Hands-On Genetic Algorithms with Python

Applying genetic algorithms to solve real-world deep learning and artificial intelligence problems

Eyal Wirsansky

Packt
www.packt.com

Hands-On Genetic Algorithms with Python

Applying genetic algorithms to solve real-world deep learning and artificial intelligence problems

Eyal Wirsansky

Packt

BIRMINGHAM - MUMBAI

Hands-On Genetic Algorithms with Python

Commissioning Editor: Sunith Shetty
Acquisition Editor: Porous Godhaa
Content Development Editor: Pratik Andrade
Senior Editor: Ayaan Hoda
Technical Editor: Mohd Riyan Khan
Copy Editor: Safis Editing
Project Coordinator: Anish Daniel
Proofreader: Safis Editing
Indexer: Priyanka Dhadke
Production Designer: Deepika Naik

First published: January 2020

Production reference: 1300120

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-83855-774-4

www.packt.com

To my wife, Jackie, for her love, patience, and support. To my children, Danielle and Liam, whose creativity and artistic talents inspired me in writing this book.

– Eyal Wirsansky

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Eyal Wirsansky is a senior software engineer, a technology community leader, and an artificial intelligence enthusiast and researcher. Eyal started his software engineering career as a pioneer in the field of voice over IP, and he now has over 20 years' experience of creating a variety of high-performing enterprise solutions. While in graduate school, he focused his research on genetic algorithms and neural networks. One outcome of his research is a novel supervised machine learning algorithm that combines the two.

Eyal leads the Jacksonville (FL) Java user group, hosts the Artificial Intelligence for Enterprise virtual user group, and writes the developer-oriented artificial intelligence blog, ai4java.

I would like to thank my family and close friends for their patience, support, and encouragement throughout the lengthy process of writing this book. Special thanks go to the Jacksonville Python Users Group (PyJax) for their feedback and support.

About the reviewer

Lisa Bang did her BS in marine biology at UC Santa Cruz, and an MS in bioinformatics at Soongsil University in Seoul under the tutelage of Dr. Kwang-Hwi Cho. Her masters' thesis was on a method for making QSARs reproducible using Jupyter Notebook, and contained a genetic algorithm component to reduce search space. This is now being developed into DEAP-VS to be compatible with Python 3. She also worked at Geisinger Health System as part of the Biomedical and Translational Informatics Institute, using next-generation sequencing and electronic health record data to analyze outcomes in cancer and other diseases. She now works at Ultragenyx Pharmaceutical, focusing on preclinical research using bioinformatics and chemoinformatics on rare genetic diseases.

Thank you to my family, my teachers, and my mentors.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

Drawing inspiration from Charles Darwin's theory of natural evolution, *genetic algorithms* are among the most fascinating techniques for solving search, optimization, and learning problems. They can often prove successful where traditional algorithms fail to provide adequate results within a reasonable timeframe.

This book will take you on a journey to mastering this extremely powerful, yet simple, approach, and applying it to a wide variety of tasks, culminating in AI applications.

Using this book, you will gain an understanding of genetic algorithms, how they work, and when to use them. In addition, the book will provide you with hands-on experience of applying genetic algorithms to various domains using the popular Python programming language.

Who this book is for

This book was written to help software developers, data scientists, and AI enthusiasts interested in harnessing genetic algorithms to carry out tasks involving learning, searching, and optimization in their applications, as well as enhancing the performance and accuracy of their existing intelligent applications.

This book is also intended for anyone who is tasked with real-life, hard-to-solve problems where traditional algorithms are not useful, or fail to provide adequate results within a practical amount of time. The book demonstrates how genetic algorithms can be used as a powerful, yet simple, approach to solving a variety of complex problems.

What this book covers

`Chapter 1`, *An Introduction to Genetic Algorithms*, introduces genetic algorithms, their underlying theory, and their basic principles of operation. You will then explore the differences between genetic algorithms and traditional methods, and learn about the best use cases for genetic algorithms.

`Chapter 2`, *Understanding the Key Components of Genetic Algorithms*, dives deeper into the key components and the implementation details of genetic algorithms. After outlining the basic genetic flow, you will learn about their different components and the various implementations for each component.

`Chapter 3`, *Using the DEAP Framework*, introduces DEAP—a powerful and flexible evolutionary computation framework capable of solving real-life problems using genetic algorithms. You will discover how to use this framework by writing a Python program that solves the OneMax problem—the 'Hello World' of genetic algorithms.

`Chapter 4`, *Combinatorial Optimization*, covers combinatorial optimization problems, such as the knapsack problem, the traveling salesman problem, and the vehicle routing problem, and how to write Python programs that solve them using genetic algorithms and the DEAP framework.

`Chapter 5`, *Constraint Satisfaction*, introduces constraint satisfaction problems, such as the N-Queen problem, the nurse scheduling problem, and the graph coloring problem, and explains how to write Python programs that solve them using genetic algorithms and the DEAP framework.

`Chapter 6`, *Optimizing Continuous Functions*, covers continuous optimization problems, and how they can be solved by means of genetic algorithms. The examples you will use include the optimization of the Eggholder function, Himmelblau's function, and Simionescu's function. Along the way, you will explore the concepts of niching, sharing, and constraint handling.

`Chapter 7`, *Enhancing Machine Learning Models Using Feature Selection*, talks about supervised machine learning models, and explains how genetic algorithms can be used to improve the performance of these models by selecting the best subset of features from the input data provided.

`Chapter 8`, *Hyperparameter Tuning of Machine Learning Models*, explains how genetic algorithms can be used to improve the performance of supervised machine learning models by tuning the hyperparameters of the models, either by applying a genetic algorithm-driven grid search, or by using a direct genetic search.

`Chapter 9`, *Architecture Optimization of Deep Learning Networks*, focuses on artificial neural networks, and discovers how genetic algorithms can be used to improve the performance of neural-based models by optimizing their network architecture. You will then learn how to combine network architecture optimization with hyperparameter tuning.

`Chapter 10`, *Reinforcement Learning with Genetic Algorithms*, covers reinforcement learning, and explains how genetic algorithms can be applied to reinforcement learning tasks while solving two benchmark environments—MountainCar and CartPole— from the OpenAI Gym toolkit.

`Chapter 11`, *Genetic Image Reconstruction*, experiments with the reconstruction of a well-known image using a set of semi-transparent polygons, orchestrated by genetic algorithms. Along the way, you will gain useful experience in image processing and the relevant Python libraries.

`Chapter 12`, *Other Evolutionary and Bio-Inspired Computation Techniques*, broadens your horizons and gets you acquainted with several other biologically inspired problem-solving techniques. Two of these methods—genetic programming and particle swarm optimization—will be demonstrated using DEAP-based Python programs.

To get the most out of this book

To get the most out of this book, you should have a working knowledge of the Python programming language, and basic knowledge of mathematics and computer science. An understanding of fundamental machine learning concepts will be beneficial, but not mandatory, as the book covers the necessary concepts in a nutshell.

To run the programming examples accompanying this book, you will need Python release 3.7 or newer, as well as several Python packages described throughout the book. A Python IDE (Integrated Development Environment), such as PyCharm or Visual Studio Code, is recommended but not required.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781838557744_ColorImages.pdf`.

Code in Action

Visit the following link to check out videos of the code being run:
`http://bit.ly/3azd7Sp`

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The `__init__()` method of the class creates the dataset."

A block of code is set as follows:

```
self.X, self.y = datasets.make_friedman1(n_samples=self.numSamples,
                                          n_features=self.numFeatures,
                                          noise=self.NOISE,
                                          random_state=self.randomSeed)
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
self.regressor = GradientBoostingRegressor(random_state=self.randomSeed)
```

Any command-line input or output is written as follows:

```
pip install deap
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Section 1: The Basics of Genetic Algorithms

In this section, you will be introduced to the key concepts of genetic algorithms and how they can be used.

This section comprises the following chapters:

- `Chapter 1`, *An Introduction to Genetic Algorithms*
- `Chapter 2`, *Understanding the Key Components of Genetic Algorithms*

1
An Introduction to Genetic Algorithms

Drawing its inspiration from Charles Darwin's theory of natural evolution, one of the most fascinating techniques for problem-solving is the algorithm family suitably named **evolutionary computation**. Within this family, the most prominent and widely used branch is known as **genetic algorithms**. This chapter is the beginning of your journey to mastering this extremely powerful, yet extremely simple, technique.

In this chapter, we will introduce **genetic algorithms** and their analogy to Darwinian evolution, and dive into their basic principles of operation as well as their underlying theory. We will then go over the differences between genetic algorithms and traditional ones and cover the advantages and limitations of genetic algorithms and their uses. We will conclude by reviewing the cases where the use of a genetic algorithm may prove beneficial.

In this introductory chapter, we will cover the following topics:

- What are genetic algorithms?
- The theory behind genetic algorithms
- Differences between genetic algorithms and traditional algorithms
- Advantages and limitations of genetic algorithms
- When to use genetic algorithms

What are genetic algorithms?

Genetic algorithms are a family of search algorithms inspired by the principles of evolution in nature. By imitating the process of natural selection and reproduction, genetic algorithms can produce high-quality solutions for various problems involving search, optimization, and learning. At the same time, their analogy to natural evolution allows genetic algorithms to overcome some of the hurdles that are encountered by traditional search and optimization algorithms, especially for problems with a large number of parameters and complex mathematical representations.

In the rest of this section, we will review the basic ideas of genetic algorithms, as well as their analogy to the evolutionary processes transpiring in nature.

Darwinian evolution

Genetic algorithms implement a simplified version of the Darwinian evolution that takes place in nature. The principles of the Darwinian evolution theory can be summarized using the following principles:

- The principle of **variation**: The traits (attributes) of individual specimens belonging to a population may vary. As a result, the specimens differ from each other to some degree; for example, in their behavior or appearance.
- The principle of **inheritance**: Some traits are consistently passed on from specimens to their offspring. As a result, offspring resemble their parents more than they resemble unrelated specimens.
- The principle of **selection**: Populations typically struggle for resources within their given environment. The specimens possessing traits that are better adapted to the environment will be more successful at surviving, and will also contribute more offspring to the next generation.

In other words, evolution maintains a population of individual specimens that vary from each other. Those who are better adapted to their environment have a greater chance of surviving, breeding, and passing their traits to the next generation. This way, as generations go by, species become more adapted to their environment and to the challenges presented to them.

An important enabler of evolution is **crossover** or **recombination** – where offspring are created with a mix of their parents' traits. Crossover helps in maintaining the diversity of the population and in bringing together the better traits over time. In addition, **mutations** – random variations in traits – can play a role in evolution by introducing changes that can result in a leap forward every once in a while.

The genetic algorithms analogy

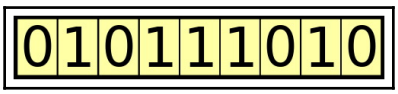
Genetic algorithms seek to find the optimal solution for a given problem. Whereas Darwinian evolution maintains a population of individual specimens, genetic algorithms maintain a population of candidate solutions, called **individuals**, for that given problem. These candidate solutions are iteratively evaluated and used to create a new generation of solutions. Those who are better at solving this problem have a greater chance of being selected and passing their qualities to the next generation of candidate solutions. This way, as generations go by, candidate solutions get better at solving the problem at hand.

In the following sections, we will describe the various components of genetic algorithms that enable this analogy for Darwinian evolution.

Genotype

In nature, breeding, reproduction, and mutation are facilitated via the **genotype** – a collection of genes that are grouped into chromosomes. If two specimens breed to create offspring, each chromosome of the offspring will carry a mix of genes from both parents.

Mimicking this concept, in the case of genetic algorithms, each individual is represented by a chromosome representing a collection of genes. For example, a chromosome can be expressed as a binary string, where each bit represents a single **gene**:

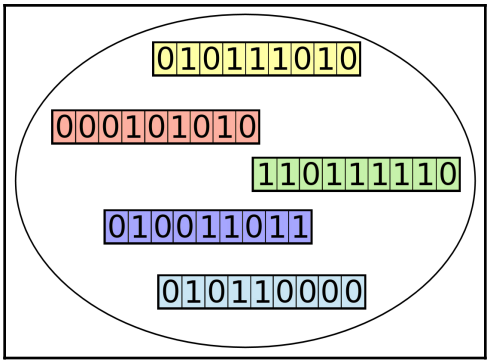


Simple binary-coded chromosome

The preceding image shows an example of one such binary-coded chromosome, representing one particular individual.

Population

At any point in time, genetic algorithms maintain a population of **individuals** – a collection of candidate solutions for the problem at hand. Since each individual is represented by some chromosome, this population of individuals can be seen as a collection of such chromosomes:



A population of individuals represented by binary-coded chromosomes

The population continually represents the current generation and evolves over time when the current generation is replaced by a new one.

Fitness function

At each iteration of the algorithm, the individuals are evaluated using a **fitness function** (also called the **target function**). This is the function we seek to optimize or the problem we attempt to solve.

Individuals who achieve a better fitness score represent better solutions and are more likely to be chosen to reproduce and be represented in the next generation. Over time, the quality of the solutions improves, the fitness values increase, and the process can stop once a solution is found with a satisfactory fitness value.

Selection

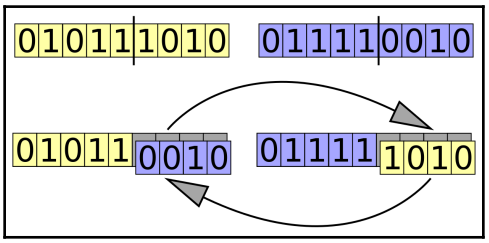
After calculating the fitness of every individual in the population, a selection process is used to determine which of the individuals in the population will get to reproduce and create the offspring that will form the next generation.

This selection process is based on the fitness score of the individuals. Those with higher score values are more likely to be chosen and pass their genetic material to the next generation.

Individuals with low fitness values can still be chosen, but with lower probability. This way, their genetic material is not completely excluded.

Crossover

To create a pair of new individuals, two parents are usually chosen from the current generation, and parts of their chromosomes are interchanged (crossed over) to create two new chromosomes representing the offspring. This operation is called crossover, or recombination:



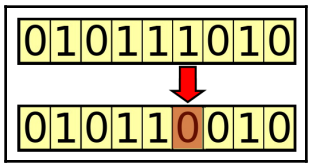
Crossover operation between two binary-coded chromosomes
Source: https://commons.wikimedia.org/wiki/File:Computational.science.Genetic.algorithm.Crossover.One.Point.svg.Image by Yearofthedragon. Licensed under Creative Commons CC BY-SA 3.0: https://creativecommons.org/licenses/by-sa/3.0/deed.en

The preceding image illustrates a simple crossover operation of creating two offspring from two parents.

Mutation

The purpose of the mutation operator is to periodically and randomly **refresh** the population, introduce new patterns into the chromosomes, and encourage search in uncharted areas of the solution space.

A mutation may manifest itself as a random change in a gene. Mutations are implemented as random changes to one or more of the chromosome values; for example, flipping a bit in a binary string:



Mutation operator applied to a binary-coded chromosome

The preceding image shows an example of the mutation operation.

Now, let's look at the theory behind genetic algorithms.

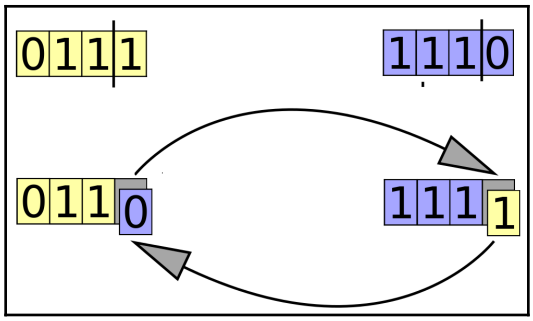
The theory behind genetic algorithms

The **building-block hypothesis** underlying genetic algorithms is that the optimal solution to the problem at hand is assembled of small building blocks, and as we bring more of these building blocks together, we get closer to this optimal solution.

Individuals in the population who contain some of the desired building blocks are identified by their superior scores. The repeated operations of selection and crossover result in the better individuals conveying these building blocks to the next generations, while possibly combining them with other successful building blocks. This creates genetic pressure, thus guiding the population toward having more and more individuals with the building blocks that form the optimal solution.

As a result, each generation is better than the previous one and contains more individuals that are closer to the optimal solution.

For example, if we have a population of four-digit binary strings and we want to find the string that has the largest possible sum of digits, the digit **1** appearing at any of the four string positions will be a good building block. As the algorithm progresses, it will identify solutions that have these building blocks and bring them together. Each generation will have more individuals with **1** values in various positions, ultimately resulting in the string **1111**, which combines all the desired building blocks. This is illustrated in the following image:



Demonstration of a crossover operation bringing the building blocks of the optimal solution together

The preceding image demonstrates how two individuals that are good solutions for this problem (each has three **1** values) create an offspring that is the best possible solution (four **1** bits, that is, the offspring on the right-hand side) when the crossover operation brings the desired building blocks of both parents together.

The schema theorem

A more formal expression of the building-block hypothesis is **Holland's schema theorem**, also called the **fundamental theorem of genetic algorithms**.

This theorem refers to schemata (plural of schema), which are patterns (or templates) that can be found within the chromosomes. Each schema represents a subset of chromosomes that have a certain similarity among them.

For example, if the set of chromosomes is represented by binary strings of length four, the schema *1*01* represents all those chromosomes that have a **1** in the leftmost position, **01** in the rightmost two positions, and either a **1** or a **0** in the second from left position, since the * represents a **wildcard** value.

For each schema, we can assign two measurements:

- **Order**: The number of digits that are fixed (not wildcards)
- **Defining length**: The distance between the two furthermost fixed digits

The following table provides several examples of four-digit binary schemata and their measurements:

Schema	Order	Defining Length
1101	4	3
1*01	3	3
*101	3	2
*1*1	2	2
**01	2	1
1***	1	0
****	0	0

Each chromosome in the population corresponds to multiple schemata in the same way that a given string matches regular expressions. The chromosome **1101**, for example, corresponds to each and every one of the schemata that appear in this table since it matches each of the patterns they represent. If this chromosome has a higher score, it is more likely to survive the selection operation, along with all the schemata it represents. As this chromosome gets crossed over with another, or as it gets mutated, some of the schemata will survive and others will disappear. The schemata of low order and short defining length are the ones more likely to survive.

Consequentially, the schema theorem states that the frequency of schemata of low order, short defining length, and above-average fitness increases exponentially in successive generations. In other words, the smaller, simpler building blocks that represent the attributes that make a solution better will become increasingly present in the population as the genetic algorithm progresses. We will look at the difference between genetic and traditional algorithms in the next section.

Differences from traditional algorithms

There are several important differences between genetic algorithms and traditional search and optimization algorithms, such as gradient-based algorithms.

The key characteristics of genetic algorithms distinguishing them from traditional algorithms are:

- Maintaining a population of solutions
- Using a genetic representation of the solutions
- Utilizing the outcome of a fitness function
- Exhibiting a probabilistic behavior

In the upcoming sections, we will describe these factors in greater detail.

Population-based

The genetic search is conducted over a population of candidate solutions (individuals) rather than a single candidate. At any point in the search, the algorithm retains a set of individuals that form the current generation. Each iteration of the genetic algorithm creates the next generation of individuals.

In contrast, most other search algorithms maintain a single solution and iteratively modify it in search of the best solution. The **gradient descent** algorithm, for example, iteratively moves the current solution in the direction of steepest descent, which is defined by the negative of the given function's gradient.

Genetic representation

Instead of operating directly on candidate solutions, genetic algorithms operate on their representations (or coding), often referred to as **chromosomes**. An example of a simple chromosome is a fixed-length binary string.

The chromosomes allow us to facilitate the genetic operations of crossover and mutation. Crossover is implemented by interchanging chromosome parts between two parents, while mutation is implemented by modifying parts of the chromosome.

A side effect of the use of genetic representation is decoupling the search from the original problem domain. Genetic algorithms are not aware of what the chromosomes represent and do not attempt to interpret them.

Fitness function

The fitness function represents the problem we would like to solve. The objective of genetic algorithms is to find the individuals that yield the highest score when this function is calculated for them.

Unlike many of the traditional search algorithms, genetic algorithms only consider the value that's obtained by the fitness function and do not rely on derivatives or any other information. This makes them suitable to handle functions that are hard or impossible to mathematically differentiate.

Probabilistic behavior

While many of the traditional algorithms are deterministic in nature, the rules that are used by genetic algorithms to advance from one generation to the next are probabilistic.

For example, when selecting the individuals that will be used to create the next generation, the probability of selecting a given individual increases with the individual's fitness, but there is still a random element in making that choice. Individuals with low score values can still be chosen as well, although with a lower probability.

Mutation is probability-driven as well, usually occurs with low likelihood, and makes changes at random location(s) in the chromosome.

The crossover operator can have a probabilistic element as well. In some variations of genetic algorithms, the crossover will only occur at a certain probability. If no crossover takes place, both parents are duplicated into the next generation without change.

Despite the probabilistic nature of this process, the genetic algorithm-based search is not random; instead, it uses the random aspect to direct the search toward areas in the search space where there is a better chance to improve the results. Now, let's look at the advantages of genetic algorithms.

Advantages of genetic algorithms

The unique characteristics of genetic algorithms that we discussed in the previous sections provide several advantages over traditional search algorithms.

The main advantages of genetic algorithms are as follows:

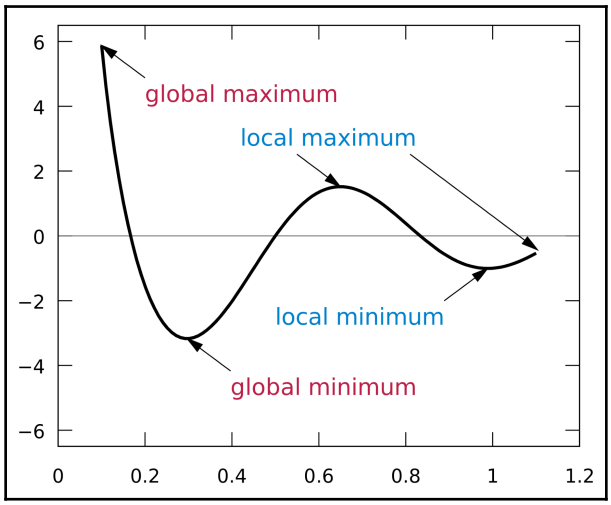
- Global optimization capability
- Handling problems with a complex mathematical representation
- Handling problems that lack mathematical representation
- Resilience to noise
- Support for parallelism and distributed processing
- Suitability for continuous learning

We will cover each of these in the upcoming sections.

Global optimization

In many cases, optimization problems have local maxima and minima points; these represent solutions that are better than those around them, but not the best overall.

The following image illustrates the differences between global and local maxima and minima points:



The global and local maxima and minima of a function
Source: https://commons.wikimedia.org/wiki/File:Computational.science.Genetic.algorithm.Crossover.One.Point.svg.
Image by KSmrq. Licensed under Creative Commons CC BY-SA 3.0: https://creativecommons.org/licenses/by-sa/3.0/

Most traditional search and optimization algorithms, and particularly those that are gradient-based, are prone to getting stuck in a local maximum rather than finding the global one. This is because, in the vicinity of a local maximum, any small change will degrade the score.

Genetic algorithms, on the other hand, are less sensitive to this phenomenon and are more likely to find the global maximum. This is due to the use of a population of candidate solutions rather than a single one, and the crossover and mutation operations that will, in many cases, result in candidate solutions that are distant from the previous ones. This holds true as long as we manage to maintain the diversity of the population and avoid **premature convergence**, as we will mention in the next section.

Handling complex problems

Since genetic algorithms only require the outcome of the fitness function for each individual and are not concerned with other aspects of the fitness function such as derivatives, they can be used for problems with complex mathematical representations or functions that are hard or impossible to differentiate.

Other complex cases where genetic algorithms excel include problems with a large number of parameters and problems with a mix of parameter types; for example, a combination of continuous and discrete parameters.

Handling a lack of mathematical representation

Genetic algorithms can be used for problems that lack mathematical representation altogether. One such case of particular interest is when the fitness score is based on human opinion. Imagine, for example, that we want to find the most attractive color palette to be used on a website. We can try different color combinations and ask users to rate the attractiveness of the site. We can apply genetic algorithms to search for the best scoring combination while using this opinion-based score as the fitness function outcome. The genetic algorithm will operate as usual, even though the fitness function lacks any mathematical representation and there is no way to calculate the score directly from a given color combination.

As we will see in the next chapter, genetic algorithms can even deal with cases where the score of each individual cannot be obtained, as long as we have a way to compare two individuals and determine which of them is better. An example of this is a machine learning algorithm that drives a car in a simulated race. A genetic algorithm-based search can optimize and tune the machine learning algorithm by having different versions of it compete against each other to determine which version is better.

Resilience to noise

Some problems present **noisy** behavior. This means that, even for similar input parameter values, the output value may be somewhat different every time it's measured. This can happen, for example, when the data that's being used is being read from sensor outputs, or in cases where the score is based on human opinion, as was discussed in the previous section.

While this kind of behavior can throw off many traditional search algorithms, genetic algorithms are generally resilient to it, thanks to the repetitive operation of reassembling and reevaluating the individuals.

Parallelism

Genetic algorithms lend themselves well to parallelization and distributed processing. Fitness is independently calculated for each individual, which means all the individuals in the population can be evaluated concurrently.

In addition, the operations of selection, crossover, and mutation can each be performed concurrently on individuals and pairs of individuals in the population.

This makes the approach of genetic algorithms a natural candidate for distributed as well as cloud-based implementation.

Continuous learning

In nature, evolution never stops. As the environmental conditions change, the population will adapt to them. Similarly, genetic algorithms can operate continuously in an ever-changing environment, and at any point in time, the best current solution can be fetched and used.

For this to be effective, the changes in the environment need to be slow in relation to the generation turnaround rate of the genetic algorithm-based search. Now that we have covered the advantages, let's look at the limitations of genetic algorithms.

Limitations of genetic algorithms

To get the most out of genetic algorithms, we need to be aware of their limitations and potential pitfalls.

The limitations of genetic algorithms are as follows:

- The need for special definitions
- The need for hyperparameter tuning
- Computationally-intensive operations
- The risk of premature convergence
- No guaranteed solution

We will cover each of these in the upcoming sections.

Special definitions

When applying genetic algorithms to a given problem, we need to create a suitable representation for them – define the fitness function and the chromosome structure, as well as the selection, crossover, and mutation operators that will work for this problem. This can often prove to be challenging and time-consuming.

Luckily, genetic algorithms have already been applied to countless different types of problems, and many of these definitions have been standardized. This book covers numerous types of real-life problems and the way they can be solved using genetic algorithms. Use this as guidance whenever you are challenged by a new problem.

Hyperparameter tuning

The behavior of genetic algorithms is controlled by a set of hyperparameters, such as the population size and mutation rate. When applying genetic algorithms to the problem at hand, there are no exact rules for making these choices.

However, this is the case for virtually all search and optimization algorithms. After going over the examples in this book and doing some experimentation of your own, you will be able to make sensible choices for these values.

Computationally-intensive

Operating on (potentially large) populations and the repetitive nature of genetic algorithms can be computationally intensive, as well as time consuming before a good result is reached.

These can be alleviated with a good choice of hyperparameters, implementing parallel processing, and in some cases, caching the intermediate results.

Premature convergence

If the fitness of one individual is much higher than the rest of the population, it may be duplicated enough that it takes over the entire population. This can lead to the genetic algorithm getting prematurely stuck in a local maximum, instead of finding the global one.

To prevent this from occurring, it is important to maintain the diversity of the population. Various ways to maintain diversity will be discussed in the next chapter.

No guaranteed solution

The use of genetic algorithms does not guarantee that the global maximum for the problem at hand will be found.

However, this is almost the case for any search and optimization algorithm, unless it is an analytical solution for a particular type of problem.

Generally, genetic algorithms, when used appropriately, are known to provide good solutions within a reasonable amount of time. Now, let's look at a few use cases of genetic algorithms.

Use cases of genetic algorithms

Based on the material we covered in the previous sections, genetic algorithms are best suited for the following types of problems:

- **Problems with complex mathematical representation**: Since genetic algorithms only require the outcome of the fitness function, they can be used for problems with target functions that are hard or impossible to differentiate, problems with a large number of parameters, and problems with a mix of parameter types.
- **Problems with no mathematical representation**: Genetic algorithms don't require a mathematical representation of the problem as long as a score value can be obtained or a method is available to compare two solutions.
- **Problems involving a noisy environment**: Genetic algorithms are resilient to problems where data may not be consistent, such as data originating from sensor output or from human-based scoring.
- **Problems involving an environment that changes over time**: Genetic algorithms can respond to slow changes in the environment by continuously creating new generations that will adapt to the changes that occur.

On the other hand, when a problem has a known and specialized way of being solved, using an existing traditional or analytic method is likely to be a more efficient choice.

Summary

In this chapter, we started by introducing genetic algorithms, their analogy to Darwinian evolution, and their basic principles of operation, including the use of population, genotype, the fitness function, and the genetic operators of selection, crossover, and mutation.

Then, we covered the theory underlying genetic algorithms by going over the building-block hypothesis and the schema theorem and illustrating how genetic algorithms work by bringing together superior, small building blocks to create the best solutions.

Next, we went over the differences between genetic algorithms and traditional ones, such as maintaining a population of solutions and using a genetic representation of the solutions.

We continued by covering the strengths of genetic algorithms, including their capacity for global optimization, handling problems with complex or non-existent mathematical representations and resilience to noise, followed by their weaknesses, including the need for special definitions and hyperparameter tuning, as well as the risk of premature convergence.

We concluded by going over the cases where the use of a genetic algorithm may prove beneficial, such as mathematically complex problems and optimization tasks in a noisy or ever-changing environment.

In the next chapter, we will delve deeper into the key components and the implementation details of genetic algorithms in preparation for the following chapters, where we will use them to code solutions for various types of problems.

Further reading

For more information on what we covered in this chapter, please refer to *Introduction to genetic algorithms*, from the book *Hands-On Artificial Intelligence for IoT* by *Amita Kapoor*, January 2019, available at `https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781788836067`.

2
Understanding the Key Components of Genetic Algorithms

In this chapter, we will dive deeper into the key components and the implementation details of genetic algorithms, in preparation for the following chapters, where we will use genetic algorithms to create solutions for various types of problems.

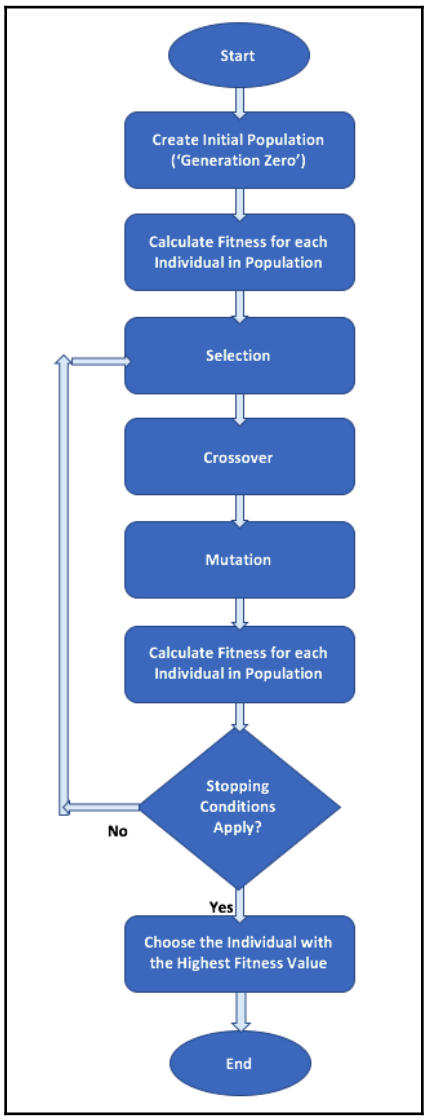
First, we will outline the basic flow of a genetic algorithm, then break it down into its different components while demonstrating various implementations of selection methods, crossover methods, and mutation methods. Next, we will look into real-coded genetic algorithms, which facilitate search in a continuous parameter space. This will be followed by an overview of the intriguing topics of elitism, niching, and sharing in genetic algorithms. Finally, we will study the art of solving problems using genetic algorithms.

At the end of this chapter, you will have achieved the following:

- Be familiar with the key components of genetic algorithms
- Understand the stages of the genetic algorithm flow
- Understand the genetic operators and become familiar with several of their variants
- Know the various options for stopping conditions
- Understand what modifications are needed to the basic genetic algorithm when applied to real numbers
- Understand the mechanism of elitism
- Understand the concepts and implementation of niching and sharing
- Know the choices you have to make when starting to work on a new problem

Basic flow of a genetic algorithm

The main stages of the basic genetic algorithm flow are shown in the following flowchart:



Basic flow of a genetic algorithm

These stages are described in detail in the following sections.

Creating the initial population

The initial population is a set of valid candidate solutions (individuals) chosen randomly. Since genetic algorithms use a chromosome to represent each individual, the initial population is actually a set of chromosomes. These chromosomes should conform to the chromosome format that we chose for the problem at hand, for example, binary strings of a certain length.

Calculating the fitness

The value of the fitness function is calculated for each individual. This is done once for the initial population, and then for every new generation after applying the genetic operators of selection, crossover, and mutation. As the fitness of each individual is independent of the others, this calculation can be done concurrently.

Since the selection stage that follows the fitness calculation usually considers individuals with higher fitness scores to be better solutions, genetic algorithms are naturally geared toward finding the maximum value(s) of the fitness function. If we have a problem where the minimum value is desired, the fitness calculation should inverse the original value, for example, through multiplying it by a value of (-1).

Applying selection, crossover, and mutation

Applying the genetic operators of selection, crossover, and mutation to the population results in the creation of a new generation that is based on better individuals than the current ones.

The **selection** operator is responsible for selecting individuals from the current population in a way that gives an advantage to better individuals. Examples of selection operators are given in the *Selection methods* section.

The **crossover** (or **recombination**) operator creates offspring from the selected individuals. This is usually done by taking two selected individuals at a time and interchanging parts of their chromosomes to create two new chromosomes representing the offspring. Examples of selection operators are given in the *Crossover methods* section.

The **mutation** operator can randomly introduce a change to one or more of the chromosome values (genes) of each newly created individual. The mutation usually occurs with a very low probability. Examples of mutation operators are given in the *Mutation methods* section.

Checking the stopping conditions

There can be multiple conditions to check against when determining whether the process can stop. The two most commonly used stopping conditions are:

- A maximum number of generations has been reached. This also serves to limit the runtime and computing resources consumed by the algorithm.
- There was no noticeable improvement over the last few generations. This can be implemented by storing the best fitness value achieved at every generation, and comparing the current best value to the one achieved a predefined number of generations ago. If the difference is smaller than a certain threshold, the algorithm can stop.

Other stopping conditions can be:

- A predetermined amount of time has elapsed since the process began.
- A certain cost or budget has been consumed, such as CPU time and/or memory.
- The best solution has taken over a portion of the population that is larger than a preset threshold.

To summarize, the genetic algorithm flow starts with a population of randomly generated candidate solutions (individuals), which are evaluated against the fitness function. The heart of the flow is a loop where the genetic operators of selection, crossover, and mutation are successively applied, followed by re-evaluation of the individuals. The loop continues until a stopping condition is encountered, upon which the best individual of the existing population is selected as our solution. Let's now look at the selection methods.

Selection methods

Selection is used at the beginning of each cycle of the genetic algorithm flow, to pick individuals from the current population that will be used as parents for the individuals of the next generation. The selection is probability-based, and the probability of an individual being picked is tied to its fitness value, in a way that gives an advantage to individuals with higher fitness values.

The following sections describe some of the commonly used selection methods and their characteristics.

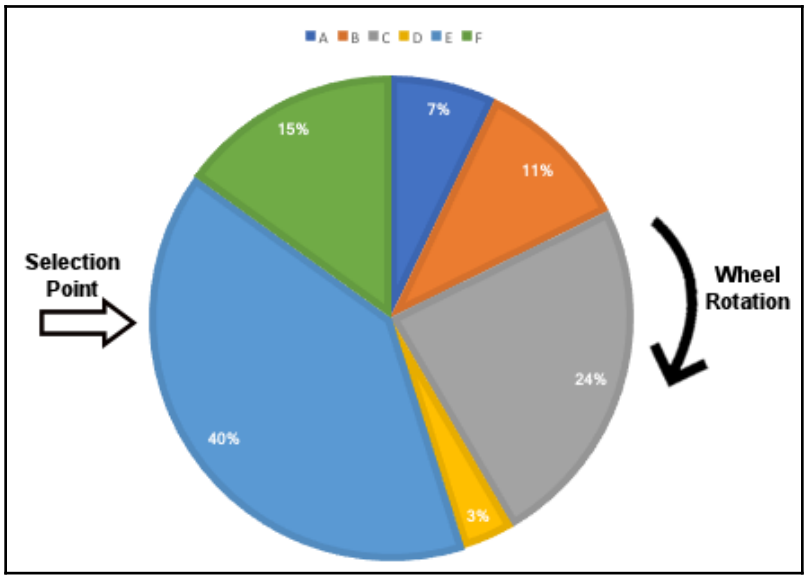
Roulette wheel selection

In the roulette wheel selection method, also known as **fitness proportionate selection** (**FPS**), the probability for selecting an individual is directly proportionate to its fitness value. This is comparable to using a roulette wheel in a casino and assigning each individual a portion of the wheel proportional to its fitness value. When the wheel is turned, the odds of each individual being selected are proportional to the size of the portion of the wheel that it occupies.

For example, suppose we have a population of six individuals with fitness values as shown in the following table. The relative portion of the roulette wheel dedicated to each individual is calculated based on these fitness values:

Individual	Fitness	Relative portion
A	8	7%
B	12	11%
C	27	24%
D	4	3%
E	45	40%
F	17	15%

The matching roulette wheel is depicted in the following diagram:

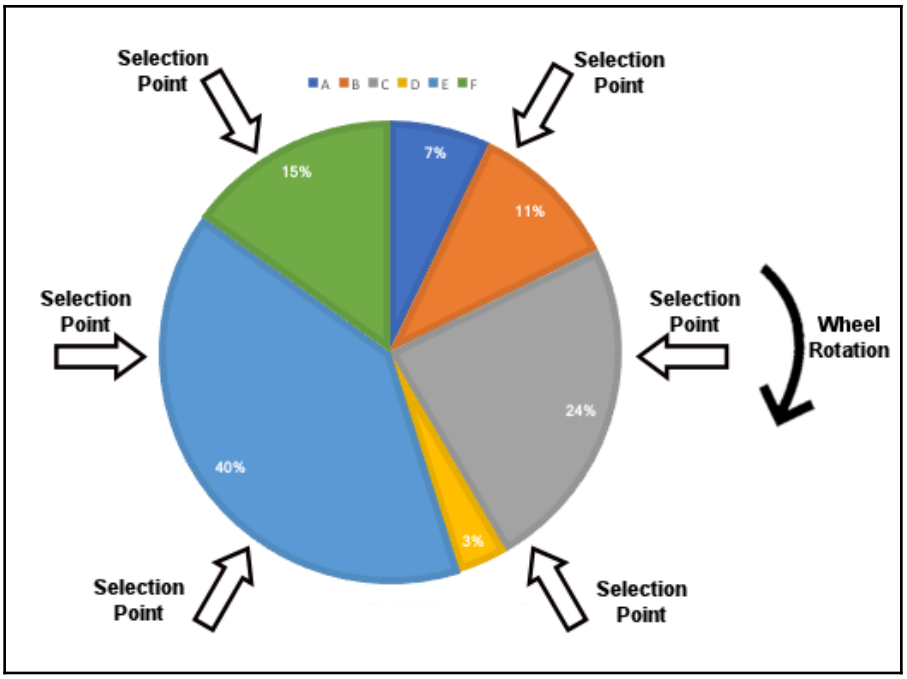


Roulette wheel selection example

Each time the wheel is turned, the selection point is used to choose a single individual from the entire population. The wheel is then turned again to select the next individual until we have enough individuals selected to fill the next generation. As a result, the same individual can be picked several times.

Stochastic universal sampling

Stochastic universal sampling (**SUS**) is a slightly modified version of the roulette wheel selection described previously. The same roulette wheel is used, with the same proportions, but instead of using a single selection point and turning the roulette wheel again and again until all needed individuals have been selected, we turn the wheel only once and use multiple selection points that are equally spaced around the wheel. This way, all the individuals are chosen at the same time, as depicted in the following diagram:



Stochastic universal sampling example

This selection method prevents individuals with particularly high fitness values from saturating the next generation by overly getting chosen over and over again. It thereby provides weaker individuals with a chance to be chosen, reducing the somewhat unfair nature of the original roulette wheel selection method.

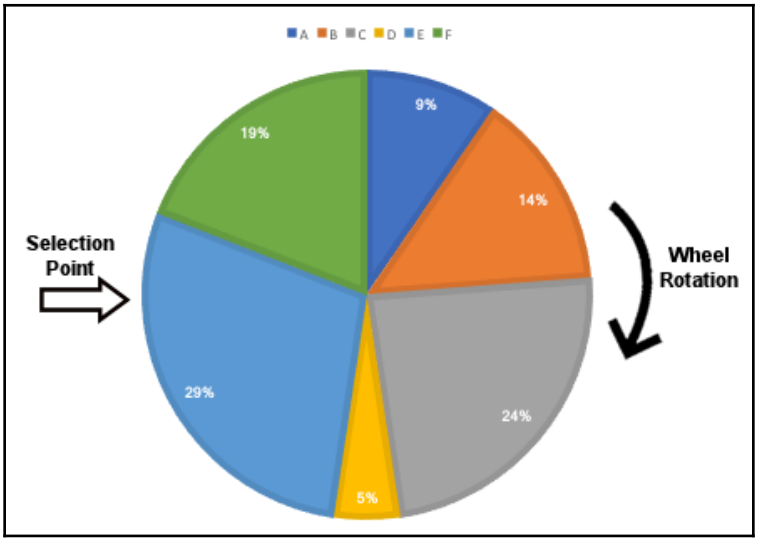
Rank-based selection

The rank-based selection method is similar to the roulette wheel selection, but instead of directly using the fitness values to calculate the probabilities for selecting each individual, the fitness is used just to sort the individuals. Once sorted, each individual is given a rank representing its position, and the roulette probabilities are calculated based on these ranks.

For example, let's take the same population of six individuals we previously used with the same fitness values. To that, we add the rank of each individual. As the population size in our example is six, the highest-ranking individual gets the rank value of 6, the next one gets the rank value of 5, and so on. The relative portion of the roulette wheel dedicated to each individual is now calculated based on these rank values instead of using the fitness values:

Individual	Fitness	Rank	Relative portion
A	8	2	9%
B	12	3	14%
C	27	5	24%
D	4	1	5%
E	45	6	29%
F	17	4	19%

The matching roulette wheel is depicted in the following diagram:



Rank-based selection example

Rank-based selection can be useful when a few individuals have much larger fitness values than all the rest. Using rank instead of raw fitness prevents these few individuals from taking over the entire population of the next generation, as ranking eliminates the large differences.

Another useful case is when all individuals have similar fitness values, where rank-based selection will spread them apart, giving a clearer advantage to the better ones even if the fitness differences are small.

Fitness scaling

While rank-based selection replaces each fitness value with the individual's rank, fitness scaling applies a scaling transformation to the raw fitness values and replaces them with the transformation's result. The transformation maps the raw fitness values into a desired range, as follows:

$$scaled\ fitness = \boldsymbol{a} \times (raw\ fitness) + \boldsymbol{b}$$

Here, $\boldsymbol{a}$ and $\boldsymbol{b}$ are constants that we can select to achieve the desired range of the scaled fitness.

For example, if we use the same values from the previous examples, the range of the raw fitness values is between 4 (lowest fitness value, individual D) and 45 (highest fitness value, individual E). Suppose we want to map the values into a new range, between 50 and 100. We can calculate the values of the $\boldsymbol{a}$ and $\boldsymbol{b}$ constants using the following equations, representing these two individuals:

- $50 = \boldsymbol{a} \times 4 + \boldsymbol{b}$ (lowest fitness value)
- $100 = \boldsymbol{a} \times 45 + \boldsymbol{b}$ (highest fitness value)

Solving this simple system of linear equations will yield the following scaling parameter values:

$$\boldsymbol{a} = 1.22,\ \boldsymbol{b} = 45.12$$

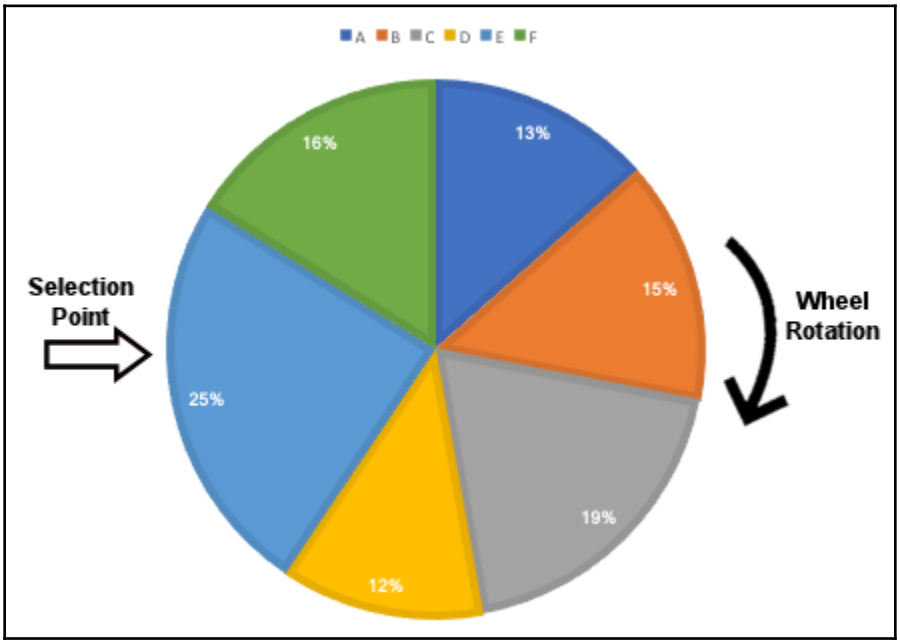
This means that the scaled fitness values can be calculated as follows:

$$scaled\ fitness = \boldsymbol{1.22} \times (raw\ fitness) + \boldsymbol{45.12}$$

After adding a new column to the table containing the scaled fitness values, we can see that the range is indeed between 0 and 50, as desired:

Individual	Fitness	Scaled fitness	Relative portion
A	8	55	13%
B	12	60	15%
C	27	78	19%
D	4	50	12%
E	45	100	25%
F	17	66	16%

The matching roulette wheel is depicted in the following diagram:



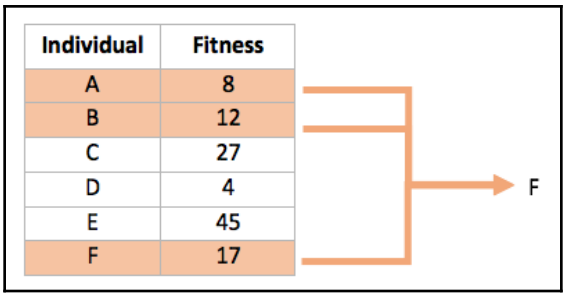
Roulette wheel selection example after fitness scaling

As the diagram illustrates, scaling the fitness values to the new range provided a much more moderate partition of the roulette wheel compared to the original partition. The best individual (with a scaled fitness value of 100) is now only twice more likely to be selected than the worst one (with a scaled fitness value of 50), instead of being more than 11 times more likely to be chosen when using the raw fitness values.

Tournament selection

In each round of the tournament selection method, two or more individuals are randomly picked from the population, and the one with the highest fitness score wins and gets selected.

For example, suppose we have the same six individuals and the same fitness values we used in the previous examples. The following diagram illustrates randomly selecting three of them (A, B, and F), then announcing F as the winner since it has the largest fitness value (17) among these three individuals:



Tournament selection example with a tournament size of three

The number of individuals participating at each tournament selection round (three in our example) is suitably called *tournament size*. The larger the tournament size, the higher the chances that the best individuals will participate in the tournaments, and the lesser the chances of low-scoring individuals winning a tournament and getting selected.

An interesting aspect of this selection method is that, as long as we can compare any two individuals and determine which of them is better, the actual value of the fitness function is not needed. We will now look at the crossover methods.

Crossover methods

The crossover operator, also referred to as recombination, corresponds to the crossover that takes place during sexual reproduction in biology, and is used to combine the genetic information of two individuals, serving as parents, to produce (usually two) offspring.

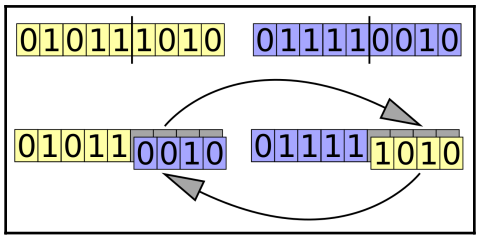
The crossover operator is typically applied with some (high) probability value. Whenever crossover is *not* applied, both parents are directly cloned into the next generation.

The following sections describe some of the commonly used crossover methods and their typical use cases. However, in certain situations, you may opt to use a problem-specific crossover method that will be more suitable for a particular case.

Single-point crossover

In the single-point crossover method, a location on the chromosomes of both parents is selected randomly. This location is referred to as the *crossover point*, or cut point. Genes to the right of that point are swapped between the two parent chromosomes. As a result, we get two offspring, where each of them carries some genetic information from both parents.

The following diagram demonstrates a single-point crossover operation conducted on a pair of binary chromosomes, with the crossover point located between the fifth and the sixth genes:



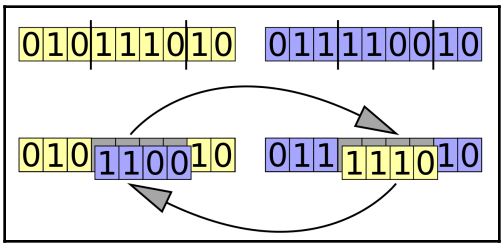
Single-point crossover example
Source: https://commons.wikimedia.org/wiki/File:Computational.science.Genetic.algorithm.Crossover.One.Point.svg. Image by Yearofthedragon. Licensed under Creative Commons CC BY-SA 3.0: https://creativecommons.org/licenses/by-sa/3.0/deed.en.

In the next section, we will cover extensions of this method, namely two-point and k-point crossover.

Two-point and k-point crossover

In the two-point crossover method, two crossover points on the chromosomes of both parents are selected randomly. The genes residing between these points are swapped between the two parent chromosomes.

The following diagram demonstrates a two-point crossover carried out on a pair of binary chromosomes, with the first crossover point located between the third and fourth genes, and the other between the seventh and eighth genes:

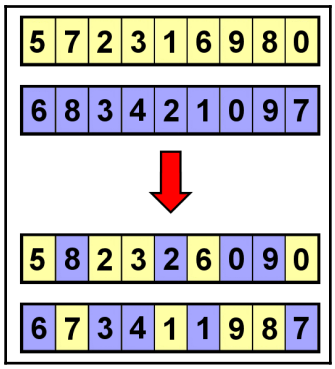


Two-point crossover example
Source: https://commons.wikimedia.org/wiki/File:Computational.science.Genetic.algorithm.Crossover.Two.Point.svg. Image by Yearofthedragon. Licensed under Creative Commons CC BY-SA 3.0: https://creativecommons.org/licenses/by-sa/3.0/deed.en.

The two-point crossover method can be implemented by carrying out two single-point crossovers, each with a different crossover point. A generalization of this method is the k-point crossover, where **k** represents a positive integer, and **k** crossover points are used.

Uniform crossover

In the uniform crossover method, each gene is independently determined by randomly choosing one of the parents. When the random distribution is 50%, each parent has the same chance of influencing the offspring, as illustrated in the following diagram:



Uniform crossover example

Note that, in this example, the second offspring was created by complementing the choices made for the first offspring, however, both offspring can also be created independently of each other.

In this example, we used integer-based chromosomes, but it would work similarly with binary ones.

Since this method does not exchange entire segments of the chromosome, it has greater potential for diversity in the resulting offspring.

Crossover for ordered lists

In the previous example, we saw the results of a crossover operation on two integer-based chromosomes. While each of the parents had every value between 0 and 9 appear exactly once, each of the resulting offspring had certain values appearing more than once (for example, 2 in the top offspring and 1 in the other), and other values were missing (such as 4 in the top offspring and 5 in the other).

In some tasks, however, integer-based chromosomes may represent indices of an ordered list. For example, suppose we have several cities, we know the distance between each, and we need to find the shortest possible route through all of them. This is known as the traveling salesman problem and will be covered in detail in one of the following chapters.

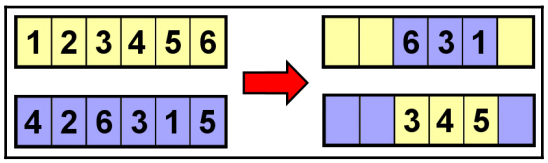
If, for instance, we have four cities, a convenient way to represent a possible solution for this problem would be a four-integer chromosome showing the order of visiting the cities, for example, (1,2,3,4) or (3,4,2,1). A chromosome having two of the same values, or missing one of the values such as (1,2,2,4), will not represent a valid solution.

For such cases, alternative crossover methods were devised to ensure that the offspring created will still be valid. One of these methods, *Ordered crossover*, is covered in the following section.

Ordered crossover

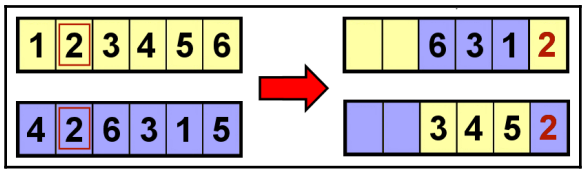
The **ordered crossover** (**OX1**) method strives to preserve the relative ordering of the parent's genes as much as possible. We will demonstrate it using chromosomes with a length of six.

The first step is a two-point crossover with random cut points, as shown in the following diagram (with the parents depicted on the left side):



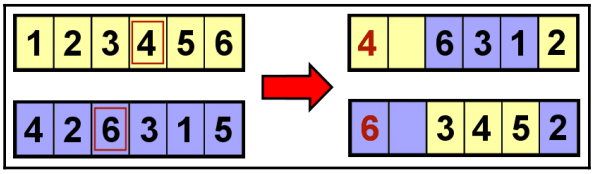
Ordered crossover example—step 1

We will now start filling in the rest of the genes in each offspring by going over all the parent's genes in their original order, starting after the second cut point. For the first parent, we find a **6**, but this is already present in the offspring, so we continue (with wrapping around) to **1**; this is already present too. The next in order is the **2**. Since **2** is not yet present in the offspring, we add it there, as shown in the figure below. For the second parent-offspring pair, we start with the parent's **5**, which is already present in the offspring, then move on to **4**, which is present as well, and end up with the **2**, which is not present yet and therefore gets added. This is shown in the diagram as well:



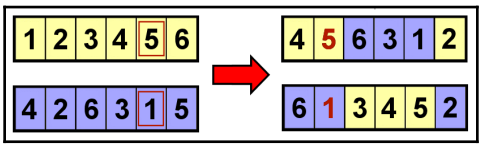
Ordered crossover example—step 2

For the top parent, we now continue to **3** (already present in the offspring), and then **4**, which gets added to the offspring. For the other parent, the next gene is **6**. Since it's not present in the matching offspring, it gets added to it. The results are illustrated in the following diagram:



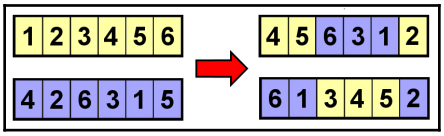
Ordered crossover example—step 3

We continue in a similar fashion with the next genes not yet present in the offspring, and fill in the last available spots, as depicted in the following diagram:



Ordered crossover example—step 4

This completes the process of producing two valid offspring chromosomes, as the following diagram demonstrates:



Ordered crossover example—step 5

There are numerous other methods to implement crossover, some of which we will encounter later in this book. However, thanks to the versatility of genetic algorithms, you can always come up with your own methods. We will now look at the mutation methods.

Mutation methods

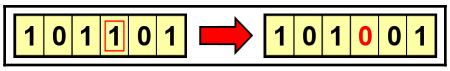
The mutation is the last genetic operator to be applied in the process of creating a new generation. The mutation operator is applied to the offspring that were created as a result of the selection and crossover operations.

The mutation is probability-based and usually occurs at a (very) low probability as it carries the risk of harming the performance of any individual it is applied to. In some versions of genetic algorithms, the mutation probability gradually increases as the generations advance to prevent stagnation and ensure diversity of the population. On the other hand, if the mutation rate is excessively increased, the genetic algorithm will turn into the equivalent of a random search.

The following sections describe some of the commonly used mutation methods and their typical use cases. However, remember that you can always choose to use your own problem-specific mutation method that you deem more suitable for your particular use case.

Flip bit mutation

When applying the flip bit mutation to a binary chromosome, one gene is randomly selected and its value is flipped (complemented), as shown in the following diagram:

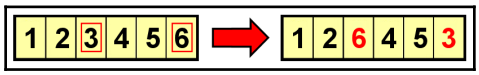


Flip bit mutation example

This can be extended to several random genes being flipped instead of just one.

Swap mutation

When applying the swap mutation to binary or integer-based chromosomes, two genes are randomly selected and their values are swapped, as shown in the following diagram:

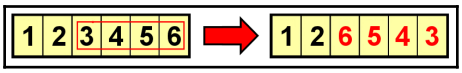


Swap mutation example

This mutation operation is suitable for the chromosomes of ordered lists, as the new chromosome still carries the same genes as the original one.

Inversion mutation

When applying the inversion mutation to binary or integer-based chromosomes, a random sequence of genes is selected and the order of the genes in that sequence is reversed, as depicted in the following diagram:

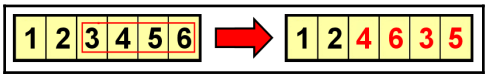


Inversion mutation example

Similar to the swap mutation, the inversion mutation operation is suitable for the chromosomes of ordered lists.

Scramble mutation

Another mutation operator suitable for the chromosomes of ordered lists is the scramble mutation. When applied, a random sequence of genes is selected and the order of the genes in that sequence is shuffled (or scrambled), illustrated as follows:



Scramble mutation example

In the next section, we will cover some other types of specialized operators created for real-coded genetic algorithms.

Real-coded genetic algorithms

So far, we have seen chromosomes that represented binary or integer parameters. Consequently, the genetic operators were suitable for working on these types of chromosomes. However, we often encounter problems where the solution space is continuous. In other words, the individuals are made up of real (floating-point) numbers.

Historically, genetic algorithms used binary strings to represent integers as well as real numbers, however, this was not ideal. The precision of a real number represented using a binary string is limited by the length of the string (number of bits). Since we need to determine this length in advance, we may end up with binary strings that are too short, resulting in insufficient precision, or are overly long.

Moreover, when a binary string is used to represent a number, the significance of each bit varies by its location—the most significant bit being on the left. This can cause imbalance related to schemas—the patterns occurring in the chromosomes. For example, the schema 1**** (representing all five-digit binary strings starting with 1) and the schema ****1 (representing all five-digit binary strings ending with 1) both have an order of 1 and a defining length of 0, however, the first one carries much more significance than the other.

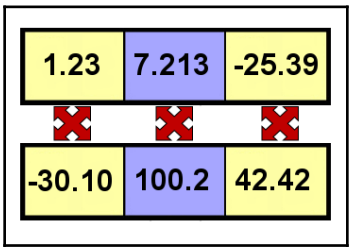
Instead of using binary strings, arrays of real-valued numbers were found to be a simpler and better approach. For example, if we have a problem involving three real-valued parameters, the chromosome will look like $[x_1, x_2, x_3]$, where x_1, x_2, x_3 represent real numbers, such as [1.23, 7.2134, -25.309] or [-30.10, 100.2, 42.424].

The various selection methods mentioned earlier in this chapter will work just the same for real-coded chromosomes as they only depend on the fitness of the individuals and not their representation.

However, the crossover and mutation methods covered so far will not be suitable for the real-coded chromosomes and so specialized ones need to be used. One important point to remember is that these crossover and mutation operations are applied separately for each dimension of the array that forms the real-coded chromosome. For example, if [1.23, 7.213, -25.39] and [-30.10, 100.2, 42.42] are parents selected for the crossover operation, the crossover will be separately done for the following pairs:

- 1.23 and -30.10 (first dimension)
- 7.213 and 100.2 (second dimension)
- -25.39 and 42.42 (third dimension)

This is illustrated in the following diagram:



Real-coded chromosomes crossover example

Similarly, the mutation operator, when applied to a real-coded chromosome, will apply separately to each dimension.

Several of these real-coded crossover and mutation methods are described in the following sections. Later, in `Chapter 6`, *Optimizing Continuous Functions*, we will get to see them in action.

Blend crossover

In the **blend crossover** (also known as **BLX**), each offspring is randomly selected from the following interval created by its parents:

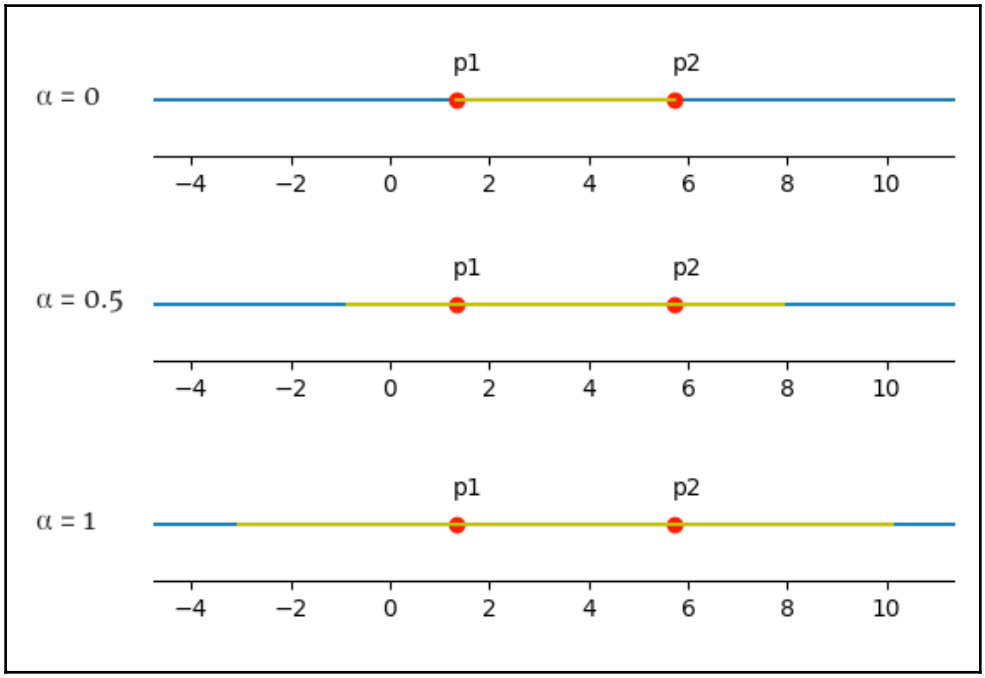
$$[parent_1 - \alpha(parent_2 - parent_1),\ \ parent_2 + \alpha(parent_2 - parent_1)]$$

The parameter α is a constant, whose value lies between 0 and 1. With larger values of α, the interval gets wider.

For example, if the parents' values are 1.33 and 5.72, the following will be the case:

- An α value of 0 will yield the interval [1.33, 5.72] (similar to the interval between the parents)
- An α value of 0.5 will yield the interval [-0.865, 7.915] (twice as wide as the interval between the parents)
- An α value of 1.0 will yield the interval [-3.06, 10.11] (three times wider than the interval between the parents)

These examples are illustrated in the following diagram where the parents are labeled by **p1** and **p2**, and the crossover interval is colored yellow:



Blend crossover example

When using this crossover method, the α value is commonly set to 0.5.

Simulated binary crossover

The idea behind the **simulated binary crossover** (**SBX**) is to imitate the properties of the single-point crossover that is commonly used with binary-coded chromosomes. One of these properties is that the average of the parents' values is equal to that of the offsprings' values.

When applying SBX, the two offspring are created from the two parents using the following formula:

$$offspring_1 = \frac{1}{2}[(1+\beta)parent_1 + (1-\beta)parent_2]$$
$$offspring_2 = \frac{1}{2}[(1-\beta)parent_1 + (1+\beta)parent_2]$$

Here, β is a random number referred to as the *spread factor*.

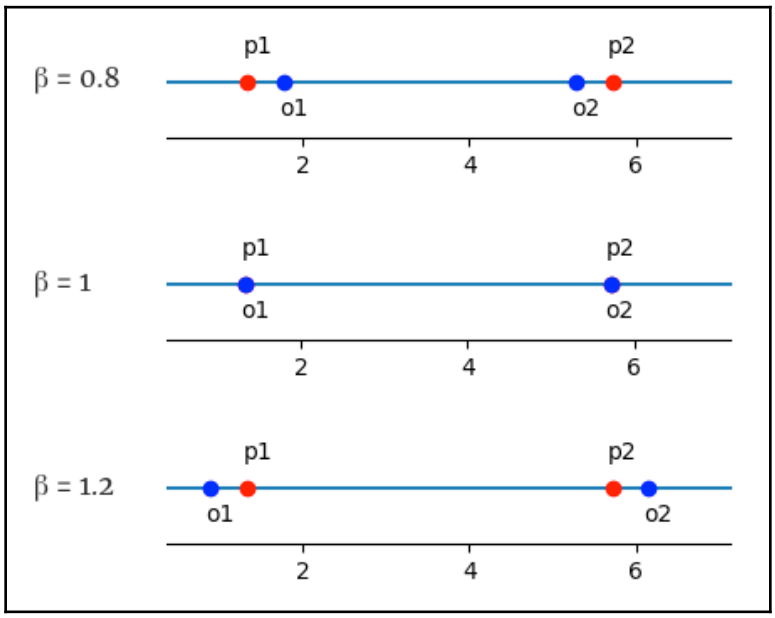
This formula has the following notable properties:

- The average of the two offspring is equal to that of the parents, regardless of the value of β.
- When the β value is 1, the offspring are duplicates of the parents.
- When the β value is smaller than 1, the offspring are closer to each other than the parents were.
- When the β value is larger than 1, the offspring are farther apart from each other than the parents were.

For example, if the parents' values are 1.33 and 5.72, the following will be the case:

- A β value of 0.8 will yield the offspring 1.769 and 5.281
- A β value of 1.0 will yield the offspring 1.33 and 5.72
- A β value of 1.2 will yield the offspring 0.891 and 6.159

These cases are illustrated in the following diagram where the parents are labeled by **p1** and **p2**, and the offspring by **o1** and **o2**:



Simulated binary crossover example

In each of the preceding cases, the average value of the two offspring is 3.525, which is equal to the average value of the two parents.

Another property of the binary single-point crossover that we would like to preserve is the similarity between offspring and parents. This translates to the random distribution of the β value. The probability of β should be much higher for values around 1, where the offspring are similar to the parents. To achieve that, the β value is calculated using another random value, denoted by *u*, that is uniformly distributed over the interval [0, 1]. Once the value of *u* is picked, β is calculated as follows:

If u ≤ 0.5: $\beta = (2u)^{\frac{1}{\eta+1}}$

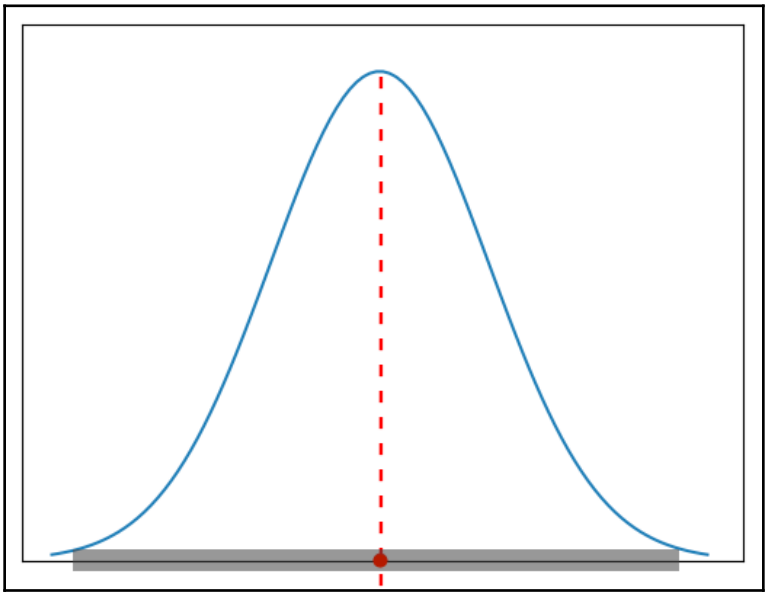
Otherwise: $\beta = \left[\frac{1}{2(1-u)}\right]^{\frac{1}{\eta+1}}$

The parameter η (eta) used in these formulas is a constant representing the **distribution index**. With larger values of η, offspring will tend to be more similar to their parents. A common value of η is 10 or 20.

Real mutation

One option for applying mutation in real-coded genetic algorithms is to replace any real value with a brand new one, generated randomly. However, this can result in a mutated individual that has no relationship to the original individual.

Another approach is to generate a random real number that resides in the vicinity of the original individual. An example of such a method is the **normally distributed (or Gaussian) mutation**: a random number is generated using a normal distribution with a mean value of zero and some predetermined standard deviation as shown in the following plot:



Gaussian mutation distribution example

In the next two sections, we will go over a couple of advanced topics, namely elitism and niching.

Understanding elitism

While the average fitness of the genetic algorithm population generally increases as generations go by, it is possible at any point that the best individual(s) of the current generation will be lost. This is due to the selection, crossover, and mutation operators altering the individuals in the process of creating the next generation. In many cases, the loss is temporary as these individuals (or better individuals) will be re-introduced into the population in a future generation.

However, if we want to guarantee that the best individual(s) always make it to the next generation, we can apply the optional elitism strategy. This means that the top *n* individuals (*n* being a small, predefined parameter) are duplicated into the next generation before we fill the rest of the available spots with offspring that are created using selection, crossover, and mutation. The elite individuals that were duplicated are still eligible for the selection process so they can still be used as the parents of new individuals.

Elitism can sometimes have a significant positive impact on the algorithm's performance as it avoids the potential time waste needed for re-discovering good solutions that were lost in the genetic flow.

Another interesting way to enhance the results of genetic algorithms is the use of niching, as described in the next section.

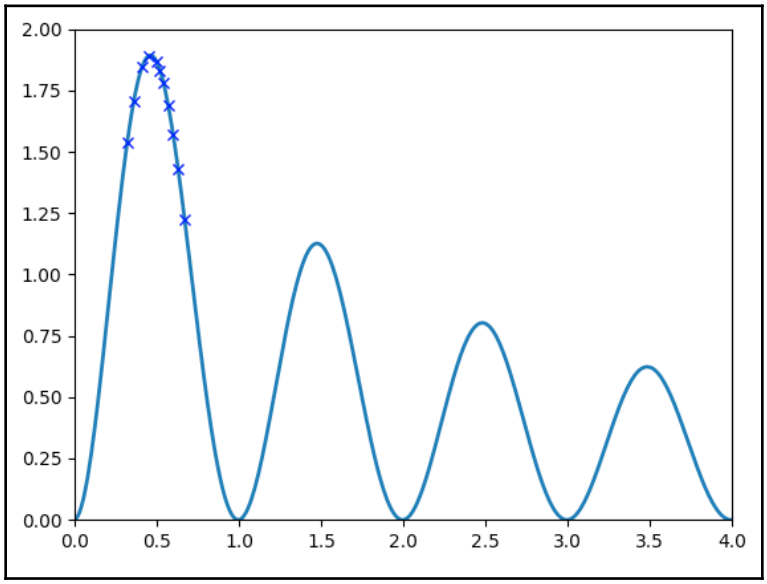
Niching and sharing

In nature, any environment is further divided into multiple sub-environments, or niches, populated by various species taking advantage of the unique resources available in each niche, such as food and shelter. For example, a forest environment is comprised of the treetops, the shrubs, the forest floor, the tree roots, and so on; each of these accommodating different species who are specialized for living in that niche and takes advantage of its resources.

When several different species coexist in the same niche, they all compete over the same resources, and a tendency is created to search for new, unpopulated niches and populate them.

In the realm of genetic algorithms, this niching phenomenon can be used to maintain the diversity of the population as well as for finding several optimal solutions, each considered a niche.

For example, suppose our genetic algorithm seeks to maximize a fitness function having several peaks of varying heights, such as the one in the following plot:

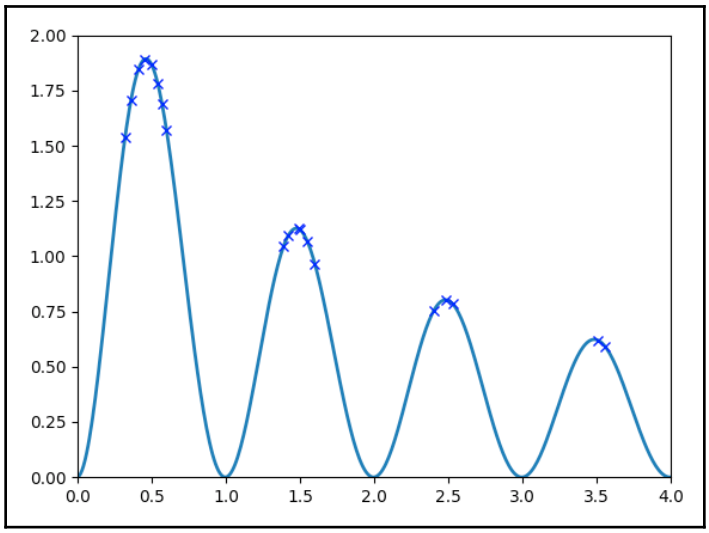


Expected genetic algorithm results without niching

As the tendency of the genetic algorithm is to find the global maximum, we expect, after a while, to see most of the population concentrating around the top peak. This is indicated in the figure by the locations of the × marks on the function graph, representing individuals in the current generation.

However, there are implementations where, in addition to the global maximum, we would like to find some (or all) of the other peaks. To make this happen, we could think of each peak as a niche, offering resources in the amount proportional to its height. We then find a way to share (or divide) these resources among the individuals occupying it. This will ideally drive the population to be distributed accordingly, with the top peak attracting the most individuals as it offers the most reward, and the other peaks populated with decreasing portions of the population as they offer smaller amounts of reward.

This ideal situation is depicted in the following figure:



Ideal genetic algorithm results with niching

The challenge now is to implement this sharing mechanism. One option to accomplish sharing is to divide the raw fitness value of each individual with (some function of) the combined distances from all the other individuals. Another option would be to divide the raw fitness of each individual with the number of other individuals within a certain radius around it.

Serial niching versus parallel niching

Unfortunately, the niching concept as described previously can prove hard to implement as it increases the complexity of the fitness calculation. In practice, it will also require the population size to be the original one multiplied by the number of the expected peaks (which is generally unknown).

One way to overcome these issues is to find the peaks one at a time (serial niching) instead of attempting to find all of them at the same time (parallel niching). To implement serial niching, we use the genetic algorithm as usual and find the best solution. We then update the fitness function so that the area of the maximum point that was found is flattened, and repeat the process of the genetic algorithm.

Ideally, we will now find the next best peak, as the original peak is no longer present. We can repeat this process iteratively and find the next best peak at each iteration.

The art of solving problems using genetic algorithms

Genetic algorithms provide us with a powerful and versatile tool that can be used to solve a wide array of problems and tasks. When we set to work on a new problem, we need to customize the tool and match it to that problem. This is done by making several choices, as described in the following paragraphs.

First, we need to determine the **fitness function**. This is how each individual will be evaluated, where larger values represent better individuals. The function does not have to be mathematical. It can be represented by an algorithm, or a call to an external service, or even a result of a game played, to list a few options. We just need a way to programmatically retrieve the fitness value for any given proposed solution (individual).

Next, we need to choose an appropriate **chromosome encoding**. This is based on the parameters we send to the fitness function. So far, we have seen binary, integer, an ordered list, and real-coded examples. However, for some problems, we may need a mix of parameter types, or may even decide to create our own custom chromosome encoding.

Next, we need to pick a **selection** method. Most selection methods will work for any kind of chromosome type. If the fitness function is not directly accessible, but we still have a way to tell which of several candidate solutions is the best, we can consider utilizing the tournament selection method.

As we have seen in the preceding sections, the choice of **crossover** and **mutation** operators will be linked to the chromosome encoding of the individuals. Binary-coded chromosomes will have different crossover and mutation schemes than those that fit real-coded problems. Similar to the choice of chromosome encoding, here, too, you can design your own methods of crossover and mutation to fit your unique use case.

Lastly, there are the hyperparameters of the algorithm. The most common parameter values we need to set are as follows:

- Population size
- Crossover rate
- Mutation rate
- Max number of generations
- Other stopping condition(s)
- Elitism (used or not; what size)

For these parameters, we can choose what we deem as reasonable values and then tweak them, similar to how hyperparameters are dealt with in almost any other optimization and learning algorithm.

If making all these choices appears to be an overwhelming task, don't fret! In the chapters that follow, we will be iterating the process of making these choices time and again for the various types of problems we will tackle. After reading this book, you will be able to look at new problems and make your own wise choices.

Summary

In this chapter, you were first introduced to the basic flow of the genetic algorithm. We then went over the key components of the flow, which included creating the population, calculating the fitness function, applying the genetic operators, and checking for stopping conditions.

Next, we went over various methods of selection, including roulette wheel selection, stochastic universal sampling, rank-based selection, fitness scaling, and tournament selection, and demonstrated the differences between them.

We continued by reviewing several methods of crossover, such as single-point, two-point, and k-point crossover, as well as ordered crossover and partially matched crossover.

You were then introduced to a number of mutation methods, such as flip bit mutation, followed by the swap, inversion, and the scramble mutation.

Real-coded genetic algorithms were presented next, with their specialized chromosome encoding as well as their custom genetic operators of crossover and mutation.

This was followed by an introduction to the concepts of elitism, as well as niching and sharing as used in genetic algorithms.

In the last part of the chapter, you were presented with the various choices that need to be made when approaching a problem to be solved using genetic algorithms; a procedure that will be repeated time and again throughout the book.

In the next chapter, the real fun begins—coding with Python! You will be introduced to DEAP—an evolutionary computation framework that can be used as a powerful tool for applying genetic algorithms to a wide array of tasks. DEAP will be used in the rest of the book as we develop Python programs that tackle numerous different challenges.

Further reading

For more information, please refer to *Chapter 8*, *Genetic Algorithms*, from the book *Artificial Intelligence with Python* by Prateek Joshi, January 2017, available at `https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781786464392/8`.

2
Section 2: Solving Problems with Genetic Algorithms

This section focuses on solving a series of real-world problems of various types using genetic algorithms with Python.

This section comprises the following chapters:

- `Chapter 3`, *Using the DEAP Framework*
- `Chapter 4`, *Combinatorial Optimization*
- `Chapter 5`, *Constraint Satisfaction*
- `Chapter 6`, *Optimizing Continuous Functions*

3
Using the DEAP Framework

In this chapter—as promised—the real fun begins! You will be introduced to DEAP—a powerful and flexible **evolutionary computation framework** capable of solving real-life problems using genetic algorithms. After a brief introduction, you will get acquainted with two of its main modules—the `creator` and the `toolbox`—and learn how to create the various components needed for the genetic algorithm flow. We will then write a Python program that solves the OneMax problem—the *Hello World* of genetic algorithms—using the DEAP framework. This will be followed by a more concise version of the same program—taking advantage of the built-in algorithms of the framework. And we saved the best for the last part of this chapter, where we will be experimenting with various settings of the genetic algorithm we created and discover the effects of our modifications.

By the end of this chapter you will:

- Be familiar with the DEAP framework and its genetic algorithms modules
- Understand the concepts of `creator` and `toolbox` in the DEAP framework
- Be able to translate a simple problem into a genetic algorithm representation
- Be able to create a genetic algorithm solution using the DEAP framework
- Understand how to use the DEAP framework's built-in algorithms to produce concise code
- Solve the OneMax problem using a genetic algorithm coded with the DEAP framework
- Be able to experiment with various settings of the genetic algorithm and interpret the differences in the results

Technical requirements

Recent versions of DEAP can be used with either Python 2 or 3. In this book, we will be using Python 3.7. Python can be downloaded from the Python Software Foundation at this link: `https://www.python.org/downloads/`. Additional useful instructions can be found here: `https://realpython.com/installing-python/`.

The recommended ways to install DEAP are using `easy_install` or `pip`, for example:

```
pip install deap
```

For more information, check this DEAP documentation link: `https://deap.readthedocs.io/en/master/installation.html`. Conda install is available at this link: `https://anaconda.org/conda-forge/deap`.

In addition, we will be using various Python packages throughout the book.

For this chapter you will need the following packages:

- NumPy: `http://www.numpy.org/`
- Matplotlib: `https://matplotlib.org/`
- Seaborn: `https://seaborn.pydata.org/`

We are now ready to use DEAP. The framework's most useful tools and utilities will be covered in the next two sections. But first, we will get acquainted with DEAP and understand why we chose this framework for working with genetic algorithms.

The programs used in this chapter can be found in the book's GitHub repository at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter03`.

Check out the following video to see the Code in Action:
`http://bit.ly/3aTym1p`

Introduction to DEAP

As we have seen in the previous chapters, the basic ideas behind genetic algorithms and the genetic flow are relatively simple, and so are many of the genetic operators. Therefore, developing a program from scratch that implements a genetic algorithm to solve a particular problem is entirely feasible.

However, as is often the case when developing software, using a tried-and-true dedicated library or framework can make our life easier. It helps us create solutions faster and with fewer bugs, and give us many options to choose from (and experiment with) right out of the box, without the need to *re-invent the wheel*.

Numerous Python frameworks have been created for working with genetic algorithms—GAFT, Pyevolve, and PyGMO, to mention a few. After looking into several options, we chose to use the DEAP framework for this book, thanks to its ease of use and a large selection of features, as well as its extensibility and ample documentation.

DEAP (short for **Distributed Evolutionary Algorithms in Python**) is a Python framework that supports the rapid development of solutions using genetic algorithms, as well as other evolutionary computation techniques. DEAP offers various data structures and tools that prove essential when implementing a wide range of genetic algorithm-based solutions.

DEAP was developed at the Canadian Laval University in 2009 and is available under the GNU **Lesser General Public License** (**LGPL**). The source code for DEAP is available at `https://github.com/DEAP/deap` and the documentation can be found at `https://deap.readthedocs.io/en/master/`.

Using the creator module

The first powerful tool provided by the DEAP framework is the `creator` module. The `creator` module is used as a meta-factory, and it enables us to extend existing classes by augmenting them with new attributes.

For example, suppose we have a class called `Employee`. Using the `creator` tool, we can extend the `Employee` class by creating a `Developer` class as follows:

```
from deap import creator
creator.create("Developer", Employee, position = "Developer",
programmingLanguages = set)
```

The first argument passed to the `create()` function is the desired name for the new class. The second argument is the existing class to be extended. Then, each additional argument defines an attribute for the new class. If the argument is assigned a class (such as a `dict` or a `set`), it will be added to the new class as an instant attribute initialized in the constructor. If the argument is not a class (for example, it is a literal), it will then be added as a class (`static`) attribute.

Consequently, the created `Developer` class will extend the `Employee` class, and will have a class attribute, `position`, set to `Developer`, and an instance attribute, `programmingLanguages` of type `set`, which is initialized in the constructor. So, effectively, the new class is equivalent to the following:

```
class Developer(Employee):
    position = "Developer"

    def __init__(self):
        self.programmingLanguages = set()
```

This new class exists within the `creator` module, and therefore needs to be referenced as `creator.Developer`.

Extending the `numpy.ndarray` class is a special case that will be discussed later in this book.

When using DEAP, the `creator` module usually serves to create the `Fitness` class as well as the `Individual` class to be used by the genetic algorithm, as we will see next.

Creating the Fitness class

When using DEAP, fitness values are encapsulated within a `Fitness` class. DEAP enables fitness to be combined into several components (also called objectives), each having its own weight. The combination of these weights defines the behavior or strategy of the fitness for the given problem.

Defining the fitness strategy

To help define this strategy, DEAP comes with the abstract `base.Fitness` class, which contains a `weights` tuple. This tuple needs to be assigned with values in order to define the strategy and make the class usable. This is done by extending the base `Fitness` class using the `creator`, in a similar manner to what we did with the preceding `Developer` class:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

This will yield a `creator.FitnessMax` class extending the `base.Fitness` class, with the `weights` class attribute initialized to a value of `(1.0,)`.

Note the trailing comma in the `weights` definition when a single weight is defined. The comma is required because `weights` is a tuple.

The strategy of this `FitnessMax` class is to maximize the fitness values of the single-objective solutions during the course of the genetic algorithm. Conversely, if we have a single-objective problem where we need to find a solution that minimizes the fitness value, we can use the following definition to create the appropriate minimizing strategy:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

We can also define a class with a strategy for optimizing more than one objective, and with varying degrees of importance:

```
creator.create("FitnessCompound", base.Fitness, weights=(1.0, 0.2, -0.5))
```

This will yield a `creator.FitnessCompound` class that will utilize three different fitness components. The first will be given a weight of `1.0`, the second `0.2`, and the third `-0.5`. This will tend to maximize the first and the second components (or objectives) and minimize the third, while in terms of importance, the first component has the most importance, followed by the third component and then the second one.

Storing the fitness values

While the `weights` tuple defines the fitness strategy, a matching tuple, called `values`, is used to contain the actual fitness values within the `base.Fitness` class. These values are obtained from a separately defined function, typically called `evaluate()`, as will be described later in this chapter. Just like the `weights` tuple, the `values` tuple contains one value for each fitness component (objective).

A third tuple, `wvalues`, contains the weighted values obtained by multiplying each component of the `values` tuple with its matching component of the `weights` tuple. Whenever the fitness values of an instance are set, the weighted values are calculated and inserted into `wvalues`. These are used internally for comparison operations between individuals.

The weighted fitnesses may be lexicographically compared using the following operators:

```
>, <, >=, <=, ==, !=
```

Once the `Fitness` class is created, we use it in the definition of the `Individual` class, as shown in the next subsection.

Creating the Individual class

The second common use of the `creator` tool in DEAP is defining the individuals that form the population for the genetic algorithm. As we saw in the previous chapters, the individuals in genetic algorithms are represented using a chromosome that can be manipulated by genetic operators. In DEAP, the `Individual` class is created by extending a base class that represents the chromosome. In addition, each individual instance in DEAP needs to contain its fitness function as an attribute.

To fulfill these two requirements, we utilize the `creator` to create the `creator.Individual` class, as shown in this example:

```
creator.create("Individual", list, fitness=creator.FitnessMax)
```

This line provides the following two effects:

- The created `Individual` class extends the Python `list` class. This means that the chromosome used is of the `list` type.
- Each instance of this `Individual` class will have an attribute called `fitness`, of the `FitnessMax` class we previously created.

We will learn to use the `Toolbox` class in the next section.

Using the Toolbox class

The second mechanism offered by the DEAP framework is the `base.Toolbox` class. The `Toolbox` is used as a container for functions (or operators), and enables us to create new operators by aliasing and customizing existing functions.

For example, suppose we have a function, `sumOfTwo()`, defined as follows:

```
def sumOfTwo(a, b):
    return a + b
```

Using `toolbox`, we can now create a new operator, `incrementByFive()`, which customizes the `sumOfTwo()` function as follows:

```
from deap import base
toolbox= base.Toolbox()
toolbox.register("incrementByFive", sumOfTwo, b=5)
```

The first argument passed to the `register()` toolbox function is the desired name (or alias) for the new operator. The second argument is the existing function to be customized. Then, each additional (optional) argument is automatically passed to the customized function whenever we call the new operator. For example, look at this definition:

```
toolbox.incrementByFive(10)
```

Calling the preceding function is equivalent to calling this:

```
sumOfTwo(10, 5)
```

This is because the `b` argument has been fixed to a value of `5` by the definition of the `incrementByFive` operator.

Creating genetic operators

In many cases, the `Toolbox` class is used to customize existing functions from the `tools` module. The tools module contains numerous handy functions related to the genetic operations of selection, crossover, and mutation, as well as initialization utilities.

For example, the following code defines three aliases that will be later used as the genetic operators:

```
from deap import tools
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.02)
```

The three aliases are defined in detail as follows:

- `select` is registered as an alias to the existing `tools` function, `selTournament()`, with the `tournsize` argument set to `3`. This creates a `toolbox.select` operator that performs tournament selection with a tournament size of `3`.
- `mate` is registered as an alias to the existing `tools` function, `cxTwoPoint()`. This results in a `toolbox.mate` operator that performs two-point crossover.
- `mutate` is registered as an alias to the existing tools function, `mutFlipBit`, with the `indpb` argument set to `0.02`, providing a `toolbox.mutate` operator that performs flip bit mutation with `0.02` as the probability for each attribute to be flipped.

The `tools` module provides implementations of various genetic operators, including several of the ones we mentioned in the previous chapter.

Selection functions can be found in the `selection.py` file. Some of them are listed as follows:

- `selRoulette()` implements roulette wheel selection.
- `selStochasticUniversalSampling()` implements **Stochastic Universal Sampling** (**SUS**).
- `selTournament()` implements tournament selection.

Crossover functions can be found in the `crossover.py` file:

- `cxOnePoint()` implements single-point crossover.
- `cxUniform()` implements uniform crossover.
- `cxOrdered()` implements **ordered crossover** (**OX1**).
- `cxPartialyMatched()` implements **partially matched crossover** (**PMX**).

A couple of the `Mutation` functions that can be found in the `mutation.py` file are as follows:

- `mutFlipBit()` implements flip bit mutation.
- `mutGaussian()` implements normally distributed mutation.

Creating the population

The `init.py` file of the tools module contains several functions that can be useful for creating and initializing the population for the genetic algorithm. One particularly useful function is `initRepeat()`, which accepts three arguments:

- The container type in which we would like to put the resulting objects
- The function used to generate objects that will be put into the container
- The number of objects we want to generate

For example, the following line of code will produce a list of 30 random numbers between 0 and 1:

```
randomList = tools.initRepeat(list, random.random, 30)
```

In this example, `list` is the type serving as the container to be filled, `random.random` is the generator function, and `30` is the number of times we will call the function to generate values that fill the container.

What if we wanted to fill the list with integer random numbers that are either `0` or `1`? We could, for example, create a function that utilizes `random.radint()` to generate a single random value of `0` or `1`, and then use it to be the generator function of `initRepeat()`, as shown in the following code snippet:

```
def zeroOrOne():
    return random.randint(0, 1)

randomList = tools.initRepeat(list, zeroOrOne, 30)
```

Or, we can take advantage of `toolbox`, as follows:

```
toolbox.register("zeroOrOne", random.randint, 0, 1)
randomList = tools.initRepeat(list, toolbox.zeroOrOne, 30)
```

Here, instead of explicitly defining the `zeroOrOne()` function, we created the `zeroOrOne` operator (or alias), which calls `random.radint()` with the fixed parameters of `0` and `1`.

Calculating the fitness

As we mentioned before, while the `Fitness` class defines the fitness weights that determine its strategy (such as maximization or minimization), the actual fitness values are obtained from a separately defined function. This fitness calculation function is typically registered with the `toolbox` module using the alias name `evaluate`, as shown in the following code snippet:

```
def someFitnessCalculationFunction(individual):
    return _some_calculation_of_the_fitness

toolbox.register("evaluate",someFitnessCalculationFunction)
```

In this example, `someFitnessCalculationFunction()` calculates the fitness for any given individual, and `evaluate` is registered as its alias.

We are finally ready to put our knowledge to use and solve our first problem using a genetic algorithm written with DEAP—as described in the next section.

The OneMax problem

The OneMax (or One-Max) problem is a simple optimization task that is often used as the *Hello World* of genetic algorithm frameworks. We will use this problem for the rest of this chapter to demonstrate how DEAP can be used to implement a genetic algorithm.

The OneMax task is to find the binary string of a given length that maximizes the sum of its digits. For example, the OneMax problem of length 5 will consider candidates such as the following:

- 10010 (sum of digits = 2)
- 01110 (sum of digits = 3)
- 11111 (sum of digits = 5)

Obviously (to us), the solution to this problem is always the string that comprises all 1s. But the genetic algorithm does not have this knowledge, and needs to blindly look for this solution using its genetic operators. If the algorithm does its job, it will find this solution, or at least one close to it, within a reasonable amount of time.

The DEAP framework documentation uses the OneMax problem as its introductory example (`https://github.com/DEAP/deap/blob/master/examples/ga/onemax.py`). In the following sections, we will describe our version for DEAP's OneMax example.

Solving the OneMax problem with DEAP

In the previous chapter, we mentioned several choices that need to be made when solving a problem using the genetic algorithm approach. As we tackle the OneMax problem, we will make these choices as a series of steps. In the chapters to follow, we will keep using the same series of steps as we apply the genetic algorithms approach to various types of problems.

Choosing the chromosome

Since the OneMax problem deals with binary strings, the choice of chromosome is easy—each individual will be represented with a binary string that directly represents a candidate solution. In the actual Python implementation, this will be implemented as a list containing integer values of either `0` or `1`. The length of the chromosome matches the size of the OneMax problem. For example, for a OneMax problem of size 5, the 10010 individual will be represented by the list `[1, 0, 0, 1, 0]`.

Calculating the fitness

Since we want to find the individual with the **largest** sum of digits, we are going to use the `FitnessMax` strategy. And as each individual is represented by a list of integer values of either `0` or `1`, the fitness value will be directly calculated as the sum of the elements in the list, for example: `sum([1, 0, 0, 1, 0]) = 2`.

Choosing the genetic operators

We now need to decide on the genetic operators to be used—selection, crossover, and mutation. In the previous chapter, we examined several different types of each of these operators. Choosing the genetic operators is not an exact science, and we can usually experiment with several choices. But while selection operators can typically work with any chromosome type, the crossover and mutation operators we choose need to match the chromosome type we use, otherwise they could produce invalid chromosomes.

For the `selection` operator, we can start with `Tournament` selection because it is simple and efficient. We can later experiment with other selection strategies, such as roulette wheel selection and SUS.

For **crossover** operator, either single-point or two-point crossover operators will be suitable, as the result of crossing over two binary strings using these methods will produce a valid binary string.

The `mutation` operator can be the simple flip bit mutation that works well for binary strings.

Setting the stopping condition

It is always a good idea to put a limit on the number of generations, to guarantee that the algorithm does not run forever. This gives us one stopping condition.

In addition, since we happen to know the best solution for the OneMax problem—a binary string with all 1s, and a fitness value equal to the length of the individual—we can use that as a second stopping condition.

Note that for a real-world problem, we typically don't have this kind of knowledge in advance.

If either of these conditions is met, that is, the number of generations reaches the limit or best solution is found, the genetic algorithm will stop.

Implementing with DEAP

Putting it all together, we can finally start coding our solution to the OneMax problem using the DEAP framework.

The complete program containing the code snippets shown in this section can be found here: `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter03/01-OneMax-long.py`.

Setting up

Before we start the actual genetic algorithm flow, we need to set things up, and the DEAP framework has a quite distinct way of doing it, as shown in the rest of this section:

1. We start by importing the essential modules of the DEAP framework, followed by a couple of useful utilities:

```
from deap import base
from deap import creator
from deap import tools

import random
import matplotlib.pyplot as plt
```

2. Next, we declare a few constants that set the parameters for the problem and control the behavior of the genetic algorithm:

```
# problem constants:
ONE_MAX_LENGTH = 100     # length of bit string to be optimized

# Genetic Algorithm constants:
POPULATION_SIZE = 200    # number of individuals in population
P_CROSSOVER = 0.9        # probability for crossover
P_MUTATION = 0.1         # probability for mutating
                         # an individual
MAX_GENERATIONS = 50     # max number of generations for
                         # stopping condition
```

3. One important aspect of the genetic algorithm is the use of probability, which introduces a random element to the behavior of the algorithm. However, when experimenting with the code, we may want to be able to run the same experiment several times and get repeatable results. To accomplish this, we set the random function seed to a constant number of some value, as shown in the following code snippet:

```
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

At some point, you may decide to remove these lines of code, so separate runs could produce somewhat different results.

4. As we have seen earlier in this chapter, the `Toolbox` class is one of the main utilities provided by the DEAP framework, enabling us to register new functions (or operators) that customize existing functions using preset arguments. Here we use it to create the `zeroOrOne` operator, which customizes the `random.randomint(a, b)` function. This function normally returns a random integer `N` such that `a ≤ N ≤ b`. By fixing the two arguments, `a` and `b`, to the values `0` and `1`, the `zeroOrOne` operator will randomly return either the value `0` or the value `1` when called later in the code. The following code snippet defines the `toolbox` variable, and then uses it to register the `zeroOrOne` operator:

```
toolbox = base.Toolbox()
toolbox.register("zeroOrOne", random.randint, 0, 1)
```

5. Next, we need to create the `Fitness` class. Since we only have one objective here—the sum of digits—and our goal is to maximize it, we choose the `FitnessMax` strategy, using a `weights` tuple with a single positive weight, as shown in the following code snippet:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

6. In DEAP, the convention is to use a class called `Individual` to represent each of the population's individuals. This class is created with the help of the `creator` tool. In our case, `list` serves as the base class, which is used as the individual's chromosome. The class is augmented with the `fitness` attribute, initialized to the `FitnessMax` class that we defined earlier:

```
creator.create("Individual", list, fitness=creator.FitnessMax)
```

7. Next, we register the `individualCreator` operator, which creates an instance of the `Individual` class, filled up with random values of either `0` or `1`. This is done by customizing the previously defined `zeroOrOne` operator. This definition makes use of the `initRepeat` operator that was mentioned earlier as the base class, and is customized here using the following arguments:
    - The `Individual` class as the container type in which the resulting objects will be placed
    - The `zeroOrOne` operator as the function used to generate objects
    - The constant `ONE_MAX_LENGTH` as the number of objects we want to generate (currently set to `100`)

    Since the objects generated by the `zeroOrOne` operator are integers with random values of either `0` or `1`, the resulting `individualCreator` operator will fill an `Individual` instance with 100 randomly generated values of `0` or `1`:

```
toolbox.register("individualCreator", tools.initRepeat,
        creator.Individual, toolbox.zeroOrOne, ONE_MAX_LENGTH)
```

8. Lastly, we register the `populationCreator` operator that creates a list of individuals. This definition, too, uses the `initRepeat` operator, with the following arguments:
    - The `list` class as the container type
    - The `individualCreator` operator defined earlier as the function used to generate the objects in the list

    The last argument for `initRepeat`—the number of objects we want to generate—is not given here. This means that when using the `populationCreator` operator, this argument will be expected and used to determine the number of individuals created, in other words, the population size:

```
toolbox.register("populationCreator", tools.initRepeat,
                 list, toolbox.individualCreator)
```

9. To facilitate the fitness calculation (or `evaluation`, in DEAP terminology), we start by defining a standalone function that accepts an instance of the `Individual` class and returns the fitness for it. Here, we defined a function named `oneMaxFitness` that computes the number of 1s in the individual. Since the individual is essentially a list with values of either `1` or `0`, the Python `sum()` function can be directly used for this purpose:

```
def oneMaxFitness(individual):
 return sum(individual), # return a tuple
```

As mentioned before, fitness values in DEAP are represented as tuples, and therefore a comma needs to follow when a single value is returned.

10. Next, we define the `evaluate` operator as an alias to the `oneMaxfitness()` function we defined earlier. As you will see later, using the `evaluate` alias for calculating the fitness is a DEAP convention:

```
toolbox.register("evaluate", oneMaxFitness)
```

11. As we mentioned in the previous section, the genetic operators are typically created by aliasing existing functions from the `tools` module and setting the argument values as needed. Here, we chose the following:
    - Tournament selection with a tournament size of `3`
    - Single point crossover
    - Flip bit mutation

    Note the `indpb` parameter of the `mutFlipBit` function. This function iterates over all the attributes of the individual, a list with values of 1s and 0s in our case, and for each attribute will use this argument value as the probability of flipping (applying the *not* operator to) the attribute value. This value is independent of the mutation probability, which is set by the `P_MUTATION` constant that we defined earlier and has not yet been used. The mutation probability serves to decide if the `mutFlipBit` function is called for a given individual in the population:

```
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("mate", tools.cxOnePoint)
toolbox.register("mutate", tools.mutFlipBit,
                                    indpb=1.0/ONE_MAX_LENGTH)
```

We are finally done with our settings and definitions, and we're ready to start the genetic flow, described in the next section.

Evolving the solution

The genetic flow is implemented in the `main()` function, as described in the following steps:

1. We start the flow by creating the initial population using the `populationCreator` operator we defined earlier, and the `POPULATION_SIZE` constant as the argument for this operator. The `generationCounter` variable, which will be used later on, is initialized here as well:

```
population = toolbox.populationCreator(n=POPULATION_SIZE)
generationCounter = 0
```

2. To calculate the fitness for each individual in the initial population, we use the Python `map()` function to apply the `evaluate` operator to each item in the population. As the `evaluate` operator is an alias for the `oneMaxFitness()` function, the resulting iterable consists of the calculated fitness tuple of each individual. It is then converted to a `list` of tuples:

```
fitnessValues = list(map(toolbox.evaluate, population))
```

3. Since the items of `fitnessValues` respectively match those in `population` (which is a list of individuals), we can use the `zip()` function to combine them and assign the matching fitness tuple to each individual:

```
for individual, fitnessValue in zip(population, fitnessValues):
    individual.fitness.values = fitnessValue
```

4. Next, since we have single-objective fitness, we extract the first value out of each fitness for gathering statistics:

```
fitnessValues = [individual.fitness.values[0] for individual in
population]
```

5. The statistics collected will be the max fitness value and the mean (average) fitness value for each generation. Two lists will be used for this purpose, and they are created next:

```
maxFitnessValues = []
meanFitnessValues = []
```

6. We are finally ready for the main loop of the genetic flow. At the top of the loop, we find the stopping conditions. As we decided earlier, one stopping condition is set by putting a limit on the number of generations, and the other by detecting that we have reached the best solution (a binary string containing all 1s):

```
while max(fitnessValues) < ONE_MAX_LENGTH and generationCounter
< MAX_GENERATIONS:
```

7. The generation counter is updated next. It is used by the stopping condition, as well as the `print` statements we will see soon:

```
generationCounter = generationCounter + 1
```

8. At the heart of the genetic algorithm are the genetic operators, which are applied next. The first is the `selection` operator, applied using the `toolbox.select` operator we earlier defined as the tournament selection. Since we already set the tournament size when the operator was defined, we only need to send the population and its length as arguments now:

```
offspring = toolbox.select(population, len(population))
```

9. The selected individuals, now residing in a list called `offspring`, are next cloned, so we can apply the next genetic operators without affecting the original population:

```
offspring = list(map(toolbox.clone, offspring))
```

Note that despite the name `offspring`, these are still clones of individuals from the previous generation, and we still need to mate them using the crossover operator to create the actual offspring.

10. The next genetic operator is the `crossover`. It was earlier defined as the `toolbox.mate` operator, and is aliasing a single-point crossover. We use Python extended slices to pair every even-indexed item of the `offspring` list with the one following it. We then utilize the `random()` function to *flip a coin* using the crossover probability set by the `P_CROSSOVER` constant. This will decide if the pair of individuals will be crossed over or remain intact. Lastly, we delete the fitness values of the children, since they have been modified and their existing fitness values are no longer valid:

```
for child1, child2 in zip(offspring[::2], offspring[1::2]):
    if random.random() < P_CROSSOVER:
        toolbox.mate(child1, child2)
        del child1.fitness.values
        del child2.fitness.values
```

Note that the `mate` function takes two individuals as arguments and modifies them in place, meaning they don't need to be reassigned.

11. The final genetic operator to be applied is the mutation, which was earlier registered as the `toolbox.mutate` operator, and was set to be a flip bit mutation operation. Iterating over all offspring items, the mutation operator will be applied at the probability set by the mutation probability constant, `P_MUTATION`. If the individual gets mutated, we make sure to delete its fitness value (if it exists). This value could have carried over with the individual from the previous generation, and after mutation it is no longer correct and needs to be recalculated:

```
for mutant in offspring:
    if random.random() < P_MUTATION:
        toolbox.mutate(mutant)
        del mutant.fitness.values
```

12. Individuals that were not crossed over or mutated remained intact, and therefore their existing fitness values, which were already calculated in a previous generation, do not need to be calculated again. The rest of the individuals will have this value empty. We now find those fresh individuals using the `Fitness` class' `valid` property, then calculate the new fitness for them in a similar manner to the original calculation of fitness values:

```
freshIndividuals = [ind for ind in offspring if not
ind.fitness.valid]
freshFitnessValues = list(map(toolbox.evaluate,
```

```
freshIndividuals))
for individual, fitnessValue in zip(freshIndividuals,
freshFitnessValues):
    individual.fitness.values = fitnessValue
```

13. Now that the genetic operators are done, it is time to replace the old population with the new one:

```
population[:] = offspring
```

14. Before we continue to the next round, the current fitness values are collected to allow for statistics gathering. Since the fitness value is a (single element) tuple, we need to select the `[0]` index:

```
fitnessValues = [ind.fitness.values[0] for ind in population]
```

15. The max and mean fitness values are then found, their values get appended to the statistics accumulators, and a summary line is printed out:

```
maxFitness = max(fitnessValues)
meanFitness = sum(fitnessValues) / len(population)
maxFitnessValues.append(maxFitness)
meanFitnessValues.append(meanFitness)
print("- Generation {}: Max Fitness = {}, Avg Fitness = {}"
      .format(generationCounter, maxFitness, meanFitness))
```

16. In addition, we locate the index of the (first) best individual using the max fitness value we just found, and print this individual out:

```
best_index = fitnessValues.index(max(fitnessValues))
print("Best Individual = ", *population[best_index], "\n")
```

17. Once a stopping condition was activated and the genetic algorithm flow concludes, we can use the statistics accumulators to plot a couple of graphs using the `matplotlib` library. We use the following code snippet to draw a graph illustrating the progress of the best and average fitness values throughout the generations:

```
plt.plot(maxFitnessValues, color='red')
plt.plot(meanFitnessValues, color='green')
plt.xlabel('Generation')
plt.ylabel('Max / Average Fitness')
plt.title('Max and Average fitness over Generations')
plt.show()
```

We are finally ready to test our first genetic algorithm—let's run it to find out if it finds the OneMax solution.

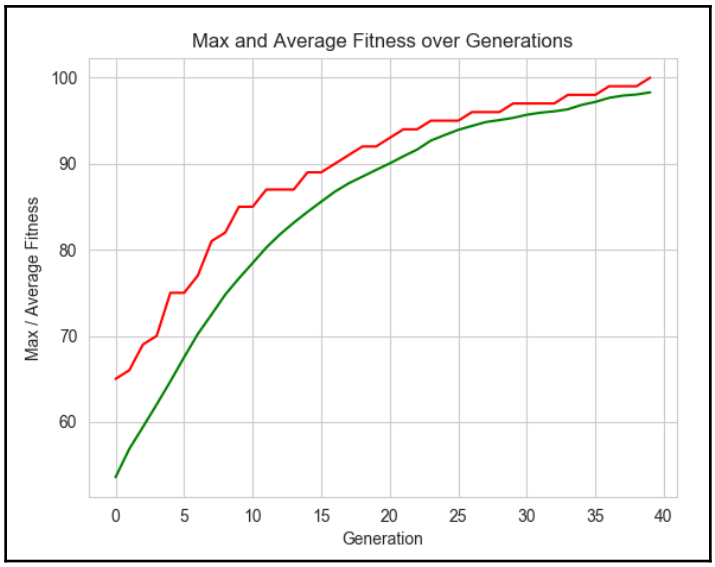
Running the program

When running the program described in the previous sections, we get the following output:

```
- Generation 1: Max Fitness = 65.0, Avg Fitness = 53.575
Best Individual = 1 1 0 1 0 1 0 0 1 0 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 1 1 1 1
1 0 1 1 1 1 0 1 0 1 1 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 0 0 0
0 1 0 1 0 1 1 1 0 1 1 0 0 0 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0

...

- Generation 40: Max Fitness = 100.0, Avg Fitness = 98.29
Best Individual = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

You can see how after 40 generations an *all-1* solution was found, which yielded a fitness of 100 and stopped the genetic flow. The average fitness, that started at a value of around 53, ended at a value close to 100.

The graph plotted by `matplotlib` is shown as follows:



Stats of the program solving the OneMax problem

This graph illustrates how max fitness increases over generations with incremental leaps, while the average fitness keeps progressing smoothly.

Now that we have solved the OneMax problem using the DEAP framework, let's move on to the next section to find out how we can make our code more concise.

Using built-in algorithms

The DEAP framework comes with several built-in evolutionary algorithms provided by the `algorithms` module. One of them, `eaSimple`, implements the genetic algorithm flow we have been using, and can replace most of the code we earlier had in the `main` method. Other useful DEAP objects, `Statistics` and `Logbook`, can be used for statistics gathering and printing, as we will soon see.

The program described in this section implements the same solution to the OneMax problem as the program discussed in the previous section, but with less code. The only differences are in the `main` method. We will describe these differences in the following code snippets.

The complete program can be found here:

`https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter03/02-OneMax-short.py`

The Statistics object

The first change we will make is in the way statistics are being gathered. To this end, we will now take advantage of the `tools.Statistics` class provided by DEAP. This utility enables us to create a statistics object using a key argument, which is a function that will be applied to the data on which the statistics are computed:

1. Since the data we plan to provide is be the population of each generation, we set the key function to one that extracts the fitness value(s) from each individual:

```
stats = tools.Statistics(lambda ind: ind.fitness.values)
```

2. We can now register various functions that can be applied to these values at each step. In our example, we only use the `max` and `mean` NumPy functions, but others (such as `min` and `std`) can be registered as well:

```
stats.register("max", numpy.max)
stats.register("avg", numpy.mean)
```

As we will see soon, the collected statistics will be returned in an object called the `logbook` at the end of the run.

The algorithm

Now it's time for the actual flow. This is done with a single call to the `algorithms.eaSimple` method, one of the built-in evolutionary algorithms provided by the `algorithms` module of DEAP. When we call the method, we provide it with the population, `toolbox`, and the statistics object among other parameters:

```
population, logbook = algorithms.eaSimple(population, toolbox,
                                          cxpb=P_CROSSOVER,
                                          mutpb=P_MUTATION,
                                          ngen=MAX_GENERATIONS,
                                   stats=stats, verbose=True)
```

The `algorithms.eaSimple` method assumes that we have previously used `toolbox` to register the following operators—`evaluate`, `select`, `mate`, and `mutate`—which we have already done when we created the original program. The stopping condition here is set by the value of `ngen`, the number of generations to run the algorithm for.

The logbook

When the flow is done, the algorithm returns two objects—the final population and a `logbook` object containing the statistics collected. We can now extract the desired statistics from the logbook using the `select()` method, so we can use them for plotting just as we did before:

```
maxFitnessValues, meanFitnessValues = logbook.select("max", "avg")
```

We are now ready to run this slimmer version of the program.

Running the program

When running the program with the same parameter values and settings as used in the previous one, the printouts will be as follows:

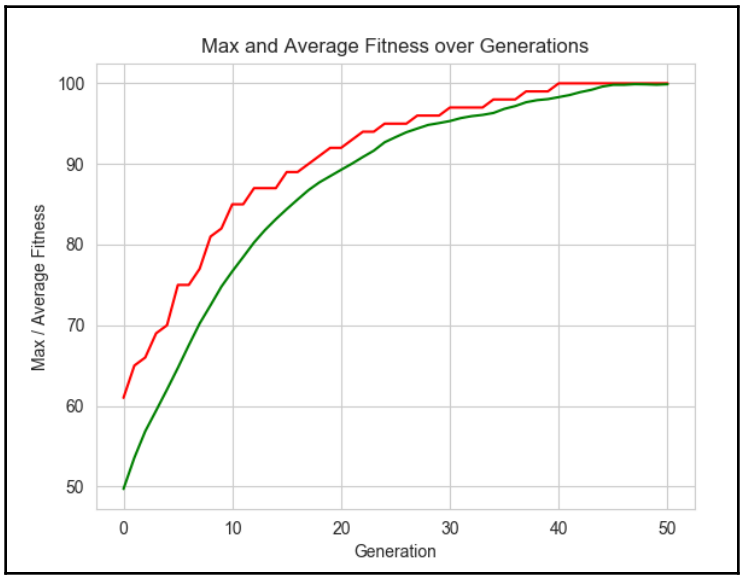
```
gen   nevals  max    avg
0     200     61     49.695
1     193     65     53.575

...

39    192     99     98.04
40    173     100    98.29

...

49    187     100    99.83
50    184     100    99.89
```

These printouts are automatically generated by the `algorithms.eaSimple` method, according to the way we defined the statistics object sent to it, as the verbose argument was set to `True`.

The results are numerically similar to what we saw in the previous program, with two differences:

- There is a printout for generation 0; this was not included in the previous program.
- The genetic flow here continues all the way to the 50th generation, as this was the only stopping condition, while in the previous program there was an additional stopping condition that stopped the flow at the 40th generation, since the best solution (that we happened to know beforehand) was reached.

We can observe the same behavior in the new graph plot. The graph is similar to the one we saw before, but continues to the 50th generation, even though the best result was already reached at the 40th generation:



Stats of the program solving the OneMax problem using the built-in algorithm

Consequently, starting at the 40th generation, the value of the best fitness no longer changes, while the average fitness keeps climbing until it almost reaches the same max value; this means that by the end of this run, nearly all individuals are identical to the best one.

Adding the hall of fame

One additional feature of the built-in `algorithms.eaSimple` method is the **hall of fame** (or **hof** for short). Implemented in the `tools` module, the `HallOfFame` class can be used to retain the best individuals that ever existed in the population during the evolution, even if they have been lost at some point due to selection, crossover, or mutation. The hall of fame is continuously sorted so that the first element is the first individual that had the best fitness value ever seen.

The complete program containing the code snippets shown in this section can be found here:

`https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter03/03-OneMax-short-hof.py`

To add the hall of fame functionality, let's make a few modifications to the previous program:

1. We start by defining a constant for the number of individuals we want to keep in the hall of fame. We will add this line to the constant definition section:

```
HALL_OF_FAME_SIZE = 10
```

2. Just before calling the `eaSimple` algorithm, we create the `HallOfFame` object with that size:

```
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)
```

3. The `HallOfFame` object is sent as an argument to the `eaSimple` algorithm, which internally updates it during the run of the genetic algorithm flow:

```
population, logbook = algorithms.eaSimple(population, toolbox,
cxpb=P_CROSSOVER, mutpb=P_MUTATION, ngen=MAX_GENERATIONS,
stats=stats, halloffame=hof, verbose=True)
```

4. When the algorithm is done, we can use the `HallOfFame` object `items` attribute to access the list of individuals inducted to the hall of fame:

```
print("Hall of Fame Individuals = ", *hof.items, sep="\n")
print("Best Ever Individual = ", hof.items[0])
```

The printed results look as follows—the best individual consisting of all 1s, followed by various individuals that have a 0 value in different locations:

```
Hall of Fame Individuals =
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

...
```

The best ever individual is the same one that was printed in *step 4* of this section:

```
Best Ever Individual = [1, 1, 1, 1, 1, ...
```

From now on, we will use these features—the `statistics` object and the logbook, the built-in algorithm, and the hall of fame—in all the programs we create.

Experimenting with the algorithm's settings

We can now experiment with the various settings and definitions we placed into the program and observe the changes in the behavior and the results.

In each of the following subsections, we start from the original program settings and make one or more changes. You are encouraged to experiment with making your own modifications, as well as combining several modifications to the same program.

Bear in mind that the effects of changes we make may be specific to the problem at hand, a simple OneMax in our case, and may be different for other types of problems.

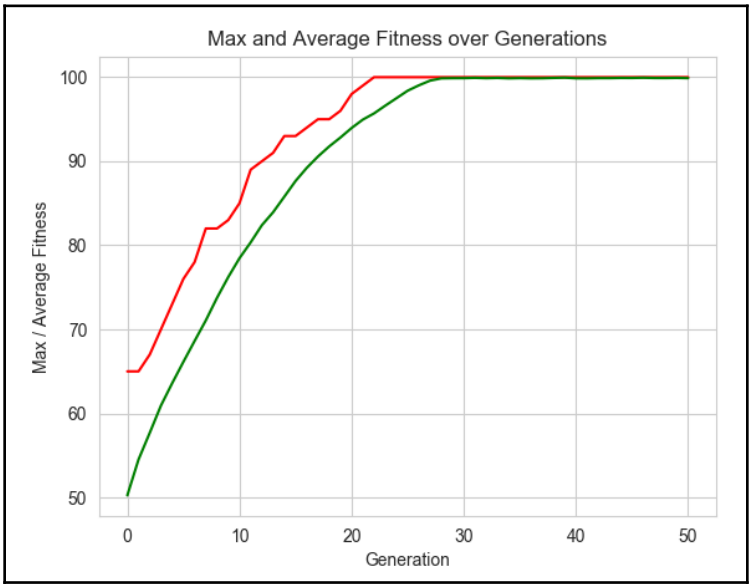
Population size and number of generations

We will start our experimentation by making modifications to the population size and the number of generations used by the genetic algorithm:

1. The size of the population is determined by the `POPULATION_SIZE` constant. We will start by increasing the value of the constant from `200` to `400`:

```
POPULATION_SIZE = 400
```

This modification accelerates the genetic flow, and the best solution is now found after 22 generations, as the following generated plot illustrates:

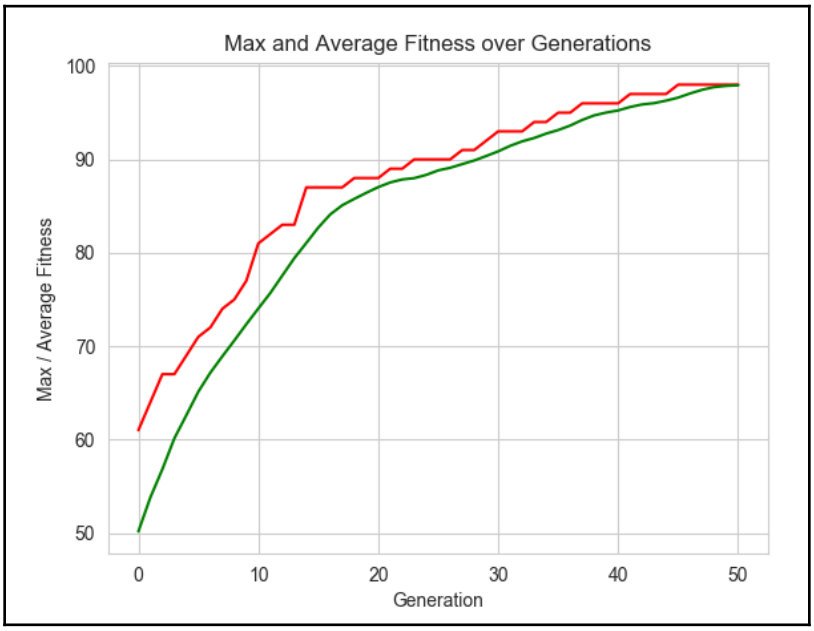


Stats of the program solving the OneMax problem, after increasing the population size to 400

2. Next, we will try reducing the population size to 100:

```
POPULATION_SIZE = 100
```

This modification slows down the convergence of the algorithm, which will no longer reach the best possible value after 50 generations:

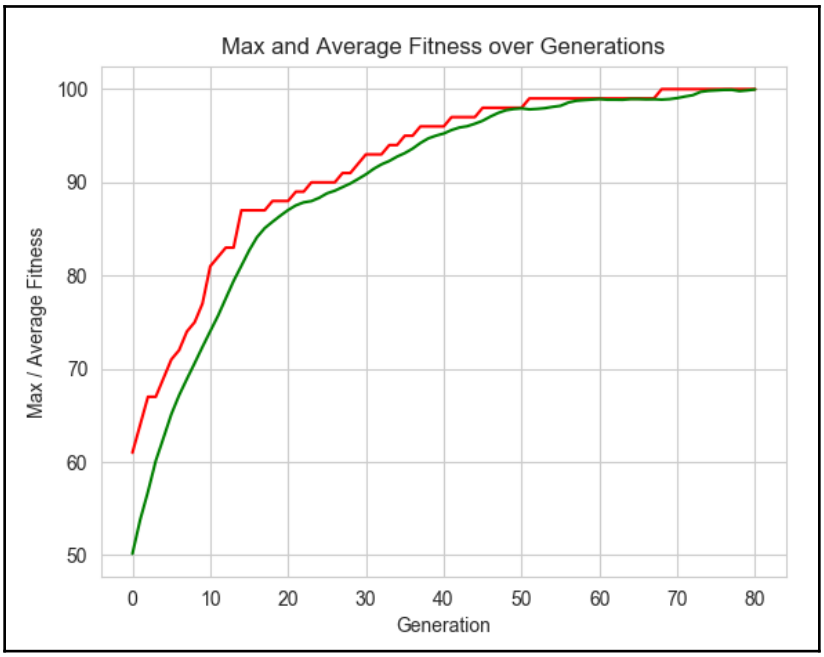


Stats of the program solving the OneMax problem, after decreasing the population size to 100

3. To compensate, let's try increasing the value of `MAX_GENERATIONS` to `80`:

```
MAX_GENERATIONS = 80
```

We find that the best solution is now reached after 68 generations:



Stats of the program solving the OneMax problem, after increasing the number of generations to 80

This behavior is typical to genetic algorithm-based solutions—increasing the population will require fewer generations to reach a solution. However, the computational and memory requirements increase with the population size, and we typically aspire to find a moderate population size that will provide a solution within a reasonable amount of time.

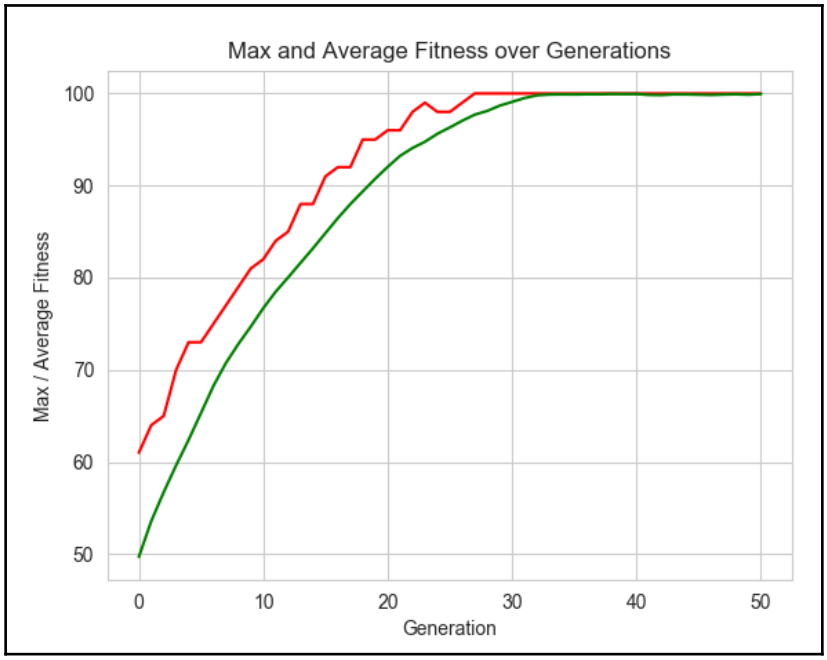
Crossover operator

Let's reset our changes and go back to the original settings (50 generations, population size 200). We are now ready to experiment with the crossover operator that's responsible for creating offspring from parent individuals.

Changing the crossover operator from a single-point to two-point crossover is simple, as we now define the crossover operator as follows:

```
toolbox.register("mate", tools.cxTwoPoint)
```

The algorithm now finds the best solution after only 27 generations:



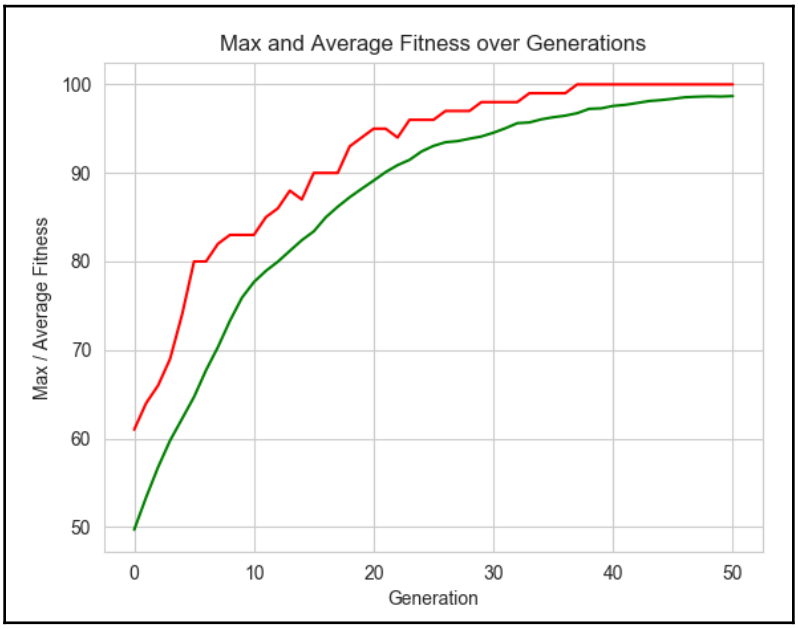
Stats of the program solving the OneMax problem after switching to two-point crossover

This behavior is typical to genetic algorithms utilizing binary string representation, as two-point crossover provides a more versatile way to combine two parents and mix their genes in comparison to the single-point crossover.

Mutation operator

We will now reset our changes again, as we get ready to experiment with the mutation operator, which is responsible for introducing random modifications to offspring:

1. We will start by increasing the value of the `P_MUTATION` constant to `0.9`. This results in the following plot:



Stats of the program solving the OneMax problem after increasing the mutation probability to 0.9

The results may seem surprising at first, as increasing the mutation rate typically causes the algorithm to behave erratically, while here the effect is seemingly unnoticeable. However, recall that there is another mutation-related parameter in our algorithm, `indpb`, which is an argument of the specific mutation operator we used here—`mutFlipBit`:

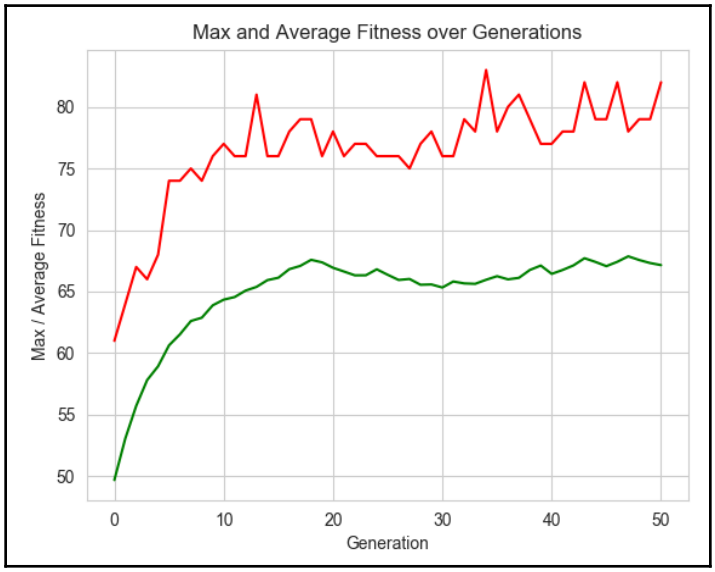
```
toolbox.register("mutate", tools.mutFlipBit,
indpb=1.0/ONE_MAX_LENGTH)
```

While the value of `P_MUTATION` determines the probability of an individual to be mutated, `indpb` determines the probability of each bit in a given individual to be flipped. In our program, we set the value of `indpb` to `1.0/ONE_MAX_LENGTH`, which means that on average a single bit will be flipped in a mutated solution. For our 100-bit-long OneMax problem, this seems to limit the effect of mutation regardless of the `P_MUTATION` constant value.

2. We now increase the value of `indpb` tenfold, as follows:

```
toolbox.register("mutate", tools.mutFlipBit,
indpb=10.0/ONE_MAX_LENGTH)
```

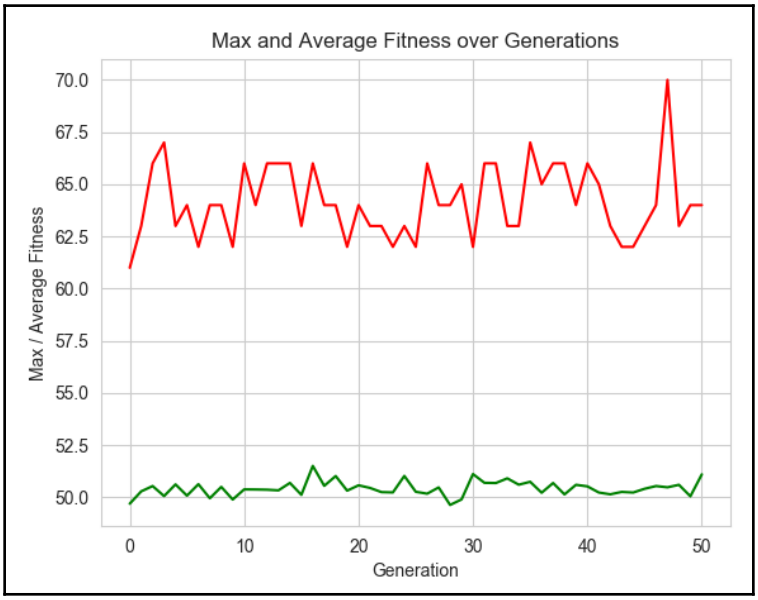
The result of running the algorithm with this value is somewhat erratic, as shown in the following plot:



Stats of the program after a tenfold increase to the per-bit mutation probability

The graphs in the preceding plot indicate that while at first, the algorithm is able to improve the results, it quickly gets stuck in a state of oscillations without being able to make significant improvements.

3. Increasing the `indpb` value further, to `50.0/ONE_MAX_LENGTH`, results in the following, unstable-looking, graph:



Stats of the program after a fifty-fold increase to the per-bit mutation probability

As evident from this plot, the genetic algorithm now turned into the equivalent of a random search—it may stumble upon the best solution by chance, but it does not make any progress towards better solutions.

Selection operator

Next, we look at the `selection` operator. First, we will change the tournament size to see the combined effect of this parameter with the mutation probability. Then we will look at using roulette selection instead of tournament selection.

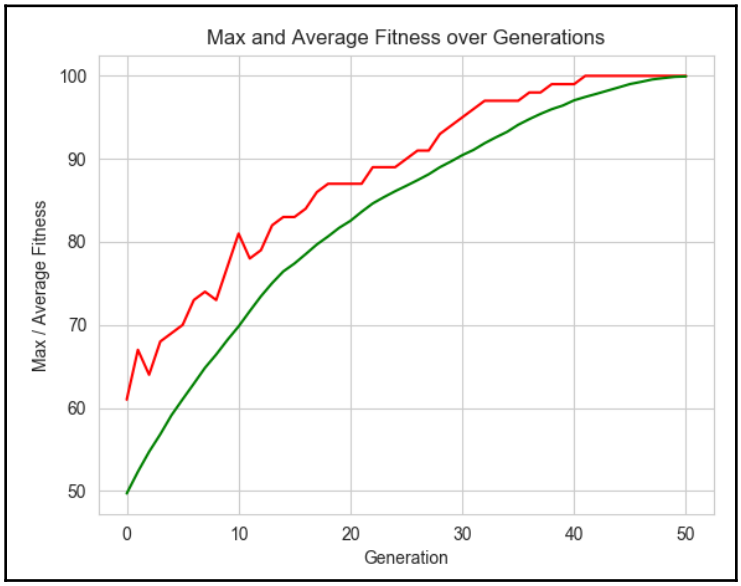
Tournament size and relation to mutation probability

Once again, we start by changing back to the original settings of the program before we make new modifications:

1. We will first modify the `tournamentSize` parameter of the tournament selection algorithm to `2` (instead of the original value of `3`):

```
toolbox.register("select", tools.selTournament, tournsize=2)
```

This does not seem to have a noticeable effect on the algorithm's behavior:

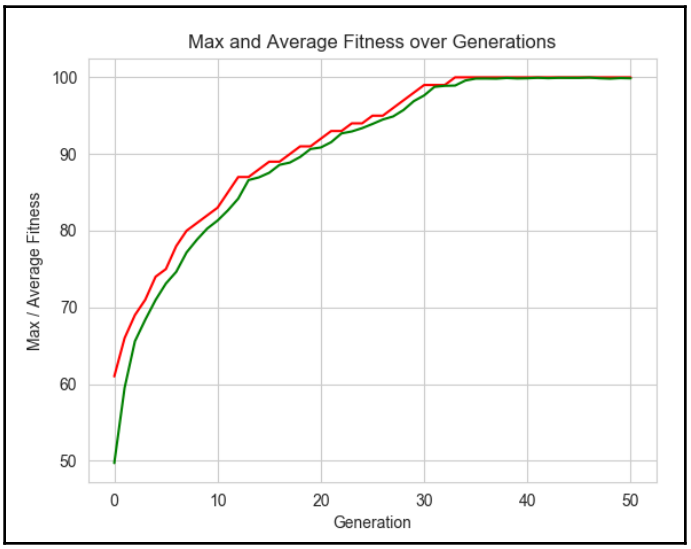


Stats of the program solving the OneMax problem, after decreasing the tournament size to two

2. What if we increase the tournament size to a very large value, say `100`? Let's see:

```
toolbox.register("select", tools.selTournament, tournsize=100)
```

The algorithm still behaves well and finds the best solution in less than 40 generations. One noticeable effect is that the max fitness now closely resembles the average fitness, as shown in the following screenshot:



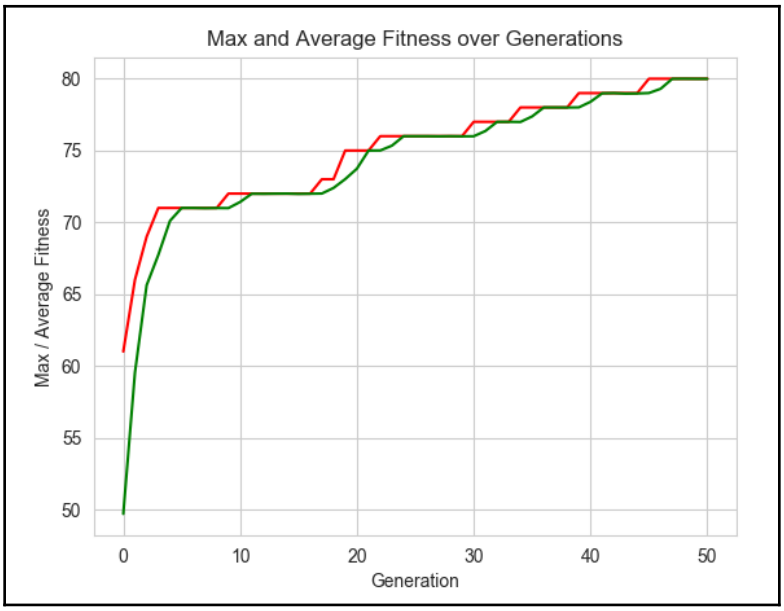
Stats of the program after increasing the tournament size to 100

This behavior is due to the fact that when the tournament size increases, the chance of weak individuals being selected diminishes, and better solutions tend to take over the population. In real-life problems, this takeover might cause suboptimal solutions to saturate the population and prevent the best solution from being found (a phenomenon known as **premature convergence**). However, in the case of the simple OneMax problem, this does not seem to be an issue. A possible explanation is that the mutation operator provides enough diversity to keep the solutions moving in the right direction.

3. To put this explanation to the test, let's reduce the mutation probability tenfold, to `0.01`:

```
P_MUTATION = 0.01
```

If we run the algorithm again, we can see that the results stop improving soon after the start of the algorithms, and then improve at a much slower pace, with an occasional improvement here and there. The overall results are far worse than the previous run:



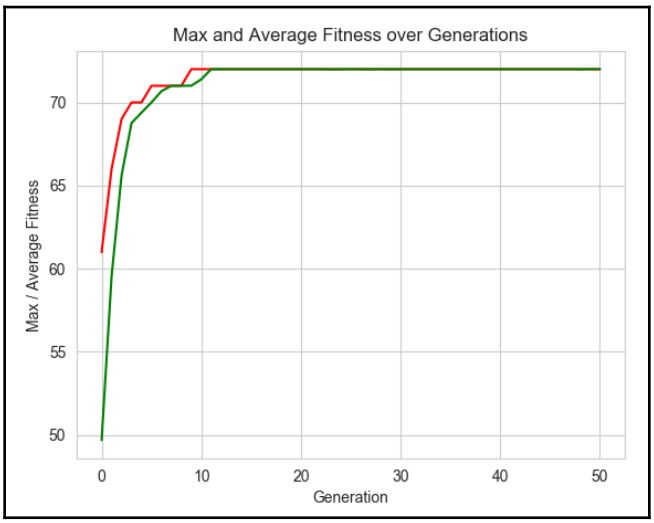
Stats of the program with a tournament size of 100 and a mutation probability of 0.01

The interpretation is that due to the large tournament size, the best individuals from the initial population take over within a small number of generations, which shows in the initial quick increase of both graphs in the plot. After that, only an occasional mutation in the right direction—one that flips a `0` to `1`—creates a better individual; this shows in the plot by a jump of the red graph. Soon after, this individual takes over the entire population again, where the green graph catches up with the red one.

4. To make this situation even more extreme, we can further reduce the mutation rate:

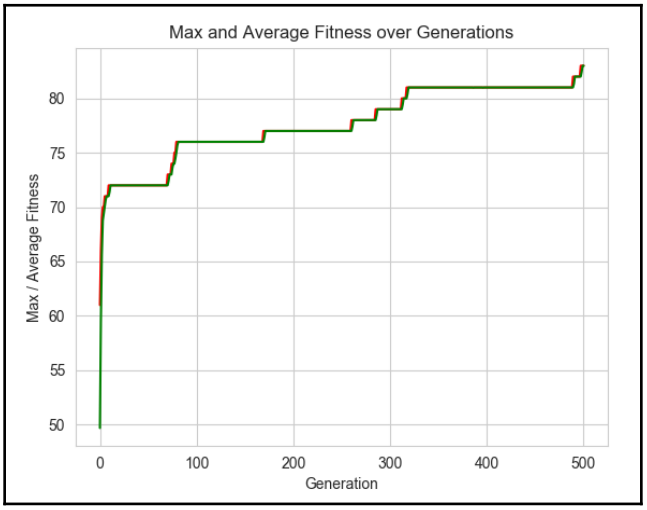
```
P_MUTATION = 0.001
```

We now see the same general behavior, but since mutations are very rare, the improvements are few and far between:



Stats of the program with a tournament size of 100 and a mutation probability of 0.001

5. If we now increase the number of generations to 500, we can see this behavior more clearly:

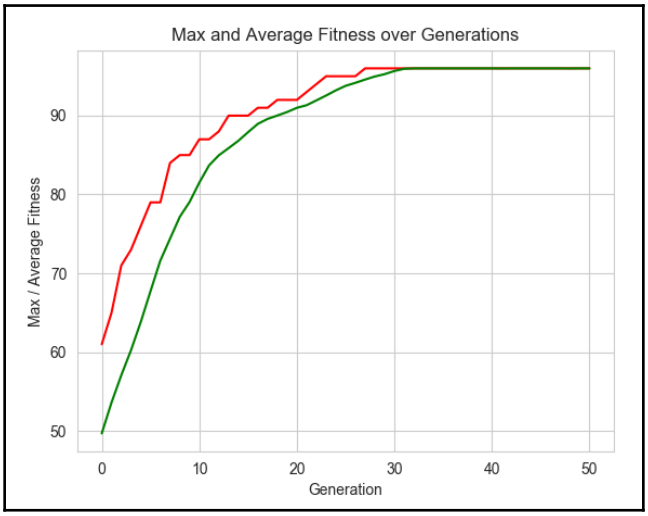


Stats of the program with a tournament size of 100 and a mutation probability of 0.001, over 500 generations

6. Just out of curiosity, let's dial back the tournament size to `3` again and restore the number of generations to `50`, leaving the small mutation rate in place:

```
MAX_GENERATIONS = 50
toolbox.register("select", tools.selTournament, tournsize=3)
```

The resulting plot is a lot closer to the original one:



Stats of the program with a tournament size of 3 and a mutation probability of 0.001, over 50 generations

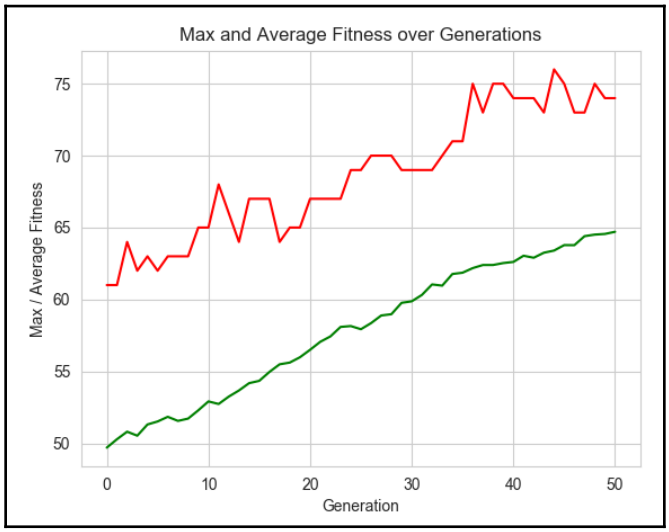
Here, it seems that a takeover occurred as well, but far later in time, around generation 30, when the best fitness was close to the max value of 100. Here, a more reasonable mutation rate would help us find the best solution, as happened with the original settings.

Roulette wheel selection

Let's go back to the original settings once more, in preparation for our last experiment, as we will now try replacing the tournament selection algorithm with roulette wheel selection, which was described in `Chapter 2`, *Understanding the Key Components of Genetic Algorithms*. This is done as follows:

```
toolbox.register("select", tools.selRoulette)
```

This change seems to have an adverse effect on the algorithm's results. As the following plot shows, there are numerous points in time where the best solution is forgotten as a result of the selection, and the max fitness value decreases, at least temporarily, although the average fitness value keeps increasing. This is because the roulette selection algorithm selects individuals with a probability proportionate to their fitness; when the differences between the individuals are relatively small, there is a better chance for weaker individuals to be selected, in comparison to the tournament selection we had before:



Stats of the program when using roulette wheel selection

To compensate for this behavior, we can use the elitist approach mentioned in `Chapter 2`, *Understanding the Key Components of Genetic Algorithms*. This approach allows a certain number of the best individuals from the current generation to carry over to the next generation unaltered, and prevents them from being lost. In the next chapter, we will explore applying the elitist approach when using the DEAP library.

Summary

In this chapter, you were introduced to DEAP—a versatile evolutionary computation framework that will be used in the rest of this book to solve real-life problems using genetic algorithms. You learned about DEAP's `creator` and `toolbox` modules, and how to use them to create the various components needed for the genetic algorithm flow. DEAP was then used to write two versions of a Python program that solves the OneMax problem, the first with full implementation of the genetic algorithm flow, and the other—more concise—taking advantage of the built-in algorithms of the framework. A third version of the program introduced the **hall-of-fame** (**HOF**) feature offered by DEAP. We then experimented with various settings of the genetic algorithm, and discovered the effects of changing the population size, as well as modifying the selection, crossover, and mutation operators.

In the next chapter, expanding on what we learned in this chapter, we will start solving real-life combinatorial problems, including the traveling salesman problem and the vehicle routing problem, using DEAP-based Python programs.

Further reading

For more information, please refer to the following resources:

- **DEAP documentation**: `https://deap.readthedocs.io/en/master/`
- **DEAP source code on GitHub**: `https://github.com/DEAP/deap`

4
Combinatorial Optimization

In this chapter, you will learn how genetic algorithms can be utilized in combinatorial optimization applications. We will start by describing search problems and combinatorial optimization, and outline several hands-on examples of combinatorial optimization problems. We will then analyze each of these problems and match them with a Python-based solution using the DEAP framework. The optimization problems we will cover are the well-known knapsack problem, the **traveling salesman problem** (**TSP**), and the **vehicle routing problem** (**VRP**). As a bonus, we will cover the topics of genotype-to-phenotype mapping and exploration versus exploitation.

In this chapter, you will do the following:

- Understand the nature of search problems and combinatorial optimization
- Solve the knapsack problem using a genetic algorithm coded with the DEAP framework
- Solve the TSP using a genetic algorithm coded with the DEAP framework
- Solve the VRP using a genetic algorithm coded with the DEAP framework
- Understand genotype-to-phenotype mapping
- Gain familiarity with the concept of exploration versus exploitation and its relation to elitism

Technical requirements

In this chapter, we will be using Python 3 with the following supporting libraries:

- `deap`
- `numpy`
- `matplotlib`
- `seaborn`

In addition, we will be using the benchmark data from the Rosetta Code Knapsack problem/0-1 (`http://rosettacode.org/wiki/Knapsack_problem/0-1`), and the TSP LIB (`http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/`).

The programs used in this chapter can be found in the book's GitHub repository at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter04`.

Check out the following video to see the Code in Action:
`http://bit.ly/2tfyfMI`

Search problems and combinatorial optimization

One main area of applying genetic algorithms is search problems, which have important applications in fields such as logistics, operations, artificial intelligence, and machine learning. Examples include determining the optimal routes for package delivery, designing hub-based airline networks, managing investment portfolios, and assigning passengers to available drivers in a fleet of taxis.

A search algorithm is focused on solving a problem through methodic evaluation of states and state transitions, aiming to find a path from the initial state to a desirable final (or goal) state. Typically, there is a cost or gain involved in every state transition, and the objective of the corresponding search algorithm is to find a path that minimizes the cost or maximizes the gain. Since the optimal path is one of many possible ones, this kind of search is related to combinatorial optimization, a topic that involves finding an optimal object from a finite, yet often extremely large, set of possible objects.

These concepts will be illustrated as we get acquainted with the knapsack problem, which is the main focus of the next section.

Solving the knapsack problem

Think of the familiar situation of packing for a long trip. There are many items that you would like to take with you, but you are limited by the capacity of your suitcase. In your mind, each item has a certain value it will add to your trip; at the same time, each has a size (and weight) associated with it, and each will compete with other items over the available space in your suitcase. This situation is just one of many real-life examples of the knapsack problem, which is considered one of the oldest and most investigated combinatorial search problems.

More formally, the knapsack problem consists of the following components:

- A set of items, each of them associated with a certain value and a certain weight
- A bag/sack/container (the knapsack) of a certain weight capacity

Our goal is to come up with a group of selected items that will provide the maximum total value, without exceeding the total weight capacity of the bag.

In the context of search algorithms, each subset of the items represents a *state,* and the set of all possible item subsets is considered the *state space*. For an instance of the knapsack 0-1 problem with n items, the size of the state space is 2^n, which can quickly grow very large, even for a modest value of n.

In this (original) version of the problem, each item can only be included once or not at all, and therefore it is sometimes referred to as the knapsack 0-1 problem. However, it can be expanded into other variants, for example, where items can be included multiple times (limited or unlimited), or where multiple knapsacks with varying capacities are present.

Applications of knapsack problems appear in many real-world processes that involve resource allocation and decision making, such as the selection of investments when building an investment portfolio, minimizing waste when cutting raw materials, and getting the most bang for your buck when selecting which questions to answer in a timed test.

To get our hands dirty with a knapsack problem, we will now look into a widely known example.

The Rosetta Code knapsack 0-1 problem

The Rosetta Code website (`rosettacode.org`) provides a collection of programming tasks, each with contributed solutions in numerous languages. One of these tasks, described at `rosettacode.org/wiki/Knapsack_problem/0-1`, is a knapsack 0-1 problem where a tourist needs to decide which items to pack for his weekend trip. The tourist has 22 items he can choose from; each item was assigned by the tourist with some value that represents its relative importance for the upcoming journey.

The weight capacity of the tourist's bag in this problem is 400, and the list of items, along with their associated values and weights, is as follows:

Item	Weight	Value
map	9	150
compass	13	35
water	153	200
sandwich	50	160
glucose	15	60
tin	68	45
banana	27	60
apple	39	40
cheese	23	30
beer	52	10
suntan cream	11	70
camera	32	30
T-shirt	24	15
trousers	48	10
umbrella	73	40
waterproof trousers	42	70
waterproof overclothes	43	75
note-case	22	80
sunglasses	7	20
towel	18	12
socks	4	50
book	30	10

Before we start solving this problem, we need to discuss one important matter—how will a potential solution be represented?

Solution representation

When solving the knapsack 0-1 problem, a straightforward way to represent a solution is using a list of binary values. Every entry in that list corresponds to one of the items in the problem. For the Rosetta Code problem, then, a solution can be represented using a list of 22 integers of the values 0 or 1. A value of 1 represents picking the corresponding item, while a value of 0 means that the item has not been picked. When applying the genetic algorithms approach, this list of binary values is going to be used as the chromosome.

We have to remember, however, that the total weight of the chosen items cannot exceed the capacity of the knapsack. One way to incorporate this restriction into the solution is to wait until it gets evaluated. We then evaluate by adding the weights of the chosen items one by one, while ignoring any chosen item that will cause the accumulated weight to exceed the maximum allowed value. From the genetic algorithm point of view, this means that the chromosome representation of an individual (**genotype**) may not entirely express itself when it gets translated into the actual solution (**phenotype**), as some of the 1 values in the chromosome may be ignored. This situation is sometimes referred to as *genotype to phenotype mapping*.

The solution representation we just discussed is implemented in the Python class described in the next subsection.

Python problem representation

To encapsulate the Rosetta Code knapsack 0-1 problem, we created a Python class called `Knapsack01Problem`. This class is contained in the `knapsack.py` file, which can be found at the following link:

`https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter04/knapsack.py`

The class provides the following methods:

- `__init_data()`: Initializes the `RosettaCode.org` knapsack 0-1 problem data by creating a list of tuples. Each tuple contains the name of an item, followed by its weight and its value.

- `getValue(zeroOneList)`: Calculates the value of the chosen items in the list, while ignoring items that will cause the accumulating weight to exceed the maximum weight.
- `printItems(zeroOneList)`: Prints the chosen items in the list, while ignoring items that will cause the accumulating weight to exceed the maximum weight.

The main method of the class creates an instance of the `Knapsack01Problem` class. It then creates a random solution and prints out its relevant information. If we run this class as a standalone Python program, a sample output may look as follows:

```
Random Solution =
[1 1 1 1 1 0 0 0 0 1 1 1 0 1 0 0 0 1 0 0 0 0]
- Adding map: weight = 9, value = 150, accumulated weight = 9, accumulated
value = 150
- Adding compass: weight = 13, value = 35, accumulated weight = 22,
accumulated value = 185
- Adding water: weight = 153, value = 200, accumulated weight = 175,
accumulated value = 385
- Adding sandwich: weight = 50, value = 160, accumulated weight = 225,
accumulated value = 545
- Adding glucose: weight = 15, value = 60, accumulated weight = 240,
accumulated value = 605
- Adding beer: weight = 52, value = 10, accumulated weight = 292,
accumulated value = 615
- Adding suntan cream: weight = 11, value = 70, accumulated weight = 303,
accumulated value = 685
- Adding camera: weight = 32, value = 30, accumulated weight = 335,
accumulated value = 715
- Adding trousers: weight = 48, value = 10, accumulated weight = 383,
accumulated value = 725
- Total weight = 383, Total value = 725
```

Note that the last occurrence of `1` in the random solution, representing the item note-case, fell victim to the genotype to phenotype mapping discussed in the previous subsection. As this item's weight is 22, it would cause the total weight to exceed 400, and as a result—this item was not included in the solution.

This random solution, as one may expect, is far from being optimal. Let's try and find the optimal solution for this problem using a genetic algorithm.

Genetic algorithms solution

To solve our knapsack 0-1 problem using a genetic algorithm, we created the `01-solve-knapsack.py` Python program located at the following URL:

`https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter04/01-solve-knapsack.py`

As a reminder, the chromosome representation we decided to use here is a list of integers with the values of 0 or 1. This makes our problem, from the point of view of the genetic algorithm, similar to the OneMax problem we solved in the previous chapter. The genetic algorithm does not care what the chromosome represents (the phenotype)—a list of items to pack, some Boolean equation coefficients, or perhaps just some binary number—it is only concerned with the chromosome itself (the genotype) and the fitness value of that chromosome. Mapping the chromosome to the solution it represents is carried out by the fitness evaluation function, which is implemented outside the genetic algorithm. In our case, this chromosome mapping and fitness calculation is implemented by the `getValue()` method, which is encapsulated within the `Knapsack01Problem` class.

The outcome of all this is that we can use the same genetic algorithm implementation as we used for the OneMax problem, with a few adaptations.

The following steps describe the main points of our solution:

1. First, we need to create an instance of the knapsack problem we would like to solve:

```
knapsack = knapsack.Knapsack01Problem()
```

2. We then instruct the genetic algorithm to use the `getValue()` method of that instance for fitness evaluation:

```
def knapsackValue(individual):
    return knapsack.getValue(individual),

toolbox.register("evaluate", knapsackValue)
```

3. The genetic operators used are compatible with the binary-list chromosome:

```
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit,
indpb=1.0/len(knapsack))
```

4. And once the genetic algorithm stops, we can use the `printItems()` method to pretty print the best solution found:

```
best = hof.items[0]

print("-- Knapsack Items = ")
knapsack.printItems(best)
```

5. We can also tweak some of the parameters of the genetic algorithm. As this particular problem uses a binary string of length 22, it seems easier than the 100-length OneMax problem we previously solved, so we can probably reduce the population size and the max number of generations.

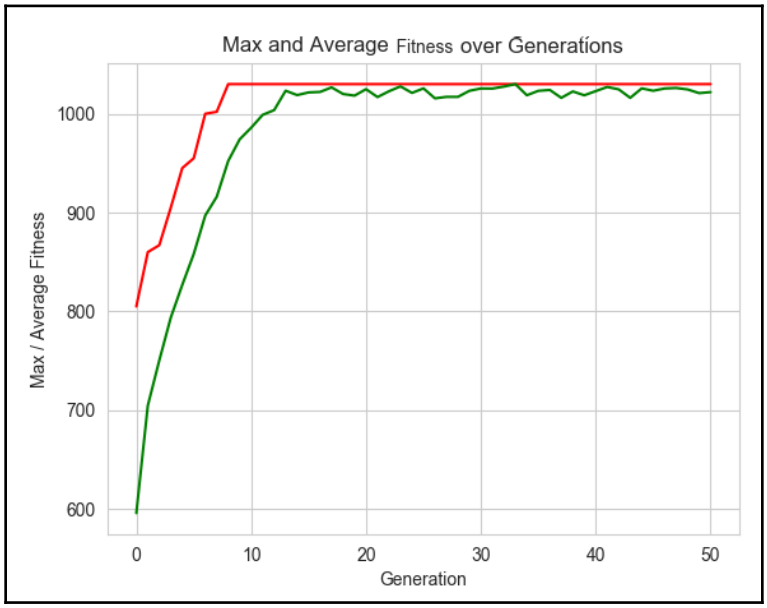
Running the algorithm for 50 generations, with a population size of 50, we get the following outcome:

```
-- Best Ever Individual = [1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0,
0, 0, 0, 1, 1, 1, 1, 0, 1, 1]
-- Best Ever Fitness = 1030.0
-- Knapsack Items =
- Adding map: weight = 9, value = 150, accumulated weight = 9,
accumulated value = 150
- Adding compass: weight = 13, value = 35, accumulated weight =
22, accumulated value = 185
- Adding water: weight = 153, value = 200, accumulated weight =
175, accumulated value = 385
- Adding sandwich: weight = 50, value = 160, accumulated weight
= 225, accumulated value = 545
- Adding glucose: weight = 15, value = 60, accumulated weight =
240, accumulated value = 605
- Adding banana: weight = 27, value = 60, accumulated weight =
267, accumulated value = 665
- Adding suntan cream: weight = 11, value = 70, accumulated
weight = 278, accumulated value = 735
- Adding waterproof trousers: weight = 42, value = 70,
accumulated weight = 320, accumulated value = 805
- Adding waterproof overclothes: weight = 43, value = 75,
accumulated weight = 363, accumulated value = 880
- Adding note-case: weight = 22, value = 80, accumulated weight
= 385, accumulated value = 960
- Adding sunglasses: weight = 7, value = 20, accumulated weight
= 392, accumulated value = 980
- Adding socks: weight = 4, value = 50, accumulated weight =
396, accumulated value = 1030
- Total weight = 396, Total value = 1030
```

The total value of `1030` is indeed the known optimal solution for this problem.

Here, too, we can see that the last occurrence of `1` in the chromosome of the best individual, representing the item `book`, was sacrificed in the mapping to the actual solution, to keep the accumulated weight from exceeding the limit of 400.

The graph depicting the max and average fitness over the generations, shown here, indicates that the best solution was found in less than 10 generations:



Stats of the program solving the knapsack 0-1 problem

In the next section, we will shift gears and tackle a more involved, yet still classic, combinatorial search task known as the TSP.

Solving the TSP

Imagine that you manage a small fulfillment center and need to deliver packages to a list of customers using a single vehicle. What is the best route for the vehicle to take so that you visit all your customers and then return to the starting point? This is an example of the classic TSP.

The TSP dates back to 1930, and since then has been and is one of the most thoroughly studied problems in optimization. It is often used to benchmark optimization algorithms. The problem has many variants, but it was originally based on a traveling salesman that needs to take a trip covering several cities:

> *"Given a list of cities and the distances between each pair of the cities, find the shortest possible path that goes through all the cities, and returns to the starting city."*

Using combinatorics, you could find that when given n cities, the number of possible paths that go through all cities is $(n - 1)!/2$.

The following screenshot represents the shortest path for the traveling salesperson problem that covers the 15 largest cities in Germany:

The shortest TSP path for the 15 largest cities in Germany
Source: https://commons.wikimedia.org/wiki/File:TSP_Deutschland_3.png
Image by Kapitän Nemo. Released to public domain

As in this case $n=15$, the number of possible routes is $14!/2$, which calculates to the staggering number of 43,589,145,600.

In the context of search algorithms, each path (or partial path) through the cities represents a *state*, and the set of all possible paths is considered the *state space*. Each of the paths has a corresponding cost—the length (distance) of the path—and we are looking for the path that will minimize this distance.

As we pointed out, the state space is very large even for a moderate number of cities, which can make it prohibitively expensive to evaluate each and every possible path. As a result, even though it is relatively easy to find a path that goes through all the cities, finding the optimal path can be very hard.

TSPLIB benchmark files

TSPLIB is a library containing sample problems for the TSP based on actual geographic locations of cities. The library is maintained by the Heidelberg University, and relevant examples can be found here: `http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/`.

The optimal distances for each problem can be found here: `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html`.

The TSPLIB examples are contained in text-based, whitespace-delimited data files. A typical file contains several informational lines followed by the city data. We are interested in files that contain the *x, y* coordinates of the participating cities, so we can plot the cities and visualize their locations. For example, the `burma14.tsp` file looks like the following (with some of the lines omitted here for brevity):

```
NAME: burma14
TYPE: TSP
...
NODE_COORD_SECTION
   1  16.47       96.10
   2  16.47       94.44
   3  20.09       92.54
   ...
  12  21.52       95.59
  13  19.41       97.13
  14  20.09       94.55
EOF
```

The interesting section for us is the lines between `NODE_COORD_SECTION` and `EOF`. In some of the files, `DISPLAY_DATA_SECTION` is used instead of `NODE_COORD_SECTION`.

Are we ready to solve a sample problem? Well, before we start doing that, we still need to figure out how a potential solution will be represented. This will be addressed in the next subsection.

Solution representation

When solving the TSP, the cities are typically represented by numbers from 0 to *n*-1, and possible solutions will be sequences of these numbers. A problem with five cities, for example, can have solutions of the form [0, 1, 2, 3, 4], [2, 4, 3, 1, 0], and so on. Each solution can be evaluated by calculating and totaling the distances between each two subsequent cities, then adding the distance between the last city to the first one. Consequently, when applying the problem using the genetic algorithms approach, we can use a similar list of integers to serve as the chromosome.

The Python class described in the next subsection reads the contents of TSPLIB files and calculates the distances between each pair of cities. In addition, it calculates the total distance covered by a given potential solution, using the list representation we just discussed.

Python problem representation

To encapsulate the TSP, we created a Python class called `TravelingSalesmanProblem`. This class is contained in the `tsp.py` file, which can be found at the following link:

`https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter04/tsp.py`

The class provides the following private methods:

- `__create_data()`: Reads the desired TSPLIB file off the internet, extracts the city coordinates, calculates the distances between every two cities, and uses them to populate a distance matrix (two-dimensional array). It then serializes the city locations and serializes the calculated distances to disk using the `pickle` utility.
- `__read_data()`: Reads the serialized data, and if not available, calls `__create_data()` to prepare it.

These methods are internally invoked by the constructor, so the data is initialized as soon as the instance is created.

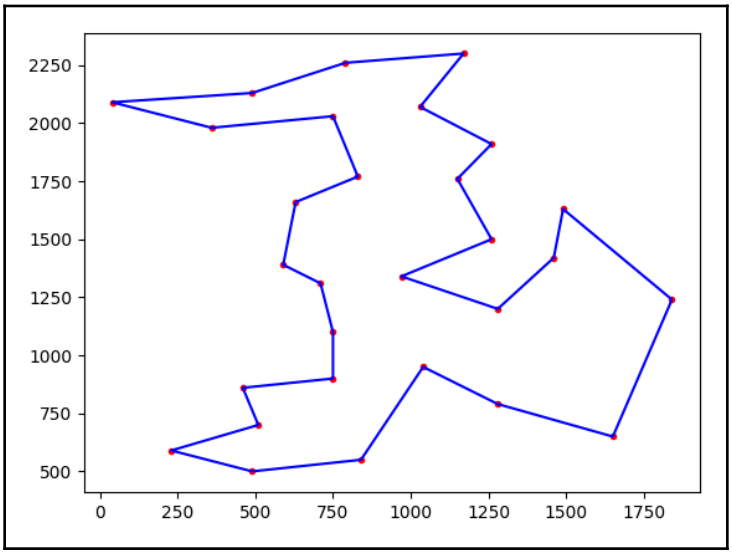
In addition, the class provides the following public methods:

- `getTotalDistance(indices)`: Calculates the total distance of the path described by the given indices of the cities
- `plotData(indices)`: Plots the path described by the given indices of the cities

The main method of the class exercises the class methods just mentioned: it first creates the `bayg29` problem (29 cities in Bavaria), then calculates the distance for the optimal solution (as described in `http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/bayg29.opt.tour`), and finally plots it. So, if we run this class as a standalone Python program, the output is as follows:

```
Problem name: bayg29
Optimal solution = [0, 27, 5, 11, 8, 25, 2, 28, 4, 20, 1, 19, 9, 3, 14, 17,
13, 16, 21, 10, 18, 24, 6, 22, 7, 26, 15, 12, 23]
Optimal distance = 9074.147
```

The plot of the optimal solution looks as follows:



A plot of the optimal solution for the bayg29 TSP. The red dots represent the cities

Next, we will try to reach this optimal solution using a genetic algorithm.

Genetic algorithms solution

For our first attempt to solve the TSP using a genetic algorithm, we created the Python program `02-solve-tsp-first-attempt.py`, located at the following URL:

`https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter04/02-solve-tsp-first-attempt.py`

The main parts of our solution are described by the following steps:

1. The program starts by creating an instance of the `"bayg29"` problem as follows:

```
TSP_NAME = "bayg29"
tsp = tsp.TravelingSalesmanProblem(TSP_NAME)
```

2. Next, we need to define the fitness strategy. Here we want to minimize the distance, which translates to a single-objective minimizing `Fitness` class, defined using a single negative weight:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

3. As we discussed a little earlier, our choice of chromosome for the genetic algorithm is a list of integers from 0 to *n*-1, where *n* is the number of cities, representing the city indices. As an example, the optimal solution we have seen earlier for the `bayg29` problem was represented with the following chromosome:

(0, 27, 5, 11, 8, 25, 2, 28, 4, 20, 1, 19, 9, 3, 14, 17, 13, 16, 21, 10, 18, 24, 6, 22, 7, 26, 15, 12, 23)

The following code snippet is responsible for implementing this chromosome. It is further explained as follows:

```
creator.create("Individual", array.array, typecode='i',
fitness=creator.FitnessMin)
toolbox.register("randomOrder", random.sample, range(len(tsp)),
len(tsp))
toolbox.register("individualCreator", tools.initIterate,
creator.Individual, toolbox.randomOrder)
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

The `Individual` class is created first, extending an array of integers and augmenting it with the `FitnessMin` class.

The `randomOrder` operator is then registered to provide the results of the `random.sample()` invocation over a range defined by the length of the TSP problem (the number of cities, or *n*). This will result in a randomly generated list of indices between 0 and *n*-1.

The `IndividualCreator` operator is created next. When called, it will, in turn, invoke the `randomOrder` operator and iterate over the results, to create a valid chromosome consisting of the city indices.

The last operator, `populationCreator`, is then created to produce a list of individuals using the `IndividualCreator` operator.

4. Once the chromosome was implemented, it is time to define the fitness evaluation function. This is carried out by the `tspDistance()` function, which directly utilizes the `getTotalDistance()` method of the `TravelingSalesmanProblem` class:

```
def tpsDistance(individual):
    return tsp.getTotalDistance(individual),  # return a tuple

toolbox.register("evaluate", tpsDistance)
```

5. Next, we need to define the genetic operators. For the selection operator, we can use tournament selection with a tournament size of `3`, as we did in previous cases:

```
toolbox.register("select", tools.selTournament, tournsize=3)
```

6. However, before picking the crossover and mutation operators, we need to remember that the chromosome we use is not just a list of integers, but a list of indices that represent the order of the cities, and therefore we cannot just mix parts of two lists together, or arbitrarily change an index in the list. Instead, we need to use specialized operators that were designed to produce valid lists of indices. In `Chapter 2`, *Understanding the Key Components of Genetic Algorithms*, we examined one of these operators, ordered crossover. Here, we use the DEAP implementation of this operator:

```
toolbox.register("mate", tools.cxOrdered)
toolbox.register("mutate", tools.mutShuffleIndexes,
indpb=1.0/len(tsp))
```

7. Finally, it is time to invoke the genetic algorithm flow. We use here the default DEAP built-in `eaSimple` algorithm, with our default `stats` and `halloffame` objects to provide information we can later use to display:

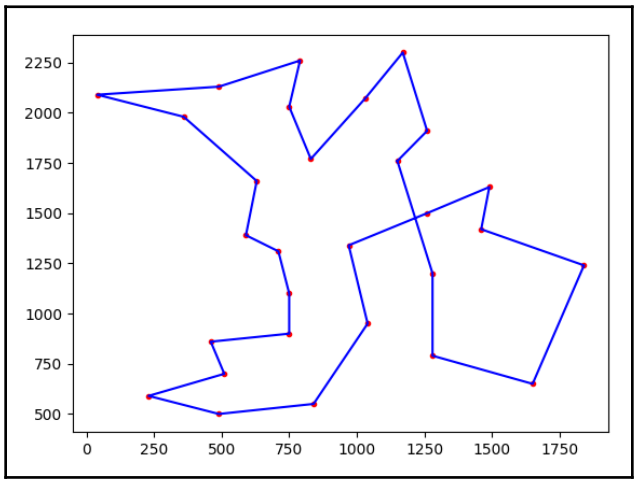
```
population, logbook = algorithms.eaSimple(population, toolbox,
                                           cxpb=P_CROSSOVER,
                                           mutpb=P_MUTATION,
                                           ngen=MAX_GENERATIONS,
                                           stats=stats,
                                           halloffame=hof,
                                           verbose=True)
```

Running this program with the constant values as they appear at the top of the file (population size of 300, 200 generations, crossover probability of 0.9, and mutation probability of 0.1) yields the following results:

```
-- Best Ever Individual = Individual('i', [0, 27, 11, 5, 20, 4, 8, 25, 2,
28, 1, 19, 9, 3, 14, 17, 13, 16, 21, 10, 18, 12, 23, 7, 26, 22, 6, 24, 15])
-- Best Ever Fitness = 9549.9853515625
```

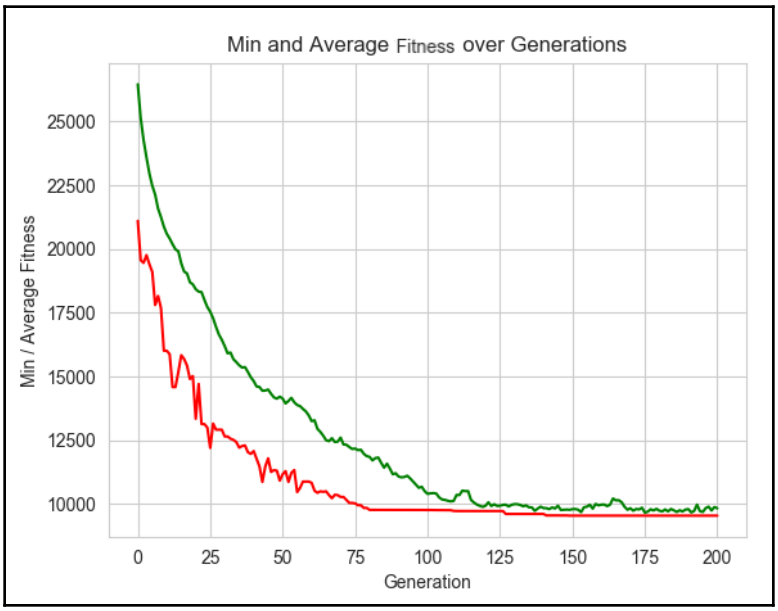
The best fitness found (9549.98) is not too far from the known optimal distance of 9074.14.

The program then produces two plots. The first plot illustrates the path of the best individual found during the run:



A plot of the best solution found by the first program attempting to solve the bayg29 TSP

The second plot shows the statistics of the genetic flow. Note that this time we chose to collect data for the minimum fitness value rather than the maximum, as the objective of this problem is to minimize the distance:

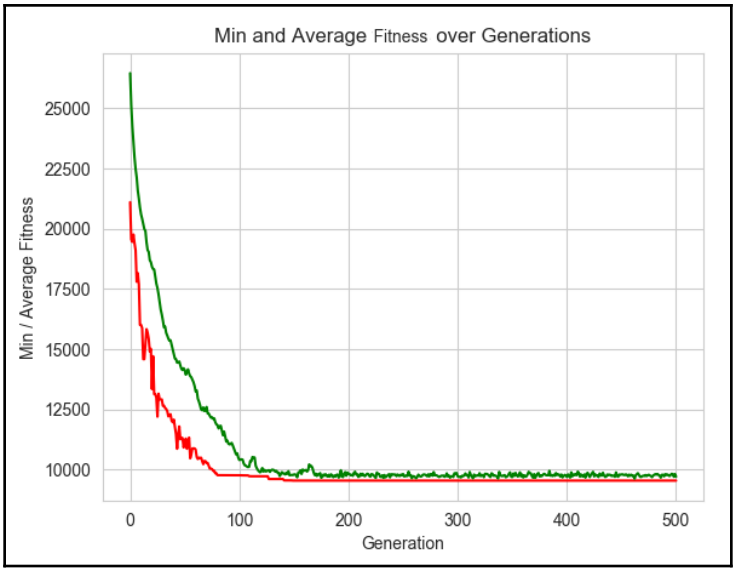


Stats of the first program attempting to solve the bayg29 TSP

As we found a good solution but not the best-known one, we can try and figure out ways to improve the results. For example, we can experiment with changing the population size, number of generations, and the probabilities. We can also replace the genetic operators with other compatible ones. We can even change the random seed we set just to see the effect on the results, or make multiple runs with different seeds. In the next section, we will try, instead, to use elitism combined with enhanced exploration to improve our results.

Improving the results with enhanced exploration and elitism

If we try to increase the number of generations in the previous program, we will realize that the solution does not improve—it is stuck in the (somewhat) suboptimal solution that was reached sometime before generation 200, as shown in the next plot, displaying 500 generations:



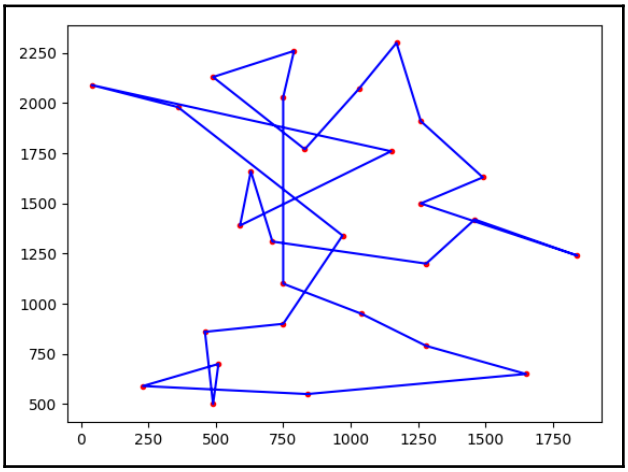
Stats of the first program, running for 500 generations

From that point on, the similarity between the average value and the best value indicates that this solution took over the population and therefore we will not see any improvement unless a lucky mutation turns up. In genetic algorithm terms, this means that **exploitation** has overpowered **exploration.** Exploitation generally means taking advantage of the currently available results, while exploration emphasizes the search for new solutions. Striking a delicate balance between the two can lead to better results.

One way to increase exploration could be reducing the tournament size of the tournament selection used from 3 to 2:

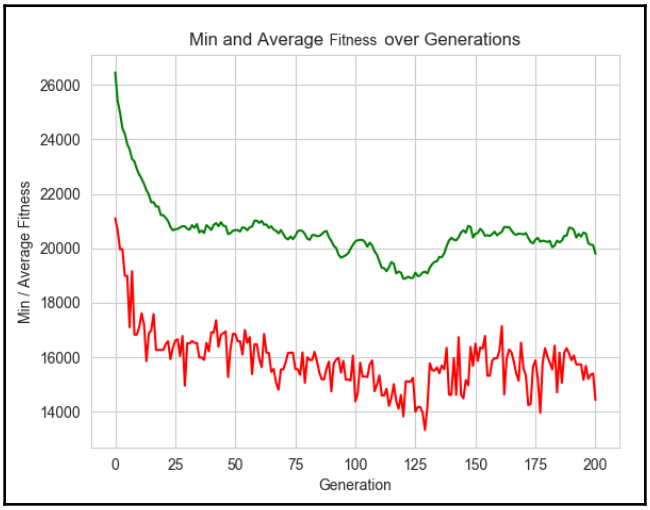
```
toolbox.register("select", tools.selTournament, tournsize=2)
```

As was discussed in `Chapter 2`, *Understanding the Key Components of Genetic Algorithms*, this will increase the chances of less successful individuals being selected. These individuals may carry the key to better future solutions. However, if we run the same program after making this change, the results are far from impressive: the best fitness value is over 13,000, and the best solution plot looks as follows:



A plot of the best solution found by the program with tournament size reduced to two

The poor results can be explained using the statistics plot:



Stats of the program with tournament size reduced to two

This plot illustrates that we cannot retain the best solutions. As is evident from the noisy graph that keeps jumping between better values and worse values, good solutions tend to quickly get lost due to the more permissive selection scheme that often enables lesser solutions to be selected. This means that we let exploration go too far, and to balance it out we need to re-introduce a measure of exploitation into the mix. This can be done using the **elitism** mechanism, which was first introduced in `Chapter 2`, *Understanding the Key Components of Genetic Algorithms*.

Elitism enables us to keep the best solutions intact by letting them skip the genetic operators of selection, crossover, and mutation during the genetic flow. To implement elitism, we will have to go under the hood and modify DEAP's `algorithms.eaSimple()` algorithm, as the framework does not provide a direct way to skip all three operators.

The modified algorithm, called `eaSimpleWithElitism()`, can be found in the `elitism.py` file, located at the following link:

`https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter04/elitism.py`

The `eaSimpleWithElitism()` method is similar to the original `eaSimple()`, with the modification that the `halloffame` object is now used to implement an elitism mechanism. The individuals contained in the `halloffame` object are directly injected into the next generation and are not subject to the genetic operators of selection, crossover, and mutation. This is essentially the outcome of the following modifications:

- Instead of selecting a number of individuals equal to the population size, this number of selected individuals is reduced by the number of hall of fame individuals:

```
offspring = toolbox.select(population, len(population) - hof_size)
```

- After the genetic operators have been applied, the hall of fame individuals are added back into the population:

```
offspring.extend(halloffame.items)
```

We can now replace the call to `algorithms.eaSimple()` with a call to `elitism.eaSimpleWithElitism()`, without changing any of the parameters. We then set the `HALL_OF_FAME_SIZE` constant to `30`, which means that we will always keep the best 30 individuals in the population.

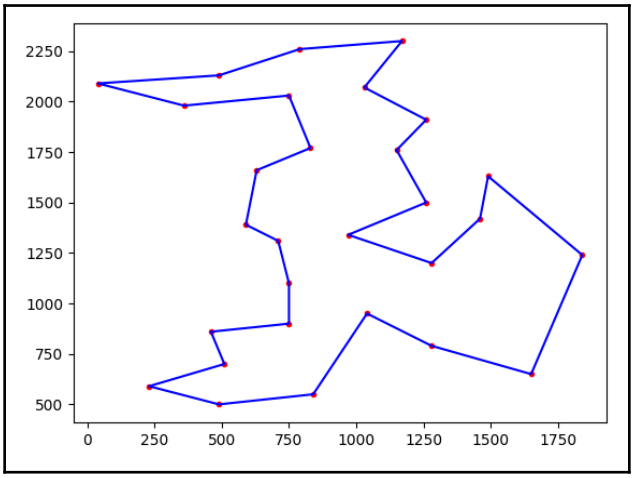
The modified Python program, `03-solve-tsp.py`, can be found at the following URL:

`https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter04/03-solve-tsp.py`

Running this new program, we are now able to hit the optimal solution:

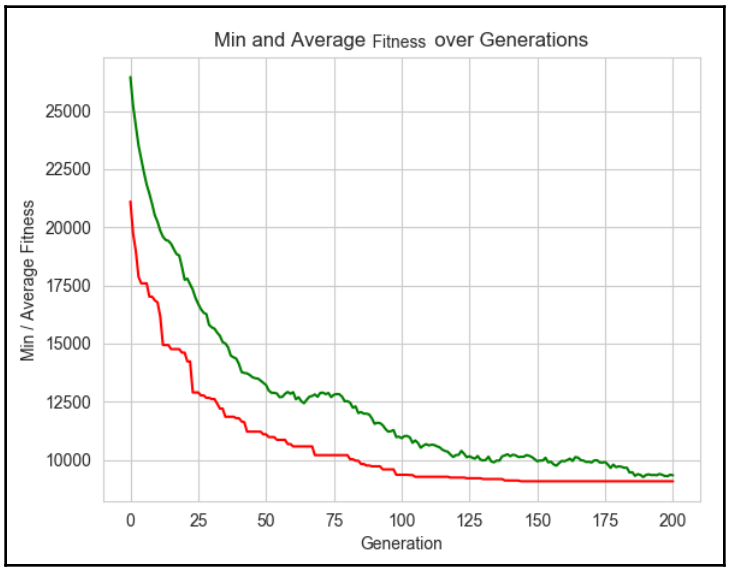
```
-- Best Ever Individual = Individual('i', [0, 23, 12, 15, 26, 7, 22, 6, 24,
18, 10, 21, 16, 13, 17, 14, 3, 9, 19, 1, 20, 4, 28, 2, 25, 8, 11, 5, 27])
-- Best Ever Fitness = 9074.146484375
```

The solution plot is identical to the optimal one we have seen before:



Plot of the best solution found by the program using a tournament size of two and elitism

The statistics plot, shown here, indicates that we were able to eliminate the noise we observed before. We were also able to keep some distance between the average value and the best values for a lot longer, compared to the original attempt:



Stats of the program using a tournament size of two and elitism

In the next section, we will look into the **vehicle routing problem** (**VRP**), which adds an interesting twist to the problem we just solved.

Solving the VRP

Imagine that you now manage a larger fulfillment center. You still need to deliver packages to a list of customers, but now you have a fleet of several vehicles at your disposal. What is the best way to deliver the packages to the customers using these vehicles?

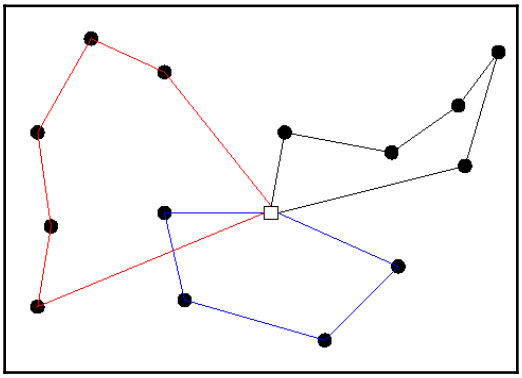
This is an example of VRP, a generalization of the TSP described in the previous section. The basic VRP consists of the following three components:

- The list of locations that need to be visited
- The number of vehicles
- The location of the depot, which is used as the starting and ending point for each one of the vehicles

The problem has numerous variations, such as several depot locations, time-critical deliveries, different types of vehicles (with varying capacity and varying fuel consumption, for instance), and many more.

The goal of the problem is to minimize the cost, which can also be defined in many different ways; for example, minimizing the time it takes to deliver all the packages, minimizing the cost of the fuel, or minimizing the variation in travel time among the vehicles used.

An illustration of a VRP with three vehicles is shown here. The cities are marked with dark circles and the depot location is marked with an empty square, while the routes of the three vehicles are marked with three different colors:



Example VRP with three vehicles
Source: https://commons.wikimedia.org/wiki/File:Figure_illustrating_the_vehicle_routing_problem.png
Image by PierreSelim. Released to public domain

In our example, we will aim to optimize the time it takes to deliver all the packages. Since all the vehicles operate simultaneously, this measure is determined by the vehicle making the longest route. We can, therefore, make it our objective to minimize the length of the longest route among the participating vehicles' routes. For example, if we have three vehicles, each solution consists of three routes. We will evaluate all three, and then only consider the longest one of them for scoring—the longer the route, the worse the score. This will inherently encourage all three routes to be shorter, as well as closer in size to each other.

Thanks to the similarity between the two problems, we can utilize the code we have previously written to solve TSP for solving VRP.

To build on the solution we created for TSP, we can represent the vehicle routing as follows:

- A TSP instance, namely a list of cities and their coordinates (or their mutual distances)
- The depot location, which is selected out of the existing cities and represented by the index of that city
- The number of vehicles used

In the next two subsections, we will show how to implement this solution.

Solution representation

As usual, the first question we now need to address is how to represent a solution to this problem. A creative way to do so while keeping the similarity to the TSP that we solved before is to represent a solution using a list that contains the numbers from 0 to (n-1) + (m - 1), where n is the number of cities in TSP and m is the number of vehicles. For example, if the number of cities is 10 and the number of vehicles is 3 (n = 10, m = 3), we will have a list containing all the integers from 0 to 11:

(1, 3, 4, 6, 11, 9, 7, 2, 10, 5, 8, 0)

The first n integers, 0 to 9 in our case, still represent the cities, just like before. However, the last (m - 1) integers, 10 and 11 in our case, will be used as delimiters (or separators) that break the list into routes. For example: (1, 3, 4, 6, **11**, 9, 7, 2, **10**, 5, 8, 0) will be broken into the following three routes:

(1, 3, 4, 6), (9, 7, 2), (5, 8, 0)

Next, the index of the depot location needs to be removed, since it is not part of a particular route. If, for example, the depot location is the index 7, the resulting routes will be as follows:

(1, 3, 4, 6), (9, 2), (5, 8, 0)

When calculating the distance that each route covers, we need to recall that each route starts and ends at the depot location (7). So, for the purpose of calculating the distances, as well as plotting the routes, we will be using the following data:

(7, 1, 3, 4, 6, 7), (7, 9, 2, 7), (7, 5, 8, 0, 7)

In the next subsection, we will look into a Python implementation of this idea.

Python problem representation

To encapsulate the VRP problem, we created a Python class called `VehicleRoutingProblem`. This class is contained in the `vrp.py` file, which can be found at the following URL:

`https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter04/vrp.py`

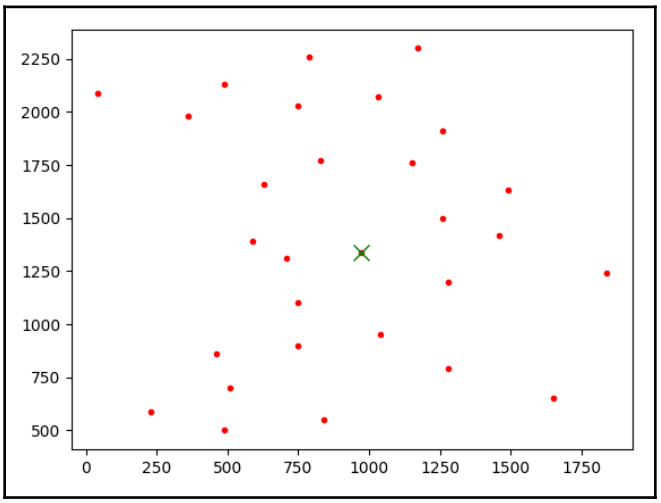
The `VehicleRoutingProblem` class contains an instance of the `TravelingSalesmanProblem` class, which is used as the container for the city indices and their corresponding locations and distances. When creating an instance of the `VehicleRoutingProblem` class, the instance of the underlying `TravelingSalesmanProblem` is internally created and initialized.

The `VehicleRoutingProblem` class is initialized using the name of the underlying `TravelingSalesmanProblem`, as well as the depot location index and the number of vehicles.

In addition, the `VehicleRoutingProblem` class provides the following public methods:

- `getRoutes(indices)`: Breaks the list of given indices into separate routes, by detecting the separator indices
- `getRouteDistance(indices)`: Calculates the total distance of the path that starts at the depot location and goes through the cities described by the given indices
- `getMaxDistance(indices)`: Calculates the max distance among the distances of the various paths described by the given indices, after breaking the indices in to separate routes
- `getTotalDistance(indices)`: Calculates the combined distance of the various paths described by the given indices
- `plotData(indices)`: Breaks the list of indices into separate routes and plots each route in a different color

When executed as a standalone program, the main method of the class exercises these methods by creating an instance of `VehicleRoutingProblem` with the underlying TSP set to `"bayg29"`—the same problem we used in the previous section. The number of vehicles is set to 3, and the depot location index is set to 12 (which maps to a city with a central location). The following shows the locations of the cities (red dots) and the depot (green X):

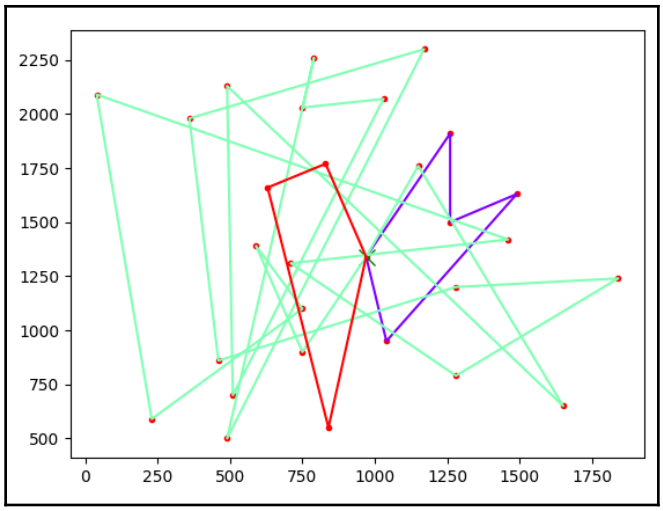


A plot of VRP based on the bayg29 TSP. Red dots mark the cities while the green X marks the depot

The main method then generates a random solution, breaks it down to routes, and calculates the distances, as shown in the following code snippet:

```
random solution = [27, 23, 7, 18, 30, 14, 19, 3, 16, 2, 26, 9, 24, 22, 15,
17, 28, 11, 21, 12, 8, 4, 5, 13, 25, 6, 0, 29, 10, 1, 20]
route breakdown = [[27, 23, 7, 18], [14, 19, 3, 16, 2, 26, 9, 24, 22, 15,
17, 28, 11, 21, 8, 4, 5, 13, 25, 6, 0], [10, 1, 20]]
total distance = 26653.845703125
max distance = 21517.686
```

Note how the original list of indices of the random solution is broken down to separate routes using the separator indices (29 and 30). The plot of this random solution is shown in the following screenshot:



A plot of a random solution for VRP with three vehicles

As we would expect from a random solution, it is far from optimal. This is evident from the inefficient order of cities along the long (green) route, as well as one route (green) being much longer than the other two (red and purple).

In the next subsection, we will attempt to produce good solutions using the genetic algorithms method.

Genetic algorithms solution

The genetic algorithm solution we created for VRP resides in the Python program `04-solve-vrp.py`, located at the following link:

`https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter04/04-solve-vrp.py`

Since we were able to build on top of TSP and used a similar representation for the solution—an array of indices—we could use the same genetic approach as we used in the previous section. We could also take advantage of elitism by reusing the elitist version that we created for the genetic flow. This makes our genetic algorithm solution very similar to the one used for TSP.

The following steps detail the main parts of our solution:

1. The program starts by creating an instance of the `VehicleRoutingProblem` class, using the `"bayg29"` TSP for its underlying data, and setting the depot location to `12` and the number of vehicles to `3`:

```
TSP_NAME = "bayg29"
NUM_OF_VEHICLES = 3
DEPOT_LOCATION = 12

vrp = vrp.VehicleRoutingProblem(TSP_NAME, NUM_OF_VEHICLES,
DEPOT_LOCATION)
```

2. The fitness function is set to minimize the distance of the longest route among the three routes produced by each solution:

```
def vrpDistance(individual):
    return vrp.getMaxDistance(individual),

toolbox.register("evaluate", vrpDistance)
```

3. For the genetic operators, we again use tournament selection with a tournament size of `2`, which is assisted by the elitist approach, and crossover and mutation operators that are specialized for ordered lists:

```
# Genetic operators:
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxUniformPartialyMatched,
indpb=2.0/len(vrp))
toolbox.register("mutate", tools.mutShuffleIndexes,
indpb=1.0/len(vrp))
```

4. As the VRP is inherently more difficult than the TSP, we chose a larger population size and number of generations than before:

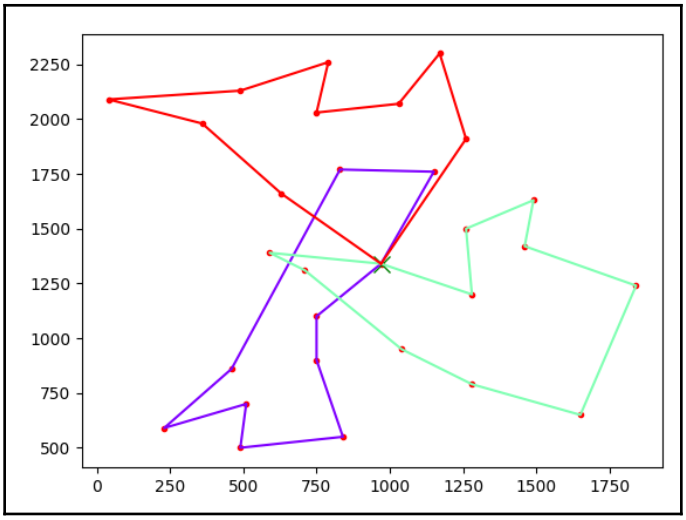
```
# Genetic Algorithm constants:
POPULATION_SIZE = 500
P_CROSSOVER = 0.9
P_MUTATION = 0.2
MAX_GENERATIONS = 1000
HALL_OF_FAME_SIZE = 30
```

And—that's it! We are ready to run the program. The results obtained with these settings are shown here—three routes, with a maximum length of 3857:

```
-- Best Ever Individual =  Individual('i', [0, 20, 17, 16, 13, 21, 10, 14,
3, 29, 15, 23, 7, 26, 12, 22, 6, 24, 18, 9, 19, 30, 27, 11, 5, 4, 8, 25, 2,
28, 1])
-- Best Ever Fitness =  3857.36376953125
-- Route Breakdown =  [[0, 20, 17, 16, 13, 21, 10, 14, 3], [15, 23, 7, 26,
22, 6, 24, 18, 9, 19], [27, 11, 5, 4, 8, 25, 2, 28, 1]]
-- total distance =  11541.875
-- max distance =  3857.3638
```

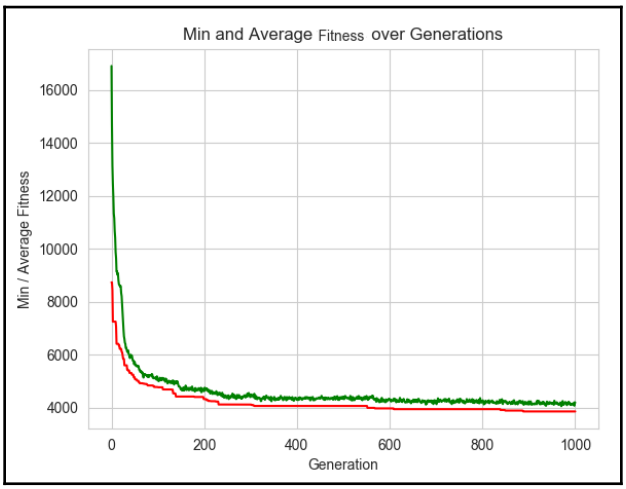
Note, again, how the solution is broken down into three separate routes using the highest two indices (29, 30) as separators, and ignoring the depot location (12). We ended up with three routes: two of them cover 9 cities each, and the third covers 10 cities.

Plotting the solution produces the following drawing, depicting the three resulting routes:



A plot of the best solution found by the program for the VRP with three vehicles

The following statistics plot shows that the algorithm did most of the optimization before reaching 300 generations, and afterward there were several small improvements:



Stats of the program solving VRP with three vehicles

How about changing the number of vehicles? Let's run the algorithm again after increasing the number of vehicles to six and making no other changes:

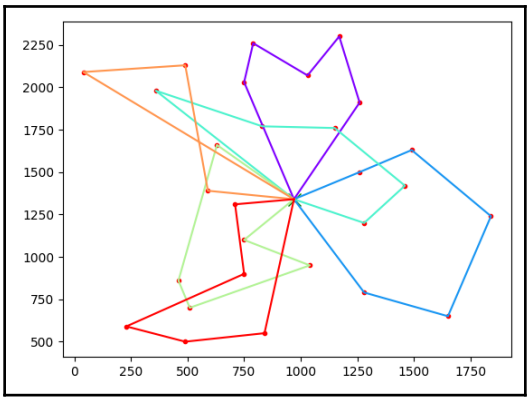
```
NUM_OF_VEHICLES = 6
```

The results for this run are shown here—six routes, with a maximum length of 2803:

```
-- Best Ever Individual =  Individual('i', [27, 11, 5, 8, 4, 33, 12, 24, 6,
22, 7, 23, 29, 28, 20, 0, 26, 15, 32, 3, 18, 13, 17, 1, 31, 19, 25, 2, 30,
9, 14, 16, 21, 10])
-- Best Ever Fitness =  2803.584716796875
-- Route Breakdown =  [[27, 11, 5, 8, 4], [24, 6, 22, 7, 23], [28, 20, 0,
26, 15], [3, 18, 13, 17, 1], [19, 25, 2], [9, 14, 16, 21, 10]]
-- total distance =  16317.9892578125
-- max distance =  2803.5847
```

Note that increasing the number of vehicles twofold did not decrease the maximum distance in a similar manner (2803 with 6 compared to 3857 with 3). This is likely due to the fact that each separate route still needs to start and end at the depot location, which is added to the cities in the route.

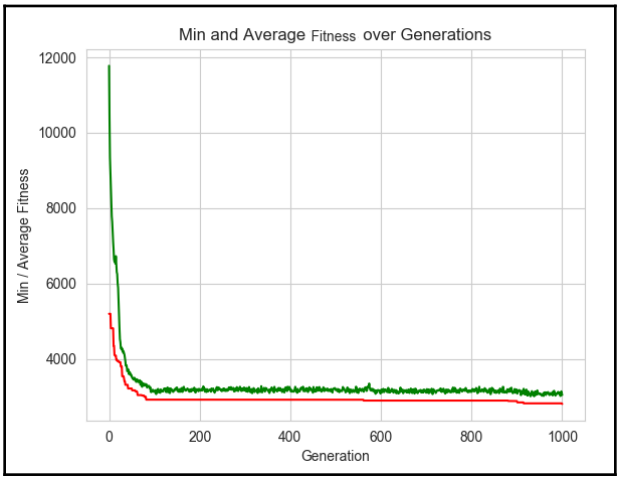
Plotting the solution produces the following drawing, depicting the six resulting routes:



A plot of the best solution found by the program for the VRP with six vehicles

One interesting point that arises for this plot is that the orange route does not seem optimized. Since we told the genetic algorithm to minimize the **longest** route, any route that is shorter than the longest route may not get further optimized. You are encouraged to modify our solution to further optimize the routes.

As with the three-vehicle case, the statistics plot here shows that the algorithm did most of the optimization before reaching 200 generations, and afterward there were several small improvements:



Stats of the program solving the VRP with six vehicles

The solution we find seems reasonable. Can we do better than that? What about other numbers of vehicles, or other depot locations? How about different genetic operators or different parameter settings—perhaps even different fitness criteria? We encourage you to experiment with all these and learn from these experiments.

Summary

In this chapter, you were first introduced to search problems and combinatorial optimization. We then examined closely three classic combinatorial problems—each with numerous real-life applications—the knapsack problem, the TSP, and the VRP. For each of these problems, we followed a similar process of finding an appropriate representation for a solution, creating a class that encapsulates the problem and evaluates a given solution, then creating a genetic algorithm solution that utilizes that class. We ended up with valid solutions for all three problems while experimenting with genotype-to-phenotype mapping and elitism-backed exploration.

In the next chapter, we will look into a family of closely related tasks, namely constraint satisfaction problems, starting with the classic N-Queen problem.

Further reading

For more information, please refer to the following resources:

- Solving the knapsack problem using Dynamic Programming, from the book *Keras Reinforcement Learning Projects* by *Giuseppe Ciaburro*, September 2018
- The Vehicle Routing Problem, from the book *Keras Reinforcement Learning Projects* by *Giuseppe Ciaburro*, September 2018

5
Constraint Satisfaction

In this chapter, you will learn how genetic algorithms can be utilized for solving constraint satisfaction problems. We will start by describing the concept of constraint satisfaction and how it applies to search problems and combinatorial optimization. Then, we will look at several hands-on examples of constraint satisfaction problems and their Python-based solutions using the DEAP framework. The problems we will cover include the well-known N-Queen problem, followed by the nurse scheduling problem, and finally the graph coloring problem. Along the way, we will learn the difference between hard and soft constraints, as well as how they can be incorporated into the solution process.

In this chapter, we will cover the following topics:

- Understanding the nature of constraint satisfaction problems
- Solving the N-Queens problem using a genetic algorithm coded with the DEAP framework
- Solving an example of the nurse scheduling problem using a genetic algorithm coded with the DEAP framework
- Solving the graph coloring problem using a genetic algorithm coded with the DEAP framework
- Understanding the concepts of hard and soft constraints, as well as how to apply them when solving a problem

Technical requirements

In this chapter, we will be using Python 3 with the following supporting libraries:

- `deap`
- `numpy`
- `matplotlib`
- `seaborn`
- `networkx` – introduced in this chapter

The programs that will be used in this chapter can be found in this book's GitHub repository at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter05`.

Check out the following video to see the Code in Action:
`http://bit.ly/39233Qn`

Constraint satisfaction in search problems

In the previous chapter, we looked at solving search problems, which focused on the methodic evaluation of states and transitions between states. Every state transition typically involves a cost or gain, and the objective of the search was to minimize the cost or maximize the gain. Constraint satisfaction problems are a variant of search problems, where the states must satisfy a number of constraints or limitations. If we are able to translate the various violations of constraints into cost and then strive to minimize the cost, solving a constraint satisfaction problem can resemble solving a general search problem.

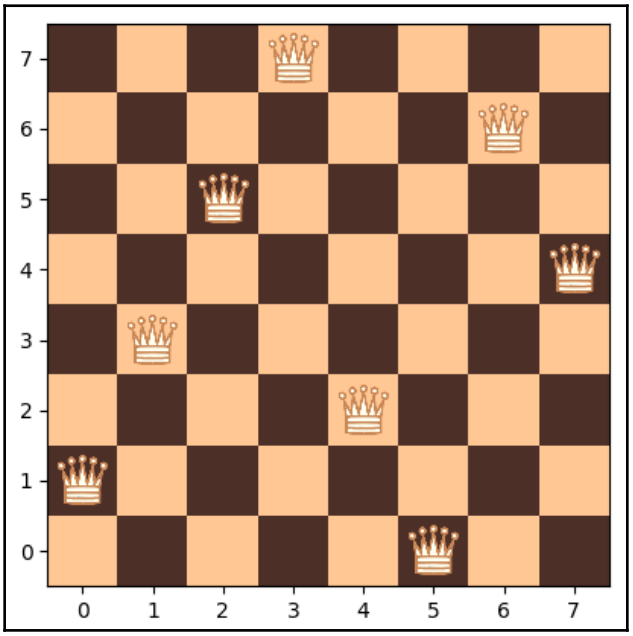
Like combinatorial optimization problems, constraint satisfaction problems have important applications in fields such as artificial intelligence, operations research, and pattern matching. A better understanding of these problems may help in solving numerous types of problems that may seem unrelated at first glance. Constraint satisfaction problems often exhibit high complexity, which makes genetic algorithms a suitable candidate for solving them.

The N-Queens problem, which will be presented in the next section, illustrates the concept of constraint satisfaction problems and demonstrates how they can be solved in a very similar manner to the problems we looked at in the previous chapter.

Solving the N-Queens problem

Originally known as the **eight-queen puzzle**, the classic N-Queens problem originated from the game of chess, and the *8x8* chessboard was its early playground. The task was to place eight chess queens on the board without any two of them threatening each other. In other words, no two queens can share the same row, same column, or same diagonal. The N-Queens problem is similar, using an $N \times N$ chessboard and N chess queens.

The problem is known to have a solution for any natural number, n, except for the cases of $n=2$ and $n=3$. For the original eight-queen case, there are 92 solutions, or 12 unique solutions if we consider symmetrical solutions to be identical. One of the solutions is as follows:



One of the 92 possible solutions for the eight-queen puzzle

By applying combinatorics, the count of all possible ways to place eight pieces on the 8×8 board yields 4,426,165,368 combinations. However, if we can create our candidate solutions in a way that ensures that no two queens will be placed on the same row or the same column, the number of possible combinations is dramatically reduced to 8! (factorial of 8), which amounts to 40,320. We are going to take advantage of this idea in the next subsection when we choose the way our solution to this problem will be represented.

Solution representation

When solving the N-Queens problem, we can take advantage of the knowledge that each row will host exactly one queen, and no two queens will share the same column. This means we can represent any candidate solution as an ordered list of integers – or a list of indices, with each index representing the column that one of the queens occupies for the current row.

For example, in a four-queen problem over a 4×4 chessboard, we have the following list of indices:

```
(3, 2, 0, 1)
```

This translate to the following positions:

- In the first row, the queen is placed in position 3 (fourth column).
- In the second row, the queen is placed in position 2 (third column).
- In the third row, the queen is placed in position 0 (first column).
- In the fourth row, the queen is placed in position 1 (second column).

This is depicted in the following illustration:

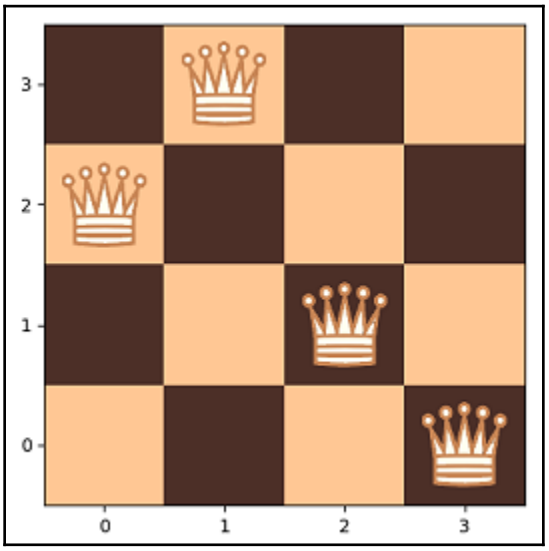


Illustration of the queen arrangement represented by the list (3, 2, 0, 1)

Similarly, another arrangement of the indices may look as follows:

```
(1, 3, 0, 2)
```

This arrangement represents the candidate solution shown in the following illustration:

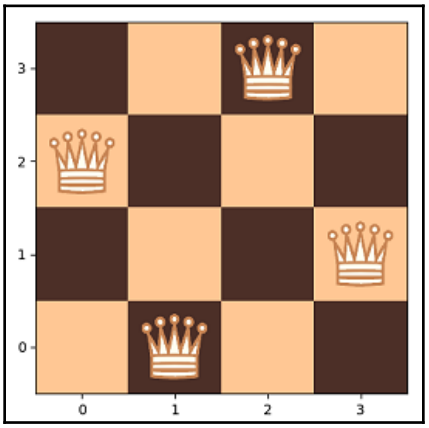


Illustration of the queen arrangement represented by the list (1, 3, 0, 2)

The only constraint violations that are possible in candidate solutions represented this way are shared diagonals between pairs of queens.

For example, the first candidate solution we discussed contains two violations, as shown here:

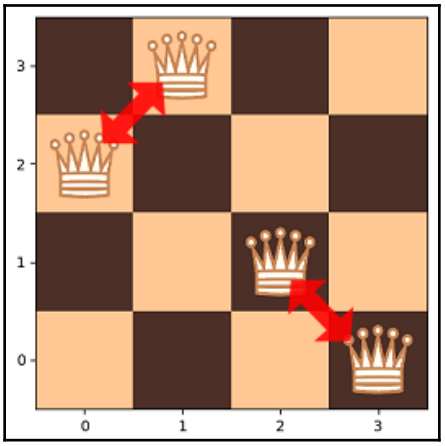


Illustration of the queen arrangement represented by the list (3, 2, 0, 1), with constraint violations indicated

However, the preceding one exhibited no violations.

This means that, when evaluating the solutions that are represented in this way, we only need to find and count the shared **diagonals** between the positions they stand for.

The solution representation we just discussed is a central part of the Python class that we will describe in the next subsection.

Python problem representation

To encapsulate the N-Queens problem, we've created a Python class called `NQueensProblem`. This class can be found in the `queens.py` file of this book's GitHub repository, at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter05/queens.py`.

The class is initialized with the desired size of the problem and provides the following public methods:

- **getViolationsCount(positions)**: Calculates the number of violations in the given solution, which is represented by a list of indices, as discussed in the previous subsection
- **plotBoard(positions)**: Plots the positions of the queens on the board according to the given solution

The main method of the class exercises the class' methods by creating an eight-queen problem and testing the following candidate solution for it:

```
(1, 2, 7, 5, 0, 3, 4, 6)
```

This is followed by plotting the candidate solution and calculating the number of constraint violations.

The resulting output is as follows:

```
Number of violations =  3
```

The plot for this is as follows – can you spot all three violations?

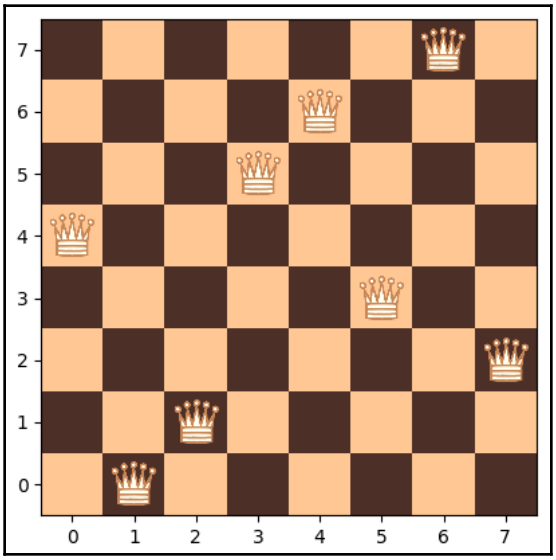


Illustration of the eight-queen arrangement represented by the list (1, 2, 7, 5, 0, 3, 4, 6)

In the next subsection, we will apply the genetic algorithm approach for solving the N-Queens problem.

Genetic algorithms solution

To solve the N-Queens problem using a genetic algorithm, we created the Python program `01-solve-n-queens.py`, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter05/01-solve-n-queens.py`.

Since the solution representation we chose for this problem is a list (or an array) of indices, which is similar to the representation we used for the **traveling salesman problem** (**TSP**) and the **vehicle routing problem** (**VRP**) in `Chapter 4`, *Combinatorial Optimization*, we were able to utilize a similar genetic approach, just like we did there. In addition, we take advantage of elitism once more by reusing the elitist version that we created for DEAP's simple genetic flow.

The following steps describe the main parts of our solution:

1. Our program starts by creating an instance of the `NQueensProblem` class using the size of the problem we would like to solve:

```
nQueens = queens.NQueensProblem(NUM_OF_QUEENS)
```

2. Since our goal is to minimize the count of violations (hopefully to a value of 0), we define a single objective, minimizing fitness strategy:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

3. Since the solution is represented by an ordered list of integers, each representing a column location of a queen, we use the following toolbox definitions to create the initial population:

```
# create an operator that generates randomly shuffled indices:
toolbox.register("randomOrder", random.sample,
range(len(nQueens)), len(nQueens))

toolbox.register("individualCreator", tools.initIterate,
creator.Individual, toolbox.randomOrder)
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

4. The actual fitness function is set to count the number of violations caused by the placement of the queens on the chessboard, as represented by each individual solution:

```
def getViolationsCount(individual):
    return nQueens.getViolationsCount(individual),

toolbox.register("evaluate", getViolationsCount)
```

5. As for the genetic operators, we use tournament selection with a tournament size of 2, as well as the crossover and mutation operators, which are specialized for ordered lists:

```
# Genetic operators:
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxUniformPartialyMatched,
indpb=2.0/len(vrp))
toolbox.register("mutate", tools.mutShuffleIndexes,
indpb=1.0/len(vrp))
```

6. In addition, we continue to use the elitist approach, where the **hall-of-fame** (**HOF**) members – the current best individuals – are always passed untouched to the next generation. As we found out in the previous chapter, this approach works well with a tournament selection of size 2:

```
population, logbook = elitism.eaSimpleWithElitism(population,
                                                   toolbox,
                                                   cxpb=P_CROSSOVER,
                                                   mutpb=P_MUTATION,
                                                   ngen=MAX_GENERATIONS,
                                                   stats=stats,
                                                   halloffame=hof,
                                                   verbose=True)
```

7. Since each N-Queens problem can have multiple possible solutions, we print out all hall-of-fame members, instead of just the top one, so that we can see how many valid solutions we found:

```
print("- Best solutions are:")
for i in range(HALL_OF_FAME_SIZE):
  print(i, ": ", hof.items[i].fitness.values[0], " -> ",
hof.items[i])
```

As we saw earlier, our solution representation reduces the eight-queen case to only about 40,000 possible combinations, which makes it a rather small problem. To make things more interesting, let's increase the size to 16 queens, where the number of possible candidate solutions will be *16!* This calculates to the colossal value of 20,922,789,888,000. The number of valid solutions to this problem is quite large too, at just under 15 million. But comparing this to the number of possible combinations, searching for a valid solution is still like trying to find a needle in a haystack.

Before we run the program, let's set the algorithm constants, as follows:

```
NUM_OF_QUEENS = 16
POPULATION_SIZE = 300
MAX_GENERATIONS = 100
HALL_OF_FAME_SIZE = 30
P_CROSSOVER = 0.9
P_MUTATION = 0.1
```

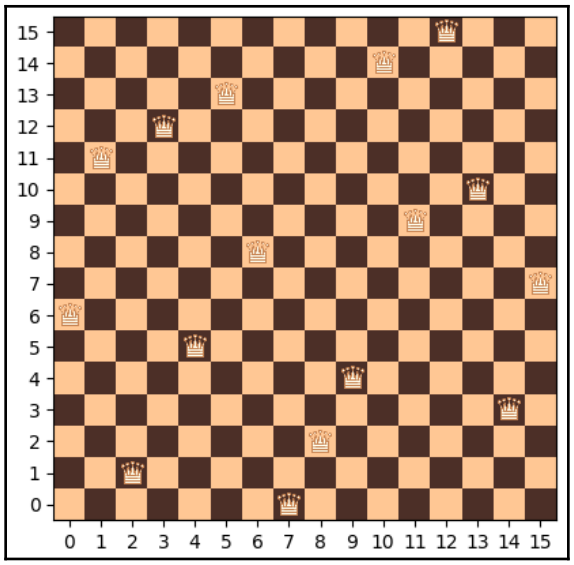
Running the program with these settings yields the following output:

```
gen nevals min avg
0 300 3 10.4533
1 246 3 8.85333
..
23 250 1 4.38
```

```
24 227 0 4.32
..
- Best solutions are:
0 : 0.0 -> Individual('i', [7, 2, 8, 14, 9, 4, 0, 15, 6, 11, 13, 1, 3, 5,
10, 12])
1 : 0.0 -> Individual('i', [7, 2, 6, 14, 9, 4, 0, 15, 8, 11, 13, 1, 3, 5,
12, 10])
..
7 : 0.0 -> Individual('i', [14, 2, 6, 12, 7, 4, 0, 15, 8, 11, 3, 1, 9, 5,
10, 13])
8 : 1.0 -> Individual('i', [2, 13, 6, 12, 7, 4, 0, 15, 8, 14, 3, 1, 9, 5,
10, 11])
..
```

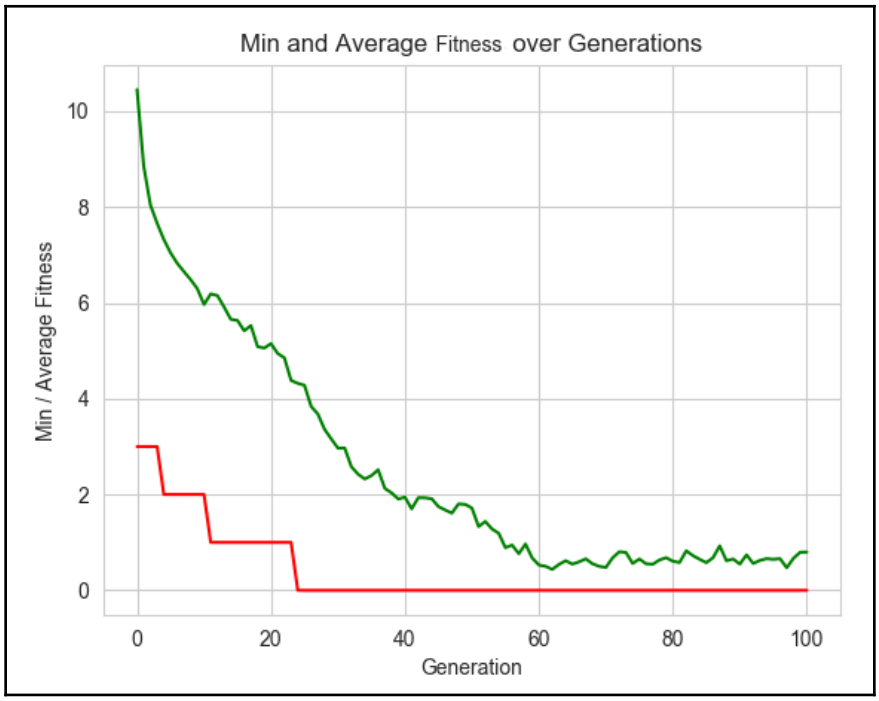
From the printouts, we can learn that a solution was first found in generation 24, where the fitness value shows as 0, which means no violations. In addition, the printout of the best solutions indicates that eight different solutions were found during the run. These solutions are the entries of 0-7 in the HOF, which have a fitness of 0. The next entry already has a fitness value of 1, denoting a violation.

The first plot that's produced by the program depicts the placement of the 16 queens on the *16x16* chessboard, as defined by the first valid solution that was found – (7, 2, 8, 14, 9, 4, 0, 15, 6, 11, 13, 1, 3, 5, 10, 12):



A plot of a valid 16-queen arrangement found by the program

The second plot contains a graph of the max and average fitness values over the generations. From this graph, we can see that even though the best fitness value of zero was found early on, around generation 24, the average fitness value kept decreasing as more solutions were found:



Stats of the program solving the 16-queen problem

Increasing the value of `MAX_GENERATIONS` to 400 without making any other changes will result in finding 38 valid solutions. If we increase `MAX_GENERATIONS` to 500, all 50 members of the hall-of-fame will contain valid solutions. You are encouraged to try out various combinations of the genetic algorithm's settings, as well as solving other sizes of the N-Queen problem.

In the next section, we will be transitioning from arranging game pieces on a board to placing workers on a work schedule.

Solving the nurse scheduling problem

Imagine you are responsible for scheduling the shifts for the nurses in your hospital department for this week. There are three shifts in a day – morning, afternoon, and night – and for each shift, you need to assign one or more of the eight nurses that work in your department. If this sounds like a simple task, take a look at the list of relevant hospital rules:

- A nurse is not allowed to work two consecutive shifts.
- A nurse is not allowed to work more than five shifts per week.
- The number of nurses per shift in your department should fall within the following limits:
    - Morning shift: 2–3 nurses
    - Afternoon shift: 2–4 nurses
    - Night shift: 1–2 nurses

In addition, each nurse can have shift preferences. For example, one nurse prefers to only work morning shifts, another nurse prefers to not work afternoon shifts, and so on.

This task is an example of the **nurse scheduling problem** (**NSP**), which can have many variants. Possible variations may include different specialties for different nurses, the ability to work on cover shifts (overtime), or even different types of shifts – such as 8-hour shifts and 12-hour shifts.

By now, it probably looks like a good idea to write a program that will do the scheduling for you. Why not apply our knowledge of genetic algorithms to implement such a program? As usual, we will start by representing the solution to the problem.

Solution representation

For solving the nurse scheduling problem, we decided to use a binary list (or array) to represent the schedule as it will be intuitive for us to interpret, and we've seen that genetic algorithms can naturally handle this representation.

For each nurse, we can have a binary string representing the 21 shifts of the week. A value of 1 represents a shift that the nurse is scheduled to work on. For example, take a look at the following binary list:

```
(0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0)
```

This can be broken into the following groups of three values, representing the shifts this nurse will be working each day of the week:

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
(0, 1, 0)	(1, 0, 1)	(0, 1, 1)	(0, 0, 0)	(0, 0, 1)	(1, 0, 0)	(0, 1, 0)
afternoon	morning and night	afternoon and night	(none)	night	morning	afternoon

The schedules of all nurses can be then concatenated together to create one long binary list representing the entire solution.

When evaluating a solution, this long list can be broken down into the schedules of the individual nurses, and violations of the constraints can be checked for. The preceding sample nurse schedule, for instance, contains two occurrences of consecutive 1 values that represent consecutive shifts being worked (afternoon followed by night, and night followed by morning). The number of weekly shifts for that same nurse can be calculated by totaling the binary values of the list, which results in 8 shifts. We can also easily check for adherence to the shift preferences by checking each day's shifts against the given preferred shifts of that nurse.

Finally, to check for the constraints of the number of nurses per shift, we can sum the weekly schedules of all nurses and look for entries that are larger than the maximum allowed or smaller than the minimum allowed.

But before we continue with our implementation, we need to discuss the difference between hard constraints and soft constraints.

Hard constraints versus soft constraints

When solving the nurse scheduling problem, we should bear in mind that some of the constraints represent hospital rules that cannot be broken. A schedule that contains one or more violations of these rules will be considered invalid. More generally, these are known as **hard constraints**.

The nurses' preferences, on the other hand, can be considered **soft constraints**. We would like to adhere to them as much as possible, and a solution that contains no violation or fewer violations of these constraints is considered better than one that contains more violations. But a violation of these constraints does not invalidate the solution.

In the case of the N-Queens problem, all the constraints – row, column, and diagonal – were hard constraints. Had we not found a solution where the number of violations was zero, we would not have a valid solution for the problem. Here, on the other hand, while the hospital rules are considered hard constraints, the nurses' preferences are soft constraints. So, we are actually looking for a solution that will not violate any of the hospital rules while minimizing the number of breaches to the nurses' preferences.

While dealing with soft constraints is similar to what we do in any optimization problem, that is, we strive to minimize them, how do we deal with the hard constraints that accompany them? There are several possible strategies:

- Find a particular representation (coding) of the solution that **eliminates the possibility** of a hard constraint violation. When solving the N-Queens problem, we were able to represent a solution in a way that eliminated the possibility for two of the three constraints – row and column, which considerably simplified our solution. But generally, such coding may be difficult to find.
- When evaluating the solutions, **discard** candidate solutions that violate any hard constraint. The disadvantage of this approach is the loss of information contained in these solutions, which may be valuable for the problem. This could considerably slow down the optimization process.
- When evaluating the solutions, **repair** candidate solutions that violate any hard constraint. In other words, find a way to manipulate the solution and modify it so it will no longer violate the constraint(s). Creating such a repair procedure can prove difficult or impossible for most problems, and at the same time, the repair process may result in significant loss of information.
- When evaluating the solutions, **penalize** candidate solutions that violate any hard constraint. This will degrade the solution's score and make it less desirable, but will not eliminate it completely, so the information contained in it is not lost. Effectively, this leads to a hard constraint to be considered similar to a soft constraint, but with a heavier penalty. When using this method, the challenge may be to find the appropriate extent of the penalty. Too harsh a penalty may lead to a de facto elimination of such solutions, while a penalty too small may lead to these solutions appearing as optimal.

In our case, we chose to apply the fourth approach and penalize the violations of the hard constraints to a larger degree than those of the soft constraints. This was done by creating a cost function, where the cost of a hard constraint violation is greater than that of a soft constraint violation. The total cost is then used as the fitness function to be minimized. This is implemented within the problem representation that will be discussed in the next subsection.

Python problem representation

To encapsulate the nurse scheduling problem we described at the beginning of this section, we created a Python class called `NurseSchedulingProblem`. This class is contained in the `nurses.py` file, which can be found at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter05/nurses.py`.

The class' constructor accepts the `hardConstraintPenalty` parameter, which represents the penalty factor for a hard constraint violation. Then, it continues to initialize the various parameters, describing the scheduling problem:

```
# list of nurses:
self.nurses = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']

# nurses' respective shift preferences - morning, evening, night:
self.shiftPreference = [[1, 0, 0], [1, 1, 0], [0, 1, 1], [0, 1, 0], [0, 0,
1], [1, 1, 1], [0, 1, 1], [1, 1, 1]]

# min and max number of nurses allowed for each shift - morning, evening,
night:
self.shiftMin = [2, 2, 1]
self.shiftMax = [3, 4, 2]

# max shifts per week allowed for each nurse
self.maxShiftsPerWeek = 5
```

The class uses the following method to convert the given schedule into a dictionary with a separate schedule for each nurse:

- `getNurseShifts(schedule)`

The following methods are used to count the various types of violations:

- `countConsecutiveShiftViolations(nurseShiftsDict)`
- `countShiftsPerWeekViolations(nurseShiftsDict)`
- `countNursesPerShiftViolations(nurseShiftsDict)`
- `countShiftPreferenceViolations(nurseShiftsDict)`

In addition, the class provides the following public methods:

- `getCost(schedule)`: Calculates the total cost of the various violations in the given schedule. This method uses the value of the `hardConstraintPenalty` variable.
- `printScheduleInfo(schedule)`: Prints the schedule and violations details.

The main method of the class exercises the class' methods by creating an instance of the nurse scheduling problem and testing a randomly generated solution for it. The resulting output may look as follows, with the value of `hardConstraintPenalty` set to 10:

```
Random Solution =
[1 0 0 0 0 1 1 1 0 0 0 1 1 0 0 0 1 0 1 0 1 1 0 1 0 1 1 1 0 1 0 1 0 1 1 1 1
 0 0 1 0 1 0 0 1 0 1 1 0 1 1 0 1 1 0 1 1 1 1 0 1 0 1 0 1 0 1 1 0 1 0 1 1 1
 1 0 1 0 0 0 1 1 0 1 1 1 1 0 1 1 0 1 1 1 1 1 0 1 0 0 1 1 0 1 1 1 0 0 0 0 0
 0 1 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 0 0 1 1 1 1 0 0 0 0 1 1 0 1 1 0
 0 1 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 1 1]

Schedule for each nurse:
A : [1 0 0 0 0 1 1 1 0 0 0 1 1 0 0 0 1 0 1 0 1]
B : [1 0 1 0 1 1 1 0 1 0 1 0 1 1 1 1 0 0 1 0 1]
C : [0 0 1 0 1 1 0 1 1 0 1 1 0 1 1 1 1 0 1 0 1]
D : [0 1 0 1 1 0 1 0 1 1 1 1 0 1 0 0 0 1 1 0 1]
E : [1 1 1 0 1 1 0 1 1 1 1 1 0 1 0 0 1 1 0 1 1]
F : [1 0 0 0 0 0 0 1 0 1 0 0 0 0 1 1 0 0 0 0 0]
G : [0 0 0 0 1 1 0 0 1 1 1 1 0 0 0 0 1 1 0 1 1]
H : [0 0 1 0 1 1 1 0 0 0 0 0 0 0 1 1 1 0 0 1 1]
consecutive shift violations = 40

weekly Shifts = [9, 13, 13, 12, 15, 5, 10, 9]
Shifts Per Week Violations = 46

Nurses Per Shift = [4, 2, 4, 1, 6, 6, 4, 4, 5, 4, 5, 5, 2, 4, 4, 4, 5, 3,
4, 3, 7]
Nurses Per Shift Violations = 30

Shift Preference Violations = 30

Total Cost = 1190
```

As is evident from these results, a randomly generated solution is likely to yield a large number of violations, and consequently a large cost value. In the next subsection, we attempt to minimize the cost and eliminate all hard constraint violations using a genetic algorithm-based solution.

Genetic algorithms solution

To solve the nurse scheduling problem using a genetic algorithm, we created the Python program called `02-solve-nurses.py`, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter05/02-solve-nurses.py`.

Since the solution representation we chose for this problem is a list (or an array) of binary values, we were able to use the same genetic approach we used for several problems we have solved already, such as the 0-1 knapsack problem we described in `Chapter 4`, *Combinatorial Optimization*.

The main parts of our solution are described in the following steps:

1. Our program starts by creating an instance of the `NurseSchedulingProblem` class with the desired value for `hardConstraintPenalty`, which is set by the `HARD_CONSTRAINT_PENALTY` constant:

```
nsp = nurses.NurseSchedulingProblem(HARD_CONSTRAINT_PENALTY)
```

2. Since our goal is to minimize the cost, we define a single objective, minimizing fitness strategy:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

3. Since the solution is represented by a list of `0` or `1` values, we use the following toolbox definitions to create the initial population:

```
toolbox.register("zeroOrOne", random.randint, 0, 1)
toolbox.register("individualCreator", tools.initRepeat,
creator.Individual, toolbox.zeroOrOne, len(nsp))
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

4. The actual fitness function is set to calculate the cost of the various violations in the schedule, represented by each individual solution:

```
def getCost(individual):
    return nsp.getCost(individual),

toolbox.register("evaluate", getCost)
```

5. As for the genetic operators, we use tournament selection with a tournament size of 2, along with two-point crossover and flip-bit mutation, since this is suitable for binary lists:

```
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit,
indpb=1.0/len(nsp))
```

6. We keep using the elitist approach, where HOF members – the current best individuals – are always passed untouched to the next generation:

```
population, logbook = elitism.eaSimpleWithElitism(population,
toolbox, cxpb=P_CROSSOVER, mutpb=P_MUTATION,
ngen=MAX_GENERATIONS, stats=stats, halloffame=hof,
verbose=True)
```

7. When the algorithm concludes, we print the details of the best solution that was found:

```
nsp.printScheduleInfo(best)
```

Before we run the program, let's set the algorithm constants, as follows:

```
POPULATION_SIZE = 300
P_CROSSOVER = 0.9
P_MUTATION = 0.1
MAX_GENERATIONS = 200
HALL_OF_FAME_SIZE = 30
```

In addition, let's start by setting the penalty for violating hard constraints to a value of `1`, which makes the cost of violating a hard constraint similar to that of violating a soft constraint:

```
HARD_CONSTRAINT_PENALTY = 1
```

Running the program with these settings yields the following output:

```
-- Best Fitness = 3.0

-- Schedule =
Schedule for each nurse:
A : [0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0]
B : [1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0]
C : [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
D : [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0]
E : [0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0]
F : [0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0]
G : [0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1]
H : [1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0]
consecutive shift violations = 0

weekly Shifts = [5, 6, 2, 5, 4, 5, 5, 5]
Shifts Per Week Violations = 1

Nurses Per Shift = [2, 2, 1, 2, 2, 1, 2, 2, 1, 2, 2, 2, 2, 2, 1, 2, 2, 2,
2, 2, 1]
Nurses Per Shift Violations = 0

Shift Preference Violations = 2
```

This may seem like a good result since we ended up with only three constraint violations. However, one of them is a shift-per-week violation – nurse B was scheduled with six shifts for the week – exceeding the maximum allowed of five. This is enough to make the entire solution unacceptable.

Attempting to eliminate this kind of violation, we proceed to increase the hard constraint penalty value to `10`:

```
HARD_CONSTRAINT_PENALTY = 10
```

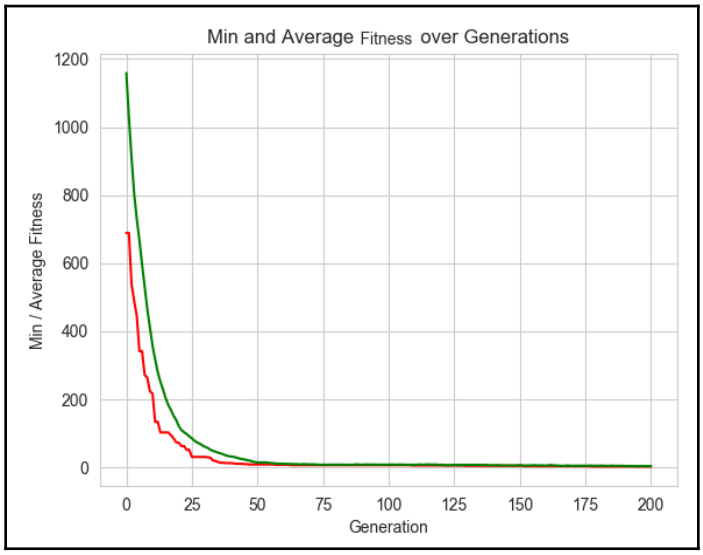
Now, the result is as follows:

```
-- Best Fitness = 3.0

-- Schedule =
Schedule for each nurse:
A : [0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
B : [1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0]
C : [0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1]
D : [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0]
E : [0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
F : [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0]
```

```
G : [0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0]
H : [1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0]
consecutive shift violations = 0

weekly Shifts = [4, 5, 5, 5, 3, 5, 5, 5]
Shifts Per Week Violations = 0

Nurses Per Shift = [2, 2, 1, 2, 2, 1, 2, 2, 2, 2, 2, 1, 2, 2, 1, 2, 2, 2,
2, 2, 1]
Nurses Per Shift Violations = 0

Shift Preference Violations = 3
```

Again, we got three violations, but this time, they were all soft constraint violations, which makes this solution valid.

The following graph, which depicts the minimum and average fitness over the generations, indicates that over the first 40-50 generations, the algorithm was able to eliminate all hard constraint violations, and from there on there were only small incremental improvements, which occurred whenever another soft constraint was eliminated:



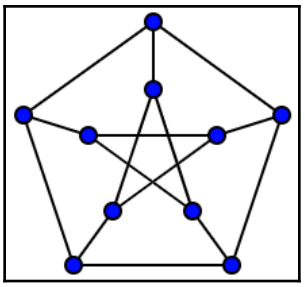
Stats of the program solving the nurse scheduling problem

It seems that, in our case, it was enough to set a ten-fold penalty on hard constraint violations. In other problems, higher values may be required. You are encouraged to experiment by altering the problem's definitions, as well as the genetic algorithm's settings.

The same trade-off we have just seen between soft and hard constraints is going to play a part in the next task we take on – the graph coloring problem.

Solving the graph coloring problem

In the mathematical branch of graph theory, a graph is a structured collection of objects that represents the relationships between pairs of these objects. The objects appear as vertices (or nodes) in the graph, while the relation between a pair of objects is represented using an edge. A common way of illustrating a graph is by drawing the vertices as circles and the edges as connecting lines, as depicted in the following diagram of the Petersen graph, named after the Danish mathematician Julius Petersen:

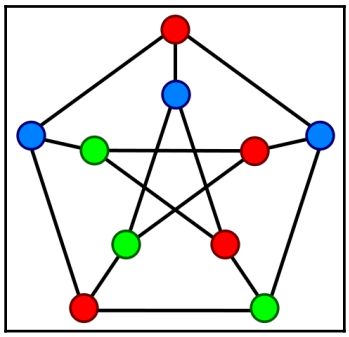


Petersen graph
Source: https://commons.wikimedia.org/wiki/File:Petersen1_tiny.svg

Graphs are remarkably useful objects as they can represent and help us research an overwhelming variety of real-life structures, patterns, and relationships, such as social networks, power grid layouts, website structures, linguistic compositions, computer networks, atomic structures, migration patterns, and more.

The graph coloring task is used to assign a color for every node in the graph in such a way that no pair of connected (adjacent) nodes will share the same color. This is also known as the **proper coloring** of the graph.

The following diagram shows the same Petersen graph, but this time colored properly:



Proper coloring of Petersen graph. Source: https://en.wikipedia.org/wiki/File:Petersen_graph_3-coloring.svg
Released to the public domain

The color assignment is often accompanied by an optimization requirement – use the **minimum possible** number of colors. For example, the Peterson graph can be properly colored using three colors, as demonstrated in the preceding diagram. But it would be impossible to properly color it using only two colors. In graph theory terms, this means that the *chromatic number* of this graph is three.

Why would we care about coloring the nodes of the graph? Apparently, many real-life problems can be translated into a graph representation in such a way that the graph coloring will stand for a solution; for example, scheduling classes for a student, or shifts for an employee can be translated into a graph, where adjacent nodes represent classes or shifts that cause a conflict. Such a conflict can be classes that fall at the same time or shifts that are consecutive (sound familiar?). Due to this conflict, assigning the same person to both classes (or both shifts) will invalidate the schedule. If each color represents a different person, assigning different colors to adjacent nodes will solve the conflicts. The N-Queen problem we encountered at the beginning of this chapter can be represented as a graph coloring problem, where every node in the graph represents a square on the chessboard, and every pair of nodes that share a row, a column, or a diagonal is connected by an edge. Other relevant examples include frequency assignments to radio stations, power grid redundancy planning, traffic light timing, and even Sudoku puzzle-solving.

Hopefully, this has convinced you that graph coloring is a problem worth solving. As usual, we will start by formulating an appropriate representation of a possible solution for this problem.

Solution representation

Expanding on the commonly used binary list (or array) representation, we can employ a list of integers, where each integer represents a unique color, while each element of the list matches one of the graph's nodes.

For example, since the Petersen graph has 10 nodes, we can assign each node an index between 0 and 9. Then, we can represent the node coloring for that graph using a list of 10 elements.

For example, let's have a look at what we have in this particular representation:

```
(0, 2, 1, 3, 1, 2, 0, 3, 3, 0)
```

Let's talk about what we have here in detail:

- Four colors are used, represented by the integers 0, 1, 2, 3.
- The first, seventh, and tenth nodes of the graph are colored with the first color (0).
- The third and fifth nodes are colored with the second color (1).
- The second and sixth nodes are colored with the third color (2).
- The fourth, eighth, and ninth nodes are colored with the fourth color (3).

To evaluate the solution, we need to iterate over each pair of adjacent nodes and check if they share the same color. If they do, this is a coloring violation, and we seek to minimize the number of violations to zero to achieve the proper coloring of the graph.

However, you may recall that we also seek to minimize the number of colors that are used. If we happen to already know this number, we can just use as many integer values as the known number of colors. But what if we don't? One way to go about this is to start with an estimate (or just a guess) for the number of colors used. If we find a proper solution using this number, we can reduce the number and try again. If no solution was found, we can increase the number and try again until we have the smallest number we could find a solution with. However, we may be able to get to this number faster by using soft and hard constraints, as described in the next subsection.

Using hard and soft constraints for the graph coloring problem

When solving the nurse scheduling problem earlier in this chapter, we noted the difference between hard constraints – those we have to adhere to for the solution to be considered valid – and soft constraints – those we strive to minimize to get the best solution. In the graph coloring problem, the color assignment requirement – where no two adjacent nodes can have the same color – is a hard constraint. We have to minimize the number of violations of this constraint to zero to achieve a valid solution.

Minimizing the number of colors used, however, can be introduced as a soft constraint. We would like to minimize this number, but not at the expense of violating the hard constraint.

This will allow us to launch the algorithm with a number of colors higher than our estimate and let the algorithm minimize it until – ideally – it reaches the actual minimal color count.

As we did in the nurse scheduling problem, we will implement this approach by creating a cost function, where the cost of a hard constraint violation is greater than the cost caused by using more colors. The total cost is then used as the fitness function to be minimized. This functionality can be incorporated into the Python class, which will be described in the next subsection.

Python problem representation

To encapsulate the graph coloring problem, we've created a Python class called `GraphColoringProblem`. This class can be found in the `graphs.py` file, which can be found at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter05/graphs.py`.

To implement this class, we utilized the open source Python package ***NetworkX*** (`https://networkx.github.io`), which enables, among the rest, the creation, manipulation, and drawing of graphs. The graph we use as the subject of the coloring problem is an instance of the NetworkX graph class. Instead of creating this graph from scratch, we can take advantage of the numerous preexisting graphs contained in this library, such as the Petersen graph we saw earlier.

The constructor of the `GraphColoringProblem` class accepts the graph to be colored as a parameter. In addition, it accepts the `hardConstraintPenalty` parameter, which represents the penalty factor for a hard constraint violation.

The constructor then creates a list of the graph's nodes, as well as an adjacency matrix, that allows us to quickly find out if any two nodes in the graph are adjacent:

```
self.nodeList = list(self.graph.nodes)
self.adjMatrix = nx.adjacency_matrix(graph).todense()
```

The class uses the following method to calculate the number of coloring violations in the given color arrangement:

- **getViolationsCount**(colorArrangement)

The following method is used to calculate the number of colors used by the given color arrangement:

- **getNumberOfColors**(colorArrangement)

In addition, the class provides the following public methods:

- **getCost(colorArrangement)**: Calculates the total cost of the given color arrangement
- **plotGraph(colorArrangement)**: Plots the graph with the nodes colored according to the given color arrangement

The main method of the class exercises the class' methods by creating a Petersen graph instance and testing a randomly generated color arrangement for it, containing up to five colors. In addition, it sets the value of `hardConstraintPenalty` to 10:

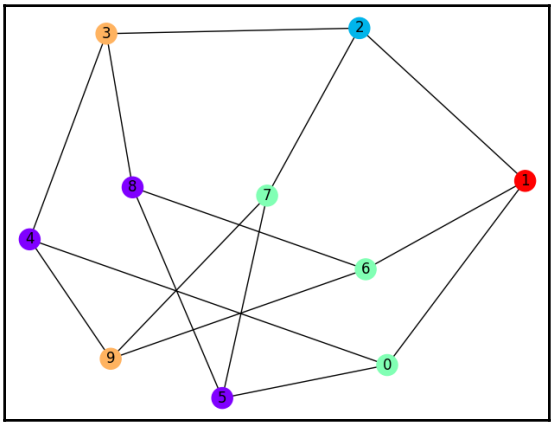
```
gcp = GraphColoringProblem(nx.petersen_graph(), 10)
solution = np.random.randint(5, size=len(gcp))
```

The resulting output may look as follows:

```
solution = [2 4 1 3 0 0 2 2 0 3]
number of colors = 5
Number of violations = 1
Cost = 15
```

Since this particular random solution uses five colors and causes one coloring violation, the calculated cost is 15.

The plot for this solution is as follows – can you spot the single coloring violation?



Petersen graph improperly colored with five colors

In the next subsection, we apply a genetic algorithm-based solution in an attempt to eliminate any coloring violations while also minimizing the number of colors that are used.

Genetic algorithms solution

To solve the graph coloring problem using a genetic algorithm, we've created the Python program `03-solve-graphs.py`, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter05/03-solve-graphs.py`.

Since the solution representation we chose for this problem is a list of integers, we need to expand the genetic approach of using a binary list a little bit.

The following steps describe the main points of our solution:

1. The program starts by creating an instance of the `GraphColoringProblem` class with the desired NetworkX graph to be solved – the familiar Petersen graph in this case, and the desired value for `hardConstraintPenalty`, which is set by the `HARD_CONSTRAINT_PENALTY` constant:

```
gcp = graphs.GraphColoringProblem(nx.petersen_graph(),
HARD_CONSTRAINT_PENALTY)
```

2. Since our goal is to minimize the cost, we define a single objective, minimizing fitness strategy:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

3. Since the solution is represented by a list of integer values representing the participating colors, we need to define a random generator that creates an integer between 0 and the number of colors minus 1. This random integer represents one of the participating colors. Then, we define a solution (individual) creator that generates a list of these random integers that matches the given graph in length – this is how we randomly assign a color for each node in the graph. Finally, we define the operator that creates an entire population of individuals:

```
toolbox.register("Integers", random.randint, 0, MAX_COLORS - 1)
toolbox.register("individualCreator", tools.initRepeat,
creator.Individual, toolbox.Integers, len(gcp))
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

4. The fitness evaluation function is set to calculate the combined cost of the coloring violations and the number of colors used, which is associated with each individual solution, by calling the `getCost()` method of the `GraphColoringProblem` class:

```
def getCost(individual):
    return gcp.getCost(individual),

toolbox.register("evaluate", getCost)
```

5. As for the genetic operators, we can still use the same selection and crossover operations we used for binary lists; however, the mutation operation needs to change. The *flip bit* mutation that's used for binary lists flips between values of `0` and `1`, while here, we need to change a given integer to another, randomly generated, integer in the allowed range. The `mutUniformInt` operator does just that – we just need to set the range similar to what we did with the preceding integers operator:

```
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutUniformInt, low=0,
up=MAX_COLORS - 1, indpb=1.0/len(gcp))
```

6. We keep using the elitist approach, where the HOF members – the current best individuals – are always passed untouched to the next generation:

```
population, logbook = elitism.eaSimpleWithElitism(population,
toolbox, cxpb=P_CROSSOVER,                mutpb=P_MUTATION,
ngen=MAX_GENERATIONS, stats=stats, halloffame=hof,
verbose=True)
```

7. When the algorithm concludes, we print the details of the best solution that was found before plotting the graphs.

Before we run the program, let's set the algorithm constants, as follows:

```
POPULATION_SIZE = 100
P_CROSSOVER = 0.9
P_MUTATION = 0.1
MAX_GENERATIONS = 100
HALL_OF_FAME_SIZE = 5
```

In addition, we need to set the penalty for violating hard constraints to a value of `10` and the number of colors to `10`:

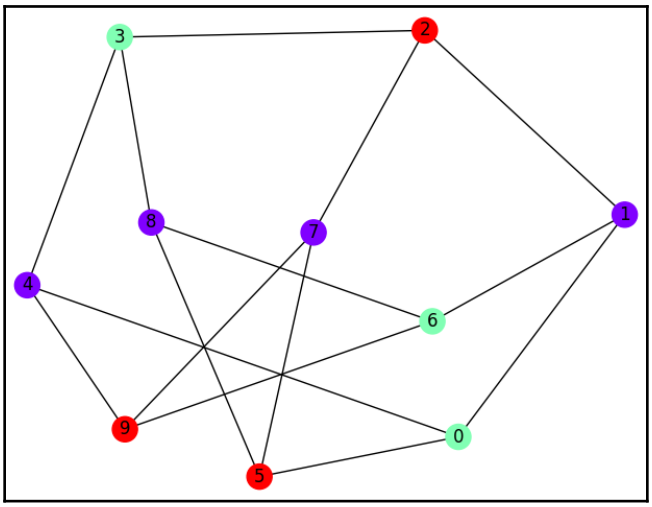
```
HARD_CONSTRAINT_PENALTY = 10
MAX_COLORS = 10
```

Running the program with these settings yields the following output:

```
-- Best Individual = [5, 0, 6, 5, 0, 6, 5, 0, 0, 6]
-- Best Fitness = 3.0

number of colors = 3
Number of violations = 0
Cost = 3
```

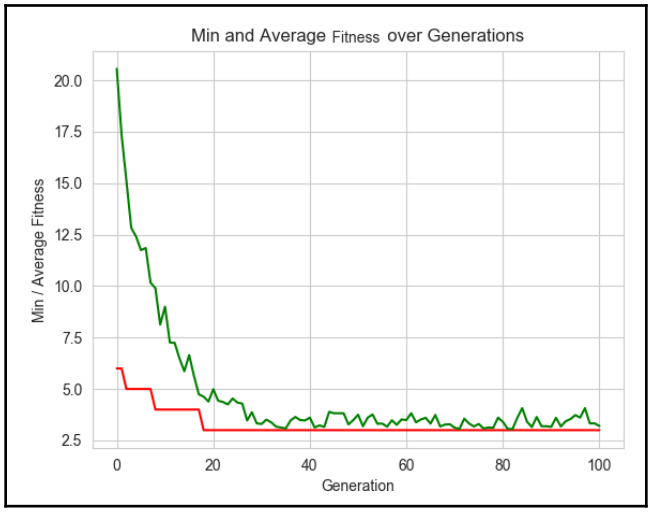
This means that the algorithm was able to find a proper coloring for the graph using three colors, denoted by the integers `0`, `5`, and `6`. As we mentioned previously, the actual integer values don't matter – it's the distinction between them that does. Three is indeed the known chromatic number of the Petersen graph.

The preceding code creates the following plot, which illustrates the solution's validity:



A plot of the Petersen graph properly colored by the program using three colors

The following graph, which depicts the minimum and average fitness over the generations, indicates that the algorithm reached the solution rather quickly since the Petersen graph is relatively small:



Stats of the program solving the graph coloring problem for the Petersen graph

To try a larger graph, let's replace the Petersen graph with a Mycielski graph of order `5`. This graph contains 23 nodes and 71 edges, and is known to have a chromatic number of `5`:

```
gcp = graphs.GraphColoringProblem(nx.mycielski_graph(5),
HARD_CONSTRAINT_PENALTY)
```

Using the same parameters as before, including the setting of 10 colors, we get the following results:

```
-- Best Individual = [9, 6, 9, 4, 0, 0, 6, 5, 4, 5, 1, 5, 1, 1, 6, 6, 9, 5,
9, 6, 5, 1, 4]
-- Best Fitness = 6.0

number of colors = 6
Number of violations = 0
Cost = 6
```

Since we happen to know that the chromatic number for this graph is `5`, this is not the optimal solution, although close. How can we get there? And what if we didn't know the chromatic number beforehand? One way to go about this is to change the parameters of the genetic algorithm; for example, increase the population size (and possibly HOF size) and/or increase the number of generations. Another approach would be to start the same search again, but with a reduced number of colors. Since the algorithm found a solution with six colors, let's reduce the maximum number of colors to five and see if the algorithm can still find a valid solution:

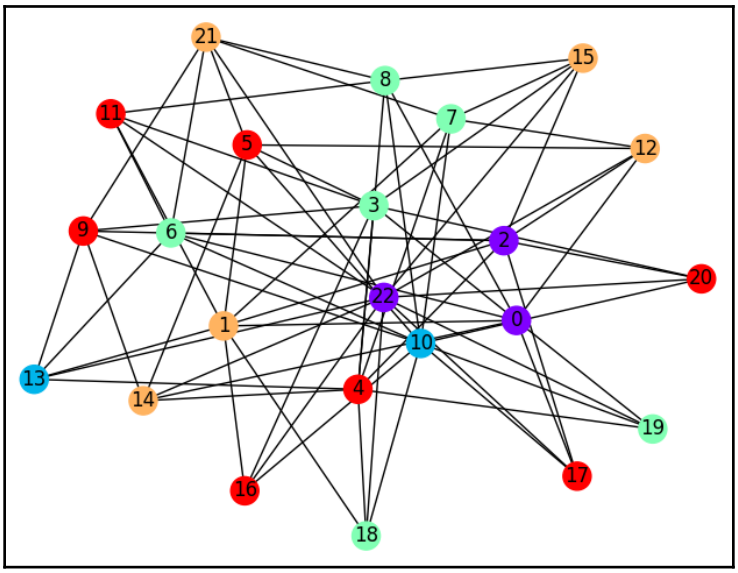
```
MAX_COLORS = 5
```

Why would the algorithm find a five-color solution now if it didn't find one in the first place? As we lower the number of colors from 10 to 5, the search space is considerably reduced – in this case, from 10^{23} to 5^{23} (since we have 23 nodes in the graph) – and the algorithm has a better chance of finding the optimal solution(s), even with a short run and a limited population size. So, while the first run of the algorithm may get us close to the solution, it could be good practice to keep decrementing the number of colors until the algorithm can't find a better solution.

In our case, when started with five colors, the algorithm was able to find a five-color solution rather easily:

```
-- Best Individual = [0, 3, 0, 2, 4, 4, 2, 2, 2, 4, 1, 4, 3, 1, 3, 3, 4, 4,
2, 2, 4, 3, 0]
-- Best Fitness = 5.0

number of colors = 5
Number of violations = 0
Cost = 5
```

The plot of the colored graph looks as follows:



A plot of the Mycielski graph properly colored by the program using five colors

Now, if we attempt to decrease the maximum number of colors to four, we will always get at least one violation.

You are encouraged to try out other graphs and experiment with the various settings of the algorithm.

Summary

In this chapter, you were introduced to constraint satisfaction problems, a close relative of the previously studied combinatorial optimization problems. Then, we explored three classic constraint satisfaction cases – the N-Queen problem, the nurse scheduling problem, and the graph coloring problem. For each of these problems, we followed the now-familiar process of finding an appropriate representation for a solution, creating a class that encapsulates the problem and evaluates a given solution, and creating a genetic algorithm solution that utilizes that class. We ended up with valid solutions for the problems while getting acquainted with the concept of hard constraints versus soft constraints.

So far, we have been looking into discrete search problems consisting of states and state transitions. In the next chapter, we will study search problems in a continuous space to demonstrate the versatility of the genetic algorithms approach.

Further reading

For more information on the topics that were covered in this chapter, please refer to the following resources:

- Constraint Satisfaction Problems, from the book *Artificial Intelligence with Python* by *Prateek Joshi*, January 2017
- Introduction to graph theory, from the book *Python Data Science Essentials – Second Edition* by *Alberto Boschetti*, *Luca Massaron*, October 2016
- NetworkX Tutorial: `https://networkx.github.io/documentation/stable/tutorial.html`

6
Optimizing Continuous Functions

This chapter describes how continuous search-space optimization problems can be solved by genetic algorithms. We will start by describing the chromosomes and genetic operators commonly used for genetic algorithms with real number-based populations, and go over the tools offered by the DEAP framework for this domain. We will then cover several hands-on examples of continuous function optimization problems and their Python-based solutions using the DEAP framework. These include the optimization of the Eggholder function, Himmelblau's function, as well as the constrained optimization of Simionescu's function. Along the way, we will learn about finding multiple solutions using niching and sharing and handling constraints.

By the end of this chapter, you will be able to do the following:

- Understand chromosomes and genetic operators used for real numbers
- Use DEAP to optimize continuous functions
- Optimize the Eggholder function
- Optimize Himmelblau's function
- Perform constrained optimization with Simionescu's function
- Utilize parallel and serial niching to locate multiple optima for multi-modal functions

Technical requirements

In this chapter, we will be using Python 3 with the following supporting libraries:

- `deap`
- `numpy`

- `matplotlib`
- `seaborn`

The programs used in this chapter can be found in the book's GitHub repository at: `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter06`.

Check out the following video to see the Code in Action:
`http://bit.ly/2REfGuT`

Chromosomes and genetic operators for real numbers

In previous chapters, we focused on search problems that inherently deal with the methodic evaluation of states and transitions between states. Consequently, the solutions for these problems were best represented by lists (or arrays) of binary or integer parameters. In contrast to that, this chapter covers problems where the solution space is continuous, meaning the solutions are made up of real (floating-point) numbers. As we mentioned in `Chapter 2`, *Understanding the Key Components of Genetic Algorithms*, representing real numbers using binary or integer lists was found to be far from ideal and, instead, lists (or arrays) of real-valued numbers are now considered to be a simpler and better approach.

Reiterating the example from `Chapter 2`, *Understanding the Key Components of Genetic Algorithms*, if we have a problem involving three real-valued parameters, the chromosome will look like the following:

$[x_1, x_2, x_3]$

Where x_1, x_2, x_3 represent real numbers, such as:

```
[1.23, 7.2134, -25.309] or [-30.10, 100.2, 42.424]
```

In addition, we mentioned that while the various selection methods work the same for either integer-based or real-based chromosomes, specialized crossover and mutation methods are needed for the real-coded chromosomes. These operators are usually applied on a dimension-by-dimension basis, illustrated as follows.

Suppose we have two chromosomes: [x_1, x_2, x_3] and [y_1, y_2, 3_3]. As the crossover operation is applied separately to each dimension, an offspring [o_1, o_2, o_3] will be created as follows:

- o_1 is the result of a crossover operator between x_1 and y_1.
- o_2 is the result of a crossover operator between x_2 and y_2.
- o_3 is the result of a crossover operator between x_3 and y_3.

Similarly, the mutation operator will be individually applied to each dimension, so each of the components o_1, o_2, and o_3 can be subject to mutation.

Some of the commonly used real-coded operators are:

- **Blend Crossover** (also known as **BLX**), where each offspring is randomly selected from the following interval created by its parents:

$$[parent_1 - \alpha(parent_2 - parent_1),\ \ parent_2 + \alpha(parent_2 - parent_1)]$$

  The α value is commonly set to 0.5, resulting in a selection interval twice as wide as the interval between the parents.

- **Simulated Binary Crossover** (**SBX**), where the two offspring are created from the two parents using the following formula, guaranteeing that the average of the offspring values is equal to that of the parents' values:

$$offspring_1 = \frac{1}{2}[(1+\beta)parent_1 + (1-\beta)parent_2]$$
$$offspring_2 = \frac{1}{2}[(1-\beta)parent_1 + (1+\beta)parent_2]$$

  The value of β, also known as the **spread factor**, is calculated using a combination of a randomly chosen value and a pre-determined parameter known as η (eta), **distribution index**, or **crowding factor**. With larger values of η, offspring will tend to be more similar to their parents. Common values of η are between 10 and 20.

- **Normally distributed (or Gaussian) mutation**, where the original value is replaced with a random number that is generated using a normal distribution, with predetermined values for mean and standard deviation.

In the next section, we will see how real-coded chromosomes and genetic operators are supported by the DEAP framework.

Using DEAP with continuous functions

When solving discrete search problems, the DEAP framework can be used for optimizing continuous functions in a very similar manner to what we have seen so far. All that's needed are a few subtle modifications.

For the chromosome encoding, we can use a list (or array) of floating-point numbers. One thing to keep in mind, though, is that the existing genetic operators of DEAP will not work well with individual objects extending the `numpy.ndarray` class due to the way these objects are being sliced, as well as the way they are being compared to each other. Using `numpy.ndarray`-based individuals will require redefining the genetic operators accordingly. This is further covered in the DEAP documentation, under *Inheriting from NumPy*. For this reason, as well as for performance reasons, ordinary Python lists or arrays of floating-point numbers are generally preferred when using DEAP.

As for real-coded genetic operators, the DEAP framework offers several implementations out of the box, contained in the crossover and the mutation modules:

- `cxBlend()` is DEAP's implementation of Blend Crossover, using the argument *alpha* as the α value.
- `cxSimulatedBinary()` implements Simulated Binary Crossover, using the argument *eta* as the η (crowding factor) value.
- `mutGaussian()` implements normally distributed mutation, using the arguments *mu* and *sigma* as the values for the mean and standard deviation, respectively.

In addition, since the optimization of continuous functions is typically done on a particular bounded region rather than on the entire space, DEAP provides a couple of operators that accept boundary parameters and guarantee that the resulting individuals will reside within these boundaries:

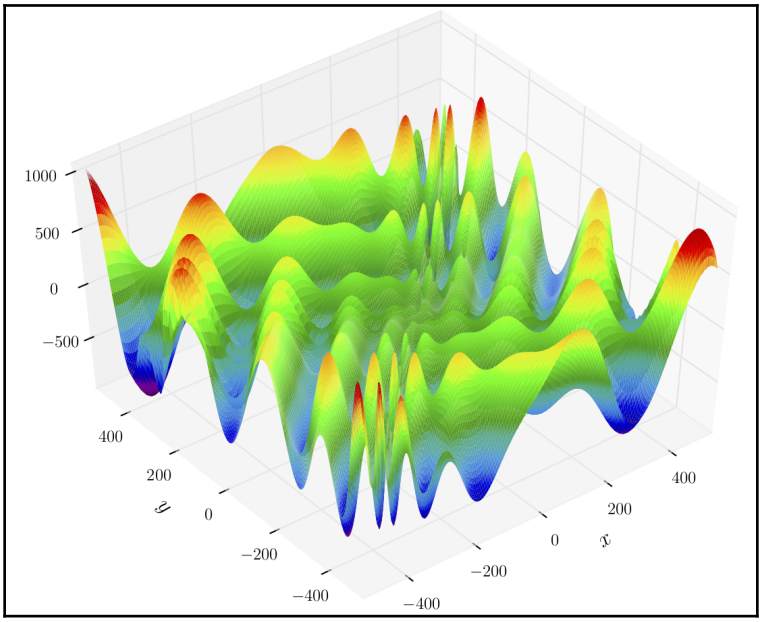
- `cxSimulatedBinaryBounded()` is a bounded version of the `cxSimulatedBinary()` operator, accepting the arguments `low` and `up` as the lower and upper boundaries of the search space, respectively.

- `mutPolynomialBounded()` is a bounded mutation operator that uses a polynomial function (instead of Gaussian) for the probability distribution. This operator also accepts the arguments `low` and `up` as the lower and upper boundaries of the search space. In addition, it uses the `eta` parameter as a crowding factor, where a high value will yield a mutant close to its original value, while a small value will produce a mutant very different from its original value.

In the next section, we will demonstrate the usage of bounded operators when optimizing a classic benchmark function.

Optimizing the Eggholder function

The Eggholder function, depicted in the following diagram, is often used as a benchmark for function optimization algorithms. Finding the single global minimum of this function is considered a difficult task due to the large number of local minima, which give it the eggholder shape:



The Eggholder function
Source: https://en.wikipedia.org/wiki/File:Eggholder_function.pdf. Image by Gaortizg.

The function can be mathematically expressed as follows:

$$f(x,y) = -(y+47) \cdot sin\sqrt{\left|\frac{x}{2}+(y+47)\right|} - x \cdot sin\sqrt{|x-(y+47)|}$$

It is usually evaluated on the search space bounded by [-512, 512] in each dimension.

The global minimum of the function is known to be at:

x=512, y = 404.2319

Where the function's value is -959.6407.

In the next subsection, we will attempt to find the global minimum using the genetic algorithms method.

Optimizing the Eggholder function with genetic algorithms

The genetic algorithm-based program we created for optimizing the Eggholder function resides in the Python `01-optimize-eggholder.py` program located at:

`https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter06/01-optimize-eggholder.py`

The following steps highlight the main parts of this program:

1. The program starts by setting the function constants, namely the number of input dimensions—2, as this function is defined over the *x*-*y* plane, and the boundaries that were mentioned previously:

```
DIMENSIONS = 2  # number of dimensions
BOUND_LOW, BOUND_UP = -512.0, 512.0  # boundaries, same for all
dimensions
```

2. Since we are dealing with floating-point numbers confined by certain boundaries, we next define a helper function that creates random floating-point numbers, uniformly distributed within the given range:

This function assumes that the upper and lower boundaries are the same for all dimensions.

```
def randomFloat(low, up):
    return [random.uniform(l, u) for l, u in zip([low] *
DIMENSIONS, [up] * DIMENSIONS)]
```

3. We next define the `attrFloat` operator. This operator utilizes the previous helper function to create a single, random floating-point number within the given boundaries. The `attrFloat` operator is then used by the `individualCreator` operator to create random individuals. This is followed by `populationCreator`, which can generate the desired number of individuals:

```
toolbox.register("attrFloat", randomFloat, BOUND_LOW, BOUND_UP)
toolbox.register("individualCreator", tools.initIterate,
creator.Individual, toolbox.attrFloat)
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

4. Given that the object to be minimized is the Eggholder function, we use it directly as the fitness evaluator. As the individual is a list of floating-point numbers with a dimension (or length) of 2, we extract the x and y values from the individual accordingly, and then calculate the function:

```
def eggholder(individual):
    x = individual[0]
    y = individual[1]
    f = (-(y + 47.0) * np.sin(np.sqrt(abs(x/2.0 + (y + 47.0))))
- x * np.sin(np.sqrt(abs(x - (y + 47.0)))))
    return f,   # return a tuple

toolbox.register("evaluate", eggholder)
```

5. Next are the genetic operators. Given that the selection operator is independent of the individual type, and we've had a good experience so far using the tournament selection with a tournament size of 2, coupled with the elitist approach, we'll continue to use it here. The crossover and mutation operators, on the other hand, need to be specialized for floating-point numbers within given boundaries, and therefore we use the DEAP-provided `cxSimulatedBinaryBounded` operator for crossover, and the `mutPolynomialBounded` operator for mutation:

```
# Genetic operators:
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxSimulatedBinaryBounded,
low=BOUND_LOW, up=BOUND_UP, eta=CROWDING_FACTOR)
toolbox.register("mutate", tools.mutPolynomialBounded,
low=BOUND_LOW, up=BOUND_UP, eta=CROWDING_FACTOR,
indpb=1.0/DIMENSIONS)
```

6. As we have done multiple times, we use our modified version of DEAP's simple genetic algorithm flow, where we added elitism—keeping the best individuals (members of the hall of fame) and moving them to the next generation, untouched by the genetic operators:

```
population, logbook = elitism.eaSimpleWithElitism(population,
toolbox, cxpb=P_CROSSOVER,                mutpb=P_MUTATION,
ngen=MAX_GENERATIONS, stats=stats, halloffame=hof,
verbose=True)
```

7. We will start with the following parameters for the genetic algorithm settings. As the Eggholder function may be somewhat difficult to optimize, we use a relatively large population size considering the low dimension count:

```
# Genetic Algorithm constants:
POPULATION_SIZE = 300
P_CROSSOVER = 0.9
P_MUTATION = 0.1
MAX_GENERATIONS = 300
HALL_OF_FAME_SIZE = 30
```

8. In addition to the previous ordinary genetic algorithm constants, we now need a new one, the crowding factor (`eta`) that is used by both the crossover and mutation operations:

```
CROWDING_FACTOR = 20.0
```

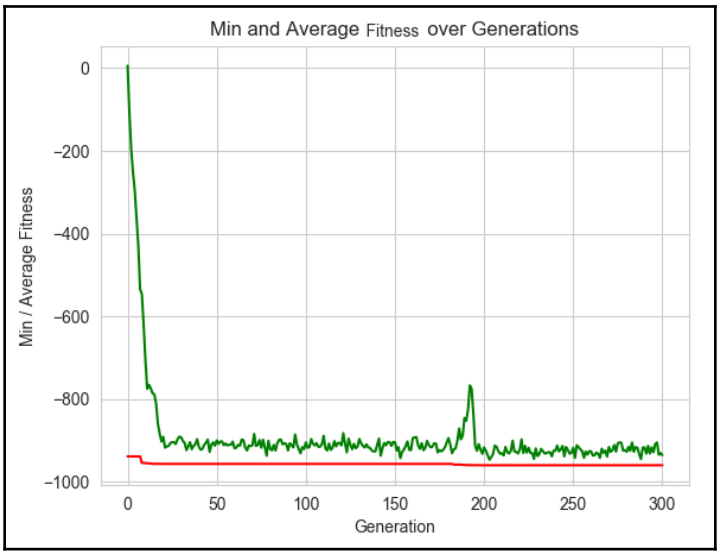
It is also possible to define separate crowding factors for crossover and for mutation.

We are finally ready to run the program. The results obtained with these settings are shown as follows:

```
-- Best Individual = [512.0, 404.23180541839946]
-- Best Fitness = -959.6406627208509
```

This means that we have found the global minimum.

If we examine the statistics plot generated by the program, shown as follows, we can tell that the algorithm found some local minima values right away and then made small incremental improvements until it eventually found the global minima:

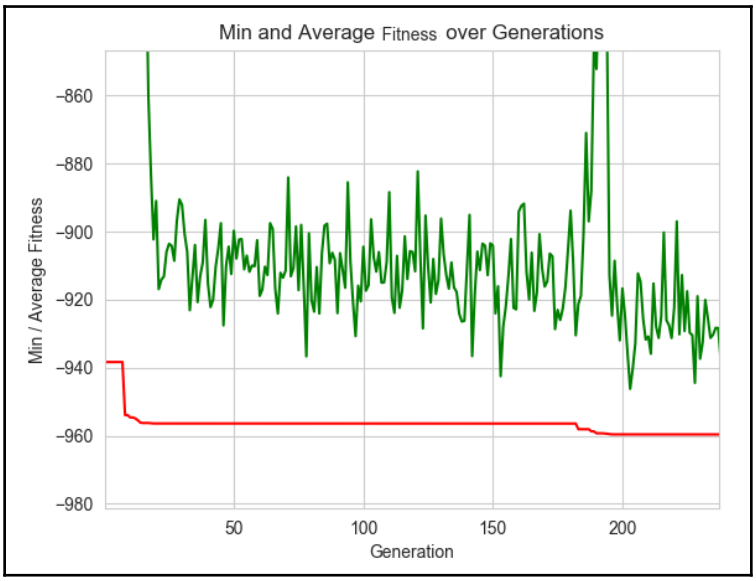


Stats of the first program optimizing the Eggholder function

One interesting area is around generation 180—let's explore it further in the next subsection.

Improving the speed with an increased mutation rate

If we zoom in at the lower part of the fitness axis, we will notice a relatively large improvement of the best result found (red line) around generation 180, accompanied by a large swing of the average results (green line):



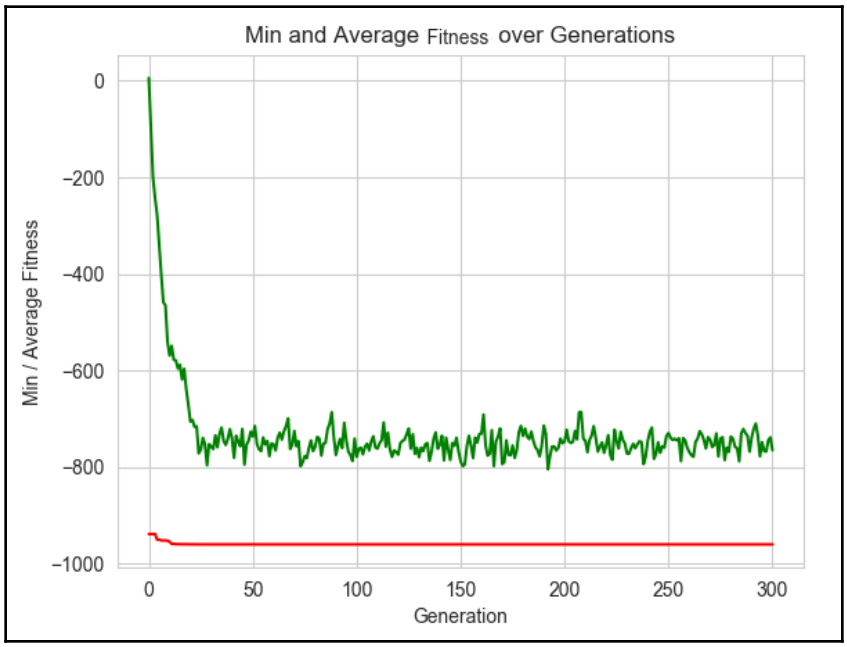
Enlarged section of the first program's stats graph

One way to interpret this observation is that perhaps introducing more noise can lead to better results faster. This could be another manifestation of the familiar principle of **exploration versus exploitation** we've discussed several times before—increasing the exploration (which manifests itself as noise in the diagram) may help us locate the global minimum faster. An easy way to increase the measure of exploration is to boost the probability for mutations. Hopefully, the use of elitism—keeping the best results untouched—will keep us from over-exploring, which leads to random search-like behavior.

To test this idea, let's increase the probability of mutation from 0.1 to 0.5:

```
P_MUTATION = 0.5
```

Running the modified program, we again found the global minimum, but much faster, as is evident from the output, as well as from the statistic plot shown as follows, where the red line (the best result) reaches the optimum quickly, while the average score (green) is noisier than before and is more distanced from the best result:

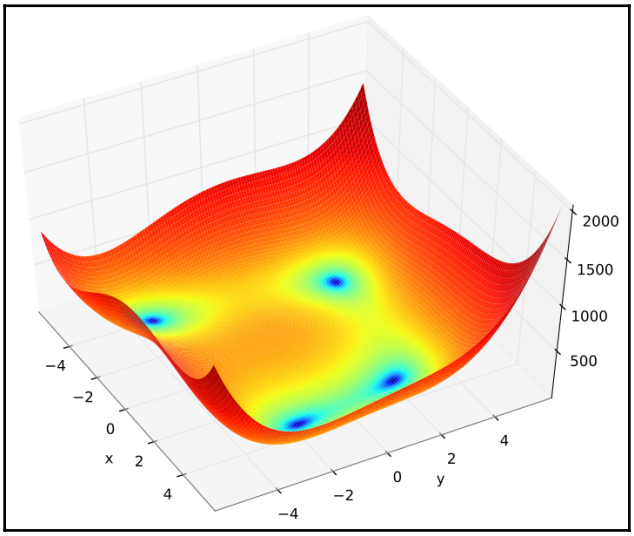


Stats of the program optimizing the Eggholder function with an increased mutation probability

We will keep this idea in mind when dealing with our next benchmark function, known as Himmelblau's function.

Optimizing Himmelblau's function

Another frequently used function for benchmarking optimization algorithms is Himmelblau's function, depicted in the following diagram:



Himmelblau's function
Source: https://commons.wikimedia.org/wiki/File:Himmelblau_function.svg.
Image by Morn the Gorn. Released to the public domain.

The function can be mathematically expressed as follows:

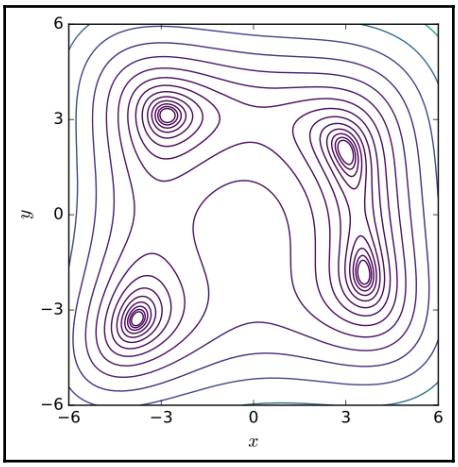
$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

It is usually evaluated on the search space bounded by [-5, 5] in each dimension.

Although this function seems simpler in comparison to the Eggholder function, it draws interest as it is *multi-modal*, in other words, it has more than one global minimum. To be exact, the function has four global minima evaluating to 0, which can be found in the following locations:

- x=3.0, y=2.0
- x=–2.805118, y=3.131312
- x=–3.779310, y=–3.283186
- x=3.584458, y=–1.848126

These locations are depicted in the function contour diagram, as follows:



Contour diagram of Himmelblau's function
Source: https://commons.wikimedia.org/wiki/File:Himmelblau_contour.svg. Image by: Nicoguaro.
Licensed under Creative Commons CC BY 4.0: https://creativecommons.org/licenses/by/4.0/deed.en.

When optimizing multi-modal functions, we are often interested in finding all (or most) minima locations. However, let's start with finding one, which we are going to do in the next subsection.

Optimizing Himmelblau's function with genetic algorithms

The genetic algorithm-based program we created for finding a single minimum of Himmelblau's function resides in the Python `02-optimize-himmelblau.py` program, located at:

`https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter06/02-optimize-himmelblau.py`

The program is similar to the one we used for optimizing the Eggholder function, with a few differences highlighted as follows:

1. We set the boundaries for this function to [-5.0, 5.0]:

```
BOUND_LOW, BOUND_UP = -5.0, 5.0  # boundaries for all
dimensions
```

2. We now use Himmelblau's function as the fitness evaluator:

```
def himmelblau(individual):
    x = individual[0]
    y = individual[1]
    f = (x ** 2 + y - 11) ** 2 + (x + y ** 2 - 7) ** 2
    return f,  # return a tuple

toolbox.register("evaluate", himmelblau)
```

3. Since the function we optimize has several minima, it may be interesting to observe the distribution of the solutions found at the end of the run. We, therefore, add a scatter graph containing the locations of the four global minima and the final population on the same *x*-*y* plane:

```
plt.figure(1)
globalMinima = [[3.0, 2.0], [-2.805118, 3.131312], [-3.779310,
-3.283186], [3.584458, -1.848126]]
plt.scatter(*zip(*globalMinima), marker='X', color='red',
zorder=1)
plt.scatter(*zip(*population), marker='.', color='blue',
zorder=0)
```

4. We also print the members of the hall of fame—the best individuals found during the run:

```
print("- Best solutions are:")
for i in range(HALL_OF_FAME_SIZE):
    print(i, ": ", hof.items[i].fitness.values[0], " -> ",
hof.items[i])
```

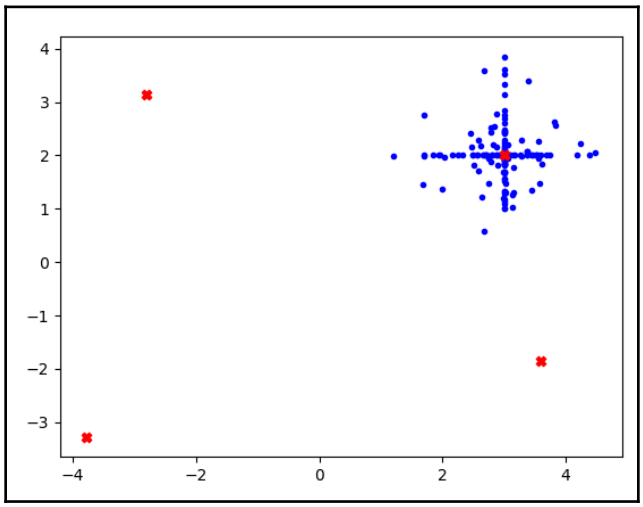
Running the program, the results indicate that we found one of the four minima (x=3.0, y=2.0):

```
-- Best Individual = [2.9999999999987943, 2.0000000000007114]
-- Best Fitness = 4.523490304795033e-23
```

The printout of the hall of fame members suggests they all represent the same solution:

```
- Best solutions are:
0 : 4.523490304795033e-23 -> [2.9999999999987943, 2.0000000000007114]
1 : 4.523732642865117e-23 -> [2.9999999999987943, 2.000000000000697]
2 : 4.523900512465748e-23 -> [2.9999999999987943, 2.0000000000006937]
3 : 4.5240633333565856e-23 -> [2.9999999999987943, 2.00000000000071]
...
```

The following diagram, illustrating the distribution of the entire population, further confirms that the genetic algorithms have converged to one of the four functions' minima—the one residing at (x=3.0, y=2.0):



Scatter graph of the population at the end of the first run, alongside the four functions' minima

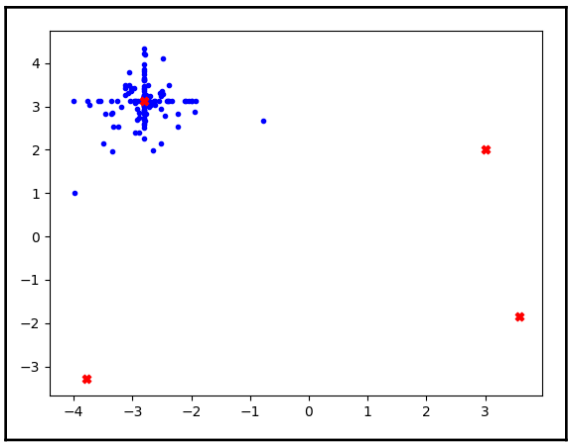
In addition, it is evident that many of the individuals in the population have either the x or y component of the minimum we found.

The previous results represent what we generally expect from the genetic algorithm—to identify a global optimum and converge to it. Since, in this case, we have several minima, it is expected to converge to one of them. Which one it will be is largely based on the random initialization of the algorithm. As you may recall, in all our programs so far, we have been using a fixed random seed (of value 42):

```
RANDOM_SEED = 42
random.seed(RANDOM_SEED)
```

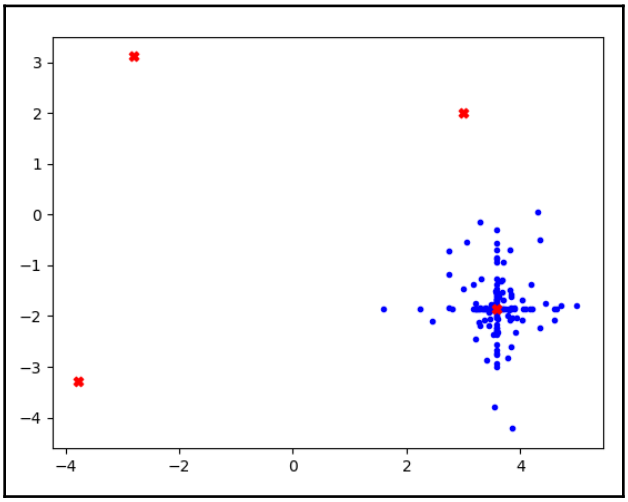
This is done to enable the repeatability of the results; however, in real life, we will typically use different random seed values for different runs, either by commenting out these lines, or by explicitly setting the constant to different values.

For example, if we set the seed value to 13, we will end up with the solution (x=–2.805118, y=3.131312), as illustrated in the following diagram:



Scatter graph of the population at the end of the second run, alongside the four functions' minima

If we proceed to change the seed value to 17, the program execution will yield the solution (x=3.584458, y=–1.848126), as illustrated by the following diagram:



Scatter graph of the population at the end of the third run, alongside the four functions' minima

However, what if we wanted to find all global minima in a single run? As we will see in the next subsection, genetic algorithms offer us a way to pursue this goal.

Using niching and sharing to find multiple solutions

In `Chapter 2`, *Understanding the Key Components of Genetic Algorithms*, we mentioned that niching and sharing in genetic algorithms mimic the way a natural environment is divided into multiple sub-environments, or niches. These niches are populated by different species, or sub-populations, taking advantage of the unique resources available in each niche, while specimens that coexist in the same niche have to compete over the same resources. Implementing a sharing mechanism within the genetic algorithm will encourage individuals to explore new niches and can be used for finding several optimal solutions, each considered a niche. One common way to accomplish sharing is to divide the raw fitness value of each individual with (some function of) the combined distances from all the other individuals, effectively penalizing a crowded population by sharing the local bounty between its individuals.

Let's try and apply this idea to Himmelblau's function optimization process and see whether it can help locate all four minima in a single run. This attempt is implemented in the `03-optimize-himmelblau-sharing.py` program, located at:
`https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter06/03-optimize-himmelblau-sharing.py`

The program is based on the previous one, but we had to make some important modifications, described as follows:

1. For starters, the implementation of a sharing mechanism usually requires us to optimize a function that produces positive fitness values and to look for maxima values rather than minima. This enables us to divide the raw fitness values as a way to decrease fitness and practically share the resources between neighboring individuals. As Himmelblau's function produces values between 0 and (roughly) 2,000, we can instead use a modified function that returns 2,000 minus the original value, which will guarantee that all function values are positive, while transforming the minima points into maxima points that return the value of 2,000. Note the locations of these points is not going to change, so finding them will still serve our original purpose:

```
def himmelblauInverted(individual):
    x = individual[0]
    y = individual[1]
```

```
        f = (x ** 2 + y - 11) ** 2 + (x + y ** 2 - 7) ** 2
        return 2000.0 - f,  # return a tuple

toolbox.register("evaluate", himmelblauInverted)
```

2. To complete the conversion, we redefine the fitness strategy to be a maximizing one:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

3. To enable the implementation of sharing, we first create two additional constants:

```
DISTANCE_THRESHOLD = 0.1
SHARING_EXTENT = 5.0
```

4. Next, we need to implement the sharing mechanism. One convenient location for this implementation is within the selection genetic operator. The selection operator is where the fitness values of all individuals are examined and used to select the parents for the next generation. This enables us to inject some code that recalculates these fitness values just before the selection takes place, then retrieve the original fitness values before continuing, for the purpose of tracking. To make this happen, we implemented a new `selTournamentWithSharing()` function, which has the same signature as the original `tools.selTournament()` function we have been using until now:

```
def selTournamentWithSharing(individuals, k, tournsize,
fit_attr="fitness"):
```

This function starts by setting the original fitnesses aside so they can be retrieved later. It then iterates over each individual and calculates a number, `sharingSum`, by which its fitness value will be divided. This sum value is accumulated by calculating the distance between the location of the current individual and the location of each of the other individuals in the population. If the distance is smaller than the threshold defined by the `DISTANCE_THRESHOLD` constant, the following value is added to the accumulating sum:

$$1 - \frac{distance}{DISTANCE\_THRESHOLD} \times \frac{1}{SHARING\_EXTENT}$$

This means that the reduction in the fitness value will be larger when:

- The (normalized) distance between the individuals is smaller
- The sharing extent constant is larger

After recalculating the fitness value for each individual, tournament selection is conducted using the new fitness values:

```
selected = tools.selTournament(individuals, k, tournsize,
fit_attr)
```

Lastly, the original fitness values are retrieved:

```
for i, ind in enumerate(individuals):
    ind.fitness.values = origFitnesses[i],
```

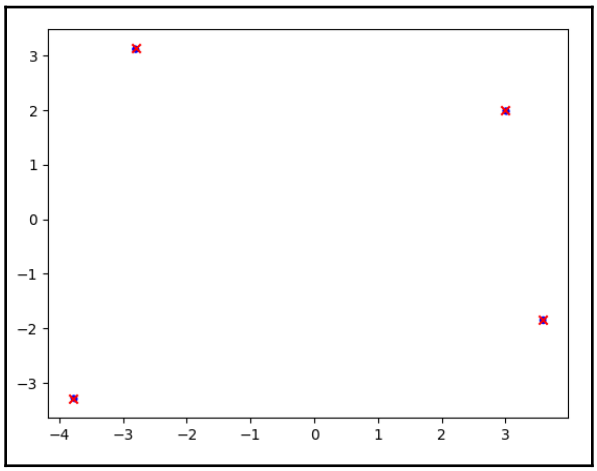
5. As a final touch, we added a plot showing the locations of the best individuals—the hall of fame members—on the *x-y* plane, alongside the known optima location, similar to what we already do for the entire population:

```
plt.figure(2)
plt.scatter(*zip(*globalMaxima), marker='x', color='red',
zorder=1)
plt.scatter(*zip(*hof.items), marker='.', color='blue',
zorder=0)
```

When we run this program, the results don't disappoint. Examining the members of the hall of fame, it seems that we have located all four optima locations:

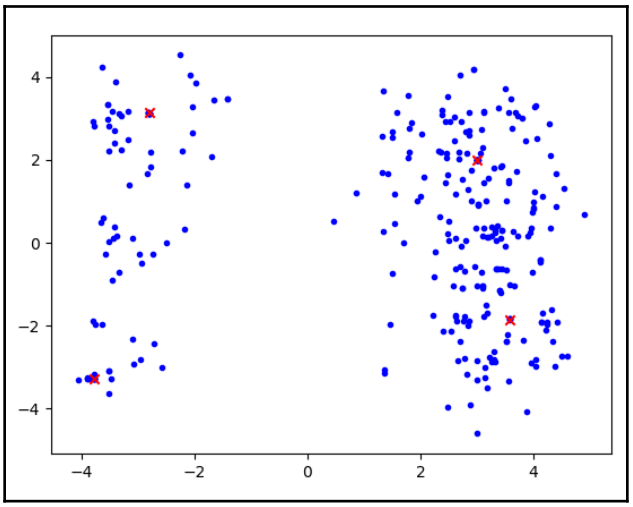
```
- Best solutions are:
0 : 1999.9997428476076 -> [3.00161237138945, 1.9958270919300878]
1 : 1999.9995532774788 -> [3.585506608049694, -1.8432407550446581]
2 : 1999.9988186889173 -> [3.585506608049694, -1.8396197402430106]
3 : 1999.9987642838498 -> [-3.7758887140006174, -3.285804345540637]
4 : 1999.9986563457114 -> [-2.8072634380293766, 3.125893564009283]
...
```

The following diagram illustrating the distribution of the hall of fame members further confirms that:



Scatter graph of the best solutions at the end of the run, alongside the four functions' minima, when using niching

Meanwhile, the diagram depicting the distribution of the entire population demonstrates how the population is scattered around the four solutions:



Scatter graph of the population at the end of the run, alongside the four functions' minima, when using niching

As impressive as this may seem, we need to remember that what we did here can prove harder to implement in real-life situations. For one, the modifications we added to the selection process increase the calculation complexity and the time consumed by the algorithm. In addition, the population size usually needs to be increased so it can sufficiently cover all areas of interest. The values of the sharing constants may be difficult to determine in some cases, for example, if we don't know in advance how close together the various peaks may be. However, we can always use this technique to roughly locate areas of interest and then further explore each one of them using the standard version of the algorithm.

An alternative approach for finding several optima points falls within the realm of constrained optimization, which is the subject of the next section.

Simionescu's function and constrained optimization

At first glance, Simionescu's function may not look particularly interesting. However, it has a constraint attached to it that makes it intriguing to work with as well as pleasant to look at.

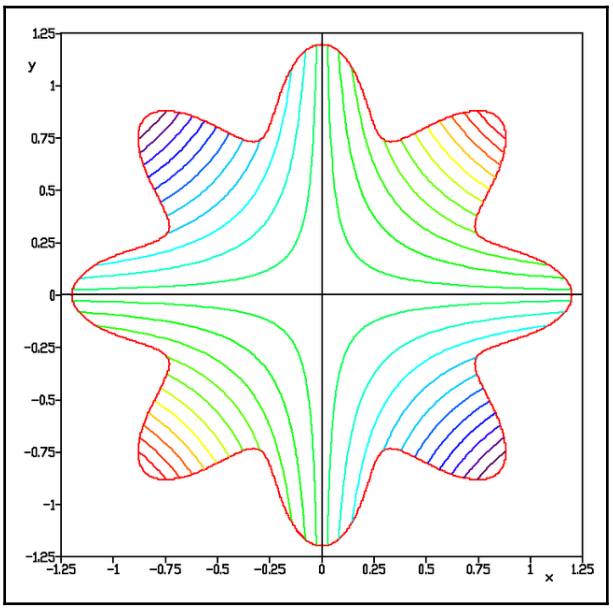
The function is usually evaluated on the search space bounded by [-1.25, 1.25] in each dimension and can be mathematically expressed as follows:

$$f(x, y) = 0.1xy$$

Where the values of x, y are subject to the following condition:

$$x^2 + y^2 \leqslant \left[1 + 0.2 \cdot cos(8 \cdot arctan\frac{x}{y})\right]^2$$

This constraint effectively limits the values of x and y that are considered valid for this function. The result is depicted in the following contour diagram:



Contour diagram of the constrained Simionescu's function
Source: https://commons.wikimedia.org/wiki/File:Simionescu%27s_function.PNG.

The flower-shaped border is created by the constraint, while the colors of the contours denote the actual value—red for the highest values and purple for the lowest. If it weren't for the constraint, the minima points would have been at (1.25, -1.25) and (-1.25, 1.25). However, after applying the constraint, the global minima of the function are located at the following locations:

- x=0.84852813, y=–0.84852813
- x=–0.84852813, y=0.84852813

These represent the tips of the two opposite petals containing the purple contours. Both minima evaluate to the value of -0.072.

In the next subsection, we will attempt to find these minima using our real-coded genetic algorithms approach.

Constrained optimization with genetic algorithms

We have already dealt with constraints in `Chapter 5`, *Constraint Satisfaction*, when we tackled constraints within the realm of search problems. However, while search problems presented us with invalid states or combinations, here we need to address constraints in the continuous space, defined as mathematical inequalities.

The approaches for both cases, however, are similar, and the differences lie in the implementation. Let's revisit these approaches:

- The best approach, when available, is to eliminate the possibility for a constraint violation. We have actually been doing it all along in this chapter as we have used bounded regions for our functions. These are actually simple constraints on each input variable. We were able to go around them by generating initial populations within the given boundaries, and by utilizing bounded genetic operators such as `cxSimulatedBinaryBounded()`, which produced results within the given boundaries. Unfortunately, this approach can prove difficult to implement when the constraints are more complex than just upper and lower bounds for an input variable.
- Another approach is to discard candidate solutions that violate any given constraint. As we mentioned before, this approach leads to loss of information contained in these solutions and can considerably slow down the optimization process.
- The next approach is to repair any candidate solution that violates a constraint by modifying it so it will no longer violate the constraint(s). This can prove difficult to implement and, at the same time, may lead to significant loss of information.
- Finally, the approach that worked for us in `Chapter 5`, *Constraint Satisfaction*, was to penalize candidate solutions that violated a constraint by degrading the solution's score and making it less desirable. For search problems, we have implemented this approach by creating a cost function that added a fixed cost to each constraint violation. Here, in the continuous space case, we can either use a fixed penalty or increase the penalty based on the degree by which the constraint was violated.

When taking the last approach—penalizing the score for constraint violations—we can utilize a feature offered by the DEAP framework, namely the *penalty function*, as we will demonstrate in the next subsection.

Optimizing Simionescu's function using genetic algorithms

The genetic algorithm-based program we created for optimizing Simionescu's function resides in the Python `04-optimize-simionescu.py` program, located at:

`https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter06/04-optimize-simionescu.py`

The program is very similar to the first one we used in this chapter, created originally for the Eggholder function, with the following highlighted differences:

1. The constants setting the boundaries are adjusted to match the domain of Simionescu's function:

```
BOUND_LOW, BOUND_UP = -1.25, 1.25
```

2. In addition, a new constant determines a fixed penalty (or cost) for violating the constraint:

```
PENALTY_VALUE = 10.0
```

3. The fitness is now determined by the definition of Simionescu's function:

```
def simionescu(individual):
    x = individual[0]
    y = individual[1]
    f = 0.1 * x * y
    return f,  # return a tuple

toolbox.register("evaluate",simionescu)
```

4. Here is where the interesting part begins: we now define a new `feasible()` function, which specifies the valid input domain using the constraints. This function returns a value of *True* for *x, y* values that comply with the constraints, and a value of *False* otherwise:

```
def feasible(individual):
    x = individual[0]
    y = individual[1]
    return x**2 + y**2 <= (1 + 0.2 * math.cos(8.0 *
math.atan2(x, y)))**2
```

5. We then use DEAP's `toolbox.decorate()` operator in combination with the `tools.DeltaPenalty()` function to modify (decorate) the original fitness function, so the fitness values will be penalized whenever the constraints are not satisfied. `DeltaPenalty` accepts the `feasible()` function and the fixed penalty value as parameters:

```
toolbox.decorate("evaluate", tools.DeltaPenalty(feasible,
PENALTY_VALUE))
```

The `DeltaPenalty` function can also accept a third parameter that represents the distance from the feasible region, causing the penalty to increase with the distance.

Now the program is ready to use! The results indicate that we have indeed found one of the two known minima locations:

```
-- Best Individual = [0.8487712463169383, -0.8482833185888866]
-- Best Fitness = -0.07199984895485578
```

What about the second location? Read on—we will be looking for it in the next subsection.

Using constraints to find multiple solutions

Earlier in this chapter, when optimizing Himmelblau's function, we were looking for more than one minimum location, and observed two possible ways to do that—one was changing the random seed, and the other was using niching and sharing. Here, we will demonstrate a third option, powered by constraints!

The niching technique we used for Himmelblau's function is sometimes called parallel niching as it attempts to locate several solutions at the same time. As we already mentioned, it is prone to several practical drawbacks. Serial niching (or sequential niching), on the other hand, is a method used to find one solution at a time. To implement serial niching, we use the genetic algorithm as usual and find the best solution. We then update the fitness function so that the area of the solution(s) already found is penalized, thereby encouraging the algorithm to explore other areas of the problem space. This can be repeated multiple times until no additional viable solutions are found.

Interestingly, penalizing the areas around the previously found solutions can be implemented by imposing constraints on the search space and, as we just learned how to apply constraints to the function at hand, we can use this knowledge to implement serial niching, demonstrated as follows.

To find the second minimum for Simionescu's function, we created the Python `05-optimize-simionescu-second.py` program, located at:

`https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter06/05-optimize-simionescu-second.py`

The program is almost identical to the previous one, with a couple of changes, as follows:

1. We first add a constant that defines the distance threshold from previously found solutions—new solutions closer than this threshold value to any of the old ones will be penalized:

```
DISTANCE_THRESHOLD = 0.1
```

2. We then add a second constraint to the definition of the `feasible()` function using a conditional statement with multiple clauses. The new constraint applies to input values closer than the threshold to the already found solution (x=0.848, y = -0.848):

```
def feasible(individual):
x = individual[0]
y = individual[1]

if x**2 + y**2 > (1 + 0.2 * math.cos(8.0 * math.atan2(x,
y)))**2:
    return False
elif (x - 0.848)**2 + (y + 0.848)**2 < DISTANCE_THRESHOLD**2:
    return False
else:
    return True
```

When running this program, the results indicate that we have indeed found the second minimum:

```
-- Best Individual = [-0.8473430282562487, 0.8496942440090975]
-- Best Fitness = -0.07199824938105727
```

You are encouraged to add this minimum point as another constraint to the `feasible()` function, and verify that running the program again does not find any other equally minimum-valued locations in the input space.

Summary

In this chapter, you were introduced to continuous search-space optimization problems and how they can be represented and solved using genetic algorithms, and specifically by utilizing the DEAP framework. We then explored several hands-on examples of continuous function optimization problems—the Eggholder function, Himmelblau's function, and Simionescu's function—along with their Python-based solutions. In addition, we covered approaches for finding multiple solutions and for handling constraints.

In the next four chapters of the book, we will demonstrate how the various techniques we've learned so far in this book can be applied when solving machine learning and artificial intelligence-related problems. The first of these chapters will provide a quick overview of supervised learning and then demonstrate how genetic algorithms can improve the outcome of learning models by selecting the most relevant portions of the given dataset.

Further reading

For more information, please refer to the following resources:

- **Mathematical optimization: finding minima of functions**: `http://scipy-lectures.org/advanced/mathematical_optimization/`
- **Optimization Test Functions and Datasets**: `https://www.sfu.ca/~ssurjano/optimization.html`
- **Introduction to Constrained Optimization**: `https://web.stanford.edu/group/sisl/k12/optimization/MO-unit3-pdfs/3.1introandgraphical.pdf`
- **Constraint handling in DEAP**: `https://deap.readthedocs.io/en/master/tutorials/advanced/constraints.html`

3
Section 3: Artificial Intelligence Applications of Genetic Algorithms

This section focuses on using genetic algorithms to improve the results obtained with various machine learning algorithms.

This section comprises the following chapters:

- `Chapter 7`, *Enhancing Machine Learning Models Using Feature Selection*
- `Chapter 8`, *Hyperparameter Tuning of Machine Learning Models*
- `Chapter 9`, *Architecture Optimization of Deep Learning Networks*
- `Chapter 10`, *Reinforcement Learning with Genetic Algorithms*

7
Enhancing Machine Learning Models Using Feature Selection

This chapter describes how genetic algorithms can be used to improve the performance of supervised machine learning models by selecting the best subset of features from the provided input data. This chapter will start with a brief introduction to machine learning and then describe the two main types of supervised machine learning tasks – regression and classification. We will then discuss the potential benefits of feature selection when it comes to the performance of these models. Next, we will demonstrate how genetic algorithms can be utilized to pinpoint the genuine features that are generated by the *Friedman-1 Test* regression problem. Then, we will use the real-life Zoo dataset to create a classification model and improve its accuracy – again by applying genetic algorithms to isolate the best features for the task.

In this chapter, we will cover the following topics:

- Understand the basic concepts of supervised machine learning, as well as regression and classification tasks
- Understand the benefits of feature selection on the performance of supervised learning models
- Enhance the performance of a regression model for the Friedman-1 Test regression problem, using feature selection carried out by a genetic algorithm coded with the DEAP framework
- Enhance the performance of a classification model for the Zoo dataset classification problem, using feature selection carried out by a genetic algorithm coded with the DEAP framework

We will start this chapter with a quick review of supervised machine learning. If you are a seasoned data scientist, feel free to skip the introductory sections.

Technical requirements

In this chapter, we will be using Python 3 with the following supporting libraries:

- `deap`
- `numpy`
- `pandas`
- `matplotlib`
- `seaborn`
- `sklearn` – introduced in this chapter

In addition, we will be using the UCI *Zoo Dataset* (`https://archive.ics.uci.edu/ml/datasets/zoo`).

The programs that will be used in this chapter can be found in this book's GitHub repository at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter07`.

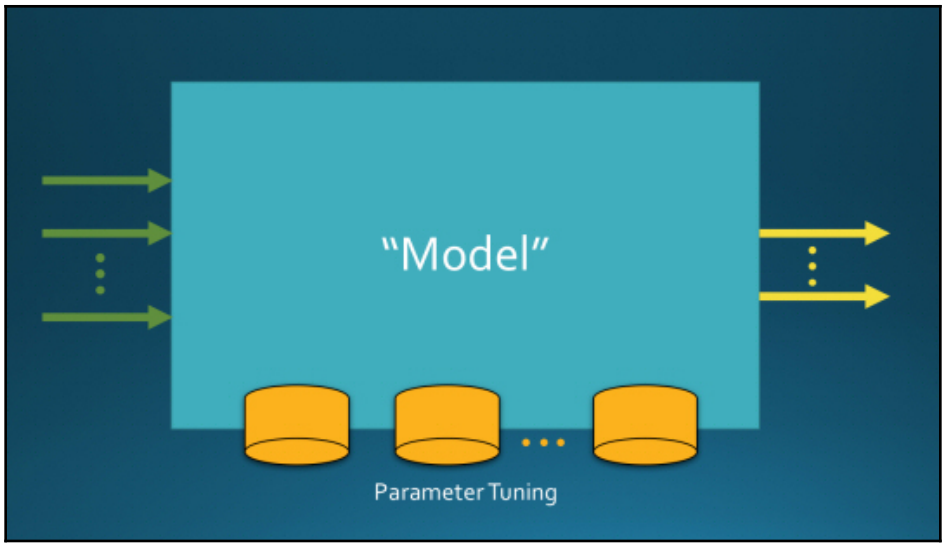
Check out the following video to see the Code in Action:
`http://bit.ly/37HCKyr`

Supervised machine learning

The term **machine learning** typically refers to a computer program that receives inputs and produces outputs. Our goal is to train this program, also known as the **model**, to produce the correct outputs for the given inputs, without explicitly programming them.

During this training process, the model learns the mapping between the inputs and the outputs by adjusting its internal parameters. One common way to train the model is by providing it with a set of inputs, for which the correct output is known. For each of these inputs, we tell the model what the correct output is so that it can adjust, or tune itself, aiming to eventually produce the desired output for each of the given inputs. This tuning is at the heart of the learning process.

Over the years, many types of machine learning models have been developed. Each model has its own particular internal parameters that can affect the mapping between the input and the output, and the values of these parameters can be tuned, as illustrated in the following image:



Parameter tuning of a machine learning model

For example, if the model was implementing a decision tree, it could contain several IF-THEN statements, which can be formulated as follows:

IF <input value> IS LESS THEN <some threshold value> THEN <go to some target branch>

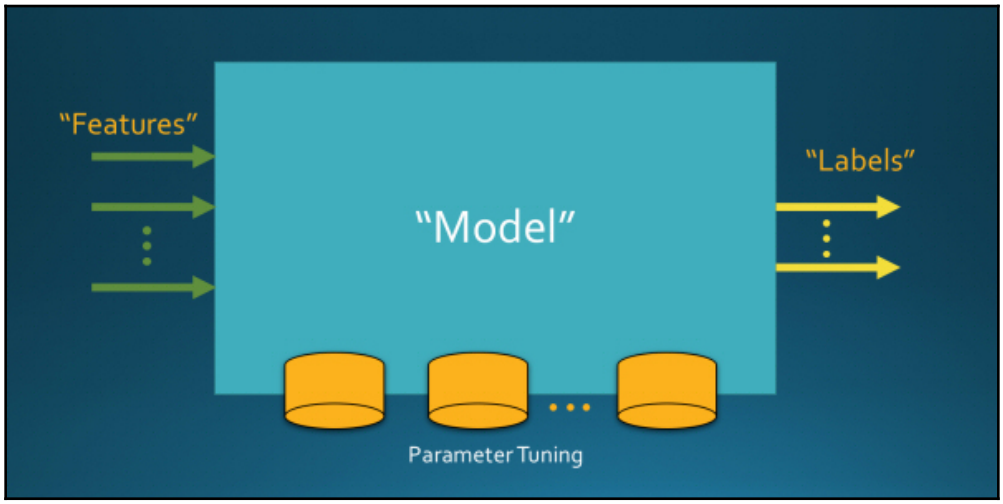
In this case, both the threshold value and the identity of the target branch are parameters that can be adjusted, or tuned, during the learning process.

To tune the internal parameters, each type of model has an accompanying **learning algorithm** that iterates over the given input and output values and seeks to match the given output for each of the given inputs. To accomplish this goal, a typical learning algorithm will measure the difference (or error) between the actual output and the desired output; the algorithm will then attempt to minimize this error by adjusting the model's internal parameters.

The two main types of supervised machine learning are **classification** and **regression**, and will be described in the following subsections.

Classification

When carrying out a classification task, the model needs to decide which category a certain input belongs to. Each category is represented by a single output (called a label), while the inputs are called **features**:

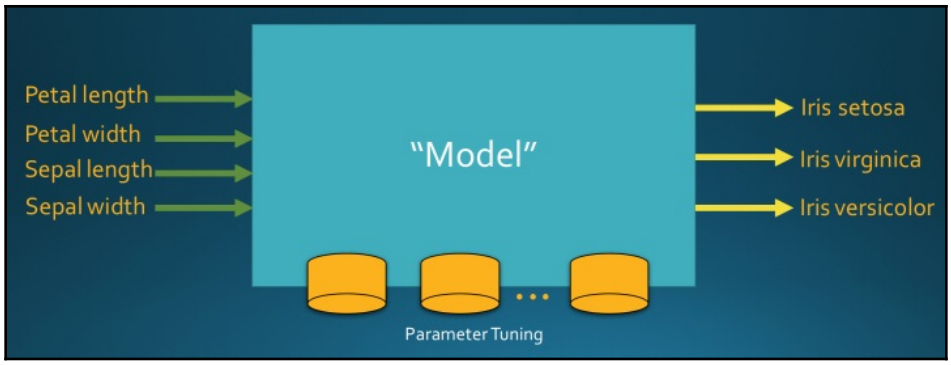


Machine learning classification model

For example, in the well-known Iris Flower dataset (`https://archive.ics.uci.edu/ml/datasets/Iris`), there are four features: **Petal length**, **Petal width**, **Sepal length**, and **Sepal width**. These represent the measurements that have been manually taken of actual Iris flowers.

In terms of the output, there are three labels: **Iris setosa**, **Iris virginica**, and **Iris versicolor**. These represent the three different types of Iris.

When the input values are present, which represent the measurements that were taken from a given Iris flower, we expect the output of the correct label to go **high** and the other two to go **low**:



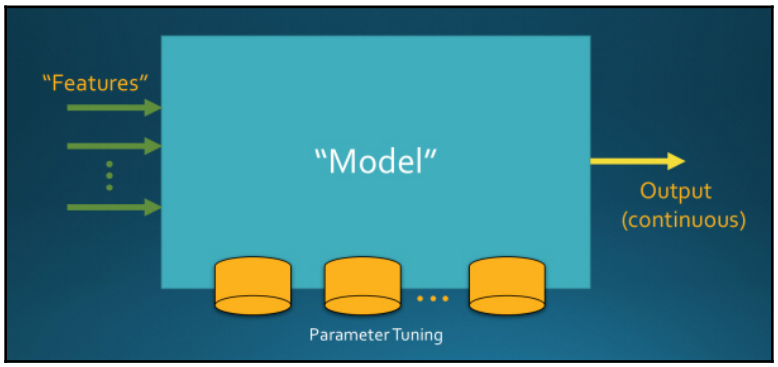
Iris Flower classifier illustrated

Classification tasks have a multitude of real-life applications, such as approval of bank loans and credit cards, email spam detection, handwritten digit recognition, and face recognition. Later in this chapter, we will be demonstrating the classification of animal types using the Zoo dataset.

The second main type of supervised machine learning, regression, will be described in the next subsection.

Regression

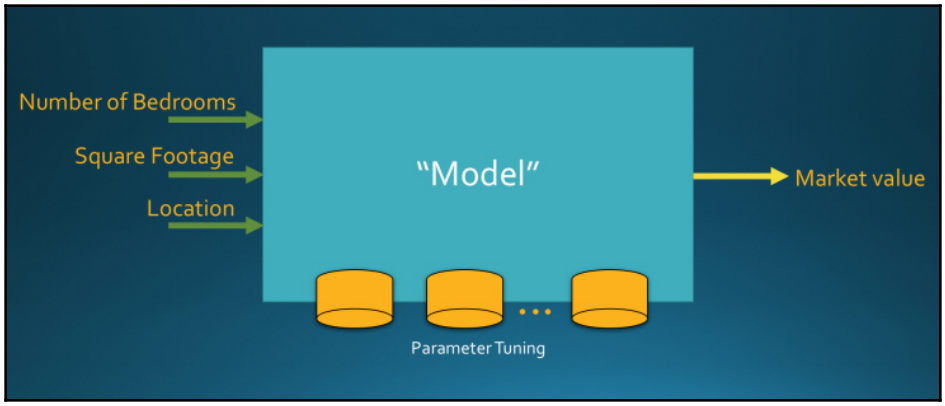
In contrast to classification tasks, the model for regression tasks maps the input values into a single output to provide a continuous value, as illustrated in the following image:



Machine learning regression model

Given the input values, the model is expected to predict the correct value of the output.

Real-life examples of regression include predicting the value of stocks, the quality of wine, or the market price of a house, as depicted in the following image:



House pricing regressor

In the preceding image, the inputs are features that provide information that describes a given house, while the output is the predicted value of the house.

Many types of models exist for carrying out classification and regression tasks – some of them are described in the following subsection.

Supervised learning algorithms

As we mentioned previously, each supervised learning model consists of a set of internal tunable parameters and an algorithm that tunes these parameters in an attempt to achieve the required result.

Some common supervised learning models/algorithms include the following:

- **Decision Trees**: A family of algorithms that utilizes a tree-like graph, where branching points represent decisions and the branches represent their consequences.
- **Random Forests**: Algorithms that create a large number of decision trees during the training phase and use a combination of their outputs.

- **Support Vector Machines**: Algorithms that map the given inputs as points in space so that the inputs that belong to separate categories are divided by the largest possible gap.
- **Artificial Neural Networks**: Models that consist of multiple simple nodes, or neurons, which can be interconnected in various ways. Each connection can have a weight that controls the level of the signal that's carried from one neuron to the next.

There are certain techniques that can be used to improve and enhance the performance of such models. One interesting technique – *feature selection* – will be discussed in the next section.

Feature selection in supervised learning

As we saw in the previous section, a supervised learning model receives a set of inputs, called **features**, and maps them to a set of outputs. The assumption is that the information described by the features is useful for determining the value of the corresponding outputs. At first glance, it may seem that the more information we can use as input, the better our chances of predicting the output(s) correctly. However, in many cases, the opposite holds true; if some of the features we use are irrelevant or redundant, the consequence could be a (sometimes significant) decrease in the accuracy of the models.

Feature selection is the process of selecting the most beneficial and essential set of features out of the entire given set of features. Besides increasing the accuracy of the model, a successful feature selection can provide the following advantages:

- The training times of the models are shorter.
- The resulting trained models are simpler and easier to interpret.
- The resulting models are likely to provide better generalization, that is, they perform better with new input data that is dissimilar to the data that was used for training.

When looking at methods to carry out feature selection, genetic algorithms are a natural candidate. We will demonstrate how they can be applied to find the best features out of an artificially generated dataset in the next section.

Selecting the features for the Friedman-1 regression problem

The Friedman-1 regression problem, which was created by Friedman and Breiman, describes a single output value, *y*, which is a function of five input values, $x_0..x_4$, and randomly generated noise, according to the following formula:

$$y(x_0, x_1, x_2, x_3, x_4) = 10 \cdot sin(\pi \cdot x_0 \cdot x_1) + 20(x_2 - 0.5)^2 + 10x_3 + 5x_4 + noise \cdot N(0, 1)$$

The input variables, $x_0..x_4$, are independent, and uniformly distributed over the interval [0, 1]. The last component in the formula is the randomly generated noise. The noise is normally distributed and multiplied by the constant *noise*, which determines its level.

In Python, the scikit-learn (`sklearn`) library provides us with the `make_friedman1()` function, which can be used to generate a dataset containing the desired number of samples. Each of the samples consists of randomly generated $x_0..x_4$ values and their corresponding calculated *y* value. The interesting part, however, is that we can tell the function to add an arbitrary number of irrelevant input variables to the five original ones by setting the `n_features` parameter to a value larger than five. If, for example, we set the value of `n_features` to `15`, we will get a dataset containing the original five input variables (or features) that were used to generate the *y* values according to the preceding formula and an additional 10 features that are completely irrelevant to the output. This can be used, for example, to test the resilience of various regression models on the noise and presence of irrelevant features in the dataset.

We can take advantage of this function to test the effectiveness of genetic algorithms as a feature selection mechanism. In our test, we will use the `make_friedman1()` function to create a dataset with 15 features and use the genetic algorithm to search for the subset of features that provides the best performance. As a result, we expect the genetic algorithm to pick the first five features and drop the rest, assuming that the model's accuracy is better when only the relevant features are used as input. The fitness function of the genetic algorithm will utilize a regression model that, for each potential solution – a subset of the feature to use – will be trained using the dataset containing only the selected features.

As usual, we will start by choosing an appropriate representation for the solution, as described in the next subsection.

Solution representation

The objective of our algorithm is to find a subset of features that yield the best performance. Therefore, a solution needs to indicate which features are chosen and which are dropped. One obvious way to go about this is to represent each individual using a list of binary values. Every entry in that list corresponds to one of the features in the dataset. A value of 1 represents selecting the corresponding feature, while a value of 0 means that the feature has not been selected. This is very similar to the approach we used in the knapsack 0-1 problem we described in `Chapter 4`, *Combinatorial Optimization*.

The presence of each 0 in the solution will be translated into dropping the corresponding feature's data column from the dataset, as we will see in the next subsection.

Python problem representation

To encapsulate the Friedman-1 feature selection problem, we've created a Python class called `Friedman1Test`. This class can be found in the `friedman.py` file, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter07/friedman.py`.

The main parts of this class are as follows:

1. The `__init__()` method of the class creates the dataset, as follows:

```
self.X, self.y = datasets.make_friedman1(n_samples=self.numSamples,
                                          n_features=self.numFeatures,
                                          noise=self.NOISE,
                                          random_state=self.randomSeed)
```

2. Then, it divides the data into two subsets –a training set and a validation set – using the **scikit-learn** `model_selection.train_test_split()` method:

```
self.X_train, self.X_validation, self.y_train,
self.y_validation = \
    model_selection.train_test_split(self.X, self.y,
test_size=self.VALIDATION_SIZE, random_state=self.randomSeed)
```

Dividing the data to a train set and a validation set allows us to train the regression model on the train set, where the correct prediction is given to the model for training purposes, and then test it with the separate validation set, where the correct predictions are not given to the model and are, instead, compared to the predictions it produces. This way, we can test how well the model is able to generalize, rather than memorize the training data.

3. Next, we create the regression model, which is of the **Gradient Boosting Regressor** (**GBR**) type. This model creates an ensemble (or aggregation) of decision trees during the training phase:

```
self.regressor =
GradientBoostingRegressor(random_state=self.randomSeed)
```

Note that we are passing the random seed along so that it can be used internally by the regressor. This way, we can make sure the results that we obtain are repeatable.

4. The `getMSE()` method of the class is used to determine the performance of our gradient boosting regression model for a set of selected features. It accepts a list of binary values corresponding to the features in the dataset – a value of 1 represents selecting the corresponding feature, while a value of 0 means that the feature is dropped. The method then deletes the columns in the training and validation sets that correspond to the unselected features:

```
zeroIndices = [i for i, n in enumerate(zeroOneList) if n == 0]
currentX_train = np.delete(self.X_train, zeroIndices, 1)
currentX_validation = np.delete(self.X_validation, zeroIndices,
1)
```

5. The modified train set – containing only the selected features – is then used to train the regressor, while the modified validation set is used to evaluate its predictions:

```
self.regressor.fit(currentX_train, self.y_train)
prediction = self.regressor.predict(currentX_validation)
return mean_squared_error(self.y_validation, prediction)
```

The metric that's used here to evaluate the regressor is called the **mean square error** (**MSE**), which finds the average squared difference between the model's predicted values and the actual values. A lower value of this measurement indicates better performance of the regressor.

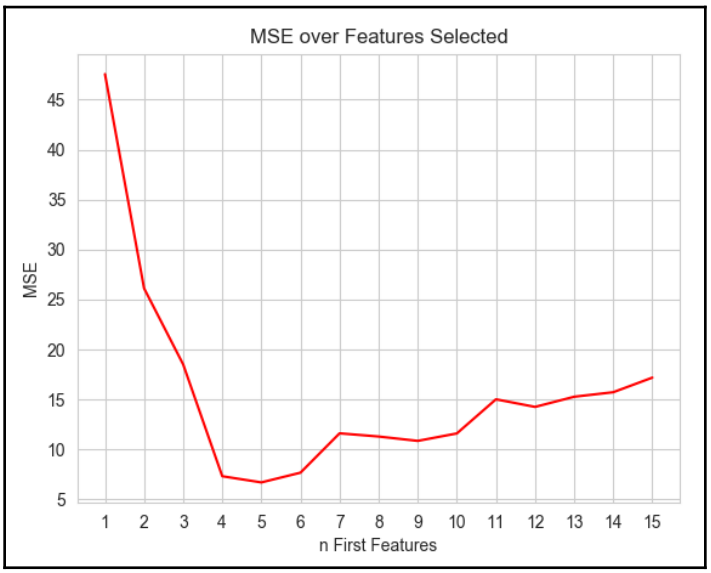
6. The `main()` method of the class creates an instance of the `Friedman1Test` class with 15 features. Then, it repeatedly uses the `getMSE()` method to evaluate the performance of the regressor with the first *n* features, while *n* is incremented from 1 to 15:

```
for n in range(1, len(test) + 1):
    nFirstFeatures = [1] * n + [0] * (len(test) - n)
    score = test.getMSE(nFirstFeatures)
```

When running the main method, the results show that, as we add the first five features one by one, the performance improves. However, afterward, each additional feature degrades the performance of the regressor:

```
1 first features: score = 47.553993
2 first features: score = 26.121143
3 first features: score = 18.509415
4 first features: score = 7.322589
5 first features: score = 6.702669
6 first features: score = 7.677197
7 first features: score = 11.614536
8 first features: score = 11.294010
9 first features: score = 10.858028
10 first features: score = 11.602919
11 first features: score = 15.017591
12 first features: score = 14.258221
13 first features: score = 15.274851
14 first features: score = 15.726690
15 first features: score = 17.187479
```

This is further illustrated by the generated plot, showing the minimum MSE value where the first five features are used:



Plot of error values for the Friedman-1 regression problem

In the next subsection, we will find out if a genetic algorithm can successfully identify these first five features.

Genetic algorithms solution

To identify the best set of features to be used for our regression test using a genetic algorithm, we've created the Python program `01-solve-friedman.py`, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter07/01-solve-friedman.py`.

As a reminder, the chromosome representation that's being used here is a list of integers with the values of 0 or 1, denoting whether a feature should be used or dropped. This makes our problem, from the point of view of the genetic algorithm, similar to the OneMax problem or the knapsack 0-1 problem we solved previously. The difference is in the fitness function returning the regression model's MSE, which is calculated within the `Friedman1Test` class.

The following steps describe the main parts of our solution:

1. First, we need to create an instance of the `Friedman1Test` class with the desired parameters:

```
friedman = friedman.Friedman1Test(NUM_OF_FEATURES,
NUM_OF_SAMPLES, RANDOM_SEED)
```

2. Since our goal is to minimize the MSE of the regression model, we define a single objective, minimizing the fitness strategy:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

3. Since the solution is represented by a list of 0 or 1 integer values, we use the following toolbox definitions to create the initial population:

```
toolbox.register("zeroOrOne", random.randint, 0, 1)
toolbox.register("individualCreator", tools.initRepeat,
creator.Individual, toolbox.zeroOrOne, len(friedman))
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

4. Then, we instruct the genetic algorithm to use the `getMSE()` method of the `Friedman1Test` instance for fitness evaluation:

```
def friedmanTestScore(individual):
    return friedman.getMSE(individual),  # return a tuple

toolbox.register("evaluate", friedmanTestScore)
```

5. As for the genetic operators, we use tournament selection with a tournament size of 2 and crossover and mutation operators that are specialized for binary list chromosomes:

```
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit,
indpb=1.0/len(friedman))
```

6. In addition, we continue to use the elitist approach, where the **hall of fame** (**HOF**) members – the current best individuals – are always passed untouched to the next generation:

```
population, logbook = elitism.eaSimpleWithElitism(population,
                                                   toolbox,
                                                   cxpb=P_CROSSOVER,
                                                   mutpb=P_MUTATION,
                                                   ngen=MAX_GENERATIONS,
                                                   stats=stats,
                                                   halloffame=hof,
                                                   verbose=True)
```

By running the algorithm for 30 generations with a population size of 30, we get the following outcome:

```
-- Best Ever Individual = [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
-- Best Ever Fitness = 6.702668910463287
```

This indicates that the first five features have been selected to provide the best MSE (about 6.7) for our test. Note that the genetic algorithm makes no assumptions about the set of features that it was looking for, meaning it did not know that we are looking for a subset of the *first n* features. It simply searched for the best possible subset of features.

In the next section, we will advance from using artificially generated data to an actual dataset and utilize the genetic algorithm to select the best features for a classification problem.

Selecting the features for the classification Zoo dataset

The UCI *Machine Learning Repository* (`https://archive.ics.uci.edu/ml/index.php`) maintains over 350 datasets as a service to the machine learning community. These datasets can be used for experimentation with various models and algorithms. A typical dataset contains a number of features (inputs) and the desired output, in a form of columns, with a description of their meaning.

In this section, we will use the UCI Zoo dataset (`https://archive.ics.uci.edu/ml/datasets/zoo`). This dataset describes 101 different animals using the following 18 features:

No.	Feature Name	Data Type
1	animal name	Unique for each instance
2	hair	Boolean
3	feathers	Boolean
4	eggs	Boolean
5	milk	Boolean
6	airborne	Boolean
7	aquatic	Boolean
8	predator	Boolean
9	toothed	Boolean
10	backbone	Boolean
11	breathes	Boolean
12	venomous	Boolean
13	fins	Boolean
14	legs	Numeric (set of values {0,2,4,5,6,8})
15	tail	Boolean
16	domestic	Boolean
17	catsize	Boolean
18	type	Numeric (integer values in range [1,7])

Most features are Boolean (value of 1 or 0), indicating the presence or absence of a certain attribute, such as hair, fins, and so on. The first feature, **animal name**, is just to provide us with some information and does not participate in the learning process.

This dataset is used for testing classification tasks, where the input features need to be mapped into two or more categories/labels. In this dataset, the last feature – called **type** – represents the category, and is used as the output value. In this dataset, there are seven categories altogether. A type value of 5, for instance, represents an animal category that includes frog, newt, and toad. To sum this up, a classification model trained with this dataset will use features 2-17 (hair, feathers, fins, and so on) to predict the value of feature 18 (animal type).

Once again, we want to use a genetic algorithm to select the features that will give us the best predictions. Let's start by creating a Python class that represents a classifier that's been trained with this dataset.

Python problem representation

To encapsulate the feature selection process for the Zoo dataset classification task, we've created a Python class called `Zoo`. This class is contained in the `zoo.py` file, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter07/zoo.py`.

The main parts of this class are highlighted as follows:

1. The `__init__()` method of the class loads the Zoo dataset from the web while skipping the first feature – animal name – as follows:

```
self.data = read_csv(self.DATASET_URL, header=None,
usecols=range(1, 18))
```

2. Then, it separates the data to input features (first remaining 16 columns) and the resulting category (last column):

```
self.X = self.data.iloc[:, 0:16]
self.y = self.data.iloc[:, 16]
```

3. Instead of just separating the data into a training set and a test set, like we did in the previous section, we're using *k-fold cross-validation*. This means that the data is split into *k* equal parts and the model is evaluated *k* times, each time using *(k-1)* parts for training and the remaining part for testing (or validation). This is easy to do in Python using the scikit-learn library's `model_selection.KFold()` method:

```
self.kfold = model_selection.KFold(n_splits=self.NUM_FOLDS,
random_state=self.randomSeed)
```

4. Next, we create a classification model based on a decision tree. This type of classifier creates a tree structure during the training phase that splits the dataset into smaller subsets, eventually resulting in a prediction:

```
self.classifier =
DecisionTreeClassifier(random_state=self.randomSeed)
```

Note that we are passing the random seed along so that it can be used internally by the classifier. This way, we can make sure the results that are obtained are repeatable.

5. The `getMeanAccuracy()` method of the class is used to evaluate the performance of the classifier for a set of selected features. Similar to the `getMSE()` method in the `Friedman1Test` class, this method accepts a list of binary values corresponding to the features in the dataset – a value of 1 represents selecting the corresponding feature, while a value of 0 means that the feature is dropped. The method then drops the columns in the dataset that correspond to the unselected features:

```
zeroIndices = [i for i, n in enumerate(zeroOneList) if n == 0]
currentX = self.X.drop(self.X.columns[zeroIndices], axis=1)
```

6. This modified dataset – containing only the selected features – is then used to perform the k-fold cross-validation process and determine the classifier's performance over the data partitions. The value of *k* in our class is set to `5`, so five evaluations take place each time:

```
cv_results = model_selection.cross_val_score(self.classifier,
currentX, self.y, cv=self.kfold, scoring='accuracy')
return cv_results.mean()
```

   The metric that's being used here to evaluate the classifier is **accuracy** – the portion of the cases that were classified correctly. An accuracy of 0.85, for example, means that 85% of the cases were classified correctly. Since, in our case, we train and evaluate the classifier k times, we use the average (mean) accuracy value that was obtained over these evaluations.

7. The `main()` method of the class creates an instance of the `Zoo` class and evaluates the classifier with all 16 features that are present using the **all-one** solution representation:

```
allOnes = [1] * len(zoo)
print("-- All features selected: ", allOnes, ", accuracy = ",
zoo.getMeanAccuracy(allOnes))
```

When running the main method of the class, the printout shows that, when testing our classifier with 5-fold cross-validation using all 16 features, the classification accuracy that's achieved is about 91%:

```
-- All features selected:  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
, accuracy =  0.9099999999999999
```

In the next subsection, we will attempt to improve the accuracy of the classifier by selecting a subset of features from the dataset, instead of using all the features. We will use – you guessed it – a genetic algorithm to select these features for us.

Genetic algorithms solution

To identify the best set of features to be used for our Zoo classification task using a genetic algorithm, we've created the Python program `02-solve-zoo.py`, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter07/02-solve-zoo.py`.

As in the previous section, the chromosome representation that's being used here is a list of integers with the values of 0 or 1, denoting whether a feature should be used or dropped.

The following steps highlight the main parts of the program:

1. First, we need to create an instance of the `Zoo` class and pass our random seed along for the sake of producing repeatable results:

```
zoo = zoo.Zoo(RANDOM_SEED)
```

2. Since our goal is to maximize the accuracy of the classifier model, we define a single objective, maximizing fitness strategy:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

3. Just like in the previous section, we use the following toolbox definitions to create the initial population of individuals, each constructed as a list of `0` or `1` integer values:

```
toolbox.register("zeroOrOne", random.randint, 0, 1)
toolbox.register("individualCreator", tools.initRepeat,
creator.Individual, toolbox.zeroOrOne, len(zoo))
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

4. Then, we instruct the genetic algorithm to use the `getMeanAccuracy()` method of the `Zoo` instance for fitness evaluation. To do this, we had to make two modifications:
    - We eliminated the possibility of no features being selected (all-zeros individual) since our classifier will throw an exception in such a case.
    - We added a small penalty for each feature being used to encourage the selection of fewer features. The penalty value is very small (0.001), so it only comes into play as a tie-breaker between two equally performing classifiers, leading the algorithm to prefer the one that uses fewer features:

```
def zooClassificationAccuracy(individual):
    numFeaturesUsed = sum(individual)
    if numFeaturesUsed == 0:
        return 0.0,
    else:
        accuracy = zoo.getMeanAccuracy(individual)
        return accuracy - FEATURE_PENALTY_FACTOR *
numFeaturesUsed,  # return a tuple


toolbox.register("evaluate",
zooClassificationAccuracy)
```

5. For the genetic operators, we again use tournament selection with a tournament size of `2` and crossover and mutation operators that are specialized for binary list chromosomes:

```
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit,
indpb=1.0/len(zoo))
```

6. And once again, we continue to use the elitist approach, where HOF members – the current best individuals – are always passed untouched to the next generation:

```
population, logbook = elitism.eaSimpleWithElitism(population,
toolbox, cxpb=P_CROSSOVER,                mutpb=P_MUTATION,
ngen=MAX_GENERATIONS, stats=stats, halloffame=hof,
verbose=True)
```

7. At the end of the run, we print out all the members of the HOF so that we can see the top results that were found by the algorithm. We print both the fitness value, which includes the penalty for the number of features, and the actual accuracy value:

```
print("- Best solutions are:")
for i in range(HALL_OF_FAME_SIZE):
    print(i, ": ", hof.items[i], ", fitness = ",
hof.items[i].fitness.values[0],
          ", accuracy = ", zoo.getMeanAccuracy(hof.items[i]),
", features = ", sum(hof.items[i]))
```

By running the algorithm for 50 generations with a population size of 50 and HOF size of 5, we get the following outcome:

```
- Best solutions are:
0 : [0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0] , fitness = 0.964 ,
accuracy = 0.97 , features = 6
1 : [0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1] , fitness = 0.963 ,
accuracy = 0.97 , features = 7
2 : [0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0] , fitness = 0.963 ,
accuracy = 0.97 , features = 7
3 : [1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0] , fitness = 0.963 ,
accuracy = 0.97 , features = 7
4 : [0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0] , fitness = 0.963 ,
accuracy = 0.97 , features = 7
```

These results indicate that all five top solutions achieved an accuracy value of 97%, using either six or seven features out of the available 16. Thanks to the penalty factor on a number of features, the top solution is the set of six features, which are as follows:

- feathers
- milk
- airborne
- backbone
- fins
- tail

In conclusion, by selecting these particular features out of the 16 given in the dataset, not only did we reduce the dimensionality of the problem, but we were also able to improve our model accuracy from 91% to 97%. If this does not seem like a large enhancement at first glance, think of it as reducing the error rate from 9% to 3% – a very significant improvement in terms of classification performance.

Summary

In this chapter, you were introduced to machine learning and the two main types of supervised machine learning tasks – regression and classification. Then, you were presented with the potential benefits of feature selection on the performance of the models carrying out these tasks. At the heart of this chapter were two demonstrations of how genetic algorithms can be utilized to enhance the performance of such models via feature selection. In the first case, we pinpointed the genuine features that were generated by the Friedman-1 Test regression problem, while, in the other case, we selected the most beneficial features of the Zoo classification dataset.

In the next chapter, we will look at another possible way of enhancing the performance of supervised machine learning models, namely hyperparameter tuning.

Further reading

For more information about the topics that were covered in this chapter, please refer to the following resources:

- *Applied Supervised Learning with Python*, *Benjamin Johnston* and *Ishita Mathur*, April 26, 2019
- *Feature Engineering Made Easy*, *Sinan Ozdemir* and *Divya Susarla*, January 22, 2018
- **Feature selection for classification, M.Dash and H.Liu, 1997**: `https://doi.org/10.1016/S1088-467X(97)00008-5`
- **UCI Machine Learning Repository**: `https://archive.ics.uci.edu/ml/index.php`

8
Hyperparameter Tuning of Machine Learning Models

This chapter describes how genetic algorithms can be used to improve the performance of supervised machine learning models by tuning the hyperparameters of the models. The chapter will start with a brief introduction to hyperparameter tuning in machine learning before describing the concept of a grid search. After introducing the Wine dataset and the adaptive boosting classifier, both of which will be used throughout this chapter, we will demonstrate hyperparameter tuning using both a conventional grid search and a genetic algorithm-driven grid search. Finally, we will attempt to enhance the results we get by using a direct genetic algorithm approach for hyperparameter tuning.

By the end of this chapter, you will:

- Understand the concept of hyperparameter tuning in machine learning
- Be familiar with the Wine dataset and the adaptive boosting classifier
- Enhance the performance of a classifier using a hyperparameter grid search
- Enhance the performance of a classifier using a genetic algorithm-driven hyperparameter grid search
- Enhance the performance of a classifier using a direct genetic algorithm approach for hyperparameter tuning

We will start this chapter with a quick overview of hyperparameters in machine learning. If you are a seasoned data scientist, feel free to skip the introductory section.

Technical requirements

In this chapter, we will be using Python 3 with the following supporting libraries:

- `deap`
- `numpy`
- `pandas`
- `matplotlib`
- `seaborn`
- `sklearn`
- `sklearn-deap` – introduced in this chapter

In addition, we will be using the UCI *Wine Dataset* (`https://archive.ics.uci.edu/ml/datasets/Wine`).

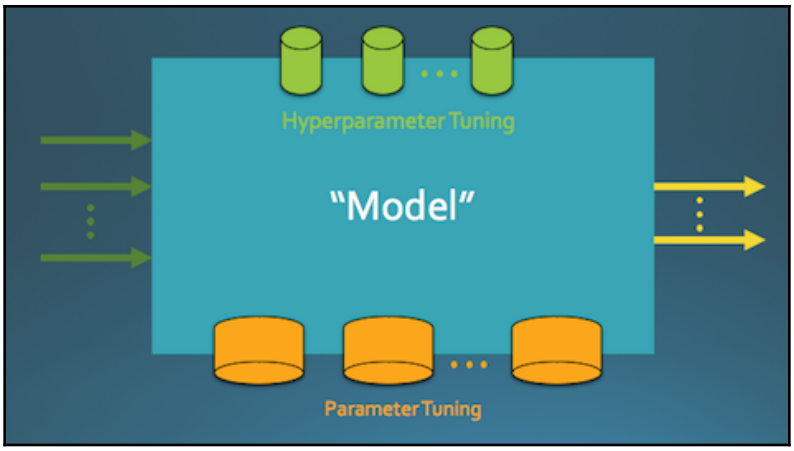
The programs that will be used in this chapter can be found in this book's GitHub repository at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter08`.

Check out the following video to see the Code in Action:
`http://bit.ly/37Q45id`

Hyperparameters in machine learning

In `Chapter 7`, *Enhancing Machine Learning Models using Feature Selection*, we described supervised learning as the programmatic process of adjusting (or tuning) the internal parameters of a model to produce the desired outputs in response to given inputs. To make this happen, each type of supervised learning model is accompanied by a learning algorithm that iteratively adjusts its internal parameters during the learning (or training) phase.

However, most models have another set of parameters that are set **before** the learning takes place. These are called **hyperparameters**, and affect the way the learning is done. The following image illustrates the two types of parameters:



Hyperparameter tuning of a machine learning model

Usually, the hyperparameters have default values that will take effect if we don't specifically set them. For example, if we look at the `sklearn` library implementation of the Decision Tree classifier (`https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html`), we will see several hyperparameters and their default values.

A few of these hyperparameters are described in the following table:

Name	**Type**	**Description**	**Default Value**
`max_depth`	int	The maximum depth of the tree	None
`splitter`	string	The strategy that's used to choose the split at each node (best or random)	best
`min_samples_split`	int or float	The minimum number of samples required to split an internal node	2

Each of these parameters affects the way the decision tree is constructed during the learning process, and their combined effect on the results of the learning process – and, consequently, on the performance of the model – can be significant.

Since the choice of hyperparameters has a considerable impact on the performance of machine learning models, data scientists often spend significant amounts of time looking for the best hyperparameter combinations, a process called **hyperparameter tuning**. Some of the methods that are used for hyperparameter tuning will be described in the next subsection.

Hyperparameter tuning

A common way to search for good combinations of hyperparameters is by using a **grid search.** Using this method, we choose a subset of values for each hyperparameter that we want to tune. As an example, given the Decision Tree classifier, we can choose the subset of values `{2, 5, 10}` for the `max_depth` parameter, while, for the `splitter` parameter, we choose both possible values – `{"best", "random"}`. Then, we try out all six possible combinations of these values. For each combination, the classifier is trained and evaluated for a certain performance criterion, for example, accuracy. At the end of the process, we pick the combination of hyperparameter values that yielded the best performance.

The main drawback of the grid search is the exhaustive search it conducts over all the possible combinations, which can prove very lengthy. One common way to produce good combinations in a shorter amount of time is **random search**, where random combinations of hyperparameters are chosen and tested.

A better option – of particular interest to us – when it comes to performing the grid search, is to harness a genetic algorithm to look for the best combination(s) of hyperparameters within the predefined grid. This method offers the potential for finding the best grid combinations in a shorter amount of time than the original, exhaustive grid search.

While grid search and random search are supported by the `sklearn` library, the genetic algorithm-driven grid search option is offered by the `sklearn-deap` library and builds upon the DEAP-based genetic algorithm's capabilities. This `library` can be installed as follows:

```
pip install sklearn-deap
```

In the following sections, we will try out and compare both versions – exhaustive and genetic algorithm-driven – of the grid search. But first, we'll take a quick look at the dataset we are going to use for our experiment – the UCI Wine dataset.

The Wine dataset

A commonly used dataset from the UCI Machine Learning Repository (`https://archive.ics.uci.edu/ml/index.php`), the Wine dataset (`https://archive.ics.uci.edu/ml/datasets/Wine`), contains the results of a chemical analysis that was conducted for 178 different wines that were grown in the same region in Italy and categorizes these wines into one of three types.

The chemical analysis consists of 13 different measurements, representing the quantities of the following constituents that are found in each wine:

- Alcohol
- Malic acid
- Ash
- Alkalinity of ash
- Magnesium
- Total phenols
- Flavanoids
- Nonflavanoid phenols
- Proanthocyanins
- Color intensity
- Hue
- OD280/OD315 of diluted wines
- Proline

Columns 2-14 of the dataset contain the values for the preceding measurements, while the classification outcome – the wine type itself (1, 2, or 3) – is found in the first column.

Next, let's look at the classifier we chose to classify this dataset.

The adaptive boosting classifier

The **adaptive boosting algorithm**, or **AdaBoost**, for short, is a powerful machine learning model that combines the outputs of multiple instances of a simple learning algorithm (weak learner) using a weighted sum. AdaBoost adds instances of the weak learner during the learning process, each of which is adjusted to improve previously misclassified inputs.

The `sklearn` library's implementation of this model, AdaboostClassifier (`https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html`), uses several hyperparameters, some of which are as follows:

Name	Type	Description	Default Value
`n_estimators`	int	The maximum number of estimators	50
`learning_rate`	float	Can be used to shrink the contribution of each classifier	1
`algorithm`	`{'SAMME', 'SAMME.R'}`	`'SAMME.R'` – uses a real boosting algorithm, `'SAMME'` – uses a discrete boosting algorithm	2

Interestingly, each of these three hyperparameters is of a different type – an int, a float, and an enumerated (or categorical) type. Later, we will find out how each tuning method handles these different types. We will start with two forms of the grid search, both of which will be described in the next section.

Tuning the hyperparameters using a genetic grid search

To encapsulate the hyperparameter tuning of the AdaBoost classifier for the wine dataset using a grid search – both the conventional version and the genetic algorithm-driven version – we created a Python class called `HyperparameterTuningGrid`. This class can be found in the `01-hyperparameter-tuning-grid.py` file, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter08/01-hyperparameter-tuning-grid.py`.

The main parts of this class are highlighted as follows:

1. The `__init__()` method of the class initializes the wine dataset, the AdaBoost classifier, the k-fold cross-validation metric, and the grid parameters:

```
self.initWineDataset()
self.initClassifier()
self.initKfold()
self.initGridParams()
```

2. The `initGridParams()` method initializes the grid search by setting the tested values of the three hyperparameters mentioned in the previous section:
    - The `n_estimators` parameter is tested across 10 values, linearly spaced between 10 and 100.
    - The `learning_rate` parameter is tested across 10 values, logarithmically spaced between 0.1 (10^{-2}) and 1 (10^{0}).
    - Both possible values of the `algorithm` parameter, `'SAMME'` and `'SAMME.R'`, are tested.

   This setup covers a total of 200 (10×10×2) different combinations of the grid parameters:

```
self.gridParams = {
    'n_estimators': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100],
    'learning_rate': np.logspace(-2, 0, num=10, base=10),
    'algorithm': ['SAMME', 'SAMME.R'],
}
```

3. The `getDefaultAccuracy()` method evaluates the accuracy of the classifier with its default hyperparameter values using the mean value of the `'accuracy'` metric:

```
cv_results = model_selection.cross_val_score(self.classifier,
                                             self.X,
                                             self.y,
                                             cv=self.kfold,
scoring='accuracy')
return cv_results.mean()
```

4. The `gridTest()` method performs a conventional grid search over the set of tested hyperparameter values we defined earlier. The best combination of parameters is determined, based on their k-fold cross-validation mean `'accuracy'` metric:

```
gridSearch = GridSearchCV(estimator=self.classifier,
                          param_grid=self.gridParams,
                          cv=self.kfold,
                          scoring='accuracy')

gridSearch.fit(self.X, self.y)
```

5. The `geneticGridTest()` method performs a genetic algorithm-driven grid search. It utilizes the `sklearn-deap` library's `EvolutionaryAlgorithmSearchCV()` method, which was designed to be called in a very similar manner to that of the conventional grid search. All we need to do is add a few genetic algorithm parameters – population size, mutation probability, tournament size, and the number of generations:

```
gridSearch =
EvolutionaryAlgorithmSearchCV(estimator=self.classifier,
                                                params=self.gridParams,
                                                cv=self.kfold,
                                                scoring='accuracy',
                                                verbose=True,
                                                population_size=20,
                                                gene_mutation_prob=0.30,
                                                tournament_size=2,
                                                generations_number=5)
gridSearch.fit(self.X, self.y)
```

6. Finally, the `main()` method of the class starts by evaluating the performance of the classifier with its default hyperparameter values. Then, it runs the conventional, exhaustive grid search, followed by the genetic algorithm-driven grid search, while timing each search.

The results of running the main method of this class are described in the next subsection.

Testing the classifier's default performance

The results of the run indicate that, with the default parameter values of `n_estimators` = `50`, `learning_rate` = `1.0`, and `algorithm` = `'SAMME.R'`, the classification accuracy of the classifier is about 65%:

```
Default Classifier Hyperparameter values:
{'algorithm': 'SAMME.R', 'base_estimator': None, 'learning_rate': 1.0,
'n_estimators': 50, 'random_state': 42}
score with default values = 0.6457142857142857
```

This is not a particularly good accuracy. Hopefully, grid search can improve this by finding a better combination of hyperparameter values.

Running the conventional grid search

The conventional, exhaustive grid search, covering all 200 possible combinations, needs to be run next. The search results indicate that the best combination was `n_estimators` = `70`, `learning_rate` ≈ `0.359`, and `algorithm` = `'SAMME.R'`.

The classification accuracy that we achieved with these values is about 93%, which is a vast improvement over the original 65%. The search runtime was about 32 seconds:

```
performing grid search...
best parameters: {'algorithm': 'SAMME.R', 'learning_rate':
0.3593813663804626, 'n_estimators': 70}
best score: 0.9325842696629213
Time Elapsed = 32.180874824523926
```

Next comes the genetic-powered grid search. Will it match these results? Let's find out.

Running the genetic algorithm-driven grid search

The last portion of the run describes the genetic algorithm-driven grid search, which is carried out with the same grid parameters. The verbose output of the search starts with a somewhat cryptic printout:

```
performing Genetic grid search...
Types [1, 2, 1] and maxint [9, 9, 1] detected
```

This printout is referring to the grid we are searching on – a list of 10 integers (`n_estimators` values), an `ndarray` of 10 elements (`learning_rate` values), and a list of two strings (`algorithm` values), as follows:

- `'Types [1, 2, 1]'` refers to the grid types of [list, ndarray, list].
- `'maxint [9, 9, 1]'` corresponds to the list/array sizes of [10, 10, 2].

The next printed line refers to the total amount of possible grid combinations (10×10×2):

```
--- Evolve in 200 possible combinations ---
```

The rest of the printout looks very familiar since it utilizes the same DEAP-based genetic algorithm tools that we have been using all along, detailing the process of evolving the generations and printing a statistics line for each generation:

```
gen nevals avg min max std
0 20 0.642135 0.117978 0.904494 0.304928
1 14 0.807865 0.123596 0.91573 0.20498
```

```
2 15 0.829775 0.123596 0.921348 0.172647
3 12 0.885393 0.679775 0.921348 0.0506055
4 13 0.903652 0.865169 0.926966 0.0176117
5 11 0.905618 0.797753 0.932584 0.027728
```

At the end of the process, the best combination is printed, along with the score value and the time that elapsed:

```
Best individual is: {'n_estimators': 70, 'learning_rate':
0.3593813663804626, 'algorithm': 'SAMME.R'}
with fitness: 0.9325842696629213
Time Elapsed = 10.997037649154663
```

These results indicate that the genetic algorithm-driven grid search was able to find the same best result that was found using the exhaustive search but in a shorter amount of time – about 11 seconds.

Please note that this is a simple example that runs very quickly. In real-life situations, we often encounter large datasets, as well as complex models and extensive hyperparameter grids. In these circumstances, running an exhaustive grid search can be prohibitively lengthy, while the genetic algorithm-driven grid search has the potential to yield good results within a reasonable amount of time.

But still, all grid searches, genetic-driven or not, are limited to the subset of hyperparameter values that are defined by the grid. What if we would like to search outside the grid, without being limited to a subset of predefined values? A possible solution is described in the next section.

Tuning the hyperparameters using a direct genetic approach

Besides offering an efficient grid search option, genetic algorithms can be utilized to directly search the entire parameter space, just as we used them to search the input space for many types of problems throughout this book. Each hyperparameter can be represented as a variable participating in the search, and the chromosome can be a combination of all these variables.

Since the hyperparameters can be of varying types, for example, float, int, and enumerated, which we have in our AdaBoost classifier, we may want to code each of them differently, and then define the genetic operations as a combination of separate operators that are adapted to each of the types. However, we can also use a lazy approach and code all of them as float parameters to simplify the implementation of the algorithm, as we will see next.

Hyperparameter representation

In `Chapter 6`, *Optimizing Continuous Functions*, we used genetic algorithms to optimize the functions of real-valued parameters. These parameters were represented as a list of float numbers, like so:

```
[1.23, 7.2134, -25.309]
```

Consequently, the genetic operators we used were specialized for handling lists of floating-point numbers.

To adapt this approach so that it can tune the hyperparameters, we are going to represent each hyperparameter as a floating-point number, regardless of its actual type. To make this work, we need to find a way to transform each parameter into a floating-point number, and back from a floating-point number to its original representation. We will implement these transformations as follows:

- `n_estimators`, originally an integer, will be represented by a float value in a certain range; for example, [1, 100]. To transform the float value back into an integer, we will use the Python `round()` function, which will round it to the nearest integer.
- `learning_rate` is already a float, so no conversion is needed. It will be bound to the range of [0.01, 1.0].
- *algorithm* can have one of two values, `'SAMME'` or `'SAMME.R'`, and will be represented by a float number in the range of [0, 1]. To transform the float value, we will round it to the nearest integer – `0` or `1`. Then, we will replace a value of `0` with `'SAMME'` and a value of `1` with `'SAMME.R'`.

These conversions will be carried out by two Python files, both of which will be described in the following subsections.

Evaluating the classifier accuracy

We start with a Python class encapsulating the classifier's accuracy evaluation, called `HyperparameterTuningGenetic`. This class can be found in the `hyperparameter_tuning_genetic_test.py` file, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter08/hyperparameter_tuning_genetic_test.py`.

The main functionality of this class is highlighted as follows:

1. The `convertParam()` method of the class takes a list called `params`, containing the float values representing the hyperparameters, and transforms them into their actual values, as discussed in the previous subsection:

```
n_estimators = round(params[0])
learning_rate = params[1]
algorithm = ['SAMME', 'SAMME.R'][round(params[2])]
```

2. The `getAccuracy()` method takes a list of float numbers representing the hyperparameter values, uses the `convertParam()` method to transform them into actual values, and initializes the ADABooost classifier with these values:

```
n_estimators, learning_rate, algorithm =
self.convertParams(params)
self.classifier = AdaBoostClassifier(n_estimators=n_estimators,
                                      learning_rate=learning_rate,
                                      algorithm=algorithm)
```

3. Then, it finds the accuracy of the classifier using the k-fold cross-validation that we created for the wine dataset:

```
cv_results = model_selection.cross_val_score(self.classifier,
                                             self.X,
                                             self.y,
                                             cv=self.kfold,
                                             scoring='accuracy')
return cv_results.mean()
```

This class is utilized by the program that implements the hyperparameter-tuning genetic algorithm. This will be described in the next section.

Tuning the hyperparameters using genetic algorithms

The genetic algorithm-based search for the best hyperparameter values is implemented by the Python program, `02-hyperparameter-tuning-genetic.py`, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter08/02-hyperparameter-tuning-genetic.py`.

The following steps describe the main parts of this program:

1. We start by setting the lower and upper boundary for each of the float values representing a hyperparameter, as described in the previous subsection – [1, 100] for `n_estimators`, [0.01, 1] for `learning_rate`, and [0, 1] for `algorithm`:

```
# [n_estimators, learning_rate, algorithm]:
BOUNDS_LOW =  [  1, 0.01, 0]
BOUNDS_HIGH = [100, 1.00, 1]
```

2. Then, we create an instance of the `HyperparameterTuningGenetic` class that will allow us to test the various combinations of the hyperparameters:

```
test =
hyperparameter_tuning_genetic.HyperparameterTuningGenetic(RANDO
M_SEED)
```

3. Since our goal is to maximize the accuracy of the classifier, we define a single objective, maximizing fitness strategy:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

4. Now comes a particularly interesting part: since the solution is represented by a list of float values, each of a different range, we use the following loop to iterate over all pairs of lower-bound, upper-bound values. For each hyperparameter, we create a separate toolbox operator, which will be used to generate random float values in the appropriate range:

```
for i in range(NUM_OF_PARAMS):
    # "hyperparameter_0", "hyperparameter_1", ...
    toolbox.register("hyperparameter_" + str(i),
                     random.uniform,
                     BOUNDS_LOW[i],
                     BOUNDS_HIGH[i])
```

5. Then, we create the `hyperparameter` tuple, which contains the separate float number generators we just created for each hyperparameter:

```
hyperparameters = ()
for i in range(NUM_OF_PARAMS):
    hyperparameters = hyperparameters + \
(toolbox.__getattribute__("hyperparameter_" + str(i)),)
```

6. Now, we can use this `hyperparameter` tuple, in conjunction with DEAP's built-in `initCycle()` operator, to create a new `individualCreator` operator that fills up an individual instance with a combination of randomly generated hyperparameter values:

```
toolbox.register("individualCreator",
                 tools.initCycle,
                 creator.Individual,
                 hyperparameters,
                 n=1)
```

7. Then, we instruct the genetic algorithm to use the `getAccuracy()` method of the `HyperparameterTuningGenetic` instance for fitness evaluation. As a reminder, the `getAccuracy()` method, which we described in the previous subsection, converts the given individual – a list of three floats – back into the classifier hyperparameter values they represent, trains the classifier with these values, and evaluates its accuracy using k-fold cross-validation:

```
def classificationAccuracy(individual):
    return test.getAccuracy(individual),

toolbox.register("evaluate", classificationAccuracy)
```

8. Now, we need to define the genetic operators. While, for the selection operator, we use the usual tournament selection with a tournament size of 2, we choose crossover and mutation operators that are specialized for bounded float-list chromosomes and provide them with the boundaries we defined for each hyperparameter:

```
toolbox.register("select", tools.selTournament, tournsize=2)

toolbox.register("mate",
                 tools.cxSimulatedBinaryBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR)

toolbox.register("mutate",
```

```
                                tools.mutPolynomialBounded,
                                low=BOUNDS_LOW,
                                up=BOUNDS_HIGH,
                                eta=CROWDING_FACTOR,
                                indpb=1.0 / NUM_OF_PARAMS)
```

9. In addition, we continue to use the elitist approach, where the HOF members – the current best individuals – are always passed untouched to the next generation:

```
population, logbook = elitism.eaSimpleWithElitism(population,
                                                   toolbox,
                                                   cxpb=P_CROSSOVER,
                                                   mutpb=P_MUTATION,
                                                   ngen=MAX_GENERATIONS,
                                                   stats=stats,
                                                   halloffame=hof,
                                                   verbose=True)
```

By running the algorithm for five generations with a population size of 20, we get the following outcome:

```
gen nevals max avg
0 20 0.92127 0.841024
1 14 0.943651 0.900603
2 13 0.943651 0.912841
3 14 0.943651 0.922476
4 15 0.949206 0.929754
5 13 0.949206 0.938563
- Best solution is:
params = 'n_estimators'= 69, 'learning_rate'=0.628, 'algorithm'=SAMME.R
Accuracy = 0.94921
```

These results indicate that the best combination that was found was `n_estimators` = `69`, `learning_rate` = `0.628`, and `algorithm` = `'SAMME.R'`.

The classification accuracy that we achieved with these values is about 94.9% – a worthy improvement over the accuracy we achieved with the grid search. Interestingly, the best values that were found for `n_estimators` and `learning_rate` are just outside the grid values we searched on.

Summary

In this chapter, you were introduced to the concept of hyperparameter tuning in machine learning. After getting acquainted with the Wine dataset and the Adaptive Boosting classifier, both of which we used for testing throughout this chapter, you were presented with the hyperparameter tuning methods of an exhaustive grid search and its genetic algorithm-driven counterpart. These two methods were then compared using our test scenario. Finally, we tried out a direct genetic algorithm approach, where all the hyperparameters were represented as float values. This approach allowed us to improve on the results of the grid search.

In the next chapter, we will look into the fascinating machine learning models of neural networks and deep learning and apply genetic algorithms to improve their performance.

Further reading

For more information on the topics that were covered in this chapter, please refer to the following resources:

- Introduction to hyperparameter tuning, from the book *Mastering Predictive Analytics with scikit-learn and TensorFlow*, *Alan Fontaine*, September 2018: `https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781789617740/2/ch02lvl1sec16/introduction-to-hyperparameter-tuning`
- **sklearn-deap at GitHub**: `https://github.com/rsteca/sklearn-deap`
- **Comparing EvolutionaryAlgorithmSearchCV against GridSearchCV and RandomizedSearchCV**: `https://github.com/rsteca/sklearn-deap/blob/master/test.ipynb`
- **sklearn ADABoost Classifier**: `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html`
- **UCI Machine Learning Repository**: `https://archive.ics.uci.edu/ml/index.php`

9
Architecture Optimization of Deep Learning Networks

This chapter describes how genetic algorithms can be used to improve the performance of artificial neural network-based models by optimizing the network architecture of these models. We will start with a brief introduction to neural networks and deep learning. After introducing the Iris dataset and the Multilayer Perceptron classifier, we will demonstrate network architecture optimization using a genetic algorithm-based solution. Then, we will extend this approach to combine network architecture optimization with model hyperparameter tuning, which will be jointly carried out by a genetic algorithm-based solution.

In this chapter, we will cover the following topics:

- Understanding the basic concepts of artificial neural networks and deep learning
- The Iris dataset and the **Multilayer Perceptron** (**MLP**) classifier
- Enhancing the performance of a deep learning classifier using network architecture optimization
- Further enhancing the performance of the deep learning classifier by combining network architecture optimization with hyperparameter tuning

We will start this chapter with an overview of artificial neural networks. If you are a seasoned data scientist, feel free to skip the introductory sections.

Technical requirements

In this chapter, we will be using Python 3 with the following supporting libraries:

- `deap`
- `numpy`
- `sklearn`

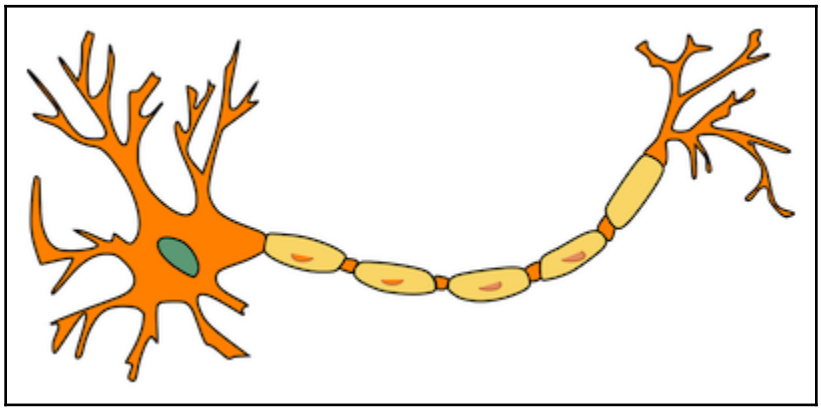
In addition, we will be using the UCI Iris flower dataset (`https://archive.ics.uci.edu/ml/datasets/Iris`).

The programs that will be used in this chapter can be found in this book's GitHub repository at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter09`.

Check out the following video to see the Code in Action:
`http://bit.ly/317KCXA`

Artificial neural networks and deep learning

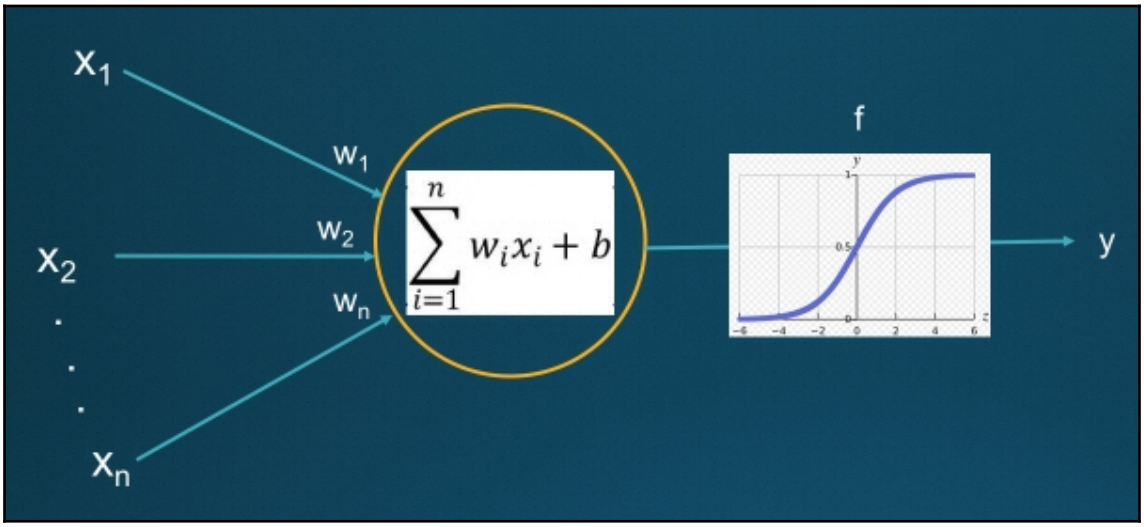
Neural networks are among the most commonly used models in machine learning and were inspired by the structure of the human brain. The basic building blocks of these networks are nodes, or neurons, which are based on the biological neuron cell, as depicted in the following diagram:



Biological neuron model
Source: https://pixabay.com/vectors/neuron-nerve-cell-axon-dendrite-296581/

The neuron cell's dendrites, which are surrounding the cell body on the left-hand side of the preceding diagram, are used as inputs from multiple similar cells, while the long axon, coming out of the cell body, serves as output and can be connected to multiple other cells.

This structure is mimicked by the artificial model called the **perceptron**, illustrated as follows:



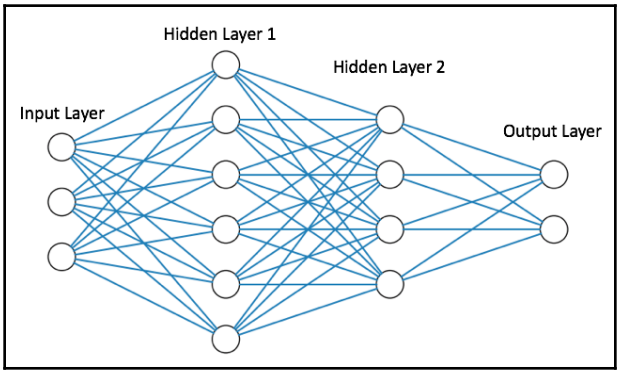
Artificial neuron model – the perceptron

The perceptron calculates the output by multiplying each of the input values by a certain weight; the results are accumulated and a bias value is added to the sum. A non-linear activation function then maps the result to the output. This functionality emulates the operation of the biological neuron, which fires (sends a series of pulses from its output) when the weighted sum of the inputs is above a certain threshold.

The perceptron model can be used for simple classification and regression tasks if we adjust its weight and bias values so that they map certain inputs to the desired output levels. However, a much more capable model can be constructed when connecting multiple perceptron units in a structure called the **Multilayer Perceptron**, which will be described in the next subsection.

Multilayer Perceptron

The **Multilayer Perceptron** (**MLP**) extends the idea of the perceptron by using numerous nodes, each one implementing a perceptron. The nodes in the MLP are arranged in layers, and each layer is connected to the next. The basic structure of the MLP is illustrated in the following diagram:



The basic structure of a Multilayer Perceptron

The Multilayer Perceptron consists of three main parts:

- **Input layer**: Receives the input values and connects each of them to every neuron in the next layer.
- **Output Layer**: Delivers the results calculated by the MLP. When the MLP is used as a classifier, each of the outputs represents one of the classes. When the MLP is used for regression, there will be a single output node, producing a continuous value.
- **Hidden Layer(s)**: Provides the true power and complexity of this model. While the preceding diagram shows only two hidden layers, there can be multiple hidden layers, each an arbitrary size, that are placed between the input and output layers. As the number of hidden layers grows, the network becomes deeper and is capable of performing an increasingly more complex and non-linear mapping between the inputs and the outputs.

Training this model involves adjusting the weight and bias values for each of the nodes. This is typically done using a family of algorithms called **backpropagation**. The basic principle of backpropagation is to minimize the error between the actual outputs and the desired ones by propagating the output error through the layers of the model, from the output layer inward. The weights of the various nodes are adjusted so that the weights that contributed more to the error are adjusted the most.

For many years, the computational limitations of backpropagation algorithms restricted MLPs to no more than two or three hidden layers, until new developments changed matters dramatically. These will be explained in the next section.

Deep learning and convolutional neural networks

In recent years, backpropagation algorithms have made a leap forward, enabling the use of a large number of hidden layers in a single network. In these deep neural networks, each layer can interpret a combination of several simpler abstract concepts that were learned by the nodes of the previous layer and produce higher-level concepts. For example, when implementing a face recognition task, the first layer will process the pixels of an image and learn to detect edges in different orientations. The next layer may assemble these into lines, corners, and so on, up to a layer that detects facial features such as nose and lips, and finally, one that combines these into the complete concept of a face.

Further advancements have brought about the idea of **convolutional neural network**s. These structures can reduce the count of nodes in deep neural networks that process two-dimensional information (such as images) by treating nearby inputs differently compared to inputs that are far apart. As a result, these models have proved especially successful when it comes to image and video processing tasks. Besides **fully connected layers**, similar to the hidden layers in the Multilayer Perceptron, these networks utilize **pooling (down-sampling) layers**, which aggregate outputs of neurons from preceding layers, and **convolutional layers**, which can be used as filters for detecting certain features (such as an edge in a particular orientation).

Training deep learning models can be computationally intensive and is often done with the aid of **Graphics Processing Units** (**GPUs**), which are more efficient than ordinary CPUs in implementing the backpropagation algorithm. Specialized deep learning libraries, such as TensorFlow, are capable of utilizing GPU-based computing platforms. In this chapter, however, for the sake of simplicity, we will be using the MLP implementation offered by the `sklearn` library and a simple dataset. The principles that will be used, however, still apply to more complex networks and datasets.

In the next section, we will find out how the architecture of the MLP can be optimized using a genetic algorithm.

Optimizing the architecture of a deep learning classifier

When creating a neural network model so that we can carry out a given machine learning task, one crucial design decision that needs to be made is the configuration of the network architecture. In the case of the Multilayer Perceptron, the number of nodes in the input and output layers is determined by the characteristics of the problem at hand. Therefore, the choices to be made are about the hidden layers – how many layers, and how many nodes in each layer. Some rules of thumb can be employed for making these decisions, but in many cases, identifying the best choices can turn into a cumbersome trial-and-error process.

One way to handle network architecture parameters is to consider them as hyperparameters of the model since they need to be determined before training is done and affect the training's results. In this section, we are going to apply this approach and use the genetic algorithms approach to find the best combination of hidden layers, in a similar manner to the way we went about choosing the best hyperparameter values in the previous chapter. Let's start with the task we want to tackle – the Iris flower classification.

The Iris flower dataset

Perhaps the most well-studied dataset, the Iris flower dataset (`https://archive.ics.uci.edu/ml/datasets/Iris`) contains measurements of the sepal and petal parts of three Iris species (Iris setosa, Iris virginica, and Iris versicolor), as taken by biologists in 1936.

The dataset contains 50 samples from each of the three species, and consists of the following four features:

- **sepal_length** (cm)
- **sepal_width** (cm)
- **petal_length** (cm)
- **petal_width** (cm)

This dataset is directly available via the `sklearn` library, and can be initialized as follows:

```
from sklearn import datasets

data = datasets.load_iris()
X = data['data']
y = data['target']
```

In our experiments, we will be using an MLP classifier in conjunction with this dataset and harness the power of genetic algorithms to find the network architecture – the number of hidden layers and the number of nodes in each layer – that will yield the best classification accuracy.

Since we are using the genetic algorithms approach, the first thing we need to do is find a way to represent this architecture using a chromosome, as described in the next subsection.

Representing the hidden layer configuration

Since the architecture of the MLP is determined by the hidden layer configuration, let's explore how this configuration can be represented in our solution. The hidden layer configuration of the `sklearn` Multilayer Perceptron (`https://scikit-learn.org/stable/modules/neural_networks_supervised.html`) model is conveyed via the `hidden_layer_sizes` tuple, which is sent as a parameter to the model's constructor. By default, the value of this tuple is (100,), which means a single hidden layer of 100 nodes. If we wanted, for example, to configure the MLP with three hidden layers of 20 nodes each, this parameter's value would be (20, 20, 20). Before we implement our genetic algorithm-based optimizer for the hidden layer configuration, we need to define a chromosome that can be translated into this pattern.

To accomplish this, we need to come up with a chromosome that can both express the number of layers and the number of nodes in each layer. A variable-length chromosome that can be directly translated into the variable-length tuple that's used as the model's `hidden_layer_sizes` parameter, is one option; however, this approach would require custom, possibly cumbersome, genetic operators. To be able to use our standard genetic operators, we will use a fixed-length representation. When using this approach, the maximum number of layers is decided in advance, and all the layers are always represented, but not necessarily expressed in the solution. For example, if we decide to limit the network to four hidden layers, the chromosome will look as follows:

$$[n1, n2, n3, n4]$$

Here, n_i denotes the number of nodes in the layer i.

However, to control the actual number of hidden layers in the network, some of these values may be zero, or negative. Such a value means that no more layers will be added to the network. The following examples illustrate this method:

- The chromosome [10, 20, -5, 15] is translated into the tuple (10, 20) since the -5 terminates the layer count.
- The chromosome [10, 0, -5, 15] is translated into the tuple (10,) since the 0 terminates the layer count.
- The chromosome [10, 20, 5, -15] is translated into the tuple (10, 20, 5) since the -15 terminates the layer count.
- The chromosome [10, 20, 5, 15] is translated into the tuple (10, 20, 5, 15).

To guarantee that there is at least one hidden layer, we can make sure that the first parameter is always greater than zero. The other layer parameters can have varying distributions around zero so that we can control their chances of being the terminating parameters.

In addition, even though this chromosome is made up of integers, we chose to utilize float numbers instead, just like we did in the previous chapter for various types of variables. Using a list of float numbers is convenient as it allows us to use existing genetic operators while being able to easily extend the chromosome so that it includes other parameters of different types. We will do this later on. The float numbers can be translated back into integers using the `round()` function. A couple of examples of this generalized approach are as follows:

- The chromosome [9.35, 10.71, -2.51, 17.99] is translated into the tuple (9, 11)
- The chromosome [9.35, 10.71, 2.51, -17.99] is translated into the tuple (9, 11, 3)

To evaluate a given architecture-representing chromosome, we will need to translate it back into the tuple of layers, create the MLP classifier implementing these layers, train it, and evaluate it. We will learn how to do this in the next subsection.

Evaluating the classifier's accuracy

Let's start with a Python class that encapsulates the MLP classifier's accuracy evaluation for the Iris dataset. The class is called `MlpLayersTest` and can be found in the `mlp_layers_test.py` file, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter09/mlp_layers_test.py`.

The main functionality of this class is highlighted as follows:

1. The `convertParam()` method of the class takes a list called `params`. This is actually the chromosome that we described in the previous subsection and contains the float values that represent up to four hidden layers. The method transforms this list of floats into the `hidden_layer_sizes` tuple:

```
if round(params[1]) <= 0:
    hiddenLayerSizes = round(params[0]),
elif round(params[2]) <= 0:
    hiddenLayerSizes = (round(params[0]), round(params[1]))
elif round(params[3]) <= 0:
    hiddenLayerSizes = (round(params[0]), round(params[1]),
round(params[2]))
else:
    hiddenLayerSizes = (round(params[0]), round(params[1]),
round(params[2]), round(params[3]))
```

2. The `getAccuracy()` method takes the `params` list representing the configuration of the hidden layers, uses the `convertParam()` method to transform it into the `hidden_layer_sizes` tuple, and initializes the MLP classifier with this tuple:

```
hiddenLayerSizes = self.convertParams(params)
self.classifier =
MLPClassifier(hidden_layer_sizes=hiddenLayerSizes)
```

   Then, it finds the accuracy of the classifier using the k-fold cross-validation that we created for the Wine dataset:

```
cv_results = model_selection.cross_val_score(self.classifier,
                                             self.X,
                                             self.y,
                                             cv=self.kfold,
scoring='accuracy')
return cv_results.mean()
```

The `MlpLayersTest` class is utilized by the genetic algorithm-based optimizer. We will explain this in the next section.

Optimizing the MLP architecture using genetic algorithms

Now that we have a way to represent the architecture configuration of the MLP that's used to classify the Iris flower dataset and a way to determine the accuracy of the MLP for each configuration, we can move on and create a genetic algorithm-based optimizer to search for the configuration – the number of hidden layers (up to 4, in our case) and the number of nodes in each layer – that will yield the best accuracy. This solution is implemented by the Python program `01-optimize-mlp-layers.py`, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter09/01-optimize-mlp-layers.py`.

The following steps describe the main parts of this program:

1. We start by setting the lower and upper boundary for each of the float values representing a hidden layer. The first hidden layer is given the range [5, 15], while the rest of the layers start from increasingly larger negative values, which increases their chances of terminating the layer count:

```
# [layer_layer_1_size, hidden_layer_2_size,
hidden_layer_3_size, hidden_layer_4_size]
BOUNDS_LOW =  [ 5,  -5, -10, -20]
BOUNDS_HIGH = [15,  10,  10,  10]
```

2. Then, we create an instance of the `MlpLayersTest` class, which will allow us to test the various combinations of the hidden layers' architecture:

```
test = mlp_layers_test.MlpLayersTest(RANDOM_SEED)
```

3. Since our goal is to maximize the accuracy of the classifier, we define a single objective, maximizing fitness strategy:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

4. Now, we employ the same approach we used in the previous chapter: since the solution is represented by a list of float values, each of a different range, we use the following loop to iterate over all pairs of lower-bound, upper-bound values, and for each range, we create a separate toolbox operator (`layer_size_attribute`) that will be used to generate random float values in the appropriate range later:

```
for i in range(NUM_OF_PARAMS):
    # "layer_size_attribute_0", "layer_size_attribute_1", ...
    toolbox.register("layer_size_attribute_" + str(i),
```

```
                                random.uniform,
                                BOUNDS_LOW[i],
                                BOUNDS_HIGH[i])
```

5. Then, we create the `layer_size_attributes` tuple, which contains the separate float number generators we just created for each hidden layer:

```
layer_size_attributes = ()
for i in range(NUM_OF_PARAMS):
    layer_size_attributes = layer_size_attributes + \
(toolbox.__getattribute__("layer_size_attribute_" + str(i)),)
```

6. Now, we can use this `layer_size_attributes` tuple in conjunction with DEAP's built-in `initCycle()` operator to create a new `individualCreator` operator that fills up an individual instance with a combination of randomly generated hidden layer size values:

```
toolbox.register("individualCreator",
                 tools.initCycle,
                 creator.Individual,
                 layer_size_attributes,
                 n=1)
```

7. Then, we instruct the genetic algorithm to use the `getAccuracy()` method of the `MlpLayersTest` instance for fitness evaluation. As a reminder, the `getAccuracy()` method, which we described in the previous subsection, converts the given individual – a list of four floats – into a tuple of hidden layer sizes. These are used to configure the MLP classifier. Then, we train the classifier and evaluate its accuracy using k-fold cross-validation:

```
def classificationAccuracy(individual):
    return test.getAccuracy(individual),

toolbox.register("evaluate", classificationAccuracy)
```

8. As for the genetic operators, we repeat the configuration from the previous chapter. While for the selection operator, we use the usual tournament selection with a tournament size of 2, we choose crossover and mutation operators that are specialized for bounded float-list chromosomes, and provide them with the boundaries we defined for each hidden layer:

```
toolbox.register("select", tools.selTournament, tournsize=2)

toolbox.register("mate",
                 tools.cxSimulatedBinaryBounded,
                 low=BOUNDS_LOW,
```

```
                        up=BOUNDS_HIGH,
                        eta=CROWDING_FACTOR)

    toolbox.register("mutate",
                        tools.mutPolynomialBounded,
                        low=BOUNDS_LOW,
                        up=BOUNDS_HIGH,
                        eta=CROWDING_FACTOR,
                        indpb=1.0 / NUM_OF_PARAMS)
```

9. In addition, we continue to use the elitist approach, where the **hall of fame** (**HOF**) members – the current best individuals – are always passed untouched to the next generation:

```
    population, logbook = elitism.eaSimpleWithElitism(population,
                                                       toolbox,
                                                       cxpb=P_CROSSOVER,
                                                       mutpb=P_MUTATION,
                                                       ngen=MAX_GENERATIONS,
                                                       stats=stats,
                                                       halloffame=hof,
                                                       verbose=True)
```

By running the algorithm for 10 generations with a population size of 20, we get the following outcome:

```
gen nevals max avg
0 20 0.666667 0.416333
1 17 0.693333 0.487
2 15 0.76 0.537333
3 14 0.76 0.550667
4 17 0.76 0.568333
5 17 0.76 0.653667
6 14 0.76 0.589333
7 15 0.76 0.618
8 16 0.866667 0.616667
9 16 0.866667 0.666333
10 16 0.866667 0.722667
- Best solution is: 'hidden_layer_sizes'=(15, 5, 8) , accuracy =
0.8666666666666666
```

The preceding results indicate that, within the ranges we defined, the best combination that was found was of three hidden layers of size 15, 5, and 8, respectively. The classification accuracy that we achieved with these values is about 86.7%.

This accuracy seems to be a reasonable result for the problem at hand. However, there's more we can do to improve it even further.

Combining architecture optimization with hyperparameter tuning

While optimizing the network architecture configuration—the hidden layer parameters—we have been using the default parameters of the MLP classifier. However, as we saw in the previous chapter, tuning the various hyperparameters has the potential to increase the classifier's performance. Can we incorporate hyperparameter tuning into our optimization? As you may have guessed, the answer is yes. But first, let's take a look at the hyperparameters we would like to optimize.

The `sklearn` implementation of the MLP classifier contains numerous tunable hyperparameters. For our demonstration, we will concentrate on the following hyperparameters:

Name	Type	Description	Default value
`activation`	`{'tanh', 'relu', 'logistic'}`	Activation function for the hidden layers	`'relu'`
`solver`	`{'sgd', 'adam', 'lbfgs'}`	The solver for weight optimization	`'adam'`
`alpha`	`float`	Regularization term parameter	`0.0001`
`learning_rate`	`{'constant', 'invscaling', 'adaptive'}`	Learning rate schedule for weight updates	`'constant'`

As we saw in the previous chapter, a floating point-based chromosome representation allows us to combine various types of hyperparameters into the genetic algorithm-based optimization process. Since we used a floating-point-based chromosome to represent the configuration of the hidden layers, we can now incorporate other hyperparameters into the optimization process by augmenting the chromosome accordingly. Let's find out how we can do this.

Solution representation

Regarding our existing network architecture configuration of four floats of the form [*n1, n2, n3, n4*], we can add the following four hyperparameters:

- `activation` can have one of three values: `'tanh'`, `'relu'`, or `'logistic'`. This can be achieved by representing it as a float number in the range of `[0, 2.99]`. To transform the float value into one of the aforementioned values, we need to apply the `floor()` function to it, which will yield either `0`, `1`, or `2`. We then replace a value of `0` with `tanh`, a value of `1` with `relu`, and a value of `2` with `logistic`.
- `solver` can have one of three values: `'sgd'`, `'adam'`, or `'lbfgs'`. Just like the `activation` parameter, it can be represented using a float number in the range of `[0, 2.99]`.
- `alpha` is already a float, so no conversion is needed. It will be bound to the range of `[0.0001, 2.0]`.
- `learning_rate` can have one of three values: `'constant'`, `'invscaling'`, or `'adaptive'`. Once again, we can use a float number in the range of `[0, 2.99]` to represent its value.

Evaluating the classifier's accuracy

The class that will be used to evaluate the MLP classifier's accuracy for the given combination of hidden layers and hyperparameters is called `MlpHyperparametersTest`, and is contained in the file `mlp_hyperparameters_test.py`, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter09/mlp_hyperparameters_test.py`.

This class is based on the one we used to optimize the configuration of the hidden layers, `MlpLayersTest`, but with a few modifications. Let's go over these:

1. The `convertParam()` method now handles a `params` list where the first four entries (`params[0]` through `params[3]`) represent the sizes of the hidden layers, just as before, but in addition, `params[4]` through `params[7]` represent the four hyperparameters we added to the evaluation. Consequently, the method has been augmented with the following lines of code, allowing it to transform the rest of the given parameters (`params[4]` through `params[7]`) into their corresponding values, which can then be fed to the MLP classifier:

```
activation = ['tanh', 'relu', 'logistic'][floor(params[4])]
solver = ['sgd', 'adam', 'lbfgs'][floor(params[5])]
alpha = params[6]
learning_rate = ['constant', 'invscaling',
'adaptive'][floor(params[7])]
```

2. Similarly, the `getAccuracy()` method now handles the augmented `params` list. It configures the MLP classifier with the converted values of all these parameters, rather than just the hidden layer's configuration:

```
hiddenLayerSizes, activation, solver, alpha, learning_rate =
self.convertParams(params)

self.classifier = MLPClassifier(random_state=self.randomSeed,
hidden_layer_sizes=hiddenLayerSizes,
                                 activation=activation,
                                 solver=solver,
                                 alpha=alpha,
                                 learning_rate=learning_rate)
```

This `MlpHyperparametersTest` class is utilized by the genetic algorithm-based optimizer. We will look at this in the next section.

Optimizing the MLP's combined configuration using genetic algorithms

The genetic algorithm-based search for the best combination of hidden layers and hyperparameters is implemented by the Python program `02-optimize-mlp-hyperparameters.py`, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter09/02-optimize-mlp-hyperparameters.py`.

Thanks to the **unified** floating number representation that's used for all parameters, this program is almost identical to the one we used in the previous section to optimize the network architecture. The main difference is in the definition of the `BOUNDS_LOW` and `BOUNDS_HIGH` lists, which contain the ranges of the parameters. Regarding the four ranges we defined previously – one for each hidden layer – we will add four more, representing the additional four hyperparameters that we discussed earlier in this section:

```
# 'hidden_layer_sizes': first four values
# 'activation':    0..2.99
# 'solver':        0..2.99
# 'alpha':    0.0001..2.0
# 'learning_rate': 0..2.99
BOUNDS_LOW =  [ 5,  -5, -10, -20, 0,     0,     0.0001, 0    ]
BOUNDS_HIGH = [15,  10,  10,  10, 2.999, 2.999, 2.0,    2.999]
```

And that's all it takes – the program is able to handle the added parameters without any further changes.

Running the algorithm for five generations with a population size of 20 produces the following outcome:

```
gen nevals max avg
0 20 0.933333 0.447333
1 16 0.933333 0.631667
2 15 0.94 0.736667
3 16 0.94 0.849
4 15 0.94 0.889667
5 17 0.946667 0.937
- Best solution is:
 'hidden_layer_sizes'=(8, 8)
 'activation'='relu'
 'solver'='lbfgs'
 'alpha'=0.572971105096338
 'learning_rate'='invscaling'
 => accuracy = 0.9466666666666667
```

Please note that, due to variations between operating systems, the results that will be produced when you run this program on your system may be somewhat different from what's being shown here.

The preceding results indicate that, within the ranges we defined, the best combination that we found for the hidden layer configuration and hyperparameters was as follows:

- Two hidden layers, of 8 nodes each
- The `activation` parameter of the `'relu'` type – the same as the default value

- The `solver` parameter of the `'lbfgs'` type – rather than the default `'adam'`
- The `learning_rate` parameter of the `'invscaling'` type – instead of the default `'constant'`
- An `alpha` value of about 0.572 – considerably larger than the default value of 0.0001

This combined optimization resulted in a classification accuracy of about 94.7% – a significant improvement over the previous results, all while using fewer hidden layers and fewer nodes than before.

Summary

In this chapter, you were introduced to the basic concepts of artificial neural networks and deep learning. After getting acquainted with the Iris dataset and the **Multilayer Perceptron** (**MLP**) classifier, you were presented with the notion of network architecture optimization. Next, we demonstrated a genetic algorithm-based optimization of network architecture for the MLP classifier. Finally, we were able to combine network architecture optimization with model hyperparameter tuning with the genetic algorithms process and enhance the performance of the classifier even further.

So far, we have concentrated on supervised learning. In the next chapter, we will look into applying genetic algorithms to reinforcement learning, an exciting and fast-developing branch of machine learning.

Further reading

For more information on the topics that we covered in this chapter, please refer to the following resources:

- *Python Deep Learning - Second Edition*, *Gianmario Spacagna*, *Daniel Slater* et al. January 16, 2019
- *Neural Network Projects with Python*, *James Loy*, February 28, 2019
- scikit-learn Multilayer Perceptron Classifier: `https://scikit-learn.org/stable/modules/neural_networks_supervised.html`
- UCI Machine Learning Repository: `https://archive.ics.uci.edu/ml/index.php`

10
Reinforcement Learning with Genetic Algorithms

In this chapter, we will demonstrate how genetic algorithms can be applied to reinforcement learning—a fast-developing branch of machine learning that is capable of tackling complex tasks. We will do this by solving two benchmark environments from the OpenAI Gym toolkit. We will start by providing an overview of reinforcement learning, followed by a brief introduction to OpenAI Gym, a toolkit that can be used to compare and develop reinforcement learning algorithms, as well as a description of its Python-based interface. Then, we will undertake two Gym environments, *MountainCar* and *CartPole*, and develop genetic algorithm-based programs to solve the challenges they present.

In this chapter, we will cover the following topics:

- Understanding the basic concepts of reinforcement learning
- Becoming familiar with the OpenAI Gym project and its shared interface
- Using genetic algorithms to solve the OpenAI Gym *MountainCar* environment
- Using genetic algorithms, in combination with a neural network, to solve the OpenAI Gym *CartPole* environment

We will start this chapter by outlining the basic concepts of reinforcement learning. If you are a seasoned data scientist, feel free to skip this introductory section.

Technical requirements

In this chapter, we will be using Python 3 with the following supporting libraries:

- `deap`
- `numpy`
- `sklearn`
- `gym`—introduced in this chapter

The Gym environments that will be used in this chapter are *MountainCar-v0* (`https://gym.openai.com/envs/MountainCar-v0/`) and *CartPole-V1* (`https://gym.openai.com/envs/CartPole-v1/`).

The programs that will be used in this chapter can be found in this book's GitHub repository at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter10`.

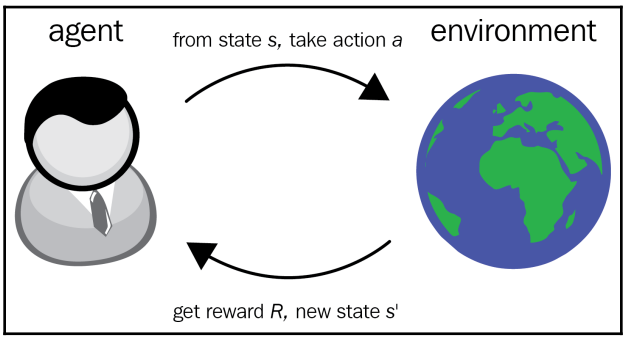
Check out the following video to see the Code in Action:
`http://bit.ly/2Oey7V0`

Reinforcement learning

In the previous chapters, we covered several topics related to machine learning and focused on supervised learning tasks. While supervised learning is immensely important and has a lot of real-life applications, reinforcement learning currently seems to be the most exciting and promising branch of machine learning. The reasons for this excitement include the complex, everyday-life-like tasks that reinforcement learning has the potential to handle. In March 2016, *AlphaGo*, a reinforcement learning-based system specializing in the highly complex game of *Go*, was able to defeat the person considered to be the greatest Go player of the past decade in a competition that was widely covered by the media.

While supervised learning requires **labeled** data for training—in other words, pairs of inputs and matching outputs—reinforcement learning does not present immediate **right/wrong** feedback; instead, it provides an environment where a longer-term, cumulative reward is sought after. This means that, sometimes, an algorithm will need to take a momentary step **backward** to eventually reach a longer-term goal, as will be demonstrated in our first example of this chapter.

The two main components of reinforcement learning task are the *environment* and the *agent*, as illustrated in the following diagram:



Reinforcement learning represented as an interaction between the agent and the environment

The **agent** represents an algorithm that interacts with the environment and attempts to solve a given problem by maximizing the cumulative reward.

The exchange that takes place between the agent and the environment can be expressed as a series of steps. In each step, the environment presents the agent with a certain state (*s*), also called an **observation**. The agent, in turn, performs an *action* (*a*). The environment responds with a new state (*s'*), as well as with an intermediate *reward* value (*R*). This exchange repeats until a certain stopping condition is met. The agent's goal is to maximize the sum of the reward values that are collected along the way.

Despite the simplicity of this formulation, it can be used to describe extremely complex tasks and situations, which makes reinforcement learning applicable to a wide range of applications, such as game theory, healthcare, control systems, supply-chain automation, and operations research.

The versatility of genetic algorithms will be demonstrated once more in this chapter since we will be harnessing them to assist with reinforcement learning tasks.

Genetic algorithms and reinforcement learning

Various dedicated algorithms have been developed for carrying out reinforcement learning tasks—Q-Learning, SARSA, and DQN, to name a few. However, since reinforcement learning tasks involve maximizing a long-term reward, we can think of them as optimization problems. As we have seen throughout this book, genetic algorithms can be used for solving optimization problems of various types. Therefore, genetic algorithms can be utilized for reinforcement learning as well, and in several different ways—two of them will be demonstrated in this chapter. In the first case, our genetic algorithm-based solution will directly provide the agent's optimal series of actions. In the second case, it will supply the optimal parameters for the neural controller providing these actions.

Before we start applying genetic algorithms to reinforcement learning tasks, let's get acquainted with the toolkit that will be used to conduct these tasks—OpenAI Gym.

OpenAI Gym

OpenAI Gym (`https://github.com/openai/gym`) is an open source library that was written to allow access to a standardized set of reinforcement learning tasks. It provides a toolkit that can be used to compare and develop reinforcement learning algorithms.

OpenAI Gym consists of a collection of environments, all presenting a common interface called `env`. This interface decouples the various environments from the agents, which can be implemented in any way we like—the only requirement from the agent is that it can interact with the environment(s) via the **env** interface. This will be described in the next subsection.

The basic package, `gym`, provides access to several environments and can be installed as follows:

```
pip install gym
```

Several other packages are available, such as `'Atari'`, `'Box2D'`, and `'MuJoCo'`, that provide access to numerous and diverse additional environments. Some of these packages have system dependencies and may only be available for certain operating systems. More information is available at `https://github.com/openai/gym#installation`.

The next subsection describes the interaction with the **env** interface.

The env interface

To create an environment, we need to use the `make()` method and the name of the desired environment, as follows:

```
env = gym.make('MountainCar-v0')
```

Detailed information about the available environments is available at the following links:

- `https://github.com/openai/gym/blob/master/docs/environments.md`
- `https://gym.openai.com/envs/`

Once the environment has been created, it can be initialized using the `reset()` method, as shown in the following code snippet:

```
observation = env.reset()
```

This method returns an `observation` object, describing the initial state of the environment. The content of the observation is environment-dependent.

Conforming with the reinforcement learning cycle that we described in the previous subsection, the ongoing interaction with the environment consists of sending it an action and, in return, receiving an intermediate reward and a new state. This is implemented by the `step()` method, as follows:

```
observation, reward, done, info = env.step(action)
```

In addition to the `observation` object, which describes the new state and the float `reward` value that represent the interim reward, this method returns the following values:

- `done`: This is a Boolean that turns true when the current run (also called **episode**) has ended, for example, the agent lost a life, or successfully completed the task.
- `info`: This is a dictionary containing optional, additional information that may be useful for debugging. However, it should not be used by the agent for learning.

At any point in time, the environment can be rendered for visual presentation, as follows:

```
env.render()
```

The rendered presentation is environment-specific.

Finally, an environment can be closed to invoke any necessary cleanup, as follows:

```
env.close()
```

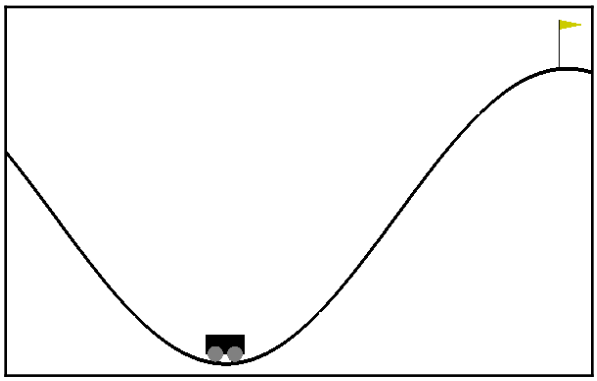
If this method isn't called, the environment will automatically close itself the next time Python runs its garbage collection process (the process of identifying and freeing memory that is no longer in use by the program), or when the program exits.

Detailed information about the `env` interface is available at `http://gym.openai.com/docs/`.

The complete cycle of interaction with the environment will be demonstrated in the next section, where we'll encounter our first gym challenge—the `MountainCar` environment.

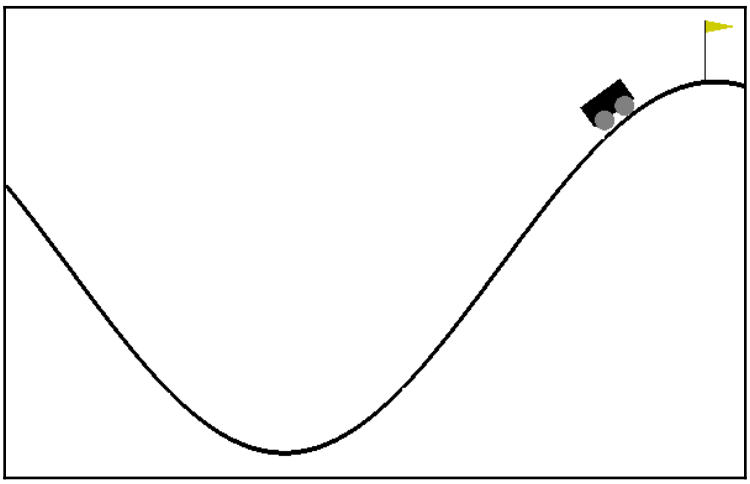
Solving the MountainCar environment

The MountainCar-v0 environment simulates a car in a one-dimensional track, situated between two hills. The simulation starts with the car placed between the hills, as shown in the following rendered output:



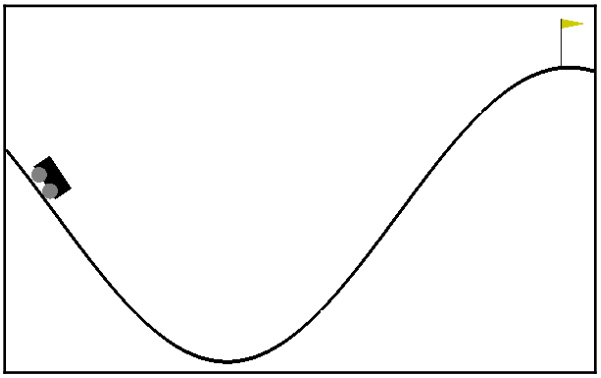
MountainCar simulation—starting point

The goal is to get the car to climb up the taller hill—the one on the right—and ultimately hit the flag:



MountainCar simulation—car climbing the hill on the right

The simulation is set up with the car's engine being too weak to directly climb the taller hill. The only way to reach the goal is to drive the car back and forth until enough momentum is built for climbing. Climbing the left hill can help to achieve this goal as reaching the left peak will bounce the car back to the right, as shown in the following screenshot:



MountainCar simulation—car bouncing off the hill on the left

This simulation is a great example where intermediate loss (moving left) can help to achieve the ultimate goal (going all the way to the right).

The expected action value in this simulation is an integer with of one of the three following values:

- 0: Push left
- 1: No push
- 2: Push right

The `observation` object contains two floats that describe the position and the velocity of the car. An example of this can be seen in the following code snippet:

```
[-1.0260268,  -0.03201975]
```

Finally, the `reward` value is `-1` for each time step, until the goal (located at position 0.5) is reached. The simulation will stop after 200 steps if the goal is not reached beforehand.

At the time of writing, the solved requirements of the gym environment for the `MountainCar` challenge haven't been defined yet, so we will simply attempt to hit the flag in 200 steps or less when given a fixed starting position and using a sequence of preselected actions. To find a sequence of actions that will get the car to climb the tall hill and hit the flag, we will craft a genetic algorithm-based solution. As usual, we will start by defining what a candidate solution for this challenge will look like.

More information about the *MountainCar-v0* environment can be found at the following links:

- **MountainCar-v0**: `https://gym.openai.com/envs/MountainCar-v0/`
- **MountainCar-v0**: `https://github.com/openai/gym/wiki/MountainCar-v0`

Solution representation

Since `MountainCar` is controlled by a sequence of actions, each with a value of 0 (push left), 1 (no push), or 2 (push right), and there can be up to 200 actions in a single episode, one obvious way to represent a candidate solution is by using a list of length 200, containing values of 0, 1, or 2. An example of this is as follows:

```
[0, 1, 2, 0, 0, 1, 2, 2, 1, ... , 0, 2, 1, 1]
```

The values in the list will be used as actions for controlling the car and hopefully driving it to the flag. If the car made it to the flag in less than 200 steps, the last few items in the list will not be used.

Next, we need to determine how to evaluate a given solution of this form.

Evaluating the solution

While evaluating a given solution, or when comparing two solutions, it is apparent that the reward value alone may not provide us with sufficient information. With the way the reward is defined, its value will always be -200 if we don't hit the flag. When we compare two candidate solutions that don't hit the flag, we would still like to know which one got closer to it and consider it a better solution. Therefore, we are going to use the final position of the car, in addition to the reward value, to determine the score of the solution. If the car did not hit the flag, the score will be the distance from the flag. Therefore, we will be looking for a solution that minimizes the score. If the car hits the flag, the base score will be zero, and from that, we deduct an additional value based on how many steps were left, making the score negative. Since we are looking for the smallest score possible, this arrangement will encourage solutions to hit the flag using the smallest possible amount of actions.

This score evaluation procedure is implemented by the `MountainCar` class, which will be described in the next subsection.

Python problem representation

To encapsulate the `MountainCar` challenge, we've created a Python class called `MountainCar`. This class is contained in the `mountain_car.py` file, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter10/mountain_car.py`.

The class is initialized with a random seed and provides the following methods:

- `getScore(actions)`: This calculates the score of a given solution, represented by the list of actions. The score is calculated by initiating an episode of the `MountainCar` environment and running it with the provided actions. The lower the score is, the better.
- `saveActions(actions)`: This saves a list of actions to a file using pickle (Python's object serialization and deserialization module).
- `replaySavedActions()`: This deserializes the last saved list of actions and replays it using the `replay` method.
- `replay(actions)`: This renders the environment and replays the given list of actions into it to visualize a given solution.

The main method of the class can be used after a solution has been serialized and saved using the `saveActions()` method. The main method will initialize the class and call `replaySavedActions()` to render and animate the last saved solution.

We typically use the main method to animate the best solution that's found by the genetic algorithm-based program. This will be described in the next subsection.

Genetic algorithms solution

To tackle the `MountainCar` challenge using the genetic algorithms approach, we've created the Python program, `01-solve-mountain-car.py`, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter10/01-solve-mountain-car.py`.

Since the solution representation we chose for this problem is a list containing the integer values 0, 1, or 2, this program bears resemblance to the one we used to solve the knapsack 0-1 problem in `Chapter 4`, *Combinatorial Optimization*, where solutions were represented as lists with the values 0 and 1.

The following steps describe the main parts of this program:

1. We start by creating an instance of the `MountainCar` class, which will allow us to score the various solutions for the `MountainCar` challenge:

```
car = mountain_car.MountainCar(RANDOM_SEED)
```

2. Since our goal is to minimize the score—in other words, hit the flag with the minimum step count; otherwise, get as close as possible to the flag—we define a single objective, minimizing the fitness strategy:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

3. Now, we need to create a toolbox operator that can produce one of the three allowed action values—0, 1, or 2:

```
toolbox.register("zeroOneOrTwo", random.randint, 0, 2)
```

4. This is followed by an operator that fills up an individual instance with these values:

```
toolbox.register("individualCreator",
                 tools.initRepeat,
                 creator.Individual,
                 toolbox.zeroOneOrTwo,
                 len(car))
```

5. Then, we instruct the genetic algorithm to use the `getScore()` method of the `MountainCar` instance for fitness evaluation.

   As a reminder, the `getScore()` method, which we described in the previous subsection, initiates an episode of the `MountainCar` environment and uses the given individual—a list of actions—as the inputs to the environment until the episode is done. Then, it evaluates the score—the lower the better—according to the final location of the car. If the car hit the flag, the score can get even lower, based on the number of steps it used to get there:

```
def carScore(individual):
    return car.getScore(individual),

toolbox.register("evaluate", carScore)
```

6. As for the genetic operators, we start with the usual tournament selection with a tournament size of 2. Since our solution representation is a list of the integer values 0, 1, or 2, we can use the two-point crossover operator, just like we did when the solution was represented by a list of 0 and 1 values. For mutation, however, rather than the FlipBit operator, which is typically used for the binary case, we need to use the `UniformInt` operator, which is used for a range of integer values, and configure it for the range of 0..2:

```
toolbox.register("select", tools.selTournament, tournsize=2)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutUniformInt, low=0, up=2,
indpb=1.0/len(car))
```

7. In addition, we continue to use the elitist approach, where the **hall of fame** (**HOF**) members—the current best individuals—are always passed untouched to the next generation:

```
population, logbook = elitism.eaSimpleWithElitism(population,
                                                   toolbox,
                                                   cxpb=P_CROSSOVER,
                                                   mutpb=P_MUTATION,
                                                   ngen=MAX_GENERATIONS,
                                                   stats=stats,
                                                   halloffame=hof,
                                                   verbose=True)
```

8. After the run, we print the best solution and save it so that we can animate it using the replay capability we built into the `MountainCar` class later:

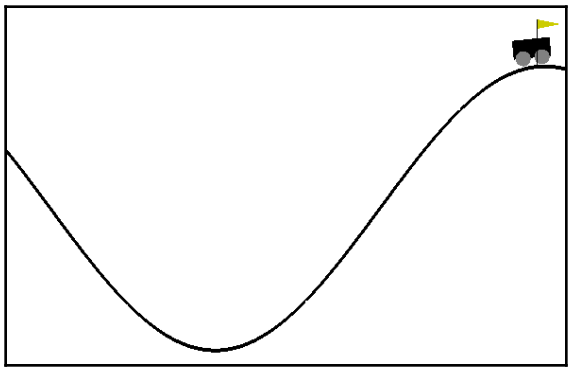
```
best = hof.items[0]
print("Best Solution = ", best)
print("Best Fitness = ", best.fitness.values[0])
car.saveActions(best)
```

Running the algorithm for 80 generations and with a population size of 100, we get the following outcome:

```
gen nevals min avg
0 100 0.659205 1.02616
1 78 0.659205 0.970209
...
60 75 0.00367593 0.100664
61 73 0.00367593 0.0997352
62 77 -0.005 0.100359
63 73 -0.005 0.103559
...
67 78 -0.015 0.0679005
68 80 -0.015 0.0793169
...
79 76 -0.02 0.020927
80 76 -0.02 0.0175934

Best Solution = [0, 1, 2, 0, 0, 1, 2, 2, 2, 2, 2, 2, 1, 1, 2, 2, 0, ... ,
1, 0, 2, 2, 0, 2, 1]
Best Fitness = -0.02
```

From the preceding output, we can see that, after about 60 generations, the best solution(s) started hitting the flag, producing score values of zero or less. From here on, the best solutions are hitting the flag in fewer steps, hence producing increasingly negative score values.

As we already mentioned, the best solution was saved at the end of the run, and we can now replay it by running the `mountain_car` program. The following screenshot illustrates how the actions of our solution drive the car back and forth between the two peaks, climbing higher each time, until the car is able to climb the lower hill on the left. Then, it bounces back, which means we have enough momentum to continue climbing the higher hill on the right, ultimately hitting the flag:



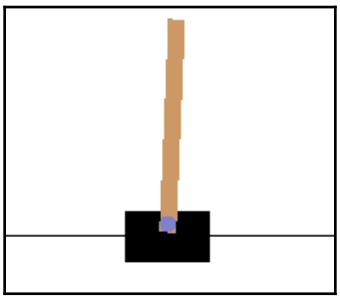
MountainCar simulation—car reaching the goal

While solving it was a lot of fun, the way this environment is set up did not require us to dynamically interact with it. We were able to climb the hill using a sequence of actions that our algorithm put together, based on the initial location of the car. In contrast to that, the next environment we are about to tackle—named CartPole—requires us to dynamically calculate our action at any time step, based upon the latest observation produced. Read on to find out how we can make this work.

Solving the CartPole environment

The CartPole-v1 environment simulates a balancing act of a pole, hinged at its bottom to a cart, which moves left and right along a track. Balancing the pole upright is carried out by applying to the cart one unit of force—to the right or to the left—at a time.

The pole, acting as a pendulum in this environment, starts upright within a small random angle, as shown in the following rendered output:



CartPole simulation—starting point

Our goal is to keep the pendulum from falling over to either side for as long as possible, that is, up to 500 time steps. For every time step that the pole remains upright, we get a reward of +1, so the maximum total reward is 500. The episode will end prematurely if one of the following occurs during the run:

- The angle of the pole from the vertical position exceeds 15 degrees.
- The cart's distance from the center exceeds 2.4 units.

Consequently, the total reward in these cases will be smaller than 500.

The expected `action` value in this simulation is an integer of one of the two following values:

- 0: Push the cart to the left
- 1: Push the cart to the right

The `observation` object contains four floats that describe the following information:

- Cart position (between -2.4 and 2.4)
- Cart velocity (between -Inf and Inf)
- Pole angle (between -41.8° and 41.8°)
- Pole velocity at the pole's tip (between -Inf and Inf)

For example, we could have [0.33676587, 0.3786464, -0.00170739, -0.36586074].

In our proposed solution, we will be using these values as inputs at every time step to determine which action to take. We will be doing this with the aid of a neural network-based controller. This is further described in the next subsection.

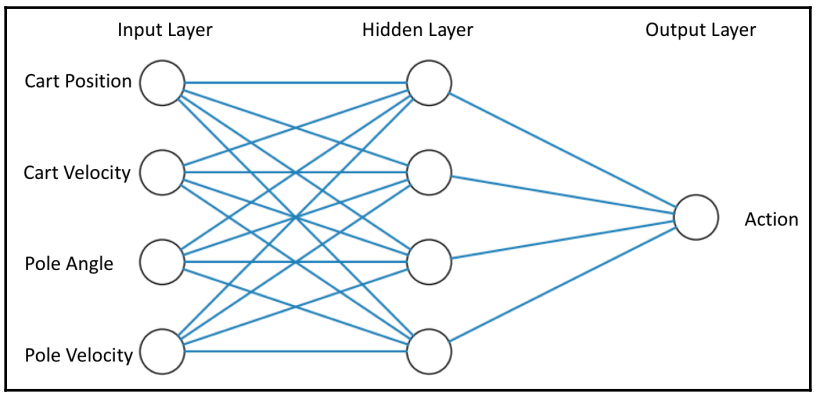
More information about the CartPole-v1 environment is available at the following links:

- `https://gym.openai.com/envs/CartPole-v1/`
- `https://github.com/openai/gym/wiki/CartPole-v0`

Controlling the CartPole with a neural network

To successfully carry out the CartPole challenge, we would like to dynamically respond to the changes in the environment. For example, when the pole starts leaning in one direction, we should probably move the cart in that direction, but possibly stop pushing when it starts to stabilize. So, the reinforcement learning task here can be thought of as teaching a controller to balance the pole by mapping the four available inputs—cart position, cart velocity, pole angle, and pole velocity—into the appropriate action at each time step. How can we implement such mapping?

One good way to implement this mapping is by using a neural network. As we saw in `Chapter 9`, *Architecture Optimization of Deep Learning Networks*, a neural network, such as a **Multilayer Perceptron** (**MLP**), can implement complex mappings between its inputs and outputs. This mapping is done with the aid of the network parameters, namely, the weights and biases of the active nodes in the network, as well as the transfer functions that are implemented by these nodes. In our case, we will use a network with a single hidden layer of four nodes. In addition, the input layer consists of four nodes, one for each of the input values provided by the environment, while the output layer has a single node since we only have one output value—the action to be taken. This network structure can be illustrated with the following diagram:



Structure of the neural network that's used to control the cart

As we have seen already, the values of the weights and biases of a neural network are typically set during a process in which the network is trained. The interesting part is that, so far, we have only seen this kind of neural network being trained using the backpropagation algorithm while implementing supervised learning. That is, in each of the previous cases, we had a training set of inputs and matching outputs, and the network was trained to map each given input to its matching given output. Here, however, as we practice reinforcement learning, we don't have such training information available. Instead, we only know how well the network did at the end of the episode. This means that instead of using the conventional training algorithms, we need a method that will allow us to find the best network parameters—weights and biases—based on the results that are obtained by running the environment's episodes. This is exactly the kind of optimization that genetic algorithms are good at: finding a set of parameters that will give us the best results, as long as you have a way to evaluate and compare these results. To do that, we need to figure out how to represent the network's parameters, as well as how to evaluate a given set of these parameters. Both of these topics will be covered in the next subsection.

Solution representation and evaluation

Since we have decided to control the cart in the CartPole challenge using a neural network of the Multilayer Perceptron type, the set of parameters that we will need to optimize are the network's weights and biases, as follows:

- **Input layer**: This layer does not participate in the network mapping; instead, it receives the input values and forwards them to every neuron in the next layer. Therefore, no parameters are needed for this network.
- **Hidden layer**: Each node in this layer is fully connected to each of the inputs, and therefore requires four weights in addition to a single bias value.
- **Output layer**: The single node in this layer is connected to each of the nodes in the hidden layer, and therefore requires four weights in addition to a single bias value.

In total, we have 20 weight values and five bias values we need to find, all of the float type. Therefore, each potential solution can be represented as a list of 25 float values, like so:

```
[0.9505049282421143, -0.8068797228337171, -0.45488246459260073,
-0.7598208314027836,
... , 0.4039043861825575, -0.874433212682847, 0.9461075409693256,
0.6720551701599038]
```

Evaluating a given solution means creating our MLP with the correct dimensions—four inputs, a four-node hidden layer, and a single output—and assigning the weight and bias values from our list of floats to the various nodes. Then, we need to use this MLP as the controller for the cart pole during one episode. The resulting total reward of the episode is used as the score value for this solution. In contrast to the previous task, here, we're aiming to *maximize* the score that's achieved. This score evaluation procedure is implemented by the `CartPole` class, which will be described in the next subsection.

Python problem representation

To encapsulate the CartPole challenge, we've created a Python class called `CartPole`. This class is contained in the `cart_pole.py` file, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter10/cart_pole.py`.

The class is initialized with an optional random seed and provides the following methods:

- `initMlp()`: This initializes a Multilayer Perceptron regressor with the desired network architecture (layers) and network parameters (weights and biases), which are derived out of the list of floats representing a candidate solution.
- `getScore()`: This calculates the score of a given solution, represented by the list of float-valued network parameters. This is done by creating a corresponding MLP regressor, initiating an episode of the CartPole environment, and running it with the MLP controlling the actions, all while using the observations as inputs. The higher the score is, the better.
- `saveParams()`: This serializes and saves a list of network parameters using pickle.
- `replayWithSavedParams()`: This deserializes the latest saved list of network parameters and uses it to replay an episode using the `replay` method.
- `replay()`: This renders the environment and uses the given network parameters to replay an episode, visualizing a given solution.

The main method of the class should be used after a solution has been serialized and saved using the `saveParams()` method. The main method will initialize the class and call `replayWithSavedParams()` to render and animate the saved solution.

We will typically use the main method to animate the best solution that's found by our genetic algorithm-driven solution. This will be described in the next subsection.

Genetic algorithms solution

To interact with the CartPole environment and solve it using a genetic algorithm, we've created the Python program, `02-solve-cart-pole.py`, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter10/02-solve-cart-pole.py`.

Since we are using a list of floats to represent a solution—the network's weights and biases—this program is very similar to the function optimization programs we looked at in `Chapter 6`, *Optimizing Continuous Functions*, such as the one we used for the `Eggholder` function's optimization.

The following steps describe the main parts of this program:

1. We start by creating an instance of the `CartPole` class, which will allow us to test the various solutions for the CartPole challenge:

```
cartPole = cart_pole.CartPole(RANDOM_SEED)
```

2. Next, we set the upper and lower boundaries for the float values we will be searching. Since all of our values represent weights and biases within a neural network, we need to set the range so that it's between -1.0 and 1.0 in every dimension:

```
BOUNDS_LOW, BOUNDS_HIGH = -1.0, 1.0
```

3. As you may recall, our goal in this challenge is to maximize the score—the duration we are able to keep the pole balanced. To do so, we define a single objective, maximizing fitness strategy:

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

4. Now, we need to create a helper function for creating random real numbers that are uniformly distributed within a given range. This function assumes that the range is the same for every dimension, as is the case in our solution:

```
def randomFloat(low, up):
    return [random.uniform(l, u) for l, u in zip([low] *
NUM_OF_PARAMS, [up] * NUM_OF_PARAMS)]
```

5. Now, we use this function to create an operator that randomly returns a list of floats in the desired range that we set earlier:

```
toolbox.register("attrFloat", randomFloat, BOUNDS_LOW,
BOUNDS_HIGH)
```

6. This is followed by an operator that fills up an individual instance using the preceding operator:

```
toolbox.register("individualCreator",
                 tools.initIterate,
                 creator.Individual,
                 toolbox.attrFloat)
```

7. Then, we instruct the genetic algorithm to use the `getScore()` method of the `CartPole` instance for fitness evaluation.

   As a reminder, the `getScore()` method, which we described in the previous subsection, initiates an episode of the CartPole environment. During this episode, the cart is controlled by a single-hidden layer MLP. The weight and bias values of this MLP are populated by the list of floats representing the current solution. Throughout the episode, the MLP dynamically maps the observation values of the environment to an action of right or left. Once the episode is done, the score is the total reward, which equates to the amount of time steps that the MLP was able to keep the pole balanced—the higher, the better:

```
def score(individual):
    return cartPole.getScore(individual),

toolbox.register("evaluate", score)
```

8. It's time to choose our genetic operators. Once again, we'll use tournament selection with a tournament size of 2 as our selection operator. Since our solution representation is a list of floats in a given range, we'll use the specialized continuous bounded crossover and mutation operators provided by the DEAP framework—`cxSimulatedBinaryBounded` and `mutPolynomialBounded`, respectively:

```
toolbox.register("select", tools.selTournament, tournsize=2)

toolbox.register("mate",
                 tools.cxSimulatedBinaryBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR)

toolbox.register("mutate",
                 tools.mutPolynomialBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR,
                 indpb=1.0/NUM_OF_PARAMS)
```

9. And, as usual, we use the elitist approach, where the HOF members—the current best individuals—are always passed untouched to the next generation:

```
population, logbook = elitism.eaSimpleWithElitism(population,
                                                   toolbox,
                                                   cxpb=P_CROSSOVER,
                                                   mutpb=P_MUTATION,
                                                   ngen=MAX_GENERATIONS,
                                                   stats=stats,
                                                   halloffame=hof,
                                                   verbose=True)
```

10. After the run, we print the best solution and save it so that we can animate it using the replay capability we built into the `MountainCar` class later:

```
best = hof.items[0]
print("Best Solution = ", best)
print("Best Score = ", best.fitness.values[0])
cartPole.saveParams(best)
```

11. In addition, since the CartPole challenge provides the solved requirements definition, we are going to check whether our solution fulfills these requirements. At the time of writing, the challenge is described as *considered solved when the average reward is greater than or equal to 195.0 over 100 consecutive trials*: `https://github.com/openai/gym/wiki/CartPole-v0#solved-requirements`.

    This definition was probably created when the maximum time steps in a single episode was 200. However, it seems that, since then, the episode length has increased to 500 time steps. So, instead, we will aim to meet or exceed an average of 490.0 over 100 consecutive trials. We will check whether the requirement was fulfilled by running 100 consecutive tests using our best individual and averaging the results from all the tests, as follows:

```
scores = []
for test in range(100):
    scores.append(cart_pole.CartPole().getScore(best))
print("scores = ", scores)
print("Avg. score = ", sum(scores) / len(scores))
```

Note that, during these test runs, the CartPole problem is randomly initiated each time, so each episode starts from a slightly different starting condition and can potentially yield a different result.

It is time to find out how well we did in this challenge. By running the algorithm for 10 generations with a population size of 20, we get the following outcome:

```
gen nevals max avg
0 20 41 13.7
1 15 54 17.3
...
5 16 157 63.9
6 17 500 87.2
...
9 15 500 270.9
10 13 500 420.3

Best Solution = [0.733351790484474, -0.8068797228337171,
-0.45488246459260073, ...
Best Score = 500.0
```

From the preceding output, we can see that, after just six generations, the best solution(s) reached the maximum score of 500, balancing the pole for the entire episode's time.

Looking at the results of the following test, it seems that all 100 tests ended with a perfect score of 500:

```
Running 100 episodes using the best solution...
scores = [500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0, 500.0,
500.0, 500.0, 500.0,
...
500.0, 500.0, 500.0, 500.0]
Avg. score = 500.0
```

As we mentioned previously, each of these 100 runs is done with a slightly different random starting point. However, the controller is powerful enough to balance the pole for the entire run, each and every time. To watch the controller in action, we can play a CartPole episode—or several episodes—with the results we saved previously by launching the `cart_pole` program. The animation illustrates how the controller dynamically responds to the pole's movement by applying actions that keep the pole balanced on the cart for the entire episode.

If you would like to contrast these results with less-than-perfect ones, you are encouraged to repeat this experiment with three (or even two) nodes in the hidden layer instead of four—just change the `HIDDEN_LAYER` constant value accordingly in the `CartPole` class.

Summary

In this chapter, you were introduced to the basic concepts of reinforcement learning. After getting acquainted with the OpenAI Gym toolkit, you were presented with the `MountainCar` challenge, where a car needs to be controlled in a way that will allow it to climb the taller of two mountains. After solving this challenge using genetic algorithms, you were introduced to the next challenge, `CartPole`, where a cart is to be precisely controlled to keep an upright pole balanced. We were able to solve this challenge by combining the power of a neural network-based controller with genetic algorithm-guided training.

In the next chapter, we will transition to the world of art and find out how genetic algorithms can be used to reconstruct images of famous paintings with a set of semi-transparent overlapping shapes.

Further reading

For more information about the topics that we covered in this chapter, please refer to the following resources:

- *Python Reinforcement Learning*, Rajalingappaa Shanmugamani and Sudharsan Ravichandiran, et al., April 17, 2019
- *Deep Reinforcement Learning Hands-On*, Maksim Lapan, June 21, 2018
- OpenAI Gym documentation: `http://gym.openai.com/docs/`
- *OpenAI Gym* (White Paper), Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, Wojciech Zaremba: `https://arxiv.org/abs/1606.01540`

4
Section 4: Related Technologies

This section describes several optimization techniques related to genetic algorithms, as well as other biologically inspired computational algorithms.

This section comprises the following chapters:

- `Chapter 11`, *Genetic Image Reconstruction*
- `Chapter 12`, *Other Evolutionary and Bio-Inspired Computation Techniques*

11
Genetic Image Reconstruction

In this chapter, we are going to experiment with one of the most popular ways genetic algorithms have been applied to image processing – the reconstruction of an image with a set of semi-transparent polygons. Along the way, we will gain useful experience in image processing, coupled with a visual insight into the evolutionary process.

We will start with an overview of image processing in Python and get acquainted with three useful libraries – `Pillow`, `scikit-image`, and `opencv-python`. Then, we will find out how an image can be drawn from scratch using polygons and how the difference between two images can be calculated. Next, we will develop a genetic algorithm-based program to reconstruct a segment of a famous painting using polygons and examine the results.

In this chapter, we will cover the following topics:

- Getting familiar with several image processing libraries for Python
- Understanding how to programmatically draw an image using polygons
- Finding out how to programmatically compare two given images
- Using genetic algorithms, in combination with image processing libraries, to reconstruct an image using polygons

We will start this chapter by providing an overview of the image reconstruction task.

Technical requirements

In this chapter, we will be using Python 3 with the following supporting libraries:

- `deap`
- `numpy`
- `matplotlib`
- `seaborn`
- `Pillow` (PIL fork) – introduced in this chapter
- `scikit-image` (skimage) – introduced in this chapter
- `OpenCV-Python` (cv2) – introduced in this chapter

The programs that will be used in this chapter can be found in this book's GitHub repository at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter11`.

Check out the following video to see the Code in Action:
`http://bit.ly/2u1ytHz`

Reconstructing images with polygons

One of the most popular examples of using genetic algorithms in image processing is the reconstruction of a given image with a set of semi-transparent, overlapping shapes. Besides the fun aspect and the opportunity to gain experience in image processing, these experiments provide an excellent visual insight into the evolutionary process and could potentially lead to a better understanding of visual arts, as well as developments in image analysis and image compression.

In these image reconstruction experiments – multiple variations of which can be found on the internet – a familiar image, often a famous painting or a segment of it, is used as a reference. The goal is to construct a similar image by assembling a collection of overlapping shapes, typically polygons, of varying colors and transparencies.

Here, we will address this challenge by utilizing the genetic algorithms approach and the `deap` library, just like we've done for numerous types of problems throughout this book. However, since we will need to draw images and compare them to a reference image, let's get acquainted with the basics of image processing in Python.

Image processing in Python

To achieve our goal, we will need to carry out various image processing operations; for example, we will need to create an image from scratch, draw shapes onto an image, plot an image, open an image file, save an image to a file, compare two images, and possibly resize an image. In the following sections, we will explore some of the ways these operations can be performed when using Python.

Python image processing libraries

Out of the wealth of image processing libraries available for Python programmers, we chose to use three of the most prominent ones. These libraries will be briefly discussed in the following subsections.

The Pillow library

`Pillow` is a currently maintained fork of the original **Python Imaging Library** (**PIL**). It offers support for opening, manipulating, and saving image files of various formats. Since it allows us to handle image files, draw shapes, control their transparency, and manipulate pixels, we will use it as our main tool in creating the reconstructed image.

The home page of this library can be found here: `https://python-pillow.org/`.

A typical installation of `Pillow` uses the `pip` command, as follows:

```
pip install Pillow
```

The `Pillow` library uses the PIL namespace. If you have the original PIL library already installed, you will have to uninstall it first. More information can be found in the documentation, which is located at `https://pillow.readthedocs.io/en/stable/index.html`.

The scikit-image library

The `scikit-image` library, which was developed by the SciPy community, extends `scipy.image` and provides a collection of algorithms for image processing, including image I/O, filtering, color manipulation, and object detection. Here, we will only utilize its *metrics* module, which is used to compare two images.

The home page of this library can be found here: `https://scikit-image.org/`.

`scikit-image` comes preinstalled with several Python distributions, such as *Anaconda* and *winPython*. If you need to install it, use the `pip` utility, as follows:

```
pip install scikit-image
```

If you are running Anaconda or miniconda, use the following command instead:

```
conda install -c conda-forge scikit-image
```

More information can be found in the `scikit-image` documentation, which is located at `https://scikit-image.org/docs/stable/index.html`.

The opencv-python library

OpenCV is an elaborate library that provides numerous algorithms related to computer vision and machine learning. It supports a wide variety of programming languages and is available on different platforms. `opencv-python` is the Python API for this library. It combines the speed of the C++ API with the ease of use of the Python language. Here, we will mainly make use of this library to calculate the difference between two images since it allows us to represent an image as a numeric array.

The home page of `opencv-python` can be found here: `https://pypi.org/project/opencv-python/`.

The library consists of four different packages, all of which use the same namespace (cv2). Only one of these packages should be selected to be installed in a single environment. For our purposes, we can use the following command, which only installs the main modules:

```
pip install opencv-python
```

More information can be found in the OpenCV documentation, which is located at `https://docs.opencv.org/master/`.

Drawing images with polygons

To draw an image from scratch, we can use Pillow's `Image` and `ImageDraw` classes, as follows:

```
image = Image.new('RGB', (width, height))
draw = ImageDraw.Draw(image, 'RGBA')
```

`'RGB'` and `'RGBA'` are the values for the **mode** argument. The `'RGB'` value indicates three 8-bit values per pixel – one for each of the colors Red ('R'), Green ('G'), and Blue ('B'). The `'RGBA'` value adds a fourth 8-bit value, `"A"`, representing the alpha (opacity) level of the drawings to be added. The combination of an RGB base image and an RGBA drawing will allow us to draw polygons of varying degrees of transparency on top of a black background.

Now, we can add a polygon to the base image by using the `ImageDraw` class' `polygon` function, as shown in the following example. The following statement will draw a triangle on the image:

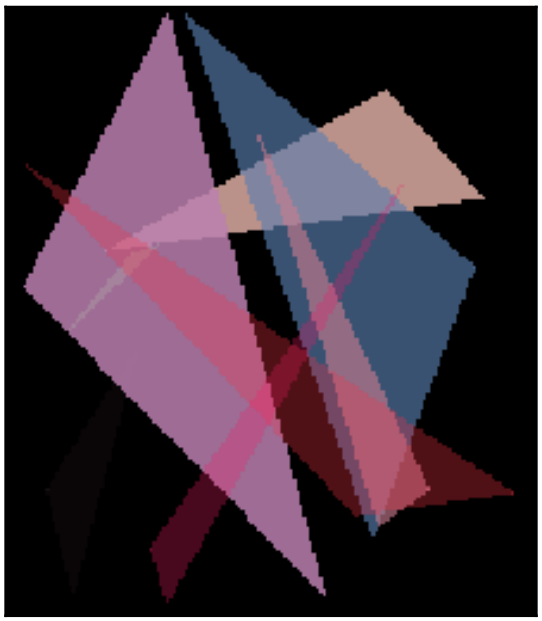
```
draw.polygon([(x1, y1), (x2, y2), (x3, y3)], (red, green, blue, alpha))
```

The following list explains the terms that were used in the preceding statement in more detail:

- The `(x1, y1)`, `(x2, y2)`, and `(x3, y3)` tuples represent the triangle's three vertices. Each tuple contains the x, y coordinates of the corresponding vertex within the image.
- `red`, `green`, and `blue` are integer values in the range of [0, 255], each representing the intensity of the corresponding color of the polygon.
- `alpha` is an integer value in the range of [0, 255], representing the opacity value of the polygon (a lower value means more transparency).

To draw a polygon with more vertices, we would need to add more (x_i, y_i) tuples to the list.

We can add more and more polygons this way, all drawn onto the same image, possibly overlapping each other, as shown in the following image:



A plot of overlapping polygons with varying colors and opacity values

Once we draw an image using polygons, we need to compare it to the reference image, as described in the next subsection.

Measuring the difference between images

Since we would like to construct an image that is as similar as possible to the original one, we need a way to evaluate the similarity or the difference between the two given images. Two possible methods are as follows:

- Pixel-based **Mean Squared Error** (**MSE**)
- **Structural Similarity** (**SSIM**)

Let's discuss both in detail.

Pixel-based Mean Squared Error

The most common way to evaluate the similarity between images is by conducting a pixel-by-pixel comparison. This requires, of course, that both images are of the same dimensions. The MSE metric can be calculated as follows:

1. Calculate the square of the difference between each pair of matching pixels from both images. Since each pixel in the drawing is represented using three separate values – red, green and blue – the difference for each pixel is calculated across these three dimensions.
2. Compute the sum of all these squares.
3. Divide the sum by the total number of pixels.

When both images are represented using the OpenCV (cv2) library, this calculation can be done in a very straightforward way, as follows:

```
MSE = np.sum((cv2Image1.astype("float") - cv2Image2.astype("float")) **
2)/float(numPixels)
```

When the two images are identical, the MSE value will be zero. Consequently, minimizing this metric can be used as the objective of our algorithm.

Structural Similarity (SSIM)

The SSIM index was created to be used to predict the image quality that's produced by a given compression algorithm by comparing the compressed image to the original one. Rather than calculating an absolute error value, which is done by the MSE method, for example, SSIM is perception-based and considers changes in structural information, as well as effects such as brightness and texture in the images.

The `metrics` module of the `scikit-image` library provides us with a function that calculates the structural similarity index between two images. When both images are represented using the OpenCV (cv2) library, this function can be used directly, as follows:

```
SSIM = structural_similarity(cv2Image1, cv2Image2)
```

The value returned is a float in the range of [-1, 1], representing the SSIM index between the two given images. A value of one indicates identical images.

By default, this function compares grayscale images. To compare color images, the optional `multichannel` argument should be set to true.

Using genetic algorithms to reconstruct images

As we discussed previously, our goal in this experiment is to use a familiar image as a reference and create a second image, as similar as possible to the reference, using a collection of overlapping polygons of varying colors and transparencies. Using the genetic algorithms approach, each candidate solution is a set of such polygons, and evaluating the solution is carried out by creating an image using these polygons and comparing it to the reference image. As usual, the first decision we need to make is how these solutions are represented. We will discuss this in the next subsection.

Solution representation and evaluation

As we mentioned previously, our solution consists of a set of polygons within the image boundaries. Each polygon has its own color and transparency. Drawing such a polygon using the `Pillow` library requires the following arguments:

- A list of tuples, $[(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)]$, representing the vertices of the polygon. Each tuple contains the x, y coordinates of the corresponding vertex within the image. Therefore, the values of the x coordinates are in the range [0, image width – 1], while the values of the y coordinates are in the range [0, image height – 1].
- Three integer values in the range of [0, 255], representing the red, green, and blue components of the polygon's color.
- An additional integer value in the range of [0, 255], representing the alpha – or opacity – value of the polygon.

This means that for each polygon in our collection, we will need $[2\times(\textit{polygon size}) + 4]$ parameters. A triangle, for example, will require 10 parameters, while a hexagon will require 16 parameters. Consequently, a collection of triangles will be represented using a list in the following format, where every 10 parameters represent a single triangle:

$$[x_{11}, y_{11}, x_{12}, y_{12}, x_{13}, y_{13}, r_1, g_1, b_1, alpha_1, x_{21}, y_{21}, x_{22}, y_{22}, x_{23}, y_{23}, r_2, g_2, b_2, alpha_2, ...]$$

To simplify this representation, we will use float numbers in the range of [0, 1] for each of the parameters. Before drawing the polygons, we will expand each parameter accordingly so that it fits within its required range – image width and height for the coordinates of the vertices and [0, 255] for the colors and opacity values.

Using this representation, a collection of 50 triangles will be represented as a list of 500 float values between 0 and 1, like so:

```
[0.1499488467959301, 0.3812631075049196, 0.0004394580562993053,
0.9988170920722447, 0.9975357316889601, 0.9997461395379549,
0.6338072268312986, 0.379170095245514, 0.29280945382368373,
0.20126488596803083,
...
0.4551462922205506, 0.9912529573649455, 0.7882252614083617,
0.01684396868069853, 0.2353587486989349, 0.003221988752732261,
0.9998952203500615, 0.48148512088979356, 0.11555604920908047,
0.08328550982740457]
```

Evaluating a given solution means dividing this long list into chunks representing individual polygons – in the case of triangles, the chunk will have a length of 10. Then, we need to create a new, blank image and draw the various polygons from the list on top of it, one by one. Finally, the difference between the resulting image and the original (reference) image needs to be calculated. As we saw in the previous section, there are two different methods we can employ to calculate the difference between the images – pixel-based MSE and the SSIM index. This (somewhat elaborate) score evaluation procedure is implemented by a Python class, which will be described in the next subsection.

Python problem representation

To encapsulate the image reconstruction challenge, we've created a Python class called `ImageTest`. This class is contained in the `image_test.py` file, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter11/image_test.py`.

The class is initialized with two parameters: the path of the file containing the reference image and the number of vertices of the polygons that are being used to construct the image. The class provides the following public methods:

- `polygonDataToImage()`: Accepts the list containing the polygon data we discussed in the previous subsection, divides this list into chunks representing individual polygons, and creates an image containing these polygons by drawing the polygons one by one onto a blank image.
- `getDifference()`: Accepts polygon data, creates an image containing these polygons, and calculates the difference between this image and the reference image using one of two methods – MSE or SSIM.

- `plotImages()`: Creates a side-by-side plot of the given image next to the reference image for visual comparison purposes.
- `saveImage()`: Accepts polygon data, creates an image containing these polygons, creates a side-by-side plot of this image next to the reference image, and saves the plot in a file.

During the run of the genetic algorithm, the `saveImage()` method will be called every 100 generations in order to save a side-by-side image comparison representing a snapshot of the reconstruction process. Calling this method will be carried out by a callback function, as described in the next subsection.

Genetic algorithm implementation

To reconstruct a given image with a set of semi-transparent overlapping polygons using a genetic algorithm, we've created a Python program called `01-reconstruct-with-polygons.py`, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter11/reconstruct-with-polygons.py`.

Since we are using a list of floats to represent a solution – the polygons' vertices, colors, and transparency values – this program is very similar to the function optimization programs we saw in `Chapter 6`, *Optimizing Continuous Functions*, such as the one we used for the `Eggholder` function's optimization.

The following steps describe the main parts of this program:

1. We start by setting the problem-related constant values. `POLYGON_SIZE` determines the number of vertices for each polygon, while `NUM_OF_POLYGONS` determines the total number of polygons that will be used to create the reconstructed image:

```
POLYGON_SIZE = 3
NUM_OF_POLYGONS = 100
```

2. We continue by creating an instance of the `ImageTest` class, which will allow us to create images from polygons and compare these images to the reference image, as well as to save snapshots of our progress:

```
imageTest = image_test.ImageTest("images/Mona_Lisa_Head.png",
POLYGON_SIZE)
```

3. Next, we set the upper and lower boundaries for the float values we will be searching for. As we mentioned previously, we will use float values for all our parameters and set them all to the same range, between 0.0 and 1.0, for convenience. When evaluating a solution, the values will be expanded to their actual range and converted into integers when needed:

```
BOUNDS_LOW, BOUNDS_HIGH = 0.0, 1.0
```

4. Since our goal is to minimize the difference between the images – the reference image and the one we are creating using polygons – we define a single objective, minimizing fitness strategy:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

5. Now, we need to create a helper function that will create random real numbers that are uniformly distributed within a given range. This function assumes that the range is the same for every dimension, as is the case in our solution:

```
def randomFloat(low, up):
    return [random.uniform(l, u) for l, u in zip([low] *
NUM_OF_PARAMS, [up] * NUM_OF_PARAMS)]
```

6. Next, we use the preceding function to create an operator that randomly returns a list of floats, all in the desired range of [0, 1]:

```
toolbox.register("attrFloat", randomFloat, BOUNDS_LOW,
BOUNDS_HIGH)
```

7. This is followed by defining an operator that fills up an individual instance using the preceding operator:

```
toolbox.register("individualCreator",
                 tools.initIterate,
                 creator.Individual,
                 toolbox.attrFloat)
```

8. Then, we instruct the genetic algorithm to use the `getDifference()` method of the `ImageTest` instance for fitness evaluation.

As a reminder, the `getDifference()` method, which we described in the previous subsection, accepts the individual representing a list of polygons, creates an image containing these polygons, and calculates the difference between this image and the reference image using one of two methods – MSE or SSIM. For starters, we will use the MSE method to calculate the difference:

```
def getDiff(individual):
    return imageTest.getDifference(individual, "MSE"),

toolbox.register("evaluate", getDiff)
```

9. It's time to choose our genetic operators. For the selection operator, we will use tournament selection with a tournament size of 2. As we saw in `Chapter 4`, *Combinatorial Optimization*, this selection scheme works well in conjunction with the elitist approach that we plan to utilize here as well:

```
toolbox.register("select", tools.selTournament, tournsize=2)
```

10. As for the crossover operator (aliased with `mate`) and the mutation operator (`mutate`), since our solution representation is a list of floats bounded to a range, we will use the specialized continuous bounded operators provided by the DEAP framework – `cxSimulatedBinaryBounded` and `mutPolynomialBounded`, respectively – which we first saw in `Chapter 6`, *Optimizing Continuous Functions*:

```
toolbox.register("mate",
                 tools.cxSimulatedBinaryBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR)

toolbox.register("mutate",
                 tools.mutPolynomialBounded,
                 low=BOUNDS_LOW,
                 up=BOUNDS_HIGH,
                 eta=CROWDING_FACTOR,
                 indpb=1.0/NUM_OF_PARAMS)
```

11. As we have done multiple times before, we will use the elitist approach, where the **hall of fame** (**HOF**) members – the current best individuals – are always passed untouched to the next generation. However, this time, we're going to add a new feature to this implementation – a callback function that will be used to save the image every 100 generations. We will discuss this callback in more detail in the next subsection:

```
population, logbook =
elitism_callback.eaSimpleWithElitismAndCallback(population,
                                                toolbox,
                                                cxpb=P_CROSSOVER,
                                                mutpb=P_MUTATION,
                                                ngen=MAX_GENERATIONS,
                                                callback=saveImage,
                                                stats=stats,
                                                halloffame=hof,
                                                verbose=True)
```

12. At the end of the run, we print the best solution and plot the image it creates next to the reference image:

```
best = hof.items[0]
print("Best Solution = ", best)
print("Best Score = ", best.fitness.values[0])

imageTest.plotImages(imageTest.polygonDataToImage(best))
```

Before we look at the results, let's discuss the implementation of the callback function.

Adding a callback to the genetic run

To be able to save the best current image every 100 generations, we need to introduce a modification to the main genetic loop. As you may recall, toward the end of `Chapter 4`, *Combinatorial Optimization*, we already made one modification to the main loop that allowed us to introduce the elitist approach. To be able to introduce that change, we created the `eaSimpleWithElitism()` method, which is contained in a file called `elitism.py`. This method was a modified version of the `deap` framework's `eaSimple()` method, which is contained in the `algorithms.py` file. We modified the original method by adding the elitism functionality, which takes the members of the HOF – the current best individuals – and passes them untouched to the next generation at every iteration of the loop. Now, for the purpose of implementing a callback, we will introduce another small modification and change the name of the method to `eaSimpleWithElitismAndCallback()`. We will also rename the file containing it to `elitism_callback.py`.

There are two parts to this modification, as follows:

1. The first part of the modification consists of adding an argument called `callback` to the method:

```
def eaSimpleWithElitismAndCallback(population, toolbox, cxpb,
                                   mutpb, ngen, callback=None,
                                   stats=None, halloffame=None,
                                   verbose=__debug__):
```

   This argument represents an external function that will be called after each iteration.

2. The other part is within the main loop. Here, the callback function is called after the new generation has been created and evaluated. The current generation number and the current best individual are passed to the callback as arguments:

```
if callback:
    callback(gen, halloffame.items[0])
```

Being able to define a callback function that will be called after each generation may prove useful in various situations. To take advantage of it here, we'll define the `saveImage()` function back in our `01-reconstruct-with-polygons.py` program. We will use it to save a side-by-side image of the current best image and the reference image, every 100 generations, as follows:

1. We use the modulus (`%`) operator to activate the method only once every 100 generations:

```
if gen % 100 == 0:
```

2. If this is one of these generations, we create a folder for the images if one does not exist. The images are saved in a folder that's named using the polygon size and the number of polygons – for example, *run-3-100* or *run-6-50*, under the `images/results/` path:

```
        folder = "images/results/run-{}-
{}".format(POLYGON_SIZE, NUM_OF_POLYGONS)
        if not os.path.exists(folder):
            os.makedirs(folder)
```

3. Next, we save the image of the best current individual in that folder. The name of the image contains the number of generations that have been passed; for example, `after-300-generations.png`:

```
imageTest.saveImage(polygonData,
                    "{}/after-{}-gen.png".format(folder, gen),
                    "After {} Generations".format(gen))
```

We are finally ready to run this algorithm with reference images and check out the results.

Image reconstruction results

To test our program, we will use the following image, which is part of the famous Mona Lisa portrait by Leonardo da Vinci, considered to be the most well-known painting in the world:

Head crop of the Mona Lisa painting
Source: https://commons.wikimedia.org/wiki/File:Mona_Lisa_headcrop.jpg. Artist: Leonardo da Vinci.
Licensed under Creative Commons CC0 1.0: https://creativecommons.org/publicdomain/zero/1.0/

The polygons that will be used to create the image will be 100 triangles:

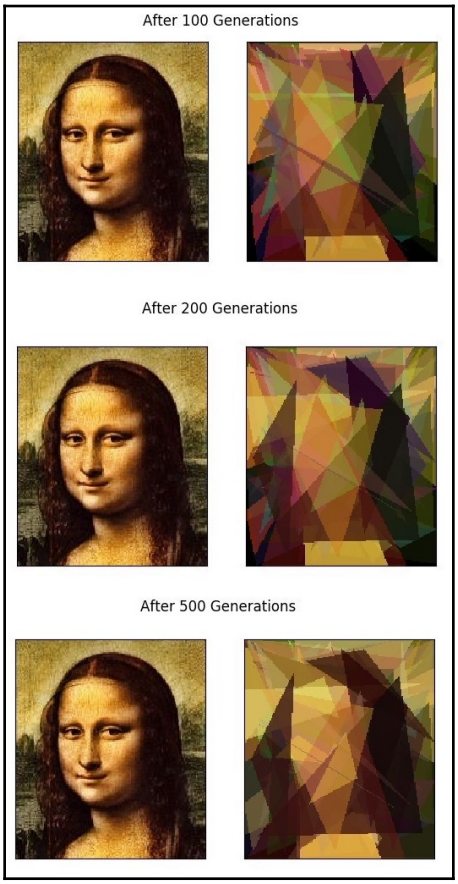
```
POLYGON_SIZE = 3
NUM_OF_POLYGONS = 100
```

We will run the algorithm for 5,000 generations with a population size of 200. As we discussed previously, a side-by-side image comparison is saved every 100 generations. At the end of the run, we can go back and examine the saved images so that we can follow the evolution of the reconstructed image.

Before we continue running the program, please be advised that, due to the length of the polygon data and the complexity of the image processing operations, the runtimes for our genetic image reconstruction experiments are much longer than the other programs we have tested so far in this book, and are typically several hours per experiment. The results of these experiments will be described in the following subsections.

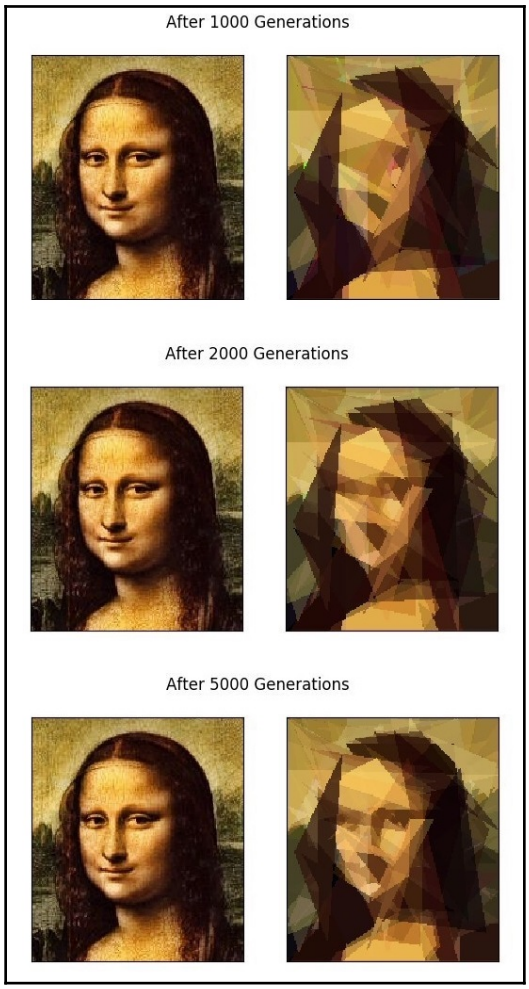
Using pixel-based Mean Squared Error

We will start by using the pixel-based MSE metric to measure the difference between the reference image and the reconstructed image. The following are several milestones from the resulting side-by-side saved images:



Milestone results of Mona Lisa reconstruction using pixel-based Mean Squared Error – part 1

The following image shows the results for the following generations:



Milestone results of Mona Lisa reconstruction using pixel-based Mean Squared Error – part 2

The end result bears a close resemblance to the original image, although it contains sharp corners and straight lines, as would be expected from a polygon-based image. Squinting while looking at the images helps blur these characteristics of the reconstructed image. A similar effect can be achieved programmatically by using the `GaussianBlur` filter that's offered by the OpenCV library, as follows:

```
origImage = cv2.imread('path/to/image')
blurredImage = cv2.GaussianBlur(origImage, (45, 45), cv2.BORDER_DEFAULT)
```

A blurred version of the last side-by-side images is as follows:

Blurred versions of the original and pixel-based reconstructed image, side by side

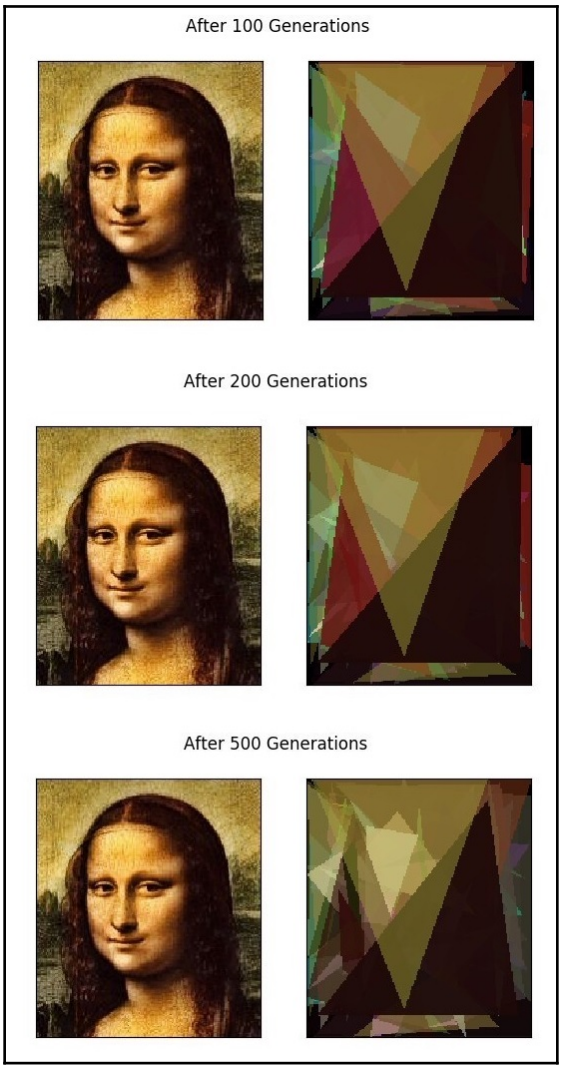
Next, we will try the other method of measuring the difference between the reference image and the reconstructed image – the SSIM index.

Using the SSIM index

Now, we'll repeat the experiment, but this time using the SSIM metric to measure the difference between the reference image and the reconstructed image. To make this happen, we'll modify the definition of `getDiff()`, as follows:

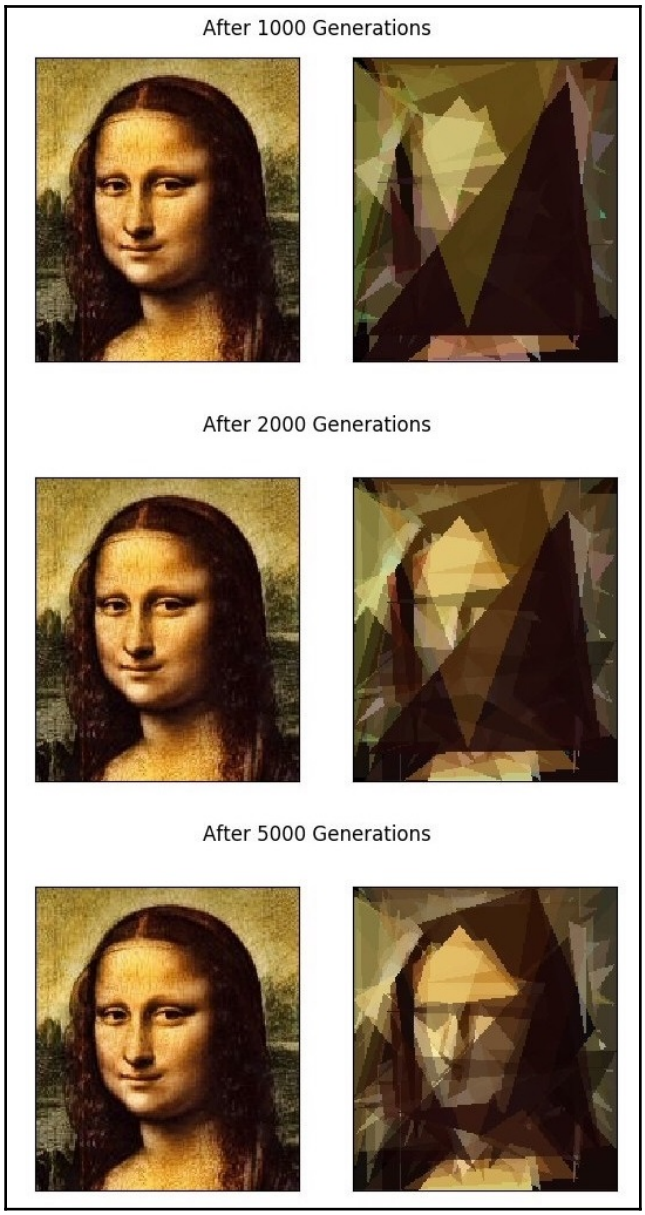
```
def getDiff(individual):
    return imageTest.getDifference(individual, "SSIM"),
```

This experiment produced the following side-by-side milestones of the saved images:



Milestone results of Mona Lisa reconstruction using the SSIM index – part 1

The following image shows the results for the following generations:



Milestone results of Mona Lisa reconstruction using the SSIM index – part 2

The result seems interesting – it captured the structure of the image but in a coarser way than the MSE-driven results. The colors seem somewhat off as well since the SSIM is more focused on structure and texture. The following is a blurred version of the final side-by-side images:

Blurred versions of the original and SSIM-based reconstructed image, side by side

It may be interesting to combine the MSE and SSIM difference-measuring methods—you are encouraged to experiment with that on your own. Other suggestions for experimentation are described in the next subsection.

Other experiments

There are lots of variations that you can try out. One simple and obvious variation is increasing the number of vertices that the polygon has. By doing this, we expect to get a more accurate result since the shapes that are being used are more versatile. Let's change the number of vertices to six, as follows:

```
POLYGON_SIZE = 6
```

Repeating this experiment, this time with 10,000 generations, produces the following final side-by-side image comparison:

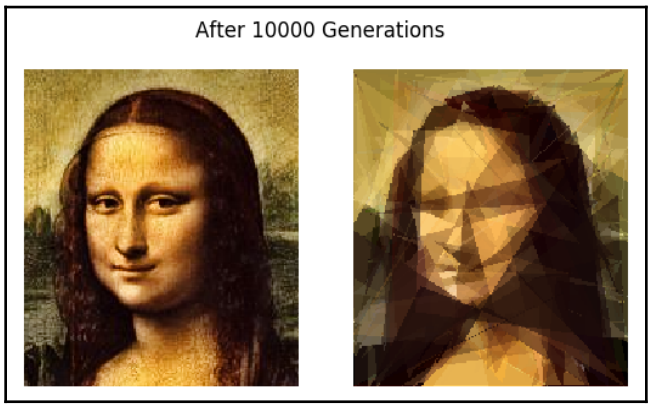


Image reconstruction results using pixel-based Mean Squared Error and a polygon size of six

As we would expect, this reconstructed image looks somewhat more refined than the triangle-based one. The following is a blurred version of this image:

Blurred versions of the original and pixel-based reconstructed image, side by side, using a polygon size of six

Besides changing the number of vertices, there are many other possibilities and combinations to experiment with, such as the following:

- Changing the total number of shapes
- Changing the population size and the number of generations
- Using non-polygon shapes (circles, ellipse) or regular shapes (squares, rectangles)

- Using different types of reference images (paintings, drawings, photos, logos)
- Using grayscale images instead of color ones

Have fun creating and working on your own experiments!

Summary

In this chapter, you were introduced to the popular concept of reconstructing existing images using a set of overlapping, semi-transparent polygons. Then, you got acquainted with several image processing libraries in Python and found out how an image can be programmatically drawn from scratch using polygons, as well as how to calculate the difference between two images. Next, we developed a genetic algorithm-based program that reconstructed a segment of a famous painting using polygons. Numerous possibilities for further experimentation were mentioned as well.

In the next chapter, we will describe and demonstrate several problem-solving techniques related to genetic algorithms, as well as other biologically inspired computational algorithms.

Further reading

For more information about the topics that were covered in this chapter, please refer to the following resources:

- Grow Your Own Picture: `https://chriscummins.cc/s/genetics/`
- Genetic Programming: Evolution of Mona Lisa: `https://rogerjohansson.blog/2008/12/07/genetic-programming-evolution-of-mona-lisa/`
- *Hands-On Image Processing with Python*, Sandipan Dey, November 30, 2018
- Wang, Z., Bovik, A. C., Sheikh, H. R., & Simoncelli, E. P. (2004), *Image quality assessment: From error visibility to structural similarity*. IEEE Transactions on Image Processing, 13, 600-612: `https://ece.uwaterloo.ca/~z70wang/publications/ssim.pdf`

12
Other Evolutionary and Bio-Inspired Computation Techniques

In this chapter, you will broaden your horizons and discover several new problem-solving and optimization techniques related to genetic algorithms. Two different techniques of this extended family – genetic programming and particle swarm optimization – will be then demonstrated by implementing problem-solving Python programs. Finally, we will provide a brief overview of a number of other related computation paradigms.

This chapter will cover the following topics:

- The evolutionary computation family of algorithms
- Understanding the concepts of genetic programming and how they differ from genetic algorithms
- Using genetic programming to solve the even parity check problem
- Using particle swarm optimization to optimize Himmelblau's function
- Understanding the principles behind several other evolutionary and biologically inspired techniques

We will start this chapter by unveiling the extended family of evolutionary computation and discussing the main characteristics shared by its members.

Technical requirements

In this chapter, we will be using Python 3 with the following supporting libraries:

- `deap`
- `numpy`
- `networkx`

The programs that will be used in this chapter can be found in this book's GitHub repository at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/tree/master/Chapter12`.

Check out the following video to see the Code in Action:
`http://bit.ly/2Gx4KsL`

Evolutionary computation and bio-inspired computing

Throughout this book, we have covered the problem-solving technique known as genetic algorithms and applied it to numerous types of problems, including combinatorial optimization, constraint satisfaction, and continuous function optimization, as well as to machine learning and artificial intelligence. However, as we mentioned in `Chapter 1`, *An Introduction to Genetic Algorithms*, genetic algorithms are just one branch within a larger family of algorithms called *evolutionary computation*. This family consists of various related problem-solving and optimization techniques, all of which draw inspiration from Charles Darwin's theory of natural evolution.

The main characteristics that are shared by these techniques are as follows:

- The starting point is an initial set (population) of candidate solutions.
- The candidate solutions (individuals) are iteratively updated to create new generations.
- The creation of a new generation involves the removal of less successful individuals (selection), as well as the introduction of small random changes (mutations) to some individuals. Other operators, such as interaction with other individuals (crossover), may also be applied.
- As a result, as generations go by, the **fitness** of the population increases; in other words, the candidate solutions grow better at solving the problem at hand.

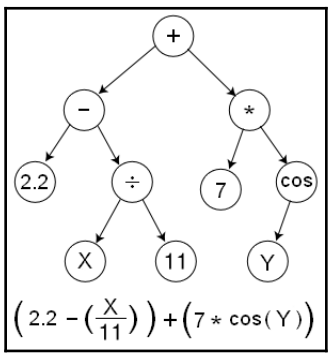
More broadly, since evolutionary computation techniques are based on various biological systems or behaviors, some of them overlap with the algorithm family known as *bio-inspired computing*.

In the following sections, we will cover some of the most frequently used members of evolutionary computation and bio-inspired computing – some of them in greater detail, while the others will only be mentioned briefly. We will start with a detailed account of a fascinating technique that allows us to evolve actual computer programs – *genetic programming*.

Genetic programming

Genetic programming (**GP**) is a special form of genetic algorithm – the technique we have been applying throughout this entire book. In this special case, the candidate solutions – or individuals – that we are evolving with the aim of finding the best one for our purpose are actual computer programs, hence the name. In other words, when we apply GP, we evolve computer programs with the goal of finding a program that will excel at performing a particular task.

As you may recall, genetic algorithms use a representation of the candidate solutions, often referred to as a chromosome. This representation is the subject of genetic operators, namely selection, crossover, and mutation. Applying these operators to the current generation results in a new generation of solutions that is expected to produce better results than its predecessor. In most of the problems we have looked at so far, this representation was a list (or an array) of values of a certain type, such as integers, Booleans, or floats. To represent a program, however, we typically use a tree structure, as shown in the following diagram:

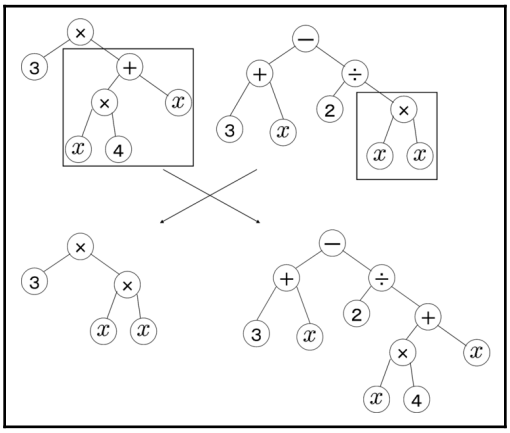


Tree structure representation of a simple program
Source: https://commons.wikimedia.org/wiki/File:Genetic_Program_Tree.png
Image by Baxelrod. Released to the public domain

The tree structure depicted in the preceding diagram represents the calculation shown underneath the tree. This calculation is equivalent to a short program (or a function) that accepts two arguments, X and Y, and returns a certain output based on their values. To create and evolve such tree structures, we need to define two different sets:

- **Terminals**, or the leaves of the tree. These are arguments and the constant values that can be used in the tree. In our example, X and Y are arguments, while 2.2, 11, and 7 are constants. Constants can also be generated randomly, within a certain range, when a tree is created.
- **Primitives**, or the internal nodes of the tree. These are functions (or operators) that accept one or more arguments and generate a single output value. In our example, +, -, *, and ÷ are primitives that accept two arguments, while *cos* is a primitive that accepts a single argument.

In `Chapter 2`, *Understanding the Key Components of Genetic Algorithms*, we demonstrated how the genetic operator of *single-point crossover* operates on binary-valued lists. The crossover operation created two offspring from two parents by cutting out a part of each parent and swapping the detached parts between the parents. Similarly, a crossover operator for the tree representation may detach a subtree (a branch or a group of branches) from each parent and swap the detached branches between the parents to create offspring trees, as demonstrated in the following diagram:



Crossover operation between two tree structures representing programs
Source: https://commons.wikimedia.org/wiki/File:GP_crossover.png
Image by U-ichi, Licensed under Creative Commons CC SA 1.0: https://creativecommons.org/licenses/sa/1.0/

In this example, the two parents on the top row have subtrees that have been swapped between them to create the two offspring in the second row. The swapped subtrees are marked by the rectangles surrounding them.

Along the same lines, the mutation operator, which intends to introduce random changes to a single individual, can be implemented by picking a subtree within the candidate solution and replacing it with a randomly generated one.

The `deap` library, which we have been using throughout this book, provides inherent support for genetic programming. In the next section, we will implement a simple genetic programming example using this library.

Genetic programming example – even parity check

For our example, we will use genetic programming to create a program that implements an even parity check. In this task, the possible values of the inputs are either 0 or 1. The output value should be 1 if the number of the inputs with the value 1 is odd, thereby producing a total even number of 1 values; otherwise, the output value should be 0. The following table lists the various possible combinations of input values for the case of three inputs, along with the matching even parity output values:

in_0	in_1	in_2	Even Parity
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

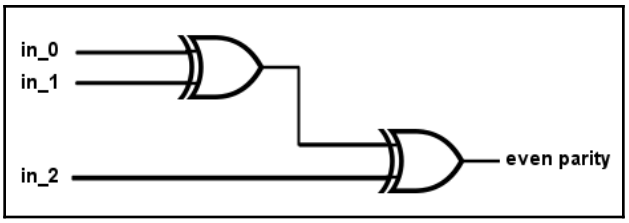
This kind of table is often referred to as the *truth table* of the operation at hand. As evident from this truth table, one reason that the parity check is often used as a benchmark is that any single change in the input values will result in a change to the output value.

The parity check can also be represented using logic gates, such as AND, OR, NOT, and **XOR** (**Exclusive OR**). While the NOT gate accepts a single input and inverts it, each of the three other gate types accepts two inputs. For the respective output to be 1, the AND gate requires both inputs to be 1, the OR gate requires at least of them to be 1, and the XOR gate requires that **exactly** one of them is 1, as shown in the following table:

in_0	in_1	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1

1	0	0	1	1
1	1	1	1	0

There are many possible ways to implement the three-input parity check using logic gates. The simplest way to do this is by using two XOR gates, as shown in the following diagram:



Three-input even parity check implemented using two XOR gates

In the next subsection, we will use genetic programming to create a small program that implements the even parity check using the logic operations of AND, OR, NOT, and XOR.

Genetic programming implementation

To evolve a program that implements the even parity check logic, we've created a genetic programming-based Python program called `01-gp-even-parity.py`, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter12/01-gp-even-parity.py`.

Since genetic programming is a special case of genetic algorithms, much of this program will look familiar to you if you have gone over the programs we presented in earlier chapters of this book.

The following steps describe the main parts of this program:

1. We start by setting the problem-related constant values. `NUM_INPUTS` determines the number of inputs for the even parity checker. We will use the value of 3 for simplicity; however, larger values can be set as well. The `NUM_COMBINATIONS` constant represents the number of possible combinations of values for the inputs, which is analogous to the number of rows in the truth table we saw earlier:

```
NUM_INPUTS = 3
NUM_COMBINATIONS = 2 ** NUM_INPUTS
```

2. This is followed by the familiar genetic algorithm constants we have seen numerous times before:

```
POPULATION_SIZE = 60
P_CROSSOVER = 0.9
P_MUTATION = 0.5
MAX_GENERATIONS = 20
HALL_OF_FAME_SIZE = 10
```

3. However, genetic programming requires several additional constants that refer to the tree representation of the candidate solutions. These are defined in the following code. We will see how they are used as we examine the rest of the program:

```
MIN_TREE_HEIGHT = 3
MAX_TREE_HEIGHT = 5
MUT_MIN_TREE_HEIGHT = 0
MUT_MAX_TREE_HEIGHT = 2
LIMIT_TREE_HEIGHT = 17
```

4. Next, we calculate the truth table of the even parity check so that we can use it as a reference when we need to check the accuracy of a given candidate solution. The `parityIn` matrix represents the input columns of the truth table, while the `parityOut` vector represents the output column. The Python `itertools.product()` function is an elegant replacement of nested `for` loops that would be otherwise required to iterate over all the combinations of input values:

```
parityIn = list(itertools.product([0, 1], repeat=NUM_INPUTS))
parityOut = []
for row in parityIn:
    parityOut.append(sum(row) % 2)
```

5. It is time to create the set of primitives, that is, the operators that will be used in our evolved programs. The first declaration creates a set using the following three arguments:
    - The name of the program to be generated using the primitives from the set (here, we called it `main`)
    - The number of inputs to the program
    - The prefix to be used when naming the inputs (optional)

   These three arguments are used to create the following primitive set:

```
primitiveSet = gp.PrimitiveSet("main", NUM_INPUTS, "in_")
```

6. Now, we fill the primitive set with the various functions (or operators) that will be used as the building blocks of the program. For each operator, we use a reference to the function we want to use and the number of arguments it expects. Although we could define our own functions for this purpose, in this case, we're making use of the existing Python `operator` module, which contains numerous useful functions, including the logical operators we need:

```
primitiveSet.addPrimitive(operator.and_, 2)
primitiveSet.addPrimitive(operator.or_, 2)
primitiveSet.addPrimitive(operator.xor, 2)
primitiveSet.addPrimitive(operator.not_, 1)
```

7. The following definitions set the terminal values to be used. As we mentioned earlier, these are constants that can be used as input values to the tree. In our case, it makes sense to use the values 0 and 1:

```
primitiveSet.addTerminal(1)
primitiveSet.addTerminal(0)
```

8. Since our goal is to create a program that implements the truth table of the even parity check, we will attempt to minimize the difference between the program output and the known output values. For this purpose, we will define a single objective, minimizing fitness strategy:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

9. Now, we will create the `Individual` class, based on the `PrimitiveTree` class provided by the `deap` library:

```
creator.create("Individual", gp.PrimitiveTree,
fitness=creator.FitnessMin)
```

10. To help us construct an individual in the population, we will create a helper function that will generate random trees using the primitive set we defined earlier. Here, we're making use of the `genFull()` function offered by `deap` and providing it with the primitive set, as well as with the values for defining the min and max height of the generated trees:

```
toolbox.register("expr", gp.genFull, pset=primitiveSet,
                 min_=MIN_TREE_HEIGHT, max_=MAX_TREE_HEIGHT)
```

11. This is followed by defining two operators, the first of which creates an individual instance using the preceding helper operator. The other generates a list of such individuals:

```
toolbox.register("individualCreator", tools.initIterate,
creator.Individual, toolbox.expr)
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.individualCreator)
```

12. Next, we create an operator to compile the primitive tree into Python code using the `compile()` function offered by `deap`. Consequently, we use this compile operator in a function we will create, called `parityError()`. This function counts the number of rows in the truth table for which the result of the calculation differs from the expected one:

```
toolbox.register("compile", gp.compile, pset=primitiveSet)

def parityError(individual):
    func = toolbox.compile(expr=individual)
    return sum(func(*pIn) != pOut for pIn, pOut in
zip(parityIn, parityOut))
```

13. Then, we instruct the genetic programming algorithm to use the `getCost()` function for fitness evaluation. This function returns the parity error we just saw in the tuple form required by the underlying evolutionary algorithm:

```
def getCost(individual):
    return parityError(individual),

toolbox.register("evaluate", getCost)
```

14. It's time to choose our genetic operators, starting with the selection operator (aliased with `select`). For genetic programming, this operator is typically the same tournament selection we have been using throughout this book. Here, we're using it with a tournament size of `2`:

```
toolbox.register("select", tools.selTournament, tournsize=2)
```

15. As for the crossover operator (aliased with `mate`), we use the specialized genetic programming `cxOnePoint()` operator that's provided by `deap`. Since the evolving programs are represented by trees, this operator takes two parent trees and exchanges sections of them to create two valid offspring trees:

```
toolbox.register("mate", gp.cxOnePoint)
```

16. Next is the mutation operator, which introduces random changes to an existing tree. The mutation is defined in two stages. First, we specify a helper operator that utilizes the specialized genetic programming `genGrow()` function, provided by `deap`. This operator creates a subtree within the limits defined by the two constants. Then, we define the mutation operator itself (aliased with `mutate`). This operator utilizes DEAP's `mutUniform()` function, which randomly replaces a subtree in a given tree with a random one that was generated using the helper operator:

```
toolbox.register("expr_mut", gp.genGrow,
min_=MUT_MIN_TREE_HEIGHT, max_=MUT_MAX_TREE_HEIGHT)
toolbox.register("mutate", gp.mutUniform,
expr=toolbox.expr_mut, pset=primitiveSet)
```

17. To prevent individuals in the population from growing into overly large trees, potentially containing an excessive number of primitives, we need to introduce **bloat control** measures. This is done using DEAP's `staticLimit()` function, which imposes a tree height restriction on the results of the crossover and mutation operations:

```
toolbox.decorate("mate",
gp.staticLimit(key=operator.attrgetter("height"),
max_value=LIMIT_TREE_HEIGHT))
toolbox.decorate("mutate",
gp.staticLimit(key=operator.attrgetter("height"),
max_value=LIMIT_TREE_HEIGHT))
```

18. The program's main loop is very similar to the ones we saw in earlier chapters. After creating the initial population, defining the statistics measurements, and creating the HOF object, we call the evolutionary algorithm. Like we've done multiple times before, we apply the elitist approach, where the HOF members – the current best individuals – are always passed untouched to the next generation:

```
population, logbook = elitism.eaSimpleWithElitism(population,
                                                   toolbox,
                                                   cxpb=P_CROSSOVER,
                                                   mutpb=P_MUTATION,
                                                   ngen=MAX_GENERATIONS,
                                                   stats=stats,
                                                   halloffame=hof,
                                                   verbose=True)
```

19. At the end of the run, we print the best solution, as well as the height of the tree that's being used to represent it and its length; that is, the total number of operators contained in the tree:

```
best = hof.items[0]
print("-- Best Individual = ", best)
print("-- length={}, height={}".format(len(best), best.height))
print("-- Best Fitness = ", best.fitness.values[0])
```

20. The last thing we need to do is plot a graphic illustration of the tree representing the best solution. To that end, we utilize the graph and networks library **networkx** (**nx**), which we introduced in `Chapter 5`, *Constraint Satisfaction*. We start by calling the `graph()` function provided by `deap`, which breaks down the individual tree into the nodes, edges, and labels that are required for the graph, and then create the graph using the appropriate networkx functions:

```
nodes, edges, labels = gp.graph(best)
g = nx.Graph()
g.add_nodes_from(nodes)
g.add_edges_from(edges)
pos = nx.spring_layout(g)
```

21. Then, we draw the nodes, edges, and labels. Since the layout of this graph is not a classic hierarchical tree, we distinguish the top node by coloring it red and enlarging it:

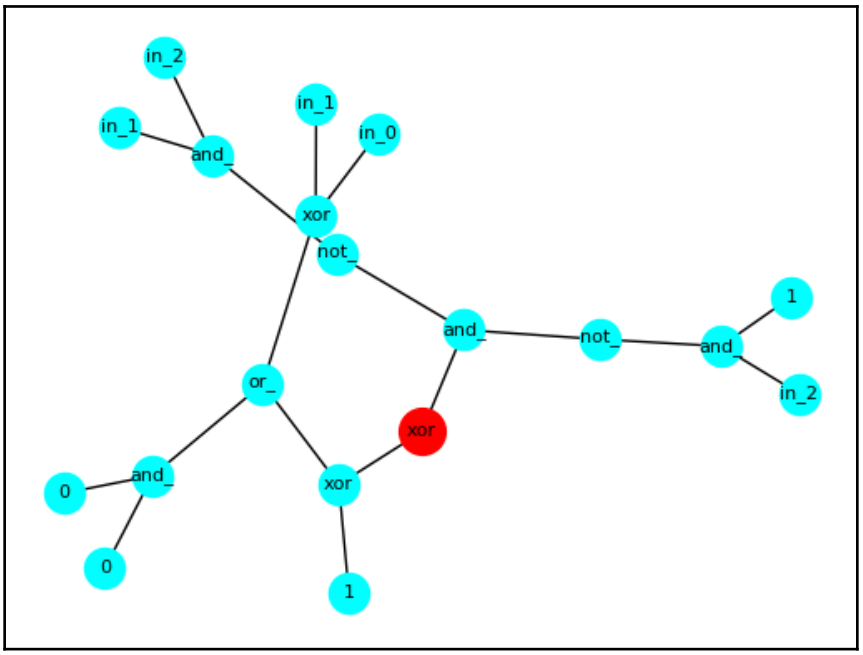
```
nx.draw_networkx_nodes(g, pos, node_color='cyan')
nx.draw_networkx_nodes(g, pos, nodelist=[0], node_color='red',
node_size=400)

nx.draw_networkx_edges(g, pos)
nx.draw_networkx_labels(g, pos, **{"labels": labels,
"font_size": 8})
```

By running this program, we get the following output:

```
gen nevals min avg
0 60 2 3.91667
1 50 1 3.75
2 47 1 3.45
...
5 47 0 3.15
...
20 48 0 1.68333

-- Best Individual = xor(and_(not_(and_(in_1, in_2)), not_(and_(1, in_2))),
xor(or_(xor(in_1, in_0), and_(0, 0)), 1))
```

```
-- length=19, height=4
-- Best Fitness = 0.0
```

Since this is a simple problem, the fitness has quickly reached the minimum value of 0, which means we were able to find a solution that correctly reproduces the even parity check truth table. However, the resulting expression, which consists of 19 elements and four levels in the hierarchy, seems overly complex. This is illustrated by the following plot that was produced by the program:



Plot representing the parity check solution that was found by the initial program

As we mentioned previously, the red node in the graph represents the top of the program's tree, which maps to the first XOR operation in the expression.

The reason for the relatively complex graph is that there is no advantage of using simpler expressions. As long as they fall within the imposed limitation of tree height, the expressions that are evaluated incur no penalty for complexity. In the next subsection, we will attempt to change this situation by introducing a small modification to the program in the hope of achieving the same outcome – the implementation of the even parity check – but with a simpler solution.

Simplifying the solution

In the implementation we have just seen, there were measures in place to restrict the size of the trees that represent the candidate solutions. However, the best solution we found seems overly complex. One way to pressure the algorithm into producing simpler results is to impose a small cost penalty for complexity. The penalty should be small enough, though, to refrain from favoring simpler solutions that fail to solve the problem. It should rather serve as a **tie-breaker** between two good solutions, so the simpler of the two will be preferred. This approach has been implemented in the Python program `02-gp-even-parity-reduced.py`, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter12/02-gp-even-parity-reduced.py`.

This program is nearly identical to the previous one, except for a couple of small changes:

1. The main change was introduced to the cost function, which the algorithm seeks to minimize. To the original calculated error, a small penalty measure was added that depends on the height of the tree:

```
def getCost(individual):
    return parityError(individual) + individual.height / 100,
```

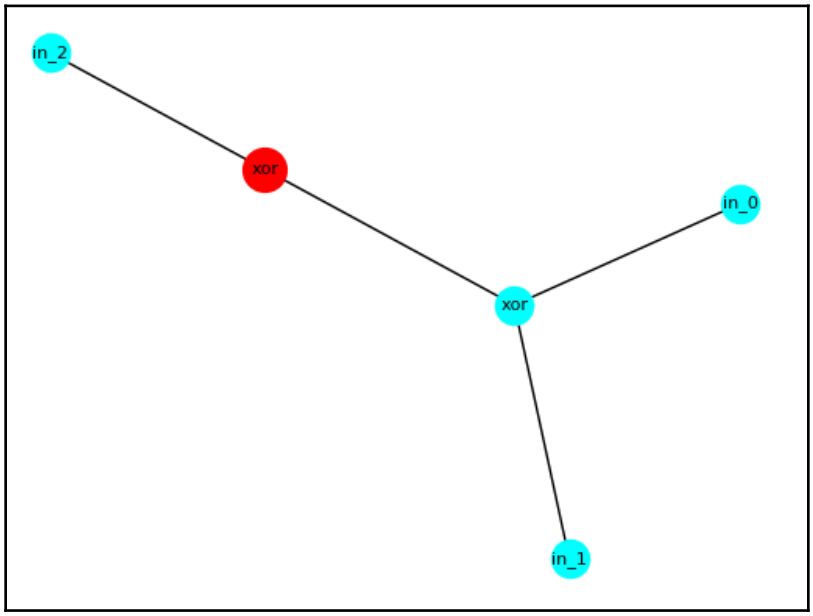
2. The only other change was at the end of the run, after printing the best solution that was found. Here, we added a printout of the actual parity error that was obtained, without the penalty that's present in the fitness:

```
print("-- Best Parity Error = ", parityError(best))
```

By running this modified version, we get the following output:

```
gen nevals min avg
0 60 2.03 3.9565
1 50 2.03 3.7885
...
5 47 0.04 3.45233
...
10 48 0.03 3.0145
...
15 49 0.02 2.57983
...
20 45 0.02 2.88533
-- Best Individual = xor(xor(in_0, in_1), in_2)
-- length=5, height=2
-- Best Fitness = 0.02
-- Best Parity Error = 0
```

From the preceding output, we can tell that, after five generations, the algorithm was able to find a solution that correctly reproduces the even parity check truth table since the fitness value at that point was nearly 0. However, as the algorithm kept running, the tree height was reduced from four (penalty of 0.04) to two (penalty of 0.02). As a result, the best solution is very simple and consists of only five elements – the three inputs and two XOR operators. In fact, the solution we found represents the simplest known solution that we saw earlier, which consists of two XOR gates. This is illustrated by the following plot, which was produced by the program:



Plot representing the parity check solution that was found by the modified program

In the next section, we will examine another biologically inspired, population-based algorithm. However, this algorithm deviates from using the familiar genetic operators of selection, crossover, and mutation, and instead utilizes a different set of rules to modify the population at each generation – welcome to the world of swarm behavior.

Particle swarm optimization

Particle swarm optimization (**PSO**) draws its inspiration from natural groupings of individual organisms, such as flocks of birds or schools of fish, generally referred to as swarms. The organisms interact within the swarm without central supervision, working together toward a common goal. This observed behavior gave rise to a computational method that can solve or optimize a given problem by using a group of candidate solutions represented by particles analogous to organisms in a swarm. The particles move in the search space, looking for the best solution, and their movement is governed by simple rules that involve their position and velocity (directional speed).

The PSO algorithm is iterative, and in each iteration, every particle's position gets evaluated, and its best location so far, as well as the best location within the entire group of particles, are updated if necessary. Then, each particle's velocity is updated according to the following information:

- The particle's current speed and direction of movement – representing inertia
- The particle's best position found so far (local best) – representing cognitive force
- The entire group's best position found so far (global best) – representing social force

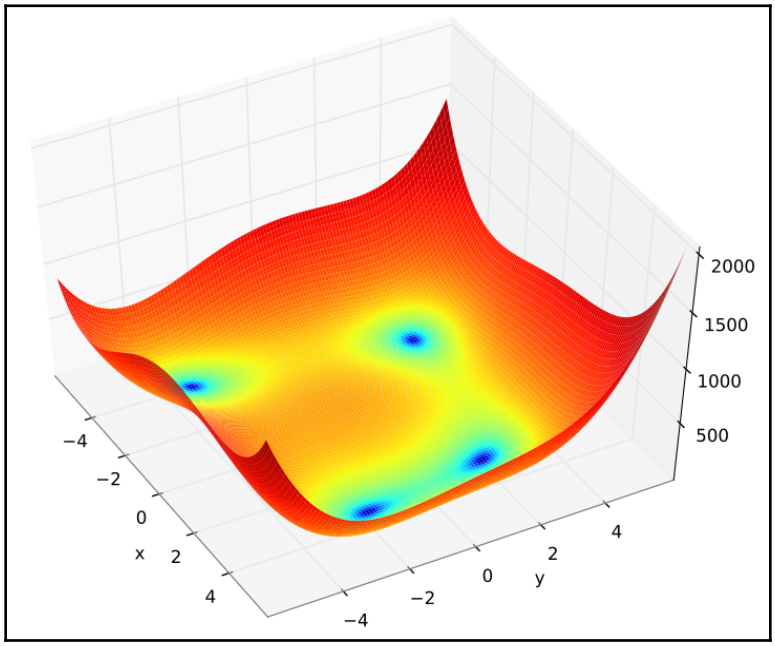
This is followed by an update to the particle's position, based on the newly calculated velocity.

The iterative process continues until some stopping condition, such as the iterations limit, is met. At this point, the group's current best position is taken as the solution by the algorithm.

This simple yet efficient process will be illustrated in detail in the next section, where we will go over a program that optimizes a function using the PSO algorithm.

PSO example – function optimization

For demonstration purposes, we will use the PSO algorithm to find the minimum location(s) of Himmelblau's function, a commonly used benchmark that we previously optimized using genetic algorithms in `Chapter 6`, *Optimizing Continuous Functions*. This function can be depicted by the following image:



Himmelblau's function
Source: https://commons.wikimedia.org/wiki/File:Himmelblau_function.svg
Image by Morn the Gorn. Released to the public domain

As a reminder, the function can be mathematically expressed as follows:

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

It has four global minima, evaluating to 0, indicated by the blue areas in the plot. These are located at the following coordinates:

- *x=3.0, y=2.0*
- *x=−2.805118, y=3.131312*

- *x=−3.779310, y=−3.283186*
- *x=3.584458, y=−1.848126*

For our example, we will attempt to find any one of these minima.

Particle swarm optimization implementation

To locate a minimum of Himmelblau's function using particle swarm optimization, we've created a Python program called `03-pso-himmelblau.py`, which is located at `https://github.com/PacktPublishing/Hands-On-Genetic-Algorithms-with-Python/blob/master/Chapter12/03-pso-himmelblau.py`.

The following steps describe the main parts of this program:

1. We start by setting various constants that will be used throughout the program. First is the dimensionality of the problem at hand – two, in our case – which, in turn, determines the dimensionality of the location and velocity of each particle. Next comes the population size – the total number of particles in the swarm, and the number of generations, or iterations, of running the algorithm:

```
DIMENSIONS = 2
POPULATION_SIZE = 20
MAX_GENERATIONS = 500
```

2. These are followed by several additional constants that affect how the particles are created and updated. We will see how they play their roles as we examine the rest of the program:

```
MIN_START_POSITION, MAX_START_POSITION = -5, 5
MIN_SPEED, MAX_SPEED = -3, 3
MAX_LOCAL_UPDATE_FACTOR = MAX_GLOBAL_UPDATE_FACTOR = 2.0
```

3. Since our goal is to locate a minimum in Himmelblau's function, we need to define a single objective, minimizing fitness strategy:

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

4. Now, we need to create the `Particle` class. Since this class represents a location in the continuous space, we could base it on an ordinary list of floats. However, here, we decided to use the `numpy` library's N-dimensional array (`ndarray`) since it lends itself to element-wise algebraic operations, such as addition and multiplication, which will be needed when we update the particle's location. Besides the current location, the `Particle` class is given several additional attributes:
    - `fitness`, using the minimizing fitness we defined earlier.
    - `speed`, which is used to hold the current speed of the particle in each dimension. Although its initial value is `None`, the speed will be populated with another `ndarray` later.
    - `best`, which represents the best location that's been recorded so far for this particular particle (local best).

   The resulting definition for the `Particle` class creator looks as follows:

```
creator.create("Particle", np.ndarray,
fitness=creator.FitnessMin, speed=None, best=None)
```

5. To help us construct an individual particle in the population, we need to define a helper function that will create and initialize a random particle. We will use the numpy library's `random.uniform()` function to randomly generate the location and speed arrays of the new particle, within the given boundaries:

```
def createParticle():
    particle = creator.Particle(np.random.uniform(
                                                MIN_START_POSITION,
                                                MAX_START_POSITION,
                                                DIMENSIONS))
    particle.speed = np.random.uniform(MIN_SPEED, MAX_SPEED,
                                                DIMENSIONS)
    return particle
```

6. This function is used in the definition of the operator that creates a particle instance. This, in turn, is used by the population creation operator:

```
toolbox.register("particleCreator", createParticle)
toolbox.register("populationCreator", tools.initRepeat, list,
toolbox.particleCreator)
```

7. Next comes the method that serves as the heart of the algorithm – `updateParticle()`. This method is responsible for updating the location and speed of each particle in the population. The arguments of this function are a single particle in the population and the best currently recorded position.

The method starts by creating two random factors – one for the local update and the other for the global update – within the preset range. Then, it calculates two corresponding speed updates (local and global) and adds them to the current particle's speed.

Note that all the values that are involved are of the `ndarray` type are two-dimensional in our case, and the calculations are performed element-wise, one per dimension.

The updated particle speed is effectively a combination of the particle's original speed (representing inertia), the particle's best-known location (cognitive force), and the best-known location of the entire population (social force):

```
def updateParticle(particle, best):

    localUpdateFactor = np.random.uniform(0,
                                          MAX_LOCAL_UPDATE_FACTOR,
                                          particle.size)
    globalUpdateFactor = np.random.uniform(0,
                                       MAX_GLOBAL_UPDATE_FACTOR,
                                       particle.size)

    localSpeedUpdate = localUpdateFactor * (particle.best -
                                                    particle)
    globalSpeedUpdate = globalUpdateFactor * (best - particle)


    particle.speed = particle.speed + (localSpeedUpdate +
                                               lobalSpeedUpdate)
```

8. The `updateParticle()` method continues by making sure that the new speed does not exceed the preset limits and updates the location of the particles using the updated speed. As we mentioned previously, both the location and speed are of the `ndarray` type and have separate components for each dimension:

```
particle.speed = np.clip(particle.speed, MIN_SPEED, MAX_SPEED)
particle[:] = particle + particle.speed
```

9. Then, we register the `updateParticle()` method as a toolbox operator that will be in the `main` loop later:

```
toolbox.register("update", updateParticle)
```

10. We still need to define the function to be optimized – Himmelblau's function, in our case – and register it as the fitness evaluation operator:

```
def himmelblau(particle):
    x = particle[0]
    y = particle[1]
    f = (x ** 2 + y - 11) ** 2 + (x + y ** 2 - 7) ** 2
    return f,  # return a tuple

toolbox.register("evaluate", himmelblau)
```

11. Now that we're finally at the `main()` method, we can start it by creating the population of particles:

```
population = toolbox.populationCreator(n=POPULATION_SIZE)
```

12. Before starting the algorithm's `main` loop, we need to create the *stats* object, in order to calculate the population's statistics, and the `logbook` object, in order to record the statistics at every iteration:

```
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("min", np.min)
stats.register("avg", np.mean)

logbook = tools.Logbook()
logbook.header = ["gen", "evals"] + stats.fields
```

13. The program's main loop contains an external loop that iterates over the generations/update cycles. Within each iteration, there are two secondary loops, each iterating over all the particles in the population. The first loop, which can be seen in the following code, evaluates each particle against the function to be optimized and updates the local best and the global best if necessary:

```
particle.fitness.values = toolbox.evaluate(particle)

# local best:
if particle.best is None or particle.best.size == 0 or
particle.best.fitness < particle.fitness:
    particle.best = creator.Particle(particle)
    particle.best.fitness.values = particle.fitness.values

# global best:
if best is None or best.size == 0 or best.fitness <
particle.fitness:
    best = creator.Particle(particle)
    best.fitness.values = particle.fitness.values
```

14. The second inner loop calls the `update` operator. As we saw previously, this operator updates the speed and the location of the particle using a combination of inertia, cognitive force, and social force:

```
toolbox.update(particle, best)
```

15. At the end of the outer loop, we record the statistics for the current generation and print them:

```
logbook.record(gen=generation, evals=len(population),
**stats.compile(population))
print(logbook.stream)
```

16. Once the outer loop is done, we print the information for the best location that was recorded during the run. This is considered the solution that the algorithm has found for the problem at hand:

```
# print info for best solution found:
print("-- Best Particle = ", best)
print("-- Best Fitness = ", best.fitness.values[0])
```

By running this program, we get the following output:

```
gen evals min avg
0 20 8.74399 167.468
1 20 19.0871 357.577
2 20 32.4961 219.132
...
497 20 7.2162 412.189
498 20 6.87945 273.712
499 20 16.1034 272.385

-- Best Particle = [-3.77695478 -3.28649153]
-- Best Fitness = 0.0010248367255068806
```

These results indicate that the algorithm was able to locate one of the minima, around $x=-3.77$ and $y=-3.28$. Looking at the stats we recorded along the way, we can see that the best result was achieved at generation 480. It is also evident that the particles move around quite a bit and, during the run, oscillate closer to the best result and away from it.

To find the other minima locations, you can rerun the algorithm with a different random seed. You can also penalize the solutions in the areas around the previously found minima, just like we did with Simionescu's function in `Chapter 6`, *Optimizing Continuous Functions*. Another approach could be using multiple simultaneous swarms to locate several minima in the same run – you are encouraged to try this on your own (see the *Further reading* section for more information).

In the next section, we will briefly review several more members of the extended evolutionary computation family.

Other related techniques

Besides the techniques we have covered so far, there are numerous other problem-solving and optimization techniques that draw their inspiration from the Darwinian evolution theory, as well as from various biological systems and behaviors. The following subsections briefly describe several more of these techniques.

Evolution strategies

Evolution strategies (**ES**) are a kind of genetic algorithm that emphasizes mutation rather than crossover as the evolutionary facilitator. The mutation is adaptive, and its strength is learned over the generations. The selection operator in evolution strategy is always rank-based rather than done using actual fitness values. A simple version of this technique is called (1 + 1). It includes only two individuals – a parent and its mutated offspring. The best of them continues to be the parent of the next mutated offspring. In the more general case, called (1 + λ), there is one parent and λ mutated offspring, and the best of the offspring continues to be the parent of the next λ offspring. Some newer variations of the algorithm include more than one parent, as well as a crossover operator.

Differential evolution

Differential evolution (**DE**) is a specialized variant of genetic algorithms that's used for the optimization of real-valued functions. DE differs from genetic algorithms in the following aspects:

- The DE population is always represented as a collection of real-valued vectors.
- Instead of replacing the entire current generation with a new generation, DE keeps iterating over the population, modifying one individual at a time, or keeping the original individual if it's better than its modified version.
- The traditional crossover and mutation operators are replaced by specialized ones, thereby modifying the value of the current individual using the values of three other individuals that are chosen at random.

Ant colony optimization

Ant colony optimization (**ACO**) algorithms are inspired by the way certain species of ants locate food. The ants start by wandering randomly, and when any of them locates food, they go back to their colony while depositing pheromones along the way, marking the path for other ants. Other ants finding food at the same location will reinforce the trail by depositing their own pheromones. The pheromone marks fade away over time, giving the shorter paths and the paths that are traveled more often an advantage.

ACO algorithms use artificial ants that move about in the search space looking for the location of the best solutions. The ants keep track of their locations and the candidate solutions they have found along the way. This information is used by the ants of the subsequent iterations so that they can find better solutions. These algorithms are often combined with the local search method, which is activated after locating an area of interest.

Artificial immune systems

Artificial immune systems (**AIS**) draw their inspiration from the characteristics of adaptive immune systems found in mammals. These systems are capable of identifying and learning new threats, as well as applying the acquired knowledge and responding faster the next time a similar threat is detected.

Recent AIS can be used in various machine learning and optimization tasks, and generally belong to one of the following three subfields:

- **Clonal selection**: Imitating the process by which the immune system selects the best cell to recognize and eliminate an antigen that enters the body. The cell is chosen out of a pool of preexisting cells with varying specificities, and once chosen, it is cloned to create a population of cells that eliminates the invading antigen. This paradigm is typically applied to optimization and pattern recognition tasks.
- **Negative selection**: This follows a process that identifies and deletes cells that may attack self-tissues. These algorithms are typically used in anomaly detection tasks, where normal patterns are used to "negatively" train filters that will then be able to detect anomalous patterns.

- **Immune network algorithms**: This is inspired by the theory that suggests that the immune system is regulated using special types of antibodies that bind to other antibodies. In this type of algorithm, antibodies represent nodes in a network and the learning process involves the creation or removal of edges between the nodes, resulting in an evolving network graph structure. These algorithms are typically used in non-supervised machine learning tasks, as well as in the fields of control and optimization.

Artificial life

Rather than being a branch of evolutionary computation, **artificial life** (**ALife**) is a broader field that involves systems and processes that imitate natural life in different ways, such as computer simulations and robotic systems.

Evolutionary computation can actually be viewed as an application of ALife, where the population seeking to optimize a certain fitness function is a metaphor for organisms searching for food. The niching and sharing mechanisms, for example, that we described in `Chapter 2`, *Understanding the Key Components of Genetic Algorithms*, draw directly from the food metaphor.

The main branches of ALife are as follows:

- **Soft**: Represents software-based (digital) simulation
- **Hard**: Represents hardware-based (physical) robotics
- **Wet**: Represents biochemical-based manipulation or synthetic biology

ALife can also be viewed as the bottom-up counterpart to artificial intelligence since ALife typically builds on the biological environment, mechanisms, and structures rather than high-level cognition.

Summary

In this chapter, you were introduced to the extended family of evolutionary computation and some of the common characteristics of its members. Then, we used genetic programming – a special case of genetic algorithms – to implement the even parity check task. This was followed by creating a program that utilized the particle swarm optimization technique to optimize Himmelblau's function. We concluded this chapter with a brief overview of several other related problem-solving techniques.

Now that this book has come to its end, I wanted to thank you for taking this journey with me and going through the various aspects and use cases of genetic algorithms and evolutionary computation. I hope you found this book interesting as well as thought-provoking. As this book demonstrated, genetic algorithms and their related techniques can be applied to a plethora of tasks in virtually any computation and engineering field, including – very likely – the ones you are currently involved with. Remember, all that is required for the genetic algorithm to start crunching a problem is a way to represent a solution and a way to evaluate a solution – or compare two solutions. Since this is the age of artificial intelligence and cloud computing, you will find that genetic algorithms lend themselves well to both, and can be a powerful tool in your arsenal when approaching a new challenge.

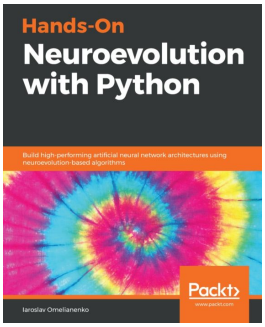
Further reading

For more information on the topics that were covered in this chapter, please refer to the following resources:

- Genetic Programming: Bio-inspired machine learning `http://geneticprogramming.com/tutorial/`
- *Machine Learning for Finance*, Jannes Klaas, May 30, 2019
- *Artificial Intelligence for Big Data*, Manish Kumar and Anand Deshpande, May 21, 2018
- *Multimodal optimization using particle swarm optimization algorithms:* CEC 2015 competition on single objective multi-niche optimization: `https://ieeexplore.ieee.org/document/7257009`

Other Books You May Enjoy

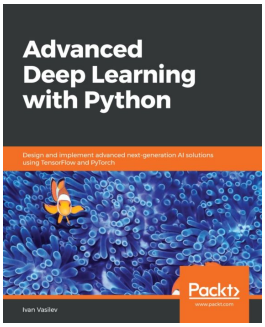
If you enjoyed this book, you may be interested in these other books by Packt:



Hands-On Neuroevolution with Python
Iaroslav Omelianenko

ISBN: 978-1-83882-491-4

- Discover the most popular neuroevolution algorithms – NEAT, HyperNEAT, and ES-HyperNEAT
- Explore how to implement neuroevolution-based algorithms in Python
- Get up to speed with advanced visualization tools to examine evolved neural network graphs
- Understand how to examine the results of experiments and analyze algorithm performance
- Delve into neuroevolution techniques to improve the performance of existing methods
- Apply deep neuroevolution to develop agents for playing Atari games



Advanced Deep Learning with Python
Ivan Vasilev

ISBN: 9-781-78995-617-7

- Cover advanced and state-of-the-art neural network architectures
- Understand the theory and math behind neural networks
- Train DNNs and apply them to modern deep learning problems
- Use CNNs for object detection and image segmentation
- Implement generative adversarial networks (GANs) and variational autoencoders to generate new images
- Solve natural language processing (NLP) tasks, such as machine translation, using sequence-to-sequence models
- Understand DL techniques, such as meta-learning and graph neural networks

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X