random_state=0)
```

5. Define the `RandomizedSearchCV` object as `reg_tuned`:

```
reg_tuned = RandomizedSearchCV(reg, param_grid, cv=5,
scoring='neg_mean_squared_error', random_state=0)
```

6. Fit `reg_tuned` to the data:

```
reg_tuned.fit(X, y)
```

7. Now, print the tuned parameters and score:

```
p = reg_tuned.best_params_
print("Best params: {}".format(p))
score = reg_tuned.best_score_
rsm = np.sqrt(-score)
print("Best score: {}".format(rsm))
```

The output is as follows:

```
Best params: {'min_samples_split': 5, 'min_samples_leaf':
2, 'max_features': 0.7, 'max_depth': 10}
Best score: 4.465574177819689
```

8. Keep in mind that with `RandomizedSearchCV`, there is no guarantee that the hyperparameters will produce the best results. Although the randomized search did well, it did not perform as well as the defaults. However, let's compare the results using 500 trees with the tuned model previously, and with the rest of the hyperparameters set to their defaults. Now, run a random forest regressor with `n_jobs=-1` and `n_estimators=500`:

```
regression_model_cv(RandomForestRegressor(n_jobs=-1, n_
estimators=500))
```

The output is as follows:

```
Reg rmse: [3.17084646 3.7593559  4.8534035  6.49732743
3.94043004]
Reg mean: 4.4442726650747915
```

9. Now, run a random forest regressor using the tuned hyperparameters from the output in *step 6*:

```
regression_model_cv(RandomForestRegressor(n_jobs=-1, n_
estimators=500, random_state=0, min_samples_split=5, min_
samples_leaf=2, max_features=0.7, max_depth=10))
```

The output is as follows:

```
Reg rmse: [3.18498898 3.59234342 4.66618434 6.43013587
3.81099639]
Reg mean: 4.336929799775126
```

The is the best score yet. One reason is that many tuned parameters are designed to prevent overfitting, so it's often the case that scores will improve with more iterations because they give the algorithm more time to learn from the data.

> **Note**
> Increasing `n_estimators` generally may produce more accurate results, but the model takes longer to build.

Hyperparameters are a primary key to building excellent ML models. Anyone with basic ML training can build ML models using default hyperparameters. Using `GridSearchCV` and `RandomizedSearchCV` to fine-tune hyperparameters to create more efficient models distinguishes advanced practitioners from beginners.

Classification models

The Boston Housing dataset was great for regression because the target column took on continuous values without limit. There are many cases when the target column takes on one or two values, such as `TRUE` or `FALSE`, or possibly a grouping of three or more values, such as `RED`, `BLUE`, or `GREEN`. When the target column may be split into distinct categories, the group of ML models that you should try is referred to as **classification**.

To make things interesting, let's load a new dataset used to detect pulsar stars in outer space. Go to `https://packt.live/33SD0IM` and click on **Data Folder**. Then, click on **HTRU2.zip**, as shown:

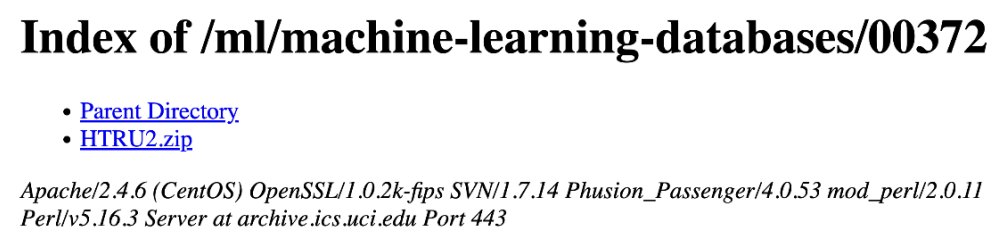


Figure 11.8 – Dataset directory on the UCI website

The dataset consists of 17,898 potential pulsar stars in space. But what are these pulsars? Pulsar stars rotate very quickly, so they have periodic light patterns. Radio frequency interference and noise, however, are attributes that make pulsars very hard to detect. This dataset contains 16,259 non-pulsars and 1,639 real pulsars.

Note

The dataset is from Dr. Robert Lyon, University of Manchester, School of Physics and Astronomy, Alan Turing Building, Manchester M13 9PL, United Kingdom, Robert.lyon'@'manchester.ac.uk, 2017.

The columns include information about an integrated pulse profile and a DM-SNR curve. All pulsars produce a unique pattern of emissions, commonly known as their "pulse profile". A pulse profile is similar to a fingerprint, but it is not consistent like a pulsar rotational period. An integrated pulse profile consists of a matrix of an array of continuous values describing the pulse intensity and phase of the pulsar. **DM** stands for **Dispersion Measure**, a constant that relates the frequency of light to the extra time required to reach the observer, and **SNR** stands for **Signal to Noise Ratio**, which relates how well an object has been measured compared to its background noise.

Here is the official list of columns in the dataset:

- Mean of the integrated profile
- Standard deviation of the integrated profile
- Excess kurtosis of the integrated profile
- Skewness of the integrated profile
- Mean of the DM-SNR curve
- Standard deviation of the DM-SNR curve
- Excess kurtosis of the DM-SNR curve
- Skewness of the DM-SNR curve
- Class

In this dataset, potential pulsars have already been classified as pulsars and non-pulsars by the astronomy community. The goal here is to see whether ML can detect patterns within the data to correctly classify new potential pulsars that emerge.

The methods that you learn for this topic will be directly applicable to a wide range of classification problems, including spam classifiers, user churn in markets, quality control, product identification, and others.

Exercise 154 – preparing the pulsar dataset and checking for null values

The goal of this exercise is to prepare the pulsar dataset for ML. The steps are as follows:

1. Open a new Jupyter Notebook in the same folder as your pulsar data file.
2. Import the libraries, load the data, and display the first five rows, as shown in the following code snippet:

```
import pandas as pd
import numpy as np
df = pd.read_csv('HTRU_2.csv')
df.head()
```

The output is as follows:

	140.5625	55.68378214	-0.234571412	-0.699648398	3.199832776	19.11042633	7.975531794	74.24222492	0
0	102.507812	58.882430	0.465318	-0.515088	1.677258	14.860146	10.576487	127.393580	0
1	103.015625	39.341649	0.323328	1.051164	3.121237	21.744669	7.735822	63.171909	0
2	136.750000	57.178449	-0.068415	-0.636238	3.642977	20.959280	6.896499	53.593661	0
3	88.726562	40.672225	0.600866	1.123492	1.178930	11.468720	14.269573	252.567306	0
4	93.570312	46.698114	0.531905	0.416721	1.636288	14.545074	10.621748	131.394004	0

Figure 11.9 – The first five rows of the pulsar dataset

Looks interesting, and problematic. Notice that the column headers appear to be in another row. It's impossible to analyze data without knowing what the columns are supposed to be, right?

Note that the last column is all `0`s in the DataFrame. This suggests that this is the `Class` column, which is our target column. When detecting the presence of something—in this case, pulsar stars—it's common to use a 1 for positive identification, and a 0 for negative identification.

Since `Class` is last in the list, let's assume that the columns are given in the correct order presented in the `Attribute Information` list. The easiest way forward is to reload the data with no header and then change the column headers to match the attribute list.

3. Now, reload the data with no header, change the column headers to match the official list, and print the first five rows, as shown in the following code snippet:

```
df = pd.read_csv('HTRU_2.csv', header=None)
df.columns = [['Mean of integrated profile', 'Standard
deviation of integrated profile','Excess kurtosis
of integrated profile', 'Skewness of integrated
profile','Mean of DM-SNR curve', 'Standard deviation
of DM-SNR curve','Excess kurtosis of DM-SNR curve',
```

```
    'Skewness of DM-SNR curve', 'Class' ]]
df.head()
```

The output is as follows:

	Mean of integrated profile	Standard deviation of integrated profile	Excess kurtosis of integrated profile	Skewness of integrated profile	Mean of DM-SNR curve	Standard deviation of DM-SNR curve	Excess kurtosis of DM-SNR curve	Skewness of DM-SNR curve	Class
0	140.562500	55.683782	-0.234571	-0.699648	3.199833	19.110426	7.975532	74.242225	0
1	102.507812	58.882430	0.465318	-0.515088	1.677258	14.860146	10.576487	127.393580	0
2	103.015625	39.341649	0.323328	1.051164	3.121237	21.744669	7.735822	63.171909	0
3	136.750000	57.178449	-0.068415	-0.636238	3.642977	20.959280	6.896499	53.593661	0
4	88.726562	40.672225	0.600866	1.123492	1.178930	11.468720	14.269573	252.567306	0

Figure 11.10 – Pulsar dataset with correct column headings

4. Now, let's find the information in the dataset using `.info()`:

```
df.info()
```

You should get the following output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17898 entries, 0 to 17897
Data columns (total 9 columns):
 #   Column                                          Non-Null Count  Dtype
---  ------                                          --------------  -----
 0   (Mean of integrated profile,)                   17898 non-null  float64
 1   (Standard deviation of integrated profile,)     17898 non-null  float64
 2   (Excess kurtosis of integrated profile,)        17898 non-null  float64
 3   (Skewness of integrated profile,)               17898 non-null  float64
 4   (Mean of DM-SNR curve,)                         17898 non-null  float64
 5   (Standard deviation of DM-SNR curve,)           17898 non-null  float64
 6   (Excess kurtosis of DM-SNR curve,)              17898 non-null  float64
 7   (Skewness of DM-SNR curve,)                     17898 non-null  float64
 8   (Class,)                                        17898 non-null  int64
dtypes: float64(8), int64(1)
memory usage: 1.2 MB
```

Figure 11.11 – Information based on the pulsar dataset

We can infer that there are no `null` values since all columns give a non-null count of 17898, which is the total number of rows. If there were `null` values, we would need to eliminate the rows or fill them in by taking the mean, the median, the mode, or another value from the columns, as explained in *Chapter 10*, *Data Analytics with pandas and NumPy*.

When it comes to preparing data for ML, it's essential to have clean, numerical data with no `null` values. Further data analysis is often warranted, depending on the goal at hand. If the goal is simply to try out some models and check them for accuracy, it's fine to go ahead. If the goal is to uncover deep insights

about the data, further statistical analysis (as introduced in the previous chapter) is always warranted. Now that we have all this basic information, we can proceed ahead with the same notebook file.

Logistic regression

When it comes to datasets that classify points, logistic regression is one of the most popular and successful ML algorithms. Logistic regression utilizes the `sigmoid` function to determine whether points should approach one value or the other. As the following diagram indicates, it's a good idea to classify the target values as 0 and 1 when utilizing logistic regression:

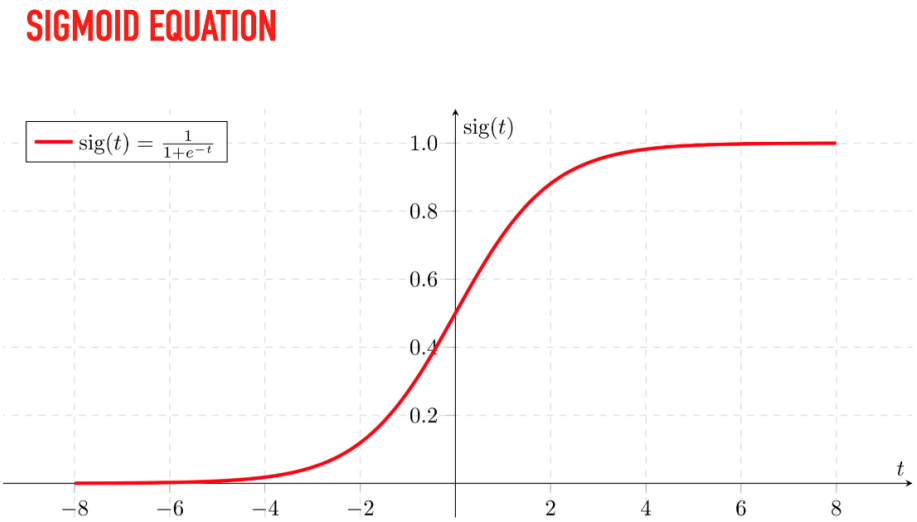


Figure 11.12 – Sigmoid curve on a plot

In the pulsar dataset, the values are already classified as 0s and 1s. If the dataset were labeled as `Red` and `Blue`, converting them in advance to 0 and 1 would be essential (you will practice converting categorical to numerical values in the activity at the end of this chapter).

The sigmoid curve in *Figure 11.12* approaches 1 from the left and 0 from the right, without ever reaching 0 or 1. In this respect, 0 and 1 function as horizontal asymptotes. Basically, every positive `x` value is given an output of 1, and every negative `x` value is given an output of 0. Furthermore, the higher up the graph, the higher the probability of a 1, and the lower down the graph, the higher the probability of 0.

Let's see how logistic regression works in action by using a similar function as before.

Note that by default, classifiers use percentage accuracy as the score output.

Exercise 155 – using logistic regression to predict data accuracy

The goal of this exercise is to use logistic regression to predict the classification of pulsar stars. The steps are set out here:

1. Import `cross_val_score` and `LogisticRegression`:

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
```

2. Set up `X` and `y` matrices to store the predictors and response variables, respectively:

```
X = df.iloc[:, 0:8]
y = df.iloc[:, 8]
```

3. Write a classifier function that takes a model as its input:

```
def clf_model(model):
```

4. Create a `clf` classifier, as shown in the following code snippet:

```
    clf = model
    scores = cross_val_score(clf, X, y)
    print('Scores:', scores)
    print('Mean score:', scores.mean())
```

5. Run the `clf_model` function with `LogisticRegression()` as the input. Be sure to set the `max_iter` parameter equal to 1000 so that the algorithm has enough time to converge, otherwise you may get an error. Basically, `max_iter` allows you to increase the maximum number of iterations for the algorithm to learn the best weights to optimize results:

```
clf_model(LogisticRegression(max_iter=1000))
```

The output is as follows:

```
Scores: [0.97486034 0.97988827 0.98184358 0.97736798
0.9782062 ]
Mean score: 0.9784332723007113
```

These numbers represent accuracy. A mean score of `0.978433` means that the logistic regression model is classifying `97.8`% of pulsars correctly.

Logistic regression is very different from linear regression. Logistic regression uses the Sigmoid function to classify all instances into one group or the other. Generally speaking, all cases that are above 0.5 are classified as a 1, and all cases that fall below 0.5 are classified as a 0, with decimals that are close to 1 more likely to be a 1, and decimals that are close to 0 more likely to be a 0. Linear regression, by

contrast, finds a straight line that minimizes the error between the straight line and the individual points. Logistic regression classifies all points into two groups; all new points will fall into one of these groups. By contrast, linear regression finds a line of best fit; all new points may fall anywhere on the line and take on any value.

Other classifiers

We used KNN, decision trees, and random forests as regressors before. This time, we need to implement them as classifiers. For instance, there is `RandomForestRegressor`, and there is `RandomForestClassifier`. Both are random forests, but they are implemented differently to meet the output of the data. The general setup is the same, but the output is different. In the next section, we will have a look at Naive Bayes.

Naive Bayes

Naive Bayes is a model based on Bayes' theorem, a famous probability theorem based on a conditional probability that assumes independent events. Similarly, Naive Bayes assumes independent attributes or columns. The mathematical details of Naive Bayes are beyond the scope of this book, but we can still apply it to our dataset.

There is a small family of ML algorithms based on Naive Bayes. The one that we will use here is `GaussianNB`. Gaussian Naive Bayes assumes that the likelihood of features is Gaussian. Other options that you may consider trying is including `MultinomialNB`, used for multinomial distributed data (such as text), and `ComplementNB`, an adaptation of `MultinomialNB` that is used for imbalanced datasets.

Let's try Naive Bayes, in addition to the KNN, decision tree, and random forest classifiers mentioned previously.

Exercise 156 – using GaussianNB, KNeighborsClassifier, DecisionTreeClassifier, and RandomForestClassifier to predict the accuracy of our dataset

The goal of this exercise is to predict pulsars using a variety of classifiers, including `GaussianNB`, `KNeighborsClassifier`, `DecisionTreeClassifier`, and `RandomForestClassifier`. Let's go through the steps:

1. Begin this exercise on the same notebook file from the previous exercise.
2. Run the `clf_model` function with `GaussianNB()` as the input:

```
from sklearn.naive_bayes import GaussianNB
clf_model(GaussianNB())
```

The output is as follows:

```
Scores: [0.96061453 0.92374302 0.94273743 0.92847164
0.96451523]
Mean score: 0.9440163679814436
```

3. Now, run the `clf_model` function with `KNeighborsClassifier()` as the input:

```
from sklearn.neighbors import KNeighborsClassifier
clf_model(KNeighborsClassifier())
```

The output is as follows:

```
Scores: [0.96955307 0.96927374 0.97318436
0.9706622  0.97289746]
Mean score: 0.9711141653437728
```

4. Run the `clf_model` function with `DecisionTreeClassifier()` as the input:

```
from sklearn.tree import DecisionTreeClassifier
clf_model(DecisionTreeClassifier(random_state=0))
```

The output is as follows:

```
Scores: [0.96843575 0.96424581 0.96871508 0.96227997
0.96954457]
Mean score: 0.9666442360073738
```

> **Note**
> The output values may differ from the values mentioned in the book.

5. Run the `clf_model` function with `RandomForestClassifier()` as the input:

```
from sklearn.ensemble import RandomForestClassifier
clf_model(RandomForestClassifier(random_state=0))
```

The output is as follows:

```
Scores: [0.97709497 0.98324022 0.98072626 0.97485331
0.97848561]
Mean score: 0.978880074800083
```

All classifiers have achieved between 94% and 98% accuracy. It's unusual for this many classifiers to all perform this well. There must be clear patterns within the data, or something is going on behind the scenes.

You may also wonder how to know when to use these classifiers. The bottom line is that whenever you have a classification problem, meaning that the data has a target column with a finite number of options (such as three kinds of wine), many classifiers are worth trying. Naive Bayes is known to work well with text data, and random forests are known to work well generally. New ML algorithms are often being developed to handle special cases. Practice and research will help to uncover more nuanced cases over time.

Confusion matrix

When discussing classification, it's important to know whether the dataset is imbalanced, as we had some doubts about the results from *Exercise 156 – using GaussianNB, KNeighborsClassifier, DecisionTreeClassifier, and RandomForestClassifier to predict the accuracy of our dataset*. An imbalance occurs if the majority of data points have one label rather than another.

Exercise 157 – finding the pulsar percentage from the dataset

The goal of this exercise is to count the percentage of pulsars in our dataset. We will use the `Class` column. Although we have primarily been using `df['Class']` as a way to reference a particular column, `df.Class` will work as well (except in limited cases, such as setting values). Follow these steps:

1. Begin this exercise in the same notebook you used in the previous exercise.
2. Use the `count.()` method on `df.Class` to obtain the number of potential pulsars:

   ```
   df.Class.count()
   ```

   The output is as follows:

   ```
   Class    17898
   dtype: int64
   ```

3. Use the `.count()` method on `df[df.Class == 1]` to obtain the number of actual pulsars:

   ```
   df[df.Class == 1].Class.count()
   ```

   The output is as follows:

   ```
   Class    1639
   dtype: int64
   ```

4. Divide *step 2* by *step 1* to obtain the percentage of pulsars:

   ```
   df[df.Class == 1].Class.count()/df.Class.count()
   ```

The output is as follows:

```
Class    0.091574
dtype: float64
```

The results show that 0.09158 or 9% of the data is pulsars. The other 91% is not pulsars. This means that it's very easy to make an ML algorithm in this case with 91% accuracy by predicting that every sample (row) is not a pulsar.

Imagine that the situation is even more extreme and we are trying to detect exoplanets, and our dataset has only classified 1% of the data as exoplanets. This means that 99% are not exoplanets. This also means that it's super easy to develop an algorithm with 99% accuracy! Just claim that everything is not an exoplanet!

A confusion matrix was designed to reveal the truth behind imbalanced datasets, as illustrated here:

		True condition	
	Total population	Condition positive	Condition negative
Predicted condition	Predicted condition positive	True positive	False positive, Type I error
	Predicted condition negative	False negative, Type II error	True negative

Figure 11.13 – Overview of a confusion matrix

As you can see from *Figure 11.13*, the confusion matrix is designed to show you what happened to each of the outputs. Every output will fall into one of four boxes, labeled **True positive**, **False positive**, **False negative**, and **True negative**, as shown here:

True positive	Prediction positive and label positive
True negative	Prediction negative and label negative
False positive	Prediction positive but label negative
False negative	Prediction negative but label positive

Figure 11.14 – Prediction of the confusion matrix based on conditions

Consider the following example. This is a confusion matrix for the decision tree classifier we used earlier. You will see the code to obtain this shortly. First, we want to focus on the interpretation:

```
[[3985   91]
[  65  334]]
```

Figure 11.15 – Confusion matrix

In `sklearn`, the default order is 0, 1. This means that the zeros or negative values are actually listed first. So, in effect, the confusion matrix is interpreted as follows:

```
          0     1
0 [[3985   91]
1 [  65  334]]
```

Figure 11.16 – Confusion matrix with the default orders

In this particular case, `3985` non-pulsars have been identified correctly, and `334` pulsars have been identified correctly. `91` in the upper-right corner indicates that the model classified `91` pulsars incorrectly, and `65` in the bottom-left corner indicates that `65` non-pulsars were misclassified as pulsars.

It can be challenging to interpret a confusion matrix, especially when positives and negatives do not always line up in the same columns. Fortunately, a classification report may be displayed along with it.

A classification report includes the total number of labels, along with various percentages to help make sense of the numbers and analyze the data.

Here is a classification report with the confusion matrix for the decision tree classifier:

```
Confusion Matrix: [[3985   91]
                   [  65  334]]

Classification Report:
                precision recall  f1-score   support

          0       0.98      0.98      0.98      4076
          1       0.79      0.84      0.81       399

avg / total       0.97      0.97      0.97      4475
```

Figure 11.17 – Classification report on the confusion matrix

In the classification report, the columns on the two ends are the easiest to interpret. On the far right, `support` is the number of labels in the dataset. It matches the indexed column on the far left, labeled 0 and 1. Support reveals that there are 4,076 non-pulsars (0s) and 399 pulsars (1s). This number is less than the total because we are only looking at the test set.

Precision is the true positives divided by all the positive predictions. In the case of 0s, this is 3985 / (3985 + 65), and in the case of 1s, this is 334 / (334 + 91).

Recall is the true positives divided by all the positive labels. For 0s, this is 3985 / (3985 + 91), and for 1s, this is 334 / (334 + 65).

The **f1-score** is the harmonic mean of the precision and recall scores. Note that the f1 scores are very different for the zeros than the ones.

The most important number in the classification report depends on what you are trying to accomplish. Consider the case of the pulsars. Is the goal to identify as many potential pulsars as possible? If so, a lower precision is okay, provided that the recall is higher. Or, perhaps an investigation would be expensive. In this case, a higher precision than recall would be desirable.

Exercise 158 – confusion matrix and classification report for the pulsar dataset

The goal of this exercise is to build a function that displays the confusion matrix along with the classification report. The following steps need to be executed for this:

1. Continue on the same notebook file from the previous exercise.
2. Now, import the `confusion_matrix` and the `classification_report` libraries:

```
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
```

   To use the confusion matrix and classification report, we need a designated test set using `train_test_split`.

3. Split the data into a training set and a test set:

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=0)
```

   Now, build a function called `confusion` that takes a model as the input and prints the confusion matrix and classification report. The `clf` classifier should be the output:

```
def confusion(model):
```

4. Create a `model` classifier:

```
    clf = model
```

5. Fit the classifier to the data:

```
    clf.fit(X_train, y_train)
```

6. Predict the labels of the `y_pred` test set:

```
    y_pred = clf.predict(X_test)
```

7. Compute and print the confusion matrix:

```
    print(confusion_matrix(y_test, y_pred))
```

8. Compute and print the classification report:

```
    print(classification_report(y_test, y_pred))
    return clf
```

Now, let's try the function on our various classifiers.

9. Run the `confusion()` function with `LogisticRegression` as the input:

```
confusion(LogisticRegression(max_iter=1000))
```

The output is as follows:

```
[[4095   20]
 [  63  297]]
              precision    recall  f1-score   support

           0       0.98      1.00      0.99      4115
           1       0.94      0.82      0.88       360

    accuracy                           0.98      4475
   macro avg       0.96      0.91      0.93      4475
weighted avg       0.98      0.98      0.98      4475
```

Figure 11.18 – Output of the confusion matrix and classification report on LogisticRegression

As you can see, the precision of classifying actual pulsars (the 1 in the classification report) is 94%. Perhaps more significantly, the f1-score, which is the average of the precision and recall scores, is 98% overall, but only 88% for the pulsars (the 1s).

10. Now, run the `confusion()` function with `KNeighborsClassifier()` as the input:

```
confusion(KNeighborsClassifier())
```

The output is as follows:

```
[[4077   38]
 [  69  291]]
              precision    recall  f1-score   support

           0       0.98      0.99      0.99      4115
           1       0.88      0.81      0.84       360

    accuracy                           0.98      4475
   macro avg       0.93      0.90      0.92      4475
weighted avg       0.98      0.98      0.98      4475
```

Figure 11.19 – Output of the confusion matrix and classification report on KNeighborsClassifier

They're all high scores overall, but the 81% recall and 84% f1-score for the pulsars are a little lacking.

11. Run the `confusion()` function with `GaussianNB()` as the input:

```
confusion(GaussianNB())
```

The output is as follows:

```
[[3946  169]
 [  52  308]]
              precision    recall  f1-score   support

           0       0.99      0.96      0.97      4115
           1       0.65      0.86      0.74       360

    accuracy                           0.95      4475
   macro avg       0.82      0.91      0.85      4475
weighted avg       0.96      0.95      0.95      4475
```

Figure 11.20 – Output of the confusion matrix and classification report on GaussianNB

In this particular case, the 65% precision of correctly identifying pulsars is not up to par.

12. Run the `confusion()` function with `RandomForestClassifer()` as the input:

```
confusion(RandomForestClassifier())
```

The output is as follows:

```
[[4095   20]
 [  59  301]]
              precision    recall  f1-score   support

           0       0.99      1.00      0.99      4115
           1       0.94      0.84      0.88       360

    accuracy                           0.98      4475
   macro avg       0.96      0.92      0.94      4475
weighted avg       0.98      0.98      0.98      4475
```

Figure 11.21 – Output of the confusion matrix and classification report on RandomForestClassifier

We've now finished this exercise, and you can see that, in this case, the f1-score of 88% for the random forest classifier is the highest that we have seen along with logistic regression.

Boosting algorithms

Random forests are a type of bagging algorithm. Bagging combines bootstrapping, selecting individual samples with replacement and aggregation, and combining all models into one ensemble. In practice, a random forest builds individual trees by randomly selecting rows of data, called samples, before combining (aggregating) all trees into one ensemble. Bagging algorithms are as good as the trees that make them up.

A comparable ML algorithm is boosting. The idea behind boosting is to transform a weak learner into a strong learner by modifying the weights for the rows that the learner got wrong. A weak learner may have an error of 49%, hardly better than a coin flip. A strong learner, by contrast, may have an error rate of 1 or 2%. With enough iterations, weak learners can be transformed into very strong learners.

Unlike bagging algorithms, boosting algorithms can improve over time. After the initial model in a booster, called the base learner, all subsequent models train on the errors of the previous model with the goal of improving the overall results.

The early success of boosting models caught the attention of the ML community. In 2003, Yoav Freund and Robert Schapire won the 2003 Gödel Prize for developing **AdaBoost**, short for **Adaptive Boosting**. Other boosters soon followed, including **XGBoost**, short for Extreme **Gradient Boosting**, which won the Kaggle competition confirming the existence of the Higgs boson. Microsoft more recently developed **LightGBM**, short for **Light Gradient Boosting Machine**, which has also won many Kaggle competitions.

LightGBM is not part of the `sklearn` library, so we will not cover it here. AdaBoost, however, is included in the `sklearn` library, and a `sklearn` wrapper for XGBoost was developed in 2019. You will build AdaBoost and XGBoost models to close out this chapter.

AdaBoost

As with many boosting algorithms, AdaBoost has both a classifier and a regressor. AdaBoost adjusts weak learners toward instances that were previously misclassified. If one learner is 45% correct, the sign can be flipped to become 55% correct. By switching the signs of negatives to positives, the problematic instances are those that are exactly 50% correct because changing the sign will not change anything. The larger the percentage that is correct, the larger the weight given to sensitive outliers.

XGBoost

XGBoost consists of multiple classifiers and regressors, including the standard gradient-boosted trees implemented here. XGBoost includes many hardware and software advances from general gradient boosting, including parallelization, regularization, and cache awareness, among others. In XGBoost, multiple advancements are combined to give it a significant edge in terms of speed and accuracy. Partially due to its success in Kaggle competitions, XGBoost has one of strongest reputations among ML ensembles in the world today.

Let's see how AdaBoost and XGBoost perform on our datasets.

Exercise 159 – using AdaBoost and XGBoost to predict pulsars

The goal of this exercise is to predict pulsars using AdaBoost and XGBoost. This will be achieved with the following steps:

1. Begin this exercise in the same notebook you used in the previous exercise.
2. Now, import `AdaBoostClassifier` and use it as the input for `clf_model()`:

```
from sklearn.ensemble import AdaBoostClassifier
clf_model(AdaBoostClassifier())
```

The output is as follows:

```
Scores: [0.97430168 0.97988827 0.98128492 0.97597094
0.97708857]
Mean score: 0.977706874833175
```

As you can see, the AdaBoost classifier gave one of the best results yet. Let's see how it performs on the confusion matrix.

3. Use `AdaBoostClassifer()` as the input for the `confusion()` function:

```
confusion(AdaBoostClassifier())
```

The output is as follows:

```
[[4094   21]
 [  63  297]]
              precision    recall  f1-score   support

           0       0.98      0.99      0.99      4115
           1       0.93      0.82      0.88       360

    accuracy                           0.98      4475
   macro avg       0.96      0.91      0.93      4475
weighted avg       0.98      0.98      0.98      4475
```

Figure 11.22 – Output of the confusion matrix and classification report on AdaBoostClassifier

Weighted averages of 98% for precision, recall, and the f1-score are outstanding. The f1-score of the positive pulsar classification (the 1s) is 93%, nearly performing as well as `RandomForestClassifier`.

XGBoost must be downloaded. You can download it to your computer inside your Jupyter Notebook, as follows:

```
import sys
!{sys.executable} -m pip install xgboost
```

4. Now, import the XGBClassifier from `xgboost` and place it inside of the `clf_model` function, as follows:

```
    from xgboost import XGBClassifier
clf_model(XGBClassifier())
```

The output is as follows:

```
Scores: [0.97765363 0.98156425 0.97932961 0.97680916
0.97876502]
Mean score: 0.9788243337532252
```

It's nearly identical to AdaBoost, but slightly higher.

5. Let's see how XGBoost works inside the confusion matrix and classification report:

```
confusion(XGBClassifier())
```

The output is as follows:

```
[[4083   32]
 [  56  304]]
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      4115
           1       0.90      0.84      0.87       360

    accuracy                           0.98      4475
   macro avg       0.95      0.92      0.93      4475
weighted avg       0.98      0.98      0.98      4475
```

Figure 11.23 – Output of the confusion matrix and classification report on XGBClassifier

The prediction of the pulsars (the 1s) could use some work, but the prediction of the non-pulsars (the 0s) is outstanding, and the combined f1-score is an impressive 98%.

Now, let's see how XGBoost and AdaBoost perform as regressors.

Exercise 160 –using AdaBoost and XGBoost to predict median house values in Boston

The goal of this exercise is to predict median house value prices in Boston using AdaBoost and XGBoost. Let's go through the steps:

1. Head to the notebook file that you used for *exercises 148-153* and run all the cells in the notebook so that all the variables are stored.

2. Now, import `AdaBoostRegressor` and use `AdaBoostRegressor()` as the input for the `regression_model_cv` function:

```
from sklearn.ensemble import AdaBoostRegressor
regression_model_cv(AdaBoostRegressor())
```

The output is as follows:

```
Reg rmse: [3.79117796 3.50477724 5.90361934 6.24188092 4.20210617]
Reg mean: 4.72871232513736
```

Figure 11.24 – Mean score output using AdaBoostRegressor

It's no surprise that AdaBoost also gives one of the best results on the housing dataset. It has a great reputation for a reason.

3. Now, import `XGBRegressor` and use `XGBRegressor()` as the input for the `regression_model_cv` function:

```
from xgboost import XGBRegressor
regression_model_cv(XGBRegressor())
```

The output is as follows:

```
Reg rmse: [3.25617197 3.70205981 5.8595083  6.47060538 3.56108012]
Reg mean: 4.569885116033572
```

Figure 11.25 – Mean score output using XGBRegressor

It's also not a surprise that XGBoost outperforms AdaBoost. XGBoost consistently performs well on a wide range of tabular datasets (meaning datasets with rows and columns). Furthermore, it has many hyperparameters to fine-tune so that you can improve your scores even more.

> **Note**
>
> For more information on XGBoost, check out the official documentation at `https://xgboost.readthedocs.io/en/latest/`.

Activity 25 – using ML to predict customer return rate accuracy

In this activity, you will use ML to solve a real-world problem. A bank wants to predict whether customers will return. When customers fail to return, this is known as churn. They want to know which customers are most likely to leave. They give you their data, and they ask you to create an ML algorithm to help them target the customers most likely to leave.

The overview for this activity will be for you to first prepare the data in the dataset, then run a variety of ML algorithms that were covered in this chapter to check their accuracy. You will then use a confusion matrix and classification report to help find the best algorithm to recall potential cases of user churn. You will select one final ML algorithm along with its confusion matrix and classification report for your output.

Here are the steps to achieve this goal:

1. Download the dataset from `https://packt.live/35NRn2C`.
2. Open `CHURN.csv` in a Jupyter Notebook and observe the first five rows.
3. Check for `NaN` values and remove any that you find in the dataset.
4. In order to use ML on all the columns, the predictive column should be in terms of numbers, not `No` and `Yes`. You may replace `No` and `Yes` with `0` and `1`, as follows:

```
df['Churn'] = df['Churn'].replace(to_replace=['No',
'Yes'], value=[0, 1])
```

5. Set `X`, the predictor columns, equal to all columns except the first and the last. Set `y`, the target column, equal to the last column.
6. You want to transform all of the predictive columns into numeric columns. This can be achieved as follows:

```
X = pd.get_dummies(X)
```

7. Write a function called `clf_model` that uses `cross_val_score` to implement a classifier. Recall that `cross_val_score` must be imported.
8. Run your function on five different ML algorithms. Choose the top three models.
9. Build a similar function using a confusion matrix and a classification report that uses `train_test_split`. Compare your top three models using this function.
10. Choose your best model, look at the hyperparameters, and optimize at least one hyperparameter.

    You should get an output similar to the following:

```
[[1186  129]
 [ 203  243]]
              precision    recall  f1-score   support

           0       0.85      0.90      0.88      1315
           1       0.65      0.54      0.59       446

    accuracy                           0.81      1761
   macro avg       0.75      0.72      0.74      1761
weighted avg       0.80      0.81      0.81      1761


AdaBoostClassifier()
```

Figure 11.26 – Expected confusion matrix output

Note
A solution for this activity can be found in *Appendix* on GitHub.

Summary

In this chapter, you have learned how to build a variety of ML models to solve regression and classification problems. You have implemented linear regression, Ridge, Lasso, logistic regression, decision trees, random forests, Naive Bayes, AdaBoost, and XGBoost. You have learned about the importance of using cross-validation to split up your training set and test set. You have learned about the dangers of overfitting and how to correct it with regularization. You have learned how to fine-tune hyperparameters using `GridSearchCV` and `RandomizedSearchCV`. You have learned how to interpret imbalanced datasets with a confusion matrix and a classification report. You have also learned how to distinguish between bagging and boosting, and precision and recall.

The value of learning these skills is that you can make meaningful and accurate predictions from big data using some of the best ML models in the world today.

In the next chapter, you will improve your ML skills by learning the foundations of **deep learning** (**DL**). In particular, you will build **sequential neural networks** (**SNNs**) using `keras`, a state-of-the-art library that runs on top of Google's TensorFlow.

12
Deep Learning with Python

Overview

By the end of this chapter, you will confidently build and tune neural networks using the Sequential deep learning algorithm provided by Keras (TensorFlow). In particular, you will apply deep learning to make meaningful predictions from tabular numerical datasets, in addition to image-based datasets. You will compare the sequential deep learning algorithm to standard machine learning algorithms using regression and classification. You will tune Keras models by modifying Dense layers, Hidden layers, Dropout nodes, and Early Stopping to optimize your neural networks. Finally, you will learn how to classify images by building convolutional neural networks, which are some of the strongest machine learning algorithms in the world today.

Introduction

Deep learning is a specific branch of machine learning modeled after the human brain, commonly referred to as neural networks.

The human brain works by transferring external stimuli through a vast network of neurons. These neurons work together to produce a desired output. For instance, when you are driving a car and your eye detects a red light, the neurons in your brain work together to rapidly output a request to stop the car. This request is based on optimizing past data that your brain has received.

According to the National Library of Medicine (`https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2776484/`), the most advanced human brains contain approximately 100 billion neurons. This is a very deep network. The general idea behind deep learning is to emulate the human brain by creating a deep algorithmic network that responds to incoming data.

Machine learning started with neural networks when Frank Rosenblatt's 1958 Perceptron demonstrated 100% efficiency in finding linear classifiers to distinguish between two classes of linearly separable data. Although time-consuming, and limited to success if the data was linearly separable, the original Perceptron revealed the potential of computers to establish meaningful neural networks (see `https://news.cornell.edu/stories/2019/09/professors-perceptron-paved-way-ai-60-years-too-soon` for more information).

In this chapter on deep learning, you will examine the infrastructure of neural networks by first comparing them to the standard machine learning networks that you built in *Chapter 11*, *Machine Learning*. After reviewing the math behind linear regression, you will better understand the mathematical advantages that deep learning provides.

You will build your first deep learning models using Keras, TensorFlow's high-level language developed by Google that is now more widespread than ever. You will build a Sequential algorithm with Hidden layers that are densely connected using the Boston Housing dataset from *Chapter 11*, *Machine Learning*.

After building your first deep learning model, you will experiment with different Hidden layers, Dropout, and Early Stopping in an attempt to optimize and regularize your model.

Next, you will build a comparable Sequential Keras model for classification, using different activation and learning loss functions for a larger dataset.

Then, you will use Keras to identify handwritten digits using the famous **Modified National Institute of Standards and Technology** (**MNIST**) dataset. For this final case study, you will learn how to use **convolutional neural networks** (**CNNs**) to classify numbers.

In summary, you will learn how to make meaningful predictions by implementing and modifying deep learning algorithms (synonymous with neural networks) in Keras for datasets that require regression, classification, and predictions based on images.

Here's a quick overview of the topics covered:

- Introduction to deep learning
- Your first deep learning model
- Regularization – Dropout
- Classification
- Convolutional neural networks

Technical requirements

You can find the code files for this chapter on GitHub at `https://github.com/PacktPublishing/The-Python-Workshop-Second-Edition/tree/main/Chapter12`, and within the following Colab notebook: `https://colab.research.google.com/drive/14FUXbsuRvz3jO6bzAm1Mgas6faJ0G61-?usp=sharing`.

The technical requirements are different for Colab notebooks and Jupyter Notebook. You will need to install Keras and TensorFlow for Jupyter Notebook, whereas they are included with Colab notebooks in advance.

Colab notebooks

In this chapter, I recommend using an online version of Jupyter Notebook, called Colab notebooks (short for **Google Colaboratory Notebooks**) for the following reasons:

- Colab notebooks allow you to use **graphical processing units** (**GPUs**), which will greatly speed up computations for high-demand processing. This is particularly beneficial for neural networks, which can be time-consuming when combining large datasets with deep networks. We will be using them in the last section on convolutional neural networks.
- Colab notebooks have become the standard online alternative to Jupyter Notebook for data science; as a practicing data scientist, it's beneficial to have practice with both.
- Colab notebooks do not require installing special libraries such as TensorFlow.
- Colab notebooks are very easy to share with others.

> **Note**
> Colab notebooks work optimally with private Gmail accounts. When logged into your Gmail account, Colab notebooks automatically save in your Google Drive in a `Colab Notebooks` folder.

For those of you using Colab notebooks, no prior installation is required.

Jupyter Notebook

If you prefer working in Jupyter Notebook, you need to install TensorFlow, which provides the backend for Keras.

If you are working with Anaconda, you may install TensorFlow with the following command in your terminal:

```
conda install -c conda-forge tensorflow
```

For more information on installing TensorFlow with Anaconda, visit `https://anaconda.org/conda-forge/tensorflow`.

If you are not working with Anaconda, you may install TensorFlow with the following commands:

```
pip install --upgrade pip
pip install tensorflow
```

You may confirm your installation of TensorFlow by running the following code, which will show the version of TensorFlow that you are running. Anything 2.8 or above should be sufficient:

```
import tensorflow as tf
tf.__version__
```

For those of you using Jupyter Notebook, after you have installed TensorFlow, you are ready to use Keras!

Introduction to deep learning

The neurons in a human brain are analogously referred to as nodes in deep learning algorithms. Individual nodes may be thought of as computational units. Although they may stand alone, they are more powerful when connected to one another.

As a visual, here is the Boston Housing DataFrame from *Chapter 11*, *Machine Learning*. Each column in the following DataFrame can be represented as a node, as can each entry:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	NaN	36.2

Figure 12.1 – Sample from the Boston Housing dataset

In linear regression, using the standard machine learning algorithm introduced in *Chapter 11*, *Machine Learning*, each column, or node, is multiplied by a constant, called a weight, and the individual weights are summed together to make a prediction, as in the following diagram:

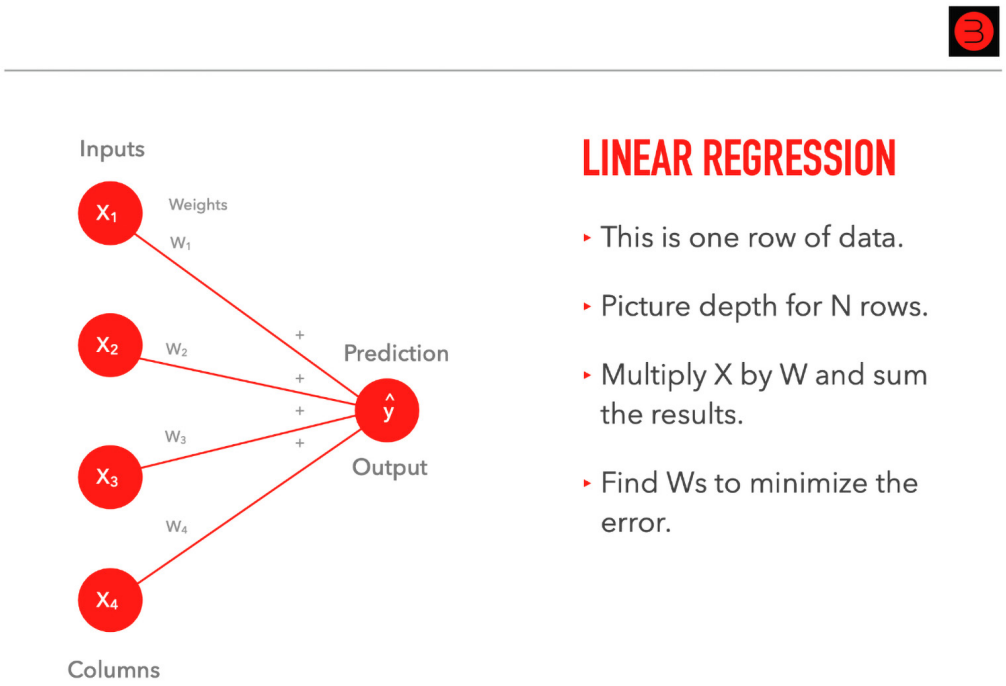


Figure 12.2 – Linear regression diagram from Berkeley Coding Academy, created by the author Corey Wade

The process is linear because it uses the simple technique of multiplication and addition; given any input X, after multiplication and addition, $Y = MX$ is a linear combination.

Complexifying the network for deep learning requires two shifts. The first is called an activation function. After multiplication and addition, the resulting nodes become activated through an activation function that allows for nonlinearity. The idea behind the activation function comes from the human brain. If you see a red light while in a car, only certain neurons become activated, such as one that tells your foot to step on the brake, and hopefully not one that tells you to reach for your cell phone. But in deep learning, the activation function runs deeper.

The output for logistic regression, as discussed in *Chapter 11*, *Machine Learning*, is the Sigmoid function, a nonlinear function used to classify data into two groups based on probability. Graphically, the Sigmoid function is clearly nonlinear, as the following figure reveals:

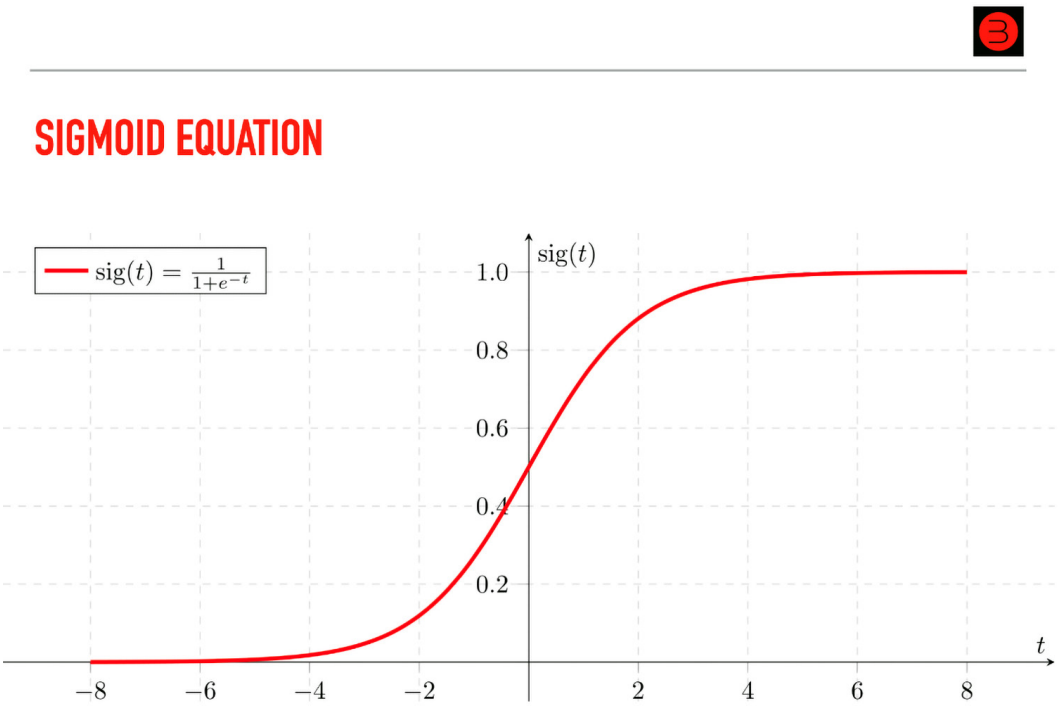


Figure 12.3 – Sigmoid curve on a plot from Berkeley Coding Academy, by the author Corey Wade

After the columns are multiplied by constants and summed together, the final result may be placed into the Sigmoid function, resulting in a nonlinear equation, as is the case with logistic regression. In deep learning, every additional node between the first and last layers becomes activated via an activation function. This general shift to nonlinearity in deep learning allows for much richer algorithms. With activation functions, all deep learning networks become non-linear. Other commonly used activation functions include the hyperbolic tangent (`tanh`), and the rectified linear unit (`relu`), which will be sampled in *Exercise 162 – using Sequential deep learning to predict the accuracy of the median house values of our dataset*.

In addition to activation functions, there is a second and perhaps more dramatic shift that differentiates deep learning from standard machine learning algorithms. Deep learning allows for as many nodes as desired. In other words, deep learning does not just aspire to the 100 billion neurons of the human brain; it exceeds it on demand.

To give you a flavor of the complexity of deep learning, consider the following diagram:

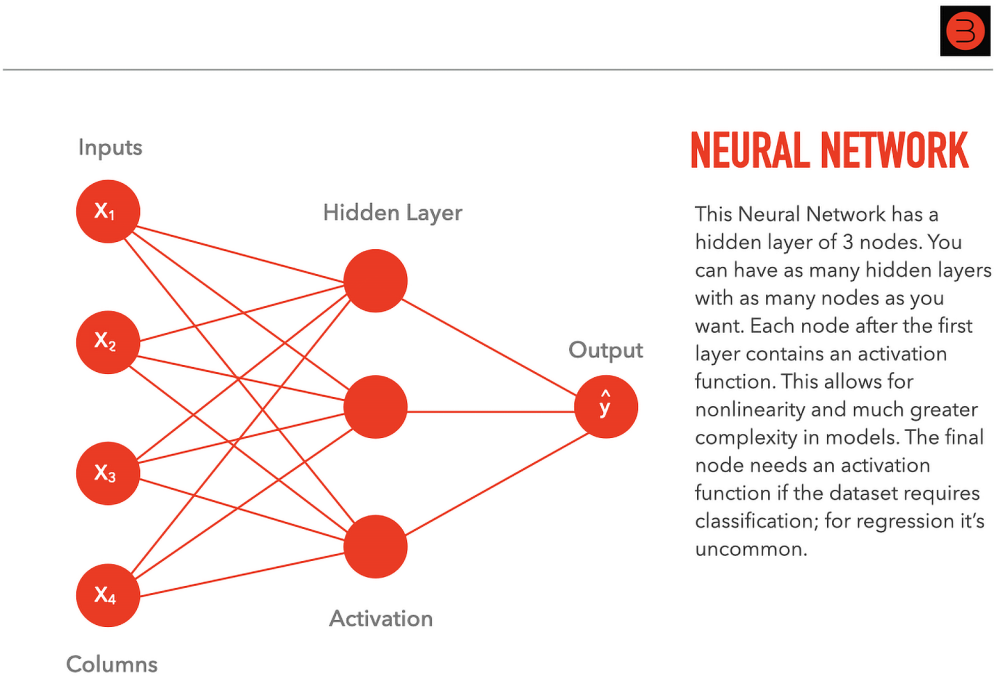


Figure 12.4 – Simple neural network from Berkeley Coding Academy, by the author Corey Wade

The first column in this figure represents columns as nodes. In this densely connected network, each line represents a mathematical weight, and each node beyond the first column includes a nonlinear activation function. There are 15 total mathematical parameters in this relatively simple diagram.

As you can see in the next diagram, it's not uncommon for deep learning algorithms to contain tens of thousands, hundreds of thousands, or even more parameters depending on the project at hand:

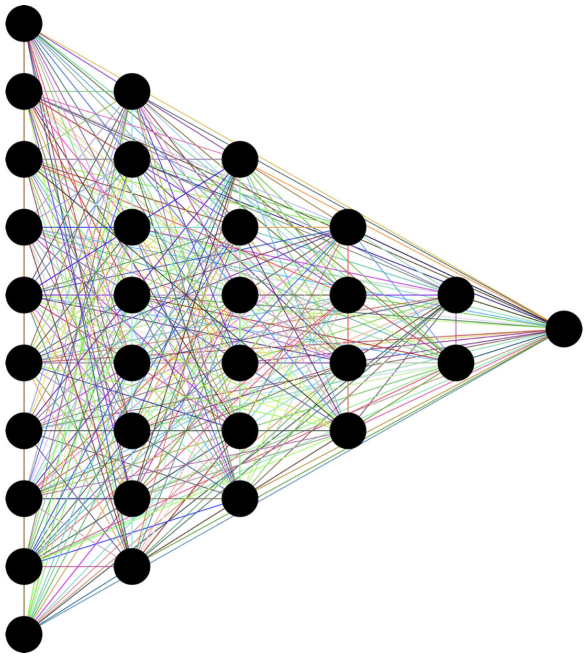


Figure 12.5 – Complex neural network, from https://openclipart.org/detail/309343/neural-network-deep-learning-prismatic

Now that you have an idea of how deep learning compares to machine learning in terms of nonlinear activation functions and a much larger network size, it's time to build your first deep learning model.

Your first deep learning model

Let's use deep learning to predict the median house values in Boston to compare our results to the standard machine learning algorithms used in *Chapter 11*, *Machine Learning*.

First deep learning libraries

Before building your first deep learning model, let's take a brief look at the libraries that we will import and use:

- `pandas`: We need data to build a deep learning model, and `pandas`, Python's data analytics library, will remain our standard from *Chapter 10*, *Data Analytics with pandas and NumPy*, and *Chapter 11*, *Machine Learning*, to read and view data.
- `train_test_split`: We will use `train_test_split` as in *Chapter 11*, *Machine Learning*, to split the data into a **training** set and a **test** set.

- `TensorFlow`: `TensorFlow` has become the gold standard in deep learning. Created by Google in 2015, `TensorFlow` is a free, open source library developed by Google Brain. `TensorFlow` works on its own, but it is also the backend for `keras`.
- `keras`: A high-level version of `TensorFlow`, `keras` is an awesome, easy-to-use interface that allows you to focus on building powerful neural networks without worrying about tensors. `keras` is widely used all over the world to build elite deep learning models.
- `Sequential`: The `Sequential` model in `keras` provides a powerful framework to build neural networks that move from left to right in sequence. All deep learning models in this chapter will use `Sequential`.
- `Dense`: The nodes in `keras` that work together in sequence are referred to as layers. It's common for these layers to be densely connected, meaning that each node in one layer connects to every other node in the subsequent layer. These densely connected layers are initiated in `keras` as `Dense` layers.
- `EarlyStopping`: A valuable `keras` callback covered later in this chapter that stops neural networks when they peak.
- `Dropout`: A regularization technique in `keras` that drops out nodes by percentage within the network to prevent overfitting, also covered later in this chapter.

> **Note**
> For more information on the `keras` libraries, check out the official `keras` documentation at `https://keras.io/`.

Now that you have an idea of the libraries that we will use to create our first deep learning model, in the next exercise, we will import the libraries, load the data, and prepare the data for deep learning.

Exercise 161 – preparing the Boston Housing dataset for deep learning

The goal of this exercise is to prepare the Boston Housing dataset to get ready for Deep Learning.

This exercise will be performed in a Colab notebook (you may use Jupyter Notebook as well).

> **Note**
> If you are working in Jupyter Notebook, to proceed with the exercises in the chapter, you will need to download the `TensorFlow` library as outlined in the previous section.

Let's see the steps for this exercise:

1. For Colab users: log in to your private Google account, then open a new Colab notebook at `https://colab.research.google.com/` (for Jupyter users, open a new Jupyter notebook).
2. Import all the necessary libraries by entering the following code snippet in a coding cell:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.callbacks import EarlyStopping
```

Press *Shift* + *Enter* in Colab to run the coding cell. After running the cell in Colab, your screen should appear as follows:

Figure 12.6 – Deep learning Colab notebook screenshot

Now that we have imported the libraries, we will load the data.

3. Load the dataset from the provided URL and view the DataFrame to look at the first five rows:

```
url ='https://raw.githubusercontent.com/PacktWorkshops/
ThePython-Workshop/master/Datasets/HousingData.csv'
df = pd.read_csv(url)
df.head()
```

After pressing *Shift* + *Enter* to run this code in Colab, you should get the following output, just as in *Chapter 11*, *Machine Learning*:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	NaN	36.2

Figure 12.7 – Output with the dataset displayed

Coding tip

You may load CSV files directly from URLs via GitHub if you click on **View Raw** after opening the CSV file.

4. Next, run the following code to clean the dataset of null values using `.dropna()`:

```
df = df.dropna()
```

Now that the data is clean, it's time to split the data.

5. Declare the `X` and `y` variables, where you use `X` for the **predictor** columns and `y` for the **target** column, then split the data into a training and test set as follows:

```
X = df.iloc[:,:-1]
y = df.iloc[:, -1]
X_train, X_test, y_train, y_test = train_test_split(X, y,
random_state=2)
```

The target column is `MEDV`, which is the median value of the Boston house prices. The predictor columns include every other column. The standard notation is to use `X` for the predictor columns and `y` for the target column. Using `random_state=2` is not required, but it guarantees that you will get the same split as ours.

At this point, the Boston Housing dataset is ready for deep learning.

In the next exercise, we will build our first deep learning model!

Exercise 162 – using sequential deep learning to predict the accuracy of the median house values of our dataset

In this exercise, we will apply the following steps to build a Sequential deep learning regressor using Keras:

1. Initialize a `Sequential()` model, as shown in the following code snippet:

```
model = Sequential()
```

This lets Python know that we are going to build a neural network that will connect nodes from left to right.

2. In `keras`, it's required to specify the number of columns as the input shape for the first Dense layer. You may set the number of columns as follows:

```
num_cols = X.shape[1]
```

3. Now it's time to specify our first densely connected layer. We can choose the number of nodes in this new layer. Each new node will take as input the multiplication and sum from each of the previous nodes, which are columns, since they are coming from the first layer. Finally, the new nodes should be activated by an activation function. We will choose `relu`, which stands for **Rectified Linear Unit**; it simply returns `0` for negative values and the same input, or `X` value, for positive values, as shown in the following diagram:

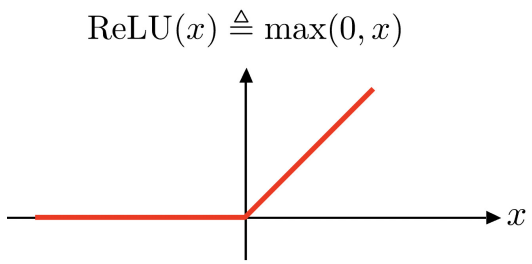


Figure 12.8 – ReLU function, from Wikimedia Commons: `https://upload.wikimedia.org/wikipedia/commons/8/85/ReLU_and_Nonnegative_Soft_Thresholding_Functions.svg`

Add a new Dense layer of 20 nodes, taking the number of columns (`num_cols`) as the input, and `relu` as the activation function, as in the following code snippet:

```
model.add(Dense(20, input_shape=(num_cols,),
activation='relu'))
```

We can continue adding new layers without the need to specify the input any longer.

4. Add a new densely connected layer of 10 nodes using the same activation function:

```
model.add(Dense(10, activation='relu'))
```

5. To arrive at an actual prediction, we must conclude with one final layer, densely connected, that only contains one node, without any activation so that all results are possible. We want all results because the desired output is the median house value, which means that we are dealing with regression. Here is the code for the last layer of our model:

```
model.add(Dense(1))
```

So far, we have not built an actual model. We have set up the parameters for the algorithm to build a model. To build the model, we need to feed it the data.

6. Add the following code to print out a summary of the model so far:

```
print(model.summary())
```

Here is the summary of the model provided by Keras:

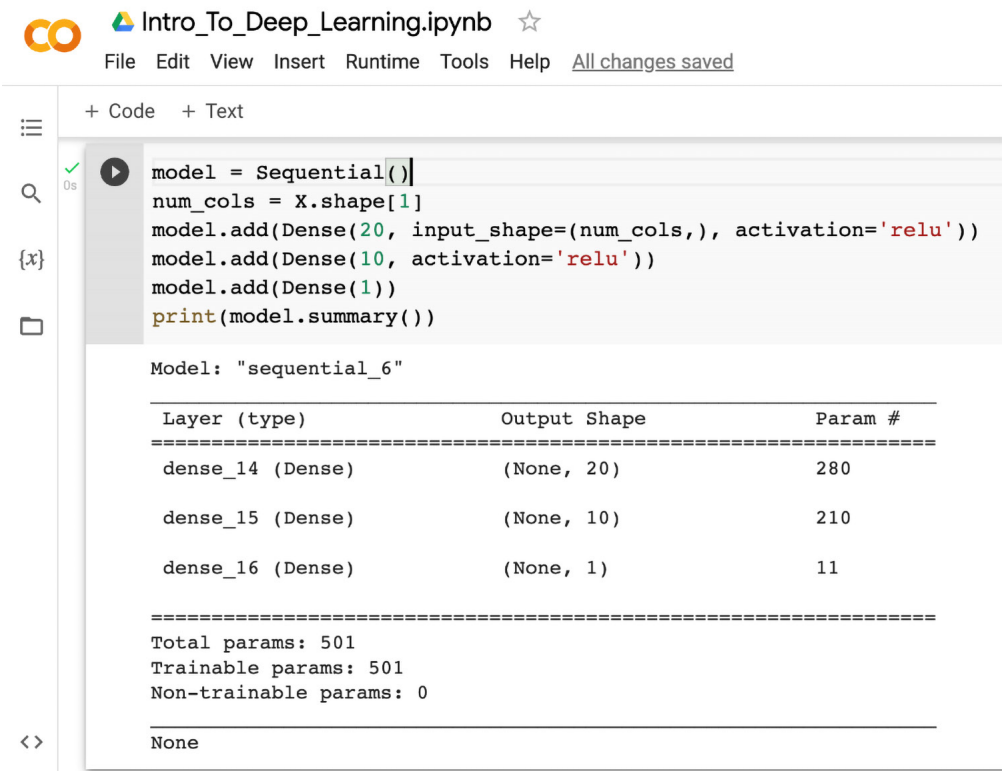


Figure 12.9 – Screenshot of initial code and model summary print-out from Colab notebook

The most interesting part of the model summary is the total number of parameters, which is `501`. This means that the model will attempt to optimize 501 weights. The parameters are computed by multiplying the number of inputs by the number of nodes per layer + 1; the addition of one accounts for the resulting parameter after summing the nodes.

Note that the greater the number of dense layers and the length of the layers, the more parameters the model requires and the longer it will take to build. The next step is to compile the model. To do so, we need to specify an optimizer to find the desired weights, along with a loss function to determine how close the predictions are to reality.

7. Understanding optimizers requires calculus, which is beyond the scope of this book. We will stick with `adam` as our optimizer. Since the root mean squared error (RMSE) is not a default option, we will select the comparable mean squared error, abbreviated as `mse`, for the loss function. Both are combined in the following code snippet:

```
model.compile(optimizer='adam', loss='mse')
```

8. Finally, it's time to train the model using the `.fit` method, with `X_train` and `y_train` as parameters, along with a set number of epochs, which is the number of times that the model updates its weights in an attempt to minimize the loss using the preselected optimizer. The more epochs you specify, the longer the model will take to build. Let's start with a reasonably small number of 10 epochs, as shown in the following code snippet:

```
model.fit(X_train, y_train, epochs=10)
```

9. After the number of epochs completes, all that remains is to obtain a score on the test set. We can get the root mean squared error by taking the square root of the test set using the following code:

```
model.evaluate(X_test, y_test)**0.5
```

After you run all the previous code by pressing *Shift* + *Enter*, you should get an output comparable to the following:

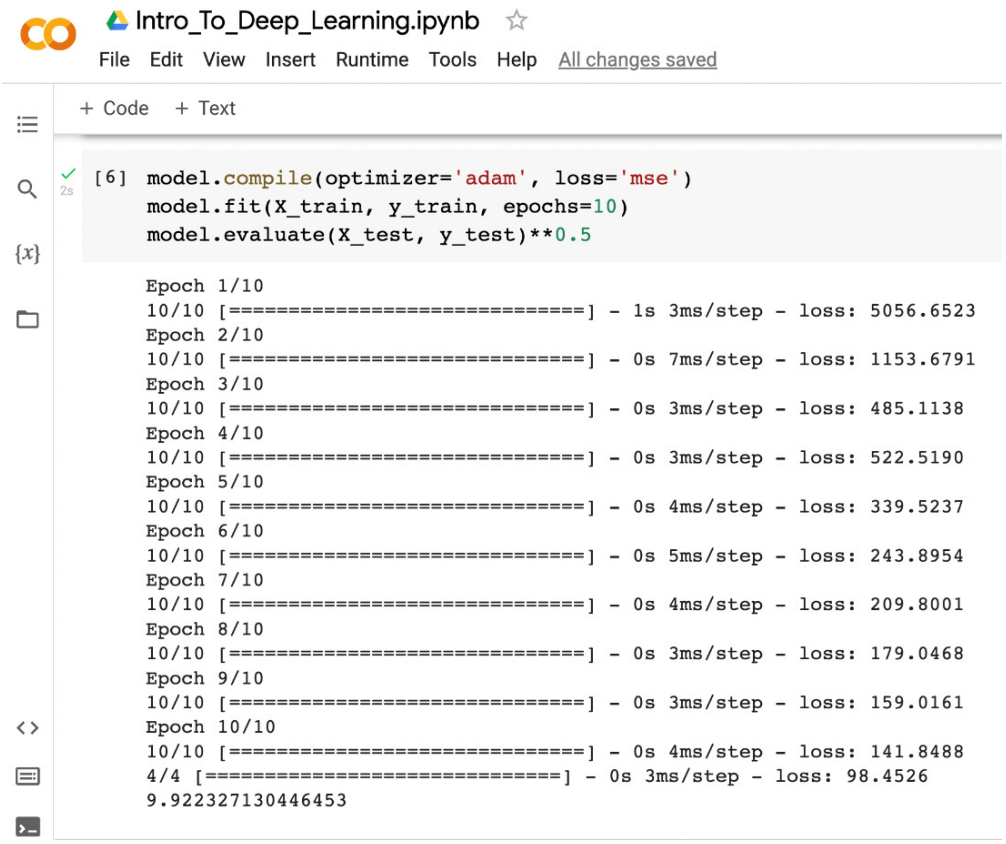


Figure 12.10 – Screenshot from Colab notebook – first deep learning score, RMSE of 9.92

If your score is much worse, don't panic. If it's better, great. Randomization is included in the weight initialization process, so the scores are going to be different.

A bigger point of emphasis is that 10 epochs are not enough time for the model to find convergence. A cool feature of Keras is that you can simply add more epochs; the following code snippet will add 50 epochs to the model:

```
model.fit(X_train, y_train, epochs=50)
model.evaluate(X_test, y_test)**0.5
```

After running this code, you should see an output similar to the following:

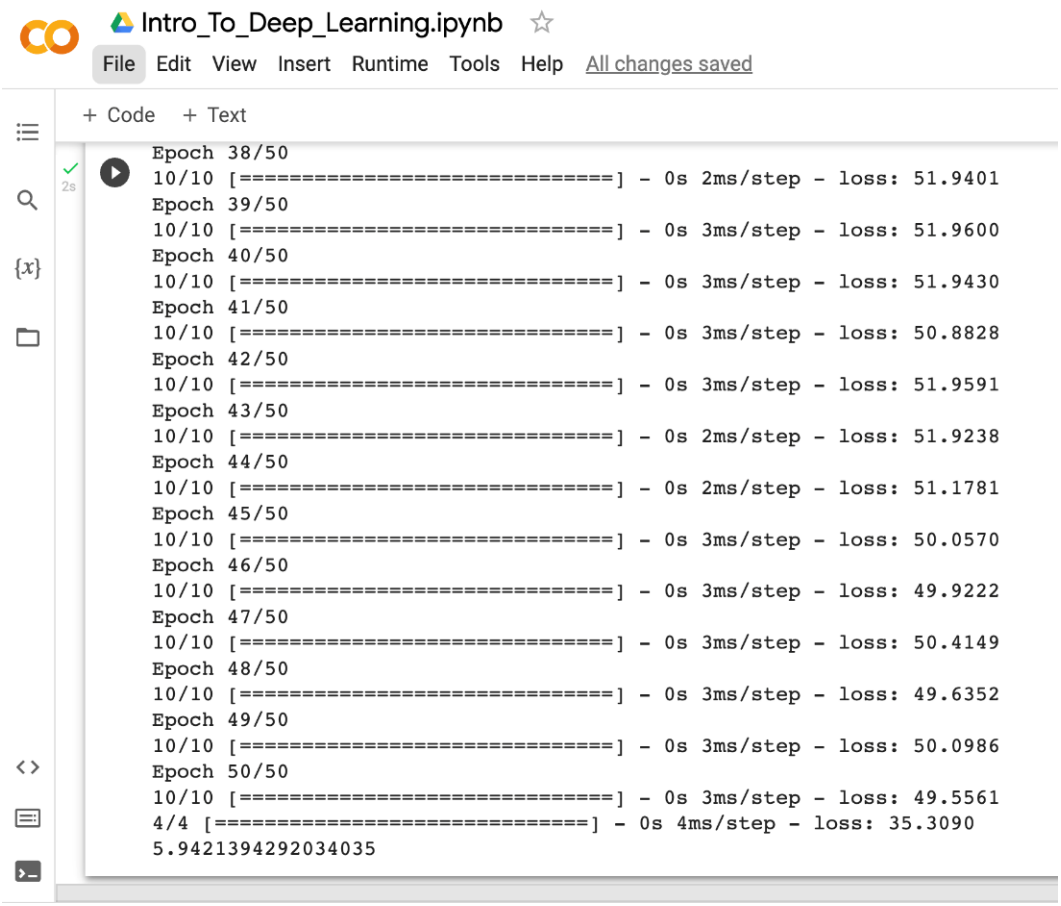


Figure 12.11 – Screenshot from Colab notebook – second deep learning score, RMSE of 5.94

A final RMSE of 5.9 is solid but does not surpass the best results obtained by linear regression in the previous chapter. Why?

First, deep learning is advantageous on very large datasets, along with unstructured data such as images or text. Second, deep learning models are very flexible in terms of the number and depth of densely connected layers, which we will explore in the next section to improve the model.

Tuning Keras models

We will look at several ways to improve scores by tuning Keras models. For additional ideas, check out the new Keras Tuner at `https://www.tensorflow.org/tutorials/keras/keras_tuner`.

Hidden layers

The densely connected layers between the input and the output are often referred to as **hidden layers** because they are not part of the input or output. The term *hidden layers* is suggestive of something going on beyond the human eye, although the programmer chooses the number and depth of these layers, and it's possible to retrieve model weights to figure out exactly what is going on.

So, what is an optimal strategy for choosing the number and depth of hidden layers?

The first answer in machine learning is always experimentation. Part of the fun of building neural networks is choosing your own hidden layers and numbers of nodes. Also, what works on one dataset may not work as well on another.

In our case, we want to improve the score of the model, so a natural first strategy is to increase the number of parameters by increasing the number of densely connected layers and nodes per layer.

To be more precise, increasing the number of parameters may be done as follows:

- Repeat the same densely connected layers multiple times
- Gradually increase or decrease the number of nodes in successive densely connected layers
- Use one large densely connected layer

Let's try these options in the next exercise.

Exercise 163 – modifying densely connected layers in a neural network to improve the score

The goal of this exercise is to lower the root mean squared error from the previous exercise by increasing the number of densely connected layers, and/or the number of nodes per layer. Let's start by increasing the number of nodes to 24, using only the first layer.

Here are the steps to change the densely connected layers and nodes:

1. Continue using the same Colab or Jupyter notebook from *Exercise 162 – using sequential deep learning to predict the accuracy of the median house values of our dataset*.

2. Create three densely connected layers of 24 nodes each with the `relu` activation function using 50 epochs in total with the following code:

```
model = Sequential()
model.add(Dense(24, input_shape=(num_cols,),
activation='relu'))
model.add(Dense(24, activation='relu'))
model.add(Dense(24, activation='relu'))
model.add(Dense(1))
print(model.summary())
model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=50)
model.evaluate(X_test, y_test)**0.5
```

After running this code, you should see the model summary before the model build as follows:

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 dense_6 (Dense)              (None, 24)                336

 dense_7 (Dense)              (None, 24)                600

 dense_8 (Dense)              (None, 24)                600

 dense_9 (Dense)              (None, 1)                 25

=================================================================
Total params: 1,561
Trainable params: 1,561
Non-trainable params: 0
```

Figure 12.12 – Screenshot from Colab notebook – model summary with 1561 parameters

As you can see from the model summary, the neural network contains a total of 1,561 parameters, which is approximately triple the previous 501.

Underneath the model summary, the results of the neural network after 50 epochs are as follows:

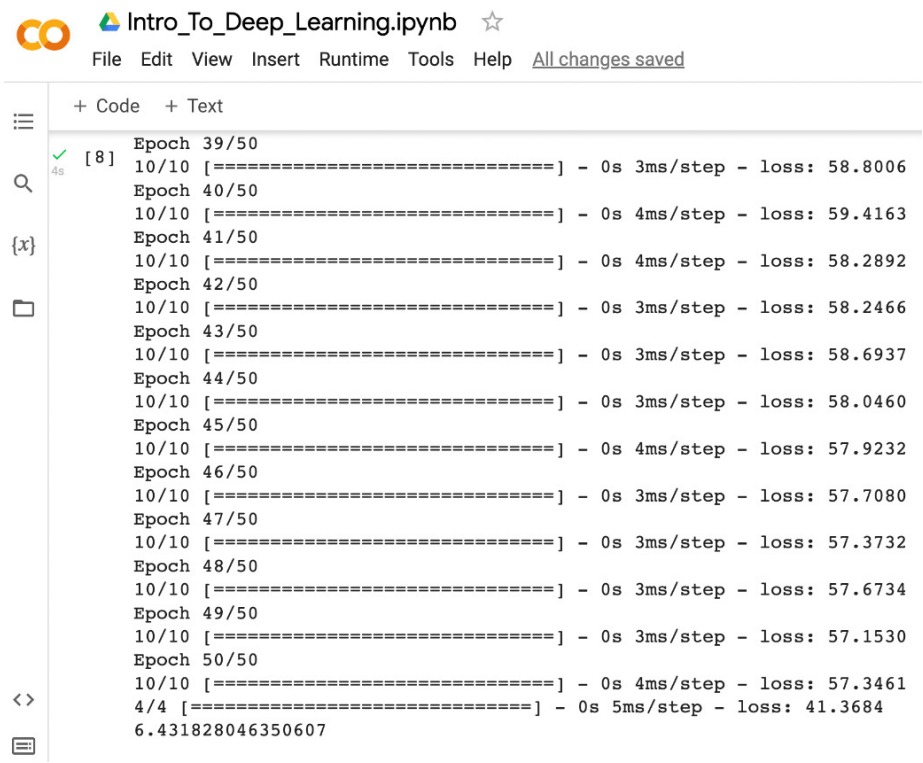


Figure 12.13 – Screenshot from Colab notebook – third deep learning score, RMSE of 6.43

Our score, however, has not improved, with an RMSE of 6.43. Notice that the loss on the right side, however, is steadily improving, so we could increase the number of epochs, a strategy that we will implement later.

3. Next, try creating two densely connected layers of 48 and 16 nodes each by running the following code snippet:

```
model = Sequential()
model.add(Dense(48, input_shape=(num_cols,),
activation='relu'))
model.add(Dense(16, activation='relu'))
model.add(Dense(1))
print(model.summary())
model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=50)
model.evaluate(X_test, y_test)**0.5
```

You should see the following model summary above the model build:

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_10 (Dense)            (None, 48)                672

 dense_11 (Dense)            (None, 16)                784

 dense_12 (Dense)            (None, 1)                 17

=================================================================
Total params: 1,473
Trainable params: 1,473
Non-trainable params: 0
```

Figure 12.14 – Screenshot from Colab notebook – model summary with 1,473 parameters

As you can see from this model summary, there are slightly fewer parameters; however, in this particular case, the following output at the end of the model build reveals that the RMSE has improved to a new low score of 5.086:

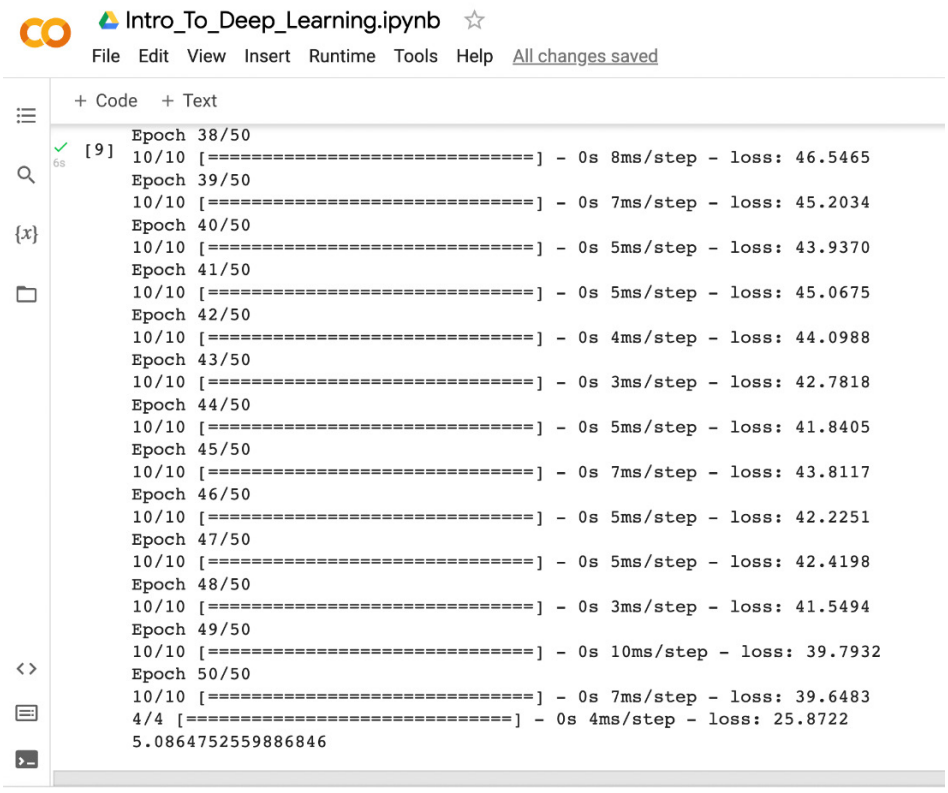


Figure 12.15 – Screenshot from Colab notebook – fourth deep learning score, RMSE of 5.09

4. Finally, create a Sequential model with one densely connected layer of 100 nodes by running the following code:

```
model = Sequential()
model.add(Dense(100, input_shape=(num_cols,),
activation='relu'))
model.add(Dense(1))
print(model.summary())
model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=50)
model.evaluate(X_test, y_test)**0.5
```

Here is the model summary, displaying a comparable number of parameters at 1,501:

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_13 (Dense)            (None, 100)               1400

 dense_14 (Dense)            (None, 1)                 101

=================================================================
Total params: 1,501
Trainable params: 1,501
Non-trainable params: 0
```

Figure 12.16 – Screenshot from Colab notebook – model summary with 1,501 parameters

Running this code at different times can give very different answers. Try it! Here are the low and high results that occurred after running the previous code a few times.

Here is the low score from a single dense layer of 100 nodes:

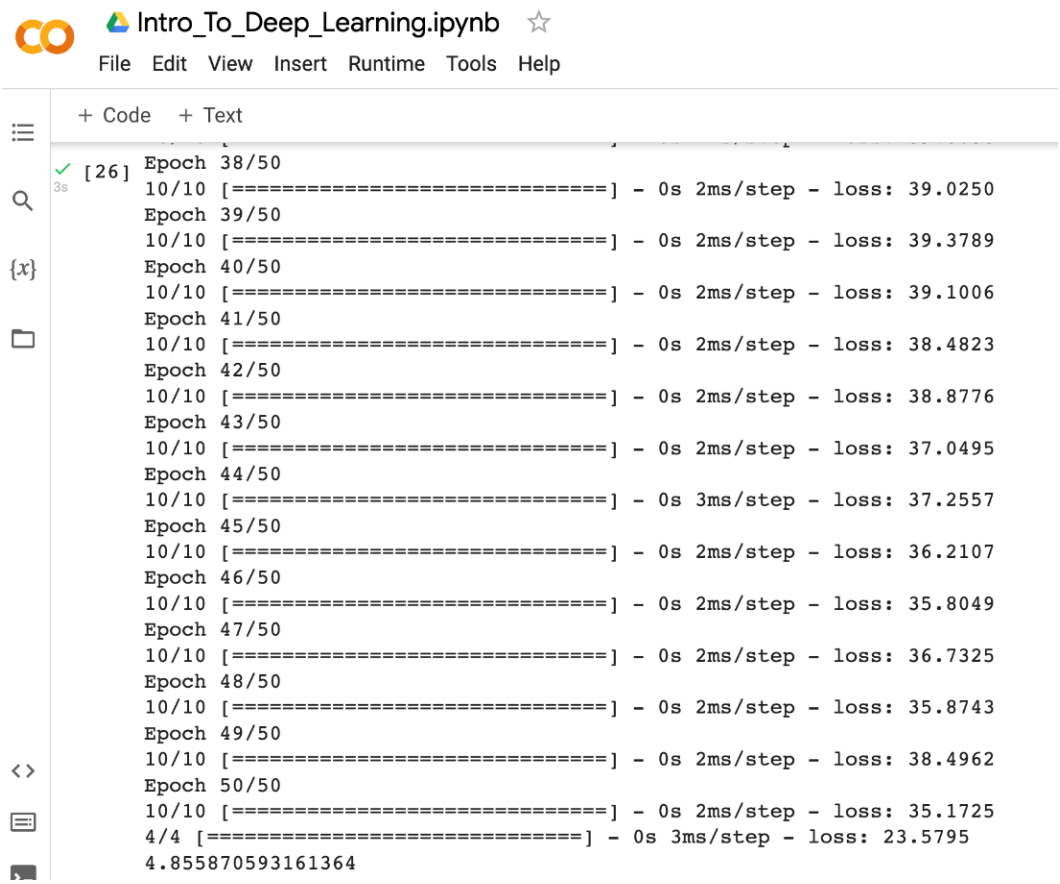


Figure 12.17 – Screenshot from Colab notebook – fifth deep learning score, RMSE of 4.86

Here is the high score from a single dense layer of 100 nodes:

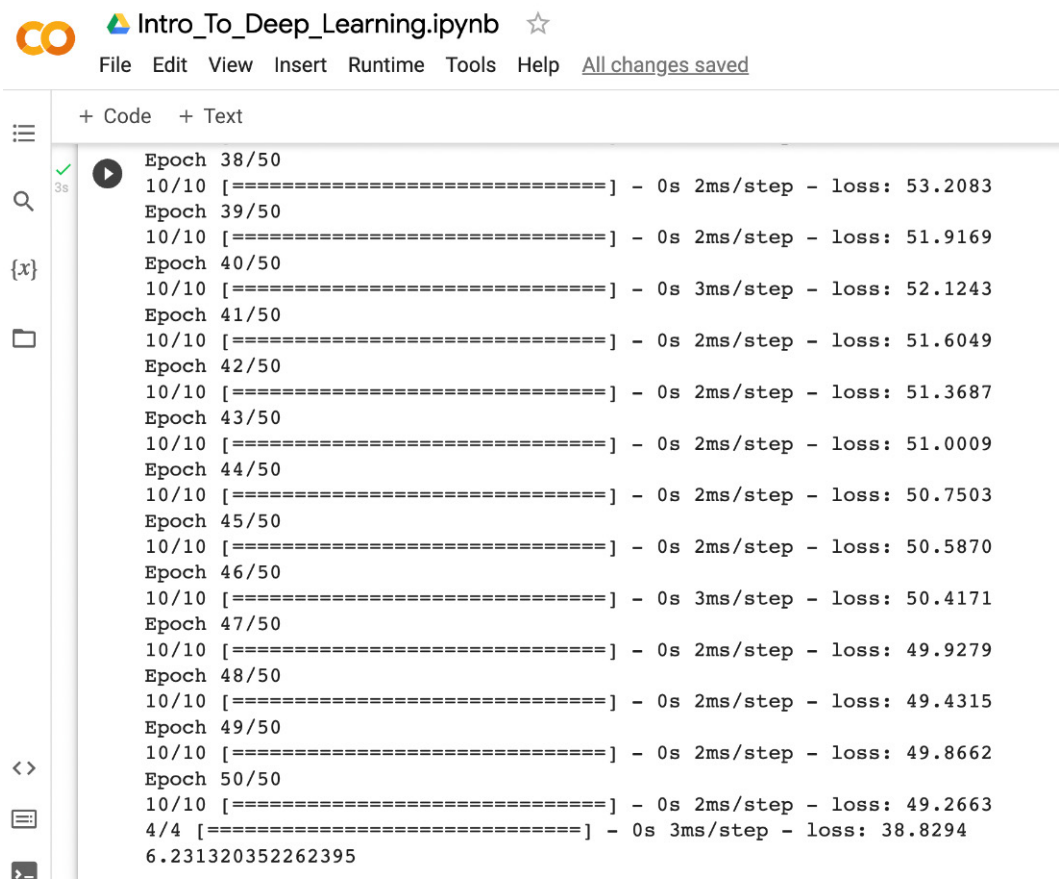


Figure 12.18 – Screenshot from Colab notebook – sixth deep learning score, RMSE of 6.23

You may wonder why the results are so different. The answer has to do with the initialization of random weights and the early learning that takes place. However, notice that in both cases, the learning loss is steadily decreasing indicating that we should use more epochs, which is what we will try next.

Number of epochs

The number of epochs, as mentioned before, is the number of times that the neural network adjusts the weights. In some respects, the more epochs the better. Overfitting the data, however, is definitely a concern, as mentioned in *Chapter 11*, *Machine Learning*, that will need addressing toward the end of this section.

Exercise 164 – modifying the number of epochs in the neural network to improve the score

The goal of this exercise is to lower the root mean squared error from the previous exercise by increasing the number of densely connected layers, and/or the number of nodes per layer:

1. Continue using the same Colab or Jupyter notebook from *Exercise 163 – modifying densely connected layers in a neural network to improve the score.*
2. Create a Sequential model with one hidden layer of 100 densely connected nodes with a `relu` activation function of 500 epochs, as shown in the following code:

```
model = Sequential()
model.add(Dense(100, input_shape=(num_cols,),
activation='relu'))
model.add(Dense(1))
print(model.summary())
model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=500)
model.evaluate(X_test, y_test)**0.5
```

The following figure shows the lowest score yet with 500 epochs using a single densely connected layer of 100 nodes:

Figure 12.19 – Screenshot from Colab notebook – seventh deep learning score, RMSE of 3.76

The 3.76 RMSE in the previous code is lower than any RMSE obtained in *Chapter 11*, *Machine Learning*. Just by estimating the number of nodes and densely connected layers, in conjunction with the number of epochs, we have achieved an optimal result.

We have tried a range of 10 epochs to 500. How can we find an optimal number of epochs?

One solution is to implement Early Stopping.

Early Stopping

Instead of guessing the number of epochs in advance, you can use the Early Stopping callback provided by Keras to find an optimal number of epochs. The Early Stopping callback works as follows:

1. You choose an arbitrarily high number of epochs that is not expected to be reached
2. The model starts working on optimizing weights as usual, one epoch at a time
3. The model stops building after the loss does not improve for *N* epochs in a row on a validation set, where *N* is a positive integer, called the patience, that you choose in advance
4. After completion, the model goes back to the top score after which no improvement was shown

The key number when using Early Stopping is the number of consecutive epochs *N*, the patience, that the model is guaranteed to build. In fact, the model will keep building, trying out new weights, until it fails to improve for *N* epochs in a row. A validation set is used to score the model on each round for early stopping instead of the training set so that the model is not at risk of grossly overfitting the training thereby undermining the Early Stopping advantage.

Choosing *N* should depend on how long it takes each epoch to run. During training, starting with a patience of 10 to 50 epochs for the early callback provides a nice balance between giving the model a chance to find new learning opportunities without waiting too long for the model to finish building.

Exercise 165 – optimizing the number of epochs with Early Stopping

The goal of this exercise is to optimize the number of epochs by using Early Stopping:

1. Continue using the same Colab or Jupyter notebook from *Exercise 164 – modifying the number of epochs in a neural network to improve the score*.
2. Import `EarlyStopping` from `keras.callbacks` and create a variable called `early_stopping_monitor` set equal to `EarlyStopping` with a parameter of patience set to `25`:

```
from keras.callbacks import EarlyStopping
early_stopping_monitor = EarlyStopping(patience=25)
```

3. Create a Sequential model with one hidden layer of 100 densely connected nodes with a `relu` activation function. Compile the model using the standard `adam` optimizer and `mse` loss function as follows:

```
model = Sequential()
model.add(Dense(100, input_shape=(num_cols,),
activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

4. Fit the model on the training set with 50,000 epochs; include the `validation_split` parameter set equal to `0.2`, and `callbacks` set equal to a list that contains `early_stopping_monitor`. Evaluate the model on the test as well. Note that the validation split will split the training set, and the final evaluation is on a separate test set. Normally, this final evaluation would be withheld until the end of training, but we present an evaluation on the test set here to build on the consistency of previous results:

```
model.fit(X_train, y_train, epochs=50000, validation_
split=0.2, callbacks=[early_stopping_monitor])
model.evaluate(X_test, y_test)**0.5
```

The Early Stopping code and the beginning of training are shown in the following figure:

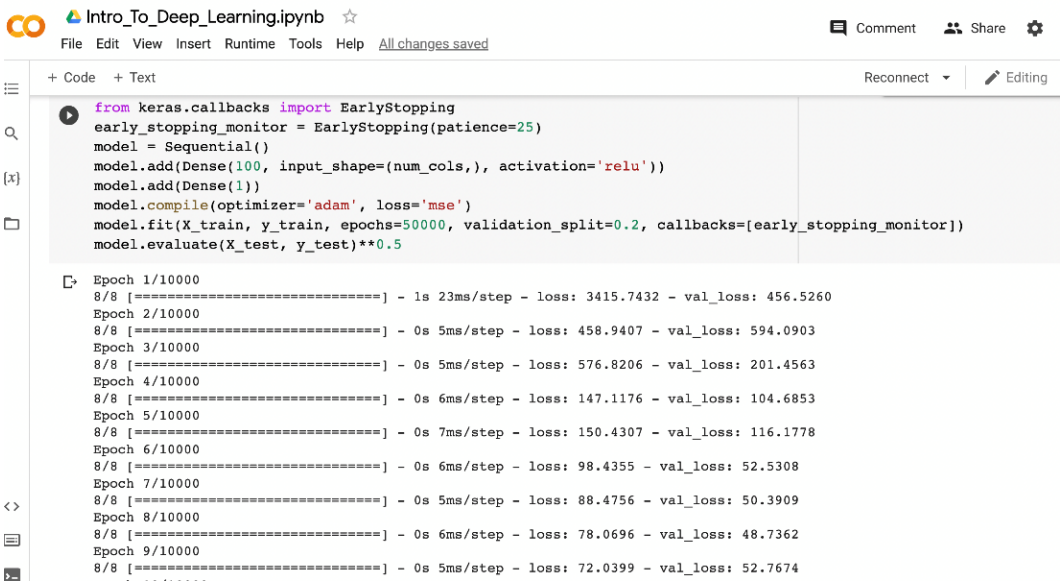


Figure 12.20 – Screenshot from Colab notebook – introducing Early Stopping

The following figure shows the end of the Early Stopping monitor with the final score:

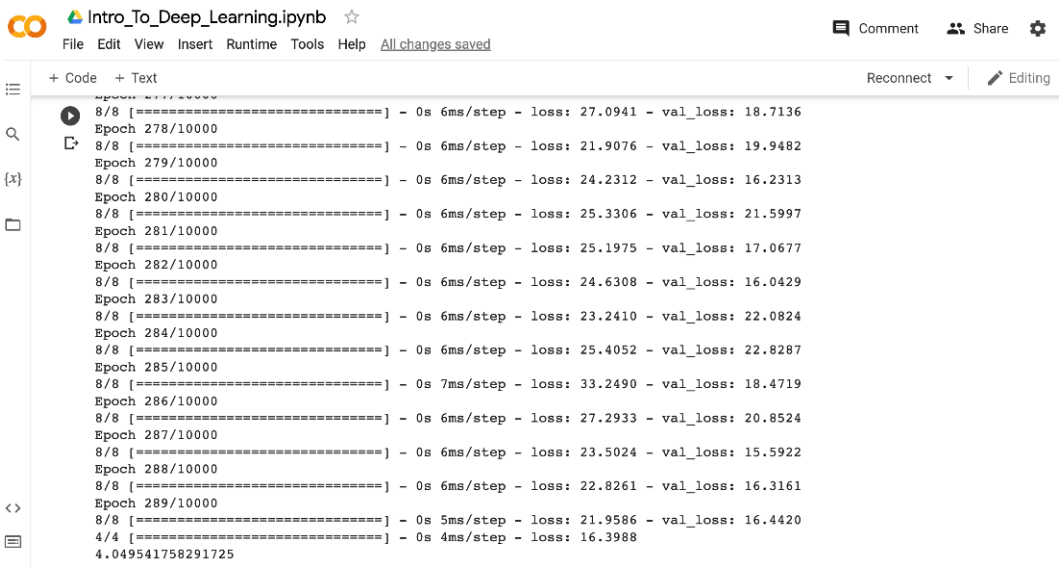


Figure 12.21 – Screenshot from Colab notebook – RMSE of 4.05 using Early Stopping

As you can see, our model finished after 289 epochs, so it failed to see an improvement in the validation score after the (*289-25=264*) 264th model.

Although the score is slightly worse than the model with 500 epochs, this is partially due to randomness. It's also possible to increase the patience. Finally, note that the Early Stopping callback stops building when the validation set fails to improve, so it's training on a smaller subset.

Additional regularization technique – Dropout

Regularization is built into the Early Stopping monitor because a validation test is used during each epoch to score against the training set. The idea is that even if the training set continues to improve, the model will stop building after the validation ceases to improve within the callback patience.

It's important to examine additional regularization techniques so that you can build even larger neural networks without overfitting the data.

Another very popular regularization technique widely used in neural networks is called the Dropout. Given multiple nodes in multiple layers result in thousands or millions of weights, neural networks can easily overfit the training set.

The idea behind Dropout is to randomly drop some nodes altogether. In densely connected networks, since all nodes in one layer are connected to all nodes in the next layer, any node may be eliminated except the last.

Dropout works in code by adding a Dropout with a certain percentage that is the probability of each node being eliminated between layers. Dropout percentages commonly range from 10 to 50%, although any number strictly between 0 and 100% is valid.

In our previous neural networks, for simplicity, we used one layer of 100 nodes. But now, by using Dropout with Early Stopping, it may be advantageous to increase the number of densely connected layers.

Exercise 166 – using Dropout in a neural network to improve the score

The goal of this exercise is to lower the root mean squared error from the previous exercise by using Dropout:

1. Open a new Colab or Jupyter Notebook. The code for our new notebook is here: `https://colab.research.google.com/drive/1lhxPKvfVfWYh6ru0OTn4EapH0qo6NBLC?usp=sharing`.
2. Import `Dropout` from `keras.layers`. Initialize a Sequential model, then add a densely connected layer with 128 nodes and a `relu` activation. After the first layer, add a Dropout of `0.1` as shown in the following code snippet:

```
from keras.layers import Dropout
model = Sequential()
model.add(Dense(128, input_shape=(num_cols,),
activation='relu'))
model.add(Dropout(0.1))
```

3. Add a `Dense` layer of 32 nodes with another `relu` activation followed by a Dropout of `0.1` and a `Dense` layer of one node. Compile the model as in the following code snippet:

```
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

4. Set Early Stopping to a patience of `50`, fit the model to the training set with an upper bound of 10,000 epochs, use a `validation_split` value of `0.2`, and evaluate on the test set as follows:

```
early_stopping_monitor = EarlyStopping(patience=50)
model.fit(X_train, y_train, epochs=10000, validation_
split=0.2, callbacks=[early_stopping_monitor])
model.evaluate(X_test, y_test)**0.5
```

The following figure shows the code altogether along with the early results using Dropout:

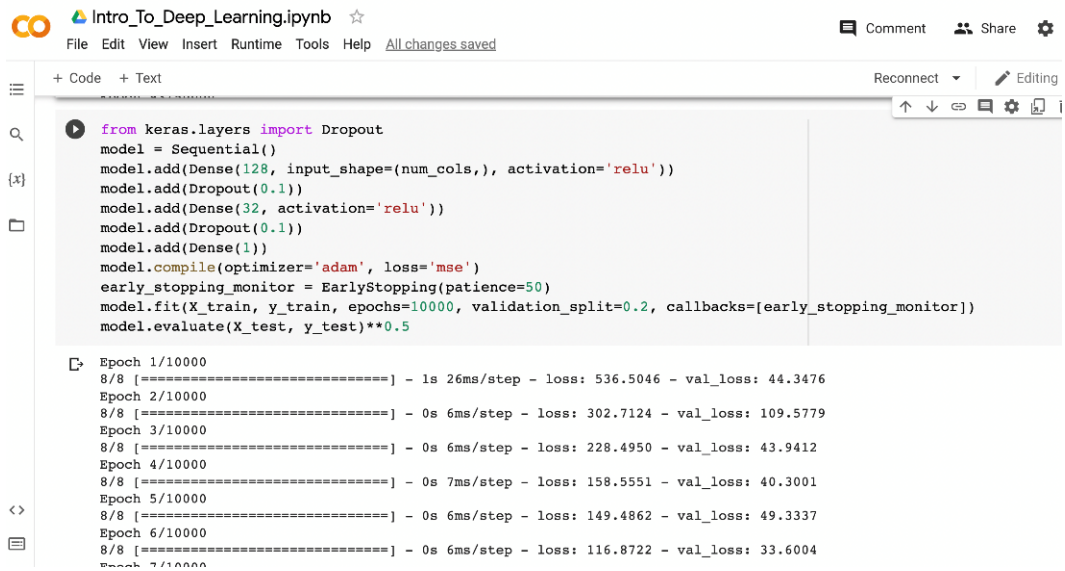


Figure 12.22 – Screenshot from Colab notebook – Dropout

The next figure shows the end of the code with the Dropout added:

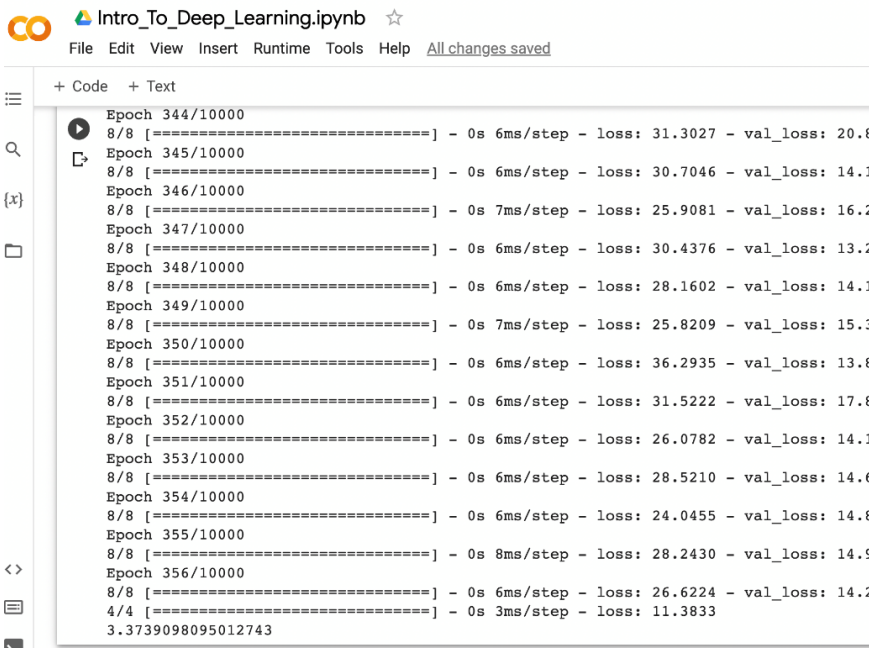


Figure 12.23 – Screenshot from Colab notebook – RMSE of 3.37 using Dropout and Early Stopping

The new RMSE of 3.37 is the lowest yet, and therefore the best. You are encouraged to experiment with the number of hidden layers and nodes, along with Dropout and Early Stopping, to aim for an even better score.

Building neural networks for classification

In the previous examples, the final output could have been any given number, so we were dealing with regression. But in many cases, the final output may be 0 or 1, "yes" or "no," or a range of distinct colors. In each of these cases, the type of machine learning algorithms that we are looking for fall under the general heading of classification.

In neural networks, one primary difference between regression and classification is the loss functions and scoring metrics. For classification, loss functions and scoring metrics are usually based on some kind of percentage of accuracy. It's standard to use `binary_crossentropy` as the loss function for classification and to include an `accuracy` metric, which is the percentage of cases the model predicts correctly.

Another important difference when building a classification model is the final node itself. In regression, we used a `Dense` layer with one node only and no activation function. All that is needed to shift the neural network to classification is a `sigmoid` activation function.

Why? Recall that the Sigmoid curve from *Chapter 11*, *Machine Learning*, transforms all possible `X` values to `y` values between `0` and `1`. In machine learning, `y` values greater than `0.5` are mapped to `1`, and `y` values less than `0.5` are mapped to `0`. So, all that is needed to convert a regression model to a classification model in a neural network, in addition to selecting the appropriate loss function and metric, is to conclude with a `Dense` layer with one node only that includes a `sigmoid` activation function.

Exercise 167 – building a neural network for classification

We are now going to build a neural network for a dataset that requires classification.

The dataset that we are going to use is the famous *Census* dataset from the UCI Machine Learning Repository, which is commonly used to predict whether adults make more or less than 50K (USD) based on census data from a variety of locations in 1994 (Data source: `https://archive.ics.uci.edu/ml/datasets/census+income`).

We will use a clean version of this dataset with `pd.get_dummies()` already applied, as taken from Packt Publishing's *Hands-On Gradient Boosting with XGBoost and Scikit-Learn*, a machine learning book written by the author, which is a great follow-up book to this one if you are interested in gaining mastery over tree-based models.

Here are the steps to build a classifier as a neural network:

1. Continue with the same Colab notebook (or Jupyter notebook) from *Exercise 166 – using Dropout in a neural network to improve the score*.
2. Load and view the data with the following code, taking `y`, the target column, as the last column, and `X` as the remaining columns. Split the data into a training and test set and view the data as in the following code snippet:

```
url = 'https://media.githubusercontent.com/media/
PacktPublishing/Hands-On-Gradient-Boosting-with-XGBoost-
and-Scikit-learn/master/Chapter08/census_cleaned.csv'
df = pd.read_csv(url)
X = df.iloc[:, :-1]
y = df.iloc[:, -1]
X_train, X_test, y_train, y_test = train_test_split(X, y,
random_state=2)
df.head()
```

Here is the expected output:

	age	fnlwgt	education-num	capital-gain	capital-loss	hours-per-week	workclass_?	workclass_Federal-gov	workclass_Local-gov	workclass_Never-worked	...	native-country_Puerto-Rico
0	39	77516	13	2174	0	40	0	0	0	0	...	0
1	50	83311	13	0	0	13	0	0	0	0	...	0
2	38	215646	9	0	0	40	0	0	0	0	...	0
3	53	234721	7	0	0	40	0	0	0	0	...	0
4	28	338409	13	0	0	40	0	0	0	0	...	0

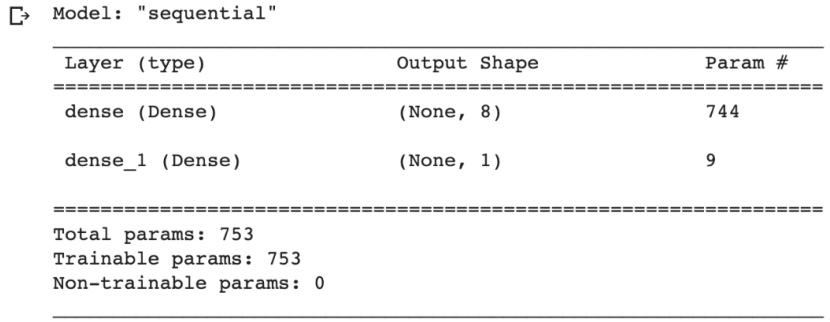
5 rows × 93 columns

Figure 12.24 – Census dataset, screenshot from Colab notebook

3. After setting the number of columns, initialize a Sequential model, then add a single dense layer with eight nodes and a `relu` activation function followed by an output layer with one node and the required `sigmoid` activation function. Include the model summary:

```
num_cols = X.shape[1]
model = Sequential()
model.add(Dense(8, input_shape=(num_cols,),
activation='relu'))
model.add(Dense(1, activation='sigmoid'))
print(model.summary())
```

Here is the expected output:



```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 8)                 744

 dense_1 (Dense)             (None, 1)                 9

=================================================================
Total params: 753
Trainable params: 753
Non-trainable params: 0
_________________________________________________________________
```

Figure 12.25 – Screenshot from Colab notebook shows 753 total parameters

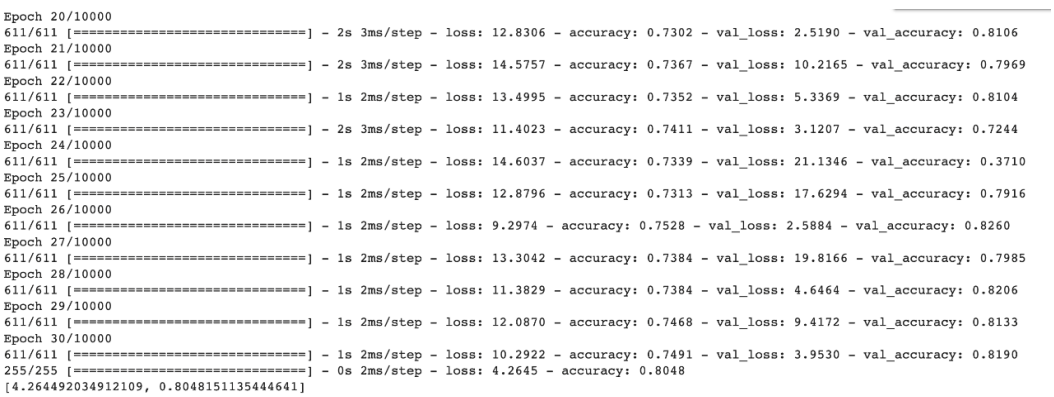
4. Compile the model with the optimizer set to `adam`, the loss set to `binary_crossentropy`, and the metrics set to a list that only includes `accuracy`, as in the following code snippet:

```
model.compile(optimizer='adam', loss='binary_
crossentropy', metrics=['accuracy'])
```

5. Set up an Early Stopping monitor with a patience of `10`. Next, fit the model to the training set, making sure to set a high number of epochs with a validation split, and add your Early Stopping monitor to the callbacks. Finally, evaluate the model on the test set and run your code as follows:

```
early_stopping_monitor = EarlyStopping(patience=10)
model.fit(X_train, y_train, epochs=10000, validation_
split=0.2, callbacks=[early_stopping_monitor])
model.evaluate(X_test, y_test)
```

Your output should be comparable to the following:



```
Epoch 20/10000
611/611 [==============================] - 2s 3ms/step - loss: 12.8306 - accuracy: 0.7302 - val_loss: 2.5190 - val_accuracy: 0.8106
Epoch 21/10000
611/611 [==============================] - 2s 3ms/step - loss: 14.5757 - accuracy: 0.7367 - val_loss: 10.2165 - val_accuracy: 0.7969
Epoch 22/10000
611/611 [==============================] - 1s 2ms/step - loss: 13.4995 - accuracy: 0.7352 - val_loss: 5.3369 - val_accuracy: 0.8104
Epoch 23/10000
611/611 [==============================] - 2s 3ms/step - loss: 11.4023 - accuracy: 0.7411 - val_loss: 3.1207 - val_accuracy: 0.7244
Epoch 24/10000
611/611 [==============================] - 1s 2ms/step - loss: 14.6037 - accuracy: 0.7339 - val_loss: 21.1346 - val_accuracy: 0.3710
Epoch 25/10000
611/611 [==============================] - 1s 2ms/step - loss: 12.8796 - accuracy: 0.7313 - val_loss: 17.6294 - val_accuracy: 0.7916
Epoch 26/10000
611/611 [==============================] - 1s 2ms/step - loss: 9.2974 - accuracy: 0.7528 - val_loss: 2.5884 - val_accuracy: 0.8260
Epoch 27/10000
611/611 [==============================] - 1s 2ms/step - loss: 13.3042 - accuracy: 0.7384 - val_loss: 19.8166 - val_accuracy: 0.7985
Epoch 28/10000
611/611 [==============================] - 1s 2ms/step - loss: 11.3829 - accuracy: 0.7384 - val_loss: 4.6464 - val_accuracy: 0.8206
Epoch 29/10000
611/611 [==============================] - 1s 2ms/step - loss: 12.0870 - accuracy: 0.7468 - val_loss: 9.4172 - val_accuracy: 0.8133
Epoch 30/10000
611/611 [==============================] - 1s 2ms/step - loss: 10.2922 - accuracy: 0.7491 - val_loss: 3.9530 - val_accuracy: 0.8190
255/255 [==============================] - 0s 2ms/step - loss: 4.2645 - accuracy: 0.8048
[4.264492034912109, 0.8048151135444641]
```

Figure 12.26 – Screenshot from Colab notebook – accuracy is on the right at 80.48 percent (loss is on the left)

Note that the output shows two numbers in a list. The number on the right, `0.8048`, is the accuracy, meaning the model is 80% accurate, and the number on the left is the loss.

That's all there is to it. As you can see, building a classifier is very similar to building a regressor provided that you take care of the loss function, the metric, and the final output layer.

Activity 26 – building your own neural network to predict whether a patient has heart disease

In this activity, you will build a neural network to solve a real-world problem. Doctors need more information to determine whether incoming patients have heart diseases after running some tests. They need a model that will correctly determine whether the patient has heart disease with 80% accuracy. They bring you on board to build a neural network that will take the patient data as input. Your goal is to predict whether new patients have heart disease. You will achieve this goal by training your neural network on the provided dataset, which includes a target column letting you know whether past patients have had heart disease or not.

Here are the steps to achieve this goal:

1. Download the dataset via the following URL: `https://media.githubusercontent.com/media/PacktPublishing/Hands-On-Gradient-Boosting-with-XGBoost-and-Scikit-learn/master/Chapter02/heart_disease.csv`.
2. Set `X`, the predictor columns, equal to all columns except the last. Set `y`, the target column, equal to the last column.
3. Split the data into a training and a test set.
4. Initialize a Sequential model.
5. Add your first Dense layer, making sure to include `input_shape=(num_cols,)`, along with the number of nodes and an activation function.
6. Include additional Dense layers and possible Dropout layers as needed.
7. Decide whether your final layer does not need an activation function, or whether it should have a `sigmoid` activation function.
8. Include an Early Stopping monitor to set the number of epochs.
9. Fit your data on the training set, making sure to include a `validation_split` value set to your desired percentage.
10. Adjust your neural network after your first scores come back until you achieve an accuracy of 80% or higher.

You should get an output similar to the following when you are finished. Recall that the accuracy is the second number in the list:

```
Epoch 72/1000
6/6 [==============================] - 0s 8ms/step - loss: 0.4675 - accuracy: 0.8232 - val_loss: 0.5302 - val_accuracy: 0.7174
Epoch 73/1000
6/6 [==============================] - 0s 9ms/step - loss: 0.3969 - accuracy: 0.8287 - val_loss: 0.3581 - val_accuracy: 0.8478
Epoch 74/1000
6/6 [==============================] - 0s 8ms/step - loss: 0.3702 - accuracy: 0.8343 - val_loss: 0.5377 - val_accuracy: 0.7174
Epoch 75/1000
6/6 [==============================] - 0s 8ms/step - loss: 0.3664 - accuracy: 0.8453 - val_loss: 0.4179 - val_accuracy: 0.8043
Epoch 76/1000
6/6 [==============================] - 0s 10ms/step - loss: 0.4303 - accuracy: 0.7845 - val_loss: 0.3773 - val_accuracy: 0.8478
Epoch 77/1000
6/6 [==============================] - 0s 8ms/step - loss: 0.4266 - accuracy: 0.8453 - val_loss: 0.6357 - val_accuracy: 0.6522
Epoch 78/1000
6/6 [==============================] - 0s 8ms/step - loss: 0.4155 - accuracy: 0.8232 - val_loss: 0.4632 - val_accuracy: 0.8261
Epoch 79/1000
6/6 [==============================] - 0s 9ms/step - loss: 0.4151 - accuracy: 0.8122 - val_loss: 0.4614 - val_accuracy: 0.7826
Epoch 80/1000
6/6 [==============================] - 0s 8ms/step - loss: 0.3488 - accuracy: 0.8453 - val_loss: 0.3515 - val_accuracy: 0.7826
3/3 [==============================] - 0s 4ms/step - loss: 0.4404 - accuracy: 0.8026
[0.44042325019836426, 0.8026315569877625]
```

Figure 12.27 – Sample final score from the neural network on the heart disease dataset

Note

A solution for this activity may be found here: `https://colab.research.google.com/drive/1O-F_0NwTlV3zMt6TrU4bUMeVhlsjLMY9#scrollTo=DUe9Oe20GUDJ`.

Convolutional neural networks

Although deep learning performs well on tabular regression and classification datasets, deep learning has a bigger advantage when making predictions from unstructured data such as images or text.

When it comes to classifying images, deep learning shines by analyzing data not one-dimensionally, but two-dimensionally, using convolutional neural networks, or CNNs for short.

Convolutional neural networks are among the strongest machine learning algorithms in the world today for classifying images. In this section, you will learn the basic theory behind convolutions before building your own CNN.

MNIST

MNIST is the name of a famous dataset of handwritten digits from 1998 that has been widely used in computer vision. The dataset consists of 60K training images and 10K test images.

Google Colab includes a smaller sample of 20K training images, along with the 10K test images, that may be directly accessed in a Colab notebook and prepared for machine learning, as in the following exercise.

Exercise 168 – preparing MNIST data for machine learning

In this exercise, you will load the MNIST data, view the data, and prepare it for machine learning with the following steps:

1. Open up a new Colab notebook at `colab.research.google.com`, then enter and run the following code in a cell:

```
import pandas as pd
df=pd.read_csv('/content/sample_data/mnist_train_small.
csv', header=None)
df_test=pd.read_csv('/content/sample_data/mnist_test.
csv',header=None)
df.head()
```

   The output is as follows:

	0	1	2	3	4	5	6	7	8	9	...	775	776	777	778	779	780	781	782	783	784
0	6	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	5	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	7	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	9	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	5	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

Figure 12.28 – MNIST one-dimensional data

> **Note**
> As the output indicates, the 0th column shows the digit that we are trying to predict, while the other 784 columns show the values of the pixels that make up the image. Most pixels have a value of 0 because there is nothing there.

2. Now, split the training and test data into `X` and `y`, choosing the 0^{th} column as the `y` value, what you are trying to predict, and the remaining 784 columns as the `X` values:

```
X = df.iloc[:, 1:]
y = df.iloc[:, 0]
X_test = df_test.iloc[:, 1:]
y_test = df_test.iloc[:, 0]
```

3. It's always worth viewing the actual images, the digits that you are trying to predict. Although this step is optional, it's definitely worth your while. Instead of viewing the data as a long string of individual pixels, it's going to be reshaped into 28x28 two-dimensional pixels and then displayed using the following annotated steps:

```
import numpy as np
# Get random index between 0 and number of rows in X
random_index = np.random.randint(0, X.shape[0])
# Get the row with random index
random_row = X.iloc[random_index, :]
# Convert random row to numpy array for reshaping
np_random_row = np.array(random_row)
# reshape image from 1D 784 cols, to 2D 28 x 28
random_image = np_random_row.reshape(28, 28)
# Show image
import matplotlib.pyplot as plt
plt.imshow(random_image, cmap='Greys')
plt.axis('off')
plt.show()
print(y[random_index])
```

The output is random, so yours will likely be different than ours, but here is one possible sample:

3

Figure 12.29 – Sample MNIST pixelated image with a correct classification label

Don't worry that the image looks pixelated. It's supposed to! It's from 1998.

4. Next, we will one-hot encode the `y` values so that instead of being represented by the number `3`, the `3` value will be encoded as follows: `[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]`. This way, the `3` value stands alone, instead of being closer to `2` and `4` than other digits, which is not what we want for image classification:

```
from keras.utils.np_utils import to_categorical
y = to_categorical(y, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
```

5. Now let's standardize `X` and reshape it into the correct size of a NumPy array, which is four-dimensional: one dimension for the number of rows, two dimensions for the 28x28 2D representation, and the final dimension is `1`, which can be adjusted to represent color. The code snippet is as follows:

```
# Standardize X
X = X/255
X_test = X_test/255
# convert X to numpy array
X = np.array(X)
X_test = np.array(X_test)
# reshape X to (rows, 28, 28, 1)
X = X.reshape(X.shape[0], 28, 28, 1)
X_test = X_test.reshape(X_test.shape[0], 28, 28, 1)
```

You are now ready to build a CNN!

CNN kernel

The main idea behind a CNN is the kernel. Since a CNN works on two-dimensional data, there must be a way of interacting with the data two-dimensionally. The general idea is to set up a kernel, which acts not on individual pixels but on squares of pixels, one square at a time.

Consider the following matrix:

$$\begin{bmatrix} 1 & 7 & 22 \\ 9 & 3 & 1 \\ 9 & 4 & 2 \end{bmatrix}$$

Figure 12.30 – Sample matrix

We can rotate through the matrix via a two-dimensional square as follows:

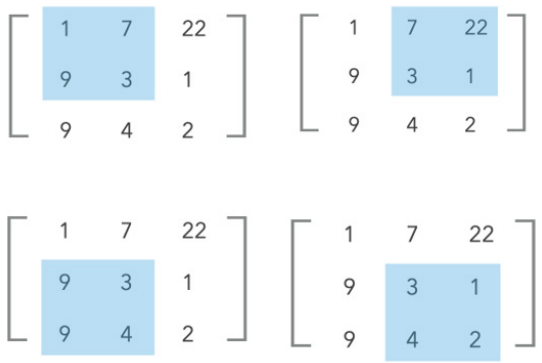


Figure 12.31 – The upper-left subset (top left), the upper-right subset (top right), the bottom-left subset (bottom left), and the bottom-right subset (bottom right) of the matrix

Each time we rotate through the matrix, we can perform a mathematical operation, also called a **convolution**, that will take each two-dimensional subset of the matrix as an input, and return a numerical output. One strategy is to choose a matrix of four numbers for the shaded square and perform the dot product with the given subset for each iteration.

Recall that the dot product of [0,1] and [1,2] is 0*1 + 1*2 = 2. The dot product multiplies each component together and sums the results.

The following three figures show how a randomly chosen matrix of [[1,2],[0,1]] may be used as a convolution to transform our original matrix into a new matrix:

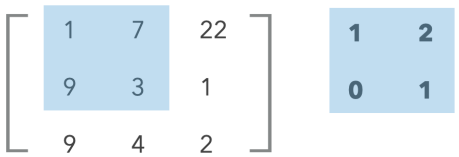


Figure 12.32 – Step A, preparing a convolution to take the dot product of the matrix on the right with the shaded matrix on the left

Figure 12.33 – Step B, the dot product of the two matrices, a convolution, gives the result of 18

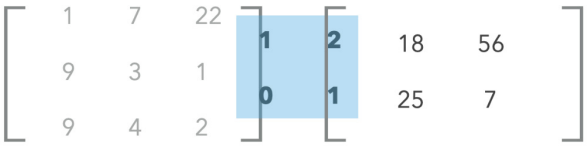


Figure 12.34 – Step C, the process of taking dot product may be repeated for each subset of the left matrix with [[1,2],[0,1]] to obtain the new matrix on the right

We started with an original matrix, then performed a convolution to transform the original matrix into a new matrix. With real data, a convolution will take the pixel values of an original image, and output variations of that image using different matrices, and different sizes of kernels. As a programmer, you choose the size of the kernel, and the neural network will choose the values of the matrix by first choosing random values, and then making adjustments to optimize predictions as before.

The theory of convolutions and computer vision is a deep and fascinating one. Although we are just scratching the surface, you now have enough background to understand the foundation behind building convolutional neural networks.

It's time to build your first CNN to optimize predictions with the MNIST dataset in the following exercise.

Exercise 169 – building a CNN to predict handwritten digits

Here are the steps to build your first CNN:

1. Import and initialize a `Sequential` model. Also import `Dense`, `Conv2D`, `Flatten`, and `MaxPool2D`:

```
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten,
MaxPool2D
model = Sequential()
```

2. Add a convolutional layer with 32 nodes that includes a kernel size of 3 and a `relu` activation. Be sure to specify `input_shape` as `(28, 28, 1)` where the `1` allows for the third dimension of color, which is not being used here. Also, note that `kernel_size` of `3` means a 3x3 matrix. It will rotate over the entire 28x28 matrix as shown in our previous example:

```
model.add(Conv2D(32, kernel_size=3, activation='relu',
input_shape=(28, 28, 1)))
```

3. Add a max pooling layer with a size of `2` (meaning 2x2). This will rotate over the entire matrix and take the largest value of every four entries. In other words, it highlights the brightest parts of the image:

```
model.add(MaxPool2D(2))
```

4. Add a second convolutional layer of 16 nodes with a kernel of `3`, and another `relu` activation. Include another `MaxPool2D` layer:

```
model.add(Conv2D(16, kernel_size=3, activation='relu'))
model.add(MaxPool2D(2))
```

5. Before creating the output layer with 10 nodes (since 10 digits) using a `softmax` activation function, which is required for multi-classification, convert the entire image into one dimension by adding a `Flatten` layer:

```
model.add(Flatten())
model.add(Dense(10, activation='softmax'))
```

6. Finally, summarize and compile your model using the same techniques described earlier in this chapter for classification. Try 20 epochs, and be sure to evaluate on the test set. Before running your model, speed up your run time by switching to a GPU in Colab: **Runtime | Change runtime type | GPU**:

```
print(model.summary())
model.compile(optimizer='adam', loss='categorical_
crossentropy',
metrics=['accuracy'])
model.fit(X, y, epochs=20)
model.evaluate(X_test, y_test)
```

The output for the model summary is as follows:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 16)        4624

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 16)         0
 2D)

 flatten (Flatten)           (None, 400)               0

 dense (Dense)               (None, 10)                4010

=================================================================
Total params: 8,954
Trainable params: 8,954
Non-trainable params: 0
```

Figure 12.35 – The CNN contains 8,954 parameters that must be optimized

The output for the model is as follows:

```
Epoch 15/20
625/625 [==============================] - 13s 21ms/step - loss: 0.0157 - accuracy: 0.9948
Epoch 16/20
625/625 [==============================] - 13s 21ms/step - loss: 0.0124 - accuracy: 0.9958
Epoch 17/20
625/625 [==============================] - 13s 21ms/step - loss: 0.0131 - accuracy: 0.9958
Epoch 18/20
625/625 [==============================] - 13s 21ms/step - loss: 0.0105 - accuracy: 0.9964
Epoch 19/20
625/625 [==============================] - 13s 21ms/step - loss: 0.0085 - accuracy: 0.9975
Epoch 20/20
625/625 [==============================] - 13s 21ms/step - loss: 0.0103 - accuracy: 0.9961
313/313 [==============================] - 3s 8ms/step - loss: 0.0760 - accuracy: 0.9810
[0.07597225904464722, 0.9810000061988831]
```

Figure 12.36 – The CNN shows an accuracy of 98.1 percent

As you can see from the preceding figure, the model successfully predicts handwritten digits with 98% accuracy, an outstanding initial result. Have fun playing around with the parameters and see whether you can get 99% or better!

Here is a link to the Colab notebook that was used in this lesson:

`https://colab.research.google.com/drive/1OQ0vMLsEj18THVLjCKqH8Ue7ZmkA7-BE?usp=sharing`.

Activity 27 – classifying MNIST Fashion images using CNNs

It's time to try CNNs on your own. Keras comes with an MNIST *Fashion* dataset that you can use to classify fashion images using the same general principles as you applied in *Exercise 169 – building a CNN to predict handwritten digits*.

The goal of this activity is to classify each fashion item that comes through a clothing sales website so that a range of sale prices may be suggested to the user depending on the type of item. Users just submit images of what they want to sell, and your job is to classify the images. You know that CNNs are the best in the world for classifying images, so you are going to use one to help your team.

Here are the steps to complete the activity.

1. Download the Keras MNIST *Fashion* dataset using the following code in a Colab notebook, making sure to set the runtime to GPU for optimal processing speed:

```
from keras.datasets import fashion_mnist
(X, y), (X_test, y_test) = fashion_mnist.load_data()
```

2. Import the necessary libraries to build your CNN. Along with `Sequential`, `Dense`, `Conv2D`, `MaxPool2D`, `Flatten`, and `Dense`, we recommend using `Dropout` to help prevent your model from overfitting and to improve scores.
3. Be sure to standardize and reshape `X`, and change `y` to `to_categorical` as in *Exercise 168 – preparing MNIST data for machine learning*. The MNIST *Fashion* dataset has the same number of categories, 10, and the data fits the same 28x28 shape as the MNIST dataset that you used to classify digits.
4. Initialize a Sequential model, then choose the size of your first convolutional layer, along with the size of your kernel.
5. Add max pooling layers, more convolutional layers, and Dropout layers as you see fit. Be judicious, recalling that more nodes result in more parameters and longer build times.
6. Evaluate your model on the test set.
7. Go back and make adjustments to improve your score.

Your output should be similar to the following:

```
Epoch 15/20
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2109 - accuracy: 0.9232
Epoch 16/20
1875/1875 [==============================] - 7s 4ms/step - loss: 0.2068 - accuracy: 0.9244
Epoch 17/20
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2037 - accuracy: 0.9257
Epoch 18/20
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2019 - accuracy: 0.9255
Epoch 19/20
1875/1875 [==============================] - 6s 3ms/step - loss: 0.1955 - accuracy: 0.9269
Epoch 20/20
1875/1875 [==============================] - 6s 3ms/step - loss: 0.1952 - accuracy: 0.9282
313/313 [==============================] - 1s 3ms/step - loss: 0.2489 - accuracy: 0.9100
[0.24885225296020508, 0.9100000262260437]
```

Figure 12.37 – An output for Activity 27 with 91% accuracy

See whether you can beat our score of 91% accuracy!

Summary

In this chapter, you learned how to build neural networks using Keras, one of the best deep learning libraries in the world. You built Sequential dense models with a variety of hidden layers and nodes using the ReLU activation function and the Adam optimizer. You used Early Stopping to find an ideal number of epochs, and you used Dropout to help prevent overfitting. Furthermore, you trained both regressors and classifiers, making sure to use `binary_crossentropy` as the classification loss function and the `sigmoid` activation function. Additionally, you learned about the foundations behind convolutions and built convolutional neural networks to classify handwritten digits with over 98% accuracy.

Congratulations on completing your deep learning journey.

The next chapter is the final chapter of the second edition of the Python Workshop, *New Features in Python*, which includes updates from Python 3.7 to Python 3.11.

13

The Evolution of Python – Discovering New Python Features

Overview

By the end of his chapter, you will understand how Python continues to evolve and how to track that evolution to be up to date with the latest development. The chapter will introduce you to the **Python Enhancement Proposals** (**PEPs**) and show you the most significant enhancements of the language, from Python 3.7 to Python 3.11, allowing you to leverage the new features of the language.

Introduction

Across this book, we have seen how to use Python effectively, and the different tools and APIs that the language offers us. However, Python is not a language set in stone; it continues to evolve with every new release.

The Python development team cuts a release of the interpreter every year and provides a window of support, where bug fixes are backported, as well as a long-term support window for critical security fixes only.

In this chapter, we will see how to keep us up to date with the development of Python, how enhancements are made, and the changes that the latest versions of Python have published.

We will be covering the following topics:

- Python Enhancement Proposals
- New features released in Python from version 3.7 to version 3.11

Python Enhancement Proposals

The Python language evolves as its reference implementation changes (**CPython**). The process to introduce a change in the reference implementation and, therefore, the language is done by following the Python developer's guide (`https://devguide.python.org/`). An important part of the evolution of the language is the **Python Enhancement Proposal** (**PEP**), a step required for any *major* change in the language. The process starts with a core developer (a person with the commit bit in `python/cpython`) who sponsors or directly submits a draft PEP in the `python/peps` repository. Those proposals are usually first discussed in the Python ideas forum to gather a quick opinion by both developers and users alike on how useful they are or what issues they might face.

> **Tip**
>
> A great way to be involved in the evolution of the language is to subscribe to the forum and participate in those conversations.

After a core developer submits a PEP for review, the steering council, the governing body of the Python language, discusses it. The steering council takes input from the rest of the core developers, and if they find the change valid, it can be marked as *final* and accepted for the language. PEPs are used to propose changes to the language implementation or any process related to the development of Python. PEP 1 documents the process of submitting and reviewing PEPs.

> **Tip**
>
> Subscribe to the PEP discussions at `https://discuss.python.org/c/peps` to follow major changes in the language.

A PEP usually includes the following sections:

- A metadata header.
- An abstract with a short description and motivation of the PEP, and why it is needed.
- A rationale explaining the decisions taken when writing the PEP.
- A detailed PEP specification, how can it impact existing Python code, whether it has any security implications, and how you expect trainers to teach the new feature.
- A reference implementation, if available, rejected ideas when exploring the PEP, and a copyright note. You can find a template of a PEP in PEP 12 (`https://peps.python.org/pep-0012/`).

If you plan to work professionally using Python, I recommend you read some of the following PEPs:

- **PEP 8**: A style guide on Python code. This is a must-read if you are writing Python professionally. It allows Python developers across the globe to write and read Python with a common style, making it easier to read for everyone.
- **PEP 1**: A PEP that documents the purpose and instructions on how to submit a PEP. It explains what a PEP is, the workflow for submitting one, and detailed instructions on how to create a *good PEP*.
- **PEP 11**: The different levels of support that CPython offers on different platforms.
- **PEP 602**: The new Python annual release schedule.

In the next sections, we will be looking at new features available in each Python version, starting with Python 3.7.

Python 3.7

Python 3.7 was released in June 2018, received bug fixes until June 2020, and will receive security patches until June 2023.

Built-in breakpoint

The new built-in `breakpoint()` function allows you to quickly drop into a debugger by just writing it anywhere in your code. Rather than having to call the common idiom of `import pdb;pdb.set_trace()`, in this version of Python, you can just use the built-in `breakpoint()`, which not only works with the default Python debugger (`pdb`) but any other debugger that you might use.

Module dynamic attributes

Python is an object-oriented programming language. Everything is an object in Python, and with PEP 562, modules can behave more like classes! With the addition of the work done by PEP 562, you can now add a `__getattr__` function to your module that allows you to dynamically evaluate the querying of an attribute.

This is useful when you need to deprecate an attribute of your module, if you need to perform some catching, or if something that you initially declared as an attribute now needs to do some runtime evaluation.

Additionally, you can combine `__getattr__` with `lru_cache` to lazily evaluate the attributes of your module. This is useful when you have a module with constants that are expensive to compute. That allows you to move from the following:

```
# constants_package.py
constant1 = expensieve_call()
constant2 = expensieve_call2()
```

To:

```
# constants_package.py
_constant_resolution = {
    "constants1": expensive_call,
    "constants2": expensive_call2,
}
@functools.lru_cache(maxsize=None)
def __getattr__(name):
    try:
        return _constant_resolution[name]()
    except KeyError:
        raise AttributeError(f"module {__name__!r} has no
attribute {name!r}")
```

The second version of the code will allow you to get the same results without greedily evaluating those constants. In addition, by using `lru_cache`, no matter how many times users query the attribute, Python will execute each function only once.

Nanosecond support in a time module

As we saw in *Chapter 6*, *The Standard Library*, we can use the Python `time` APIs to get the current time in multiple ways via functions such as `time.time` and `time.monotonic`, but those APIs return the time in `float`, which is usually sufficient in most scenarios, but it might not be adequate if we need an accurate result that can be used with detailed precision. This resulted in PEP 564, which adds a new function to the `time` module that allows you to get the time with nanosecond precision as `integer`. The PEP added new functions that end with the `_ns` prefix, which can be used in situations where we care about getting the precise time. This new API allows the user to work with time using integers, therefore assuring that their computations will always preserve nanosecond precision.

The dict insertion order is preserved

Since Python 3.7, we can rely on Python dictionaries preserving their insertion order, allowing us to iterate them while having a deterministic result. You can see its effect by running the following code in an interpreter before 3.6 and one after 3.6, as even if this was already happening in 3.6, it was not until 3.7 that it was guaranteed by the standard:

```
x = {}
x["a"] = 1
x["b"] = 2
x[0] = 3
print(list(x))
```

In Python 2.7, the result will be `['a', 0, 'b']`, and you should not rely on the order of the keys, as there are no guarantees. However, if you are using Python 3.7+, you can be sure that the order of the keys is always going to be `['a', 'b', 0]`. That is fantastic, as it makes the dictionary (and sets) an ordered container (which is different from a sorted one). This is a property that few languages provide.

Dataclasses

PEP 567 brought dataclasses to Python 3.7. Before this version, users relied on the third-party `attrs` package, which had a continuously growing popularity. To know more about how to use dataclasses, refer to *Chapter 6*, *The Standard Library*, and *Exercise 86 – using the dataclass module*.

Importlib.resources

This new module in the standard library allows developers to load and read resources within modules. Using `importlib.resources` allows us to tell an interpreter to read a resource without having to provide paths, making it resilient to package managers that might relocate files. This module also allows us to model packages that might not have a disk representation.

Loading data that is part of a module could not be easier with this module now. There are two APIs that you will usually rely on: `importlib.resources.files(package)` to get all the files that a Python package provides and `importlib.resources.open_text/open_binary(package, resource)` to load a file.

Python 3.8

Python 3.8 was released in October 2019, received bug fixes until May 2021, and will receive security patches until October 2024.

Assignment expression

One of the most known additions to Python 3.8 is the assignment expression, also known as the **walrus** operator. It was quite a controversial addition to Python, which many people attribute to the stepping down of Guido van Rossum from the Python's final decision-making role in the CPython evolution.

This new syntax allows developers to write an assignment in the place of an expression. This allows for shorter code by combining what otherwise needs to be multiple lines of code. This is quite useful in control flow operations when combined with reading data or using regular expressions. See the following examples.

This is without PEP 572:

```
running = True
while running:
    data = get_more_data()
    if not data:
        running = check_if_running()
    business_logic(data)
```

This is with PEP 572:

```
while data := get_more_data():
    business_logic(data)
```

In the example, you can see how by using the `:=` operator, we save multiple lines of code, making the code quicker and arguably easier to read. You can treat the result of the assignment expression as an expression, allowing you to write the following code:

```
while len(data := get_more_data) >= 1
```

functools.cached_property

This new and terribly handy function allows you to optimize your code by allowing you to do in one line a common Python idiom that was used to cache a class attribute, which might be expensive to compute. Before Python 3.8, you would commonly find code like the following:

```
class MyClass:
    def __init__(self):
        self._myvar = None
    @property
    def myvar(self):
```

```
        if self._myvar is None:
            self._myvar = expensive_operation()
        return self._myvar
```

With the addition of `cached_property`, you can now simplify that to the following:

```
class MyClass:
    @functools.cached_property
    def myvar(self):
        return expensive_operation()
```

importlib.metadata

A new module was added in Python 3.8 that lets us read metadata about third-party packages that we have installed in our system. `importlib.metadata` can be used to replace usage of less efficient and third-party dependent code that relies on `pkg_resources`. See the following examples of how this new module is useful on a Python installation with `pytest` installed:

```
import importlib.metadata
importlib.metadata.version("pytest")
```

You get the following result:

```
0.32.3
```

Figure 13.1 – The pytest version

You can get any kind of metadata by getting it as a dictionary, by invoking the `metadata` function:

```
import importlib.metadata
importlib.metadata.metadata("pytest")["License"]
```

Here is the output:

```
MIT license
```

Figure 13.2 – The pytest license

typing.TypedDict, typing.Final, and typing.Literal

If you like to type your Python code, 3.8 brings three new classes to the typing module, which are quite useful to better qualify the types you use in your code.

Using `typing.Literal` allows you to type your code to specify what concrete values it can get beyond just documenting the type. This is specifically useful in situations where strings can be passed but there is only a known list of values. See the following example:

```
MODE = Literal['r', 'rb', 'w', 'wb']
def open_helper(file: str, mode: MODE) -> str:
```

Without `typing.Literal`, you will need to type `mode` as `str`, allowing other strings that are not valid types. In 3.8, you can also use `typing.Final`, which allows you to mark a variable as a constant, and the type checker will flag an error if you try to change the value of the variable.

Finally, we have `typing.TypedDict`, a great way to type your dictionaries when you know they need to have a specific set of keys. If you create a type with `Point2D = TypedDict('Point2D', x=int, y=int)`, the type checker will flag errors when you create dictionaries with a key that is neither `x` nor `y`.

f-string debug support via =

How many times have you written the name of a variable followed by its value? With Python 3.8, this just became a lot easier with debug support in f-strings using =. With this addition, you can now write code as follows to quickly debug your variables:

```
import datetime
name = "Python"
birthday = datetime.date(1991, 2, 20)
print(f'{name=} {birthday=}' )
```

This will produce the following output:

```
name='Python' birthday=datetime.date(1991, 2, 20)
```

Figure 13.3 – An f-string example

Positional-only parameters

If you are an API provider, you will definitely like this new addition to Python. With PEP570, you can now mark parameters as positional only, making the name of the function parameter `private`, and allowing you to change it in the future if so desired. Before Python 3.8, if you were creating an API

with a signature such as `def convert_to_int(variable: float):`, users could call your function with the `convert_to_int(variable=3.14)` syntax. That could be an issue if you wanted to rename your variable in the future or wanted to move to `varargs`. With the addition of positional-only parameters to the language, you can now use new syntax to mark those arguments as positional only, preventing them from being passed using a `def convert_to_int(variable: float, /):` keyword. When `/` is specified, all arguments before it will be marked as positional only, similar to how `*` can be used to mark all arguments after it as keyword-only.

Python 3.9

Python 3.9 was released in October 2020, received bug fixes until May 2022, and will receive security patches until October 2025.

PEG parser

One of the most significant changes in Python 3.9 is the rewrite of the parser that sits at the core of an interpreter. After 30 years of using the LL1 parser, which was quite useful for Python, the core development team decided to move to a more modern and powerful parser, which enabled many enhancements to the language – from new syntax to better error messages. While this did not result in any big change directly for developers, it has helped the language to continue evolving. Take a read at `https://peps.python.org/pep-0617/` to understand the work that was done and how it is helping Python evolve.

Support for the IANA database

If you are working with time zones, you probably have used the IANA database (`https://www.iana.org/time-zones`) before. This database allows you to map strings to data that defines what offset to set for that time zone when given a date time. Before Python 3.9, two third-party packages, `dateutil` and `pytz`, provided this data to developers. With the implementation of PEP 615, developers can now fetch time zone information from their OS without the need to rely on a third-party package.

See the following example that converts a date time from the New York time zone to Los Angeles, all with the standard library:

```
import datetime
from zoneinfo import ZoneInfo
nyc_tz = ZoneInfo("America/New_York")
la_tz = ZoneInfo("America/Los_Angeles")
dt = datetime.datetime(2022, 5, 21, hour=12, tzinfo=nyc_tz)
print(dt.isoformat())
```

You will get the following result:

```
2022-05-21T12:00:00-04:00
```

Figure 13.4 – The datetime iso formatted

We can see how both the time and the offset change when we convert the `datetime` instance to a different time zone using `astimezone`:

```
print(dt.astimezone(la_tz).isoformat())
```

Now, the output will be the following:

```
2022-05-21T09:00:00-07:00
```

Figure 13.5 – The datetime iso formatted after the time zone change

Merge (|) and update (|=) syntax for dicts

Sets and dictionaries are getting closer and closer functionally. In this version of Python, dicts got support for the | union operator. This allows you to combine dictionaries with the following syntax:

```
d1 = dict(key1="d1", key3="d1")
d2 = dict(key2="d2", key3="d2")
print(d1 | d2)
```

This is the output:

```
{'key1': 'd1', 'key3': 'd2', 'key2': 'd2'}
```

Figure 13.6 – The dict merge output

Something to note is that if a key is present in both dictionaries, it will take the value from the last seen dictionary. Additionally, you can use the |= operator to merge an existing dictionary with another:

```
d1 = dict(key1="d1", key3="d1")
d1 |= dict(key2="d2", key3="d2")
print(d1)
```

The output observed is as follows:

```
{'key1': 'd1', 'key3': 'd2', 'key2': 'd2'}
```

Figure 13.7 – The dict merge operator output

str.removeprefix and str.removesuffix

With these two functions, we can remove the suffix or prefix of a string, something that many developers mistakenly used to do with `strip`. The `strip` function takes an optional list of characters to override the default and developers got confused, thinking that it was the exact string that would be removed. See the following example:

```
print("filepy.py".rstrip(".py"))
```

This gives the output as the following:

```
file
```

Figure 13.8 – The rstrip output

Users might have expected `filepy` as the result, but instead, just `file` is returned, as `strip` has been instructed to delete all `p`, `y`, and `.` characters from the end of the string. If you want to remove the suffix of a string, you can now use `str.removesuffix` instead:

```
print("filepy.py".removesuffix(".py"))
```

We will now get the expected output:

```
filepy
```

Figure 13.9 – The removesuffix output

Type hints with standard collections

Before Python 3.9, typing collections needed to import their types from the typing module. With the addition of PEP 585, developers can now use the standard library collections when type-hinting their code. This transforms the existing code from the following:

```
from typing import Dict, List
def myfunc(values: Dict["str", List[int]]) -> None:
```

To the following:

```
def myfunc(values: dict["str", list[int]]) -> None:
```

Python 3.10

Python 3.10 was released in October 2021, will receive bug fixes until May 2023, and will receive security patches until October 2026.

Pattern matching – PEP 634

By far, the most controversial addition to the Python 3.10 pattern matches was bringing `match` and `case` to the language. This addition consists of three different PEPS: PEP 634, PEP 635, and PEP 636. This new syntax allows you to mirror-switch structures that you might have seen in other languages:

```
match code:
    case 1:
        print("Working as expected")
    case -1 | -2 | -3:
        print("Internal Error")
    case _:
        print("Unknown code")
```

Note that to specify one of the multiple values, you need to use the | operator and not a comma. Using a comma will try to match a list. However, using dictionaries will be more correct for the previous example; the power of pattern matching comes from matching a variable, whose type or length in the case of containers is a lot more dynamic. Pattern matching allows you to evaluate specific properties of an object and copy those in variables when doing a match. See the following example:

```
match x:
    case {"warning": value}:
        print("warning passed with value:", value)
    case ["error", value]:
        print("Error array passed with value:", value)
```

Pattern matching is also useful when interacting with data in the form of containers and having to take different actions or create different objects based on their values. See the following example from the Python standard library:

```
match json_pet:
    case {"type": "cat", "name": name, "pattern": pattern}:
        return Cat(name, pattern)
    case {"type": "dog", "name": name, "breed": breed}:
        return Dog(name, breed)
    case _:
        raise ValueError("Not a suitable pet")
```

Note how pattern matching not only routes the code through one branch or another based on the attributes that we are matching but also captures others with specific variables. If you want to know more about pattern matching and understand how it works, we recommend you read `https://peps.python.org/pep-0636/`, which is a tutorial on how to use structural pattern matching.

Parenthesized context managers

Thanks to the introduction of the new PEG parser in Python 3.9, 3.10 was able to address a long-standing issue in Python grammar – allowing the use of parentheses in context managers.

If you have written multiple context managers in Python, you are probably aware of how hard it is to nicely format that code. This change allows you to move from having to write code such as the following:

```
with CtxManager1(
    ) as example1, CtxManager2(
    ) as example2, CtxManager3(
    ) as example3
):
```

To being able to write code such as the following:

```
with (
    CtxManager1() as example1,
    CtxManager2() as example2,
    CtxManager3() as example3,
):
```

Better error messages

Another advantage of the new parser is the new ability to write code to better handle errors in an interpreter. While Python errors are usually quite informative compared to other languages, when an error happens at parsing time, it is often quite cryptic.

Let's take the following code, which is missing a closing bracket in the first line:

```
d = {"key": "value", "key2": ["value"]
def func(): pass
```

Running it in a Python interpreter before Python 3.10 will give us the following error, which does not reference the first line at all and, therefore, is quite hard to debug:

```
$ python2.7 example.py
  File "example.py", line 2
    def func(): pass
      ^
SyntaxError: invalid syntax
```

Figure 13.10 – A previous error output

In Python 3.10, the error message will be the following:

```
$ python3.10 example.py
  File "/home/mcorcherojim/tmp/packt/example.py", line 1
    d = {"key": "value", "key2": ["value"]
        ^
SyntaxError: '{' was never closed
```

Figure 13.11 – The improved error output

This nicely points developers to the root cause of the issue.

Similar to missing brackets, there have been similar improvements to many other syntaxes, which saves developers time when developing by pointing them to the source of the issue.

Type union operator (|) – PEP 604

Python 3.10 brings some additional syntax sugar for typing. A common situation when type-hinting your code is that a parameter might have one of many types. This used to be handled by using the `typing.Union` type, but since Python 3.10, you can use the `|` operator to represent the same.

That allows you to move from writing code as the following:

```
def parse_number(text: str, pattern: typing.Union[str,
re.Pattern]) -> typing.Union[int, float]
```

To the following instead:

```
def parse_number(text: str, pattern: str | re.Pattern) ->int |
float
```

Statistics – covariance, correlation, and linear_regression

The Python 3.10 release adds functions to compute the covariance, the correlation, and the linear regression given two inputs:

```
>>> x = range(9)
>>> y = [*range(3)] * 3
>>> import statistics
>>> statistics.covariance(x, y)
0.75
>>> statistics.correlation(x, y)
0.31622776601683794
>>> statistics.linear_regression(x, y)
LinearRegression(slope=0.1, intercept=0.6)
```

Python 3.11

Python 3.11 was released in October 2022, will receive bug fixes until May 2024, and will receive security patches until October 2027.

Faster runtime

The new 3.11 is 22% faster than 3.10 when measured with the Python performance benchmark suite. The result depends a lot on your application and will usually range between 10% and 60%. A series of optimization into how code is parsed and run together with startup improvements have made this possible, as part of a project branded as **Faster CPython** that is focusing on making an interpreter faster.

Enhanced errors in tracebacks

Building on the success achieved with the improvement of error messages in Python 3.10, 3.11 has done substantial work to facilitate the debugging of errors in traceback through PEP 659. The interpreter will now point to the exact expression that caused the exception, allowing a developer to quickly figure out the root issue without using a debugger.

This is quite useful when navigating dictionaries, given the following code:

```
d = dict(key1=dict(key2=None, key3=None))
print(d["key1"]["key2"]["key3"])
```

Before Python 3.11, we will get the following error:

```
$ python3.9 example.py
Traceback (most recent call last):
  File "/home/mcorcherojim/tmp/packt/example.py", line 2, in <module>
    print(d["key"]["key2"]["key3"])
TypeError: 'NoneType' object is not subscriptable
```

Figure 13.12 – The previous dict error output

With Python 3.11, we get the following:

```
$ python3.11 example.py
Traceback (most recent call last):
  File "/home/mcorcherojim/tmp/packt/example.py", line 2, in <module>
    print(d["key"]["key2"]["key3"])
          ~~~~~~~~~~~~~~~~~^^^^^^^^
TypeError: 'NoneType' object is not subscriptable
```

Figure 13.13 – The enhanced dict error output

Note how the interpreter is now pointing us to the lookup that caused the error. Without this information, it would be hard to know where that `NoneType` was coming from. Here, the developer can easily realize that the exception was triggered when querying `key3`, meaning that the result of looking up `key2` was `None`.

This is also quite useful when doing math operations. See the following code example:

```
x = 1
y = 2
str_num = "2"
print((x + y) * int(str_num) + y + str_num)
```

Before Python 3.11, we would get the following error:

```
$ python3.10 example.py
Traceback (most recent call last):
  File "/home/mcorcherojim/tmp/packt/example.py", line 4, in <module>
    print((x + y) * int(str_num) + y + str_num)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Figure 13.14 – The previous addition error output

In Python 3.11, we get the following instead:

```
$ python3.11 example.py
Traceback (most recent call last):
  File "/home/mcorcherojim/tmp/packt/example.py", line 4, in <module>
    print((x + y) * int(str_num) + y + str_num)
          ~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Figure 13.15 – The enhanced addition error output

The new tomllib package

Given the standardization and raising popularity of `pyproject.toml`, Python 3.11 has added a new module to facilitate reading TOML files. The `tomllib` package can be used to easily read your project configuration in files such as `pyproject.toml`. As an example, let's take the following `.toml` file:

```
[build-system]
requires = ["setuptools", "setuptools-scm"]
build-backend = "setuptools.build_meta"
[project]
name = "packt_package"
description = "An example package"
dependencies = [
        "flask",
        "python-dateutil",
]
[project.scripts]
example-script = "packt_package._main:main"
```

We can now read it in Python with the standard library with the following code:

```
import tomllib
import pprint
with open("pyproject.toml", "rb") as f:
    data = tomllib.load(f)
pprint.pprint(data)
```

This generates the following output:

```
$ python3.11 tomllib_example.py
{'build-system': {'build-backend': 'setuptools.build_meta',
                  'requires': ['setuptools', 'setuptools-scm']},
 'project': {'dependencies': ['flask', 'python-dateutil'],
             'description': 'An example package',
             'name': 'packt_package',
             'scripts': {'example-script': 'packt_package._main:main'}}}
```

Figure 13.16 – The tomllib output

This allows us to handle TOML similar to how we can handle JSON with `stdlib`. The main difference is that the `tomllib` module does not come with a method to generate TOML, for which developers have to rely on third-party packages, which have different ways of customization and formatting.

Required keys in dicts

If you have been type-hinting your code, this will allow you to go a more strict level in your Python dictionaries. In the past, we saw how we could use `TypeDict` to declare what keys a dictionary could take, but now with PEP655, there is a new way to mark whether keys are required or not. Using our previous example of a point, we can now add an optional `map` attribute as `TypedDict('Point2D', x=int, y=int, map=NotRequired[str])`. That will result in the type checker allowing `dict(x=1, y=2)` and `dict(x=1, y=2, map="new_york")` but not one that misses either the `x` or `y` keys, such as `dict(y=2, map="new_york")`.

The new LiteralString type

Another addition to type-hinting is the new `LiteralString` type. This is useful when we are passing strings that are going to be used in SQL statements or shell commands, as a type checker will require that only static strings be passed. That helps developers protect their code from SQL injection and other similar attacks that take advantage of strings interpolation. See the following example that defines an API for a database:

```
 def get_all_tables(schema_name: str) -> list[str]:
    sql = "SELECT table_name FROM tables WHERE schema=" +
schema_name
    …
```

The developer of this API intended that function to allow other developers to call it as a quick way to get all tables given a schema. The developer considered it safe code as long as the `schema_name` argument was under the control of the developer, but there was nothing to prevent that. A user of this API could write the following code:

```
schema = input()
print(get_all_tables(schema))
```

This allows the user to perform a SQL injection attack by passing to be input a string such as `X; DROP TABLES`. With PEP 675, the library developer can now mark `schema_name` as `LiteralString`, which will make the type checker raise an error if the string is not static and a part of the application code.

Exceptions notes – PEP 678

PEP 678 adds a new method, `add_note`, to all exceptions, allowing developers to enrich an exception without the need of having to raise a new one. Before this addition, it was quite common to find the following code, as developers wanted to enrich an exception with some additional information:

```
def func(x, y):
    return x / y
def secret_function(number):
    try:
        func(10_000 , number)
    except ArithmeticError as e:
        raise ArithmeticError(f"Failed secret function: {e}")
from e
```

With exception notes, we can now write the following:

```
def func(x, y):
    return x / y
def secret_function(number):
    try:
        func(10_000 , number)
    except ArithmeticError as e:
        e.add_note("A note to help with debugging")
        raise
```

This allows the exception to keep all its original information. Let's now run the following code:

```
secret_function(0)
```

We see the following traceback:

```
$ python3.11 exception_notes.py
Traceback (most recent call last):
  File "/home/mcorcherojim/tmp/packt/Chapter13/exception_notes.py", line 11, in <module>
    secret_function(0)
  File "/home/mcorcherojim/tmp/packt/Chapter13/exception_notes.py", line 6, in secret_function
    func(10_000, number)
  File "/home/mcorcherojim/tmp/packt/Chapter13/exception_notes.py", line 2, in func
    return x / y
           ~~^~~
ZeroDivisionError: division by zero
A note to help with debugging
```

Figure 13.17 – An exceptions notes example

With this, we conclude our review of the new Python features.

Summary

In this final chapter, you have taken your Python knowledge one step further by learning how to continue your journey of improving your Python skills. We have seen the process to enhance Python and the enhancements that the language has accommodated in the most recent releases. You are all set up to continue your Python learning and even ready to submit a proposal for enhancements if you have any good ideas on how to improve the language itself!