Designing Machine Learning Systems with Python

Machine learning is one of the fastest growing trends in modern computing. It has applications in a wide range of fields, including economics, the natural sciences, web development, and business modeling. In order to harness the power of these systems, it is essential that the practitioner develops a solid understanding of the underlying design principles.

There are many reasons why machine learning models may not give accurate results. By looking at these systems from a design perspective, we gain a deeper understanding of the underlying algorithms and the optimization methods that are available. This book will give you a solid foundation in the machine learning design process, and enable you to build customized machine learning models to solve unique problems. You may already know about, or have worked with, some of the off-the-shelf machine learning models for solving common problems such as spam detection or movie classification, but to begin solving more complex problems, it is important to adapt these models to your own specific needs. This book will give you this understanding and more.

Who this book is written for
This book is for data scientists, scientists, or just the curious. To get the most out of this book, you will need to know some linear algebra and some Python, and have a basic knowledge of machine learning concepts.

What you will learn from this book
- Gain an understanding of the machine learning design process
- Optimize machine learning systems for improved accuracy
- Understand common programming tools and techniques for machine learning
- Develop techniques and strategies for dealing with large amounts of data from a variety of sources
- Build models to solve unique tasks

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Turning Data into Information'
- A synopsis of the book’s content
- More information on *Designing Machine Learning Systems with Python*
David Julian is currently working on a machine learning project with Urban Ecological Systems Ltd and Blue Smart Farms (http://www.bluesmartfarms.com.au) to detect and predict insect infestation in greenhouse crops. He is currently collecting a labeled training set that includes images and environmental data (temperature, humidity, soil moisture, and pH), linking this data to observations of infestation (the target variable), and using it to train neural net models. The aim is to create a model that will reduce the need for direct observation, be able to anticipate insect outbreaks, and subsequently control conditions. There is a brief outline of the project at http://davejulian.net/projects/ues. David also works as a data analyst, I.T. consultant, and trainer.
Preface

Machine learning is one of the biggest trends that the world of computing has seen. Machine learning systems have a profound and exciting ability to provide important insights on an amazing variety of applications, from ground-breaking and lifesaving medical research to discovering fundamental physical aspects of our universe; from providing us with better, cleaner food to web analytics and economic modeling. In fact, there is hardly any area of our lives that is not touched by this technology in some way. Everyone wants to get into the field of machine learning, and in order to obtain sufficient recognition in this field, one must be able to understand and design a machine learning system that serves the needs of a project.

What this book covers

Chapter 1, Thinking in Machine Learning, gets you started with the basics of machine learning, and as the title says, it will help you think in the machine learning paradigm. You will learn the design principles and various models involved in machine learning.

Chapter 2, Tools and Techniques, explains that Python comes equipped with a large library of packages for machine learning tasks. This chapter will give you a flavor of some huge libraries. It will cover packages such as NumPy, SciPy, Matplotlib, and Scikit-learn.

Chapter 3, Turning Data into Information, explains that raw data can be in many different formats and can be of varying quantity and quality. Sometimes, we are overwhelmed by data, and sometimes we struggle to get every last drop of information from our data. For data to become information, it requires some meaningful structure. In this chapter, we will introduce some broad topics such as big data, data properties, data sources, and data processing and analysis.
Chapter 4, Models – Learning from Information, takes you through the logical models—where we explore a logical language and create a hypothesis space mapping, tree models—where we will find that they can be applied to a wide range of tasks and are both descriptive and easy to interpret; and rule models—where we discuss both ordered rule list- and unordered rule set-based models.

Chapter 5, Linear Models, introduces one of the most widely used models that forms the foundation of many advanced nonlinear techniques, such as support vector machines and neural networks. In this chapter, we will study some of the most commonly used techniques in machine learning. We will create hypothesis representations for linear and logistic regression.

Chapter 6, Neural Networks, introduces the powerful machine learning algorithm of artificial neural networks. We will see how these networks are a simplified model of neurons in the brain.

Chapter 7, Features – How Algorithms See the World, goes through the different types of feature—the Quantitative, Ordinal, and Categorical features. We will also learn the Structured and Transforming features in detail.

Chapter 8, Learning with Ensembles, explains the reason behind the motivation for creating machine learning ensembles, which comes from clear intuitions and is grounded in a rich theoretical history. The types of machine learning ensemble that can be created are as diverse as the models themselves, and the main considerations revolve around three things: how we divide our data, how we select the models, and the methods we use to combine their results.

Chapter 9, Design Strategies and Case Studies, looks at some design strategies to ensure your machine learning applications perform optimally. We will learn model selection and parameter tuning techniques, and apply them to several case studies.
Turning Data into Information

Raw data can be in many different formats and of varying quantity and quality. Sometimes, we are overwhelmed with data, and sometimes we struggle to get every last drop of information from our data. For data to become information, it requires some meaningful structure. We often have to deal with incompatible formats, inconsistencies, errors, and missing data. It is important to be able to access different parts of the dataset or extract subsets of the data based on some relational criteria. We need to spot patterns in our data and get a feel for how the data is distributed. We can use many tools to find this information hidden in data from visualizations, running algorithms, or just looking at the data in a spreadsheet.

In this chapter, we are going to introduce the following broad topics:

- Big data
- Data properties
- Data sources
- Data processing and analysis

But first, let's take a look into the following explanations:
What is data?

Data can be stored on a hard drive, streamed through a network, or captured live through sensors such as video cameras and microphones. If we are sampling from physical phenomena, such as a video or sound recording, the space is continuous and effectively infinite. Once this space is sampled, that is digitalized, a finite subset of this space has been created and at least some minimal structure has been imposed on it. The data is on a hard drive, encoded in bits, given some attributes such as a name, creation date, and so on. Beyond this, if the data is to be made use of in an application, we need to ask, "how is the data organized and what kinds of queries does it efficiently support?"

When faced with an unseen dataset, the first phase is exploration. Data exploration involves examining the components and structure of data. How many samples does it contain, and how many dimensions are in each sample? What are the data types of each dimension? We should also get a feel for the relationships between variables and how they are distributed. We need to check whether the data values are in line with what we expect. Are there any obvious errors or gaps in the data?

Data exploration must be framed within the scope of a particular problem. Obviously, the first thing to find out is if it is likely that the dataset will provide useful answers. Is it worth our while to continue, or do we need to collect more data? Exploratory data analysis is not necessarily carried out with a particular hypothesis in mind, but perhaps with a sense of which hypotheses are likely to provide useful information.

Data is evidence that can either support or disprove a hypothesis. This evidence is only meaningful if it can be compared to a competing hypothesis. In any scientific process, we use a control. To test a hypothesis, we need to compare it to an equivalent system where the set of variables we are interested in remain fixed. We should attempt to show causality with a mechanism and explanation. We need a plausible reason for our observations. We should also consider that the real world is composed of multiple interacting components, and dealing with multivariate data can lead to exponentially increasing complexity.

It is with these things in mind, a sketch of the territory we are seeking to explore, that we approach new datasets. We have an objective, a point we hope to get to, and our data is a map through this unknown terrain.

Big data

The amount of data that's being created and stored on a global level is almost inconceivable, and it just keeps growing. Big data is a term that describes the large volume of data—both structured and unstructured. Let's now delve deeper into big data, beginning with the challenges of big data.
Challenges of big data
Big data is characterized by three challenges. They are as follows:

- The volume of the data
- The velocity of the data
- The variety of the data

Data volume
The volume problem can be approached from three different directions: efficiency, scalability, and parallelism. Efficiency is about minimizing the time it takes for an algorithm to process a unit of information. A component of this is the underlying processing power of the hardware. The other component, and the one that we have more control over, is ensuring that our algorithms are not wasting precious processing cycles with unnecessary tasks.

Scalability is really about brute force and throwing as much hardware at a problem as you can. Taking into account Moore's law, which states that the trend of computer power doubling every two years, will continue until it reaches its limit; it is clear that scalability is not, by itself, going to be able to keep up with the ever-increasing amounts of data. Simply adding more memory and faster processors is not, in many cases, going to be a cost effective solution.

Parallelism is a growing area of machine learning, and it encompasses a number of different approaches, from harnessing the capabilities of multi-core processors, to large-scale distributed computing on many different platforms. Probably, the most common method is to simply run the same algorithm on many machines, each with a different set of parameters. Another method is to decompose a learning algorithm into an adaptive sequence of queries, and have these queries processed in parallel. A common implementation of this technique is known as MapReduce, or its open source version, Hadoop.

Data velocity
The velocity problem is often approached in terms of data producers and data consumers. The rate of data transfer between the two is called the velocity, and it can be measured in interactive response times. This is the time it takes from a query being made to its response being delivered. Response times are constrained by latencies, such as hard disk read and write times, and the time it takes to transmit data across a network.
Data is being produced at ever greater rates, and this is largely driven by the rapid expansion of mobile networks and devices. The increasing instrumentation of daily life is revolutionizing the way products and services are delivered. This increasing flow of data has led to the idea of **streaming processing**. When input data is at a velocity that makes it impossible to store in its entirety, a level of analysis is necessary as the data streams, in essence, deciding what data is useful and should be stored, and what data can be thrown away. An extreme example is the **Large Hadron Collider** at CERN, where the vast majority of data is discarded. A sophisticated algorithm must scan the data as it is being generated, looking at the information needle in the data haystack. Another instance that processing data streams may be important is when an application requires an immediate response. This is becoming increasingly used in applications such as online gaming and stock market trading.

It is not just the velocity of incoming data that we are interested in; in many applications, particularly on the web, the velocity of a systems output is also important. Consider applications such as recommender systems that need to process a large amount of data and present a response in the time it takes for a web page to load.

**Data variety**

Collecting data from different sources invariably means dealing with misaligned data structures and incompatible formats. It also often means dealing with different semantics and having to understand a data system that may have been built on a fairly different set of logical premises. We have to remember that, very often, data is repurposed for an entirely different application from the one it was originally intended for. There is a huge variety of data formats and underlying platforms. Significant time can be spent converting data into one consistent format. Even when this is done, the data itself needs to be aligned such that each record consists of the same number of features and is measured in the same units.

Consider the relatively simple task of harvesting data from web pages. The data is already structured through the use of a mark language, typically HTML or XML, and this can help give us some initial structure. Yet, we just have to peruse the web to see that there is no standard way of presenting and tagging content in an information-relevant way. The aim of XML is to include content-relevant information in markup tags, for instance, by using tags for *author* or *subject*. However, the usage of such tags is far from universal and consistent. Furthermore, the web is a dynamic environment and many web sites go through frequent structural changes. These changes will often break web applications that expect a specific page structure.
The following diagram shows two dimensions of the big data challenge. I have included a few examples where these domains might approximately sit in this space. Astronomy, for example, has very few sources. It has a relatively small number of telescopes and observatories. Yet the volume of data that astronomers deal with is huge. On the other hand, perhaps, let's compare it to something like environmental sciences, where the data comes from a variety of sources, such as remote sensors, field surveys, validated secondary materials, and so on.

Integrating different data sets can take a significant amount of development time; up to 90 percent in some cases. Each project's data requirements will be different, and an important part of the design process is positioning our data sets with regard to these three elements.

**Data models**

A fundamental question for the data scientist is how the data is stored. We can talk about the hardware, and in this respect, we mean nonvolatile memory such as the hard drive of a computer or flash disk. Another way of interpreting the question (a more logical way) is how is the data organized? In a personal computer, the most visible way that data is stored is hierarchically, in nested folders and files. Data can also be stored in a table format or in a spreadsheet. When we are thinking about structure, we are interested in categories and category types, and how they are related. In a table, how many columns do we need, and in a relational data base, how are tables linked? A data model should not try to impose a structure on the data, but rather find a structure that most naturally emerges from the data.
Data models consist of three components:

- **Structure**: A table is organized into columns and rows; tree structures have nodes and edges, and dictionaries have the structure of key value pairs.

- **Constraints**: This defines the type of valid structures. For a table, this would include the fact that all rows have the same number of columns, and each column contains the same data type for every row. For example, a column, items sold, would only contain integer values. For hierarchical structures, a constraint would be a folder that can only have one immediate parent.

- **Operations**: This includes actions such as finding a particular value, given a key, or finding all rows where the items sold are greater than 100. This is sometimes considered separate from the data model because it is often a higher-level software layer. However, all three of these components are tightly coupled, so it makes sense to think of the operations as part of the data model.

To encapsulate raw data with a data model, we create databases. Databases solve some key problems:

- **They allow us to share data**: It gives multiple users access to the same data with varying read and write privileges.

- **They enforce a data model**: This includes not only the constraints imposed by the structure, say parent child relationships in a hierarchy, but also higher-level constraints such as only allowing one user named bob, or being a number between one and eight.

- **They allow us to scale**: Once the data is larger than the allocated size of our volatile memory, mechanisms are needed to both facilitate the transfer of data and also allow the efficient traversal of a large number of rows and columns.

- **Databases allow flexibility**: They essentially try to hide complexity and provide a standard way of interacting with data.

**Data distributions**

A key characteristic of data is its probability distribution. The most familiar distribution is the normal or Gaussian distribution. This distribution is found in many (all?) physical systems, and it underlies any random process. The normal function can be defined in terms of a **probability density function**:

\[
 f(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-(x-\mu)^2 / (2\sigma^2)}
\]
Here, $\delta$ (sigma) is the **standard deviation** and $\mu$ (mu) is the **mean**. This equation simply describes the relative likelihood a random variable, $x$, will take on a given value. We can interpret the standard deviation as the width of a bell curve, and the mean as its center. Sometimes, the term **variance** is used, and this is simply the square of the standard deviation. The standard deviation essentially measures how spread out the values are. As a general rule of thumb, in a normal distribution, 68% of the values are within 1 standard deviation of the mean, 95% of values are within 2 standard deviations of the mean, and 99.7% are within 3 standard deviations of the mean.

We can get a feel for what these terms do by running the following code and calling the `normal()` function with different values for the mean and variance. In this example, we create the plot of a normal distribution, with a mean of 1 and a variance of 0.5:

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

def normal(mean = 0, var = 1):
    sigma = np.sqrt(var)
    x = np.linspace(-3,3,100)
    plt.plot(x,mlab.normpdf(x,mean,sigma))
    plt.show()

normal(1,0.5)
```
Related to the Gaussian distribution is the binomial distribution. We actually obtain a normal distribution by repeating a binomial process, such as tossing a coin. Over time, the probability approaches that half the tosses will result in heads.

\[ P(x) = \frac{n!}{x!(n-x)!} p^x q^{n-x} \]

In this formula, \( n \) is the number of coin tosses, \( p \) is the probability that half the tosses are heads, and \( q \) is the probability \((1-p)\) that half the tosses are tails. In a typical experiment, say to determine the probability of various outcomes of a series of coin tosses, \( n \), we can perform this many times, and obviously the more times we perform the experiment, the better our understanding of the statistical behavior of the system:

```python
from scipy.stats import binom

def binomial(x=10, n=10, p=0.5):
    fig, ax = plt.subplots(1, 1)
    x = range(x)
    rv = binom(n, p)
    plt.vlines(x, 0, (rv.pmf(x)), colors='k', linestyles='-')
    plt.show()

binomial()
```

You will observe the following output:
Another aspect of discrete distributions is understanding the likelihood of a given number of events occurring within a particular space and/or time. If we know that a given event occurs at an average rate, and each event occurs independently, we can describe it as a Poisson distribution. We can best understand this distribution using a probability mass function. This measures the probability of a given event that will occur at a given point in space/time.

The Poisson distribution has two parameters associated with it: \texttt{lambda}, \( \lambda \), a real number greater than 0, and \( k \), an integer that is 0, 1, 2, and so on.

\[
f(k; \lambda) = \Pr(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}
\]

Here, we generate the plot of a Poisson distribution using the \texttt{scipy.stats} module:

```python
from scipy.stats import poisson
def poisson(x=1000):
xr=range(x)
ps=poisson(xr)
plt.plot(ps.pmf(x/2))
pois()
```

The output of the preceding commands is as shown in the following diagram:
We can describe continuous data distributions using probability density functions. This describes the likelihood that a continuous random variable will take on a specified value. For univariate distributions, that is, those where there is only one random variable, the probability of finding a point $X$ on an interval $(a,b)$ is given by the following:

$$\int_a^b f_X(x) \, dx$$

This describes the fraction of a sampled population for which a value, $x$, lies between $a$ and $b$. Density functions really only have meaning when they are integrated, and this will tell us how densely a population is distributed around certain values. Intuitively, we understand this as the area under the graph of its probability function between these two points. The **Cumulative Density Function (CDF)** is defined as the integral of its probability density functions, $f_X$:

$$F_X(x) = \int_{-\infty}^x f_X(u) \, du$$

The CDF describes the proportion of a sampled population having values for a particular variable that is less than $x$. The following code shows a discrete (binomial) cumulative distribution function. The $s1$ and $s2$ shape parameters determine the step size:

```python
import scipy.stats as stats
def cdf(s1=50, s2=0.2):
    x = np.linspace(0, s2 * 100, s1 * 2)
    cd = stats.binom.cdf
    plt.plot(x, cd(x, s1, s2))
    plt.show()
```
Data from databases

We generally interact with databases via a query language. One of the most popular query languages is MySQL. Python has a database specification, PEP 0249, which creates a consistent way to work with numerous database types. This makes the code we write more portable across databases and allows a richer span of database connectivity. To illustrate how simple this is, we are going to use the mysql.connector class as an example. MySQL is one of the most popular database formats, with a straightforward, human-readable query language. To practice using this class, you will need to have a MySQL server installed on your machine. This is available from https://dev.mysql.com/downloads/mysql/.

This should also come with a test database called world, which includes statistical data on world cities.

Ensure that the MySQL server is running, and run the following code:

```python
import mysql.connector
from mysql.connector import errorcode

cnx = mysql.connector.connect(user='root', password='password',
                               database='world', buffered=True)

cursor=cnx.cursor(buffered=True)
query=('select * from city where population > 1000000 order by population')
cursor.execute(query)
worldList=[]
for (city) in cursor:
    worldList.append([city[1],city[4]])
cursor.close()
cnx.close()
```

Data from the Web

Information on the web is structured into HTML or XML documents. Markup tags give us clear hooks for us to sample our data. Numeric data will often appear in a table, and this makes it relatively easy to use because it is already structured in a meaningful way. Let's look at a typical excerpt from an HTML document:

```html
<table border="0" cellpadding="5" cellspacing="2" class="details" width="95%">
```

```html
</table>
```
Turning Data into Information

This shows the first two rows of a table, with a heading and one row of data containing two values. Python has an excellent library, Beautiful Soup, for extracting data from HTML and XML documents. Here, we read some test data into an array, and put it into a format that would be suitable for input in a machine learning algorithm, say a linear classifier:

```python
import urllib
from bs4 import BeautifulSoup
import numpy as np

url = urllib.request.urlopen("http://interthing.org/dmls/species.html");
html = url.read()
soup = BeautifulSoup(html, "lxml")
table = soup.find("table")

headings = [th.get_text() for th in table.find("tr").find_all("th")]
datasets = []
for row in table.find_all("tr")[1:]:
    dataset = list(zip(headings, (td.get_text() for td in row.find_all("td"))));
    datasets.append(dataset)

nd=np.array(datasets)
```
features=nd[:,1:,1].astype('float')

targets=(nd[:,0,1:]).astype('str')

print(features)
print(targets)

As we can see, this is relatively straight forward. What we need to be aware of is that we are relying on our source web page to remain unchanged, at least in terms of its overall structure. One of the major difficulties with harvesting data off the web in this way is that if the owners of the site decide to change the layout of their page, it will likely break our code.

Another data format you are likely to come across is the JSON format. Originally used for serializing Javascript objects, JSON is not, however, dependent on JavaScript. It is merely an encoding format. JSON is useful because it can represent hierarchical and multivariate data structures. It is basically a collection of key value pairs:

```json
{"Languages": [{"Language":"Python","Version":"0"}, {"Language":"PHP","Version":"5"}],
 "OS": {"Microsoft": "Windows 10", "Linux": "Ubuntu 14"},
 "Name": "John the fictional Doe",
 "location": {"Street": "Some Street", "Suburb": "Some Suburb"},
 "Languages": [{"Language": "Python", "Version": "0"}, {"Language": "PHP", "Version": "5"}]
}
```

If we save the preceding JSON to a file called jsondata.json:

```python
import json
from pprint import pprint

with open('jsondata.json') as file:
    data = json.load(file)

pprint(data)
```
Data from natural language

Natural language processing is one of the more difficult things to do in machine learning because it focuses on what machines, at the moment, are not very good at: understanding the structure in complex phenomena.

As a starting point, we can make a few statements about the problem space we are considering. The number of words in any language is usually very large compared to the subset of words that are used in a particular conversation. Our data is sparse compared to the space it exists in. Moreover, words tend to appear in predefined sequences. Certain words are more likely to appear together. Sentences have a certain structure. Different social settings, such as at work, home, or out socializing; or in formal settings such as communicating with regulatory authorities, government, and bureaucratic settings, all require the use overlapping subsets of a vocabulary. A part from cues such as body language, intonation eye contact, and so forth, the social setting is probably the most important factor when trying to extract meaning from natural language.

To work with natural language in Python, we can use the the Natural Language Tool Kit (NLTK). If it is not installed, you can execute the `pip install -U nltk` command.

The NLTK also comes with a large library of lexical resources. You will need to download these separately, and NLTK has a download manager accessible through the following code:

```python
import nltk
nltk.download()
```

A window should open where you can browse through the various files. This includes a range of books and other written material, as well as various lexical models. To get started, you can just download the package, Book.

A text corpus is a large body of text consisting of numerous individual text files. NLTK comes with corpora from a variety of sources such as classical literature (the Gutenberg Corpus), the web and chat text, Reuter news, and corpus containing text categorized by genres such as new, editorial, religion, fiction, and so on. You can also load any collection of text files using the following code:

```python
from nltk.corpus import PlaintextCorpusReader
corpusRoot= 'path/to/corpus'
yourCorpus=PlaintextCorpusReader(corpusRoot, '.*')
```
The second argument to the `PlaintextCorpusReader` method is a regular expression indicating the files to include. Here, it simply indicates that all the files in that directory are included. This second parameter could also be a list of file locations, such as `['file1', 'dir2/file2']`.

Let's take a look at one of the existing corpora, and as an example, we are going to load the Brown corpus:

```python
from nltk.corpus import brown

cat=brown.categories()
print(cat)

['adventure', 'belles_lettres', 'editorial', 'fiction', 'government', 'hobbies', 'humor', 'learned', 'lore', 'mystery', 'news', 'religion', 'reviews', 'romance', 'science_fiction']
```

The Brown corpus is useful because it enables us to study the systemic differences between genres. Here is an example:

```python
from nltk.corpus import brown
cats=brown.categories()
for cat in cats:
    text=brown.words(categories=cat)
    fdist = nltk.FreqDist(w.lower() for w in text)
    posmod = ['love', 'happy', 'good', 'clean']
    negmod = ['hate', 'sad', 'bad', 'dirty']
    pcount=[]
    ncount=[]
    for m in posmod:
        pcount.append(fdist[m])
    for m in negmod:
        ncount.append(fdist[m])
    print(cat + ' positive: ' + str(sum(pcount)))
    print(cat + ' negative: ' + str(sum(ncount)))
    rat=sum(pcount)/sum(ncount)
    print('ratio= %s' %rat )
    print()
```

Here, we have sort of extracted sentiment data from different genres by comparing the occurrences of four positive sentiment words with their antonyms.
Data from images

Images are a rich and easily available source of data, and they are useful for learning applications such as object recognition, grouping, grading objects, as well as image enhancement. Images, of course, can be put together as a time series. Animating images is useful for both presentation and analysis; for example, we can use video to study trajectories, monitor environments, and learn dynamic behavior.

Image data is structured as a grid or matrix with color values assigned to each pixel. We can get a feel of how this works by using the Python Image Library. For this example, you will need to execute the following lines:

```python
from PIL import Image
from matplotlib import pyplot as plt
import numpy as np

image= np.array(Image.open('data/sampleImage.jpg'))
plt.imshow(image, interpolation='nearest')
plt.show()
print(image.shape)
```

Out[10]: (536, 800, 3)

We can see that this particular image is 536 pixels wide and 800 pixels high. There are 3 values per pixel, representing color values between 0 and 255, for red, green, and blue respectively. Note that the co-ordinate system's origin (0,0) is the top left corner. Once we have our images as NumPy arrays, we can start working with them in interesting ways, for example, taking slices:

```python
im2=image[0:100,0:100,2]
```

Data from application programming interfaces

Many social networking platforms have Application programming interfaces (APIs) that give the programmer access to various features. These interfaces can generate quite large amounts of streaming data. Many of these APIs have variable support for Python 3 and some other operating systems, so be prepared to do some research regarding the compatibility of systems.

Gaining access to a platform's API usually involves registering an application with the vendor and then using supplied security credentials, such as public and private keys, to authenticate your application.
Let’s take a look at the Twitter API, which is relatively easy to access and has a well-developed library for Python. To get started, we need to load the Twitter library. If you do not have it already, simply execute the `pip install twitter` command from your Python command prompt.

You will need a Twitter account. Sign in and go to apps.twitter.com. Click on the Create New App button and fill out the details on the Create An Application page. Once you have submitted this, you can access your credential information by clicking on your app from the application management page and then clicking on the Keys and Access Tokens tab.

The four items we are interested in here are the API Key, the API Secret, The Access token, and the Access Token secret. Now, to create our Twitter object:

```python
from twitter import Twitter, OAuth
# create our twitter object
auth=OAuth(accesToken, secretToken, apiKey, apiSecret)
t = Twitter(auth=auth)

# get our home time line
home=t.statuses.home_timeline()

# get a public timeline
anyone= t.statuses.user_timeline(screen_name="abc730")

# search for a hash tag
pycon=t.search.tweets(q="#pycon")

# The screen name of the user who wrote the first 'tweet'
user=anyone[0]['user']['screen_name']

# time tweet was created
created=anyone[0]['created_at']

# the text of the tweet
text= anyone[0]['text']
```

You will, of course, need to fill in the authorization credentials that you obtained from Twitter earlier. Remember that in a publicly accessible application, you never have these credentials in a human-readable form, and certainly not in the file itself, and preferably encrypted outside a public directory.
Signals

A form of data that is often encountered in primary scientific research is various binary streams. There are specific codecs for video and audio transmission and storage, and often, we are looking for higher-level tools to deal with each specific format. There are various signal sources we might be considering such as from a radio telescopes, sensor on a camera, or the electrical impulses from a microphone. Signals all share the same underlying principles based on wave mechanics and harmonic motion.

Signals are generally studied using time frequency analysis. The central concept here is that a continuous signal in time and space can be decomposed into frequency components. We use what is known as a **Fourier Transform** to move between the time and frequency domains. This utilizes the interesting fact that states that any given function, including non periodic functions, can be represented by a series of sine and cosine functions. This is illustrated by the following:

\[
F(x) = \frac{a_0}{2} + \sum_{n=1}^{m} (a_n \cos nx + b_n \sin nx)
\]

To make this useful, we need to find the values for \(a_n\) and \(b_n\). We do this by multiplying both sides of the equation cosine, \(mX\), and integrating. Here \(m\) is an integer.

\[
\int_{-\pi}^{\pi} f(x) \cos mx \, dx = \frac{a_0}{2} \int_{-\pi}^{\pi} \cos mx \, dx + \sum_{n=1}^{m} a_n \int_{-\pi}^{\pi} \cos nx \cos mx \, dx + b_n \int_{-\pi}^{\pi} \sin nx \cos mx \, dx
\]

This is called an **orthogonal function**, in a similar notion to how we consider \(x\), \(y\), and \(z\) to be orthogonal in a vector space. Now, if you can remember all your trigonometric functions, you will know that \(\sin\) times \(\cos\) with integer coefficients is always zero between negative \(\pi\) and \(\pi\). If we do the calculation, it turns out that the middle term on the left-hand side is zero, except when \(n\) equals \(m\). In this case, the term equals \(\pi\). Knowing this, we can write the following:

\[
a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos nx \, dx
\]
So, in the first step, if we multiply by $\sin mx$ instead of $\cosine mx$, then we can derive the value of $b_n$.

$$b_n = \int_{-\pi}^{\pi} f(x) \sin nx \, dx$$

We can see that we have decomposed a signal into a series of $\sin$ values and $\cos$ values. This enables us to separate the frequency components of a signal.

**Data from sound**

One of the most common and easy to study signals is audio. We are going to use the `soundfile` module. You can install it via `pip` if you do not have it. The `soundfile` module has a `wavfile.read` class that returns the `.wav` file data as a NumPy array. To try the following code, you will need a short 16 bit wave file called `audioSamp.wav`. This can be downloaded from davejulian.net/mlbook. Save it in your data directory, in your working directory:

```python
import soundfile as sf
import matplotlib.pyplot as plt
import numpy as np

sig, samplerate = sf.read('data/audioSamp.wav')
sig.shape
```

We see that the sound file is represented by a number of samples, each with two values. This is effectively the function as a vector, which describes the `.wav` file. We can, of course, create slices of our sound file:

```python
slice=sig[0:500, :]
```

Here, we slice the first 500 samples. Let’s calculate the Fourier transform of the slice and plot it:

```python
ft=np.abs(np.fft.fft(slice))
plt.plot(ft)
```

Finally, let’s plot the result:

```python
plt.plot(slice)
```
The output of the preceding commands is as follows:

![Graph of temperature data]

**Cleaning data**

To gain an understanding of which cleaning operations may be required for a particular dataset, we need to consider how the data was collected. One of the major cleaning operations involves dealing with missing data. We have already encountered an example of this in the last chapter, when we examined the temperature data. In this instance, the data had a quality parameter, so we could simply exclude the incomplete data. However, this may not be the best solution for many applications. It may be necessary to fill in the missing data. How do we decide what data to use? In the case of our temperature data, we could fill the missing values in with the average values for that time of year. Notice that we presuppose some domain knowledge, for example, the data is more or less periodic; it is in line with the seasonal cycle. So, it is a fair assumption that we could take the average for that particular date for every year we have a reliable record. However, consider that we are attempting to find a signal representing an increase in temperature due to climate change. In that case, taking the average for all years would distort the data and potentially hide a signal that could indicate warming. Once again, this requires extra knowledge and is specific about what we actually want to learn from the data.

Another consideration is that missing data may be one of three types, which are as follows:

- empty
- zero
- null
Different programming environments may treat these slightly differently. Out of the three, only zero is a measurable quantity. We know that zero can be placed on a number line before 1, 2, 3, and so on, and we can compare other numbers to zero. So, normally zero is encoded as numeric data. Empties are not necessarily numeric, and despite being empty, they may convey information. For example, if there is a field for middle name in a form, and the person filling out the form does not have a middle name, then an empty field accurately represents a particular situation, that is, having no middle name. Once again, this depends on the domain. In our temperature data, an empty field indicates missing data as it does not make sense for a particular day to have no maximum temperature. Null values, on the other hand, in computing, mean something slightly different from its everyday usage. For the computer scientist, null is not the same thing as no value or zero. Null values cannot be compared to anything else; they indicate that a field has a legitimate reason for not having an entry. Nulls are different than empty values. In our middle name example, a null value would indicate that it is unknown if the person has a middle name or not.

Another common data cleaning task is converting the data to a particular format. For our purposes here, the end data format we are interested in is a Python data structure such as a NumPy array. We have already looked at converting data from the JSON and HTML formats, and this is fairly straight forward.

Another format that we are likely to come across is the Acrobat's Portable Document Format (PDF). Importing data from PDF files can be quite difficult because PDF files are built on page layout primitives, and unlike HTML or JSON, they do not have meaningful markup tags. There are several non-Python tools for turning PDFs into text such as pdftotext. This is a command line tool that is included in many Linux distributions and is also available for Windows. Once we have converted the PDF file into text, we still need to extract the data, and the data embedded in the document determines how we can extract it. If the data is separated from the rest of the document, say in a table, then we can use Python's text parsing tools to extract it. Alternatively, we can use a Python library for working with PDF documents such as pdfminer3k.

Another common cleaning task is converting between data types. There is always the risk of losing data when converting between types. This happens when the target type stores less data than the source, for instance, converting to float 16 from float 32. Sometimes, we need to convert data at the file level. This occurs when a file has an implicit typing structure, for example, a spreadsheet. This is usually done within the application that created the file. For example, an Excel spreadsheet can be saved as a comma separated text file and then imported into a Python application.
Visualizing data

There are a number of reasons for why we visually represent the data. At the data exploration stage, we can gain an immediate understanding of data properties. Visual representation serves to highlight patterns in data and suggest modeling strategies. Exploratory graphs are usually made quickly and in large numbers. We are not so much concerned with aesthetic or stylistic issues, but we simply want to see what the data looks like.

Beyond using graphs to explore data, they are a primary means of communicating information about our data. Visual representation helps clarify data properties and stimulate viewer engagement. The human visual system is the highest bandwidth channel to the brain, and visualization is the most efficient way to present a large amount of information. By creating a visualization, we can immediately get a sense of important parameters, such as the maximum, minimum, and trends that may be present in the data. Of course, this information can be extracted from data through statistical analysis, however, analysis may not reveal specific patterns in the data that visualization will. The human visual pattern recognition system is, at the moment, significantly superior to that of a machine. Unless we have clues as to what we are looking for, algorithms may not pick out important patterns that a human visual system will.

The central problem for data visualization is mapping data elements to visual attributes. We do this by first classifying the data types as nominal, ordinal, or quantitative, and then determining which visual attributes represent each data type most effectively. Nominal or categorical data refers to a name, such as the species, male or female, and so on. Nominal data does not have a specific order or numeric value. Ordinal data has an intrinsic order, such as house numbers in a street, but is different from quantitative data in that it does not imply a mathematical interval. For example, it does not make much sense to multiply or divide house numbers. Quantitative data has a numeric value such as size or volume. Clearly, certain visual attributes are inappropriate for nominal data, such as size or position; they imply ordinal or quantitative information.
Sometimes, it is not immediately clear what each data type in a particular dataset is. One way to disambiguate this is to find what operations are applicable for each data type. For example, when we are comparing nominal data, we can use equals, for instance, the species *Whitefly* is not equal to the species *Thrip*. However, we cannot use operations such as greater than or less than. It does not make sense to say, in an ordinal sense, that one species is greater than another. With ordinal data, we can apply operations such as greater than or less than. Ordinal data has an implicit order that we can map on a number line. For quantitative data, this consists of an interval, such as a date range, to which we can apply additional operations such as subtractions. For example, we can not only say that a particular date occurs after another date, but we can also calculate the difference between the two dates. With quantitative data that has a fixed axis, that is a ratio of some fixed amount as opposed to an interval, we can use operations such as division. We can say that a particular object weighs twice as much or is twice as long as another object.

Once we are clear on our data types, we can start mapping them to attributes. Here, we will consider six visual attributes. They are position, size, texture, color, orientation, and shape. Of these, only position and size can accurately represent all three types of data. Texture, color, orientation, and shape, on the other hand, can only accurately represent nominal data. We cannot say that one shape or color is greater than another. However, we can associate a particular color or texture with a name.

Another thing to consider is the perceptual properties of these visual attributes. Research in psychology and psycho physics have established that visual attributes can be ranked in terms of how accurately they are perceived. Position is perceived most accurately, followed by length, angle, slope, area, volume, and finally, color and density, which are perceived with the least accuracy. It makes sense, therefore, to assign position and then length to the most important quantitative data. Finally, it should also be mentioned that we can encode, to some extent, ordinal data in a color value (from dark to light) or continuous data in a color gradient. We cannot generally encode this data in a color hue. For instance, there is no reason to perceive the color blue as somehow greater than the color red, unless you are making a reference to its frequency.

![The color gradient to represent ordinal data](image-url)
The next thing to consider is the number of dimensions that we need to display. For uni-variate data, that is, where we only need to display one variable, we have many choices such as dots, lines, or box plots. For bi-variate data, where we need to display two dimensions, the most common is with a scatter plot. For tri-variate data, it is possible to use a 3D plot, and this can be useful for plotting geometric functions such as manifolds. However, 3D plots have some drawbacks for many data types. It can be a problem to work out relative distances on a 3D plot. For instance, in the following figure, it is difficult to gauge the exact positions of each element. However, if we encode the z dimension as size, the relative values become more apparent:

![Encoding Three Dimensions](image)

There is a large design space for encoding data into visual attributes. The challenge is to find the best mapping for our particular dataset and purpose. The starting point should be to encode the most important information in the most perceptually accurate way. Effective visual coding will depict all the data and not imply anything that is not in the data. For example, length implies quantitative data, so encoding non-quantitative data into length is incorrect. Another aspect to consider is consistency. We should choose attributes that make the most sense for each data type and use consistent and well-defined visual styles.
Summary
You have learned that there are a large number of data source, formats, and structures. You have hopefully gained some understanding of how to begin working with some of them. It is important to point out that in any machine learning project, working with the data at this fundamental level can comprise a significant proportion of the overall project development time.

In the next chapter, we will look at how we can put our data to work by exploring the most common machine learning models.
Where to buy this book

You can buy Designing Machine Learning Systems with Python from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.