Prateek Joshi

Artificial Intelligence with Python

Build real-world Artificial Intelligence applications with Python to intelligently interact with the world around you

Packt>

Artificial Intelligence with Python

Build real-world Artificial Intelligence applications with Python to intelligently interact with the world around you

Prateek Joshi

BIRMINGHAM - MUMBAI

Artificial Intelligence with Python

First published: January 2017

Production reference: 1230117

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78646-439-2

www.packtpub.com

Credits

Author

Prateek Joshi

Reviewer

Richard Marsden

Commissioning Editor

Veena Pagare

Acquisition Editor

Tushar Gupta

Content Development Editor

Aishwarya Pandere

Technical Editor

Karan Thakkar

Copy Editors

Vikrant Phadkay

Safis Editing

Project Coordinator

Nidhi Joshi

Proofreader

Safis Editing

Indexer

Mariammal Chettiyar

Production Coordinator

Shantanu N. Zagade

About the Author

Prateek Joshi is an artificial intelligence researcher, published author of five books, and TEDx speaker. He is the founder of Pluto AI, a venture-funded Silicon Valley startup building an analytics platform for smart water management powered by deep learning. His work in this field has led to patents, tech demos, and research papers at major IEEE conferences. He has been an invited speaker at technology and entrepreneurship conferences including TEDx, AT&T Foundry, Silicon Valley Deep Learning, and Open Silicon Valley. Prateek has also been featured as a guest author in prominent tech magazines.

His tech blog (`www.prateekjoshi.com`) has received more than 1.2 million page views from 200 over countries and has over 6,600+ followers. He frequently writes on topics such as artificial intelligence, Python programming, and abstract mathematics. He is an avid coder and has won many hackathons utilizing a wide variety of technologies. He graduated from University of Southern California with a master's degree specializing in artificial intelligence. He has worked at companies such as Nvidia and Microsoft Research. You can learn more about him on his personal website at `www.prateekj.com`.

About the Reviewer

Richard Marsden has over 20 years of professional software development experience. After starting in the field of geophysical surveying for the oil industry, he has spent the last ten years running the Winwaed Software Technology LLC independent software vendor. Winwaed specializes in geospatial tools and applications including web applications, and operates the `http://www.mapping-tools.com` website for tools and add-ins for geospatial applications such as Caliper Maptitude and Microsoft MapPoint.

Richard was also a technical reviewer of the following Packt publications: *Python Geospatial Development* and *Python Geospatial Analysis Essentials*, both by *Erik Westra*; *Python Geospatial Analysis Cookbook* by *Michael Diener*; *Mastering Python Forensics* by *Drs Michael Spreitzenbarth* and *Dr Johann Uhrmann*; and *Effective Python Penetration Testing* by *Rejah Rehim.*

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`.

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.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.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.

`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thank you for purchasing this Packt book. We take our commitment to improving our content and products to meet your needs seriously—that's why your feedback is so valuable. Whatever your feelings about your purchase, please consider leaving a review on this book's Amazon page. Not only will this help us, more importantly it will also help others in the community to make an informed decision about the resources that they invest in to learn.

You can also review for us on a regular basis by joining our reviewers' club. **If you're interested in joining, or would like to learn more about the benefits we offer, please contact us**: `customerreviews@packtpub.com`.

Table of Contents

Preface

Artificial intelligence is becoming increasingly relevant in the modern world where everything is driven by data and automation. It is used extensively across many fields such as image recognition, robotics, search engines, and self-driving cars. In this book, we will explore various real-world scenarios. We will understand what algorithms to use in a given context and write functional code using this exciting book.

We will start by talking about various realms of artificial intelligence. We'll then move on to discuss more complex algorithms, such as Extremely Random Forests, Hidden Markov Models, Genetic Algorithms, Artificial Neural Networks, and Convolutional Neural Networks, and so on. This book is for Python programmers looking to use artificial intelligence algorithms to create real-world applications. This book is friendly to Python beginners, but familiarity with Python programming would certainly be helpful so you can play around with the code. It is also useful to experienced Python programmers who are looking to implement artificial intelligence techniques.

You will learn how to make informed decisions about the type of algorithms you need to use and how to implement those algorithms to get the best possible results. If you want to build versatile applications that can make sense of images, text, speech, or some other form of data, this book on artificial intelligence will definitely come to your rescue!

What this book covers

`Chapter 1`, *Introduction to Artificial Intelligence*, teaches you various introductory concepts in artificial intelligence. It talks about applications, branches, and modeling of Artificial Intelligence. It walks the reader through the installation of necessary Python packages.

`Chapter 2`, *Classification and Regression Using Supervised Learning*, covers various supervised learning techniques for classification and regression. You will learn how to analyze income data and predict housing prices.

`Chapter 3`, *Predictive Analytics with Ensemble Learning*, explains predictive modeling techniques using Ensemble Learning, particularly focused on Random Forests. We will learn how to apply these techniques to predict traffic on the roads near sports stadiums.

`Chapter 4`, *Detecting Patterns with Unsupervised Learning*, covers unsupervised learning algorithms including K-means and Mean Shift Clustering. We will learn how to apply these algorithms to stock market data and customer segmentation.

`Chapter 5`, *Building Recommender Systems*, illustrates algorithms used to build recommendation engines. You will learn how to apply these algorithms to collaborative filtering and movie recommendations.

`Chapter 6`, *Logic Programming*, covers the building blocks of logic programming. We will see various applications, including expression matching, parsing family trees, and solving puzzles.

`Chapter 7`, *Heuristic Search Techniques*, shows heuristic search techniques that are used to search the solution space. We will learn about various applications such as simulated annealing, region coloring, and maze solving.

`Chapter 8`, *Genetic Algorithms*, covers evolutionary algorithms and genetic programming. We will learn about various concepts such as crossover, mutation, and fitness functions. We will then use these concepts to solve the symbol regression problem and build an intelligent robot controller.

`Chapter 9`, *Building Games with Artificial Intelligence*, teaches you how to build games with artificial intelligence. We will learn how to build various games including Tic Tac Toe, Connect Four, and Hexapawn.

`Chapter 10`, *Natural Language Processing*, covers techniques used to analyze text data including tokenization, stemming, bag of words, and so on. We will learn how to use these techniques to do sentiment analysis and topic modeling.

`Chapter 11`, *Probabilistic Reasoning for Sequential Data*, shows you techniques used to analyze time series and sequential data including Hidden Markov models and Conditional Random Fields. We will learn how to apply these techniques to text sequence analysis and stock market predictions.

`Chapter 12`, *Building A Speech Recognizer*, demonstrates algorithms used to analyze speech data. We will learn how to build speech recognition systems.

`Chapter 13`, *Object Detection and Tracking*, It covers algorithms related to object detection and tracking in live video. We will learn about various techniques including optical flow, face tracking, and eye tracking.

`Chapter 14`, *Artificial Neural Networks*, covers algorithms used to build neural networks. We will learn how to build an Optical Character Recognition system using neural networks.

`Chapter 15`, *Reinforcement Learning*, teaches the techniques used to build reinforcement learning systems. We will learn how to build learning agents that can learn from interacting with the environment.

`Chapter 16`, *Deep Learning with Convolutional Neural Networks*, covers algorithms used to build deep learning systems using Convolutional Neural Networks. We will learn how to use TensorFlow to build neural networks. We will then use it to build an image classifier using convolutional neural networks.

What you need for this book

This book is focused on artificial intelligence in Python as opposed to the Python itself. We have used Python 3 to build various applications. We focus on how to utilize various Python libraries in the best possible way to build real world applications. In that spirit, we have tried to keep all of the code as friendly and readable as possible. We feel that this will enable our readers to easily understand the code and readily use it in different scenarios.

Who this book is for

This book is for Python developers who want to build real-world artificial intelligence applications. This book is friendly to Python beginners, but being familiar with Python would be useful to play around with the code. It will also be useful for experienced Python programmers who are looking to use artificial intelligence techniques in their existing technology stacks.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

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

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

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

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
     /etc/asterisk/cdr_mysql.conf
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "The shortcuts in this book are based on the `Mac OS X 10.5+` scheme."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

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

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

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

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/ArtificialIntelligencewithPython_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support`and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Introduction to Artificial Intelligence

In this chapter, we are going to discuss the concept of **Artificial Intelligence** (**AI**) and how it's applied in the real world. We spend a significant portion of our everyday life interacting with smart systems. It can be in the form of searching for something on the internet, Biometric face recognition, or converting spoken words to text. Artificial Intelligence is at the heart of all this and it's becoming an important part of our modern lifestyle. All these system are complex real-world applications and Artificial Intelligence solves these problems with mathematics and algorithms. During the course of this book, we will learn the fundamental principles that are used to build such applications and then implement them as well. Our overarching goal is to enable you to take up new and challenging Artificial Intelligence problems that you might encounter in your everyday life.

By the end of this chapter, you will know:

- What is AI and why do we need to study it?
- Applications of AI
- Branches of AI
- Turing test
- Rational agents

- General Problem Solvers
- Building an intelligent agent
- Installing Python 3 on various operating systems
- Installing the necessary Python packages

What is Artificial Intelligence?

Artificial Intelligence (**AI**) is a way to make machines think and behave intelligently. These machines are controlled by software inside them, so AI has a lot to do with intelligent software programs that control these machines. It is a science of finding theories and methodologies that can help machines understand the world and accordingly react to situations in the same way that humans do.

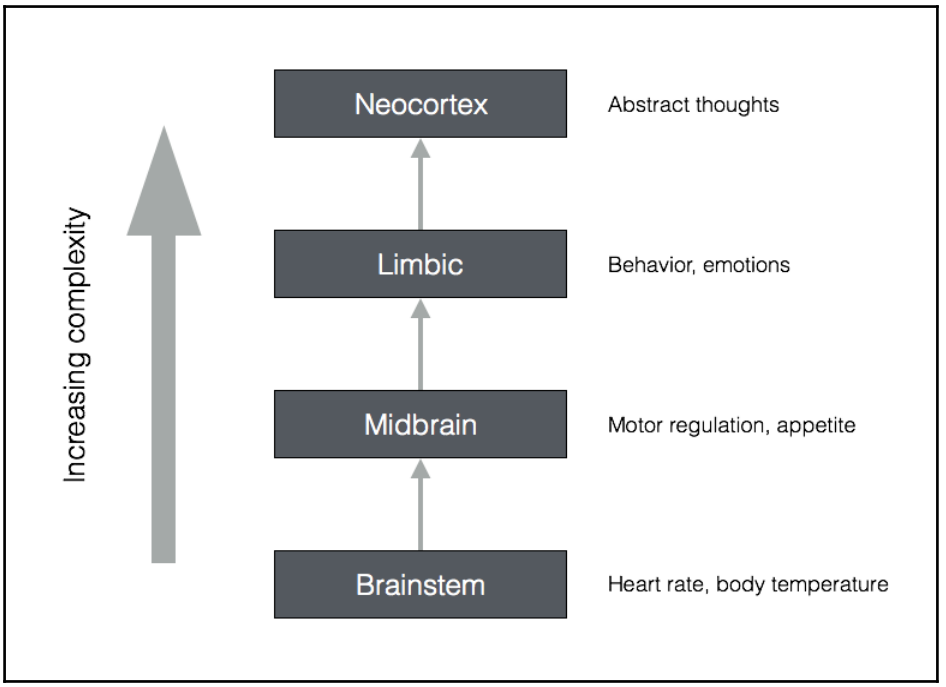
If we look closely at how the field of AI has emerged over the last couple of decades, you will see that different researchers tend to focus on different concepts to define AI. In the modern world, AI is used across many verticals in many different forms. We want the machines to sense, reason, think, and act. We want our machines to be rational too.

AI is closely related to the study of human brain. Researchers believe that AI can be accomplished by understanding how the human brain works. By mimicking the way the human brain learns, thinks, and takes action, we can build a machine that can do the same. This can be used as a platform to develop intelligent systems that are capable of learning.

Why do we need to study AI?

AI has the ability to impact every aspect of our lives. The field of AI tries to understand patterns and behaviors of entities. With AI, we want to build smart systems and understand the concept of intelligence as well. The intelligent systems that we construct are very useful in understanding how an intelligent system like our brain goes about constructing another intelligent system.

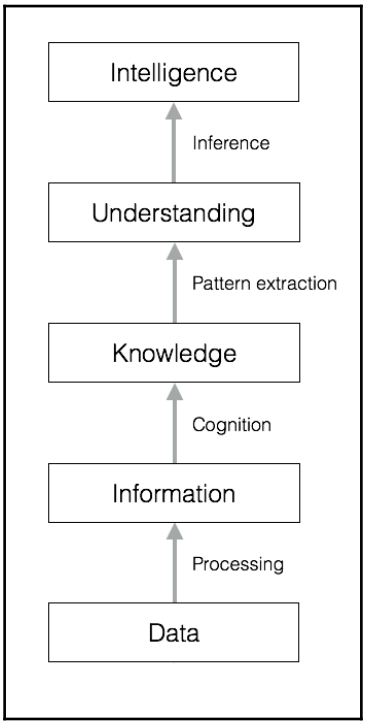
Let's take a look at how our brain processes information:



Compared to some other fields such as Mathematics or Physics that have been around for centuries, AI is relatively in its infancy. Over the last couple of decades, AI has produced some spectacular products such as self-driving cars and intelligent robots that can walk. Based on the direction in which we are heading, it's pretty obvious that achieving intelligence will have a great impact on our lives in the coming years.

We can't help but wonder how the human brain manages to do so much with such effortless ease. We can recognize objects, understand languages, learn new things, and perform many more sophisticated tasks with our brain. How does the human brain do this? When you try to do this with a machine, you will see that it falls way behind! For example, when we try to look for things such as extraterrestrial life or time travel, we don't know if those things exist. The good thing about the holy grail of AI is that we know it exists. Our brain is the holy grail! It is a spectacular example of an intelligent system. All we have to do is to mimic its functionality to create an intelligent system that can do something similar, possibly even more.

Let's see how raw data gets converted to wisdom through various levels of processing:



One of the main reasons we want to study AI is to automate many things. We live in a world where:

- We deal with huge and insurmountable amounts of data. The human brain can't keep track of so much data.
- Data originates from multiple sources simultaneously.
- The data is unorganized and chaotic.
- Knowledge derived from this data has to be updated constantly because the data itself keeps changing.
- The sensing and actuation has to happen in real time with high precision.

Even though the human brain is great at analyzing things around us, it cannot keep up with the preceding conditions. Hence, we need to design and develop intelligent machines that can do this. We need AI systems that can:

- Handle large amounts of data in an efficient way. With the advent of Cloud Computing, we are now able to store huge amounts of data.
- Ingest data simultaneously from multiple sources without any lag.
- Index and organize data in a way that allows us to derive insights.
- Learn from new data and update constantly using the right learning algorithms.
- Think and respond to situations based on the conditions in real time.

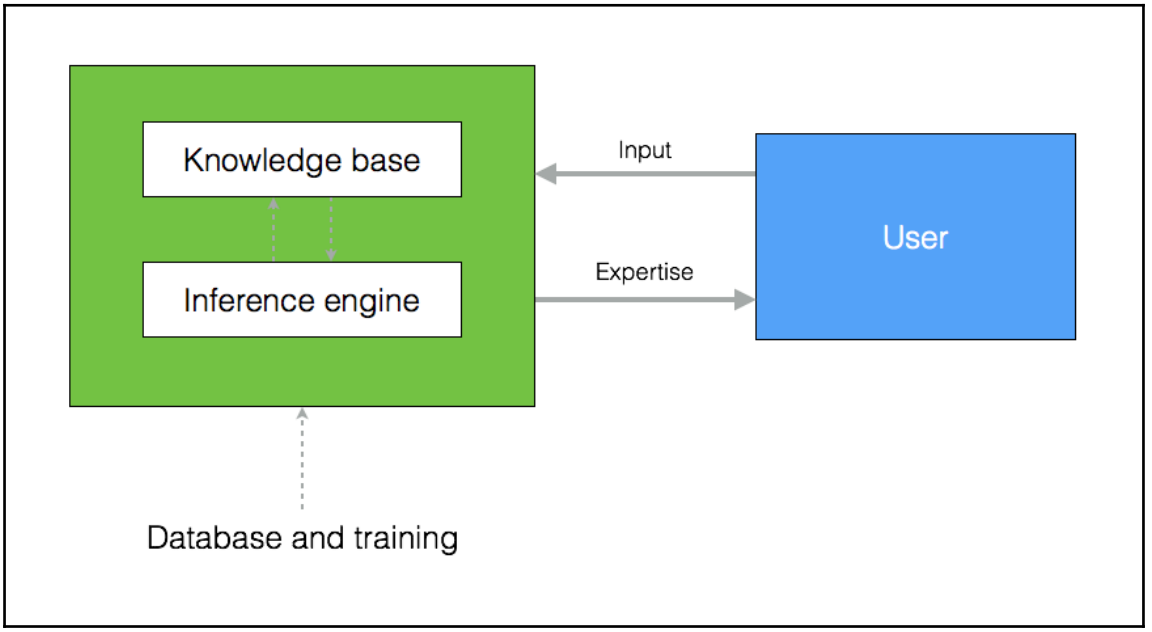
AI techniques are actively being used to make existing machines smarter, so that they can execute faster and more efficiently.

Applications of AI

Now that we know how information gets processed, let's see where AI appears in the real world. AI manifests itself in various different forms across multiple fields, so it's important to understand how it's useful in various domains. AI has been used across many industries and it continues to expand rapidly. Some of the most popular areas include:

- **Computer Vision**: These are the systems that deal with visual data such as images and videos. These systems understand the content and extract insights based on the use case. For example, Google uses reverse image search to search for visually similar images across the Web.

- **Natural Language Processing**: This field deals with understanding text. We can interact with a machine by typing natural language sentences. Search engines use this extensively to deliver the right search results.
- **Speech Recognition**: These systems are capable of hearing and understanding spoken words. For example, there are intelligent personal assistants on our smartphones that can understand what we are saying and give relevant information or perform an action based on that.
- **Expert Systems**: These systems use AI techniques to provide advice or make decisions. They usually use databases of expert knowledge areas such as finance, medicine, marketing, and so on to give advice about what to do next. Let's see what an expert system looks like and how it interacts with the user:



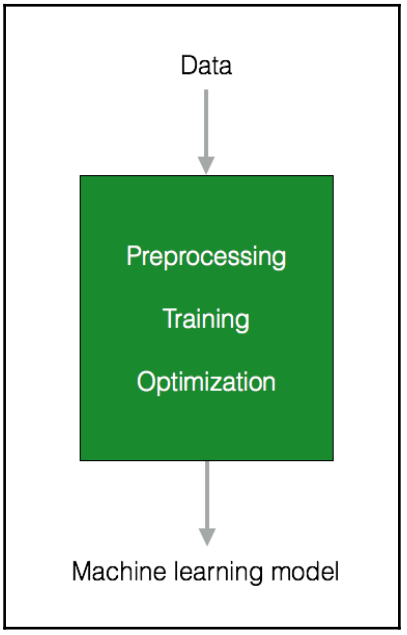
- **Games**: AI is used extensively in the gaming industry. It is used to design intelligent agents that can compete with humans. For example, **AlphaGo** is a computer program that can play the strategy game Go. It is also used in designing many other types of games where we expect the computer to behave intelligently.

- **Robotics**: Robotic systems actually combine many concepts in AI. These systems are able to perform many different tasks. Depending on the situation, robots have sensors and actuators that can do different things. These sensors can see things in front of them and measure the temperature, heat, movements, and so on. They have processors on board that compute various things in real time. They are also capable of adapting to the new environments.

Branches of AI

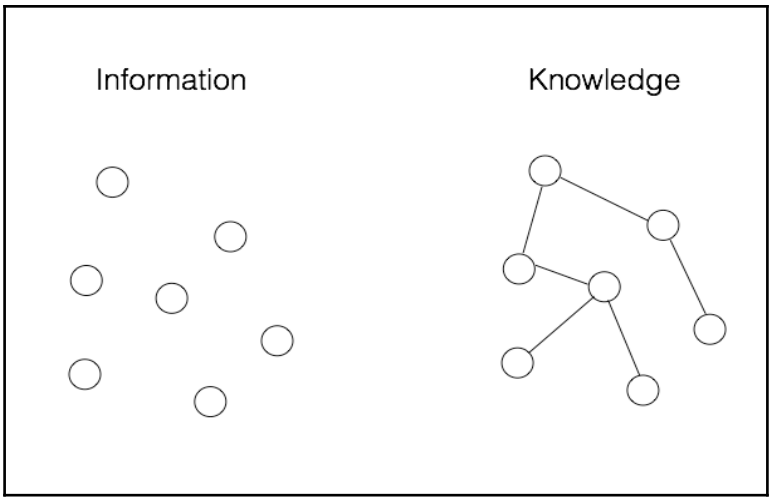
It is important to understand the various fields of study within AI so that we can choose the right framework to solve a given real-world problem. Here's a list of topics that are dominant:

- **Machine learning and pattern recognition**: This is perhaps the most popular form of AI out there. We design and develop software that can learn from data. Based on these learning models, we perform predictions on unknown data. One of the main constraints here is that these programs are limited to the power of the data. If the dataset is small, then the learning models would be limited as well. Let's see what a typical machine learning system looks like:



When a system makes an observation, it is trained to compare it with what it has already seen in the form of a pattern. For example, in a face recognition system, the software will try to match the pattern of eyes, nose, lips, eyebrows, and so on in order to find a face in the existing database of users.

- **Logic-based AI**: Mathematical logic is used to execute computer programs in logic-based AI. A program written in logic-based AI is basically a set of statements in logical form that express facts and rules about a particular problem domain. This is used extensively in pattern matching, language parsing, semantic analysis, and so on.
- **Search**: The Search techniques are used extensively in AI programs. These programs examine a large number of possibilities and then pick the most optimal path. For example, this is used a lot in strategy games such as Chess, networking, resource allocation, scheduling, and so on.
- **Knowledge representation**: The facts about the world around us need to be represented in some way for a system to make sense of them. The languages of mathematical logic are frequently used here. If knowledge is represented efficiently, systems can be smarter and more intelligent. Ontology is a closely related field of study that deals with the kinds of objects that exist. It is a formal definition of the properties and relationships of the entities that exist in a particular domain. This is usually done with a particular taxonomy or a hierarchical structure of some kind. The following diagram shows the difference between information and knowledge:



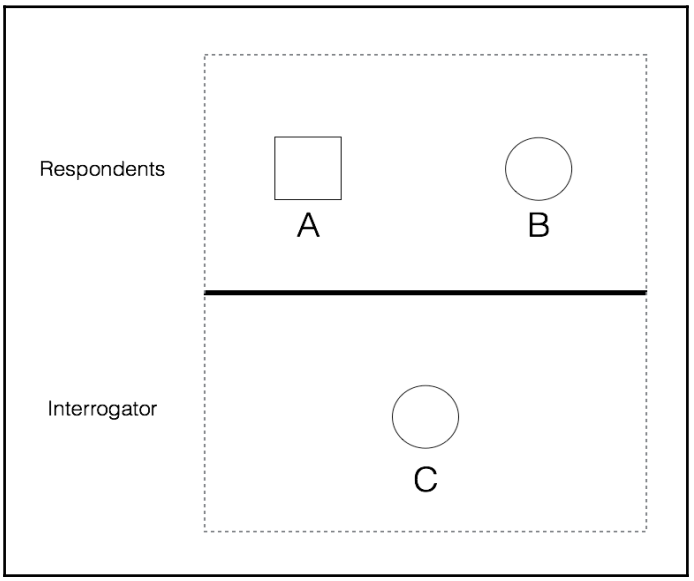
- **Planning**: This field deals with optimal planning that gives us maximum returns with minimal costs. These software programs start with facts about the particular situation and a statement of a goal. These programs are also aware of the facts of the world, so that they know what the rules are. From this information, they generate the most optimal plan to achieve the goal.
- **Heuristics**: A heuristic is a technique used to solve a given problem that's practical and useful in solving the problem in the short term, but not guaranteed to be optimal. This is more like an educated guess on what approach we should take to solve a problem. In AI, we frequently encounter situations where we cannot check every single possibility to pick the best option. So we need to use heuristics to achieve the goal. They are used extensively in AI in fields such as robotics, search engines, and so on.
- **Genetic programming**: Genetic programming is a way to get programs to solve a task, by mating programs and selecting the fittest. The programs are encoded as a set of genes, using an algorithm to get a program that is able to perform the given task really well.

Defining intelligence using Turing Test

The legendary computer scientist and mathematician, *Alan Turing*, proposed the Turing Test to provide a definition of intelligence. It is a test to see if a computer can learn to mimic human behavior. He defined intelligent behavior as the ability to achieve human-level intelligence during a conversation. This performance should be sufficient to trick an interrogator into thinking that the answers are coming from a human.

To see if a machine can do this, he proposed a test setup: he proposed that a human should interrogate the machine through a text interface. Another constraint is that the human cannot know who's on the other side of the interrogation, which means it can either be a machine or a human. To enable this setup, a human will be interacting with two entities through a text interface. These two entities are called respondents. One of them will be a human and the other one will be the machine.

The respondent machine passes the test if the interrogator is unable to tell whether the answers are coming from a machine or a human. The following diagram shows the setup of a Turing Test:



As you can imagine, this is quite a difficult task for the respondent machine. There are a lot of things going on during a conversation. At the very minimum, the machine needs to be well versed with the following things:

- **Natural Language Processing**: The machine needs this to communicate with the interrogator. The machine needs to parse the sentence, extract the context, and give an appropriate answer.
- **Knowledge Representation**: The machine needs to store the information provided before the interrogation. It also needs to keep track of the information being provided during the conversation so that it can respond appropriately if it comes up again.

- **Reasoning**: It's important for the machine to understand how to interpret the information that gets stored. Humans tend to do this automatically to draw conclusions in real time.
- **Machine Learning**: This is needed so that the machine can adapt to new conditions in real time. The machine needs to analyze and detect patterns so that it can draw inferences.

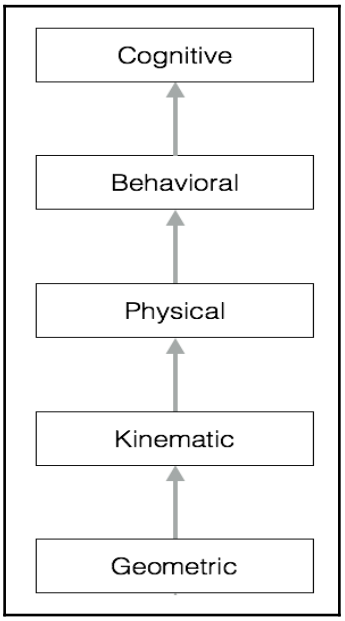
You must be wondering why the human is communicating with a text interface. According to Turing, physical simulation of a person is unnecessary for intelligence. That's the reason the Turing Test avoids direct physical interaction between the human and the machine. There is another thing called the Total Turing Test that deals with vision and movement. To pass this test, the machine needs to see objects using computer vision and move around using Robotics.

Making machines think like humans

For decades, we have been trying to get the machine to think like a human. In order to make this happen, we need to understand how humans think in the first place. How do we understand the nature of human thinking? One way to do this would be to note down how we respond to things. But this quickly becomes intractable, because there are too many things to note down. Another way to do this is to conduct an experiment based on a predefined format. We develop a certain number of questions to encompass a wide variety of human topics, and then see how people respond to it.

Once we gather enough data, we can create a model to simulate the human process. This model can be used to create software that can think like humans. Of course this is easier said than done! All we care about is the output of the program given a particular input. If the program behaves in a way that matches human behavior, then we can say that humans have a similar thinking mechanism.

The following diagram shows different levels of thinking and how our brain prioritizes things:

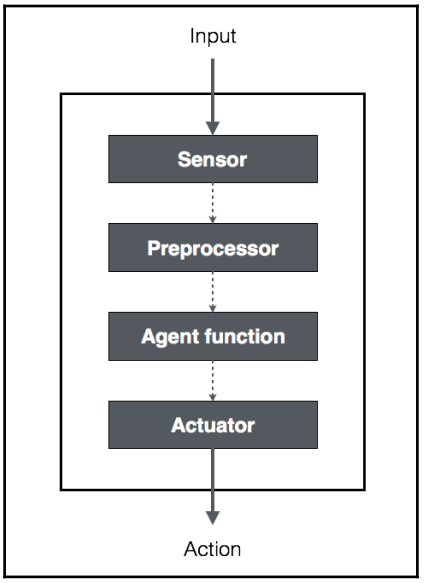


Within computer science, there is a field of study called **Cognitive Modeling** that deals with simulating the human thinking process. It tries to understand how humans solve problems. It takes the mental processes that go into this problem solving process and turns it into a software model. This model can then be used to simulate human behavior. Cognitive modeling is used in a variety of AI applications such as deep learning, expert systems, Natural Language Processing, robotics, and so on.

Building rational agents

A lot of research in AI is focused on building rational agents. What exactly is a rational agent? Before that, let us define the word rationality. Rationality refers to doing the right thing in a given circumstance. This needs to be performed in such a way that there is maximum benefit to the entity performing the action. An agent is said to act rationally if, given a set of rules, it takes actions to achieve its goals. It just perceives and acts according to the information that's available. This system is used a lot in AI to design robots when they are sent to navigate unknown terrains.

How do we define the *right* thing? The answer is that it depends on the objectives of the agent. The agent is supposed to be intelligent and independent. We want to impart the ability to adapt to new situations. It should understand its environment and then act accordingly to achieve an outcome that is in its best interests. The best interests are dictated by the overall goal it wants to achieve. Let's see how an input gets converted to action:



How do we define the performance measure for a rational agent? One might say that it is directly proportional to the degree of success. The agent is set up to achieve a particular task, so the performance measure depends on what percentage of that task is complete. But we must think as to what constitutes rationality in its entirety. If it's just about results, can the agent take any action to get there?

Making the right inferences is definitely a part of being rational, because the agent has to act rationally to achieve its goals. This will help it draw conclusions that can be used successively. What about situations where there are no provably right things to do? There are situations where the agent doesn't know what to do, but it still has to do something. In this situation, we cannot include the concept of inference to define rational behavior.

General Problem Solver

The **General Problem Solver** (**GPS**) was an AI program proposed by *Herbert Simon*, *J.C. Shaw*, and *Allen Newell*. It was the first useful computer program that came into existence in the AI world. The goal was to make it work as a universal problem-solving machine. Of course there were many software programs that existed before, but these programs performed specific tasks. GPS was the first program that was intended to solve any general problem. GPS was supposed to solve all the problems using the same base algorithm for every problem.

As you must have realized, this is quite an uphill battle! To program the GPS, the authors created a new language called **Information Processing Language** (**IPL**). The basic premise is to express any problem with a set of well-formed formulas. These formulas would be a part of a directed graph with multiple sources and sinks. In a graph, the source refers to the starting node and the sink refers to the ending node. In the case of GPS, the source refers to axioms and the sink refers to the conclusions.

Even though GPS was intended to be a general purpose, it could only solve well-defined problems, such as proving mathematical theorems in geometry and logic. It could also solve word puzzles and play chess. The reason was that these problems could be formalized to a reasonable extent. But in the real world, this quickly becomes intractable because of the number of possible paths you can take. If it tries to brute force a problem by counting the number of walks in a graph, it becomes computationally infeasible.

Solving a problem with GPS

Let's see how to structure a given problem to solve it using GPS:

1. The first step is to define the goals. Let's say our goal is to get some milk from the grocery store.
2. The next step is to define the preconditions. These preconditions are in reference to the goals. To get milk from the grocery store, we need to have a mode of transportation and the grocery store should have milk available.
3. After this, we need to define the operators. If my mode of transportation is a car and if the car is low on fuel, then we need to ensure that we can pay the fueling station. We need to ensure that you can pay for the milk at the store.

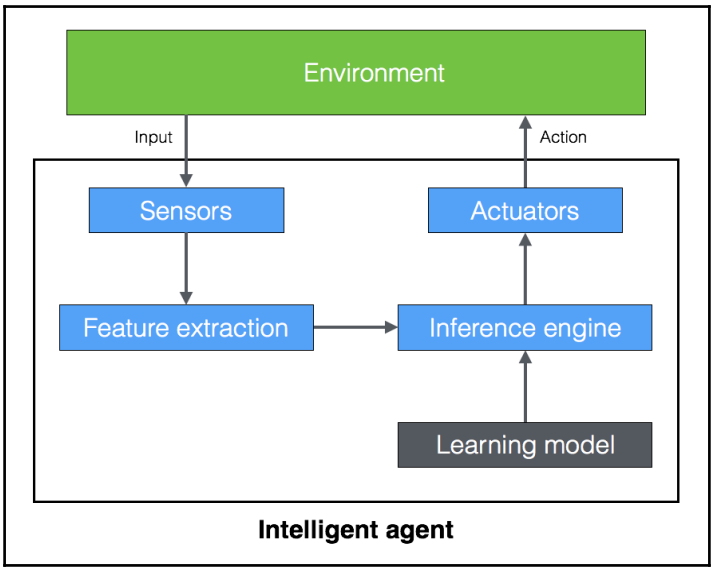
An operator takes care of the conditions and everything that affects them. It consists of actions, preconditions, and the changes resulting from taking actions. In this case, the action is giving money to the grocery store. Of course, this is contingent upon you having the money in the first place, which is the precondition. By giving them the money, you are changing your money condition, which will result in you getting the milk.

GPS will work as long as you can frame the problem like we did just now. The constraint is that it uses the search process to perform its job, which is way too computationally complex and time consuming for any meaningful real-world application.

Building an intelligent agent

There are many ways to impart intelligence to an agent. The most commonly used techniques include machine learning, stored knowledge, rules, and so on. In this section, we will focus on machine learning. In this method, the way we impart intelligence to an agent is through data and training.

Let's see how an intelligent agent interacts with the environment:



With machine learning, we want to program our machines to use labeled data to solve a given problem. By going through the data and the associated labels, the machine learns how to extract patterns and relationships.

In the preceding example, the intelligent agent depends on the learning model to run the inference engine. Once the sensor perceives the input, it sends it to the feature extraction block. Once the relevant features are extracted, the trained inference engine performs a prediction based on the learning model. This learning model is built using machine learning. The inference engine then takes a decision and sends it to the actuator, which then takes the required action in the real world.

There are many applications of machine learning that exist today. It is used in image recognition, robotics, speech recognition, predicting stock market behavior, and so on. In order to understand machine learning and build a complete solution, you will have to be familiar with many techniques from different fields such as pattern recognition, artificial neural networks, data mining, statistics, and so on.

Types of models

There are two types of models in the AI world: Analytical models and Learned models. Before we had machines that could compute, people used to rely on analytical models. These models were derived using a mathematical formulation, which is basically a sequence of steps followed to arrive at a final equation. The problem with this approach is that it was based on human judgment. Hence these models were simplistic and inaccurate with just a few parameters.

We then entered the world of computers. These computers were good at analyzing data. So, people increasingly started using learned models. These models are obtained through the process of training. During training, the machines look at many examples of inputs and outputs to arrive at the equation. These learned models are usually complex and accurate, with thousands of parameters. This gives rise to a very complex mathematical equation that governs the data.

Machine Learning allows us to obtain these learned models that can be used in an inference engine. One of the best things about this is the fact that we don't need to derive the underlying mathematical formula. You don't need to know complex mathematics, because the machine derives the formula based on data. All we need to do is create the list of inputs and the corresponding outputs. The learned model that we get is just the relationship between labeled inputs and the desired outputs.

Installing Python 3

We will be using Python 3 throughout this book. Make sure you have installed the latest version of Python 3 on your machine. Type the following command on your Terminal to check:

```
$ python3 --version
```

If you see something like Python 3.x.x (where x.x are version numbers) printed on your terminal, you are good to go. If not, installing it is pretty straightforward.

Installing on Ubuntu

Python 3 is already installed by default on Ubuntu 14.xx and above. If not, you can install it using the following command:

```
$ sudo apt-get install python3
```

Run the check command like we did earlier:

```
$ python3 --version
```

You should see the version number printed on your Terminal.

Installing on Mac OS X

If you are on Mac OS X, it is recommended that you use Homebrew to install Python 3. It is a great package installer for Mac OS X and it is really easy to use. If you don't have Homebrew, you can install it using the following command:

```
$ ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Let's update the package manager:

```
$ brew update
```

Let's install Python 3:

```
$ brew install python3
```

Run the check command like we did earlier:

```
$ python3 --version
```

You should see the version number printed on your Terminal.

Installing on Windows

If you use Windows, it is recommended that you use a `SciPy-stack` compatible distribution of Python 3. Anaconda is pretty popular and easy to use. You can find the installation instructions at: `https://www.continuum.io/downloads`.

If you want to check out other SciPy-stack compatible distributions of Python 3, you can find them at `http://www.scipy.org/install.html`. The good part about these distributions is that they come with all the necessary packages preinstalled. If you use one of these versions, you don't need to install the packages separately.

Once you install it, run the check command like we did earlier:

```
$ python3 --version
```

You should see the version number printed on your Terminal.

Installing packages

During the course of this book, we will use various packages such as NumPy, SciPy, scikit-learn, and matplotlib. Make sure you install these packages before you proceed.

If you use Ubuntu or Mac OS X, installing these packages is pretty straightforward. All these packages can be installed using a one-line command on the terminal. Here are the relevant links for installation:

- **NumPy**: `http://docs.scipy.org/doc/numpy-1.10.1/user/install.html`
- **SciPy**: `http://www.scipy.org/install.html`
- **scikit-learn**: `http://scikit-learn.org/stable/install.html`
- **matplotlib**: `http://matplotlib.org/1.4.2/users/installing.html`

If you are on Windows, you should have installed a `SciPy-stack` compatible version of Python 3.

Loading data

In order to build a learning model, we need data that's representative of the world. Now that we have installed the necessary Python packages, let's see how to use the packages to interact with data. Go into the Python terminal by typing the following command:

```
$ python3
```

Let's import the package containing all the datasets:

```
>>> from sklearn import datasets
```

Let's load the house prices dataset:

```
>>> house_prices = datasets.load_boston()
```

Print the data:

```
>>> print(house_prices.data)
```

You will see an output like this printed on your Terminal:

```
>>> print(house_prices.data)
[[  6.32000000e-03   1.80000000e+01   2.31000000e+00 ...,   1.53000000e+01
    3.96900000e+02   4.98000000e+00]
 [  2.73100000e-02   0.00000000e+00   7.07000000e+00 ...,   1.78000000e+01
    3.96900000e+02   9.14000000e+00]
 [  2.72900000e-02   0.00000000e+00   7.07000000e+00 ...,   1.78000000e+01
    3.92830000e+02   4.03000000e+00]
 ...,
 [  6.07600000e-02   0.00000000e+00   1.19300000e+01 ...,   2.10000000e+01
    3.96900000e+02   5.64000000e+00]
 [  1.09590000e-01   0.00000000e+00   1.19300000e+01 ...,   2.10000000e+01
    3.93450000e+02   6.48000000e+00]
 [  4.74100000e-02   0.00000000e+00   1.19300000e+01 ...,   2.10000000e+01
    3.96900000e+02   7.88000000e+00]]
```

Let's check out the labels:

You will see the following printed on your Terminal:

```
>>> print(house_prices.target)
[ 24.   21.6  34.7  33.4  36.2  28.7  22.9  27.1  16.5  18.9  15.   18.9
  21.7  20.4  18.2  19.9  23.1  17.5  20.2  18.2  13.6  19.6  15.2  14.5
  15.6  13.9  16.6  14.8  18.4  21.   12.7  14.5  13.2  13.1  13.5  18.9
  20.   21.   24.7  30.8  34.9  26.6  25.3  24.7  21.2  19.3  20.   16.6
  14.4  19.4  19.7  20.5  25.   23.4  18.9  35.4  24.7  31.6  23.3  19.6
  18.7  16.   22.2  25.   33.   23.5  19.4  22.   17.4  20.9  24.2  21.7
  22.8  23.4  24.1  21.4  20.   20.8  21.2  20.3  28.   23.9  24.8  22.9
  23.9  26.6  22.5  22.2  23.6  28.7  22.6  22.   22.9  25.   20.6  28.4
  21.4  38.7  43.8  33.2  27.5  26.5  18.6  19.3  20.1  19.5  19.5  20.4
  19.8  19.4  21.7  22.8  18.8  18.7  18.5  18.3  21.2  19.2  20.4  19.3
  22.   20.3  20.5  17.3  18.8  21.4  15.7  16.2  18.   14.3  19.2  19.6
  23.   18.4  15.6  18.1  17.4  17.1  13.3  17.8  14.   14.4  13.4  15.6
  11.8  13.8  15.6  14.6  17.8  15.4  21.5  19.6  15.3  19.4  17.   15.6
  13.1  41.3  24.3  23.3  27.   50.   50.   50.   22.7  25.   50.   23.8
  23.8  22.3  17.4  19.1  23.1  23.6  22.6  29.4  23.2  24.6  29.9  37.2
  39.8  36.2  37.9  32.5  26.4  29.6  50.   32.   29.8  34.9  37.   30.5
  36.4  31.1  29.1  50.   33.3  30.3  34.6  34.9  32.9  24.1  42.3  48.5
  50.   22.6  24.4  22.5  24.4  20.   21.7  19.3  22.4  28.1  23.7  25.
  23.3  28.7  21.5  23.   26.7  21.7  27.5  30.1  44.8  50.   37.6  31.6
  46.7  31.5  24.3  31.7  41.7  48.3  29.   24.   25.1  31.5  23.7  23.3
```

The actual array is larger, so the image represents the first few values in that array.

There are also image datasets available in the scikit-learn package. Each image is of shape 8×8. Let's load it:

```
>>> digits = datasets.load_digits()
```

Print the fifth image:

```
>>> print(digits.images[4])
```

You will see the following on your Terminal:

```
>>> print(digits.images[4])
[[  0.   0.   0.   1.  11.   0.   0.   0.]
 [  0.   0.   0.   7.   8.   0.   0.   0.]
 [  0.   0.   1.  13.   6.   2.   2.   0.]
 [  0.   0.   7.  15.   0.   9.   8.   0.]
 [  0.   5.  16.  10.   0.  16.   6.   0.]
 [  0.   4.  15.  16.  13.  16.   1.   0.]
 [  0.   0.   0.   3.  15.  10.   0.   0.]
 [  0.   0.   0.   2.  16.   4.   0.   0.]]
```

As you can see, it has eight rows and eight columns.

Summary

In this chapter, we learned what AI is all about and why we need to study it. We discussed various applications and branches of AI. We understood what the Turing test is and how it's conducted. We learned how to make machines think like humans. We discussed the concept of rational agents and how they should be designed. We learned about General Problem Solver (GPS) and how to solve a problem using GPS. We discussed how to develop an intelligent agent using machine learning. We covered different types of models as well.

We discussed how to install Python 3 on various operating systems. We learned how to install the necessary packages required to build AI applications. We discussed how to use the packages to load data that's available in scikit-learn. In the next chapter, we will learn about supervised learning and how to build models for classification and regression.

2
Classification and Regression Using Supervised Learning

In this chapter, we are going to learn about classification and regression of data using supervised learning techniques. By the end of this chapter, you will know about these topics:

- What is the difference between supervised and unsupervised learning?
- What is classification?
- How to preprocess data using various methods
- What is label encoding?
- How to build a logistic regression classifier
- What is Naïve Bayes classifier?
- What is a confusion matrix?
- What are Support Vector Machines and how to build a classifier based on that?
- What is linear and polynomial regression?
- How to build a linear regressor for single variable and multivariable data
- How to estimate housing prices using Support Vector Regressor

Supervised versus unsupervised learning

One of the most common ways to impart artificial intelligence into a machine is through machine learning. The world of machine learning is broadly divided into supervised and unsupervised learning. There are other divisions too, but we'll discuss those later.

Supervised learning refers to the process of building a machine learning model that is based on labeled training data. For example, let's say that we want to build a system to automatically predict the income of a person, based on various parameters such as age, education, location, and so on. To do this, we need to create a database of people with all the necessary details and label it. By doing this, we are telling our algorithm what parameters correspond to what income. Based on this mapping, the algorithm will learn how to calculate the income of a person using the parameters provided to it.

Unsupervised learning refers to the process of building a machine learning model without relying on labeled training data. In some sense, it is the opposite of what we just discussed in the previous paragraph. Since there are no labels available, you need to extract insights based on just the data given to you. For example, let's say that we want to build a system where we have to separate a set of data points into multiple groups. The tricky thing here is that we don't know exactly what the criteria of separation should be. Hence, an unsupervised learning algorithm needs to separate the given dataset into a number of groups in the best way possible.

What is classification?

In this chapter, we will discuss supervised classification techniques. The process of classification is one such technique where we classify data into a given number of classes. During classification, we arrange data into a fixed number of categories so that it can be used most effectively and efficiently.

In machine learning, classification solves the problem of identifying the category to which a new data point belongs. We build the classification model based on the training dataset containing data points and the corresponding labels. For example, let's say that we want to check whether the given image contains a person's face or not. We would build a training dataset containing classes corresponding to these two classes: `face` and `no-face`. We then train the model based on the training samples we have. This trained model is then used for inference.

A good classification system makes it easy to find and retrieve data. This is used extensively in face recognition, spam identification, recommendation engines, and so on. The algorithms for data classification will come up with the right criteria to separate the given data into the given number of classes.

We need to provide a sufficiently large number of samples so that it can generalize those criteria. If there is an insufficient number of samples, then the algorithm will overfit to the training data. This means that it won't perform well on unknown data because it fine-tuned the model too much to fit into the patterns observed in training data. This is actually a very common problem that occurs in the world of machine learning. It's good to consider this factor when you build various machine learning models.

Preprocessing data

We deal with a lot of raw data in the real world. Machine learning algorithms expect data to be formatted in a certain way before they start the training process. In order to prepare the data for ingestion by machine learning algorithms, we have to preprocess it and convert it into the right format. Let's see how to do it.

Create a new Python file and import the following packages:

```
import numpy as np
from sklearn import preprocessing
```

Let's define some sample data:

```
input_data = np.array([[5.1, -2.9, 3.3],
                       [-1.2, 7.8, -6.1],
                       [3.9, 0.4, 2.1],
                       [7.3, -9.9, -4.5]])
```

We will be talking about several different preprocessing techniques. Let's start with binarization:

- Binarization
- Mean removal
- Scaling
- Normalization

Let's take a look at each technique, starting with the first.

Binarization

This process is used when we want to convert our numerical values into boolean values. Let's use an inbuilt method to binarize input data using `2.1` as the threshold value.

Add the following lines to the same Python file:

```
# Binarize data
data_binarized =
preprocessing.Binarizer(threshold=2.1).transform(input_data)
print("\nBinarized data:\n", data_binarized)
```

If you run the code, you will see the following output:

```
Binarized data:
 [[ 1.  0.  1.]
 [ 0.  1.  0.]
 [ 1.  0.  0.]
 [ 1.  0.  0.]]
```

As we can see here, all the values above `2.1` become `1`. The remaining values become `0`.

Mean removal

Removing the mean is a common preprocessing technique used in machine learning. It's usually useful to remove the mean from our feature vector, so that each feature is centered on zero. We do this in order to remove bias from the features in our feature vector.

Add the following lines to the same Python file as in the previous section:

```
# Print mean and standard deviation
print("\nBEFORE:")
print("Mean =", input_data.mean(axis=0))
print("Std deviation =", input_data.std(axis=0))
```

The preceding line displays the mean and standard deviation of the input data. Let's remove the mean:

```
# Remove mean
data_scaled = preprocessing.scale(input_data)
print("\nAFTER:")
print("Mean =", data_scaled.mean(axis=0))
print("Std deviation =", data_scaled.std(axis=0))
```

If you run the code, you will see the following printed on your Terminal:

```
BEFORE:
Mean = [ 3.775 -1.15  -1.3  ]
Std deviation = [ 3.12039661  6.36651396  4.0620192 ]
AFTER:
Mean = [  1.11022302e-16   0.00000000e+00   2.77555756e-17]
Std deviation = [ 1.  1.  1.]
```

As seen from the values obtained, the mean value is very close to `0` and standard deviation is `1`.

Scaling

In our feature vector, the value of each feature can vary between many random values. So it becomes important to scale those features so that it is a level playing field for the machine learning algorithm to train on. We don't want any feature to be artificially large or small just because of the nature of the measurements.

Add the following line to the same Python file:

```
# Min max scaling
data_scaler_minmax = preprocessing.MinMaxScaler(feature_range=(0, 1))
data_scaled_minmax = data_scaler_minmax.fit_transform(input_data)
print("\nMin max scaled data:\n", data_scaled_minmax)
```

If you run the code, you will see the following printed on your Terminal:

```
Min max scaled data:
 [[ 0.74117647  0.39548023  1.        ]
 [ 0.          1.          0.        ]
 [ 0.6         0.5819209   0.87234043]
 [ 1.          0.          0.17021277]]
```

Each row is scaled so that the maximum value is `1` and all the other values are relative to this value.

Normalization

We use the process of normalization to modify the values in the feature vector so that we can measure them on a common scale. In machine learning, we use many different forms of normalization. Some of the most common forms of normalization aim to modify the values so that they sum up to `1`. **L1 normalization**, which refers to **Least Absolute Deviations**, works by making sure that the sum of absolute values is *1* in each row. **L2 normalization**, which refers to least squares, works by making sure that the sum of squares is *1*.

In general, L1 normalization technique is considered more robust than L2 normalization technique. L1 normalization technique is robust because it is resistant to outliers in the data. A lot of times, data tends to contain outliers and we cannot do anything about it. We want to use techniques that can safely and effectively ignore them during the calculations. If we are solving a problem where outliers are important, then maybe L2 normalization becomes a better choice.

Add the following lines to the same Python file:

```
# Normalize data
data_normalized_l1 = preprocessing.normalize(input_data, norm='l1')
data_normalized_l2 = preprocessing.normalize(input_data, norm='l2')
print("\nL1 normalized data:\n", data_normalized_l1)
print("\nL2 normalized data:\n", data_normalized_l2)
```

If you run the code, you will see the following printed on your Terminal:

```
L1 normalized data:
 [[ 0.45132743 -0.25663717  0.2920354 ]
 [-0.0794702   0.51655629 -0.40397351]
 [ 0.609375    0.0625      0.328125  ]
 [ 0.33640553 -0.4562212  -0.20737327]]
L2 normalized data:
 [[ 0.75765788 -0.43082507  0.49024922]
 [-0.12030718  0.78199664 -0.61156148]
 [ 0.87690281  0.08993875  0.47217844]
 [ 0.55734935 -0.75585734 -0.34357152]]
```

The code for this entire section is given in the `preprocessing.py` file.

Label encoding

When we perform classification, we usually deal with a lot of labels. These labels can be in the form of words, numbers, or something else. The machine learning functions in **sklearn** expect them to be numbers. So if they are already numbers, then we can use them directly to start training. But this is not usually the case.

In the real world, labels are in the form of words, because words are human readable. We label our training data with words so that the mapping can be tracked. To convert word labels into numbers, we need to use a label encoder. Label encoding refers to the process of transforming the word labels into numerical form. This enables the algorithms to operate on our data.

Create a new Python file and import the following packages:

```
import numpy as np
from sklearn import preprocessing
```

Define some sample labels:

```
# Sample input labels
input_labels = ['red', 'black', 'red', 'green', 'black', 'yellow', 'white']
```

Create the label encoder object and train it:

```
# Create label encoder and fit the labels
encoder = preprocessing.LabelEncoder()
encoder.fit(input_labels)
```

Print the mapping between words and numbers:

```
# Print the mapping
print("\nLabel mapping:")
for i, item in enumerate(encoder.classes_):
    print(item, '-->', i)
```

Let's encode a set of randomly ordered labels to see how it performs:

```
# Encode a set of labels using the encoder
test_labels = ['green', 'red', 'black']
encoded_values = encoder.transform(test_labels)
print("\nLabels =", test_labels)
print("Encoded values =", list(encoded_values))
```

Let's decode a random set of numbers:

```
# Decode a set of values using the encoder
encoded_values = [3, 0, 4, 1]
decoded_list = encoder.inverse_transform(encoded_values)
print("\nEncoded values =", encoded_values)
print("Decoded labels =", list(decoded_list))
```

If you run the code, you will see the following output:

```
Label mapping:
black --> 0
green --> 1
red --> 2
white --> 3
yellow --> 4

Labels = ['green', 'red', 'black']
Encoded values = [1, 2, 0]

Encoded values = [3, 0, 4, 1]
Decoded labels = ['white', 'black', 'yellow', 'green']
```

You can check the mapping to see that the encoding and decoding steps are correct. The code for this section is given in the `label_encoder.py` file.

Logistic Regression classifier

Logistic regression is a technique that is used to explain the relationship between input variables and output variables. The input variables are assumed to be independent and the output variable is referred to as the dependent variable. The dependent variable can take only a fixed set of values. These values correspond to the classes of the classification problem.

Our goal is to identify the relationship between the independent variables and the dependent variables by estimating the probabilities using a logistic function. This logistic function is a **sigmoid curve** that's used to build the function with various parameters. It is very closely related to generalized linear model analysis, where we try to fit a line to a bunch of points to minimize the error. Instead of using linear regression, we use logistic regression. Logistic regression by itself is actually not a classification technique, but we use it in this way so as to facilitate classification. It is used very commonly in machine learning because of its simplicity. Let's see how to build a classifier using logistic regression. Make sure you have `Tkinter` package installed on your system before you proceed. If you don't, you can find it at: `https://docs.python.org/2/library/tkinter.html`.

Create a new Python file and import the following packages. We will be importing a function from the file `utilities.py`. We will be looking into that function very soon. But for now, let's import it:

```
import numpy as np
from sklearn import linear_model
import matplotlib.pyplot as plt

from utilities import visualize_classifier
```

Define sample input data with two-dimensional vectors and corresponding labels:

```
# Define sample input data
X = np.array([[3.1, 7.2], [4, 6.7], [2.9, 8], [5.1, 4.5], [6, 5], [5.6, 5],
[3.3, 0.4], [3.9, 0.9], [2.8, 1], [0.5, 3.4], [1, 4], [0.6, 4.9]])
y = np.array([0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3])
```

We will train the classifier using this labeled data. Now create the logistic regression classifier object:

```
# Create the logistic regression classifier
classifier = linear_model.LogisticRegression(solver='liblinear', C=1)
```

Train the classifier using the data that we defined earlier:

```
# Train the classifier
classifier.fit(X, y)
```

Visualize the performance of the classifier by looking at the boundaries of the classes:

```
# Visualize the performance of the classifier
visualize_classifier(classifier, X, y)
```

We need to define this function before we can use it. We will be using this multiple times in this chapter, so it's better to define it in a separate file and import the function. This function is given in the `utilities.py` file provided to you.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
```

Create the function definition by taking the classifier object, input data, and labels as input parameters:

```
def visualize_classifier(classifier, X, y):
    # Define the minimum and maximum values for X and Y
    # that will be used in the mesh grid
    min_x, max_x = X[:, 0].min() - 1.0, X[:, 0].max() + 1.0
    min_y, max_y = X[:, 1].min() - 1.0, X[:, 1].max() + 1.0
```

We also defined the minimum and maximum values of `X` and `Y` directions that will be used in our mesh grid. This grid is basically a set of values that is used to evaluate the function, so that we can visualize the boundaries of the classes. Define the step size for the grid and create it using the minimum and maximum values:

```
    # Define the step size to use in plotting the mesh grid
    mesh_step_size = 0.01

    # Define the mesh grid of X and Y values
    x_vals, y_vals = np.meshgrid(np.arange(min_x, max_x, mesh_step_size),
np.arange(min_y, max_y, mesh_step_size))
```

Run the classifier on all the points on the grid:

```
    # Run the classifier on the mesh grid
    output = classifier.predict(np.c_[x_vals.ravel(), y_vals.ravel()])

    # Reshape the output array
    output = output.reshape(x_vals.shape)
```

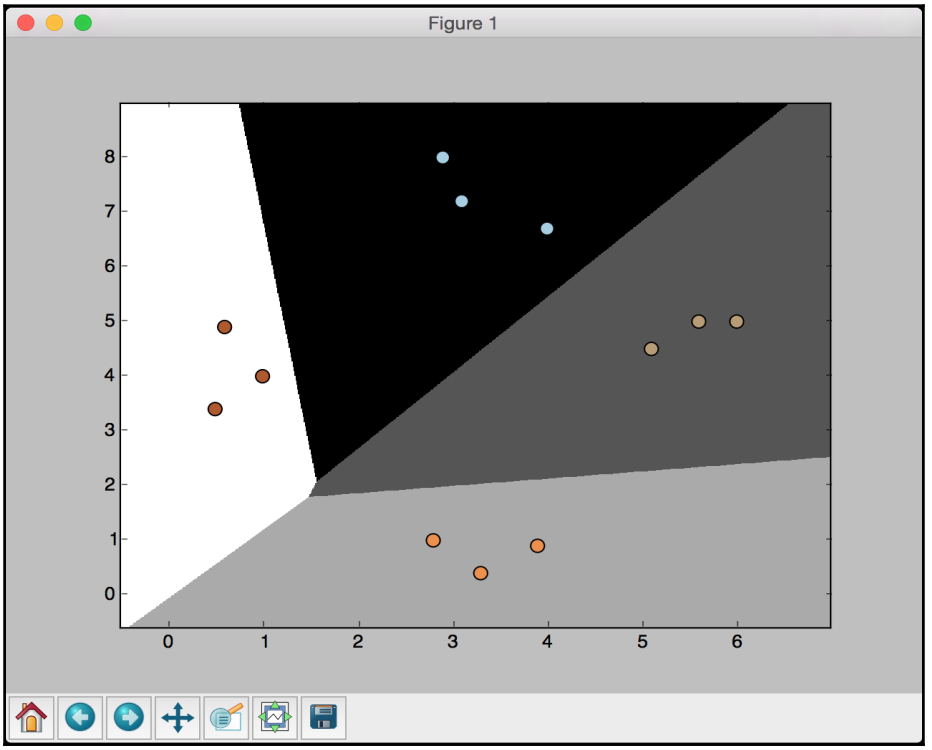
Create the figure, pick a color scheme, and overlay all the points:

```
    # Create a plot
    plt.figure()

    # Choose a color scheme for the plot
    plt.pcolormesh(x_vals, y_vals, output, cmap=plt.cm.gray)

    # Overlay the training points on the plot
    plt.scatter(X[:, 0], X[:, 1], c=y, s=75, edgecolors='black',
linewidth=1, cmap=plt.cm.Paired)
```

Specify the boundaries of the plots using the minimum and maximum values, add the tick marks, and display the figure:

```
    # Specify the boundaries of the plot
    plt.xlim(x_vals.min(), x_vals.max())
    plt.ylim(y_vals.min(), y_vals.max())

    # Specify the ticks on the X and Y axes
    plt.xticks((np.arange(int(X[:, 0].min() - 1), int(X[:, 0].max() + 1),
1.0)))
    plt.yticks((np.arange(int(X[:, 1].min() - 1), int(X[:, 1].max() + 1),
1.0)))

    plt.show()
```

If you run the code, you will see the following screenshot:

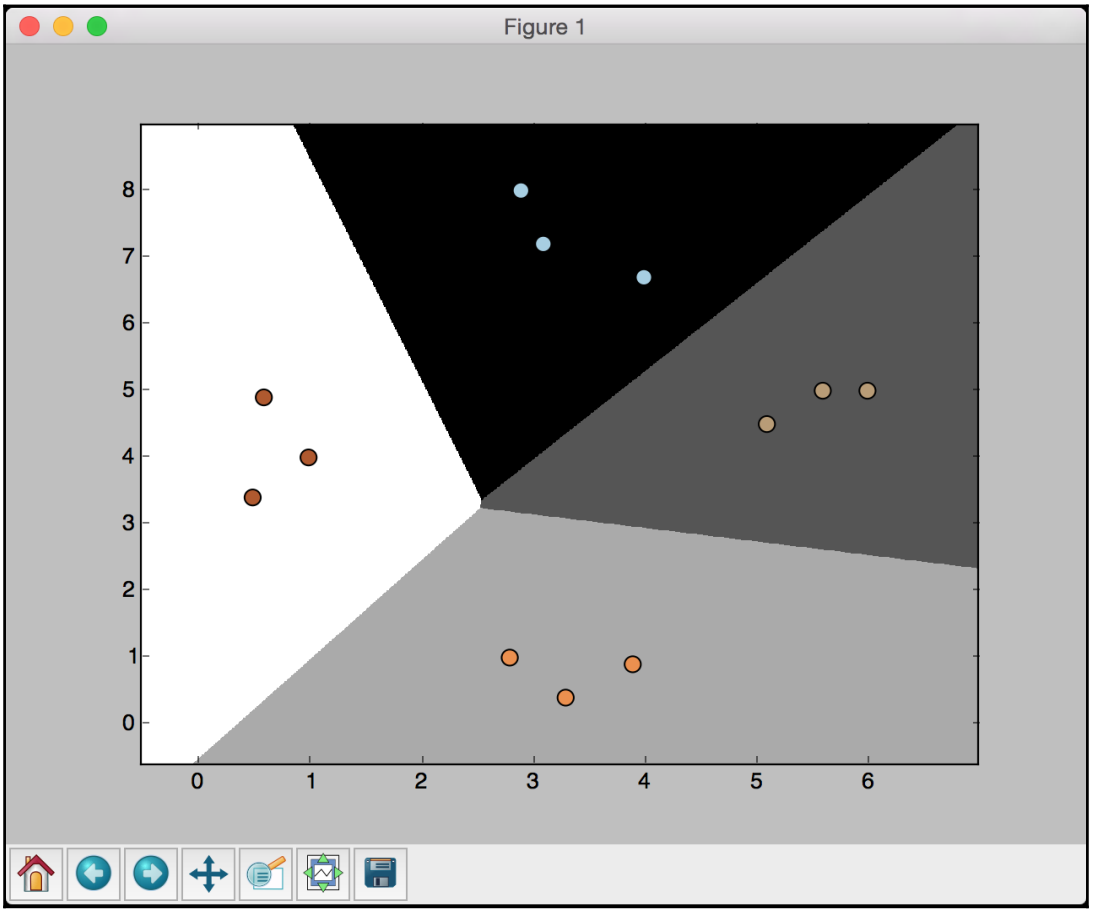


If you change the value of `C` to *100* in the following line, you will see that the boundaries become more accurate:

```
classifier = linear_model.LogisticRegression(solver='liblinear', C=100)
```

The reason is that `C` imposes a certain penalty on misclassification, so the algorithm customizes more to the training data. You should be careful with this parameter, because if you increase it by a lot, it will overfit to the training data and it won't generalize well.

If you run the code with `C` set to 100, you will see the following screenshot:



If you compare with the earlier figure, you will see that the boundaries are now better. The code for this section is given in the `logistic_regression.py` file.

Naïve Bayes classifier

Naïve Bayes is a technique used to build classifiers using Bayes theorem. Bayes theorem describes the probability of an event occurring based on different conditions that are related to this event. We build a Naïve Bayes classifier by assigning class labels to problem instances. These problem instances are represented as vectors of feature values. The assumption here is that the value of any given feature is independent of the value of any other feature. This is called the independence assumption, which is the *naïve* part of a Naïve Bayes classifier.

Given the class variable, we can just see how a given feature affects, it regardless of its affect on other features. For example, an animal may be considered a cheetah if it is spotted, has four legs, has a tail, and runs at about *70* MPH. A Naïve Bayes classifier considers that each of these features contributes independently to the outcome. The outcome refers to the probability that this animal is a cheetah. We don't concern ourselves with the correlations that may exist between skin patterns, number of legs, presence of a tail, and movement speed. Let's see how to build a Naïve Bayes classifier.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.Naïve_bayes import GaussianNB
from sklearn import cross_validation

from utilities import visualize_classifier
```

We will be using the file `data_multivar_nb.txt` as the source of data. This file contains comma separated values in each line:

```
# Input file containing data
input_file = 'data_multivar_nb.txt'
```

Let's load the data from this file:

```
# Load data from input file
data = np.loadtxt(input_file, delimiter=',')
X, y = data[:, :-1], data[:, -1]
```

Create an instance of the Naïve Bayes classifier. We will be using the Gaussian Naïve Bayes classifier here. In this type of classifier, we assume that the values associated in each class follow a Gaussian distribution:

```
# Create Naïve Bayes classifier
classifier = GaussianNB()
```

Train the classifier using the training data:

```
# Train the classifier
classifier.fit(X, y)
```

Run the classifier on the training data and predict the output:

```
# Predict the values for training data
y_pred = classifier.predict(X)
```

Let's compute the accuracy of the classifier by comparing the predicted values with the true labels, and then visualize the performance:

```
# Compute accuracy
accuracy = 100.0 * (y == y_pred).sum() / X.shape[0]
print("Accuracy of Naïve Bayes classifier =", round(accuracy, 2), "%")

# Visualize the performance of the classifier
visualize_classifier(classifier, X, y)
```

The preceding method to compute the accuracy of the classifier is not very robust. We need to perform cross validation, so that we don't use the same training data when we are testing it.

Split the data into training and testing subsets. As specified by the `test_size` parameter in the line below, we will allocate 80% for training and the remaining 20% for testing. We'll then train a Naïve Bayes classifier on this data:

```
# Split data into training and test data
X_train, X_test, y_train, y_test = cross_validation.train_test_split(X, y,
test_size=0.2, random_state=3)
classifier_new = GaussianNB()
classifier_new.fit(X_train, y_train)
y_test_pred = classifier_new.predict(X_test)
```

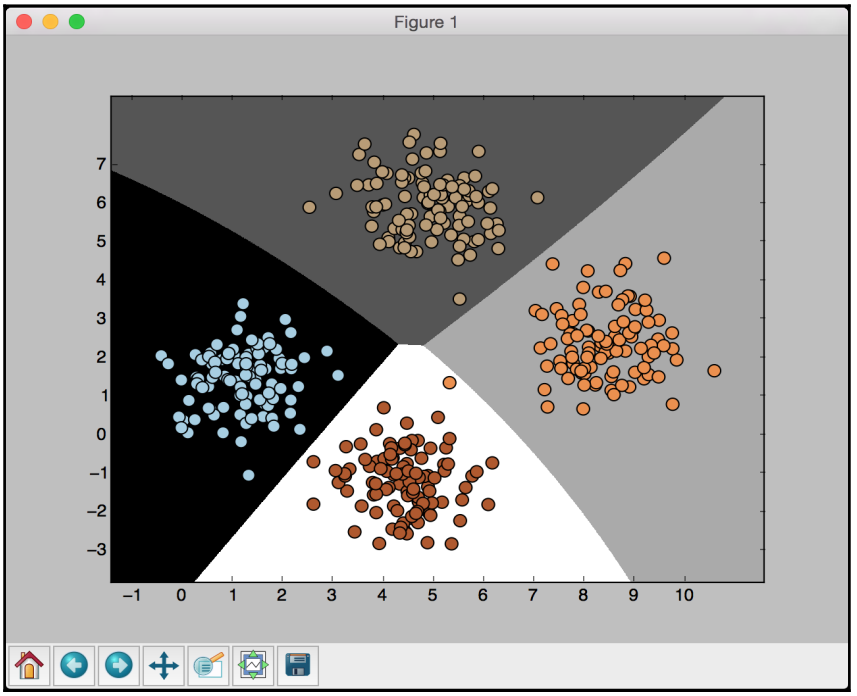
Compute the accuracy of the classifier and visualize the performance:

```
# compute accuracy of the classifier
accuracy = 100.0 * (y_test == y_test_pred).sum() / X_test.shape[0]
print("Accuracy of the new classifier =", round(accuracy, 2), "%")

# Visualize the performance of the classifier
visualize_classifier(classifier_new, X_test, y_test)
```

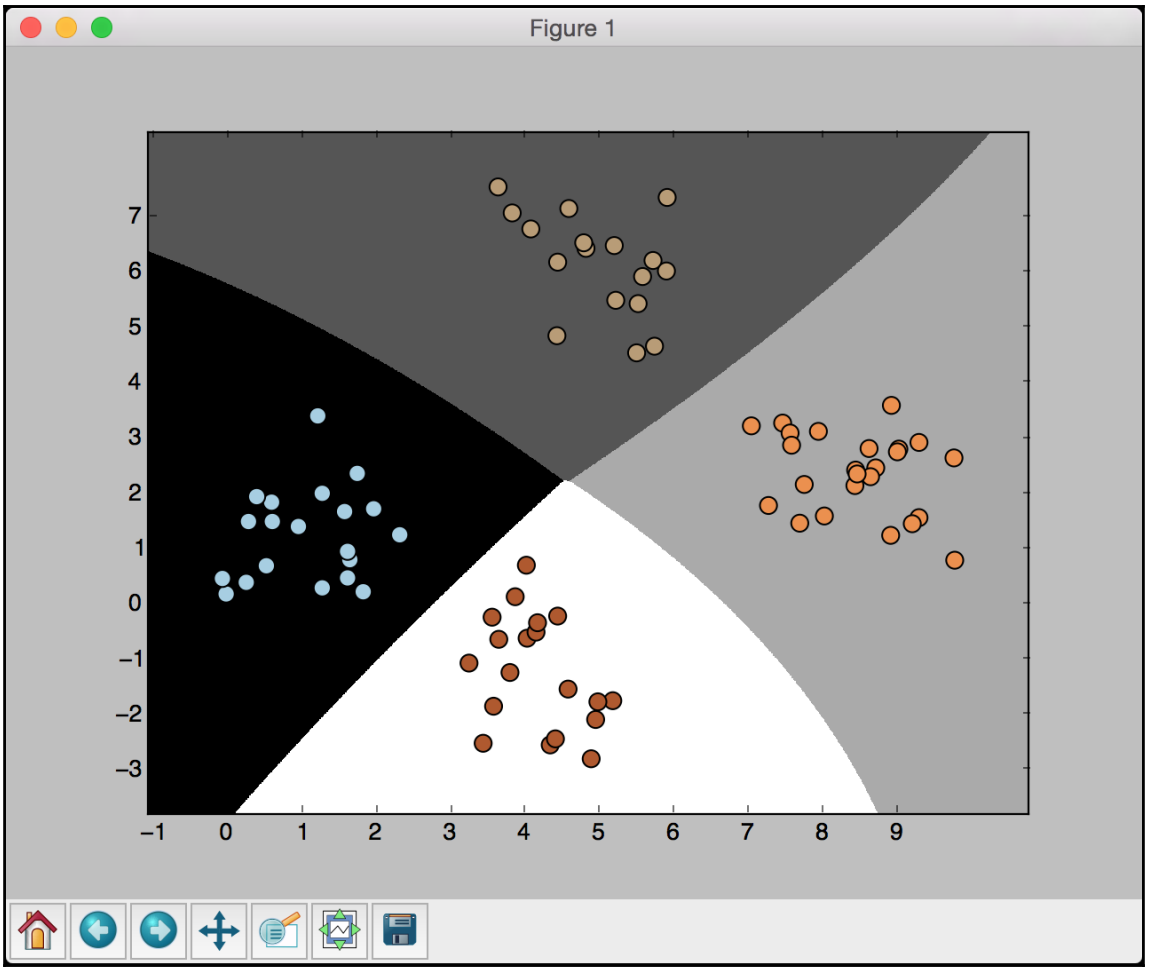
Let's use the inbuilt functions to calculate the accuracy, precision, and recall values based on threefold cross validation:

```
num_folds = 3
accuracy_values = cross_validation.cross_val_score(classifier,
        X, y, scoring='accuracy', cv=num_folds)
print("Accuracy: " + str(round(100*accuracy_values.mean(), 2)) + "%")

precision_values = cross_validation.cross_val_score(classifier,
        X, y, scoring='precision_weighted', cv=num_folds)
print("Precision: " + str(round(100*precision_values.mean(), 2)) + "%")

recall_values = cross_validation.cross_val_score(classifier,
        X, y, scoring='recall_weighted', cv=num_folds)
print("Recall: " + str(round(100*recall_values.mean(), 2)) + "%")

f1_values = cross_validation.cross_val_score(classifier,
        X, y, scoring='f1_weighted', cv=num_folds)
print("F1: " + str(round(100*f1_values.mean(), 2)) + "%")
```

If you run the code, you will see this for the first training run:



The preceding screenshot shows the boundaries obtained from the classifier. We can see that they separate the 4 clusters well and create regions with boundaries based on the distribution of the input datapoints. You will see in the following screenshot the second training run with cross validation:



You will see the following printed on your Terminal:

```
Accuracy of Naïve Bayes classifier = 99.75 %
Accuracy of the new classifier = 100.0 %
Accuracy: 99.75%
Precision: 99.76%
Recall: 99.75%
F1: 99.75%
```

The code for this section is given in the file `naive_bayes.py`.

Confusion matrix

A **Confusion matrix** is a figure or a table that is used to describe the performance of a classifier. It is usually extracted from a test dataset for which the ground truth is known. We compare each class with every other class and see how many samples are misclassified. During the construction of this table, we actually come across several key metrics that are very important in the field of machine learning. Let's consider a binary classification case where the output is either *0* or *1*:

- **True positives**: These are the samples for which we predicted *1* as the output and the ground truth is *1* too.
- **True negatives**: These are the samples for which we predicted *0* as the output and the ground truth is *0* too.
- **False positives**: These are the samples for which we predicted *1* as the output but the ground truth is *0*. This is also known as a *Type I error*.
- **False negatives:** These are the samples for which we predicted *0* as the output but the ground truth is *1*. This is also known as a *Type II error*.

Depending on the problem at hand, we may have to optimize our algorithm to reduce the false positive or the false negative rate. For example, in a biometric identification system, it is very important to avoid false positives, because the wrong people might get access to sensitive information. Let's see how to create a confusion matrix.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
```

Define some samples labels for the ground truth and the predicted output:

```
# Define sample labels
true_labels = [2, 0, 0, 2, 4, 4, 1, 0, 3, 3, 3]
pred_labels = [2, 1, 0, 2, 4, 3, 1, 0, 1, 3, 3]
```

Create the confusion matrix using the labels we just defined:

```
# Create confusion matrix
confusion_mat = confusion_matrix(true_labels, pred_labels)
```

Visualize the confusion matrix:

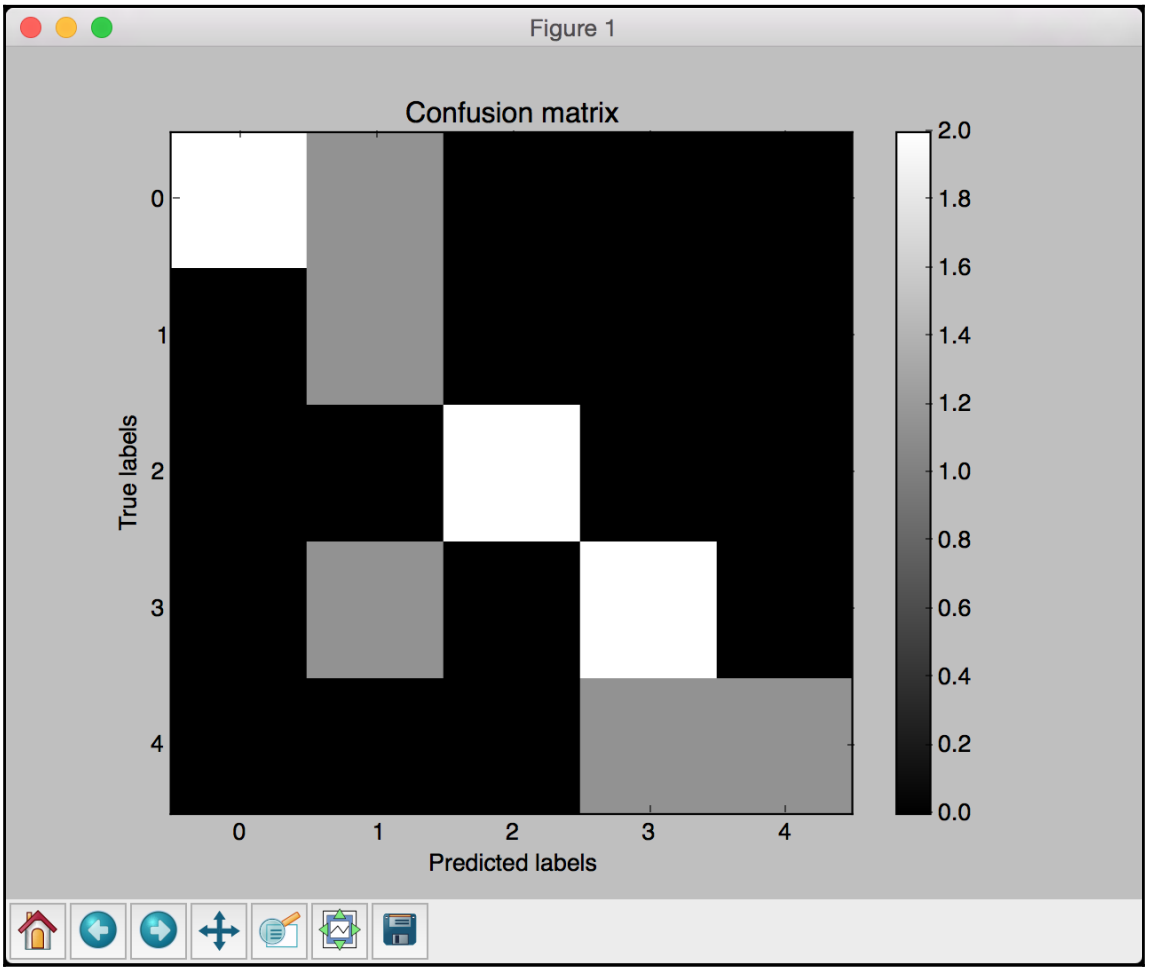
```
# Visualize confusion matrix
plt.imshow(confusion_mat, interpolation='nearest', cmap=plt.cm.gray)
plt.title('Confusion matrix')
plt.colorbar()
ticks = np.arange(5)
plt.xticks(ticks, ticks)
plt.yticks(ticks, ticks)
plt.ylabel('True labels')
plt.xlabel('Predicted labels')
plt.show()
```

In the above visualization code, the `ticks` variable refers to the number of distinct classes. In our case, we have five distinct labels.

Let's print the classification report:

```
# Classification report
targets = ['Class-0', 'Class-1', 'Class-2', 'Class-3', 'Class-4']
print('\n', classification_report(true_labels, pred_labels,
target_names=targets))
```

The classification report prints the performance for each class. If you run the code, you will see the following screenshot:



White indicates higher values, whereas black indicates lower values as seen on the color map slider. In an ideal scenario, the diagonal squares will be all white and everything else will be black. This indicates 100% accuracy.

You will see the following printed on your Terminal:

```
             precision    recall  f1-score   support

    Class-0       1.00      0.67      0.80         3
    Class-1       0.33      1.00      0.50         1
    Class-2       1.00      1.00      1.00         2
    Class-3       0.67      0.67      0.67         3
    Class-4       1.00      0.50      0.67         2

avg / total       0.85      0.73      0.75        11
```

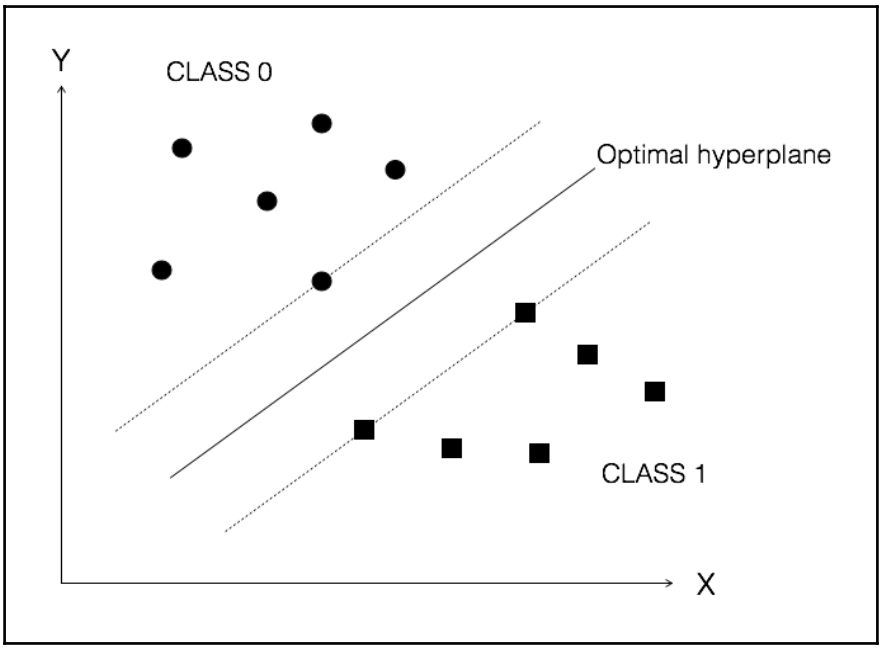
The code for this section is given in the file `confusion_matrix.py`.

Support Vector Machines

A **Support Vector Machine (SVM)** is a classifier that is defined using a separating hyperplane between the classes. This **hyperplane** is the N-dimensional version of a line. Given labeled training data and a binary classification problem, the SVM finds the optimal hyperplane that separates the training data into two classes. This can easily be extended to the problem with `N` classes.

Let's consider a two-dimensional case with two classes of points. Given that it's 2D, we only have to deal with points and lines in a 2D plane. This is easier to visualize than vectors and hyperplanes in a high-dimensional space. Of course, this is a simplified version of the SVM problem, but it is important to understand it and visualize it before we can apply it to high-dimensional data.

Consider the following figure:



There are two classes of points and we want to find the optimal hyperplane to separate the two classes. But how do we define optimal? In this picture, the solid line represents the best hyperplane. You can draw many different lines to separate the two classes of points, but this line is the best separator, because it maximizes the distance of each point from the separating line. The points on the dotted lines are called Support Vectors. The perpendicular distance between the two dotted lines is called maximum margin.

Classifying income data using Support Vector Machines

We will build a Support Vector Machine classifier to predict the income bracket of a given person based on 14 attributes. Our goal is to see where the income is higher or lower than *$50,000* per year. Hence this is a binary classification problem. We will be using the census income dataset available at `https://archive.ics.uci.edu/ml/datasets/Census+Income`. One thing to note in this dataset is that each `datapoint` is a mixture of words and numbers. We cannot use the data in its raw format, because the algorithms don't know how to deal with words. We cannot convert everything using label encoder because numerical data is valuable. Hence we need to use a combination of label encoders and raw numerical data to build an effective classifier.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import preprocessing
from sklearn.svm import LinearSVC
from sklearn.multiclass import OneVsOneClassifier
from sklearn import cross_validation
```

We will be using the file `income_data.txt` to load the data. This file contains the income details:

```
# Input file containing data
input_file = 'income_data.txt'
```

In order to load the data from the file, we need to preprocess it so that we can prepare it for classification. We will use at most 25,000 data points for each class:

```
# Read the data
X = []
y = []
count_class1 = 0
count_class2 = 0
max_datapoints = 25000
```

Open the file and start reading the lines:

```
with open(input_file, 'r') as f:
    for line in f.readlines():
        if count_class1 >= max_datapoints and count_class2 >=
max_datapoints:
            break

        if '?' in line:
            continue
```

Each line is comma separated, so we need to split it accordingly. The last element in each line represents the label. Depending on that label, we will assign it to a class:

```
        data = line[:-1].split(', ')

        if data[-1] == '<=50K' and count_class1 < max_datapoints:
            X.append(data)
            count_class1 += 1

        if data[-1] == '>50K' and count_class2 < max_datapoints:
            X.append(data)
            count_class2 += 1
```

Convert the list into a `numpy` array so that we can give it as an input to the `sklearn` function:

```
# Convert to numpy array
X = np.array(X)
```

If any attribute is a string, then we need to encode it. If it is a number, we can keep it as it is. Note that we will end up with multiple label encoders and we need to keep track of all of them:

```
# Convert string data to numerical data
label_encoder = []
X_encoded = np.empty(X.shape)
for i,item in enumerate(X[0]):
    if item.isdigit():
        X_encoded[:, i] = X[:, i]
    else:
        label_encoder.append(preprocessing.LabelEncoder())
        X_encoded[:, i] = label_encoder[-1].fit_transform(X[:, i])

X = X_encoded[:, :-1].astype(int)
y = X_encoded[:, -1].astype(int)
```

Create the `SVM` classifier with a linear kernel:

```
# Create SVM classifier
classifier = OneVsOneClassifier(LinearSVC(random_state=0))
```

Train the classifier:

```
# Train the classifier
classifier.fit(X, y)
```

Perform cross validation using an 80/20 split for training and testing, and then predict the output for training data:

```
# Cross validation
X_train, X_test, y_train, y_test = cross_validation.train_test_split(X, y,
test_size=0.2, random_state=5)
classifier = OneVsOneClassifier(LinearSVC(random_state=0))
classifier.fit(X_train, y_train)
y_test_pred = classifier.predict(X_test)
```

Compute the F1 score for the classifier:

```
# Compute the F1 score of the SVM classifier
f1 = cross_validation.cross_val_score(classifier, X, y,
scoring='f1_weighted', cv=3)
print("F1 score: " + str(round(100*f1.mean(), 2)) + "%")
```

Now that the classifier is ready, let's see how to take a random input data point and predict the output. Let's define one such data point:

```
# Predict output for a test datapoint
input_data = ['37', 'Private', '215646', 'HS-grad', '9', 'Never-married',
'Handlers-cleaners', 'Not-in-family', 'White', 'Male', '0', '0', '40',
'United-States']
```

Before we can perform prediction, we need to encode this data point using the label encoders we created earlier:

```
# Encode test datapoint
input_data_encoded = [-1] * len(input_data)
count = 0
for i, item in enumerate(input_data):
    if item.isdigit():
        input_data_encoded[i] = int(input_data[i])
    else:
        input_data_encoded[i] =
int(label_encoder[count].transform(input_data[i]))
        count += 1
```

```
input_data_encoded = np.array(input_data_encoded)
```

We are now ready to predict the output using the classifier:

```
# Run classifier on encoded datapoint and print output
predicted_class = classifier.predict(input_data_encoded)
print(label_encoder[-1].inverse_transform(predicted_class)[0])
```

If you run the code, it will take a few seconds to train the classifier. Once it's done, you will see the following printed on your Terminal:

```
F1 score: 66.82%
```

You will also see the output for the test data point:

```
<=50K
```

If you check the values in that data point, you will see that it closely corresponds to the data points in the less than 50K class. You can change the performance of the classifier (F1 score, precision, or recall) by using various different kernels and trying out multiple combinations of the parameters.

The code for this section is given in the file `income_classifier.py`.

What is Regression?

Regression is the process of estimating the relationship between input and output variables. One thing to note is that the output variables are continuous-valued real numbers. Hence there are an infinite number of possibilities. This is in contrast with classification, where the number of output classes is fixed. The classes belong to a finite set of possibilities.

In regression, it is assumed that the output variables depend on the input variables, so we want to see how they are related. Consequently, the input variables are called independent variables, also known as predictors, and output variables are called dependent variables, also known as criterion variables. It is not necessary that the input variables are independent of each other. There are a lot of situations where there are correlations between input variables.

Regression analysis helps us in understanding how the value of the output variable changes when we vary some input variables while keeping other input variables fixed. In linear regression, we assume that the relationship between input and output is linear. This puts a constraint on our modeling procedure, but it's fast and efficient.

Sometimes, linear regression is not sufficient to explain the relationship between input and output. Hence we use polynomial regression, where we use a polynomial to explain the relationship between input and output. This is more computationally complex, but gives higher accuracy. Depending on the problem at hand, we use different forms of regression to extract the relationship. Regression is frequently used for prediction of prices, economics, variations, and so on.

Building a single variable regressor

Let's see how to build a single variable regression model. Create a new Python file and import the following packages:

```
import pickle

import numpy as np
from sklearn import linear_model
import sklearn.metrics as sm
import matplotlib.pyplot as plt
```

We will use the file `data_singlevar_regr.txt` provided to you. This is our source of data:

```
# Input file containing data
input_file = 'data_singlevar_regr.txt'
```

It's a comma-separated file, so we can easily load it using a one-line function call:

```
# Read data
data = np.loadtxt(input_file, delimiter=',')
X, y = data[:, :-1], data[:, -1]
```

Split it into training and testing:

```
# Train and test split
num_training = int(0.8 * len(X))
num_test = len(X) - num_training

# Training data
X_train, y_train = X[:num_training], y[:num_training]

# Test data
X_test, y_test = X[num_training:], y[num_training:]
```

Create a linear regressor object and train it using the training data:

```
# Create linear regressor object
regressor = linear_model.LinearRegression()

# Train the model using the training sets
regressor.fit(X_train, y_train)
```

Predict the output for the testing dataset using the training model:

```
# Predict the output
y_test_pred = regressor.predict(X_test)
```

Plot the output:

```
# Plot outputs
plt.scatter(X_test, y_test, color='green')
plt.plot(X_test, y_test_pred, color='black', linewidth=4)
plt.xticks(())
plt.yticks(())
plt.show()
```

Compute the performance metrics for the regressor by comparing the ground truth, which refers to the actual outputs, with the predicted outputs:

```
# Compute performance metrics
print("Linear regressor performance:")
print("Mean absolute error =", round(sm.mean_absolute_error(y_test,
y_test_pred), 2))
print("Mean squared error =", round(sm.mean_squared_error(y_test,
y_test_pred), 2))
print("Median absolute error =", round(sm.median_absolute_error(y_test,
y_test_pred), 2))
print("Explain variance score =", round(sm.explained_variance_score(y_test,
y_test_pred), 2))
print("R2 score =", round(sm.r2_score(y_test, y_test_pred), 2))
```

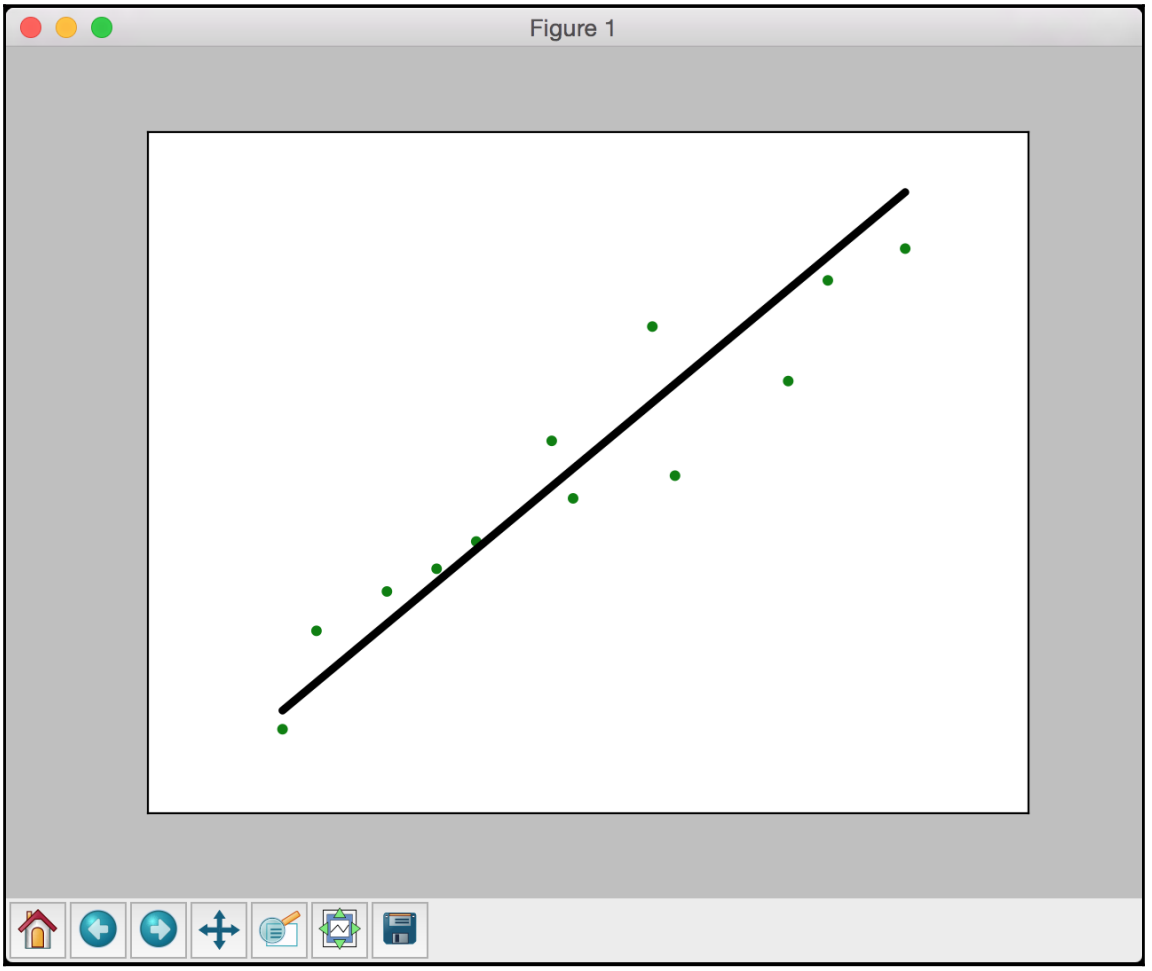
Once the model has been created, we can save it into a file so that we can use it later. Python provides a nice module called `pickle` that enables us to do this:

```
# Model persistence
output_model_file = 'model.pkl'

# Save the model
with open(output_model_file, 'wb') as f:
    pickle.dump(regressor, f)
```

Let's load the model from the file on the disk and perform prediction:

```
# Load the model
with open(output_model_file, 'rb') as f:
    regressor_model = pickle.load(f)

# Perform prediction on test data
y_test_pred_new = regressor_model.predict(X_test)
print("\nNew mean absolute error =", round(sm.mean_absolute_error(y_test,
y_test_pred_new), 2))
```

If you run the code, you will see the following screenshot:



You will see the following printed on your Terminal:

```
Linear regressor performance:
Mean absolute error = 0.59
Mean squared error = 0.49
Median absolute error = 0.51
Explain variance score = 0.86
R2 score = 0.86
New mean absolute error = 0.59
```

The code for this section is given in the file `regressor_singlevar.py`.

Building a multivariable regressor

In the previous section, we discussed how to build a regression model for a single variable. In this section, we will deal with multidimensional data. Create a new Python file and import the following packages:

```
import numpy as np
from sklearn import linear_model
import sklearn.metrics as sm
from sklearn.preprocessing import PolynomialFeatures
```

We will use the file `data_multivar_regr.txt` provided to you.

```
# Input file containing data
input_file = 'data_multivar_regr.txt'
```

This is a comma-separated file, so we can load it easily with a one-line function call:

```
# Load the data from the input file
data = np.loadtxt(input_file, delimiter=',')
X, y = data[:, :-1], data[:, -1]
```

Split the data into training and testing:

```
# Split data into training and testing
num_training = int(0.8 * len(X))
num_test = len(X) - num_training

# Training data
X_train, y_train = X[:num_training], y[:num_training]

# Test data
X_test, y_test = X[num_training:], y[num_training:]
```

Create and train the linear regressor model:

```
# Create the linear regressor model
linear_regressor = linear_model.LinearRegression()

# Train the model using the training sets
linear_regressor.fit(X_train, y_train)
```

Predict the output for the test dataset:

```
# Predict the output
y_test_pred = linear_regressor.predict(X_test)
```

Print the performance metrics:

```
# Measure performance
print("Linear Regressor performance:")
print("Mean absolute error =", round(sm.mean_absolute_error(y_test,
y_test_pred), 2))
print("Mean squared error =", round(sm.mean_squared_error(y_test,
y_test_pred), 2))
print("Median absolute error =", round(sm.median_absolute_error(y_test,
y_test_pred), 2))
print("Explained variance score =",
round(sm.explained_variance_score(y_test, y_test_pred), 2))
print("R2 score =", round(sm.r2_score(y_test, y_test_pred), 2))
```

Create a polynomial regressor of degree 10. Train the regressor on the training dataset. Let's take a sample data point and see how to perform prediction. The first step is to transform it into a polynomial:

```
# Polynomial regression
polynomial = PolynomialFeatures(degree=10)
X_train_transformed = polynomial.fit_transform(X_train)
datapoint = [[7.75, 6.35, 5.56]]
poly_datapoint = polynomial.fit_transform(datapoint)
```

If you look closely, this data point is very close to the data point on line 11 in our data file, which is *[7.66, 6.29, 5.66]*. So, a good regressor should predict an output that's close to 41.35. Create a linear regressor object and perform the polynomial fit. Perform the prediction using both linear and polynomial regressors to see the difference:

```
poly_linear_model = linear_model.LinearRegression()
poly_linear_model.fit(X_train_transformed, y_train)
print("\nLinear regression:\n", linear_regressor.predict(datapoint))
print("\nPolynomial regression:\n",
poly_linear_model.predict(poly_datapoint))
```

If you run the code, you will see the following printed on your Terminal:

```
Linear Regressor performance:
Mean absolute error = 3.58
Mean squared error = 20.31
Median absolute error = 2.99
Explained variance score = 0.86
R2 score = 0.86
```

You will see the following as well:

```
Linear regression:
 [ 36.05286276]
Polynomial regression:
 [ 41.46961676]
```

As you can see, the polynomial regressor is closer to 41.35. The code for this section is given in the file `regressor_multivar.py`.

Estimating housing prices using a Support Vector Regressor

Let's see how to use the SVM concept to build a regressor to estimate the housing prices. We will use the dataset available in `sklearn` where each data point is define, by 13 attributes. Our goal is to estimate the housing prices based on these attributes.

Create a new Python file and import the following packages:

```
import numpy as np
from sklearn import datasets
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, explained_variance_score
from sklearn.utils import shuffle
```

Load the housing dataset:

```
# Load housing data
data = datasets.load_boston()
```

Let's shuffle the data so that we don't bias our analysis:

```
# Shuffle the data
X, y = shuffle(data.data, data.target, random_state=7)
```

Split the dataset into training and testing in an 80/20 format:

```
# Split the data into training and testing datasets
num_training = int(0.8 * len(X))
X_train, y_train = X[:num_training], y[:num_training]
X_test, y_test = X[num_training:], y[num_training:]
```

Create and train the Support Vector Regressor using a linear kernel. The C parameter represents the penalty for training error. If you increase the value of C, the model will fine-tune it more to fit the training data. But this might lead to overfitting and cause it to lose its generality. The epsilon parameter specifies a threshold; there is no penalty for training error if the predicted value is within this distance from the actual value:

```
# Create Support Vector Regression model
sv_regressor = SVR(kernel='linear', C=1.0, epsilon=0.1)

# Train Support Vector Regressor
sv_regressor.fit(X_train, y_train)
```

Evaluate the performance of the regressor and print the metrics:

```
# Evaluate performance of Support Vector Regressor
y_test_pred = sv_regressor.predict(X_test)
mse = mean_squared_error(y_test, y_test_pred)
evs = explained_variance_score(y_test, y_test_pred)
print("\n#### Performance ####")
print("Mean squared error =", round(mse, 2))
print("Explained variance score =", round(evs, 2))
```

Let's take a test data point and perform prediction:

```
# Test the regressor on test datapoint
test_data = [3.7, 0, 18.4, 1, 0.87, 5.95, 91, 2.5052, 26, 666, 20.2,
351.34, 15.27]
print("\nPredicted price:", sv_regressor.predict([test_data])[0])
```

If you run the code, you will see the following printed on the Terminal:

```
#### Performance ####
Mean squared error = 15.41
Explained variance score = 0.82
Predicted price: 18.5217801073
```

The code for this section is given in the file `house_prices.py`.

Summary

In this chapter, we learned the difference between supervised and unsupervised learning. We discussed the data classification problem and how to solve it. We understood how to preprocess data using various methods. We also learned about label encoding and how to build a label encoder. We discussed logistic regression and built a logistic regression classifier. We understood what Naïve Bayes classifier is and learned how to build it. We also learned how to build a confusion matrix.

We discussed Support Vector Machines and understood how to build a classifier based on that. We learned about regression and understood how to use linear and polynomial regression for single and multivariable data. We then used Support Vector Regressor to estimate the housing prices using input attributes.

In the next chapter, we will learn about predictive analytics and how to build a predictive engine using ensemble learning.

3
Predictive Analytics with Ensemble Learning

In this chapter, we are going to learn about Ensemble Learning and how to use it for predictive analytics. By the end of this chapter, you will know these topics:

- Building learning models with Ensemble Learning
- What are Decision Trees and how to build a Decision Trees classifier
- What are Random Forests and Extremely Random Forests, and how to build classifiers based on them
- Estimating the confidence measure of the predictions
- Dealing with class imbalance
- Finding optimal training parameters using grid search
- Computing relative feature importance
- Predicting traffic using Extremely Random Forests regressor

What is Ensemble Learning?

Ensemble Learning refers to the process of building multiple models and then combining them in a way that can produce better results than individual models. These individual models can be classifiers, regressors, or anything else that models data in some way. Ensemble learning is used extensively across multiple fields including data classification, predictive modeling, anomaly detection, and so on.

Why do we need ensemble learning in the first place? In order to understand this, let's take a real-life example. You want to buy a new TV, but you don't know what the latest models are. Your goal is to get the best value for your money, but you don't have enough knowledge on this topic to make an informed decision. When you have to make a decision about something like this, you go around and try to get the opinions of multiple experts in the domain. This will help you make the best decision. More often than not, instead of just relying on a single opinion, you tend to make a final decision by combining the individual decisions of those experts. The reason we do that is because we want to minimize the possibility of a wrong or suboptimal decision.

Building learning models with Ensemble Learning

When we select a model, the most commonly used procedure is to choose the one with the smallest error on the training dataset. The problem with this approach is that it will not always work. The model might get biased or overfit the training data. Even when we compute the model using cross validation, it can perform poorly on unknown data.

One of the main reasons ensemble learning is so effective is because it reduces the overall risk of making a poor model selection. This enables it to train in a diverse manner and then perform well on unknown data. When we build a model using ensemble learning, the individual models need to exhibit some diversity. This would allow them to capture various nuances in our data; hence the overall model becomes more accurate.

The diversity is achieved by using different training parameters for each individual model. This allows individual models to generate different decision boundaries for training data. This means that each model will use different rules to make an inference, which is a powerful way of validating the final result. If there is agreement among the models, then we know that the output is correct.

What are Decision Trees?

A **Decision Tree** is a structure that allows us to split the dataset into branches and then make simple decisions at each level. This will allow us to arrive at the final decision by walking down the tree. Decision Trees are produced by training algorithms, which identify how we can split the data in the best possible way.

Any decision process starts at the root node at the top of the tree. Each node in the tree is basically a decision rule. Algorithms construct these rules based on the relationship between the input data and the target labels in the training data. The values in the input data are utilized to estimate the value for the output.

Now that we understand basic concept of Decision Trees, the next thing is to understand how the trees are automatically constructed. We need algorithms that can construct the optimal tree based on our data. In order to understand it, we need to understand the concept of entropy. In this context, entropy refers to information entropy and not thermodynamic entropy. Entropy is basically a measure of uncertainty. One of the main goals of a decision tree is to reduce uncertainty as we move from the root node towards the leaf nodes. When we see an unknown data point, we are completely uncertain about the output. By the time we reach the leaf node, we are certain about the output. This means that we need to construct the decision tree in a way that will reduce the uncertainty at each level. This implies that we need to reduce the entropy as we progress down the tree.

You can learn more about this at: `https://prateekvjoshi.com/2016/03/22/how-are-decision-trees-constructed-in-machine-learning`.

Building a Decision Tree classifier

Let's see how to build a classifier using Decision Trees in Python. Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report
from sklearn import cross_validation
from sklearn.tree import DecisionTreeClassifier

from utilities import visualize_classifier
```

We will be using the data in the `data_decision_trees.txt` file that's provided to you. In this file, each line contains comma-separated values. The first two values correspond to the input data and the last value corresponds to the target label. Let's load the data from that file:

```
# Load input data
input_file = 'data_decision_trees.txt'
data = np.loadtxt(input_file, delimiter=',')
X, y = data[:, :-1], data[:, -1]
```

Separate the input data into two separate classes based on the labels:

```
# Separate input data into two classes based on labels
class_0 = np.array(X[y==0])
class_1 = np.array(X[y==1])
```

Let's visualize the input data using a scatter plot:

```
# Visualize input data
plt.figure()
plt.scatter(class_0[:, 0], class_0[:, 1], s=75, facecolors='black',
            edgecolors='black', linewidth=1, marker='x')
plt.scatter(class_1[:, 0], class_1[:, 1], s=75, facecolors='white',
            edgecolors='black', linewidth=1, marker='o')
plt.title('Input data')
```

We need to split the data into training and testing datasets:

```
# Split data into training and testing datasets
X_train, X_test, y_train, y_test = cross_validation.train_test_split(
            X, y, test_size=0.25, random_state=5)
```

Create, build, and visualize a decision tree classifier based on the training dataset. The `random_state` parameter refers to the seed used by the random number generator required for the initialization of the decision tree classification algorithm. The `max_depth` parameter refers to the maximum depth of the tree that we want to construct:

```
# Decision Trees classifier
params = {'random_state': 0, 'max_depth': 4}
classifier = DecisionTreeClassifier(**params)
classifier.fit(X_train, y_train)
visualize_classifier(classifier, X_train, y_train, 'Training dataset')
```

Compute the output of the classifier on the test dataset and visualize it:

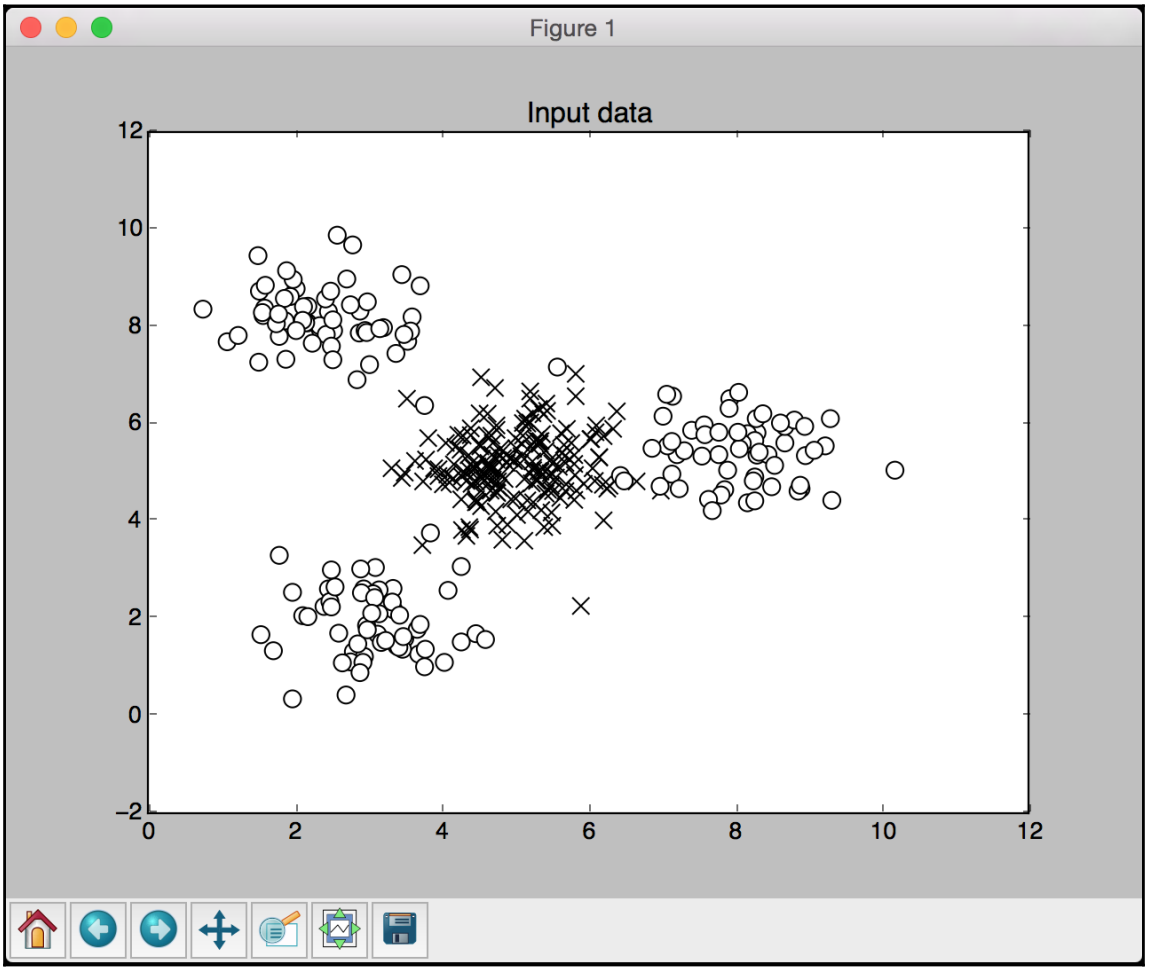
```
y_test_pred = classifier.predict(X_test)
visualize_classifier(classifier, X_test, y_test, 'Test dataset')
```

Evaluate the performance of the classifier by printing the classification report:

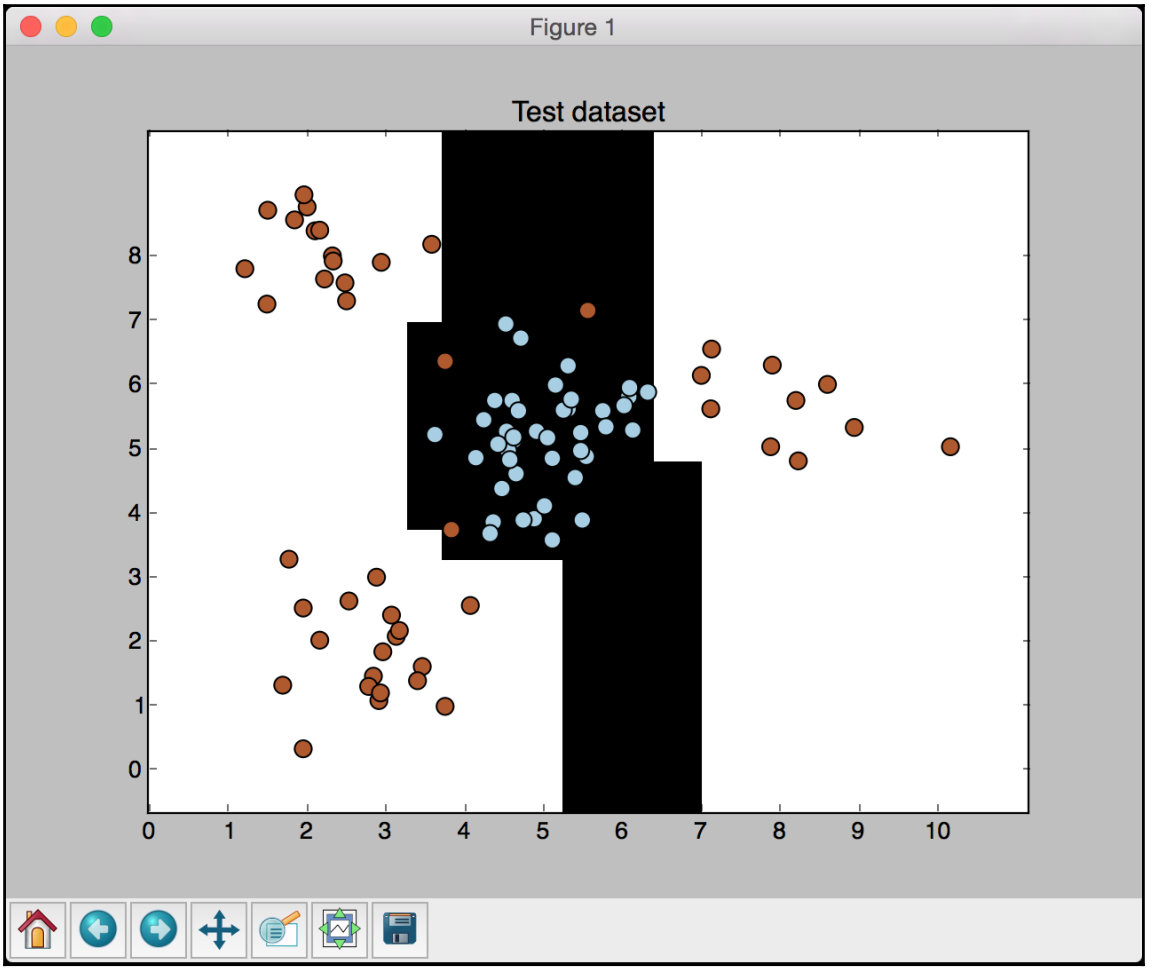
```
# Evaluate classifier performance
class_names = ['Class-0', 'Class-1']
print("\n" + "#"*40)
print("\nClassifier performance on training dataset\n")
print(classification_report(y_train, classifier.predict(X_train),
target_names=class_names))
print("#"*40 + "\n")
```

```
print("#"*40)
print("\nClassifier performance on test dataset\n")
print(classification_report(y_test, y_test_pred, target_names=class_names))
print("#"*40 + "\n")

plt.show()
```

The full code is given in the `decision_trees.py` file. If you run the code, you will see a few figures. The first screenshot is the visualization of input data:



The second screenshot shows the classifier boundaries on the test dataset:



You will see the following printed on your Terminal:

```
########################################

Classifier performance on training dataset

             precision    recall  f1-score   support

    Class-0       0.99      1.00      1.00       137
    Class-1       1.00      0.99      1.00       133

avg / total       1.00      1.00      1.00       270

########################################

########################################

Classifier performance on test dataset

             precision    recall  f1-score   support

    Class-0       0.93      1.00      0.97        43
    Class-1       1.00      0.94      0.97        47

avg / total       0.97      0.97      0.97        90

########################################
```

The performance of a classifier is characterized by `precision`, `recall`, and `f1-scores`. Precision refers to the accuracy of the classification and recall refers to the number of items that were retrieved as a percentage of the overall number of items that were supposed to be retrieved. A good classifier will have high precision and high recall, but it is usually a trade-off between the two. Hence we have `f1-score` to characterize that. F1 score is the harmonic mean of precision and recall, which gives it a good balance between precision and recall values.

What are Random Forests and Extremely Random Forests?

A **Random Forest** is a particular instance of ensemble learning where individual models are constructed using Decision Trees. This ensemble of Decision Trees is then used to predict the output value. We use a random subset of training data to construct each Decision Tree. This will ensure diversity among various decision trees. In the first section, we discussed that one of the most important things in ensemble learning is to ensure that there's diversity among individual models.

One of the best things about Random Forests is that they do not overfit. As we know, overfitting is a problem that we encounter frequently in machine learning. By constructing a diverse set of Decision Trees using various random subsets, we ensure that the model does not overfit the training data. During the construction of the tree, the nodes are split successively and the best thresholds are chosen to reduce the entropy at each level. This split doesn't consider all the features in the input dataset. Instead, it chooses the best split among the random subset of the features that is under consideration. Adding this randomness tends to increase the bias of the random forest, but the variance decreases because of averaging. Hence, we end up with a robust model.

Extremely Random Forests take randomness to the next level. Along with taking a random subset of features, the thresholds are chosen at random too. These randomly generated thresholds are chosen as the splitting rules, which reduce the variance of the model even further. Hence the decision boundaries obtained using Extremely Random Forests tend to be smoother than the ones obtained using Random Forests.

Building Random Forest and Extremely Random Forest classifiers

Let's see how to build a classifier based on Random Forests and Extremely Random Forests. The way to construct both classifiers is very similar, so we will use an input flag to specify which classifier needs to be built.

Create a new Python file and import the following packages:

```
import argparse

import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report
from sklearn import cross_validation
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn import cross_validation
from sklearn.metrics import classification_report

from utilities import visualize_classifier
```

Define an argument parser for Python so that we can take the classifier type as an input parameter. Depending on this parameter, we can construct a Random Forest classifier or an Extremely Random forest classifier:

```
# Argument parser
def build_arg_parser():
      parser = argparse.ArgumentParser(description='Classify data using \
                  Ensemble Learning techniques')
      parser.add_argument('--classifier-type', dest='classifier_type',
                  required=True, choices=['rf', 'erf'], help="Type of
classifier
                              \to use; can be either 'rf' or 'erf'")
      return parser
```

Define the main function and parse the input arguments:

```
if __name__=='__main__':
      # Parse the input arguments
      args = build_arg_parser().parse_args()
      classifier_type = args.classifier_type
```

We will be using the data from the `data_random_forests.txt` file that is provided to you. Each line in this file contains comma-separated values. The first two values correspond to the input data and the last value corresponds to the target label. We have three distinct classes in this dataset. Let's load the data from that file:

```
      # Load input data
      input_file = 'data_random_forests.txt'
      data = np.loadtxt(input_file, delimiter=',')
      X, y = data[:, :-1], data[:, -1]
```

Separate the input data into three classes:

```
# Separate input data into three classes based on labels
class_0 = np.array(X[y==0])
class_1 = np.array(X[y==1])
class_2 = np.array(X[y==2])
```

Let's visualize the input data:

```
# Visualize input data
plt.figure()
plt.scatter(class_0[:, 0], class_0[:, 1], s=75, facecolors='white',
                        edgecolors='black', linewidth=1, marker='s')
plt.scatter(class_1[:, 0], class_1[:, 1], s=75, facecolors='white',
                        edgecolors='black', linewidth=1, marker='o')
plt.scatter(class_2[:, 0], class_2[:, 1], s=75, facecolors='white',
                        edgecolors='black', linewidth=1, marker='^')
plt.title('Input data')
```

Split the data into training and testing datasets:

```
# Split data into training and testing datasets
X_train, X_test, y_train, y_test = cross_validation.train_test_split(
            X, y, test_size=0.25, random_state=5)
```

Define the parameters to be used when we construct the classifier. The `n_estimators` parameter refers to the number of trees that will be constructed. The `max_depth` parameter refers to the maximum number of levels in each tree. The `random_state` parameter refers to the seed value of the random number generator needed to initialize the random forest classifier algorithm:

```
# Ensemble Learning classifier
params = {'n_estimators': 100, 'max_depth': 4, 'random_state': 0}
```

Depending on the input parameter, we either construct a random forest classifier or an extremely random forest classifier:

```
if classifier_type == 'rf':
      classifier = RandomForestClassifier(**params)
else:
      classifier = ExtraTreesClassifier(**params)
```

Train and visualize the classifier:

```
classifier.fit(X_train, y_train)
visualize_classifier(classifier, X_train, y_train, 'Training
dataset')
```

Compute the output based on the test dataset and visualize it:

```
        y_test_pred = classifier.predict(X_test)
        visualize_classifier(classifier, X_test, y_test, 'Test dataset')
```

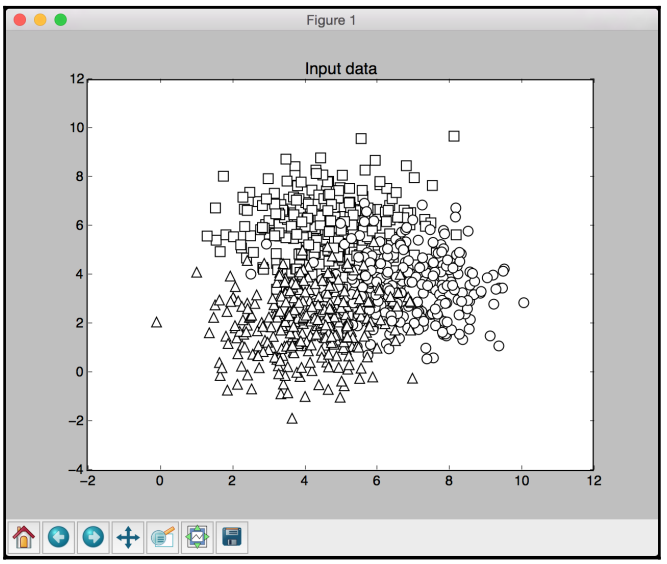
Evaluate the performance of the classifier by printing the classification report:

```
        # Evaluate classifier performance
        class_names = ['Class-0', 'Class-1', 'Class-2']
        print("\n" + "#"*40)
        print("\nClassifier performance on training dataset\n")
        print(classification_report(y_train, classifier.predict(X_train),
target_names=class_names))
        print("#"*40 + "\n")

        print("#"*40)
        print("\nClassifier performance on test dataset\n")
        print(classification_report(y_test, y_test_pred,
target_names=class_names))
        print("#"*40 + "\n")
```

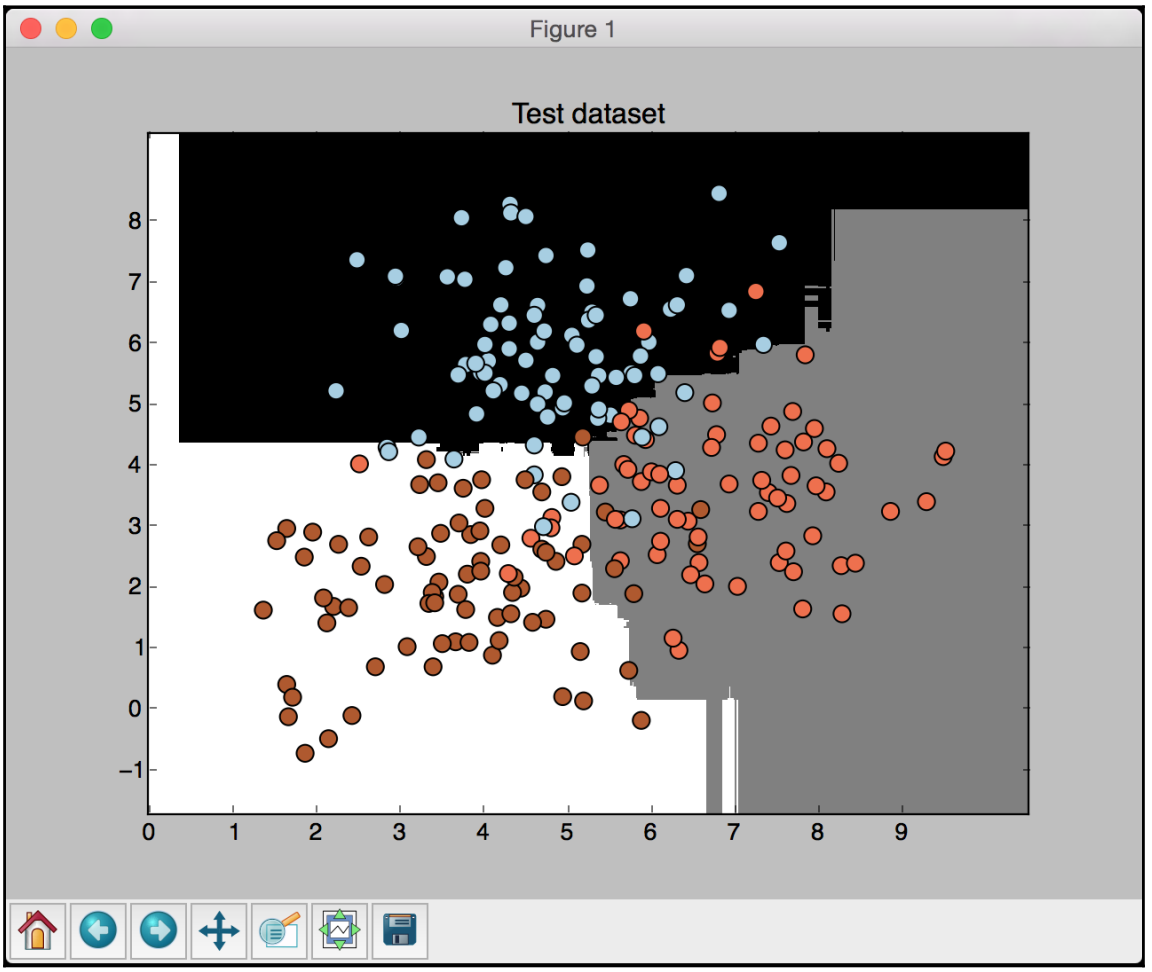
The full code is given in the `random_forests.py` file. Let's run the code with the Random Forest classifier using the `rf` flag in the input argument. Run the following command on your Terminal:

```
$ python3 random_forests.py --classifier-type rf
```

You will see a few figures pop up. The first screenshot is the input data:



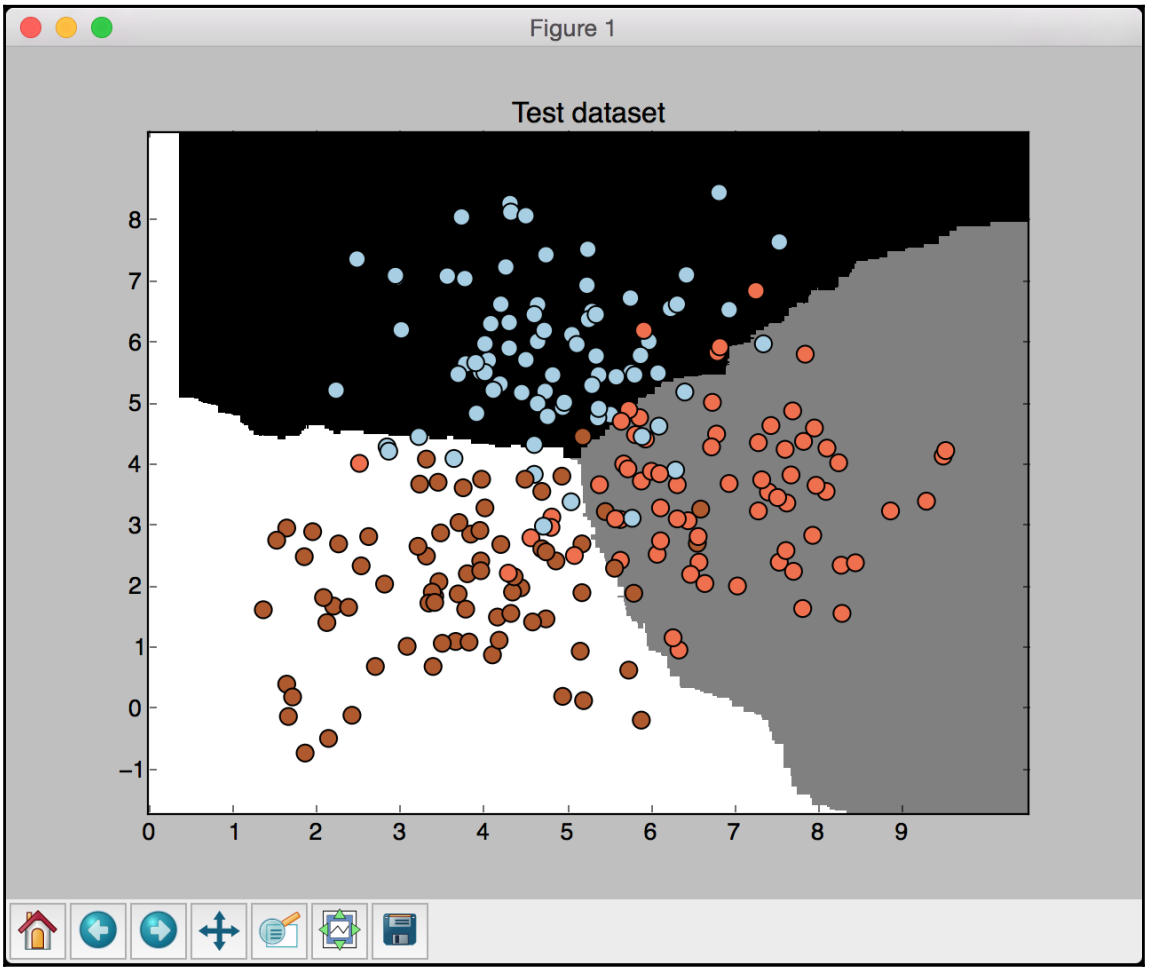
In the preceding screenshot, the three classes are being represented by squares, circles, and triangles. We see that there is a lot of overlap between classes, but that should be fine for now. The second screenshot shows the classifier boundaries:



Now let's run the code with the Extremely Random Forest classifier by using the `erf` flag in the input argument. Run the following command on your Terminal:

```
$ python3 random_forests.py --classifier-type erf
```

You will see a few figures pop up. We already know what the input data looks like. The second screenshot shows the classifier boundaries:



If you compare the preceding screenshot with the boundaries obtained from Random Forest classifier, you will see that these boundaries are smoother. The reason is that Extremely Random Forests have more freedom during the training process to come up with good Decision Trees, hence they usually produce better boundaries.

Estimating the confidence measure of the predictions

If you observe the outputs obtained on the terminal, you will see that the probabilities are printed for each data point. These probabilities are used to measure the confidence values for each class. Estimating the confidence values is an important task in machine learning. In the same python file, add the following line to define an array of test data points:

```
        # Compute confidence
        test_datapoints = np.array([[5, 5], [3, 6], [6, 4], [7, 2], [4, 4],
[5, 2]])
```

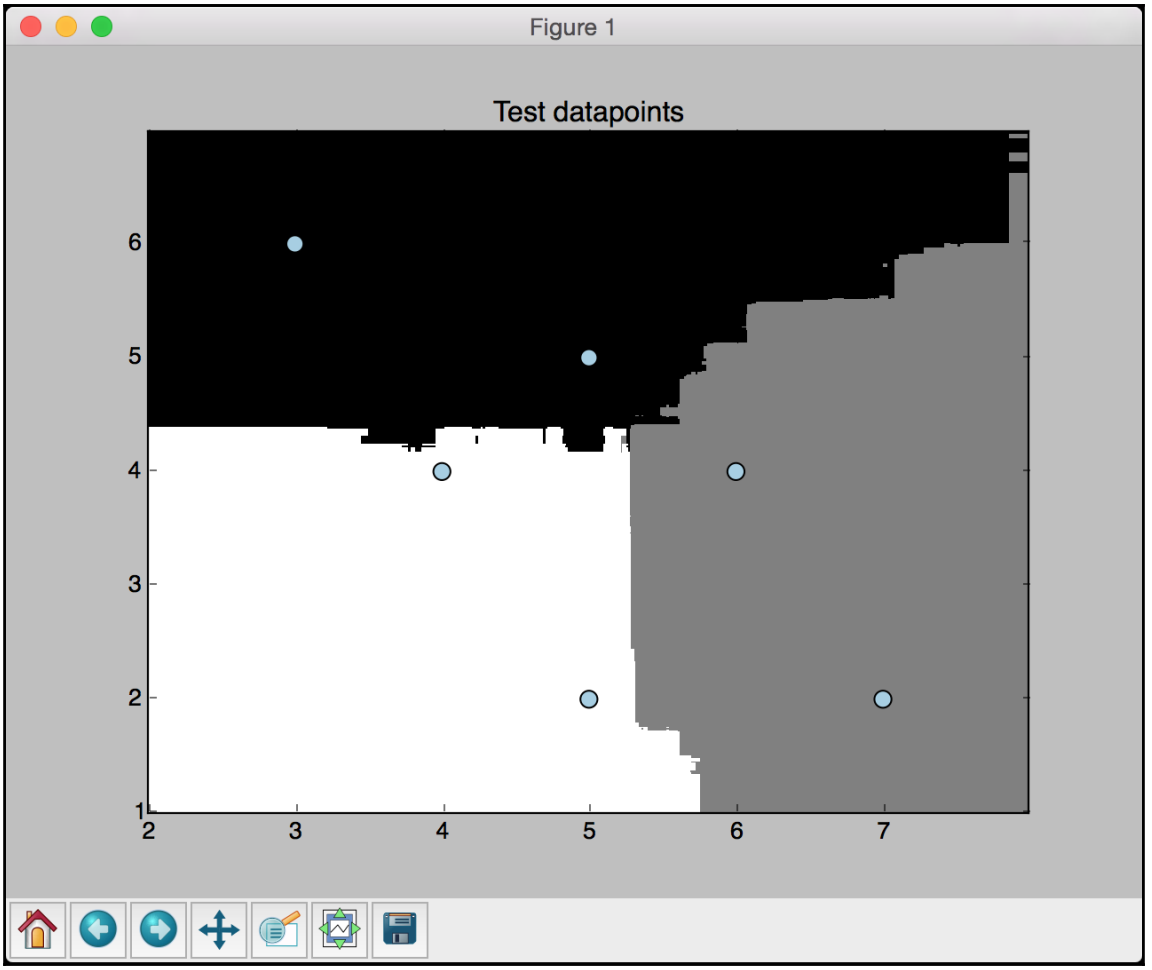
The classifier object has an inbuilt method to compute the confidence measure. Let's classify each point and compute the confidence values:

```
        print("\nConfidence measure:")
        for datapoint in test_datapoints:
              probabilities = classifier.predict_proba([datapoint])[0]
              predicted_class = 'Class-' + str(np.argmax(probabilities))
              print('\nDatapoint:', datapoint)
              print('Predicted class:', predicted_class)
```

Visualize the test data points based on classifier boundaries:

```
        # Visualize the datapoints
        visualize_classifier(classifier, test_datapoints,
                             [0]*len(test_datapoints),
                             'Test datapoints')
        plt.show()
```

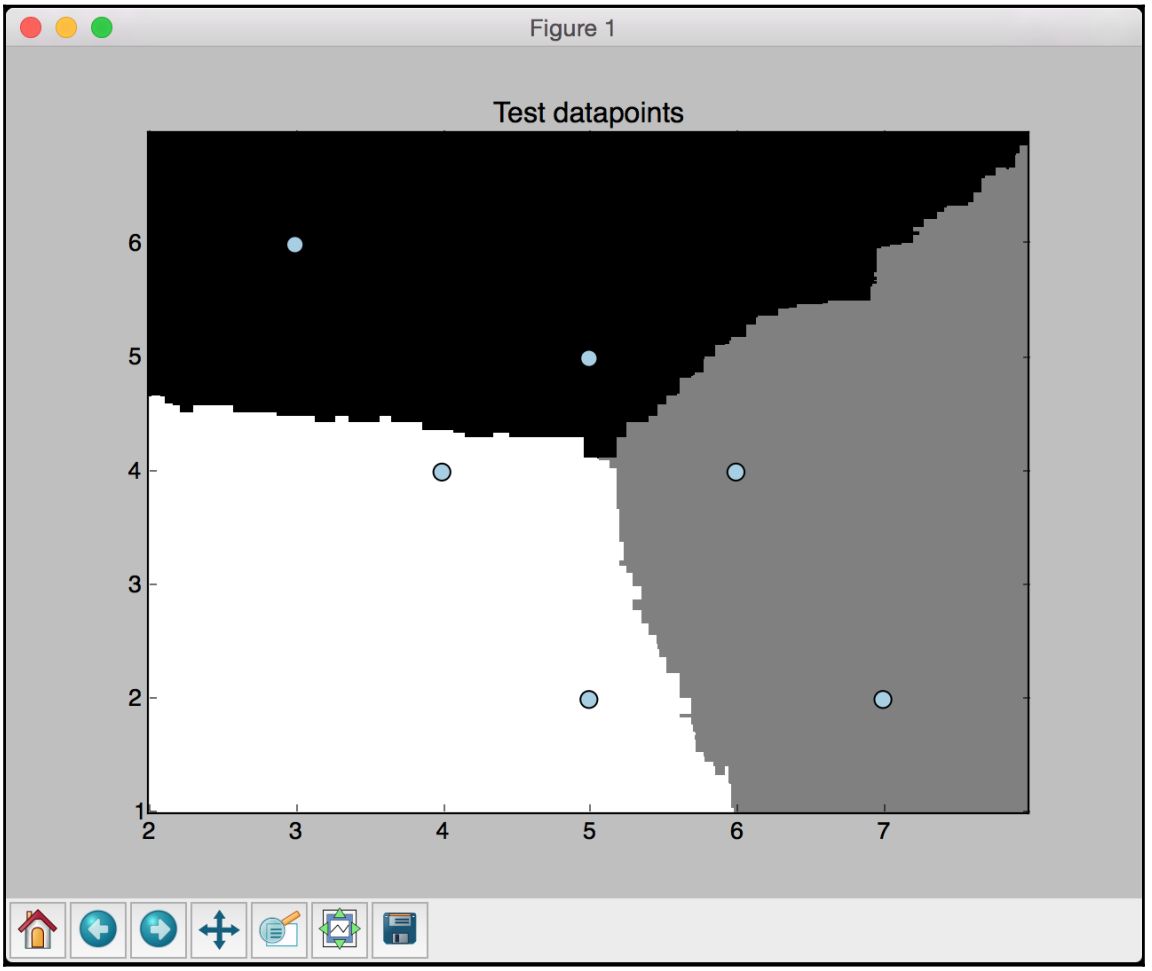
If you run the code with the `rf` flag, you will get the following output:



You will get the following output on your Terminal:

```
Datapoint: [5 5]
Probabilities: [ 0.81427532  0.08639273  0.09933195]
Predicted class: Class-0

Datapoint: [3 6]
Probabilities: [ 0.93574458  0.02465345  0.03960197]
Predicted class: Class-0

Datapoint: [6 4]
Probabilities: [ 0.12232404  0.7451078   0.13256816]
Predicted class: Class-1

Datapoint: [7 2]
Probabilities: [ 0.05415465  0.70660226  0.23924309]
Predicted class: Class-1

Datapoint: [4 4]
Probabilities: [ 0.20594744  0.15523491  0.63881765]
Predicted class: Class-2

Datapoint: [5 2]
Probabilities: [ 0.05403583  0.0931115   0.85285267]
Predicted class: Class-2
```

For each data point, it computes the probability of that point belonging to our three classes. We pick the one with the highest confidence. If you run the code with the `erf` flag, you will get the following output:



You will get the following output on your Terminal:

```
Datapoint: [5 5]
Probabilities: [ 0.48904419  0.28020114  0.23075467]
Predicted class: Class-0

Datapoint: [3 6]
Probabilities: [ 0.66707383  0.12424406  0.20868211]
Predicted class: Class-0

Datapoint: [6 4]
Probabilities: [ 0.25788769  0.49535144  0.24676087]
Predicted class: Class-1

Datapoint: [7 2]
Probabilities: [ 0.10794013  0.6246677   0.26739217]
Predicted class: Class-1

Datapoint: [4 4]
Probabilities: [ 0.33383778  0.21495182  0.45121039]
Predicted class: Class-2

Datapoint: [5 2]
Probabilities: [ 0.18671115  0.28760896  0.52567989]
Predicted class: Class-2
```

As we can see, the outputs are consistent with our observations.

Dealing with class imbalance

A classifier is only as good as the data that's used for training. One of the most common problems we face in the real world is the quality of data. For a classifier to perform well, it needs to see equal number of points for each class. But when we collect data in the real world, it's not always possible to ensure that each class has the exact same number of data points. If one class has 10 times the number of data points of the other class, then the classifier tends to get biased towards the first class. Hence we need to make sure that we account for this imbalance algorithmically. Let's see how to do that.

Create a new Python file and import the following packages:

```
import sys

import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import ExtraTreesClassifier
from sklearn import cross_validation
from sklearn.metrics import classification_report

from utilities import visualize_classifier
```

We will use the data in the file `data_imbalance.txt` for our analysis. Let's load the data. Each line in this file contains comma-separated values. The first two values correspond to the input data and the last value corresponds to the target label. We have two classes in this dataset. Let's load the data from that file:

```
# Load input data
input_file = 'data_imbalance.txt'
data = np.loadtxt(input_file, delimiter=',')
X, y = data[:, :-1], data[:, -1]
```

Separate the input data into two classes:

```
# Separate input data into two classes based on labels
class_0 = np.array(X[y==0])
class_1 = np.array(X[y==1])
```

Visualize the input data using scatter plot:

```
# Visualize input data
plt.figure()
plt.scatter(class_0[:, 0], class_0[:, 1], s=75, facecolors='black',
                        edgecolors='black', linewidth=1, marker='x')
plt.scatter(class_1[:, 0], class_1[:, 1], s=75, facecolors='white',
                        edgecolors='black', linewidth=1, marker='o')
plt.title('Input data')
```

Split the data into training and testing datasets:

```
# Split data into training and testing datasets
X_train, X_test, y_train, y_test = cross_validation.train_test_split(
            X, y, test_size=0.25, random_state=5)
```

Next, we define the parameters for the Extremely Random Forest classifier. Note that there is an input parameter called `balance` that controls whether or not we want to algorithmically account for class imbalance. If so, then we need to add another parameter called `class_weight` that tells the classifier that it should balance the weight, so that it's proportional to the number of data points in each class:

```
# Extremely Random Forests classifier
params = {'n_estimators': 100, 'max_depth': 4, 'random_state': 0}
if len(sys.argv) > 1:
      if sys.argv[1] == 'balance':
            params = {'n_estimators': 100, 'max_depth': 4, 'random_state':
0, 'class_weight': 'balanced'}
      else:
            raise TypeError("Invalid input argument; should be 'balance'")
```

Build, train, and visualize the classifier using training data:

```
classifier = ExtraTreesClassifier(**params)
classifier.fit(X_train, y_train)
visualize_classifier(classifier, X_train, y_train, 'Training dataset')
```

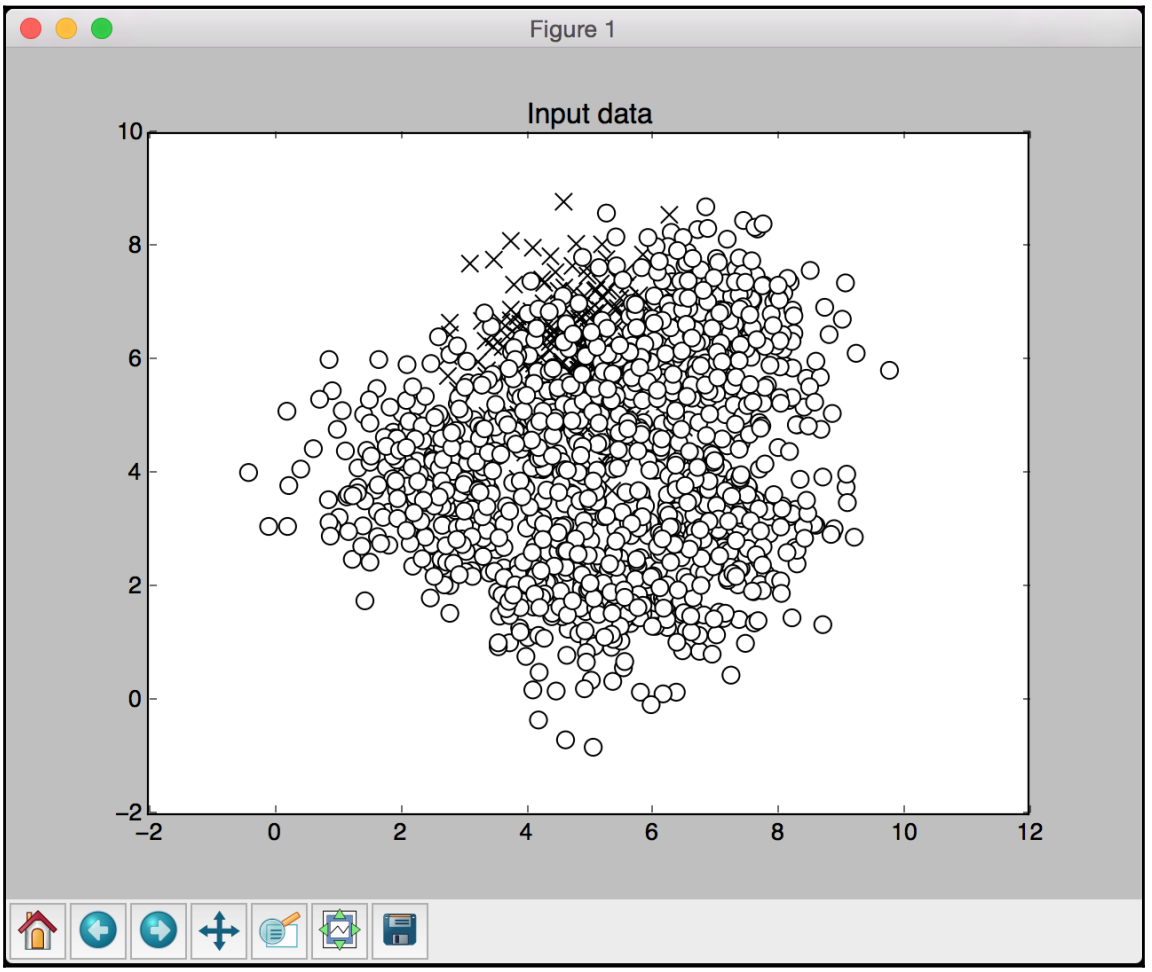
Predict the output for test dataset and visualize the output:

```
y_test_pred = classifier.predict(X_test)
visualize_classifier(classifier, X_test, y_test, 'Test dataset')
```

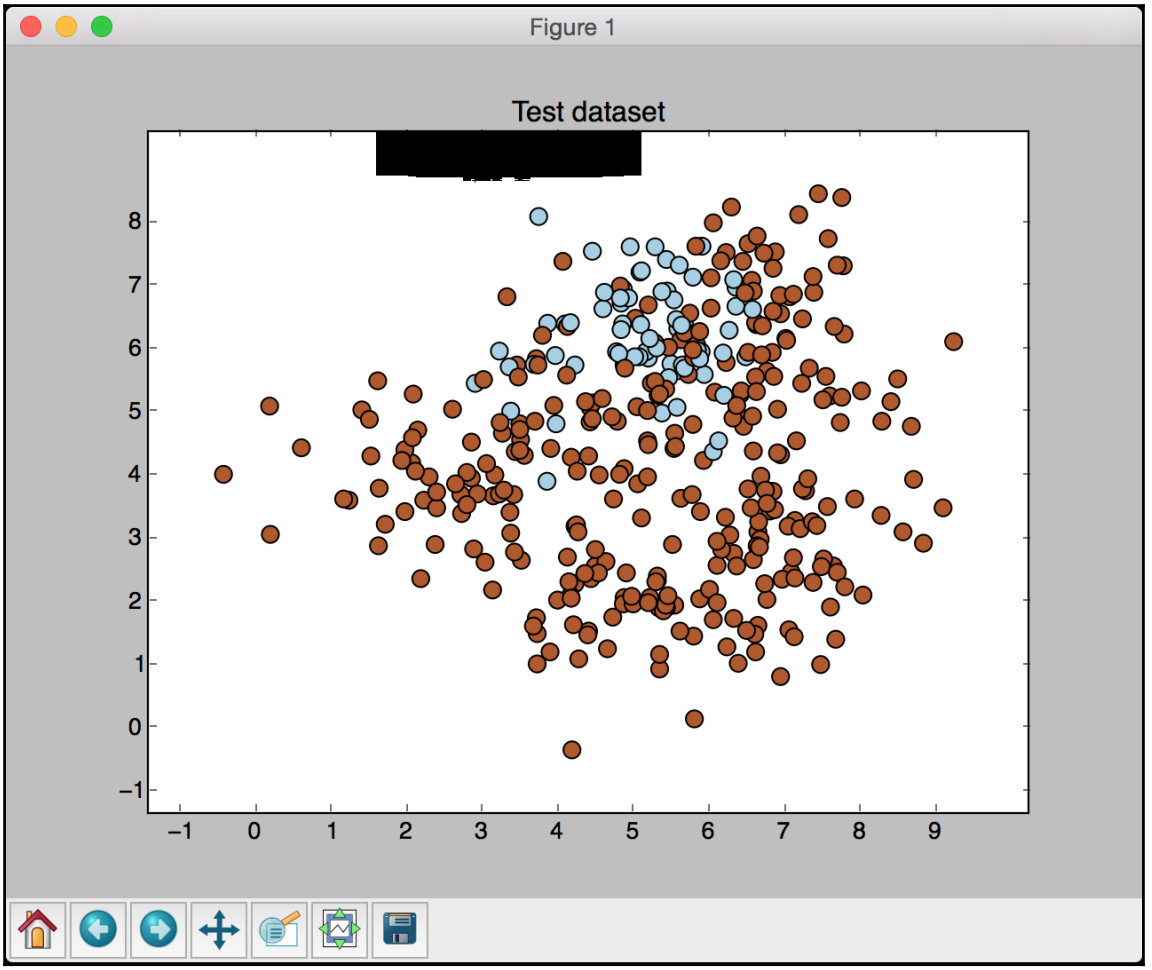
Compute the performance of the classifier and print the classification report:

```
# Evaluate classifier performance
class_names = ['Class-0', 'Class-1']
print("\n" + "#"*40)
print("\nClassifier performance on training dataset\n")
print(classification_report(y_train, classifier.predict(X_train),
target_names=class_names))
print("#"*40 + "\n")

print("#"*40)
print("\nClassifier performance on test dataset\n")
print(classification_report(y_test, y_test_pred, target_names=class_names))
print("#"*40 + "\n")

plt.show()
```

The full code is given in the file `class_imbalance.py`. If you run the code, you will see a few screenshots. The first screenshot shows the input data:



The second screenshot shows the classifier boundary for the test dataset:



The preceding screenshot indicates that the boundary was not able to capture the actual boundary between the two classes. The black patch near the top represents the boundary. You should see the following output on your Terminal:

```
########################################

Classifier performance on test dataset

             precision    recall  f1-score   support

    Class-0       0.00      0.00      0.00        69
    Class-1       0.82      1.00      0.90       306

avg / total       0.67      0.82      0.73       375

########################################
```

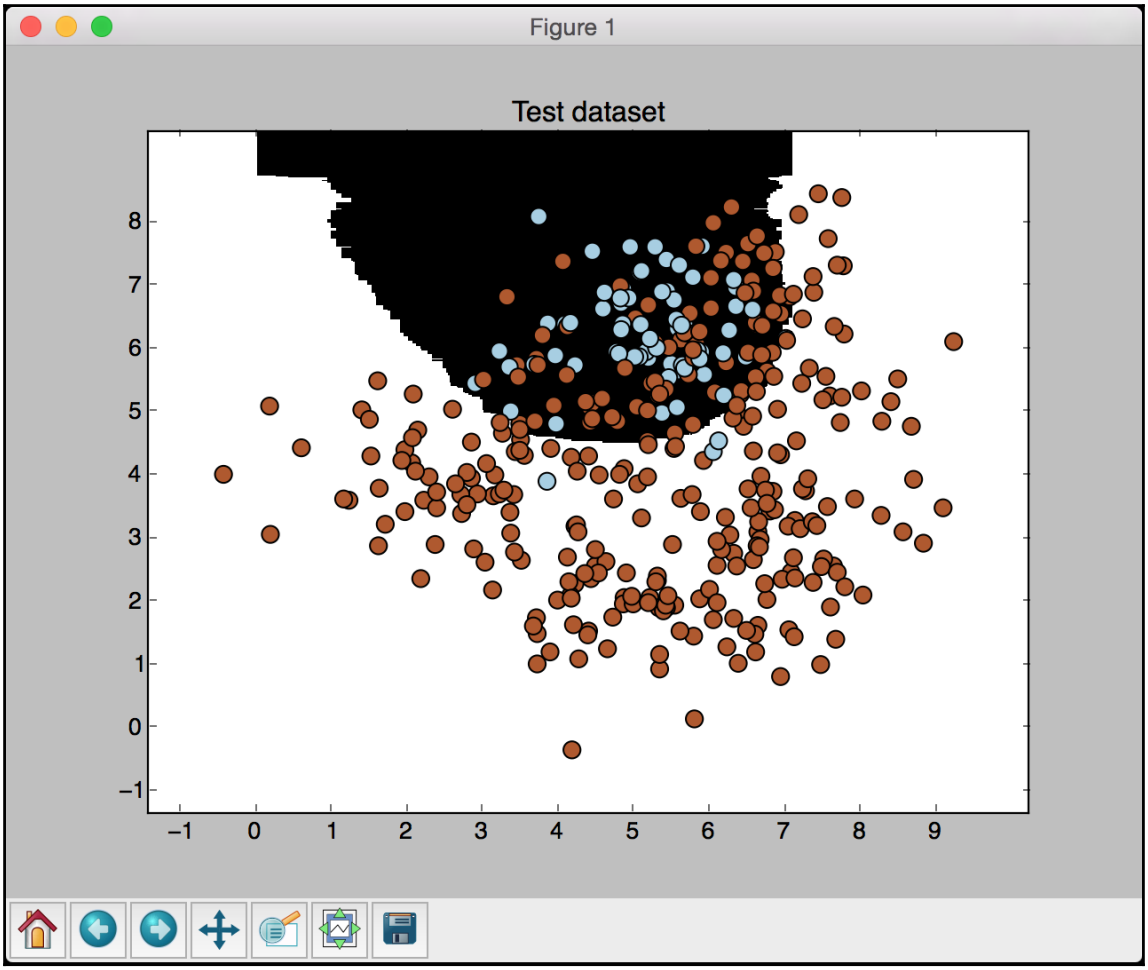
You see a warning because the values are *0* in the first row, which leads to a divide-by-zero error (`ZeroDivisionError` exception) when we compute the `f1-score`. Run the code on the terminal using the ignore flag so that you do not see the divide-by-zero warning:

```
$ python3 --W ignore class_imbalance.py
```

Now if you want to account for class imbalance, run it with the balance flag:

```
$ python3 class_imbalance.py balance
```

The classifier output looks like this:



You should see the following output on your Terminal:

```
#######################################

Classifier performance on test dataset

             precision    recall  f1-score   support

    Class-0       0.45      0.94      0.61        69
    Class-1       0.98      0.74      0.84       306

avg / total       0.88      0.78      0.80       375

#######################################
```

By accounting for the class imbalance, we were able to classify the data points in `class-0` with non-zero accuracy.

Finding optimal training parameters using grid search

When you are working with classifiers, you do not always know what the best parameters are. You cannot brute-force it by checking for all possible combinations manually. This is where grid search becomes useful. Grid search allows us to specify a range of values and the classifier will automatically run various configurations to figure out the best combination of parameters. Let's see how to do it.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report
from sklearn import cross_validation, grid_search
from sklearn.ensemble import ExtraTreesClassifier
from sklearn import cross_validation
from sklearn.metrics import classification_report

from utilities import visualize_classifier
```

We will use the data available in `data_random_forests.txt` for analysis:

```
# Load input data
input_file = 'data_random_forests.txt'
data = np.loadtxt(input_file, delimiter=',')
X, y = data[:, :-1], data[:, -1]
```

Separate the data into three classes:

```
# Separate input data into three classes based on labels
class_0 = np.array(X[y==0])
class_1 = np.array(X[y==1])
class_2 = np.array(X[y==2])
```

Split the data into training and testing datasets:

```
# Split the data into training and testing datasets
X_train, X_test, y_train, y_test = cross_validation.train_test_split(
            X, y, test_size=0.25, random_state=5)
```

Specify the grid of parameters that you want the classifier to test. Usually we keep one parameter constant and vary the other parameter. We then do it vice versa to figure out the best combination. In this case, we want to find the best values for n_estimators and `max_depth`. Let's specify the parameter grid:

```
# Define the parameter grid
parameter_grid = [ {'n_estimators': [100], 'max_depth': [2, 4, 7, 12, 16]},
                         {'max_depth': [4], 'n_estimators': [25, 50, 100,
250]}
                         ]
```

Let's define the metrics that the classifier should use to find the best combination of parameters:

```
metrics = ['precision_weighted', 'recall_weighted']
```

For each metric, we need to run the grid search, where we train the classifier for a particular combination of parameters:

```
for metric in metrics:
      print("\n##### Searching optimal parameters for", metric)

      classifier = grid_search.GridSearchCV(
                  ExtraTreesClassifier(random_state=0),
                  parameter_grid, cv=5, scoring=metric)
      classifier.fit(X_train, y_train)
```

Print the score for each parameter combination:

```
print("\nGrid scores for the parameter grid:")
for params, avg_score, _ in classifier.grid_scores_:
      print(params, '-->', round(avg_score, 3))

print("\nBest parameters:", classifier.best_params_)
```

Print the performance report:

```
y_pred = classifier.predict(X_test)
print("\nPerformance report:\n")
print(classification_report(y_test, y_pred))
```

The full code is given in the file `run_grid_search.py`. If you run the code, you will get this output on the Terminal for the precision metric:

```
##### Searching optimal parameters for precision_weighted

Grid scores for the parameter grid:
{'n_estimators': 100, 'max_depth': 2} --> 0.847
{'n_estimators': 100, 'max_depth': 4} --> 0.841
{'n_estimators': 100, 'max_depth': 7} --> 0.844
{'n_estimators': 100, 'max_depth': 12} --> 0.836
{'n_estimators': 100, 'max_depth': 16} --> 0.818
{'n_estimators': 25, 'max_depth': 4} --> 0.846
{'n_estimators': 50, 'max_depth': 4} --> 0.84
{'n_estimators': 100, 'max_depth': 4} --> 0.841
{'n_estimators': 250, 'max_depth': 4} --> 0.845

Best parameters: {'n_estimators': 100, 'max_depth': 2}

Performance report:

             precision    recall  f1-score   support

        0.0       0.94      0.81      0.87        79
        1.0       0.81      0.86      0.83        70
        2.0       0.83      0.91      0.87        76

avg / total       0.86      0.86      0.86       225
```

Based on the combinations in the grid search, it will print out the best combination for the precision metric. If we want to know the best combination for recall, we need to check the following output on the Terminal:

```
##### Searching optimal parameters for recall_weighted

Grid scores for the parameter grid:
{'n_estimators': 100, 'max_depth': 2} --> 0.84
{'n_estimators': 100, 'max_depth': 4} --> 0.837
{'n_estimators': 100, 'max_depth': 7} --> 0.841
{'n_estimators': 100, 'max_depth': 12} --> 0.834
{'n_estimators': 100, 'max_depth': 16} --> 0.816
{'n_estimators': 25, 'max_depth': 4} --> 0.843
{'n_estimators': 50, 'max_depth': 4} --> 0.836
{'n_estimators': 100, 'max_depth': 4} --> 0.837
{'n_estimators': 250, 'max_depth': 4} --> 0.841

Best parameters: {'n_estimators': 25, 'max_depth': 4}

Performance report:

             precision    recall  f1-score   support

        0.0       0.93      0.84      0.88        79
        1.0       0.85      0.86      0.85        70
        2.0       0.84      0.92      0.88        76

avg / total       0.87      0.87      0.87       225
```

It is a different combination for recall, which makes sense because precision and recall are different metrics that demand different parameter combinations.

Computing relative feature importance

When we are working with a dataset that contains N-dimensional data points, we have to understand that not all features are equally important. Some are more discriminative than others. If we have this information, we can use it to reduce the dimensional. This is very useful in reducing the complexity and increasing the speed of the algorithm. Sometimes, a few features are completely redundant. Hence they can be easily removed from the dataset.

We will be using the `AdaBoost` regressor to compute feature importance. `AdaBoost`, short for Adaptive Boosting, is an algorithm that's frequently used in conjunction with other machine learning algorithms to improve their performance. In `AdaBoost`, the training data points are drawn from a distribution to train the current classifier. This distribution is updated iteratively so that the subsequent classifiers get to focus on the more difficult data points. The difficult data points are the ones that are misclassified. This is done by updating the distribution at each step. This will make the data points that were previously misclassified more likely to come up in the next sample dataset that's used for training. These classifiers are then cascaded and the decision is taken through weighted majority voting.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import AdaBoostRegressor
from sklearn import datasets
from sklearn.metrics import mean_squared_error, explained_variance_score
from sklearn import cross_validation
from sklearn.utils import shuffle
```

We will use the inbuilt housing dataset available in scikit-learn:

```
# Load housing data
housing_data = datasets.load_boston()
```

Shuffle the data so that we don't bias our analysis:

```
# Shuffle the data
X, y = shuffle(housing_data.data, housing_data.target, random_state=7)
```

Split the dataset into training and testing:

```
# Split data into training and testing datasets
X_train, X_test, y_train, y_test = cross_validation.train_test_split(
            X, y, test_size=0.2, random_state=7)
```

Define and train an `AdaBoostregressor` using the Decision Tree regressor as the individual model:

```
# AdaBoost Regressor model
regressor = AdaBoostRegressor(DecisionTreeRegressor(max_depth=4),
            n_estimators=400, random_state=7)
regressor.fit(X_train, y_train)
```

Estimate the performance of the regressor:

```
# Evaluate performance of AdaBoost regressor
y_pred = regressor.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
evs = explained_variance_score(y_test, y_pred )
print("\nADABOOST REGRESSOR")
print("Mean squared error =", round(mse, 2))
print("Explained variance score =", round(evs, 2))
```

This regressor has an inbuilt method that can be called to compute the relative feature importance:

```
# Extract feature importances
feature_importances = regressor.feature_importances_
feature_names = housing_data.feature_names
```

Normalize the values of the relative feature importance:

```
# Normalize the importance values
feature_importances = 100.0 * (feature_importances /
max(feature_importances))
```

Sort them so that they can be plotted:

```
# Sort the values and flip them
index_sorted = np.flipud(np.argsort(feature_importances))
```

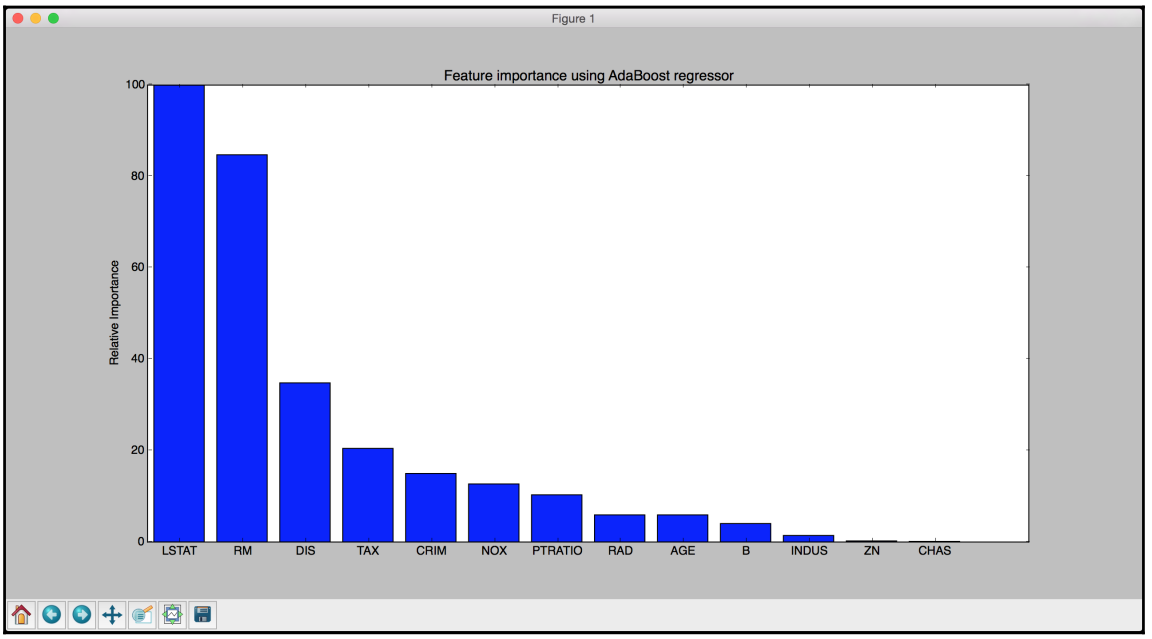
Arrange the ticks on the X axis for the bar graph:

```
# Arrange the X ticks
pos = np.arange(index_sorted.shape[0]) + 0.5
```

Plot the bar graph:

```
# Plot the bar graph
plt.figure()
plt.bar(pos, feature_importances[index_sorted], align='center')
plt.xticks(pos, feature_names[index_sorted])
plt.ylabel('Relative Importance')
plt.title('Feature importance using AdaBoost regressor')
plt.show()
```

The full code is given in the file `feature_importance.py`. If you run the code, you should see the following output:



According to this analysis, the feature LSTAT is the most important feature in that dataset.

Predicting traffic using Extremely Random Forest regressor

Let's apply the concepts we learned in the previous sections to a real world problem. We will be using the dataset available at: `https://archive.ics.uci.edu/ml/datasets/Dodgers+Loop+Sensor`. This dataset consists of data that counts the number of vehicles passing by on the road during baseball games played at Los Angeles Dodgers stadium. In order to make the data readily available for analysis, we need to pre-process it. The pre-processed data is in the file `traffic_data.txt`. In this file, each line contains comma-separated strings. Let's take the first line as an example:

Tuesday,00:00,San Francisco,no,3

With reference to the preceding line, it is formatted as follows:

Day of the week, time of the day, opponent team, binary value indicating whether or not a baseball game is currently going on (yes/no), number of vehicles passing by.

Our goal is to predict the number of vehicles going by using the given information. Since the output variable is continuous valued, we need to build a regressor that can predict the output. We will be using Extremely Random Forests to build this regressor. Let's go ahead and see how to do that.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, mean_absolute_error
from sklearn import cross_validation, preprocessing
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.metrics import classification_report
```

Load the data in the file `traffic_data.txt`:

```
# Load input data
input_file = 'traffic_data.txt'
data = []
with open(input_file, 'r') as f:
      for line in f.readlines():
            items = line[:-1].split(',')
            data.append(items)

data = np.array(data)
```

We need to encode the non-numerical features in the data. We also need to ensure that we don't encode numerical features. Each feature that needs to be encoded needs to have a separate label encoder. We need to keep track of these encoders because we will need them when we want to compute the output for an unknown data point. Let's create those label encoders:

```
# Convert string data to numerical data
label_encoder = []
X_encoded = np.empty(data.shape)
for i, item in enumerate(data[0]):
      if item.isdigit():
            X_encoded[:, i] = data[:, i]
      else:
            label_encoder.append(preprocessing.LabelEncoder())
            X_encoded[:, i] = label_encoder[-1].fit_transform(data[:, i])
```

```
X = X_encoded[:, :-1].astype(int)
y = X_encoded[:, -1].astype(int)
```

Split the data into training and testing datasets:

```
# Split data into training and testing datasets
X_train, X_test, y_train, y_test = cross_validation.train_test_split(
            X, y, test_size=0.25, random_state=5)
```

Train an extremely Random Forests regressor:

```
# Extremely Random Forests regressor
params = {'n_estimators': 100, 'max_depth': 4, 'random_state': 0}
regressor = ExtraTreesRegressor(**params)
regressor.fit(X_train, y_train)
```

Compute the performance of the regressor on testing data:

```
# Compute the regressor performance on test data
y_pred = regressor.predict(X_test)
print("Mean absolute error:", round(mean_absolute_error(y_test, y_pred),
2))
```

Let's see how to compute the output for an unknown data point. We will be using those label encoders to convert non-numerical features into numerical values:

```
# Testing encoding on single data instance
test_datapoint = ['Saturday', '10:20', 'Atlanta', 'no']
test_datapoint_encoded = [-1] * len(test_datapoint)
count = 0
for i, item in enumerate(test_datapoint):
      if item.isdigit():
            test_datapoint_encoded[i] = int(test_datapoint[i])
      else:
            test_datapoint_encoded[i] =
int(label_encoder[count].transform(test_datapoint[i]))
            count = count + 1

test_datapoint_encoded = np.array(test_datapoint_encoded)
```

Predict the output:

```
# Predict the output for the test datapoint
print("Predicted traffic:",
int(regressor.predict([test_datapoint_encoded])[0]))
```

The full code is given in the file `traffic_prediction.py`. If you run the code, you will get *26* as the output, which is pretty close to the actual value. You can confirm this from the data file.

Summary

In this chapter, we learned about Ensemble Learning and how it can be used in the real world. We discussed Decision Trees and how to build a classifier based on it.

We learned about Random Forests and Extremely Random Forests. We discussed how to build classifiers based on them. We understood how to estimate the confidence measure of the predictions. We also learned how to deal with the class imbalance problem.

We discussed how to find the most optimal training parameters to build the models using grid search. We learned how to compute relative feature importance. We then applied ensemble learning techniques to a real-world problem, where we predicted traffic using Extremely Random Forest regressor.

In the next chapter, we will discuss unsupervised learning and how to detect patterns in stock market data.

4

Detecting Patterns with Unsupervised Learning

In this chapter, we are going to learn about unsupervised learning and how to use it in the real world. By the end of this chapter, you will know these things:

- What is unsupervised learning?
- Clustering data with K-Means algorithm
- Estimating the number of clusters with Mean Shift algorithm
- Estimating the quality of clustering with silhouette scores
- What are Gaussian Mixture Models?
- Building a classifier based on Gaussian Mixture Models
- Finding subgroups in stock market using Affinity Propagation model
- Segmenting the market based on shopping patterns

What is unsupervised learning?

Unsupervised learning refers to the process of building machine learning models without using labeled training data. Unsupervised learning finds applications in diverse fields of study, including market segmentation, stock markets, natural language processing, computer vision, and so on.

In the previous chapters, we were dealing with data that had labels associated with it. When we have labeled training data, the algorithms learn to classify data based on those labels. In the real world, we might not always have access to labeled data. Sometimes, we just have a lot of data and we need to categorize it in some way. This is where unsupervised learning comes into picture. Unsupervised learning algorithms attempt to build learning models that can find subgroups within the given dataset using some similarity metric.

Let's see how we formulate the learning problem in unsupervised learning. When we have a dataset without any labels, we assume that the data is generated because of latent variables that govern the distribution in some way. The process of learning can then proceed in a hierarchical manner, starting from the individual data points. We can build deeper levels of representation for the data.

Clustering data with K-Means algorithm

Clustering is one of the most popular unsupervised learning techniques. This technique is used to analyze data and find clusters within that data. In order to find these clusters, we use some kind of similarity measure such as Euclidean distance, to find the subgroups. This similarity measure can estimate the tightness of a cluster. We can say that clustering is the process of organizing our data into subgroups whose elements are similar to each other.

Our goal is to identify the intrinsic properties of data points that make them belong to the same subgroup. There is no universal similarity metric that works for all the cases. It depends on the problem at hand. For example, we might be interested in finding the representative data point for each subgroup or we might be interested in finding the outliers in our data. Depending on the situation, we will end up choosing the appropriate metric.

K-Means algorithm is a well-known algorithm for clustering data. In order to use this algorithm, we need to assume that the number of clusters is known beforehand. We then segment data into K subgroups using various data attributes. We start by fixing the number of clusters and classify our data based on that. The central idea here is that we need to update the locations of these K centroids with each iteration. We continue iterating until we have placed the centroids at their optimal locations.

We can see that the initial placement of centroids plays an important role in the algorithm. These centroids should be placed in a clever manner, because this directly impacts the results. A good strategy is to place them as far away from each other as possible. The basic K-Means algorithm places these centroids randomly where `K-Means++` chooses these points algorithmically from the input list of data points. It tries to place the initial centroids far from each other so that it converges quickly. We then go through our training dataset and assign each data point to the closest centroid.

Once we go through the entire dataset, we say that the first iteration is over. We have grouped the points based on the initialized centroids. We now need to recalculate the location of the centroids based on the new clusters that we obtain at the end of the first iteration. Once we obtain the new set of K centroids, we repeat the process again, where we iterate through the dataset and assign each point to the closest centroid.

As we keep repeating these steps, the centroids keep moving to their equilibrium position. After a certain number of iterations, the centroids do not change their locations anymore. This means that we have arrived at the final locations of the centroids. These K centroids are the final K Means that will be used for inference.

Let's apply K-Means clustering on two-dimensional data to see how it works. We will be using the data in the `data_clustering.txt` file provided to you. Each line contains two comma-separated numbers.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn import metrics
```

Load the input data from the file:

```
# Load input data
X = np.loadtxt('data_clustering.txt', delimiter=',')
```

We need to define the number of clusters before we can apply `K-Means` algorithm:

```
num_clusters = 5
```

Visualize the input data to see what the spread looks like:

```
# Plot input data
plt.figure()
plt.scatter(X[:,0], X[:,1], marker='o', facecolors='none',
        edgecolors='black', s=80)
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
```

```
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
plt.title('Input data')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())
```

We can visually see that there are five groups within this data. Create the `KMeans` object using the initialization parameters. The `init` parameter represents the method of initialization to select the initial centers of clusters. Instead of selecting them randomly, we use `k-means++` to select these centers in a smarter way. This ensures that the algorithm converges quickly. The `n_clusters` parameter refers to the number of clusters. The `n_init` parameter refers to the number of times the algorithm should run before deciding upon the best outcome:

```
# Create KMeans object
kmeans = KMeans(init='k-means++', n_clusters=num_clusters, n_init=10)
```

Train the K-Means model with the input data:

```
# Train the KMeans clustering model
kmeans.fit(X)
```

To visualize the boundaries, we need to create a grid of points and evaluate the model on all those points. Let's define the step size of this grid:

```
# Step size of the mesh
step_size = 0.01
```

We define the grid of points and ensure that we are covering all the values in our input data:

```
# Define the grid of points to plot the boundaries
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
x_vals, y_vals = np.meshgrid(np.arange(x_min, x_max, step_size),
        np.arange(y_min, y_max, step_size))
```

Predict the outputs for all the points on the grid using the trained K-Means model:

```
# Predict output labels for all the points on the grid
output = kmeans.predict(np.c_[x_vals.ravel(), y_vals.ravel()])
```

Plot all output values and color each region:

```
# Plot different regions and color them
output = output.reshape(x_vals.shape)
plt.figure()
plt.clf()
plt.imshow(output, interpolation='nearest',
           extent=(x_vals.min(), x_vals.max(),
               y_vals.min(), y_vals.max()),
           cmap=plt.cm.Paired,
           aspect='auto',
           origin='lower')
```

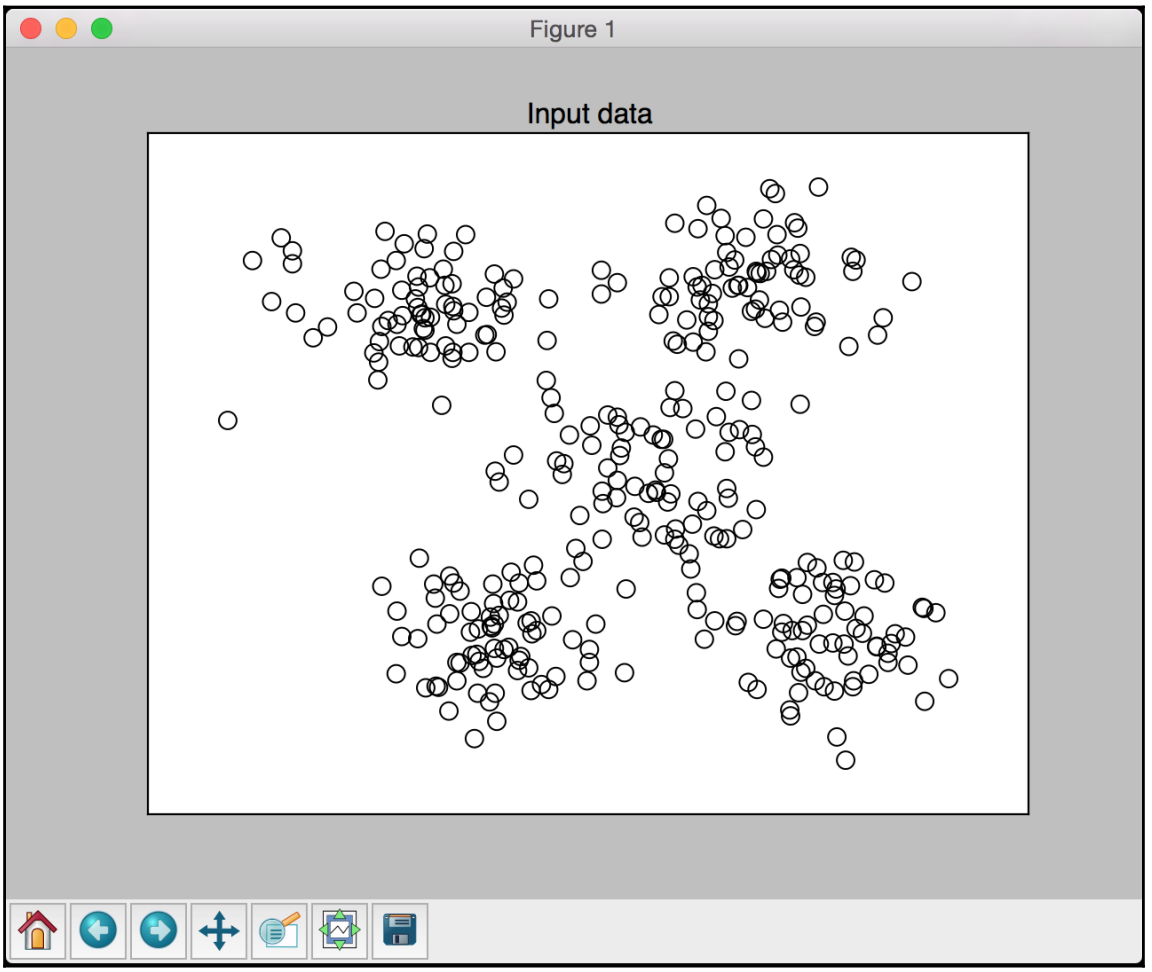
Overlay input data points on top of these colored regions:

```
# Overlay input points
plt.scatter(X[:,0], X[:,1], marker='o', facecolors='none',
        edgecolors='black', s=80)
```

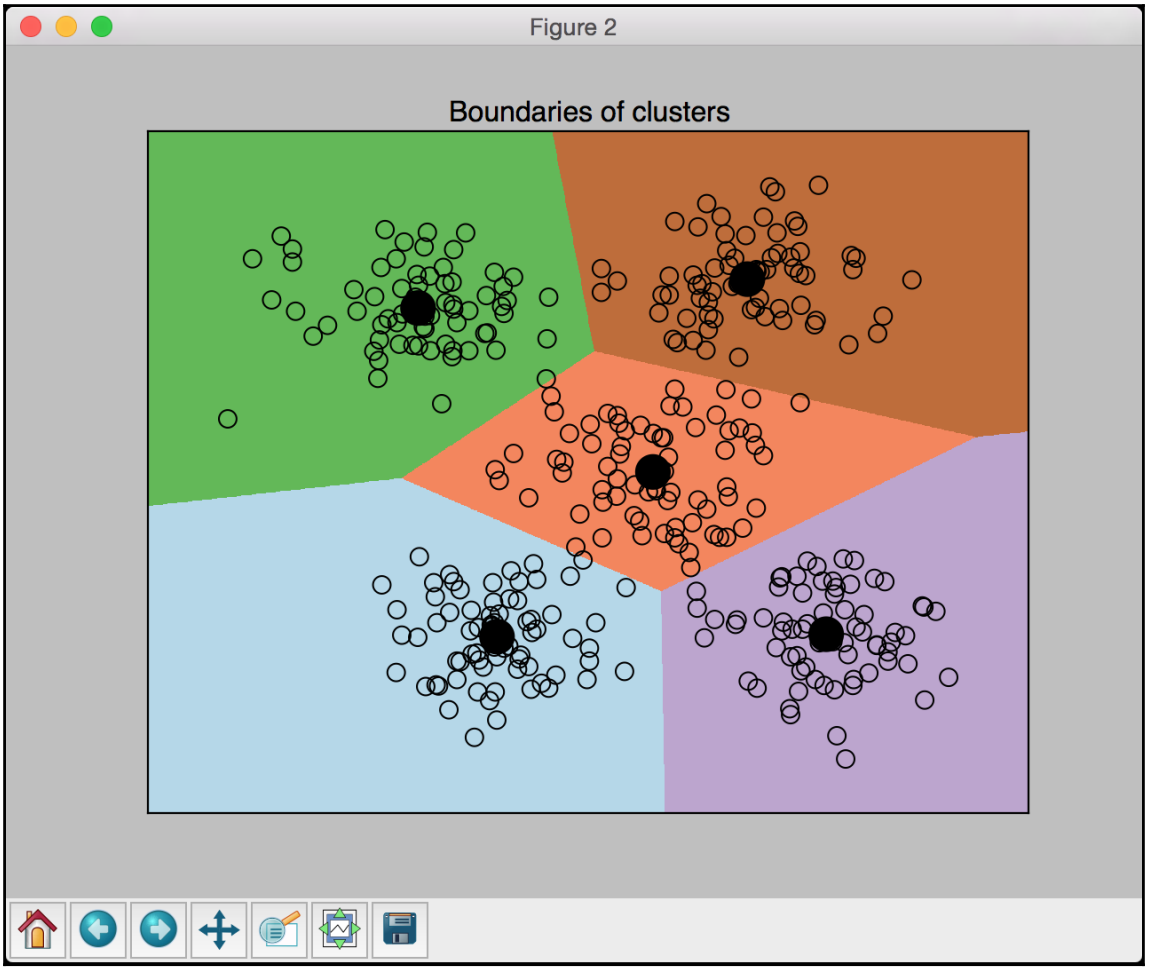
Plot the centers of the clusters obtained using the K-Means algorithm:

```
# Plot the centers of clusters
cluster_centers = kmeans.cluster_centers_
plt.scatter(cluster_centers[:,0], cluster_centers[:,1],
        marker='o', s=210, linewidths=4, color='black',
        zorder=12, facecolors='black')

x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
plt.title('Boundaries of clusters')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())
plt.show()
```

The full code is given in the `kmeans.py` file. If you run the code, you will see two screenshot. The first screenshot is the input data:



The second screenshot represents the boundaries obtained using K-Means:



The black filled circle at the center of each cluster represents the centroid of that cluster.

Estimating the number of clusters with Mean Shift algorithm

Mean Shift is a powerful algorithm used in unsupervised learning. It is a non-parametric algorithm used frequently for clustering. It is non-parametric because it does not make any assumptions about the underlying distributions. This is in contrast to parametric techniques, where we assume that the underlying data follows a standard probability distribution. Mean Shift finds a lot of applications in fields like object tracking and real-time data analysis.

In the Mean Shift algorithm, we consider the whole feature space as a probability density function. We start with the training dataset and assume that they have been sampled from a probability density function. In this framework, the clusters correspond to the local maxima of the underlying distribution. If there are K clusters, then there are K peaks in the underlying data distribution and Mean Shift will identify those peaks.

The goal of Mean Shift is to identify the location of centroids. For each data point in the training dataset, it defines a window around it. It then computes the centroid for this window and updates the location to this new centroid. It then repeats the process for this new location by defining a window around it. As we keep doing this, we move closer to the peak of the cluster. Each data point will move towards the cluster it belongs to. The movement is towards a region of higher density.

We keep shifting the centroids, also called means, towards the peaks of each cluster. Since we keep shifting the means, it is called Mean Shift! We keep doing this until the algorithm converges, at which stage the centroids don't move anymore.

Let's see how to use `MeanShift` to estimate the optimal number of clusters in the given dataset. We will be using data in the `data_clustering.txt` file for analysis. It is the same file we used in the *KMeans* section.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import MeanShift, estimate_bandwidth
from itertools import cycle
```

Load input data:

```
# Load data from input file
X = np.loadtxt('data_clustering.txt', delimiter=',')
```

Estimate the bandwidth of the input data. Bandwidth is a parameter of the underlying kernel density estimation process used in Mean Shift algorithm. The bandwidth affects the overall convergence rate of the algorithm and the number of clusters that we will end up with in the end. Hence this is a crucial parameter. If the bandwidth is small, it might results in too many clusters, where as if the value is large, then it will merge distinct clusters.

The `quantile` parameter impacts how the bandwidth is estimated. A higher value for quantile will increase the estimated bandwidth, resulting in a lesser number of clusters:

```
# Estimate the bandwidth of X
bandwidth_X = estimate_bandwidth(X, quantile=0.1, n_samples=len(X))
```

Let's train the Mean Shift clustering model using the estimated bandwidth:

```
# Cluster data with MeanShift
meanshift_model = MeanShift(bandwidth=bandwidth_X, bin_seeding=True)
meanshift_model.fit(X)
```

Extract the centers of all the clusters:

```
# Extract the centers of clusters
cluster_centers = meanshift_model.cluster_centers_
print('\nCenters of clusters:\n', cluster_centers)
```

Extract the number of clusters:

```
# Estimate the number of clusters
labels = meanshift_model.labels_
num_clusters = len(np.unique(labels))
print("\nNumber of clusters in input data =", num_clusters)
```

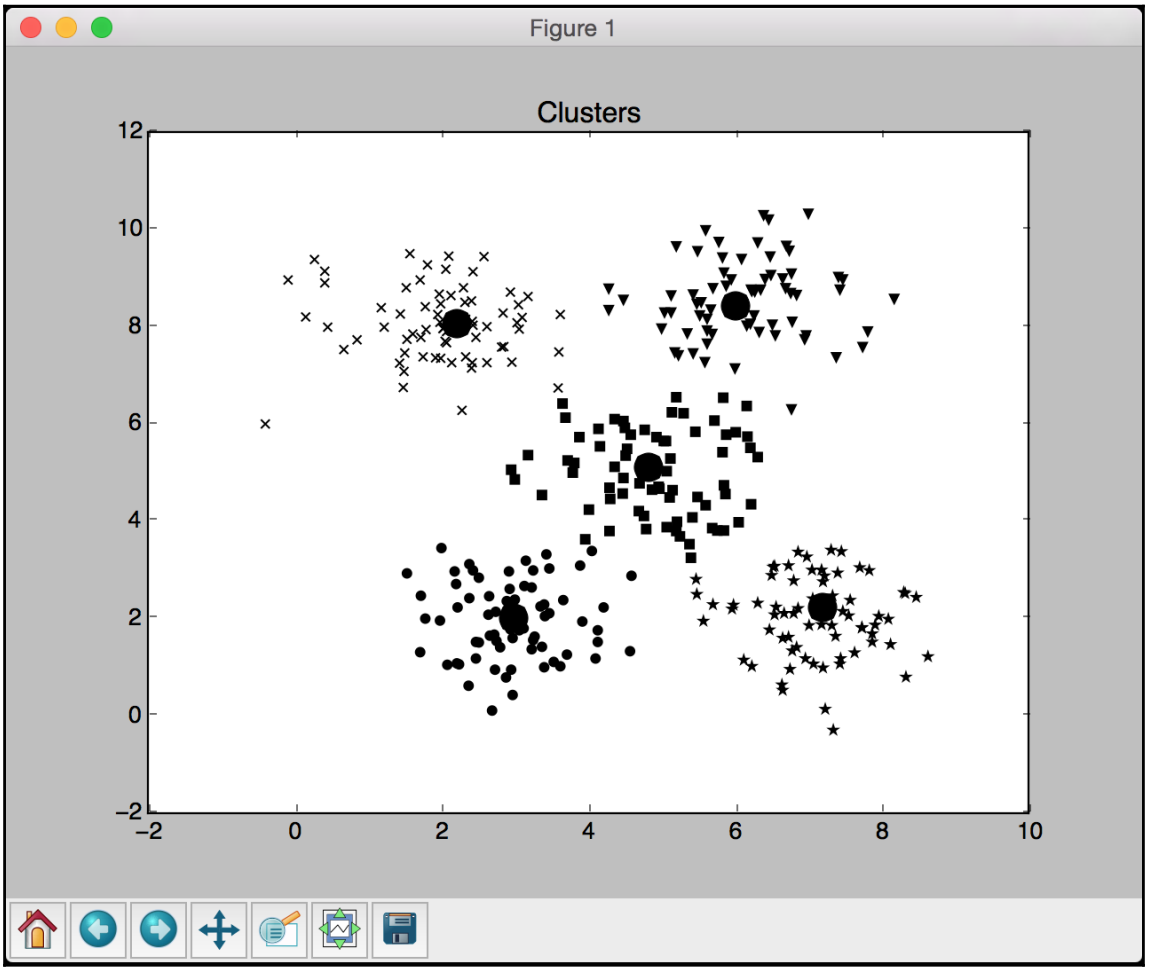
Visualize the data points:

```
# Plot the points and cluster centers
plt.figure()
markers = 'o*xvs'
for i, marker in zip(range(num_clusters), markers):
    # Plot points that belong to the current cluster
    plt.scatter(X[labels==i, 0], X[labels==i, 1], marker=marker,
color='black')
```

Plot the center of the current cluster:

```
        # Plot the cluster center
        cluster_center = cluster_centers[i]
        plt.plot(cluster_center[0], cluster_center[1], marker='o',
                markerfacecolor='black', markeredgecolor='black',
                markersize=15)

    plt.title('Clusters')
    plt.show()
```

The full code is given in the `mean_shift.py` file. If you run the code, you will see the following screenshot representing the clusters and their centers:



You will see the following on your Terminal:

```
Centers of clusters:
 [[ 2.95568966  1.95775862]
 [ 7.17563636  2.18145455]
 [ 2.17603774  8.03283019]
 [ 5.97960784  8.39078431]
 [ 4.81044444  5.07111111]]

Number of clusters in input data = 5
```

Estimating the quality of clustering with silhouette scores

If the data is naturally organized into a number of distinct clusters, then it is easy to visually examine it and draw some inferences. But this is rarely the case in the real world. The data in the real world is huge and messy. So we need a way to quantify the quality of the clustering.

Silhouette refers to a method used to check the consistency of clusters in our data. It gives an estimate of how well each data point fits with its cluster. The silhouette score is a metric that measures how similar a data point is to its own cluster, as compared to other clusters. The silhouette score works with any similarity metric.

For each data point, the silhouette score is computed using the following formula:

silhouette score = (p – q) / max(p, q)

Here, *p* is the mean distance to the points in the nearest cluster that the data point is not a part of, and *q* is the mean intra-cluster distance to all the points in its own cluster.

The value of the silhouette score range lies between *-1* to *1*. A score closer to *1* indicates that the data point is very similar to other data points in the cluster, whereas a score closer to *-1* indicates that the data point is not similar to the data points in its cluster. One way to think about it is if you get too many points with negative silhouette scores, then we may have too few or too many clusters in our data. We need to run the clustering algorithm again to find the optimal number of clusters.

Let's see how to estimate the clustering performance using silhouette scores. Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import metrics
from sklearn.cluster import KMeans
```

We will be using the data in the `data_quality.txt` file provided to you. Each line contains two comma-separated numbers:

```
# Load data from input file
X = np.loadtxt('data_quality.txt', delimiter=',')
```

Initialize the variables. The `values` array will contain a list of values we want to iterate on and find the optimal number of clusters:

```
# Initialize variables
scores = []
values = np.arange(2, 10)
```

Iterate through all the values and build a K-Means model during each iteration:

```
# Iterate through the defined range
for num_clusters in values:
    # Train the KMeans clustering model
    kmeans = KMeans(init='k-means++', n_clusters=num_clusters, n_init=10)
    kmeans.fit(X)
```

Estimate the silhouette score for the current clustering model using Euclidean distance metric:

```
score = metrics.silhouette_score(X, kmeans.labels_,
            metric='euclidean', sample_size=len(X))
```

Print the silhouette score for the current value:

```
print("\nNumber of clusters =", num_clusters)
print("Silhouette score =", score)
scores.append(score)
```

Visualize the silhouette scores for various values:

```
# Plot silhouette scores
plt.figure()
plt.bar(values, scores, width=0.7, color='black', align='center')
plt.title('Silhouette score vs number of clusters')
```

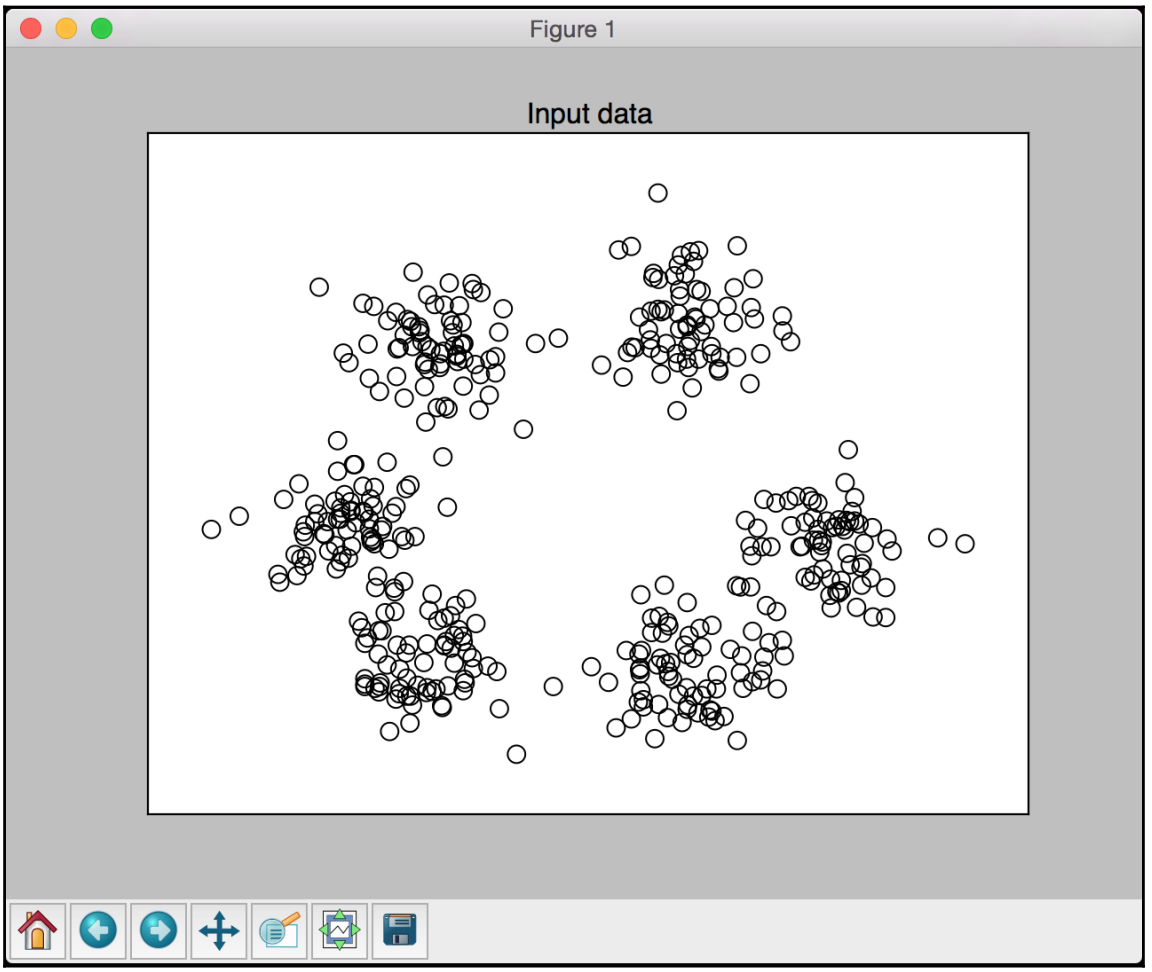
Extract the best score and the corresponding value for the number of clusters:

```
# Extract best score and optimal number of clusters
num_clusters = np.argmax(scores) + values[0]
print('\nOptimal number of clusters =', num_clusters)
```

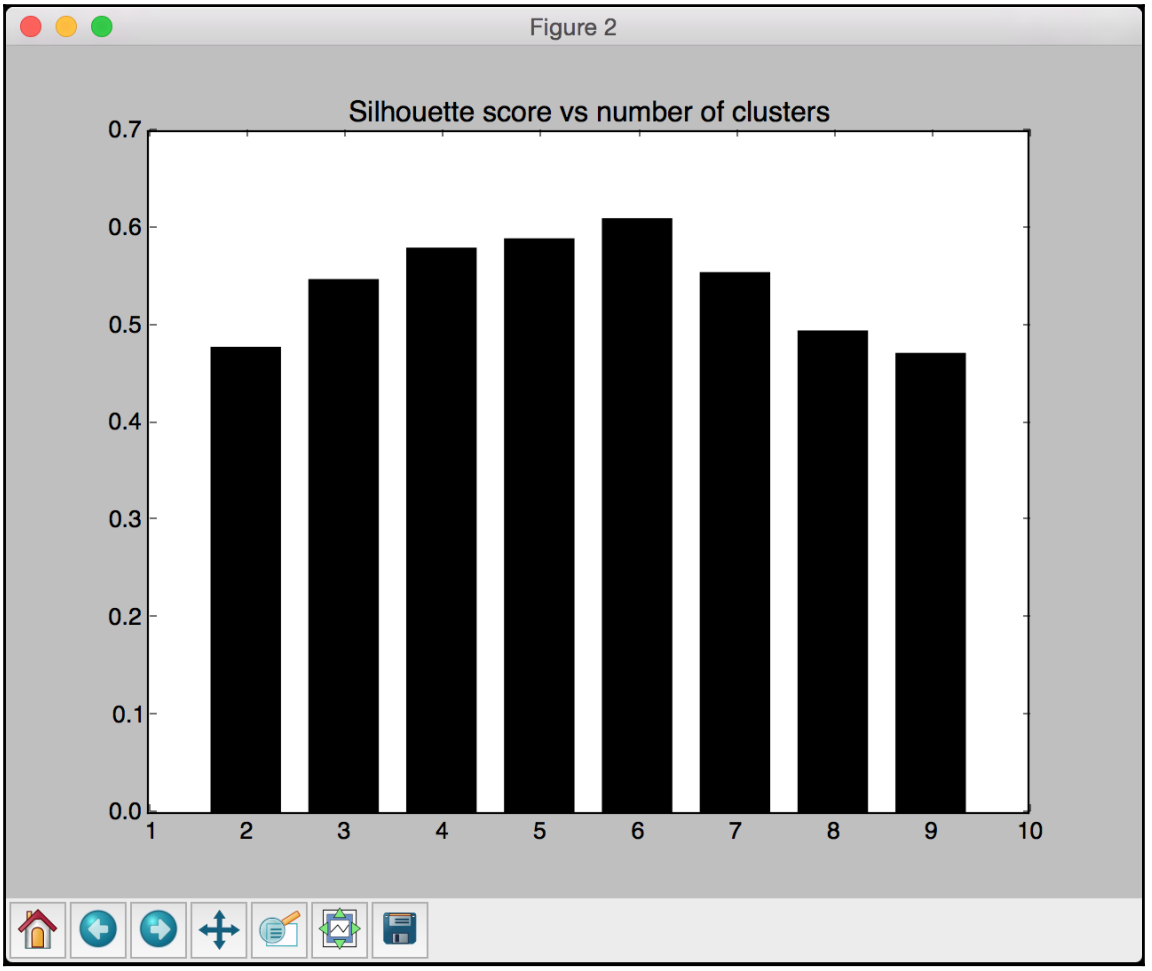
Visualize input data:

```
# Plot data
plt.figure()
plt.scatter(X[:,0], X[:,1], color='black', s=80, marker='o',
facecolors='none')
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
plt.title('Input data')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.xticks(())
plt.yticks(())

plt.show()
```

The full code is given in the file `clustering_quality.py`. If you run the code, you will see two screenshot. The first screenshot is the input data:



We can see that there are six clusters in our data. The second screenshot represents the scores for various values of number of clusters:



We can verify that the silhouette score is peaking at the value of *6*, which is consistent with our data. You will see the following on your Terminal:

```
Number of clusters = 2
Silhouette score = 0.477626248705

Number of clusters = 3
Silhouette score = 0.547174241173

Number of clusters = 4
Silhouette score = 0.579480188969

Number of clusters = 5
Silhouette score = 0.589003263565

Number of clusters = 6
Silhouette score = 0.609690411895

Number of clusters = 7
Silhouette score = 0.554310234032

Number of clusters = 8
Silhouette score = 0.494433661954

Number of clusters = 9
Silhouette score = 0.471414689437

Optimal number of clusters = 6
```

What are Gaussian Mixture Models?

Before we discuss **Gaussian Mixture Models** (**GMM**s), let's understand what Mixture Models are. A Mixture Model is a type of probability density model where we assume that the data is governed by a number of component distributions. If these distributions are Gaussian, then the model becomes a Gaussian Mixture Model. These component distributions are combined in order to provide a multi-modal density function, which becomes a mixture model.

Let's look at an example to understand how Mixture Models work. We want to model the shopping habits of all the people in South America. One way to do it would be model the whole continent and fit everything into a single model. But we know that people in different countries shop differently. We need to understand how people in individual countries shop and how they behave.

If we want to get a good representative model, we need to account for all the variations within the continent. In this case, we can use mixture models to model the shopping habits of individual countries and then combine all of them into a Mixture Model. This way, we are not missing the nuances of the underlying behavior of individual countries. By not enforcing a single model on all the countries, we are able to extract a more accurate model.

An interesting thing to note is that mixture models are semi-parametric, which means that they are partially dependent on a set of predefined functions. They are able to provide greater precision and flexibility in modeling the underlying distributions of our data. They can smooth the gaps that result from having sparse data.

If we define the function, then the mixture model goes from being semi-parametric to parametric. Hence a GMM is a parametric model represented as a weighted summation of component Gaussian functions. We assume that the data is being generated by a set of Gaussian models that are combined in some way. GMMs are very powerful and are used across many fields. The parameters of the GMM are estimated from training data using algorithms like **Expectation–Maximization** (**EM**) or **Maximum A-Posteriori** (**MAP**) estimation. Some of the popular applications of GMM include image database retrieval, modeling stock market fluctuations, biometric verification, and so on.

Building a classifier based on Gaussian Mixture Models

Let's build a classifier based on a Gaussian Mixture Model. Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import patches

from sklearn import datasets
from sklearn.mixture import GMM
from sklearn.cross_validation import StratifiedKFold
```

Let's use the iris dataset available in scikit-learn for analysis:

```
# Load the iris dataset
iris = datasets.load_iris()
```

Split the dataset into training and testing using an 80/20 split. The `n_folds` parameter specifies the number of subsets you'll obtain. We are using a value of 5, which means the dataset will be split into five parts. We will use four parts for training and one part for testing, which gives a split of 80/20:

```
# Split dataset into training and testing (80/20 split)
indices = StratifiedKFold(iris.target, n_folds=5)
```

Extract the training data:

```
# Take the first fold
train_index, test_index = next(iter(indices))

# Extract training data and labels
X_train = iris.data[train_index]
y_train = iris.target[train_index]

# Extract testing data and labels
X_test = iris.data[test_index]
y_test = iris.target[test_index]
```

Extract the number of classes in the training data:

```
# Extract the number of classes
num_classes = len(np.unique(y_train))
```

Build a GMM-based classifier using the relevant parameters. The `n_components` parameter specifies the number of components in the underlying distribution. In this case, it will be the number of distinct classes in our data. We need to specify the type of covariance to use. In this case, we will be using full covariance. The `init_params` parameter controls the parameters that need to be updated during the training process. We have used `wc`, which means weights and covariance parameters will be updated during training. The `n_iter` parameter refers to the number of Expectation-Maximization iterations that will be performed during training:

```
# Build GMM
classifier = GMM(n_components=num_classes, covariance_type='full',
        init_params='wc', n_iter=20)
```

Initialize the means of the classifier:

```
# Initialize the GMM means
classifier.means_ = np.array([X_train[y_train == i].mean(axis=0)
                              for i in range(num_classes)])
```

Train the Gaussian mixture model classifier using the training data:

```
# Train the GMM classifier
classifier.fit(X_train)
```

Visualize the boundaries of the classifier. We will extract the eigenvalues and eigenvectors to estimate how to draw the elliptical boundaries around the clusters. If you need a quick refresher on eigenvalues and eigenvectors, please refer to: `https://www.math.hmc.edu/calculus/tutorials/eigenstuff`. Let's go ahead and plot:

```
# Draw boundaries
plt.figure()
colors = 'bgr'
for i, color in enumerate(colors):
    # Extract eigenvalues and eigenvectors
    eigenvalues, eigenvectors = np.linalg.eigh(
            classifier._get_covars()[i][:2, :2])
```

Normalize the first eigenvector:

```
    # Normalize the first eigenvector
    norm_vec = eigenvectors[0] / np.linalg.norm(eigenvectors[0])
```

The ellipses need to be rotated to accurately show the distribution. Estimate the angle:

```
    # Extract the angle of tilt
    angle = np.arctan2(norm_vec[1], norm_vec[0])
    angle = 180 * angle / np.pi
```

Magnify the ellipses for visualization. The eigenvalues control the size of the ellipses:

```
    # Scaling factor to magnify the ellipses
    # (random value chosen to suit our needs)
    scaling_factor = 8
    eigenvalues *= scaling_factor
```

Draw the ellipses:

```
        # Draw the ellipse
        ellipse = patches.Ellipse(classifier.means_[i, :2],
                eigenvalues[0], eigenvalues[1], 180 + angle,
                color=color)
        axis_handle = plt.subplot(1, 1, 1)
        ellipse.set_clip_box(axis_handle.bbox)
        ellipse.set_alpha(0.6)
        axis_handle.add_artist(ellipse)
```

Overlay input data on the figure:

```
# Plot the data
colors = 'bgr'
for i, color in enumerate(colors):
    cur_data = iris.data[iris.target == i]
    plt.scatter(cur_data[:,0], cur_data[:,1], marker='o',
            facecolors='none', edgecolors='black', s=40,
            label=iris.target_names[i])
```

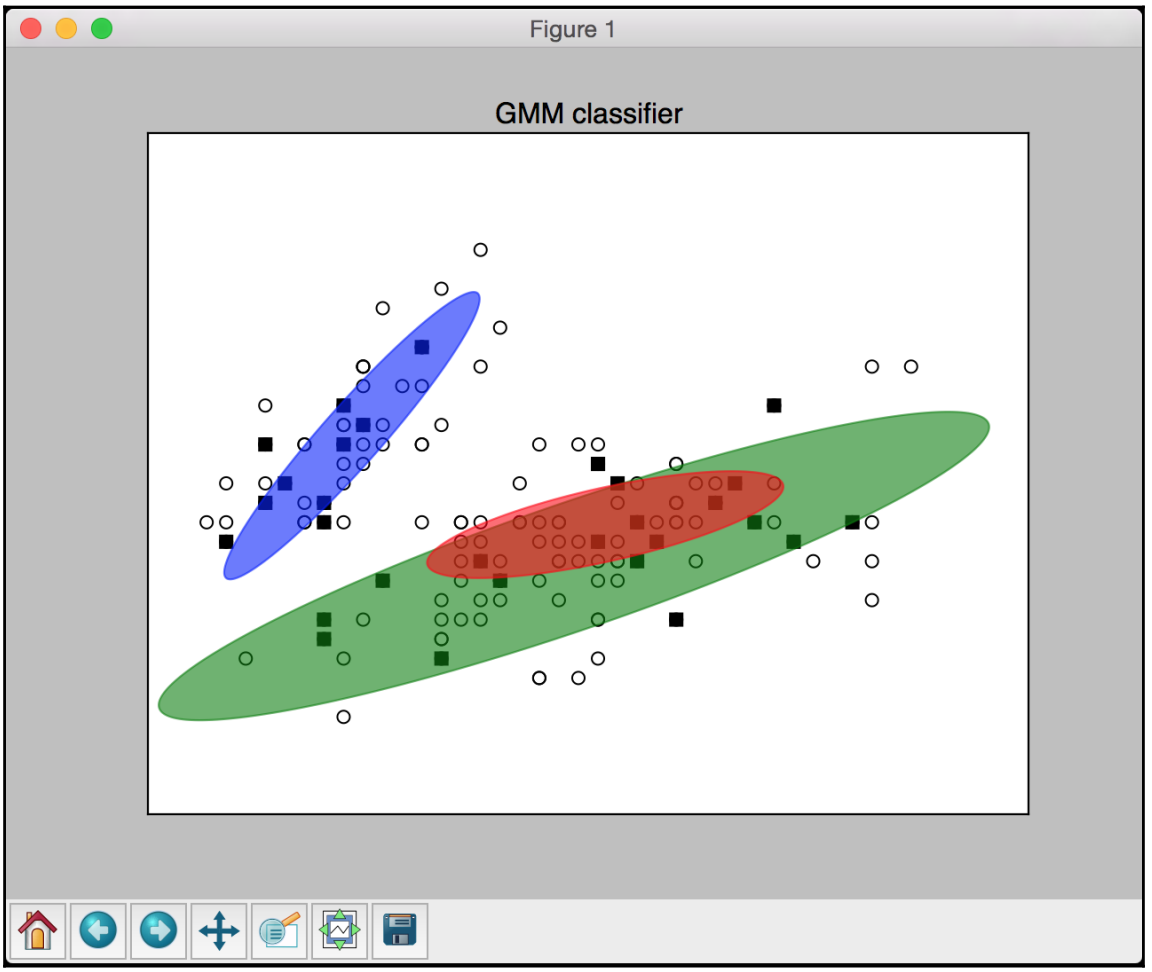
Overlay test data on this figure:

```
        test_data = X_test[y_test == i]
        plt.scatter(test_data[:,0], test_data[:,1], marker='s',
                facecolors='black', edgecolors='black', s=40,
                label=iris.target_names[i])
```

Compute the predicted output for training and testing data:

```
# Compute predictions for training and testing data
y_train_pred = classifier.predict(X_train)
accuracy_training = np.mean(y_train_pred.ravel() == y_train.ravel()) * 100
print('Accuracy on training data =', accuracy_training)

y_test_pred = classifier.predict(X_test)
accuracy_testing = np.mean(y_test_pred.ravel() == y_test.ravel()) * 100
print('Accuracy on testing data =', accuracy_testing)

plt.title('GMM classifier')
plt.xticks(())
plt.yticks(())

plt.show()
```

The full code is given in the file `gmm_classifier.py`. If you run the code, you will see the following output:



The input data consists of three distributions. The three ellipses of various sizes and angles represent the underlying distributions in the input data. You will see the following printed on your Terminal:

```
Accuracy on training data = 87.5
Accuracy on testing data = 86.6666666667
```

Finding subgroups in stock market using Affinity Propagation model

Affinity Propagation is a clustering algorithm that doesn't require us to specify the number of clusters beforehand. Because of its generic nature and simplicity of implementation, it has found a lot of applications across many fields. It finds out representatives of clusters, called exemplars, using a technique called message passing. We start by specifying the measures of similarity that we want it to consider. It simultaneously considers all training data points as potential exemplars. It then passes messages between the data points until it finds a set of exemplars.

The message passing happens in two alternate steps, called **responsibility** and **availability.** Responsibility refers to the message sent from members of the cluster to candidate exemplars, indicating how well suited the data point would be as a member of this exemplar's cluster. Availability refers to the message sent from candidate exemplars to potential members of the cluster, indicating how well suited it would be as an exemplar. It keeps doing this until the algorithm converges on an optimal set of exemplars.

There is also a parameter called preference that controls the number of exemplars that will be found. If you choose a high value, then it will cause the algorithm to find too many clusters. If you choose a low value, then it will lead to a small number of clusters. A good value to choose would be the median similarity between the points.

Let's use Affinity Propagation model to find subgroups in the stock market. We will be using the stock quote variation between opening and closing as the governing feature. Create a new Python file and import the following packages:

```
import datetime
import json

import numpy as np
import matplotlib.pyplot as plt
from sklearn import covariance, cluster
from matplotlib.finance import quotes_historical_yahoo_ochl as quotes_yahoo
```

We will be using the stock market data available in `matplotlib`. The company symbols are mapped to their full names in the file `company_symbol_mapping.json`:

```
# Input file containing company symbols
input_file = 'company_symbol_mapping.json'
```

Load the company symbol map from the file:

```
# Load the company symbol map
with open(input_file, 'r') as f:
    company_symbols_map = json.loads(f.read())

symbols, names = np.array(list(company_symbols_map.items())).T
```

Load the stock quotes from matplotlib:

```
# Load the historical stock quotes
start_date = datetime.datetime(2003, 7, 3)
end_date = datetime.datetime(2007, 5, 4)
quotes = [quotes_yahoo(symbol, start_date, end_date, asobject=True)
                for symbol in symbols]
```

Compute the difference between opening and closing quotes:

```
# Extract opening and closing quotes
opening_quotes = np.array([quote.open for quote in
quotes]).astype(np.float)
closing_quotes = np.array([quote.close for quote in
quotes]).astype(np.float)

# Compute differences between opening and closing quotes
quotes_diff = closing_quotes - opening_quotes
```

Normalize the data:

```
# Normalize the data
X = quotes_diff.copy().T
X /= X.std(axis=0)
```

Create a graph model:

```
# Create a graph model
edge_model = covariance.GraphLassoCV()
```

Train the model:

```
# Train the model
with np.errstate(invalid='ignore'):
    edge_model.fit(X)
```

Build the affinity propagation clustering model using the edge model we just trained:

```
# Build clustering model using Affinity Propagation model
_, labels = cluster.affinity_propagation(edge_model.covariance_)
num_labels = labels.max()
```

Print the output:

```
# Print the results of clustering
for i in range(num_labels + 1):
    print("Cluster", i+1, "==>", ', '.join(names[labels == i]))
```

The full code is given in the file `stocks.py`. If you run the code, you will see the following output on your Terminal:

```
Clustering of stocks based on difference in opening and closing quotes:

Cluster 1 ==> Kraft Foods
Cluster 2 ==> CVS, Walgreen
Cluster 3 ==> Amazon, Yahoo
Cluster 4 ==> Cablevision
Cluster 5 ==> Pfizer, Sanofi-Aventis, GlaxoSmithKline, Novartis
Cluster 6 ==> HP, General Electrics, 3M, Microsoft, Cisco, IBM, Texas instruments, Dell
Cluster 7 ==> Coca Cola, Kimberly-Clark, Pepsi, Procter Gamble, Kellogg, Colgate-Palmolive
Cluster 8 ==> Comcast, Wells Fargo, Xerox, Home Depot, Wal-Mart, Marriott, Navistar, DuPont de Nemours, A
merican express, Ryder, JPMorgan Chase, AIG, Time Warner, Bank of America, Goldman Sachs
Cluster 9 ==> Canon, Unilever, Mitsubishi, Apple, Mc Donalds, Boeing, Toyota, Caterpillar, Ford, Honda, S
AP, Sony
Cluster 10 ==> Valero Energy, Exxon, ConocoPhillips, Chevron, Total
Cluster 11 ==> Raytheon, General Dynamics, Lookheed Martin, Northrop Grumman
```

This output represents the various subgroups in the stock market during that time period. Please note that the clusters might appear in a different order when you run the code.

Segmenting the market based on shopping patterns

Let's see how to apply unsupervised learning techniques to segment the market based on customer shopping habits. You have been provided with a file named `sales.csv`. This file contains the sales details of a variety of tops from a number of retail clothing stores. Our goal is to identify the patterns and segment the market based on the number of units sold in these stores.

Create a new Python file and import the following packages:

```
import csv

import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import MeanShift, estimate_bandwidth
```

Load the data from the input file. Since it's a csv file, we can use the csv reader in python to read the data from this file and convert it into a `NumPy` array:

```
# Load data from input file
input_file = 'sales.csv'
file_reader = csv.reader(open(input_file, 'r'), delimiter=',')

X = []
for count, row in enumerate(file_reader):
    if not count:
        names = row[1:]
        continue

    X.append([float(x) for x in row[1:]])

# Convert to numpy array
X = np.array(X)
```

Let's estimate the bandwidth of the input data:

```
# Estimating the bandwidth of input data
bandwidth = estimate_bandwidth(X, quantile=0.8, n_samples=len(X))
```

Train a mean shift model based on the estimated bandwidth:

```
# Compute clustering with MeanShift
meanshift_model = MeanShift(bandwidth=bandwidth, bin_seeding=True)
meanshift_model.fit(X)
```

Extract the labels and the centers of each cluster:

```
labels = meanshift_model.labels_
cluster_centers = meanshift_model.cluster_centers_
num_clusters = len(np.unique(labels))
```

Print the number of clusters and the cluster centers:

```
print("\nNumber of clusters in input data =", num_clusters)

print("\nCenters of clusters:")
print('\t'.join([name[:3] for name in names]))
for cluster_center in cluster_centers:
    print('\t'.join([str(int(x)) for x in cluster_center]))
```

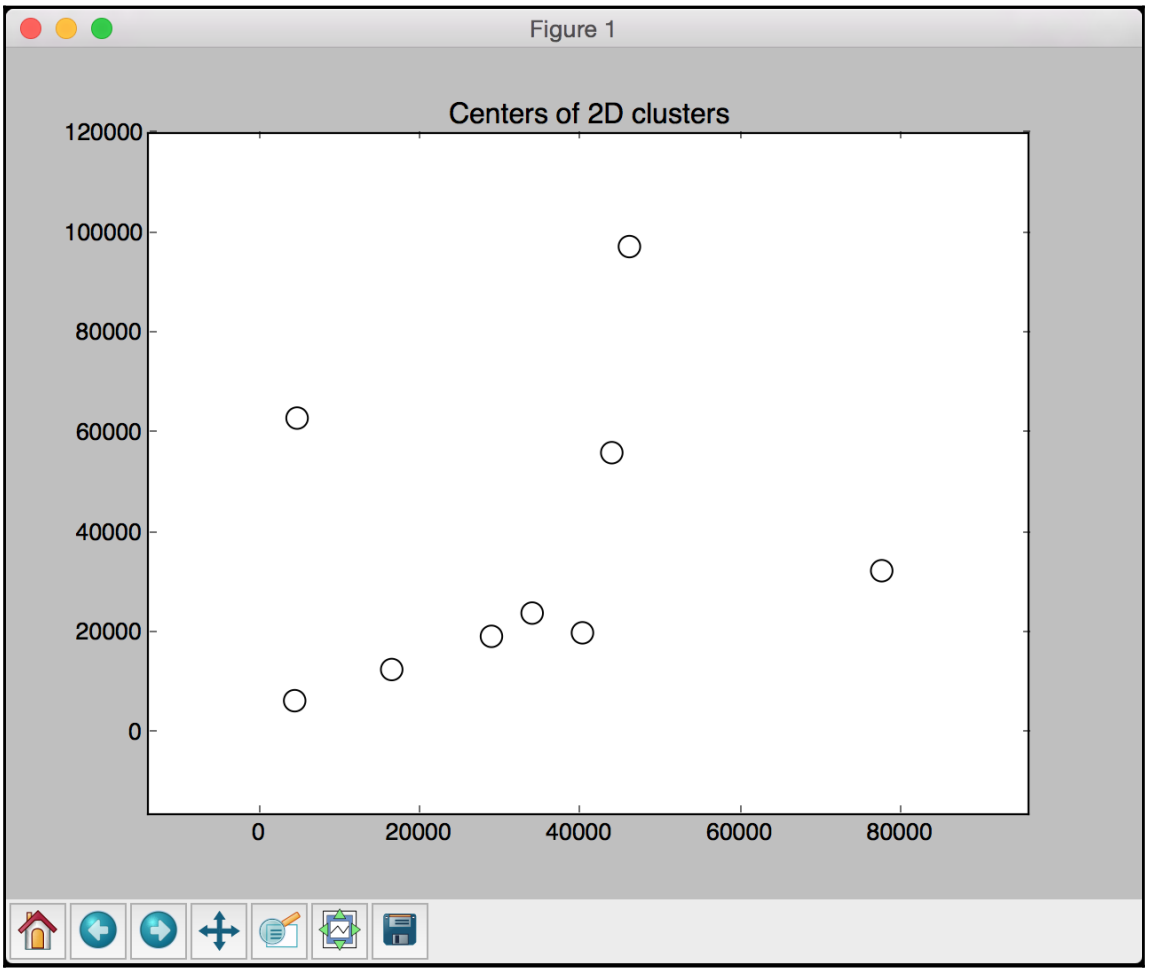
We are dealing with six-dimensional data. In order to visualize the data, let's take two-dimensional data formed using second and third dimensions:

```
# Extract two features for visualization
cluster_centers_2d = cluster_centers[:, 1:3]
```

Plot the centers of clusters:

```
# Plot the cluster centers
plt.figure()
plt.scatter(cluster_centers_2d[:,0], cluster_centers_2d[:,1],
        s=120, edgecolors='black', facecolors='none')

offset = 0.25
plt.xlim(cluster_centers_2d[:,0].min() - offset *
cluster_centers_2d[:,0].ptp(),
        cluster_centers_2d[:,0].max() + offset *
cluster_centers_2d[:,0].ptp(),)
plt.ylim(cluster_centers_2d[:,1].min() - offset *
cluster_centers_2d[:,1].ptp(),
        cluster_centers_2d[:,1].max() + offset *
cluster_centers_2d[:,1].ptp())

plt.title('Centers of 2D clusters')
plt.show()
```

The full code is given in the file `market_segmentation.py`. If you run the code, you will see the following output:



You will see the following on your Terminal:

```
Number of clusters in input data = 9

Centers of clusters:
Tsh     Tan     Hal     Tur     Tub     Swe
9823    4637    6539    2607    2228    1239
38589   44199   56158   5030    24674   4125
7852    4939    63081   134     40066   1332
35314   16745   12775   66900   1298    5613
22617   77873   32543   1005    21035   837
104972  29186   19415   16016   5060    9372
38741   40539   20120   35059   255     50710
28333   34263   24065   5575    4229    18076
14987   46397   97393   1127    37315   3235
```

Summary

In this chapter, we started by discussing unsupervised learning and its applications. We then learned about clustering and how to cluster data using the K-Means algorithm. We discussed how to estimate the number of clusters with Mean Shift algorithm. We talked about silhouette scores and how to estimate the quality of clustering. We learned about Gaussian Mixture Models and how to build a classifier based on that. We also discussed Affinity Propagation model and used it to find subgroups within the stock market. We then applied the Mean Shift algorithm to segment the market based on shopping patterns. In the next chapter, we will learn how to build a recommendation engine.

5
Building Recommender Systems

In this chapter, we are going to learn how to build a movie recommendation system. We will discuss how to create a training pipeline that can be trained with custom parameters. We will then learn about the Nearest Neighbors classifier and see how to implement it. We use these concepts to discuss collaborative filtering and then use it to build a recommender system.

By the end of this chapter, you will learn about the following:

- Creating a training pipeline
- Extracting the nearest neighbors
- Building a K Nearest Neighbors classifier
- Computing similarity scores
- Finding similar users using collaborative filtering
- Building a movie recommendation system

Creating a training pipeline

Machine-learning systems are usually built using different modules. These modules are combined in a particular way to achieve an end goal. The `scikit-learn` library has functions that enable us to build these pipelines by concatenating various modules together. We just need to specify the modules along with the corresponding parameters. It will then build a pipeline using these modules that processes the data and trains the system.

The pipeline can include modules that perform various functions like feature selection, preprocessing, random forests, clustering, and so on. In this section, we will see how to build a pipeline to select the top K features from an input data point and then classify them using an Extremely Random Forest classifier.

Create a new Python file and import the following packages:

```
from sklearn.datasets import samples_generator
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.pipeline import Pipeline
from sklearn.ensemble import ExtraTreesClassifier
```

Let's generate some labeled sample data for training and testing. The `scikit-learn` package has a built-in function that handles it. In the line to follow, we create 150 data points, where each data point is a 25-dimensional feature vector. The numbers in each feature vector will be generated using a random sample generator. Each data point has six informative features and no redundant features. Use the following code:

```
# Generate data
X, y = samples_generator.make_classification(n_samples=150,
        n_features=25, n_classes=3, n_informative=6,
        n_redundant=0, random_state=7)
```

The first block in the pipeline is the feature selector. This block selects the K best features. Let's set the value of K to 9, as follows:

```
# Select top K features
k_best_selector = SelectKBest(f_regression, k=9)
```

The next block in the pipeline is an Extremely Random Forests classifier with 60 estimators and a maximum depth of four. Use the following code:

```
# Initialize Extremely Random Forests classifier
classifier = ExtraTreesClassifier(n_estimators=60, max_depth=4)
```

Let's construct the pipeline by joining the individual blocks that we've constructed. We can name each block so that it's easier to track:

```
# Construct the pipeline
processor_pipeline = Pipeline([('selector', k_best_selector), ('erf',
classifier)])
```

We can change the parameters of the individual blocks. Let's change the value of `K` in the first block to 7 and the number of estimators in the second block to 30. We will use the names we assigned in the previous line to define the scope:

```
# Set the parameters
processor_pipeline.set_params(selector__k=7, erf__n_estimators=30)
```

Train the pipeline using the sample data that we generated earlier:

```
# Training the pipeline
processor_pipeline.fit(X, y)
```

Predict the output for all the input values and print it:

```
# Predict outputs for the input data
output = processor_pipeline.predict(X)
print("\nPredicted output:\n", output)
```

Compute the score using the labeled training data:

```
# Print scores
print("\nScore:", processor_pipeline.score(X, y))
```

Extract the features chosen by the selector block. We specified that we wanted to choose 7 features out of 25. Use the following code:

```
# Print the features chosen by the pipeline selector
status = processor_pipeline.named_steps['selector'].get_support()

# Extract and print indices of selected features
selected = [i for i, x in enumerate(status) if x]
print("\nIndices of selected features:", ', '.join([str(x) for x in
selected]))
```

The full code is given in the file `pipeline_trainer.py`. If you run the code, you will see the following output on your Terminal:

```
Predicted output:
 [1 2 2 0 2 2 0 2 0 1 2 0 2 1 0 0 2 2 2 1 0 2 0 1 2 1 1 1 0 0 1 2 1 0 0 0 2
 1 1 0 2 0 0 0 1 2 0 2 1 0 1 0 0 0 2 1 1 1 1 1 0 1 2 2 2 0 2 0 2 2 0 1 2 0
 2 0 2 0 1 0 2 2 1 1 1 2 0 0 0 0 2 2 0 2 1 1 2 0 1 1 2 1 1 0 1 0 2 2 2 0 0
 1 2 1 1 0 2 0 0 0 0 0 2 2 1 1 1 2 0 2 2 1 0 2 0 0 0 1 1 2 2 2 2 2 2 2 1 1 0
 2 0]

Score: 0.893333333333

Indices of selected features: 13, 15, 18, 19, 21, 23, 24
```

The predicted output list in the preceding screenshot shows the output labels predicted using the processor. The score represents the effectiveness of the processor. The last line indicates the indices of the chosen features.

Extracting the nearest neighbors

Recommender systems employ the concept of nearest neighbors to find good recommendations. Nearest neighbors refers to the process of finding the closest points to the input point from the given dataset. This is frequently used to build classification systems that classify a datapoint based on the proximity of the input data point to various classes. Let's see how to find the nearest neighbors of a given data point.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import NearestNeighbors
```

Define sample 2D datapoints:

```
# Input data
X = np.array([[2.1, 1.3], [1.3, 3.2], [2.9, 2.5], [2.7, 5.4], [3.8, 0.9],
        [7.3, 2.1], [4.2, 6.5], [3.8, 3.7], [2.5, 4.1], [3.4, 1.9],
        [5.7, 3.5], [6.1, 4.3], [5.1, 2.2], [6.2, 1.1]])
```

Define the number of nearest neighbors you want to extract:

```
# Number of nearest neighbors
k = 5
```

Define a test datapoint that will be used to extract the K nearest neighbors:

```
# Test datapoint
test_datapoint = [4.3, 2.7]
```

Plot the input data using circular shaped black markers:

```
# Plot input data
plt.figure()
plt.title('Input data')
plt.scatter(X[:,0], X[:,1], marker='o', s=75, color='black')
```

Create and train a **K Nearest Neighbors** model using the input data. Use this model to extract the nearest neighbors to our test data point:

```
# Build K Nearest Neighbors model
knn_model = NearestNeighbors(n_neighbors=k, algorithm='ball_tree').fit(X)
distances, indices = knn_model.kneighbors(test_datapoint)
```

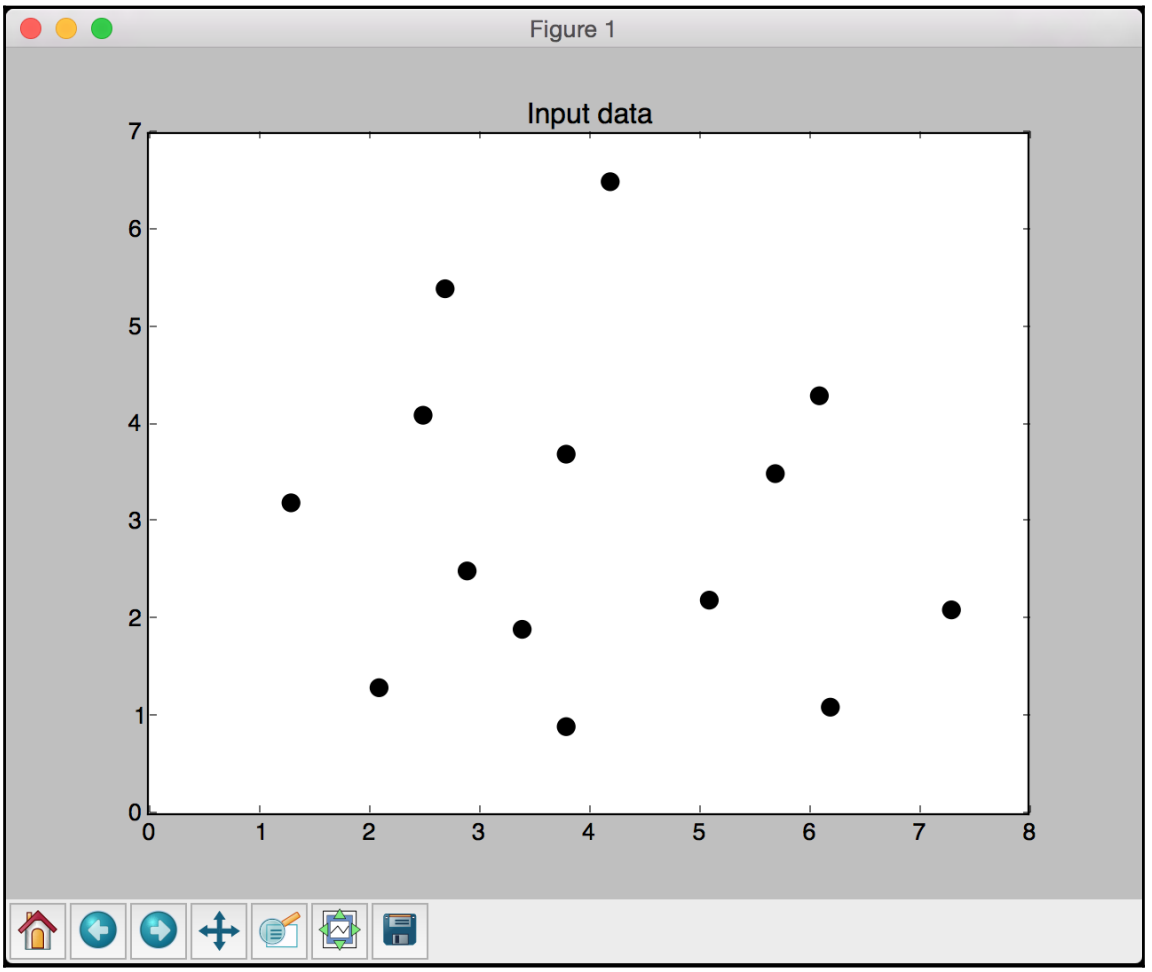
Print the nearest neighbors extracted from the model:

```
# Print the 'k' nearest neighbors
print("\nK Nearest Neighbors:")
for rank, index in enumerate(indices[0][:k], start=1):
    print(str(rank) + " ==>", X[index])
```

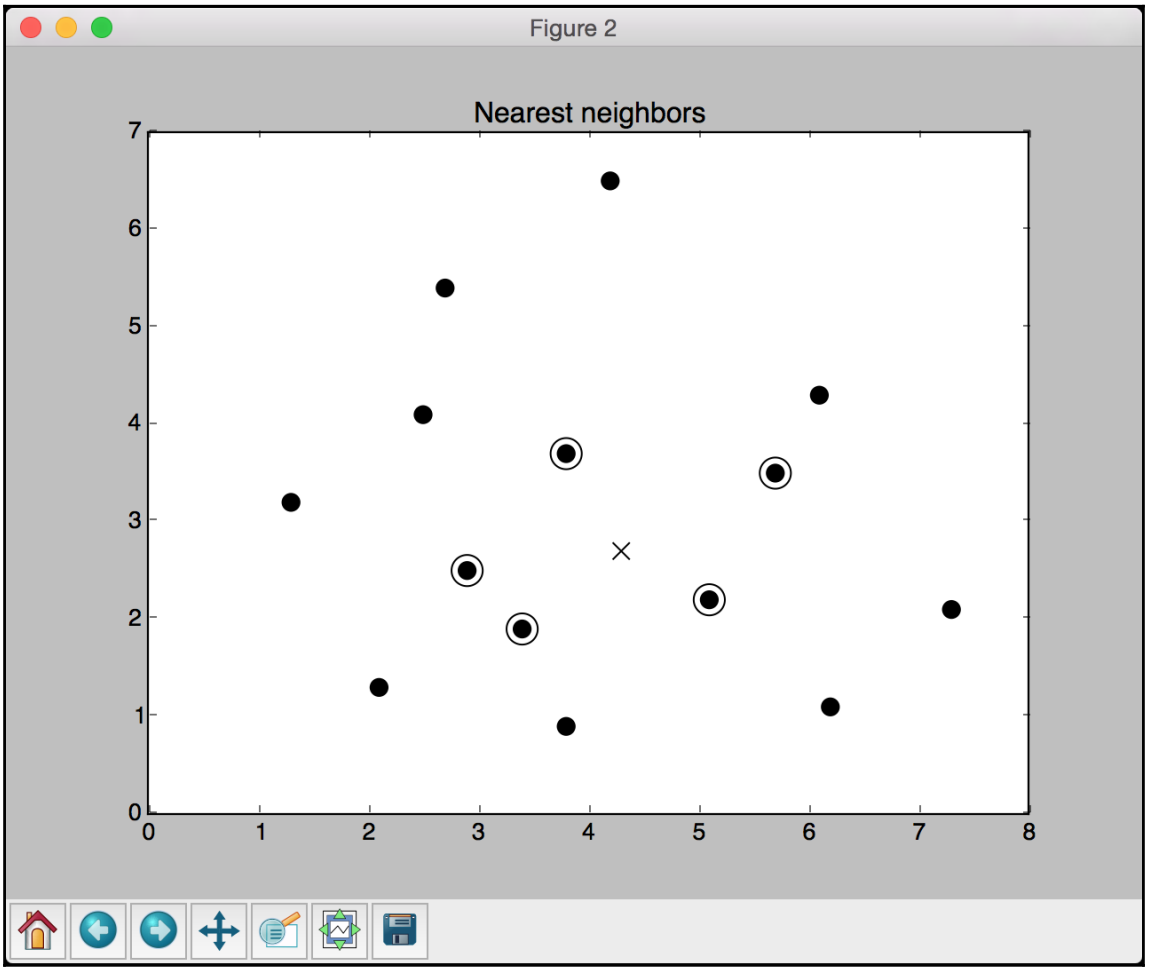
Visualize the nearest neighbors:

```
# Visualize the nearest neighbors along with the test datapoint
plt.figure()
plt.title('Nearest neighbors')
plt.scatter(X[:, 0], X[:, 1], marker='o', s=75, color='k')
plt.scatter(X[indices][0][:][:, 0], X[indices][0][:][:, 1],
        marker='o', s=250, color='k', facecolors='none')
plt.scatter(test_datapoint[0], test_datapoint[1],
        marker='x', s=75, color='k')

plt.show()
```

The full code is given in the file `k_nearest_neighbors.py`. If you run the code, you will see two screenshot. The first screenshot represents the input data:



The second screenshot represents the five nearest neighbors. The test data point is shown using a cross and the nearest neighbor points have been circled:



You will see the following output on your Terminal:

```
K Nearest Neighbors:
1 ==> [ 5.1  2.2]
2 ==> [ 3.8  3.7]
3 ==> [ 3.4  1.9]
4 ==> [ 2.9  2.5]
5 ==> [ 5.7  3.5]
```

The preceding figure shows the five points that are closest to the test data point.

Building a K-Nearest Neighbors classifier

A K-Nearest Neighbors classifier is a classification model that uses the nearest neighbors algorithm to classify a given data point. The algorithm finds the `K` closest data points in the training dataset to identify the category of the input data point. It will then assign a class to this data point based on a majority vote. From the list of those `K` data points, we look at the corresponding classes and pick the one with the highest number of votes. Let's see how to build a classifier using this model. The value of `K` depends on the problem at hand.

Create a new Python file and import the following packages:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from sklearn import neighbors, datasets
```

Load the input data from `data.txt`. Each line contains comma-separated values and the data contains four classes:

```
# Load input data
input_file = 'data.txt'
data = np.loadtxt(input_file, delimiter=',')
X, y = data[:, :-1], data[:, -1].astype(np.int)
```

Visualize the input data using four different marker shapes. We need to map the labels to corresponding markers, which is where the `mapper` variable comes into the picture:

```
# Plot input data
plt.figure()
plt.title('Input data')
```

```
marker_shapes = 'v^os'
mapper = [marker_shapes[i] for i in y]
for i in range(X.shape[0]):
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[i],
            s=75, edgecolors='black', facecolors='none')
```

Define the number of nearest neighbors we want to use:

```
# Number of nearest neighbors
num_neighbors = 12
```

Define the step size of the grid that will be used to visualize the boundaries of the classifier model:

```
# Step size of the visualization grid
step_size = 0.01
```

Create the K Nearest Neighbors classifier model:

```
# Create a K Nearest Neighbors classifier model
classifier = neighbors.KNeighborsClassifier(num_neighbors,
weights='distance')
```

Train the model using training data:

```
# Train the K Nearest Neighbours model
classifier.fit(X, y)
```

Create the mesh grid of values that will be used to visualize the grid:

```
# Create the mesh to plot the boundaries
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
x_values, y_values = np.meshgrid(np.arange(x_min, x_max, step_size),
        np.arange(y_min, y_max, step_size))
```

Evaluate the classifier on all the points on the grid to create a visualization of the boundaries:

```
# Evaluate the classifier on all the points on the grid
output = classifier.predict(np.c_[x_values.ravel(), y_values.ravel()])
```

Create a color mesh to visualize the output:

```
# Visualize the predicted output
output = output.reshape(x_values.shape)
plt.figure()
plt.pcolormesh(x_values, y_values, output, cmap=cm.Paired)
```

Overlay training data on top of this color mesh to visualize the data relative to the boundaries:

```
# Overlay the training points on the map
for i in range(X.shape[0]):
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[i],
            s=50, edgecolors='black', facecolors='none')
```

Set the X and Y limits along with the title:

```
plt.xlim(x_values.min(), x_values.max())
plt.ylim(y_values.min(), y_values.max())
plt.title('K Nearest Neighbors classifier model boundaries')
```

Define a test datapoint to see how the classifier performs. Create a figure with training data points and a test data point to see where it lies:

```
# Test input datapoint
test_datapoint = [5.1, 3.6]
plt.figure()
plt.title('Test datapoint')
for i in range(X.shape[0]):
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[i],
            s=75, edgecolors='black', facecolors='none')

plt.scatter(test_datapoint[0], test_datapoint[1], marker='x',
        linewidth=6, s=200, facecolors='black')
```

Extract the K Nearest Neighbors to the test data point, based on the classifier model:

```
# Extract the K nearest neighbors
_, indices = classifier.kneighbors([test_datapoint])
indices = indices.astype(np.int)[0]
```

Plot the K nearest neighbors obtained in the previous step:

```
# Plot k nearest neighbors
plt.figure()
plt.title('K Nearest Neighbors')

for i in indices:
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[y[i]],
            linewidth=3, s=100, facecolors='black')
```

Overlay the test data point:

```
plt.scatter(test_datapoint[0], test_datapoint[1], marker='x',
        linewidth=6, s=200, facecolors='black')
```

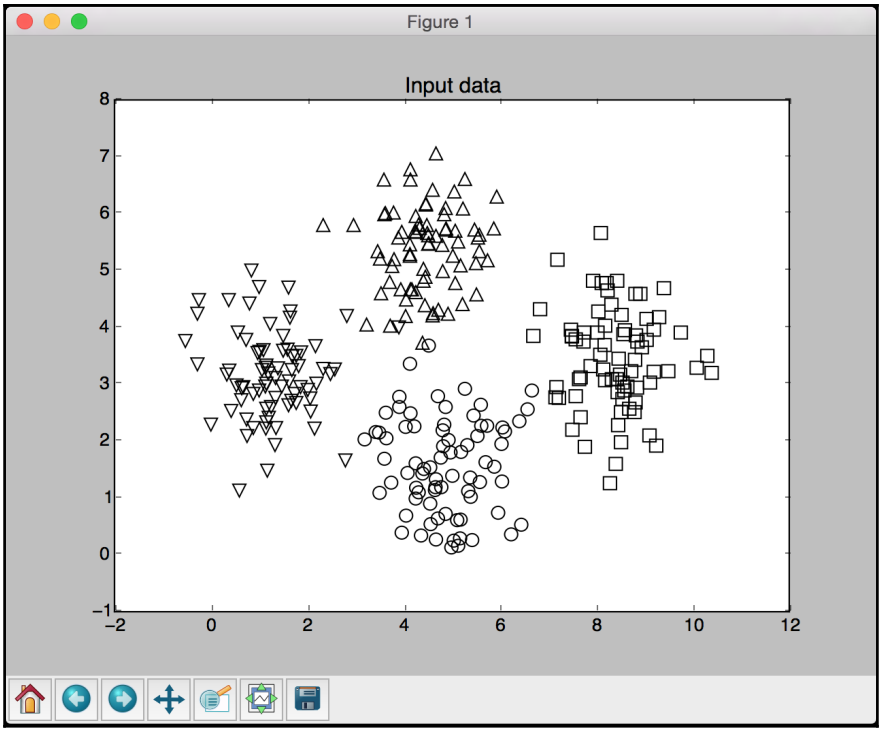
Overlay the input data:

```
for i in range(X.shape[0]):
    plt.scatter(X[i, 0], X[i, 1], marker=mapper[i],
            s=75, edgecolors='black', facecolors='none')
```

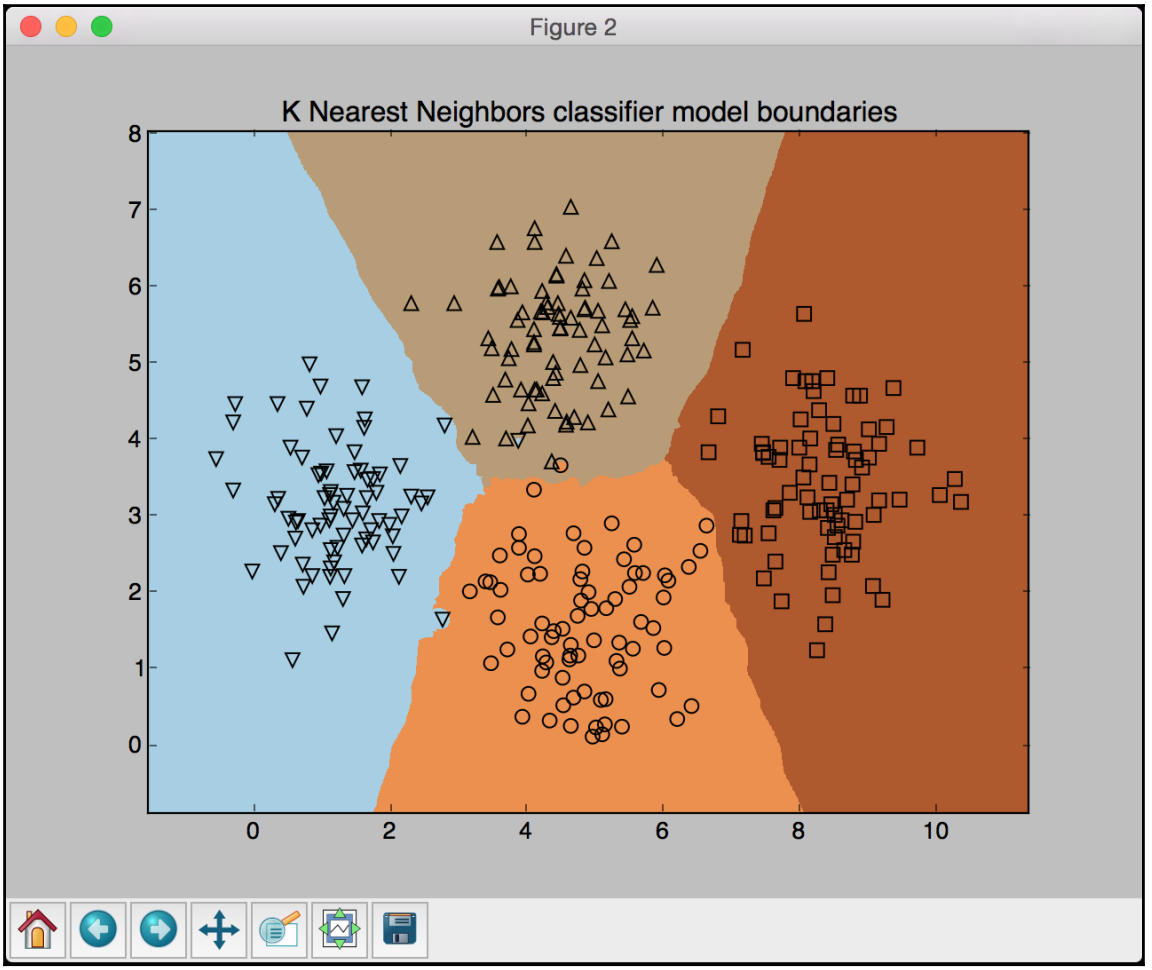
Print the predicted output:

```
print("Predicted output:", classifier.predict([test_datapoint])[0])

plt.show()
```

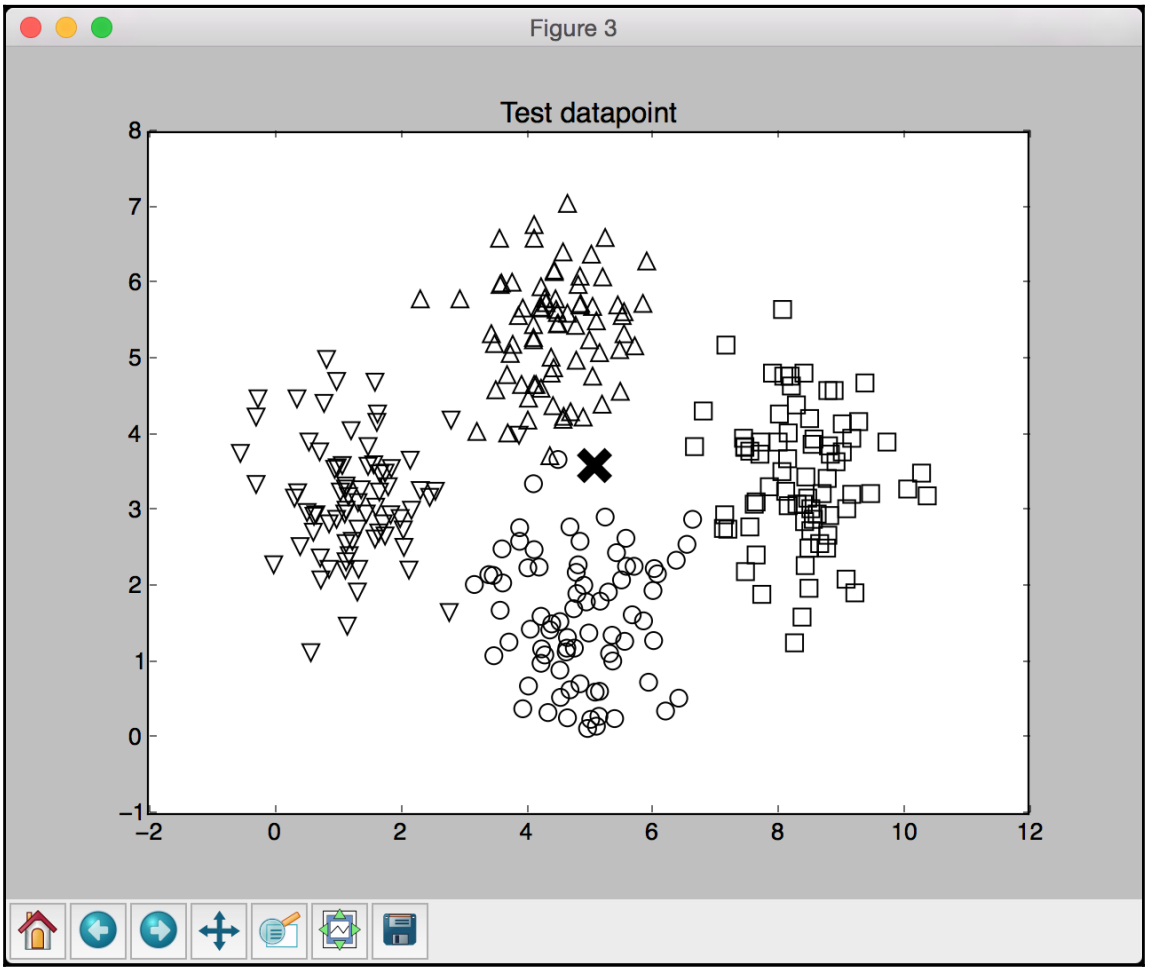
The full code is given in the file `nearest_neighbors_classifier.py`. If you run the code, you will see four screenshot. The first screenshot represents the input data:



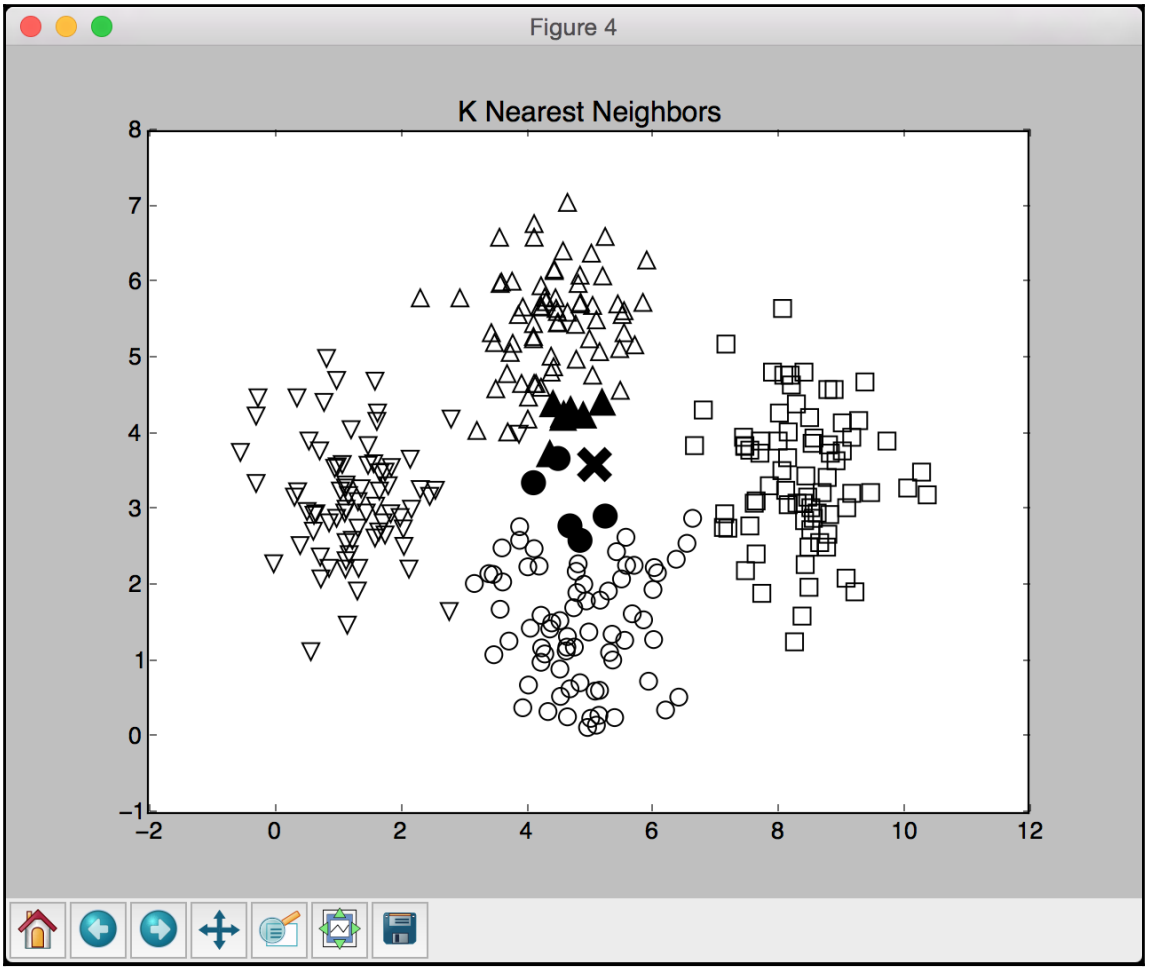
The second screenshot represents the classifier boundaries:



The third screenshot shows the test data point relative to the input dataset. The test data point is shown using a cross:



The fourth screenshot shows the 12 nearest neighbors to the test data point:



You will see the following output on the Terminal, indicating that the test data point belongs to class `1`:

```
Predicted output: 1
```

Computing similarity scores

In order to build a recommendation system, it is important to understand how to compare various objects in our dataset. Let's say our dataset consists of people and their various movie preferences. In order to recommend something, we need to understand how to compare any two people with each other. This is where the similarity score becomes very important. The similarity score gives us an idea of how similar two objects are.

There are two scores that are used frequently in this domain — Euclidean score and Pearson score. **Euclidean score** uses the Euclidean distance between two data points to compute the score. If you need a quick refresher on how Euclidean distance is computed, you can go to `https://en.wikipedia.org/wiki/Euclidean_distance`. The value of the Euclidean distance can be unbounded. Hence we take this value and convert it in a way that the Euclidean score ranges from *0* to *1*. If the Euclidean distance between two objects is large, then the Euclidean score should be low because a low score indicates that the objects are not similar. Hence Euclidean distance is inversely proportional to Euclidean score.

Pearson score is a measure of correlation between two objects. It uses the covariance between the two objects along with their individual standard deviations to compute the score. The score can range from *-1* to *+1*. A score of *+1* indicates that the objects are very similar where a score of *-1* would indicate that the objects are very dissimilar. A score of 0 would indicate that there is no correlation between the two objects. Let's see how to compute these scores.

Create a new Python file and import the following packages:

```
import argparse
import json
import numpy as np
```

Build an argument parser to process the input arguments. It will accept two users and the type of score that it needs to use to compute the similarity score:

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Compute similarity
score')
    parser.add_argument('--user1', dest='user1', required=True,
            help='First user')
    parser.add_argument('--user2', dest='user2', required=True,
            help='Second user')
    parser.add_argument("--score-type", dest="score_type", required=True,
            choices=['Euclidean', 'Pearson'], help='Similarity metric to be
used')
    return parser
```

Define a function to compute the Euclidean score between the input users. If the users are not in the dataset, raise an error:

```
# Compute the Euclidean distance score between user1 and user2
def euclidean_score(dataset, user1, user2):
    if user1 not in dataset:
        raise TypeError('Cannot find ' + user1 + ' in the dataset')

    if user2 not in dataset:
        raise TypeError('Cannot find ' + user2 + ' in the dataset')
```

Define a variable to track the movies that have been rated by both the users:

```
    # Movies rated by both user1 and user2
    common_movies = {}
```

Extract the movies rated by both users:

```
    for item in dataset[user1]:
        if item in dataset[user2]:
            common_movies[item] = 1
```

If there are no common movies, then we cannot compute the similarity score:

```
    # If there are no common movies between the users,
    # then the score is 0
    if len(common_movies) == 0:
        return 0
```

Compute the squared differences between the ratings and use it to compute the Euclidean score:

```
    squared_diff = []

    for item in dataset[user1]:
        if item in dataset[user2]:
            squared_diff.append(np.square(dataset[user1][item] -
dataset[user2][item]))
    return 1 / (1 + np.sqrt(np.sum(squared_diff)))
```

Define a function to compute the Pearson score between the input users in the given dataset. If the users are not found in the dataset, raise an error:

```
# Compute the Pearson correlation score between user1 and user2
def pearson_score(dataset, user1, user2):
    if user1 not in dataset:
        raise TypeError('Cannot find ' + user1 + ' in the dataset')
```

```
    if user2 not in dataset:
        raise TypeError('Cannot find ' + user2 + ' in the dataset')
```

Define a variable to track the movies that have been rated by both the users:

```
    # Movies rated by both user1 and user2
    common_movies = {}
```

Extract the movies rated by both users:

```
    for item in dataset[user1]:
        if item in dataset[user2]:
            common_movies[item] = 1
```

If there are no common movies, then we cannot compute the similarity score:

```
    num_ratings = len(common_movies)

    # If there are no common movies between user1 and user2, then the score
is 0
    if num_ratings == 0:
        return 0
```

Calculate the sum of ratings of all the movies that have been rated by both the users:

```
    # Calculate the sum of ratings of all the common movies
    user1_sum = np.sum([dataset[user1][item] for item in common_movies])
    user2_sum = np.sum([dataset[user2][item] for item in common_movies])
```

Calculate the sum of squares of the ratings all the movies that have been rated by both the users:

```
    # Calculate the sum of squares of ratings of all the common movies
    user1_squared_sum = np.sum([np.square(dataset[user1][item]) for item in
common_movies])
    user2_squared_sum = np.sum([np.square(dataset[user2][item]) for item in
common_movies])
```

Calculate the sum of products of the ratings of all the movies rated by both the input users:

```
    # Calculate the sum of products of the ratings of the common movies
    sum_of_products = np.sum([dataset[user1][item] * dataset[user2][item]
for item in common_movies])
```

Calculate the various parameters required to compute the Pearson score using the preceding computations:

```
# Calculate the Pearson correlation score
Sxy = sum_of_products - (user1_sum * user2_sum / num_ratings)
Sxx = user1_squared_sum - np.square(user1_sum) / num_ratings
Syy = user2_squared_sum - np.square(user2_sum) / num_ratings
```

If there is no deviation, then the score is *0*:

```
if Sxx * Syy == 0:
    return 0
```

Return the Pearson score:

```
return Sxy / np.sqrt(Sxx * Syy)
```

Define the main function and parse the input arguments:

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    user1 = args.user1
    user2 = args.user2
    score_type = args.score_type
```

Load the ratings from the file `ratings.json` into a dictionary:

```
ratings_file = 'ratings.json'

with open(ratings_file, 'r') as f:
    data = json.loads(f.read())
```

Compute the similarity score based on the input arguments:

```
if score_type == 'Euclidean':
    print("\nEuclidean score:")
    print(euclidean_score(data, user1, user2))
else:
    print("\nPearson score:")
    print(pearson_score(data, user1, user2))
```

The full code is given in the file `compute_scores.py`. Let's run the code with a few combinations. Let's say we want to compute the Euclidean score between `David Smith` and `Bill Duffy`:

```
    $ python3 compute_scores.py --user1 "David Smith" --user2 "Bill Duffy"
--score-type Euclidean
```

If you run the above command, you will get the following output on your Terminal:

```
Euclidean score:
0.585786437627
```

If you want to compute the Pearson score between the same pair, run the following command on your Terminal:

```
    $ python3 compute_scores.py --user1 "David Smith" --user2 "Bill Duffy"
--score-type Pearson
```

You will see the following on your Terminal:

```
Pearson score:
0.99099243041
```

You can run it using other combinations of parameters as well.

Finding similar users using collaborative filtering

Collaborative filtering refers to the process of identifying patterns among the objects in a dataset in order to make a decision about a new object. In the context of recommendation engines, we use collaborative filtering to provide recommendations by looking at similar users in the dataset.

By collecting the preferences of different users in the dataset, we collaborate that information to filter the users. Hence the name collaborative filtering.

The assumption here is that if two people have similar ratings for a particular set of movies, then their choices in a set of new unknown movies would be similar too. By identifying patterns in those common movies, we make predictions about new movies. In the previous section, we learned how to compare different users in the dataset. We will use these scoring techniques to find similar users in our dataset. Collaborative filtering is typically used when we have huge datasets. These methods can be used for various verticals like finance, online shopping, marketing, customer studies, and so on.

Create a new Python file and import the following packages:

```
import argparse
import json
import numpy as np

from compute_scores import pearson_score
```

Define a function to parse the input arguments. The only input argument would be the name of the user:

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Find users who are
similar to the input user ')
    parser.add_argument('--user', dest='user', required=True,
            help='Input user')
    return parser
```

Define a function to find the users in the dataset that are similar to the given user. If the user is not in the dataset, raise an error:

```
# Finds users in the dataset that are similar to the input user
def find_similar_users(dataset, user, num_users):
    if user not in dataset:
        raise TypeError('Cannot find ' + user + ' in the dataset')
```

We have already imported the function to compute the Pearson score. Let's use that function to compute the Pearson score between the input user and all the other users in the dataset:

```
    # Compute Pearson score between input user
    # and all the users in the dataset
    scores = np.array([[x, pearson_score(dataset, user,
            x)] for x in dataset if x != user])
```

Sort the scores in descending order:

```
    # Sort the scores in decreasing order
    scores_sorted = np.argsort(scores[:, 1])[::-1]
```

Extract the top `num_users` number of users as specified by the input argument and return the array:

```
    # Extract the top 'num_users' scores
    top_users = scores_sorted[:num_users]
    return scores[top_users]
```

Define the main function and parse the input arguments to extract the name of the user:

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    user = args.user
```

Load the data from the movie ratings file `ratings.json`. This file contains the names of people and their ratings for various movies:

```
ratings_file = 'ratings.json'

with open(ratings_file, 'r') as f:
    data = json.loads(f.read())
```

Find the top three users who are similar to the user specified by the input argument. You can change it to any number of users depending on your choice. Print the output along with the scores:

```
print('\nUsers similar to ' + user + ':\n')
similar_users = find_similar_users(data, user, 3)
print('User\t\t\tSimilarity score')
print('-'*41)
for item in similar_users:
    print(item[0], '\t\t', round(float(item[1]), 2))
```

The full code is given in the file `collaborative_filtering.py`. Let's run the code and find out the users who are similar to Bill Duffy:

```
$ python3 collaborative_filtering.py --user "Bill Duffy"
```

You will get the following output on your Terminal:

```
Users similar to Bill Duffy:

User                    Similarity score
-----------------------------------------
David Smith              0.99
Samuel Miller            0.88
Adam Cohen               0.86
```

Let's run the code and find out the users who are similar to `Clarissa Jackson`:

```
$ python3 collaborative_filtering.py --user "Clarissa Jackson"
```

You will get the following output on your Terminal:

```
Users similar to Clarissa Jackson:

User                    Similarity score
-----------------------------------------
Chris Duncan             1.0
Bill Duffy               0.83
Samuel Miller            0.73
```

Building a movie recommendation system

Now that we have all the building blocks in place, it's time to build a movie recommendation system. We learned all the underlying concepts that are needed to build a recommendation system. In this section, we will build a movie recommendation system based on the data provided in the file `ratings.json`. This file contains a set of people and their ratings for various movies. When we want to find movie recommendations for a given user, we will need to find similar users in the dataset and then come up with recommendations for this person.

Create a new Python file and import the following packages:

```
import argparse
import json
import numpy as np

from compute_scores import pearson_score
from collaborative_filtering import find_similar_users
```

Define a function to parse the input arguments. The only input argument would be the name of the user:

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Find the movie
recommendations for the given user')
    parser.add_argument('--user', dest='user', required=True,
            help='Input user')
    return parser
```

Define a function to get the movie recommendations for a given user. If the user doesn't exist in the dataset, raise an error:

```
# Get movie recommendations for the input user
def get_recommendations(dataset, input_user):
    if input_user not in dataset:
        raise TypeError('Cannot find ' + input_user + ' in the dataset')
```

Define the variables to track the scores:

```
    overall_scores = {}
    similarity_scores = {}
```

Compute a similarity score between the input user and all the other users in the dataset:

```
    for user in [x for x in dataset if x != input_user]:
        similarity_score = pearson_score(dataset, input_user, user)
```

If the similarity score is less than 0, you can continue with the next user in the dataset:

```
        if similarity_score <= 0:
            continue
```

Extract a list of movies that have been rated by the current user but haven't been rated by the input user:

```
    filtered_list = [x for x in dataset[user] if x not in \
        dataset[input_user] or dataset[input_user][x] == 0]
```

For each item in the filtered list, keep a track of the weighted rating based on the similarity score. Also keep a track of the similarity scores:

```
        for item in filtered_list:
            overall_scores.update({item: dataset[user][item] *
similarity_score})
            similarity_scores.update({item: similarity_score})
```

If there are no such movies, then we cannot recommend anything:

```
    if len(overall_scores) == 0:
        return ['No recommendations possible']
```

Normalize the scores based on the weighted scores:

```
    # Generate movie ranks by normalization
    movie_scores = np.array([[score/similarity_scores[item], item]
            for item, score in overall_scores.items()])
```

Sort the scores and extract the movie recommendations:

```
        # Sort in decreasing order
        movie_scores = movie_scores[np.argsort(movie_scores[:, 0])[::-1]]

        # Extract the movie recommendations
        movie_recommendations = [movie for _, movie in movie_scores]

        return movie_recommendations
```

Define the main function and parse the input arguments to extract the name of the input user:

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
    user = args.user
```

Load the movie ratings data from the file `ratings.json`:

```
        ratings_file = 'ratings.json'

        with open(ratings_file, 'r') as f:
            data = json.loads(f.read())
```

Extract the movie recommendations and print the output:

```
        print("\nMovie recommendations for " + user + ":")
        movies = get_recommendations(data, user)
        for i, movie in enumerate(movies):
            print(str(i+1) + '. ' + movie)
```

The full code is given in the file `movie_recommender.py`. Let's find out the movie recommendations for `Chris Duncan`:

```
$ python3 movie_recommender.py --user "Chris Duncan"
```

You will see the following output on your Terminal:

```
Movie recommendations for Chris Duncan:
1. Vertigo
2. Goodfellas
3. Scarface
4. Roman Holiday
```

Let's find out the movie recommendations for `Julie Hammel`:

```
$ python3 movie_recommender.py --user "Julie Hammel"
```

You will see the following output on your Terminal:

```
Movie recommendations for Julie Hammel:
1. The Apartment
2. Vertigo
3. Raging Bull
```

Summary

In this chapter, we learned how to create a data processor pipeline that can be used to train a machine-learning system. We learned how to extract K nearest neighbors to any given data point from a given dataset. We then used this concept to build the K Nearest Neighbors classifier. We discussed how to compute similarity scores such as the Euclidean and Pearson scores. We learned how to use collaborative filtering to find similar users from a given dataset and used it to build a movie recommendation system.

In the next chapter, we will learn about logic programming and see how to build an inference engine that can solve a real world problem.

6
Logic Programming

In this chapter, we are going to learn how to write programs using logic programming. We will discuss various programming paradigms and see how programs are constructed with logic programming. We will learn about the building blocks of logic programming and see how to solve problems in this domain. We will implement Python programs to build various solvers that solve a variety of problems.

By the end of this chapter, you will know about the following:

- What is logic programming?
- Understanding the building blocks of logic programming
- Solving problems using logic programming
- Installing Python packages
- Matching mathematical expressions
- Validating primes
- Parsing a family tree
- Analyzing geography
- Building a puzzle solver

What is logic programming?

Logic programming is a programming paradigm, which basically means it is a particular way to approach programming. Before we talk about what it constitutes and how it is relevant in Artificial Intelligence, let's talk a bit about programming paradigms.

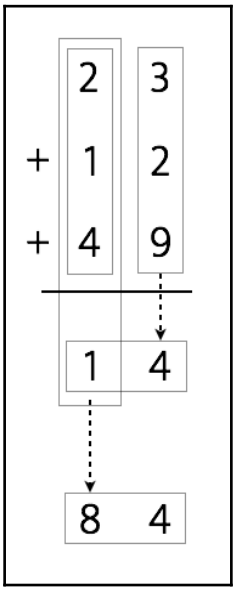
The concept of programming paradigms arises owing to the need to classify programming languages. It refers to the way computer programs solve problems through code. Some programming paradigms are primarily concerned with implications or the sequence of operations used to achieve the result. Other programming paradigms are concerned about how we organize the code.

Here are some of the more popular programming paradigms:

- **Imperative**: This uses statements to change a program's state, thus allowing for side effects.
- **Functional**: This treats computation as an evaluation of mathematical functions and does not allow changing states or mutable data.
- **Declarative**: This is a way of programming where you write your programs by describing what you want to do and not how you want to do it. You express the logic of the underlying computation without explicitly describing the control flow.
- **Object Oriented**: This groups the code within the program in such a way that each object is responsible for itself. The objects contain data and methods that specify how the changes happen.
- **Procedural**: This groups the code into functions and each function is responsible for a particular series of steps.
- **Symbolic**: This uses a particular style of syntax and grammar through which the program can modify its own components by treating them as plain data.
- **Logic**: This views computation as automatic reasoning over a database of knowledge consisting of facts and rules.

In order to understand logic programming, let's understand the concepts of computation and deduction. To compute something, we start with an expression and a set of rules. This set of rules is basically the program.

We use these expressions and rules to generate the output. For example, let's say we want to compute the sum of 23, 12, and 49:



The procedure would be as follows:

23 + 12 + 49 => (2 + 1 + 4 + 1)4 => 84

On the other hand, if we want to deduce something, we need to start from a conjecture. We then need to construct a proof according to a set of rules. In essence, the process computation is mechanical, whereas the process of deduction is more creative.

When we write a program in the logic programming paradigm, we specify a set of statements based on facts and rules about the problem domain and the solver solves it using this information.

Understanding the building blocks of logic programming

In programming object-oriented or imperative paradigms, we always have to specify how a variable is defined. In logic programming, things work a bit differently. We can pass an uninstantiated argument to a function and the interpreter will instantiate these variables for us by looking at the facts defined by the user. This is a powerful way of approaching the variable matching problem. The process of matching variables with different items is called unification. This is one of the places logic programming really stands apart. We need to specify something called relations in logic programming. These relations are defined by means of clauses called facts and rules.

Facts are just statements that are truths about our program and the data that it's operating on. The syntax is pretty straightforward. For example, Donald is Allan's son, can be a fact whereas, Who is Allan's son? cannot be a fact. Every logic program needs facts to work with, so that it can achieve the given goal based on them.

Rules are the things we have learned about how to express various facts and how to query them. They are the constraints that we have to work with and they allow us to make conclusions about the problem domain. For example, let's say you are working on building a chess engine. You need to specify all the rules about how each piece can move on the chessboard. In essence, the final conclusion is valid only if all the relations are true.

Solving problems using logic programming

Logic programming looks for solutions by using facts and rules. We need to specify a goal for each program. In the case where a logic program and a goal don't contain any variables, the solver comes up with a tree that constitutes the search space for solving the problem and getting to the goal.

One of the most important things about logic programming is how we treat the rules. Rules can be viewed as logical statements. Let's consider the following:

Kathy likes chocolate => Alexander loves Kathy

This can be read as an implication that says, *If Kathy likes chocolate, then Alexander loves Kathy*. It can also be construed as *Kathy likes chocolate implies Alexander loves Kathy*. Similarly, let's consider the following rule:

Crime movies, English => Martin Scorsese

It can be read as the implication *If you like crime movies in English, then you would like movies made by Martin Scorsese*.

This construction is used in various forms throughout logic programming to solve various types of problems. Let's go ahead and see how to solve these problems in Python.

Installing Python packages

Before we start logic programming in Python, we need to install a couple of packages. The package `logpy` is a Python package that enables logic programming in Python. We will also be using SymPy for some of the problems. So let's go ahead and install `logpy` and `sympy` using `pip`:

```
$ pip3 install logpy
$ pip3 install sympy
```

If you get an error during the installation process for `logpy`, you can install from source at `https://github.com/logpy/logpy`. Once you have successfully installed these packages, you can proceed to the next section.

Matching mathematical expressions

We encounter mathematical operations all the time. Logic programming is a very efficient way of comparing expressions and finding out unknown values. Let's see how to do that.

Create a new Python file and import the following packages:

```
from logpy import run, var, fact
import logpy.assoccomm as la
```

Define a couple of mathematical operations:

```
# Define mathematical operations
add = 'addition'
mul = 'multiplication'
```

Both addition and multiplication are commutative operations. Let's specify that:

```
# Declare that these operations are commutative
# using the facts system
fact(la.commutative, mul)
fact(la.commutative, add)
fact(la.associative, mul)
fact(la.associative, add)
```

Let's define some variables:

```
# Define some variables
a, b, c = var('a'), var('b'), var('c')
```

Consider the following expression:

```
expression_orig = 3 x (-2) + (1 + 2 x 3) x (-1)
```

Let's generate this expression with masked variables. The first expression would be:

- *expression1 = (1 + 2 x a) x b + 3 x c*

The second expression would be:

- *expression2 = c x 3 + b x (2 x a + 1)*

The third expression would be:

- *expression3 = (((2 x a) x b) + b) + 3 x c*

If you observe carefully, all three expressions represent the same basic expression. Our goal is to match these expressions with the original expression to extract the unknown values:

```
# Generate expressions
expression_orig = (add, (mul, 3, -2), (mul, (add, 1, (mul, 2, 3)), -1))
expression1 = (add, (mul, (add, 1, (mul, 2, a)), b), (mul, 3, c))
expression2 = (add, (mul, c, 3), (mul, b, (add, (mul, 2, a), 1)))
expression3 = (add, (add, (mul, (mul, 2, a), b), b), (mul, 3, c))
```

Compare the expressions with the original expression. The method run is commonly used in `logpy`. This method takes the input arguments and runs the expression. The first argument is the number of values, the second argument is a variable, and the third argument is a function:

```
# Compare expressions
print(run(0, (a, b, c), la.eq_assoccomm(expression1, expression_orig)))
print(run(0, (a, b, c), la.eq_assoccomm(expression2, expression_orig)))
print(run(0, (a, b, c), la.eq_assoccomm(expression3, expression_orig)))
```

The full code is given in `expression_matcher.py`. If you run the code, you will see the following output on your Terminal:

```
((3, -1, -2),)
((3, -1, -2),)
()
```

The three values in the first two lines represent the values for `a`, `b`, and `c`. The first two expressions matched with the original expression, whereas the third one returned nothing. This is because even though the third expression is mathematically the same, it is structurally different. Pattern comparison works by comparing the structure of the expressions.

Validating primes

Let's see how to use logic programming to check for prime numbers. We will use the constructs available in `logpy` to determine which numbers in the given list are prime, as well as finding out if a given number is a prime or not.

Create a new Python file and import the following packages:

```
import itertools as it
import logpy.core as lc
from sympy.ntheory.generate import prime, isprime
```

Next, define a function that checks if the given number is prime depending on the type of data. If it's a number, then it's pretty straightforward. If it's a variable, then we have to run the sequential operation. To give a bit of background, the method `conde` is a goal constructor that provides logical `AND` and `OR` operations. The method `condeseq` is like `conde`, but it supports generic iterator of goals:

```
# Check if the elements of x are prime
def check_prime(x):
    if lc.isvar(x):
        return lc.condeseq([(lc.eq, x, p)] for p in map(prime,
it.count(1)))
    else:
        return lc.success if isprime(x) else lc.fail
```

Declare the variable `x` that will be used:

```
# Declate the variable
x = lc.var()
```

Define a set of numbers and check which numbers are prime. The method `membero` checks if a given number is a member of the list of numbers specified in the input argument:

```
# Check if an element in the list is a prime number
list_nums = (23, 4, 27, 17, 13, 10, 21, 29, 3, 32, 11, 19)
print('\nList of primes in the list:')
print(set(lc.run(0, x, (lc.membero, x, list_nums), (check_prime, x))))
```

Let's use the function in a slightly different way now by printing the first *7* prime numbers:

```
# Print first 7 prime numbers
print('\nList of first 7 prime numbers:')
print(lc.run(7, x, check_prime(x)))
```

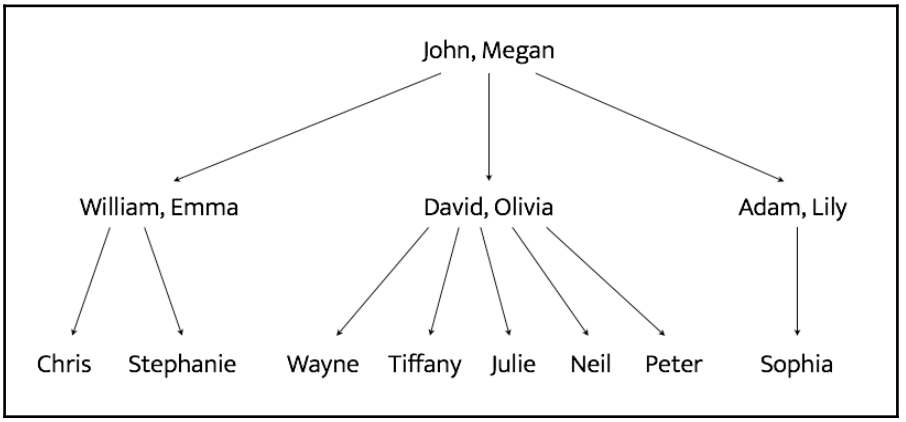
The full code is given in `prime.py`. If you run the code, you will see the following output:

```
List of primes in the list:
{3, 11, 13, 17, 19, 23, 29}
List of first 7 prime numbers:
(2, 3, 5, 7, 11, 13, 17)
```

You can confirm that the output values are correct.

Parsing a family tree

Now that we are more familiar with logic programming, let's use it to solve an interesting problem. Consider the following family tree:



John and Megan have three sons – William, David, and Adam. The wives of William, David, and Adam are Emma, Olivia, and Lily respectively. William and Emma have two children – Chris and Stephanie. David and Olivia have five children – Wayne, Tiffany, Julie, Neil, and Peter. Adam and Lily have one child – Sophia. Based on these facts, we can create a program that can tell us the name of Wayne's grandfather or Sophia's uncles are. Even though we have not explicitly specified anything about the grandparent or uncle relationships, logic programming can infer them.

These relationships are specified in a file called `relationships.json` provided for you. The file looks like the following:

```
{
      "father":
      [
            {"John": "William"},
            {"John": "David"},
            {"John": "Adam"},
            {"William": "Chris"},
            {"William": "Stephanie"},
            {"David": "Wayne"},
            {"David": "Tiffany"},
            {"David": "Julie"},
            {"David": "Neil"},
            {"David": "Peter"},
            {"Adam": "Sophia"}
```

```
        ],
        "mother":
        [
              {"Megan": "William"},
              {"Megan": "David"},
              {"Megan": "Adam"},
              {"Emma": "Stephanie"},
              {"Emma": "Chris"},
              {"Olivia": "Tiffany"},
              {"Olivia": "Julie"},
              {"Olivia": "Neil"},
              {"Olivia": "Peter"},
              {"Lily": "Sophia"}
        ]
}
```

It is a simple `json` file that specifies only the father and mother relationships. Note that we haven't specified anything about husband and wife, grandparents, or uncles.

Create a new Python file and import the following packages:

```
import json
from logpy import Relation, facts, run, conde, var, eq
```

Define a function to check if `x` is the parent of `y`. We will use the logic that if x is the parent of y, then x is either the father or the mother. We have already defined “father” and “mother” in our fact base:

```
# Check if 'x' is the parent of 'y'
def parent(x, y):
    return conde([father(x, y)], [mother(x, y)])
```

Define a function to check if `x` is the grandparent of `y`. We will use the logic that if `x` is the grandparent of `y`, then the offspring of `x` will be the parent of `y`:

```
# Check if 'x' is the grandparent of 'y'
def grandparent(x, y):
    temp = var()
    return conde((parent(x, temp), parent(temp, y)))
```

Define a function to check if `x` is the sibling of `y`. We will use the logic that if `x` is the sibling of `y`, then `x` and `y` will have the same parents. Notice that there is a slight modification needed here because when we list out all the siblings of `x`, `x` will be listed as well because `x` satisfies these conditions. So when we print the output, we will have to remove `x` from the list. We will discuss this in the main function:

```
# Check for sibling relationship between 'a' and 'b'
def sibling(x, y):
    temp = var()
    return conde((parent(temp, x), parent(temp, y)))
```

Define a function to check if `x` is `y`'s uncle. We will use the logic that if `x` is `y`'s uncle, then `x` grandparents will be the same as `y`'s parents. Notice that there is a slight modification needed here because when we list out all the uncles of `x`, `x`'s father will be listed as well because `x`'s father satisfies these conditions. So when we print the output, we will have to remove `x`'s father from the list. We will discuss this in the main function:

```
# Check if x is y's uncle
def uncle(x, y):
    temp = var()
    return conde((father(temp, x), grandparent(temp, y)))
```

Define the main function and initialize the relations `father` and `mother`:

```
if __name__=='__main__':
    father = Relation()
    mother = Relation()
```

Load the data from the `relationships.json` file:

```
    with open('relationships.json') as f:
        d = json.loads(f.read())
```

Read the data and add them to our fact base:

```
    for item in d['father']:
        facts(father, (list(item.keys())[0], list(item.values())[0]))

    for item in d['mother']:
        facts(mother, (list(item.keys())[0], list(item.values())[0]))
```

Define the variable `x`:

```
x = var()
```

We are now ready to ask some questions and see if our solver can come up with the right answers. Let's ask who John's children are:

```
# John's children
name = 'John'
output = run(0, x, father(name, x))
print("\nList of " + name + "'s children:")
for item in output:
    print(item)
```

Who is William's mother?

```
# William's mother
name = 'William'
output = run(0, x, mother(x, name))[0]
print("\n" + name + "'s mother:\n" + output)
```

Who are Adam's parents?

```
# Adam's parents
name = 'Adam'
output = run(0, x, parent(x, name))
print("\nList of " + name + "'s parents:")
for item in output:
    print(item)
```

Who are Wayne's grandparents?

```
# Wayne's grandparents
name = 'Wayne'
output = run(0, x, grandparent(x, name))
print("\nList of " + name + "'s grandparents:")
for item in output:
    print(item)
```

Who are Megan's grandchildren?

```
# Megan's grandchildren
name = 'Megan'
output = run(0, x, grandparent(name, x))
print("\nList of " + name + "'s grandchildren:")
for item in output:
    print(item)
```

Who are David's siblings?

```
# David's siblings
name = 'David'
output = run(0, x, sibling(x, name))
siblings = [x for x in output if x != name]
print("\nList of " + name + "'s siblings:")
for item in siblings:
    print(item)
```

Who are Tiffany's uncles?

```
# Tiffany's uncles
name = 'Tiffany'
name_father = run(0, x, father(x, name))[0]
output = run(0, x, uncle(x, name))
output = [x for x in output if x != name_father]
print("\nList of " + name + "'s uncles:")
for item in output:
    print(item)
```

List out all the spouses in the family:

```
# All spouses
a, b, c = var(), var(), var()
output = run(0, (a, b), (father, a, c), (mother, b, c))
print("\nList of all spouses:")
for item in output:
    print('Husband:', item[0], '<==> Wife:', item[1])
```

The full code is given in `family.py`. If you run the code, you will see many things on your Terminal. The first half looks like the following:

```
List of John's children:
David
William
Adam

William's mother:
Megan

List of Adam's parents:
John
Megan

List of Wayne's grandparents:
John
Megan
```

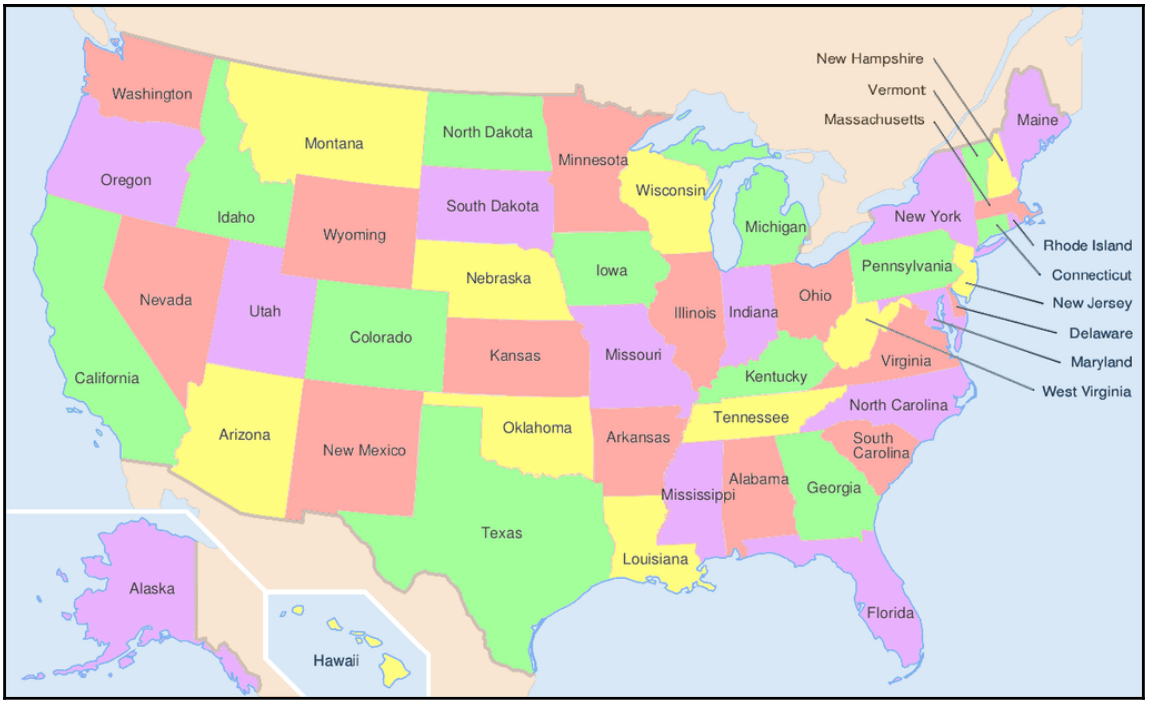
The second half looks like the following:

```
List of Megan's grandchildren:
Chris
Sophia
Peter
Stephanie
Julie
Tiffany
Neil
Wayne

List of David's siblings:
William
Adam

List of Tiffany's uncles:
William
Adam

List of all spouses:
Husband: Adam <==> Wife: Lily
Husband: David <==> Wife: Olivia
Husband: John <==> Wife: Megan
Husband: William <==> Wife: Emma
```

You can compare the outputs with the family tree to ensure that the answers are indeed correct.

Analyzing geography

Let's use logic programming to build a solver to analyze geography. In this problem, we will specify information about the location of various states in the US and then query our program to answer various questions based on those facts and rules. The following is a map of the US:



You have been provided with two text files named `adjacent_states.txt` and `coastal_states.txt`. These files contain the details about which states are adjacent to each other and which states are coastal. Based on this, we can get interesting information like What states are adjacent to both Oklahoma and Texas? or Which coastal state is adjacent to both New Mexico and Louisiana?

Create a new Python file and import the following:

```
from logpy import run, fact, eq, Relation, var
```

Initialize the relations:

```
adjacent = Relation()
coastal = Relation()
```

Define the input files to load the data from:

```
file_coastal = 'coastal_states.txt'
file_adjacent = 'adjacent_states.txt'
```

Load the data:

```
# Read the file containing the coastal states
with open(file_coastal, 'r') as f:
    line = f.read()
    coastal_states = line.split(',')
```

Add the information to the fact base:

```
# Add the info to the fact base
for state in coastal_states:
    fact(coastal, state)
```

Read the adjacency data:

```
# Read the file containing the coastal states
with open(file_adjacent, 'r') as f:
    adjlist = [line.strip().split(',') for line in f if line and
line[0].isalpha()]
```

Add the adjacency information to the fact base:

```
# Add the info to the fact base
for L in adjlist:
    head, tail = L[0], L[1:]
    for state in tail:
        fact(adjacent, head, state)
```

Initialize the variables `x` and `y`:

```
# Initialize the variables
x = var()
y = var()
```

We are now ready to ask some questions. Check if Nevada is adjacent to Louisiana:

```
# Is Nevada adjacent to Louisiana?
output = run(0, x, adjacent('Nevada', 'Louisiana'))
print('\nIs Nevada adjacent to Louisiana?:')
print('Yes' if len(output) else 'No')
```

Print out all the states that are adjacent to Oregon:

```
# States adjacent to Oregon
output = run(0, x, adjacent('Oregon', x))
print('\nList of states adjacent to Oregon:')
for item in output:
    print(item)
```

List all the coastal states that are adjacent to Mississippi:

```
# States adjacent to Mississippi that are coastal
output = run(0, x, adjacent('Mississippi', x), coastal(x))
print('\nList of coastal states adjacent to Mississippi:')
for item in output:
    print(item)
```

List seven states that border a coastal state:

```
# List of 'n' states that border a coastal state
n = 7
output = run(n, x, coastal(y), adjacent(x, y))
print('\nList of ' + str(n) + ' states that border a coastal state:')
for item in output:
    print(item)
```

List states that are adjacent to both Arkansas and Kentucky:

```
# List of states that adjacent to the two given states
output = run(0, x, adjacent('Arkansas', x), adjacent('Kentucky', x))
print('\nList of states that are adjacent to Arkansas and Kentucky:')
for item in output:
    print(item)
```

The full code is given in `states.py`. If you run the code, you will see the following output:

```
Is Nevada adjacent to Louisiana?:
No

List of states adjacent to Oregon:
Washington
California
Nevada
Idaho

List of coastal states adjacent to Mississippi:
Alabama
Louisiana

List of 7 states that border a coastal state:
Georgia
Pennsylvania
Massachusetts
Wisconsin
Maine
Oregon
Ohio

List of states that are adjacent to Arkansas and Kentucky:
Missouri
Tennessee
```

You can cross-check the output with the US map to verify if the answers are right. You can also add more questions to the program to see if it can answer them.

Building a puzzle solver

Another interesting application of logic programming is in solving puzzles. We can specify the conditions of a puzzle and the program will come up with a solution. In this section, we will specify various bits and pieces of information about four people and ask for the missing piece of information.

In the logic program, we specify the puzzle as follows:

- Steve has a blue car
- The person who owns the cat lives in Canada
- Matthew lives in USA
- The person with the black car lives in Australia
- Jack has a cat
- Alfred lives in Australia
- The person who has a dog lives in France
- Who has a rabbit?

The goal is the find the person who has a rabbit. Here are the full details about the four people:

Name	Pet	Car color	Country
Steve	dog	blue	France
Jack	cat	green	Canada
Matthew	rabbit	yellow	USA
Alfred	parrot	black	Australia

Create a new Python file and import the following packages:

```
from logpy import *
from logpy.core import lall
```

Declare the variable `people`:

```
# Declare the variable
people = var()
```

Define all the rules using `lall`. The first rule is that there are four people:

```
# Define the rules
rules = lall(
    # There are 4 people
    (eq, (var(), var(), var(), var()), people),
```

The person named Steve has a blue car:

```
    # Steve's car is blue
    (membero, ('Steve', var(), 'blue', var()), people),
```

The person who has a cat lives in Canada:

```
        # Person who has a cat lives in Canada
        (membero, (var(), 'cat', var(), 'Canada'), people),
```

The person named Matthew lives in USA:

```
        # Matthew lives in USA
        (membero, ('Matthew', var(), var(), 'USA'), people),
```

The person who has a black car lives in Australia:

```
        # The person who has a black car lives in Australia
        (membero, (var(), var(), 'black', 'Australia'), people),
```

The person named Jack has a cat:

```
        # Jack has a cat
        (membero, ('Jack', 'cat', var(), var()), people),
```

The person named Alfred lives in Australia:

```
        # Alfred lives in Australia
        (membero, ('Alfred', var(), var(), 'Australia'), people),
```

The person who has a dog lives in France:

```
        # Person who owns the dog lives in France
        (membero, (var(), 'dog', var(), 'France'), people),
```

One of the people in this group has a rabbit. Who is that person?

```
        # Who has a rabbit?
        (membero, (var(), 'rabbit', var(), var()), people)
    )
```

Run the solver with the preceding constraints:

```
    # Run the solver
    solutions = run(0, people, rules)
```

Extract the output from the solution:

```
    # Extract the output
    output = [house for house in solutions[0] if 'rabbit' in house][0][0]
```

Print the full matrix obtained from the solver:

```
# Print the output
print('\n' + output + ' is the owner of the rabbit')
print('\nHere are all the details:')
attribs = ['Name', 'Pet', 'Color', 'Country']
print('\n' + '\t\t'.join(attribs))
print('=' * 57)
for item in solutions[0]:
    print('')
    print('\t\t'.join([str(x) for x in item]))
```

The full code is given in `puzzle.py`. If you run the code, you will see the following output:

```
Matthew is the owner of the rabbit

Here are all the details:

Name            Pet             Color           Country
=========================================================

Steve           dog             blue            France

Jack            cat             ~_9             Canada

Matthew         rabbit          ~_11            USA

Alfred          ~_13            black           Australia
```

The preceding figure shows all the values obtained using the solver. Some of them are still unknown as indicated by numbered names. Even though the information was incomplete, our solver was able to answer our question. But in order to answer every single question, you may need to add more rules. This program was to demonstrate how to solve a puzzle with incomplete information. You can play around with it and see how you can build puzzle solvers for various scenarios.

Summary

In this chapter, we learned how to write Python programs using logic programming. We discussed how various programming paradigms deal with building programs. We understood how programs are built in logic programming. We learned about various building blocks of logic programming and discussed how to solve problems in this domain.

We implemented various Python programs to solve interesting problems and puzzles. In the next chapter, we will learn about heuristic search techniques and use those algorithms to solve real world problems.

7
Heuristic Search Techniques

In this chapter, we are going to learn about heuristic search techniques. Heuristic search techniques are used to search through the solution space to come up with answers. The search is conducted using heuristics that guide the search algorithm. This heuristic allows the algorithm to speed up the process, which would otherwise take a really long time to arrive at the solution.

By the end of this chapter, you will know about the following:

- What is heuristic search?
- Uninformed vs. informed search
- Constraint Satisfaction Problems
- Local search techniques
- Simulated annealing
- Constructing a string using greedy search
- Solving a problem with constraints
- Solving the region coloring problem
- Building an 8-puzzle solver
- Building a maze solver

What is heuristic search?

Searching and organizing data is an important topic within Artificial Intelligence. There are many problems that require searching for an answer within the solution domain. There are many possible solutions to a given problem and we do not know which ones are correct. By efficiently organizing the data, we can search for solutions quickly and effectively.

More often, there are so many possible options to solve a given problem that no algorithm can be developed to find a right solution. Also, going through every single solution is not possible because it is prohibitively expensive. In such cases, we rely on a rule of thumb that helps us narrow down the search by eliminating the options that are obviously wrong. This rule of thumb is called a **heuristic**. The method of using heuristics to guide our search is called **heuristic search**.

Heuristic techniques are very handy because they help us speed up the process. Even if the heuristic is not able to completely eliminate some options, it will help us to order those options so that we are more likely to get to the better solutions first.

Uninformed versus Informed search

If you are familiar with computer science, you should have heard about search techniques like **Depth First Search** (**DFS**), **Breadth First Search** (**BFS**), and **Uniform Cost Search** (**UCS**). These are search techniques that are commonly used on graphs to get to the solution. These are examples of uninformed search. They do not use any prior information or rules to eliminate some paths. They check all the plausible paths and pick the optimal one.

Heuristic search, on the other hand, is called **Informed search** because it uses prior information or rules to eliminate unnecessary paths. Uninformed search techniques do not take the goal into account. These techniques don't really know where they are trying to go unless they just stumble upon the goal in the process.

In the graph problem, we can use heuristics to guide the search. For example, at each node, we can define a heuristic function that returns a score that represents the estimate of the cost of the path from the current node to the goal. By defining this heuristic function, we are informing the search technique about the right direction to reach the goal. This will allow the algorithm to identify which neighbor will lead to the goal.

We need to note that heuristic search might not always find the most optimal solution. This is because we are not exploring every single possibility and we are relying on a heuristic. But it is guaranteed to find a good solution in a reasonable time, which is what we expect from a practical solution. In real-world scenarios, we need solutions that are fast and effective. Heuristic searches provide an efficient solution by arriving at a reasonable solution quickly. They are used in cases where the problems cannot be solved in any other way or would take a really long time to solve.

Constraint Satisfaction Problems

There are many problems that have to be solved under constraints. These constraints are basically conditions that cannot be violated during the process of solving the problem. These problems are referred to as **Constraint Satisfaction Problems** (**CSPs**).

CSPs are basically mathematical problems that are defined as a set of variables that must satisfy a number of constraints. When we arrive at the final solution, the states of the variables must obey all the constraints. This technique represents the entities involved in a given problem as a collection of a fixed number of constraints over variables. These variables need to be solved by constraint satisfaction methods.

These problems require a combination of heuristics and other search techniques to be solved in a reasonable amount of time. In this case, we will use constraint satisfaction techniques to solve problems on finite domains. A finite domain consists of a finite number of elements. Since we are dealing with finite domains, we can use search techniques to arrive at the solution.

Local search techniques

Local search is a particular way of solving a CSP. It keeps improving the values until all the constraints are satisfied. It iteratively keeps updating the variables until we arrive at the destination. These algorithms modify the value during each step of the process that gets us closer to the goal. In the solution space, the updated value is closer to the goal than the previous value. Hence it is known as a local search.

Local search algorithm is a heuristic search algorithm. These algorithms use a function that calculates the quality of each update. For example, it can count the number of constraints that are being violated by the current update or it can see how the update affects the distance to the goal. This is referred to as the cost of the assignment. The overall goal of local search is to find the minimal cost update at each step.

Hill climbing is a popular local search technique. It uses a heuristic function that measures the difference between the current state and the goal. When we start, it checks if the state is the final goal. If it is, then it stops. If not, then it selects an update and generates a new state. If it's closer to the goal than the current state, then it makes that the current state. If not, it ignores it and continues the process until it checks all possible updates. It basically climbs the hill until it reaches the summit.

Simulated Annealing

Simulated Annealing is a type of local search as well as a stochastic search technique. Stochastic search techniques are used extensively in various fields such as robotics, chemistry, manufacturing, medicine, economics, and so on. We can perform things like optimizing the design of a robot, determining the timing strategies for automated control in factories, and planning traffic. Stochastic algorithms are used to solve many real-world problems.

Simulated Annealing is a variation of the hill climbing technique. One of the main problems of hill climbing is that it ends up climbing false foothills. This means that it gets stuck in local maxima. So it is better to check out the whole space before we make any climbing decisions. In order to achieve this, the whole space is initially explored to see what it is like. This helps us avoid getting stuck in a plateau or local maxima.

In Simulated Annealing, we reformulate the problem and solve it for minimization, as opposed to maximization. So, we are now descending into valleys as opposed to climbing hills. We are pretty much doing the same thing, but in a different way. We use an objective function to guide the search. This objective function serves as our heuristic.

The reason it is called Simulated Annealing is because it is derived from the metallurgical process. We first heat metals up and then let them cool until they reach the optimal energy state.

The rate at which we cool the system is called the **annealing schedule**. The rate of cooling is important because it directly impacts the final result. In the real world case of metals, if the rate of cooling is too fast, it ends up settling for the local maximum. For example, if we take the heated metal and put it in cold water, it ends up quickly settling for the sub-optimal local maximum.

If the rate of cooling is slow and controlled, we give the metal a chance to arrive at the globally optimum state. The chances of taking big steps quickly towards any particular hill are lower in this case. Since the rate of cooling is slow, it will take its time to choose the best state. We do something similar with data in our case.

We first evaluate the current state and see if it is the goal. If it is, then we stop. If not, then we set the best state variable to the current state. We then define our annealing schedule that controls how quickly it descends into a valley. We compute the difference between the current state and the new state. If the new state is not better, then we make it the current state with a certain predefined probability. We do this using a random number generator and making a decision based on a threshold. If it is above the threshold, then we set the best state to this state. Based on this, we update the annealing schedule depending on the number of nodes. We keep doing this until we arrive at the goal.

Constructing a string using greedy search

Greedy search is an algorithmic paradigm that makes the locally optimal choice at each stage in order to find the global optimum. But in many problems, greedy algorithms do not produce globally optimum solutions. An advantage of using greedy algorithms is that they produce an approximate solution in a reasonable time. The hope is that this approximate solution is reasonably close to the global optimal solution.

Greedy algorithms do not refine their solutions based on new information during the search. For example, let's say you are planning on a road trip and you want to take the best route possible. If you use a greedy algorithm to plan the route, it would ask you to take routes that are shorter but might end up taking more time. It can also lead you to paths that may seem faster in the short term, but might lead to traffic jams later. This happens because greedy algorithms only think about the next step and not the globally optimal final solution.

Let's see how to solve a problem using a greedy search. In this problem, we will try to recreate the input string based on the alphabets. We will ask the algorithm to search the solution space and construct a path to the solution.

We will be using a package called `simpleai` throughout this chapter. It contains various routines that are useful in building solutions using heuristic search techniques. It's available at `https://github.com/simpleai-team/simpleai`. We need to make a few changes to the source code in order to make it work in Python3. A file called `simpleai.zip` has been provided along with the code for the book. Unzip this file into a folder called `simpleai`. This folder contains all the necessary changes to the original library necessary to make it work in Python3. Place the `simpleai` folder in the same folder as your code and you'll be able to run your code smoothly.

Create a new Python file and import the following packages:

```
import argparse
import simpleai.search as ss
```

Define a function to parse the input arguments:

```
def build_arg_parser():
    parser = argparse.ArgumentParser(description='Creates the input string
\
            using the greedy algorithm')
    parser.add_argument("--input-string", dest="input_string",
required=True,
            help="Input string")
    parser.add_argument("--initial-state", dest="initial_state",
required=False,
            default='', help="Starting point for the search")
    return parser
```

Create a class that contains the methods needed to solve the problem. This class inherits the `SearchProblem` class available in the library. We just need to override a couple of methods to suit our needs. The first method `set_target` is a custom method that we define to set the target string:

```
class CustomProblem(ss.SearchProblem):
    def set_target(self, target_string):
      self.target_string = target_string
```

The `actions` is a method that comes with a `SearchProblem` and we need to override it. It's responsible for taking the right steps towards the goal. If the length of the current string is less than the length of the target string, it will return the list of possible alphabets to choose from. If not, it will return an empty string:

```
    # Check the current state and take the right action
    def actions(self, cur_state):
        if len(cur_state) < len(self.target_string):
            alphabets = 'abcdefghijklmnopqrstuvwxyz'
            return list(alphabets + ' ' + alphabets.upper())
        else:
            return []
```

a method to compute the result by concatenating the current string and the action that needs to be taken. This method comes with a `SearchProblem` and we are overriding it:

```
    # Concatenate state and action to get the result
    def result(self, cur_state, action):
      return cur_state + action
```

The method `is_goal` is a part of the `SearchProblem` and it's used to check if we have arrived at the goal:

```
# Check if goal has been achieved
def is_goal(self, cur_state):
  return cur_state == self.target_string
```

The method `heuristic` is also a part of the `SearchProblem` and we need to override it. We will define our own heuristic that will be used to solve the problem. We will calculate how far we are from the goal and use that as the heuristic to guide it towards the goal:

```
# Define the heuristic that will be used
    def heuristic(self, cur_state):
        # Compare current string with target string
        dist = sum([1 if cur_state[i] != self.target_string[i] else 0
                    for i in range(len(cur_state))])

        # Difference between the lengths
        diff = len(self.target_string) - len(cur_state)

        return dist + diff
```

e input arguments:

```
if __name__=='__main__':
    args = build_arg_parser().parse_args()
```

Initialize the `CustomProblem` object:

```
# Initialize the object
problem = CustomProblem()
```

Set the starting point as well as the goal we want to achieve:

```
# Set target string and initial state
problem.set_target(args.input_string)
problem.initial_state = args.initial_state
```

Run the solver:

```
# Solve the problem
output = ss.greedy(problem)
```

Print the path to the solution:

```
print('\nTarget string:', args.input_string)
print('\nPath to the solution:')
for item in output.path():
    print(item)
```

The full code is given in the file `greedy_search.py`. If you run the code with an empty initial state:

```
$ python3 greedy_search.py --input-string 'Artificial Intelligence' --
initial-state ''
```

You will get the following output:

```
Path to the solution:
(None, '')
('A', 'A')
('r', 'Ar')
('t', 'Art')
('i', 'Arti')
('f', 'Artif')
('i', 'Artifi')
('c', 'Artific')
('i', 'Artifici')
('a', 'Artificia')
('l', 'Artificial')
(' ', 'Artificial ')
('I', 'Artificial I')
('n', 'Artificial In')
('t', 'Artificial Int')
('e', 'Artificial Inte')
('l', 'Artificial Intel')
('l', 'Artificial Intell')
('i', 'Artificial Intelli')
('g', 'Artificial Intellig')
('e', 'Artificial Intellige')
('n', 'Artificial Intelligen')
('c', 'Artificial Intelligenc')
('e', 'Artificial Intelligence')
```

If you run the code with a non-empty starting point:

```
$ python3 greedy_search.py --input-string 'Artificial Intelligence with
Python' --initial-state 'Artificial Inte'
```

You will get the following output:

```
Path to the solution:
(None, 'Artificial Inte')
('l', 'Artificial Intel')
('l', 'Artificial Intell')
('i', 'Artificial Intelli')
('g', 'Artificial Intellig')
('e', 'Artificial Intellige')
('n', 'Artificial Intelligen')
('c', 'Artificial Intelligenc')
('e', 'Artificial Intelligence')
(' ', 'Artificial Intelligence ')
('w', 'Artificial Intelligence w')
('i', 'Artificial Intelligence wi')
('t', 'Artificial Intelligence wit')
('h', 'Artificial Intelligence with')
(' ', 'Artificial Intelligence with ')
('P', 'Artificial Intelligence with P')
('y', 'Artificial Intelligence with Py')
('t', 'Artificial Intelligence with Pyt')
('h', 'Artificial Intelligence with Pyth')
('o', 'Artificial Intelligence with Pytho')
('n', 'Artificial Intelligence with Python')
```

Solving a problem with constraints

We have already discussed how Constraint Satisfaction Problems are formulated. Let's apply them to a real-world problem. In this problem, we have a list of names and each name can only take a fixed set of values. We also have a set of constraints between these people that needs to be satisfied. Let's see how to do it.

Create a new Python file and import the following packages:

```
from simpleai.search import CspProblem, backtrack, \
    min_conflicts, MOST_CONSTRAINED_VARIABLE, \
    HIGHEST_DEGREE_VARIABLE, LEAST_CONSTRAINING_VALUE
```

Define the constraint that specifies that all the variables in the input list should have unique values:

```
# Constraint that expects all the different variables
# to have different values
def constraint_unique(variables, values):
    # Check if all the values are unique
    return len(values) == len(set(values))
```

Define the constraint that specifies that the first variable should be bigger than the second variable:

```
# Constraint that specifies that one variable
# should be bigger than other
def constraint_bigger(variables, values):
    return values[0] > values[1]
```

Define the constraint that specifies that if the first variable is odd, then the second variable should be even and vice versa:

```
# Constraint that specifies that there should be
# one odd and one even variables in the two variables
def constraint_odd_even(variables, values):
    # If first variable is even, then second should
    # be odd and vice versa
    if values[0] % 2 == 0:
        return values[1] % 2 == 1
    else:
        return values[1] % 2 == 0
```

Define the `main` function and define the variables:

```
if __name__=='__main__':
    variables = ('John', 'Anna', 'Tom', 'Patricia')
```

Define the list of values that each variable can take:

```
    domains = {
        'John': [1, 2, 3],
        'Anna': [1, 3],
        'Tom': [2, 4],
        'Patricia': [2, 3, 4],
    }
```

Define the constraints for various scenarios. In this case, we specify three constraints as follows:

- John, Anna, and Tom should have different values
- Tom's value should be bigger than Anna's value
- If John's value is odd, then Patricia's value should be even and vice versa

Use the following code:

```
constraints = [
    (('John', 'Anna', 'Tom'), constraint_unique),
    (('Tom', 'Anna'), constraint_bigger),
    (('John', 'Patricia'), constraint_odd_even),
]
```

Use the preceding variables and the constraints to initialize the `CspProblem` object:

```
problem = CspProblem(variables, domains, constraints)
```

Compute the solution and print it:

```
print('\nSolutions:\n\nNormal:', backtrack(problem))
```

Compute the solution using the `MOST_CONSTRAINED_VARIABLE` heuristic:

```
print('\nMost constrained variable:', backtrack(problem,
        variable_heuristic=MOST_CONSTRAINED_VARIABLE))
```

Compute the solution using the `HIGHEST_DEGREE_VARIABLE` heuristic:

```
print('\nHighest degree variable:', backtrack(problem,
        variable_heuristic=HIGHEST_DEGREE_VARIABLE))
```

Compute the solution using the `LEAST_CONSTRAINING_VALUE` heuristic:

```
print('\nLeast constraining value:', backtrack(problem,
        value_heuristic=LEAST_CONSTRAINING_VALUE))
```

Compute the solution using the `MOST_CONSTRAINED_VARIABLE` variable heuristic and `LEAST_CONSTRAINING_VALUE` value heuristic:

```
print('\nMost constrained variable and least constraining value:',
        backtrack(problem,
variable_heuristic=MOST_CONSTRAINED_VARIABLE,
        value_heuristic=LEAST_CONSTRAINING_VALUE))
```

Compute the solution using the `HIGHEST_DEGREE_VARIABLE` variable heuristic and `LEAST_CONSTRAINING_VALUE` value heuristic:

```
print('\nHighest degree and least constraining value:',
        backtrack(problem, variable_heuristic=HIGHEST_DEGREE_VARIABLE,
        value_heuristic=LEAST_CONSTRAINING_VALUE))
```

Compute the solution using the minimum conflicts heuristic:

```
print('\nMinimum conflicts:', min_conflicts(problem))
```

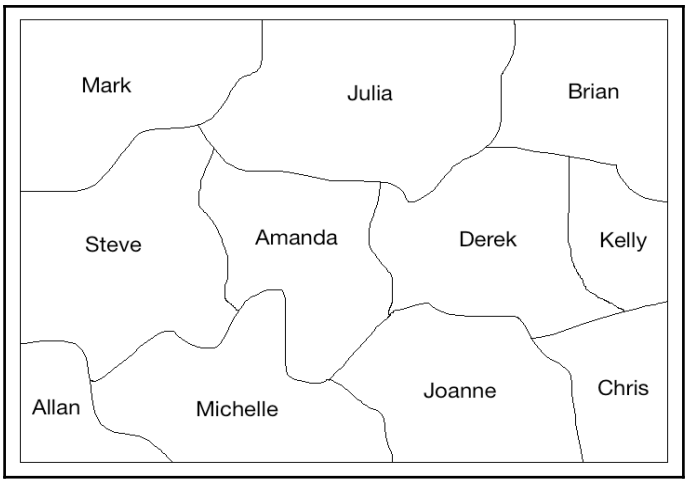
The full code is given in the file `constrained_problem.py`. If you run the code, you will get the following output:

```
Solutions:

Normal: {'Patricia': 2, 'John': 1, 'Anna': 3, 'Tom': 4}

Most constrained variable: {'Patricia': 2, 'John': 3, 'Anna': 1, 'Tom': 2}

Highest degree variable: {'Patricia': 2, 'John': 1, 'Anna': 3, 'Tom': 4}

Least constraining value: {'Patricia': 2, 'John': 1, 'Anna': 3, 'Tom': 4}

Most constrained variable and least constraining value: {'Patricia': 2, 'John': 3, 'Anna': 1, 'Tom': 2}

Highest degree and least constraining value: {'Patricia': 2, 'John': 1, 'Anna': 3, 'Tom': 4}

Minimum conflicts: {'Patricia': 4, 'John': 1, 'Anna': 3, 'Tom': 4}
```

You can check the constraints to see if the solutions satisfy all those constraints.

Solving the region-coloring problem

Let's use the Constraint Satisfaction framework to solve the region-coloring problem. Consider the following screenshot:



We have a few regions in the preceding figure that are labeled with names. Our goal is to color with four colors so that no adjacent regions have the same color.

Create a new Python file and import the following packages:

```
from simpleai.search import CspProblem, backtrack
```

Define the constraint that specifies that the values should be different:

```
# Define the function that imposes the constraint
# that neighbors should be different
def constraint_func(names, values):
    return values[0] != values[1]
```

Define the `main` function and specify the list of names:

```
if __name__=='__main__':
    # Specify the variables
    names = ('Mark', 'Julia', 'Steve', 'Amanda', 'Brian',
            'Joanne', 'Derek', 'Allan', 'Michelle', 'Kelly')
```

Define the list of possible colors:

```
    # Define the possible colors
    colors = dict((name, ['red', 'green', 'blue', 'gray']) for name in
names)
```

We need to convert the map information into something that the algorithm can understand. Let's define the constraints by specifying the list of people who are adjacent to each other:

```
# Define the constraints
constraints = [
  (('Mark', 'Julia'), constraint_func),
  (('Mark', 'Steve'), constraint_func),
  (('Julia', 'Steve'), constraint_func),
  (('Julia', 'Amanda'), constraint_func),
  (('Julia', 'Derek'), constraint_func),
  (('Julia', 'Brian'), constraint_func),
  (('Steve', 'Amanda'), constraint_func),
  (('Steve', 'Allan'), constraint_func),
  (('Steve', 'Michelle'), constraint_func),
  (('Amanda', 'Michelle'), constraint_func),
  (('Amanda', 'Joanne'), constraint_func),
  (('Amanda', 'Derek'), constraint_func),
  (('Brian', 'Derek'), constraint_func),
  (('Brian', 'Kelly'), constraint_func),
  (('Joanne', 'Michelle'), constraint_func),
  (('Joanne', 'Amanda'), constraint_func),
  (('Joanne', 'Derek'), constraint_func),
  (('Joanne', 'Kelly'), constraint_func),
  (('Derek', 'Kelly'), constraint_func),
]
```

Use the variables and constraints to initialize the object:

```
# Solve the problem
problem = CspProblem(names, colors, constraints)
```

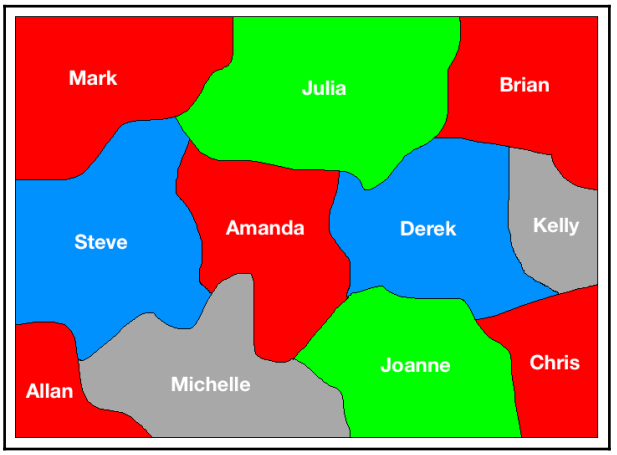
Solve the problem and print the solution:

```
# Print the solution
output = backtrack(problem)
print('\nColor mapping:\n')
for k, v in output.items():
    print(k, '==>', v)
```

The full code is given in the file `coloring.py`. If you run the code, you will get the following output on your Terminal:

```
Color mapping:

Derek ==> blue
Michelle ==> gray
Allan ==> red
Steve ==> blue
Julia ==> green
Amanda ==> red
Joanne ==> green
Mark ==> red
Kelly ==> gray
Brian ==> red
```

If you color the regions based on this output, you will get the following:



You can check that no two adjacent regions have the same color.

Building an 8-puzzle solver

8-puzzle is a variant of the 15-puzzle. You can check it out at `https://en.wikipedia.org/wiki/15_puzzle`. You will be presented with a randomized grid and your goal is to get it back to the original ordered configuration. You can play the game to get familiar with it at `http://mypuzzle.org/sliding`.

We will use an **A* algorithm** to solve this problem. It is an algorithm that's used to find paths to the solution in a graph. This algorithm is a combination of **Dijkstra's algorithm** and a greedy best-first search. Instead of blindly guessing where to go next, the A* algorithm picks the one that looks the most promising. At each node, we generate the list of all possibilities and then pick the one with the minimal cost required to reach the goal.

Let's see how to define the cost function. At each node, we need to compute the cost. This cost is basically the sum of two costs – the first cost is the cost of getting to the current node and the second cost is the cost of reaching the goal from the current node.

We use this summation as our heuristic. As we can see, the second cost is basically an estimate that's not perfect. If this is perfect, then the A* algorithm arrives at the solution quickly. But it's not usually the case. It takes some time to find the best path to the solution. But A* is very effective in finding the optimal paths and is one of the most popular techniques out there.

Let's use the A* algorithm to build an 8-puzzle solver. This is a variant of the solution given in the `simpleai` library. Create a new Python file and import the following packages:

```
from simpleai.search import astar, SearchProblem
```

Define a class that contains the methods to solve the 8-puzzle:

```
# Class containing methods to solve the puzzle
class PuzzleSolver(SearchProblem):
```

Override the `actions` method to align it with our problem:

```
    # Action method to get the list of the possible
    # numbers that can be moved in to the empty space
    def actions(self, cur_state):
      rows = string_to_list(cur_state)
      row_empty, col_empty = get_location(rows, 'e')
```

Check the location of the empty space and create the new action:

```
      actions = []
      if row_empty > 0:
          actions.append(rows[row_empty - 1][col_empty])
      if row_empty < 2:
          actions.append(rows[row_empty + 1][col_empty])
      if col_empty > 0:
          actions.append(rows[row_empty][col_empty - 1])
      if col_empty < 2:
          actions.append(rows[row_empty][col_empty + 1])

      return actions
```

Override the `result` method. Convert the string to a list and extract the location of the empty space. Generate the result by updating the locations:

```
    # Return the resulting state after moving a piece to the empty space
    def result(self, state, action):
        rows = string_to_list(state)
        row_empty, col_empty = get_location(rows, 'e')
        row_new, col_new = get_location(rows, action)

        rows[row_empty][col_empty], rows[row_new][col_new] = \
                rows[row_new][col_new], rows[row_empty][col_empty]

        return list_to_string(rows)
```

Check if the goal has been reached:

```
    # Returns true if a state is the goal state
    def is_goal(self, state):
      return state == GOAL
```

Define the `heuristic` method. We will use the heuristic that computes the distance between the current state and goal state using Manhattan distance:

```
    # Returns an estimate of the distance from a state to
    # the goal using the manhattan distance
    def heuristic(self, state):
      rows = string_to_list(state)

      distance = 0
```

Compute the distance:

```
        for number in '12345678e':
            row_new, col_new = get_location(rows, number)
            row_new_goal, col_new_goal = goal_positions[number]

            distance += abs(row_new - row_new_goal) + abs(col_new -
col_new_goal)

        return distance
```

Define a function to convert a list to string:

```
# Convert list to string
def list_to_string(input_list):
    return '\n'.join(['-'.join(x) for x in        input_list])
```

Define a function to convert a string to a list:

```
# Convert string to list
def string_to_list(input_string):
    return [x.split('-') for x in
      input_string.split('\n')]
```

Define a function to get the location of a given element in the grid:

```
# Find the 2D location of the input element
def get_location(rows, input_element):
    for i, row in enumerate(rows):
      for j, item in enumerate(row):
        if item == input_element:
          return i, j
```

Define the initial state and the final goal we want to achieve:

```
# Final result that we want to achieve
GOAL = '''1-2-3
4-5-6
7-8-e'''

# Starting point
INITIAL = '''1-e-2
6-3-4
7-5-8'''
```

Track the goal positions for each piece by creating a variable:

```
# Create a cache for the goal position of each piece
goal_positions = {}
rows_goal = string_to_list(GOAL)
for number in '12345678e':
    goal_positions[number] = get_location(rows_goal, number)
```

Create the A* solver object using the initial state we defined earlier and extract the result:

```
# Create the solver object
result = astar(PuzzleSolver(INITIAL))
```

Print the solution:

```
# Print the results
for i, (action, state) in enumerate(result.path()):
    print()
    if action == None:
        print('Initial configuration')
    elif i == len(result.path()) - 1:
        print('After moving', action, 'into the empty space. Goal
achieved!')
    else:
        print('After moving', action, 'into the empty space')

    print(state)
```

The full code is given in the file `puzzle.py`. If you run the code, you will get a long output on your Terminal. It will start as follows:

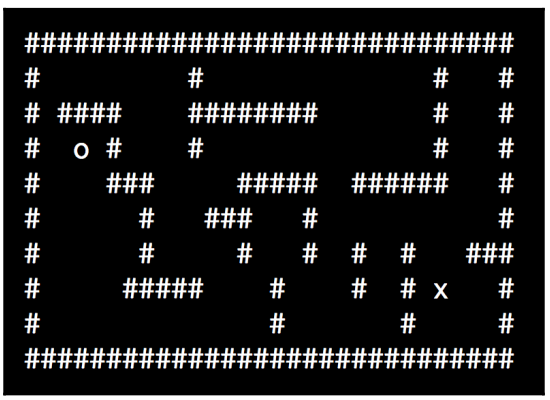
```
Initial configuration
1-e-2
6-3-4
7-5-8

After moving 2 into the empty space
1-2-e
6-3-4
7-5-8

After moving 4 into the empty space
1-2-4
6-3-e
7-5-8

After moving 3 into the empty space
1-2-4
6-e-3
7-5-8

After moving 6 into the empty space
1-2-4
e-6-3
7-5-8
```

If you scroll down, you will see the steps taken to arrive at the solution. At the end, you will see the following on your Terminal:

```
After moving 2 into the empty space
e-2-3
1-4-6
7-5-8

After moving 1 into the empty space
1-2-3
e-4-6
7-5-8

After moving 4 into the empty space
1-2-3
4-e-6
7-5-8

After moving 5 into the empty space
1-2-3
4-5-6
7-e-8

After moving 8 into the empty space. Goal achieved!
1-2-3
4-5-6
7-8-e
```

Building a maze solver

Let's use the A* algorithm to solve a maze. Consider the following figure:



The `#` symbols indicate obstacles. The symbol `o` represents the starting point and `x` represents the goal. Our goal is to find the shortest path from the start to the end point. Let's see how to do it in Python. The following solution is a variant of the solution provided in the `simpleai` library. Create a new Python file and import the following packages:

```
import math
from simpleai.search import SearchProblem, astar
```

Create a class that contains the methods needed to solve the problem:

```
# Class containing the methods to solve the maze
class MazeSolver(SearchProblem):
```

Define the initializer method:

```
    # Initialize the class
    def __init__(self, board):
      self.board = board
      self.goal = (0, 0)
```

Extract the initial and final positions:

```
        for y in range(len(self.board)):
            for x in range(len(self.board[y])):
                if self.board[y][x].lower() == "o":
                    self.initial = (x, y)
                elif self.board[y][x].lower() == "x":
                    self.goal = (x, y)

        super(MazeSolver, self).__init__(initial_state=self.initial)
```

Override the `actions` method. At each position, we need to check the cost of going to the neighboring cells and then append all the possible actions. If the neighboring cell is blocked, then that action is not considered:

```
    # Define the method that takes actions
    # to arrive at the solution
    def actions(self, state):
        actions = []
        for action in COSTS.keys():
            newx, newy = self.result(state, action)
            if self.board[newy][newx] != "#":
                actions.append(action)

        return actions
```

Override the `result` method. Depending on the current state and the input action, update the `x` and y coordinates:

```
# Update the state based on the action
def result(self, state, action):
    x, y = state

    if action.count("up"):
        y -= 1
    if action.count("down"):
        y += 1
    if action.count("left"):
        x -= 1
    if action.count("right"):
        x += 1

    new_state = (x, y)

    return new_state
```

Check if we have arrived at the goal:

```
# Check if we have reached the goal
def is_goal(self, state):
  return state == self.goal
```

We need to define the `cost` function. This is the cost of moving to a neighboring cell, and it's different for vertical/horizontal and diagonal moves. We will define these later:

```
# Compute the cost of taking an action
def cost(self, state, action, state2):
  return COSTS[action]
```

Define the heuristic that will be used. In this case, we will use the Euclidean distance:

```
# Heuristic that we use to arrive at the solution
def heuristic(self, state):
    x, y = state
    gx, gy = self.goal

    return math.sqrt((x - gx) ** 2 + (y - gy) ** 2)
```

Define the `main` function and also define the map we discussed earlier:

```
if __name__ == "__main__":
    # Define the map
    MAP = """
    ##############################
    #          #               #   #
    # ####     ########        #   #
    #  o #     #               #   #
    #    ###      #####  ######   #
    #      #   ###   #            #
    #      #     #   #  #  #   ###
    #     #####    #    #  # x   #
    #              #       #     #
    ##############################
    """
```

Convert the map information into a list:

```
    # Convert map to a list
    print(MAP)
    MAP = [list(x) for x in MAP.split("\n") if x]
```

Define the cost of moving around the map. The diagonal move is more expensive than horizontal or vertical moves:

```
    # Define cost of moving around the map
    cost_regular = 1.0
    cost_diagonal = 1.7
```

Assign the costs to the corresponding moves:

```
    # Create the cost dictionary
    COSTS = {
        "up": cost_regular,
        "down": cost_regular,
        "left": cost_regular,
        "right": cost_regular,
        "up left": cost_diagonal,
        "up right": cost_diagonal,
        "down left": cost_diagonal,
        "down right": cost_diagonal,
    }
```

Create a solver object using the custom class we defined earlier:

```
# Create maze solver object
problem = MazeSolver(MAP)
```

Run the solver on the map and extract the result:

```
# Run the solver
result = astar(problem, graph_search=True)
```

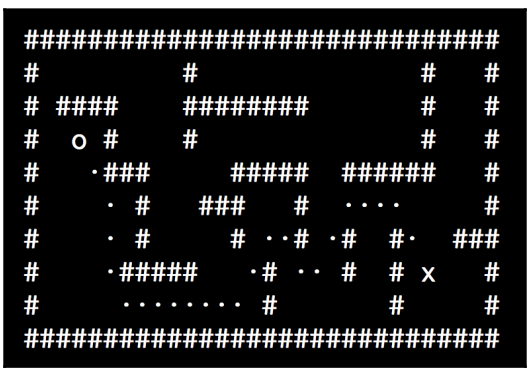
Extract the path from the result:

```
# Extract the path
path = [x[1] for x in result.path()]
```

Print the output:

```
# Print the result
print()
for y in range(len(MAP)):
    for x in range(len(MAP[y])):
        if (x, y) == problem.initial:
            print('o', end='')
        elif (x, y) == problem.goal:
            print('x', end='')
        elif (x, y) in path:
            print(' ', end='')
        else:
            print(MAP[y][x], end='')

    print()
```

The full code is given in the file `maze.py`. If you run the code, you will get the following output:



Summary

In this chapter, we learned how heuristic search techniques work. We discussed the difference between uninformed and informed search. We learned about constraint satisfaction problems and how we can solve problems using this paradigm. We discussed how local search techniques work and why simulated annealing is used in practice. We implemented greedy search for a string problem. We solved a problem using the CSP formulation.

We used this approach to solve the region-coloring problem. We then discussed the A* algorithm and how it can used to find the optimal paths to the solution. We used it to build an 8-puzzle solver as well as a maze solver. In the next chapter, we will discuss genetic algorithms and how they can used to solve real-world problems.

8
Genetic Algorithms

In this chapter, we are going to learn about genetic algorithms. We will discuss the concepts of evolutionary algorithms and genetic programming, and see how they are related to genetic algorithms. We will learn about the fundamental building blocks of genetic algorithms including crossover, mutation, and fitness functions. We will then use these concepts to build various systems.

By the end of this chapter, you will know about the following:

- Understanding evolutionary and genetic algorithms
- Fundamental concepts in genetic algorithms
- Generating a bit pattern with predefined parameters
- Visualizing the progress of the evolution
- Solving the symbol regression problem
- Building an intelligent robot controller

Understanding evolutionary and genetic algorithms

A genetic algorithm is a type of evolutionary algorithm. So, in order to understand genetic algorithms, we need to discuss evolutionary algorithms. An evolutionary algorithm is a meta heuristic optimization algorithm that applies the principles of evolution to solve problems. The concept of evolution is similar to the one we find in nature. We directly use the problem's functions and variables to arrive at the solution. But in a genetic algorithm, any given problem is encoded in bit patterns that are manipulated by the algorithm.

The underlying idea in all evolutionary algorithms is that we take a population of individuals and apply the natural selection process. We start with a set of randomly selected individuals and then identify the strongest among them. The strength of each individual is determined using a **fitness function** that's predefined. In a way, we use the **survival of the fittest** approach.

We then take these selected individuals and create the next generation of individuals by recombination and mutation. We will discuss the concepts of recombination and mutation in the next section. For now, let's think of these techniques as mechanisms to create the next generation by treating the selected individuals as parents.

Once we execute recombination and mutation, we create a new set of individuals who will compete with the old ones for a place in the next generation. By discarding the weakest individuals and replacing them with offspring, we are increasing the overall fitness level of the population. We continue to iterate until the desired overall fitness is achieved.

A genetic algorithm is an evolutionary algorithm where we use a heuristic to find a string of bits that solves a problem. We continuously iterate on a population to arrive at a solution. We do this by generating new populations containing stronger individuals. We apply probabilistic operators such as **selection**, **crossover**, and **mutation** in order to generate the next generation of individuals. The individuals are basically strings, where every string is the encoded version of a potential solution.

A fitness function is used that evaluates the fitness measure of each string telling us how well suited it is to solve this problem. This fitness function is also referred to as an **evaluation function**. Genetic algorithms apply operators that are inspired from nature, which is why the nomenclature is closely related to the terms found in biology.

Fundamental concepts in genetic algorithms

In order to build a genetic algorithm, we need to understand several key concepts and terminology. These concepts are used extensively throughout the field of genetic algorithms to build solutions to various problems. One of the most important aspects of genetic algorithms is the randomness. In order to iterate, it relies on the random sampling of individuals. This means that the process is non-deterministic. So, if you run the same algorithm multiple times, you might end up with different solutions.

Let's talk about population. A population is a set of individuals that are possible candidate solutions. In a genetic algorithm, we do not maintain a single best solution at any given stage. It maintains a set of potential solutions, one of which is the best. But the other solutions play an important role during the search. Since we have a population of solutions, it is less likely that will get stuck in a local optimum. Getting stuck in the local optimum is a classic problem faced by other optimization techniques.

Now that we know about population and the stochastic nature of genetic algorithms, let's talk about the operators. In order to create the next generation of individuals, we need to make sure that they come from the strongest individuals in the current generation. Mutation is one of the ways to do it. A genetic algorithm makes random changes to one or more individuals of the current generation to yield a new candidate solution. This change is called mutation. Now this change might make that individual better or worse than existing individuals.

The next concept here is recombination, which is also called crossover. This is directly related to the role of reproduction in the evolution process. A genetic algorithm tries to combine individuals from the current generation to create a new solution. It combines some of the features of each parent individual to create this offspring. This process is called crossover. The goal is to replace the weaker individuals in the current generation with offspring generated from stronger individuals in the population.

In order to apply crossover and mutation, we need to have selection criteria. The concept of selection is inspired by the theory of natural selection. During each iteration, the genetic algorithm performs a selection process. The strongest individuals are chosen using this selection process and the weaker individuals are terminated. This is where the survival of the fittest concept comes into play. The selection process is carried out using a fitness function that computes the strength of each individual.

Generating a bit pattern with predefined parameters

Now that we know how a genetic algorithm works, let's see how to use it to solve some problems. We will be using a Python package called `DEAP`. You can find all the details about it at `http://deap.readthedocs.io/en/master`. Let's go ahead and install it by running the following command on your Terminal:

```
$ pip3 install deap
```

Now that the package is installed, let's quickly test it. Go into the Python shell by typing the following on your Terminal:

```
$ python3
```

Once you are inside, type the following:

```
>>> import deap
```

If you do not see an error message, we are good to go.

In this section, we will solve a variant of the **One Max problem**. The One Max problem is about generating a bit string that contains the maximum number of ones. It is a simple problem, but it's very helpful in getting familiar with the library as well as understanding how to implement solutions using genetic algorithms. In our case, we will try to generate a bit string that contains a predefined number of ones. You will see that the underlying structure and part of the code is similar to the example used in the `DEAP` library.

Create a new Python file and import the following:

```
import random

from deap import base, creator, tools
```

Let's say we want to generate a bit pattern of length 75, and we want it to contain `45` ones. We need to define an evaluation function that can be used to target this objective:

```
# Evaluation function
def eval_func(individual):
    target_sum = 45
    return len(individual) - abs(sum(individual) - target_sum),
```

If you look at the formula used in the preceding function, you can see that it reaches its maximum value when the number of ones is equal to `45`. The length of each individual is *75*. When the number of ones is equal to `45`, the return value would be *75.*

We now need to define a function to create the toolbox. Let's define a `creator` object for the fitness function and to keep track of the individuals. The `Fitness` class used here is an abstract class and it needs the `weights` attribute to be defined. We are building a maximizing fitness using positive weights:

```
# Create the toolbox with the right parameters
def create_toolbox(num_bits):
    creator.create("FitnessMax", base.Fitness, weights=(1.0,))
    creator.create("Individual", list, fitness=creator.FitnessMax)
```

The first line creates a single objective maximizing fitness named `FitnessMax`. The second line deals with producing the individual. In a given process, the first individual that is created is a list of floats. In order to produce this individual, we must create an `Individual` class using the `creator`. The fitness attribute will use `FitnessMax` defined earlier.

A `toolbox` is an object that is commonly used in `DEAP` It is used to store various functions along with their arguments. Let's create this object:

```
# Initialize the toolbox
toolbox = base.Toolbox()
```

We will now start registering various functions to this `toolbox`. Let's start with the random number generator that generates a random integer between `0` and `1`. This is basically to generate the bit strings:

```
# Generate attributes
toolbox.register("attr_bool", random.randint, 0, 1)
```

Let's register the `individual` function. The method `initRepeat` takes three arguments – a container class for the individual, a function used to fill the container, and the number of times we want the function to repeat itself:

```
# Initialize structures
toolbox.register("individual", tools.initRepeat, creator.Individual,
    toolbox.attr_bool, num_bits)
```

We need to register the `population` function. We want the population to be a list of individuals:

```
# Define the population to be a list of individuals
toolbox.register("population", tools.initRepeat, list,
toolbox.individual)
```

We now need to register the genetic operators. Register the evaluation function that we defined earlier, which will act as our fitness function. We want the individual, which is a bit pattern, to have `45` ones:

```
# Register the evaluation operator
toolbox.register("evaluate", eval_func)
```

Register the crossover operator named `mate` using the `cxTwoPoint` method:

```
# Register the crossover operator
toolbox.register("mate", tools.cxTwoPoint)
```

Register the mutation operator named mutate using `mutFlipBit`. We need to specify the probability of each attribute to be mutated using `indpb`:

```
    # Register a mutation operator
    toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
```

Register the selection operator using `selTournament`. It specifies which individuals will be selected for breeding:

```
    # Operator for selecting individuals for breeding
    toolbox.register("select", tools.selTournament, tournsize=3)
    return toolbox
```

This is basically the implementation of all the concepts we discussed in the preceding section. A toolbox generator function is very common in `DEAP` and we will use it throughout this chapter. So it's important to spend some time to understand how we generated this toolbox.

Define the `main` function by starting with the length of the bit pattern:

```
if __name__ == "__main__":
    # Define the number of bits
    num_bits = 75
```

Create a toolbox using the function we defined earlier:

```
    # Create a toolbox using the above parameter
    toolbox = create_toolbox(num_bits)
```

We need to seed the random number generator so that we get repeatable results:

```
    # Seed the random number generator
    random.seed(7)
```

Create an initial population of, say, `500` individuals using the method available in the `toolbox` object. Feel free to change this number and experiment with it:

```
    # Create an initial population of 500 individuals
    population = toolbox.population(n=500)
```

Define the probabilities of crossing and mutating. Again, these are parameters that are defined by the user. So you can change these parameters and see how they affect the result:

```
    # Define probabilities of crossing and mutating
    probab_crossing, probab_mutating  = 0.5, 0.2
```

Define the number of generations that we need to iterate until the process is terminated. If you increase the number of generations, you are giving it more freedom to improve the strength of the population:

```
# Define the number of generations
num_generations = 60
```

Evaluate all the individuals in the population using the fitness functions:

```
print('\nStarting the evolution process')
# Evaluate the entire population
fitnesses = list(map(toolbox.evaluate, population))
for ind, fit in zip(population, fitnesses):
    ind.fitness.values = fit
```

Start iterating through the generations:

```
print('\nEvaluated', len(population), 'individuals')
# Iterate through generations
for g in range(num_generations):
    print("\n===== Generation", g)
```

In each generation, select the next generation individuals using the selection operator that we registered to the toolbox earlier:

```
    # Select the next generation individuals
    offspring = toolbox.select(population, len(population))
```

Clone the selected individuals:

```
    # Clone the selected individuals
    offspring = list(map(toolbox.clone, offspring))
```

Apply crossover and mutation on the next generation individuals using the probability values defined earlier. Once it's done, we need to reset the fitness values:

```
    # Apply crossover and mutation on the offspring
    for child1, child2 in zip(offspring[::2], offspring[1::2]):
        # Cross two individuals
        if random.random() < probab_crossing:
            toolbox.mate(child1, child2)

            # "Forget" the fitness values of the children
            del child1.fitness.values
            del child2.fitness.values
```

Apply mutation to the next generation individuals using the corresponding probability value that we defined earlier. Once it's done, reset the fitness value:

```
        # Apply mutation
        for mutant in offspring:
            # Mutate an individual
            if random.random() < probab_mutating:
                toolbox.mutate(mutant)
                del mutant.fitness.values
```

Evaluate the individuals with invalid fitness values:

```
        # Evaluate the individuals with an invalid fitness
        invalid_ind = [ind for ind in offspring if not ind.fitness.valid]
        fitnesses = map(toolbox.evaluate, invalid_ind)
        for ind, fit in zip(invalid_ind, fitnesses):
            ind.fitness.values = fit
        print('Evaluated', len(invalid_ind), 'individuals')
```

Replace the population with the next generation individuals:

```
        # The population is entirely replaced by the offspring
        population[:] = offspring
```

Print the stats for the current generation to see how it's progressing:

```
        # Gather all the fitnesses in one list and print the stats
        fits = [ind.fitness.values[0] for ind in population]

        length = len(population)
        mean = sum(fits) / length
        sum2 = sum(x*x for x in fits)
        std = abs(sum2 / length - mean**2)**0.5

        print('Min =', min(fits), ', Max =', max(fits))
        print('Average =', round(mean, 2), ', Standard deviation =',
                round(std, 2))
    print("\n==== End of evolution")
```

Print the final output:

```
    best_ind = tools.selBest(population, 1)[0]
    print('\nBest individual:\n', best_ind)
    print('\nNumber of ones:', sum(best_ind))
```

The full code is given in the file `bit_counter.py`. If you run the code, you will see iterations printed to your Terminal. At the start, you will see something like the following:

```
Starting the evolution process

Evaluated 500 individuals

===== Generation 0
Evaluated 297 individuals
Min = 58.0 , Max = 75.0
Average = 70.43 , Standard deviation = 2.91

===== Generation 1
Evaluated 303 individuals
Min = 63.0 , Max = 75.0
Average = 72.44 , Standard deviation = 2.16

===== Generation 2
Evaluated 310 individuals
Min = 65.0 , Max = 75.0
Average = 73.31 , Standard deviation = 1.6

===== Generation 3
Evaluated 273 individuals
Min = 67.0 , Max = 75.0
Average = 73.76 , Standard deviation = 1.41
```

At the end, you will see something like the following that indicates the end of the evolution:

```
===== Generation 57
Evaluated 306 individuals
Min = 68.0 , Max = 75.0
Average = 74.02 , Standard deviation = 1.27

===== Generation 58
Evaluated 276 individuals
Min = 69.0 , Max = 75.0
Average = 74.15 , Standard deviation = 1.18

===== Generation 59
Evaluated 288 individuals
Min = 69.0 , Max = 75.0
Average = 74.12 , Standard deviation = 1.24

==== End of evolution

Best individual:
 [1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1
, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0
, 1, 0, 0, 0, 1]

Number of ones: 45
```

As seen in the preceding figure, the evolution process ends after 60 generations (zero-indexed). Once it's done, the best individual is picked and printed on the output. It has 45 ones in the best individual, which is like a confirmation for us because the target sum is 45 in our evaluation function.

Visualizing the evolution

Let's see how we can visualize the evolution process. In `DEAP`, they have used a method called **Covariance Matrix Adaptation Evolution Strategy** (**CMA-ES**) to visualize the evolution. It is an evolutionary algorithm that's used to solve non-linear problems in the continuous domain. CMA-ES technique is robust, well studied, and is considered as state of the art in evolutionary algorithms. Let's see how it works by delving into the code provided in their source code. The following code is a slight variation of the example shown in the `DEAP` library.

Create a new Python file and import the following:

```
import numpy as np
import matplotlib.pyplot as plt
from deap import algorithms, base, benchmarks, \
        cma, creator, tools
```

Define a function to create the toolbox. We will define a `FitnessMin` function using negative weights:

```
# Function to create a toolbox
def create_toolbox(strategy):
    creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
    creator.create("Individual", list, fitness=creator.FitnessMin)
```

Create the toolbox and register the evaluation function, as follows:

```
    toolbox = base.Toolbox()
    toolbox.register("evaluate", benchmarks.rastrigin)

    # Seed the random number generator
    np.random.seed(7)
```

Register the `generate` and `update` methods. This is related to the generate-update paradigm where we generate a population from a strategy and this strategy is updated based on the population:

```
    toolbox.register("generate", strategy.generate, creator.Individual)
    toolbox.register("update", strategy.update)

    return toolbox
```

Define the `main` function. Start by defining the number of individuals and the number of generations:

```
if __name__ == "__main__":
    # Problem size
    num_individuals = 10
    num_generations = 125
```

We need to define a strategy before we start the process:

```
    # Create a strategy using CMA-ES algorithm
    strategy = cma.Strategy(centroid=[5.0]*num_individuals, sigma=5.0,
            lambda_=20*num_individuals)
```

Create the toolbox based on the strategy:

```
    # Create toolbox based on the above strategy
    toolbox = create_toolbox(strategy)
```

Create a `HallOfFame` object. The `HallOfFame` object contains the best individual that ever existed in the population. This object is kept in a sorted format at all times. This way, the first element in this object is the individual that has the best fitness value ever seen during the evolution process:

```
    # Create hall of fame object
    hall_of_fame = tools.HallOfFame(1)
```

Register the stats using the `Statistics` method:

```
    # Register the relevant stats
    stats = tools.Statistics(lambda x: x.fitness.values)
    stats.register("avg", np.mean)
    stats.register("std", np.std)
    stats.register("min", np.min)
    stats.register("max", np.max)
```

Define the `logbook` to keep track of the evolution records. It is basically a chronological list of dictionaries:

```
logbook = tools.Logbook()
logbook.header = "gen", "evals", "std", "min", "avg", "max"
```

Define objects to compile all the data:

```
# Objects that will compile the data
sigma = np.ndarray((num_generations, 1))
axis_ratio = np.ndarray((num_generations, 1))
diagD = np.ndarray((num_generations, num_individuals))
fbest = np.ndarray((num_generations,1))
best = np.ndarray((num_generations, num_individuals))
std = np.ndarray((num_generations, num_individuals))
```

Iterate through the generations:

```
for gen in range(num_generations):
    # Generate a new population
    population = toolbox.generate()
```

Evaluate individuals using the fitness function:

```
    # Evaluate the individuals
    fitnesses = toolbox.map(toolbox.evaluate, population)
    for ind, fit in zip(population, fitnesses):
        ind.fitness.values = fit
```

Update the strategy based on the population:

```
    # Update the strategy with the evaluated individuals
    toolbox.update(population)
```

Update the hall of fame and statistics with the current generation of individuals:

```
    # Update the hall of fame and the statistics with the
    # currently evaluated population
    hall_of_fame.update(population)
    record = stats.compile(population)
    logbook.record(evals=len(population), gen=gen, **record)
    print(logbook.stream)
```

Save the data for plotting:

```
            # Save more data along the evolution for plotting
            sigma[gen] = strategy.sigma
            axis_ratio[gen] = max(strategy.diagD)**2/min(strategy.diagD)**2
            diagD[gen, :num_individuals] = strategy.diagD**2
            fbest[gen] = hall_of_fame[0].fitness.values
            best[gen, :num_individuals] = hall_of_fame[0]
            std[gen, :num_individuals] = np.std(population, axis=0)
```

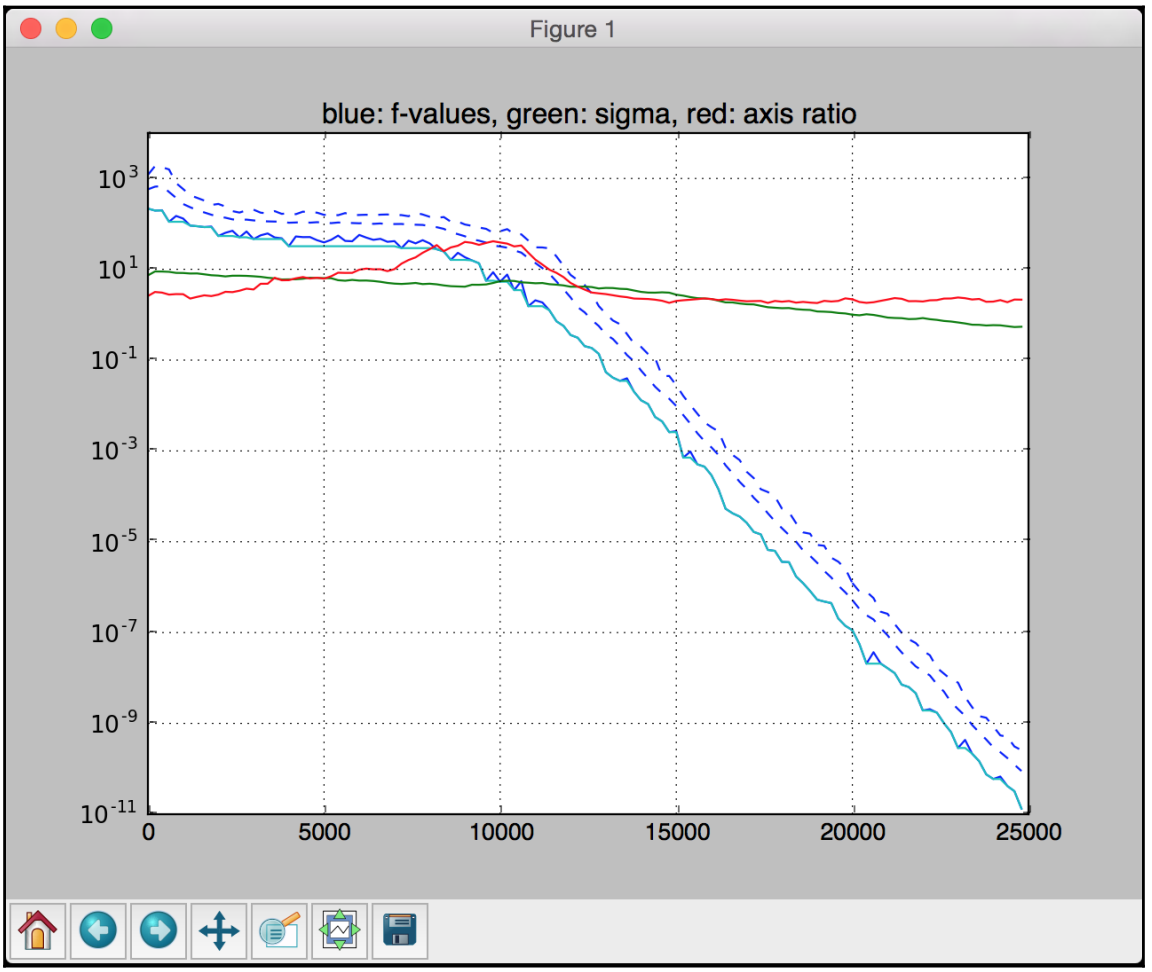
Define the `x` axis and plot the stats:

```
        # The x-axis will be the number of evaluations
        x = list(range(0, strategy.lambda_ * num_generations,
    strategy.lambda_))
        avg, max_, min_ = logbook.select("avg", "max", "min")
        plt.figure()
        plt.semilogy(x, avg, "--b")
        plt.semilogy(x, max_, "--b")
        plt.semilogy(x, min_, "-b")
        plt.semilogy(x, fbest, "-c")
        plt.semilogy(x, sigma, "-g")
        plt.semilogy(x, axis_ratio, "-r")
        plt.grid(True)
        plt.title("blue: f-values, green: sigma, red: axis ratio")
```

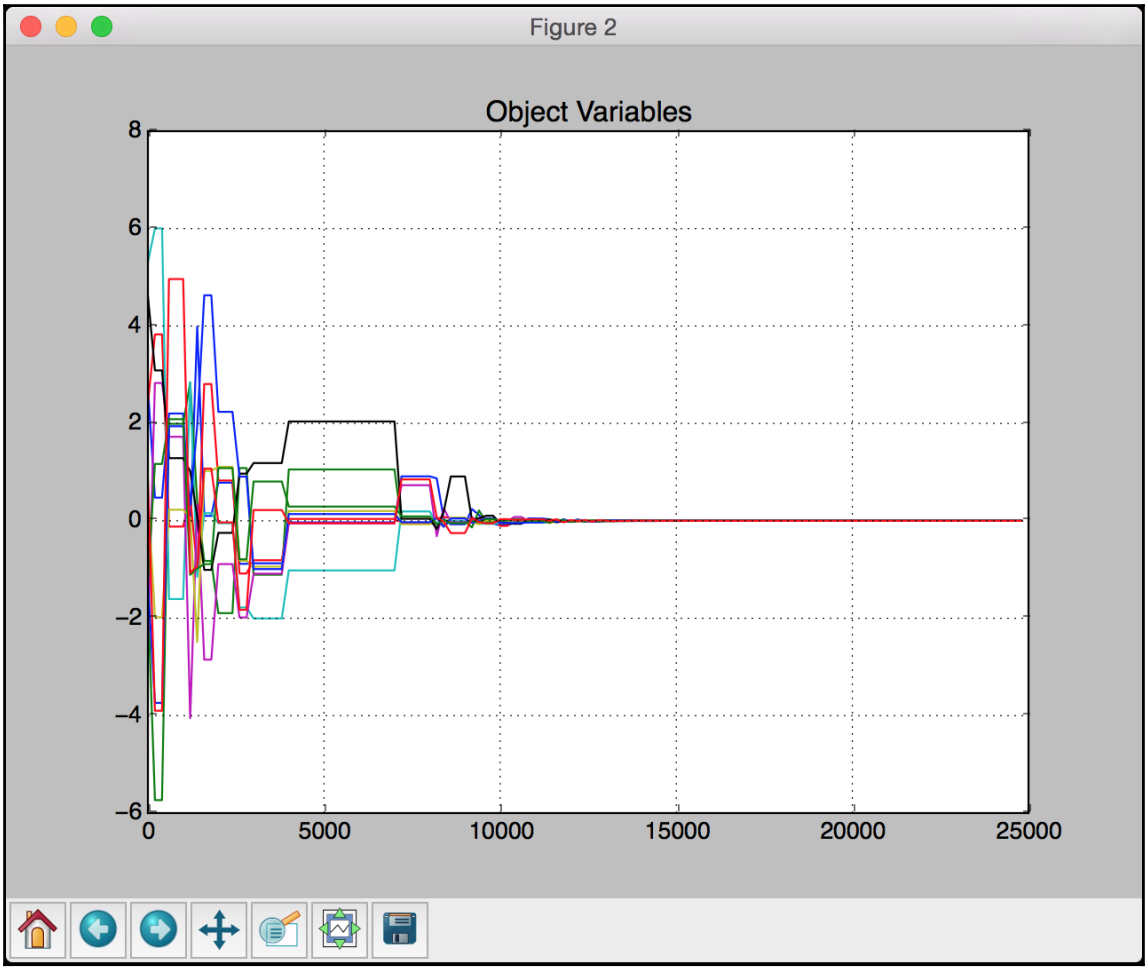
Plot the progress:

```
        plt.figure()
        plt.plot(x, best)
        plt.grid(True)
        plt.title("Object Variables")

        plt.figure()
        plt.semilogy(x, diagD)
        plt.grid(True)
        plt.title("Scaling (All Main Axes)")

        plt.figure()
        plt.semilogy(x, std)
        plt.grid(True)
        plt.title("Standard Deviations in All Coordinates")
        plt.show()
```

The full code is given in the file `visualization.py`. If you run the code, you will see four screenshots. The first screenshot shows various parameters:



The second screenshot shows object variables:



The third screenshot shows scaling: