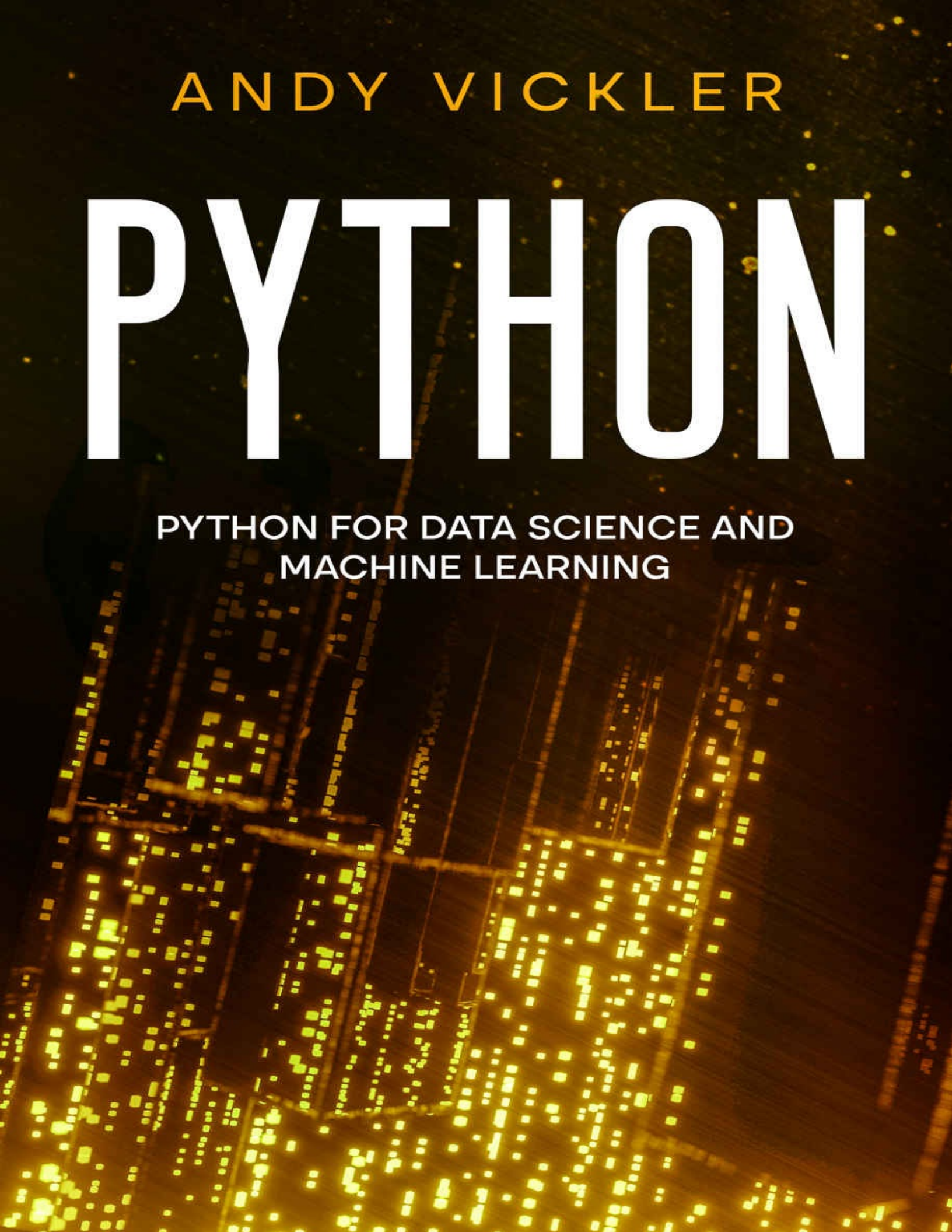
ANDY VICKLER

PYTHON

PYTHON FOR DATA SCIENCE AND MACHINE LEARNING



Python

Python for Data Science and Machine Learning

Table of Contents

Introduction

Part One – An Introduction to Data Science and Machine Learning

What Is Data Science?

How Important Is Data Science?

Data Science Limitations

What Is Machine Learning?

How Important Is Machine Learning?

Machine Learning Limitations

Data Science vs. Machine Learning

Part Two – Introducing NumPy

What Is NumPy Library?

How to Create a NumPy Array

Shaping and Reshaping a NumPy array

Index and Slice a NumPy Array

Stack and Concatenate NumPy Arrays

Broadcasting in NumPy Arrays

NumPy Ufuncs

Doing Math with NumPy Arrays

NumPy Arrays and Images

Part Three – Data Manipulation with Pandas

Question One: How Do I Create a Pandas DataFrame?

Question Two – How Do I Select a Column or Index from a DataFrame?

Question Three: How Do I Add a Row, Column, or Index to a DataFrame?

Question Four: How Do I Delete Indices, Rows, or Columns From a Data Frame?

Question Five: How Do I Rename the Columns or Index of a DataFrame?

Question Six: How Do I Format the DataFrame Data?

Question Seven: How Do I Create an Empty DataFrame?

Question Eight: When I Import Data, Will Pandas Recognize Dates?
Question Nine: When Should a DataFrame Be Reshaped? Why and How?
Question Ten: How Do I Iterate Over a DataFrame?
Question Eleven: How Do I Write a DataFrame to a File?

Part Four – Data Visualization with Matplotlib and Seaborn

Using Matplotlib to Generate Histograms
Using Matplotlib to Generate Scatter Plots
Using Matplotlib to Generate Bar Charts
Using Matplotlib to Generate Pie Charts
Visualizing Data with Seaborn
Using Seaborn to Generate Histograms
Using Seaborn to Generate Scatter Plots
Using Seaborn to Generate Heatmaps
Using Seaborn to Generate Pairs Plot

Part Five – An In-Depth Guide to Machine Learning

Machine Learning Past to Present
Machine Learning Features
Different Types of Machine Learning
Common Machine Learning Algorithms
Gaussian Naive Bayes classifier
K-Nearest Neighbors
Support Vector Machine Learning Algorithm
Fitting Support Vector Machines
Linear Regression Machine Learning Algorithm
Logistic Regression Machine Learning Algorithm
A Logistic Regression Model
Decision Tree Machine Learning Algorithm
Random Forest Machine Learning Algorithm
Artificial Neural Networks Machine Learning Algorithm
Machine Learning Steps

Evaluating a Machine Learning Model
Model Evaluation Metrics
Regression Metrics
Implementing Machine Learning Algorithms with Python
Advantages and Disadvantages of Machine Learning

Conclusion

References

Introduction

Thank you for purchasing this guide about data science and machine learning with Python. One of the first questions people ask is, what is data science? This is not a particularly easy question to answer because the term is widespread these days, used just about everywhere. Some say it is nothing more than an unnecessary label, given that science is all about data anyway, while others say it is just a buzzword.

However, many fail to see or choose to ignore because data science is quite possibly the ONLY label that could be given to the cross-discipline skill set that is fast becoming one of the most important in applications. It isn't possible to place data science firmly into one discipline; that much is definitely true. It comprises three areas, all distinct and all overlapping:

- **Statistician –** these people are skilled in modeling datasets and summarizing them. This is an even more important skill set these days, given the ever-increasing size of today's datasets.
- **Computer Scientist –** these people design algorithms and use them to store data, and process it, and visualize it efficiently.
- **Domain Expertise -** is akin to being classically trained in any subject, i.e. having expert knowledge. This is important to ensure the right questions are asked and the answers placed in the right context.

So, going by this, it wouldn't be remiss of me to encourage you to see data science as a set of skills you can learn and apply in your own expertise area, rather than being something new you have to learn from scratch. It doesn't matter if you are examining microspore images looking for microorganisms, forecasting returns on stocks, optimizing online ads to get more clicks or any other field where you work with data. This book will give you the fundamentals you need to learn to ask the right questions in your chosen expertise area.

Who This Book Is For

This book is aimed at those with experience of programming in Python, designed to help them further their skills and learn how to use their chosen programming language to go much further than before. It is not aimed at those with no experience of Python or programming, and I will not be giving you a primer on the language.

The book assumes that you already have experience defining and using

functions, calling object methods, assigning variables, controlling program flow, and all the other basic Python tasks. It is designed to introduce you to the data science libraries in Python, such as NumPy, Pandas, Seaborn, Scikit-Learn, and so on, showing you how to use them to store and manipulate data and gain valuable insights you can use in your work.

Why Use Python?

Over the last few decades, Python has emerged as the best and most popular tool for analyzing and visualizing data, among other scientific tasks. When Python was first designed, it certainly wasn't with data science in mind. Instead, its use in these areas has come from the addition of third-party packages, including:

- **NumPy –** for manipulating array-based data of the same kind
- **Pandas –** for manipulating array-based data of different kinds
- **SciPy –** for common tasks in scientific computing
- **Matplotlib –** for high quality, accurate visualizations
- **Scikit-Learn** – for machine learning

And so many more.

Python 2 vs. 3

The code used in this book follows the Python 3 syntax. Many enhancements are not backward compatible with Python 2, which is why you are always urged to install the latest Python version.

Despite Python 3 being released around 13 years ago, people have been slow to

adopt it, especially in web development and scientific communities. This is mostly down to the third-party packages not being made compatible right from the start. However, from 2014, those packages started to become available in stable releases, and more people have begun to adopt Python 3.

What's in This Book?

This book has been separated into five sections, each focusing on a specific area:

- **Part One –** introduces you to data science and machine learning
- **Part Two –** focuses on NumPy and how to use it for storing and manipulating data arrays
- **Part Three –** focuses on Pandas and how to use it for storing and manipulating columnar or labeled data arrays
- **Part Four –** focuses on using Matplotlib and Seaborn for data visualization
- **Part Five –** takes up approximately half of the book and focuses on machine learning, including a discussion on algorithms.

There is more to data science than all this, but these are the main areas you should focus on as you start your data science and machine learning journey.

Installation

Given that you should have experience in Python, I am assuming you already have Python on your machine. However, if you don't, there are several ways to install it, but Anaconda is the most common way. There are two versions of this:

- **Miniconda –** provides you with the Python interpreter and conda, a command-line tool
- **Anaconda –** provides conda and Python, along with many other packages aimed at data science. This is a much larger installation, so make sure you have plenty of free space.

It's also worth noting that the packages discussed in this book can be installed on top of Miniconda, so it's down to you which version you choose to install. If you choose Miniconda, you can use the following command to install all the packages you need:

```
[~]$ conda install numpy pandas scikit-learn matplotlib seaborn jupyter
```

All packages and tools can easily be installed with the following:

```
conda install packagename
```

ensuring you type the correct name in place of packagename.

Let's not waste any more time and dive into our introduction to data science and machine learning.

Part One – An Introduction to Data Science and Machine Learning

The definition of data science tells us that it is a research area used to derive valuable insight from data, and it does this using a theoretical method. In short, data science is a combination of simulation, technology, and market management.

Machine learning is a bit different. This is all about learning the data science techniques the computers use to learn from the data we give them. We use these technologies to gain outcomes without needing to program any specific laws.

Machine learning and data science are trending high these days and, although many people use the terms interchangeably, they are different. So, let's break this down and learn what we need to know to continue with the rest of the book.

What Is Data Science?

The real meaning of data science is found in the deep analysis of huge amounts of data contained in a company's archive. This involves working out where that data originated from, whether the data is of sufficient quality, and whether it can be used to move the business on in the future.

Often, an organization stores its data in one of two formats – unstructured or organized. Examining the data gives us useful insights into the consumer and industry dynamics, which can be used to provide an advantage to the company over their rivals—this is done by looking for patterns in the data.

Data scientists know how to turn raw data into something a business can use. They know how to spot algorithmic coding and process the data, with knowledge of statistics and artificial learning. Some of the top companies in the world use data analytics, including Netflix, Amazon, airlines, fraud prevention departments, healthcare, etc.

Data Science Careers

Most businesses use data analysis effectively to expand, and data scientists are in the highest demand across many industries. If you are considering getting involved in data science, these are some of the best careers you can train for:

- **Data Scientist –** investigates trends in the data to see what effect they will have on a business. Their primary job is to determine the significance of the data and clarify it so that everyone can understand it.
- **Data Analyst –** analyzes data to see what industry trends exist. They help develop simplistic views of where a business stands in its industry.

- **Data Engineer –** often called the backbone of a business, data engineers create databases to store data, designing them for the best use, and manage them. Their job entails data pipeline design, ensuring the data flows adequately, and getting to where it needs to be.
- **Business Intelligence Analyst -** sometimes called a market intelligence consultant-study data to improve income and productivity for a business. The job is less theoretical and more scientific and requires a deep understanding of common machines.

How Important Is Data Science?

In today's digital world, a world full of data, it is one of the most important jobs. Here are some of the reasons why.

- First, it gives businesses a better way of identifying who their customers are. Given that customers keep a business's wheels turning, they determine whether that business fails or succeeds. With data science, businesses can now communicate with customers in the right way, in different ways for different customers.
- Second, data science makes it possible for a product to tell its story in an entertaining and compelling way, one of the main reasons why data science is so popular. Businesses and brands can use the information gained from data science to communicate what they want their customers to know, ensuring much stronger relationships between them.
- Next, the results gained from data science can be used in just about every field, from schooling to banking, healthcare to tourism, etc. Data science allows a business to see problems and respond to them accordingly.
- A business can use data analytics to better communicate with its customers/consumers and provide the business with a better view of how its products are used.
- Data science is quickly gaining traction in just about every market and is now a critical part of how products are developed. That has led to a rise in the need for data scientists to help manage data and find the answers to some of the more complex problems a business faces.

Data Science Limitations

Data science may be one of the most lucrative careers in the world right now, but it isn't perfect and has its fair share of drawbacks. It wouldn't be right to brag about what data science can do without pointing out its limitations:

- Data science is incredibly difficult to master. That's because it isn't just one discipline involving a combination of information science, statistics, and mathematics, to name a few. It isn't easy to master every discipline involved to a competent level.
- Data science also requires a high level of domain awareness. In fact, it's fair to say that it relies on it. If you have no knowledge of computer science and statistics, you will find it hard to solve any data science problem.
- Data privacy is a big issue. Data makes the world go around, and data scientists create data-driven solutions for businesses. However, there is always a chance that the data they use could infringe privacy. A business will always have access to sensitive data and, if data protection is compromised in any way, it can lead to data breaches.

What Is Machine Learning?

Machine learning is a subset of data science, enabling machines to learn things without the need for a human to program them. Machine learning analyzes data using algorithms and prepares possible predictions without human intervention. It involves the computer learning a series of inputs in the form of details, observations, or commands, and it is used extensively by companies such as Google and Facebook.

Machine Learning Careers

There are several directions you can take once you have mastered the relevant skills. These are three of those directions:

- **Machine Learning Engineer –** this is one of the most prized careers in the data science world. Engineers typically use machine learning algorithms to ensure machine learning applications and systems are as effective as they can be. A machine learning engineer is responsible for molding self-learning applications, improving their effectiveness and efficiency by using the test results to fine-tune them and run statistical analyses. Python is one of the most used programming languages for performing machine learning experiments.
- **NLP Scientist –** these are concerned with Natural Language Processing, and their job is to ensure machines can interpret natural languages. They design and engineer software and computers to learn human speech habits and convert the spoken word into different languages. Their aim is to get

computers to understand human languages the same way we do, and two of the best examples are apps called DuoLingo and Grammarly.

- **Developer/Engineer of Software –** these develop intelligent computer programs, and they specialize in machine learning and artificial intelligence. Their main aim is to create algorithms that work effectively and implement them in the right way. They design complex functions, plan flowcharts, graphs, layouts, product documentation, tables, and other visual aids. They are also responsible for composing code and evaluating it, creating tech specs, updating programs and managing them, and more.

How Important Is Machine Learning?

Machine learning is an ever-changing world and, the faster it evolves, the higher its significance is, and the more demand grows. One of the main explanations as to why data scientists can't live without machine learning is this – "high-value forecasts that direct smart decisions and behavior in real-time, without interference from humans."

It is fast becoming popular to interpret huge amounts of data and helping to automate the tasks that data scientists do daily. It's fair to say that machine learning has changed the way we extract data and visualize it. Given how much businesses rely on data these days, data-driven decisions help determine if a company is likely to succeed or fall behind its competitors.

Machine Learning Limitations

Like data science, machine learning also has its limitations, and these are some of the biggest ones:

- Algorithms need to store huge amounts of training data. Rather than being programmed, an AI program is educated. This means they require vast amounts of data to learn how to do something and execute it at a human level. In many cases, these huge data sets are not easy to generate for specific uses, despite the rapid level at which data is being produced.
  Neural networks are taught to identify things, for example, images, and they do this by being trained on massive amounts of labeled data in a process known as supervised learning. No matter how large or small, any change to the assignment requires a new collection of data, which also needs to be prepared. It's fair to say that a neural network cannot truly operate at a human intelligence level because of the brute force needed to

get it there – that will change in the future.

- Machine learning needs too much time to mark the training data. AI uses supervised learning on a series of deep neural nets. It is critical to label the data in the processing stage, and this is done using predefined goal attributes taken from historical data. Marking the data is where the data is cleaned and sorted to allow the neural nets to work on it.
  There's no doubt that vast amounts of labeled information are needed for deep learning and, while this isn't particularly hard to grasp, it isn't the easiest job to do. If unlabeled results are used, the program cannot learn to get smarter.
- AI algorithms don't always work well together. Although there have been some breakthroughs in recent times, an AI model may still only generalize a situation if it is asked to do something it didn't do in training. AI models cannot always move data between groups of conditions, implying that whatever is accomplished by a paradigm with a specific use case can only be relevant to that case. Consequently, businesses must use extra resources to keep models trained and train new ones, despite the use cases being the same in many cases.

Data Science vs. Machine Learning

Data scientists need to understand data analysis in-depth, besides having excellent programming abilities. Based on the business hiring the data scientist, there is a range of expertise, and the skills can be split down into two different categories:

Technical Skills

You need to specialize in:

- Algebra
- Computer Engineering
- Statistics

You must also be competent in:

- Computer programming
- Analytical tools, such as R, Spark, Hadoop, and SAS
- Working with unstructured data obtained from different networks

Non-Technical Skills

Most of a data scientist's abilities are non-technical and include:

- A good sense of business
- The ability to interact
- Data intuition

Machine learning experts also need command over certain skills, including:

- **Probability and statistics –** theory experience is closely linked to how you comprehend algorithms, such as Naïve Bayes. Hidden Markov, Gaussian Mixture, and many more. Being experienced in numbers and chance makes these easier to grasp.
- **Evaluating and modeling data –** frequently evaluating model efficacy is a critical part of maintaining measurement reliability in machine learning. Classification, regression, and other methods are used to evaluate consistency and error rates in models, along with assessment plans, and knowledge of these is vital.
- **Machine learning algorithms –** there are tons of machine learning algorithms, and you need to have knowledge of how they operate to know which one to use for each scenario. You will need knowledge of quadratic programming, gradient descent, convex optimization, partial differential equations, and other similar topics.
- **Programming languages –** you will need experience in programming in one or more languages – Python, R, Java, C++, etc.
- **Signal processing techniques –** feature extraction is one of the most critical factors in machine learning. You will need to know some specialized processing algorithms, such as shearlets, bandlets, curvelets, and contourlets.

Data science is a cross-discipline sector that uses huge amounts of data to gain insights. At the same time, machine learning is an exciting subset of data science, used to encourage machines to learn by themselves from the data provided.

Both have a huge amount of use cases, but they do have limits. Data science may be strong, but, like anything, it is only effective if the proper training and best-quality data are used.

Let's move on and start looking into using Python for data science. Next, we introduce you to NumPy.

Part Two – Introducing NumPy

NumPy is one of the most basic Python libraries, and, over time, you will come to rely on it in all kinds of data science tasks, be they simple math to classifying images. NumPy can handle data sets of all sizes efficiently, even the largest ones, and having a solid grasp on how it works is essential to your success in data science.

First, we will look at what NumPy is and why you should use it over other methods, such as Python lists. Then we will look at some of its operations to understand exactly how to use it – this will include plenty of code examples for you to study.

What Is NumPy Library?

NumPy is a shortened version of **Num** erical **Py** thon, and it is, by far, the most scientific library Python has. It supports multidimensional array objects, along with all the tools needed to work with those arrays. Many other popular libraries, like Pandas, Scikit-Learn, and Matplotlib, are all built on NumPy.

So, what is an array? If you know your basic Python, you know that an array is a collection of values or elements with one or more dimensions. A one-dimensional array is a Vector, while a two-dimensional array is a Matrix.

NumPy arrays are known as N-dimensional arrays, otherwise called ndarrays, and the elements they store are the same type and same size. They are high-performance, efficient at data operations, and an effective form of storage as the arrays grow larger.

When you install Anaconda, you get NumPy with it, but you can install it on your machine separately by typing the following command into the terminal:

```
pip install numpy
```

Once you have done that, the library must be imported, using this command:

```
import numpy as np
```

NOTE

np is the common abbreviation for NumPy

Python Lists vs. NumPy Arrays

Those used to using Python will probably be wondering why we would use the NumPy arrays when we have the perfectly good Python Lists at our disposal. These lists already act as arrays to store various types of elements, after all, so

why change things?

Well, the answer to this lies in how objects are stored in memory by Python.

Python objects are pointers to memory locations where object details are stored. These details include the value and the number of bytes. This additional information is part of the reason why Python is classed as a dynamically typed language, but it also comes with an additional cost. That cost becomes clear when large object collections, such as arrays are stored.

A Python list is an array of pointers. Each pointer points to a specific location containing information that relates to the element. As far as computation and memory go, this adds a significant overhead cost. To make matters worse, when all the stored objects are the same type, most of the additional information becomes redundant, meaning the extra cost for no reason.

NumPy arrays overcome this issue. These arrays only store objects of the same type, otherwise known as homogenous objects. Doing it this way makes storing the array and manipulating it far more efficient, and we can see this difference clearly when there are vast amounts of elements in the array, say millions of them. We can also carry out element-wise operations on NumPy arrays, something we cannot do with a Python list.

This is why we prefer to use these arrays over lists when we want mathematical operations performed on large quantities of data.

How to Create a NumPy Array

Basic ndarray

NumPy arrays are simple to create, regardless of which one it is and the complexity of the problem it is solving. We'll start with the most basic NumPy array, the ndarray.

All you need is the following np.array() method, ensuring you pass the array values as a list:

```
np.array([1,2,3,4])
```

The output is:

```
array([1, 2, 3, 4])
```

Here, we have included integer values and the dtype argument is used to specify the data type:

```
np.array([1,2,3,4],dtype=np.float32)
```

The output is:

```
array([1., 2., 3., 4.], dtype=float32)
```

Because you can only include homogenous data types in a NumPy array if the data types don't match, the values are upcast:

```
np.array([1,2.0,3,4])
```

The output is:

```
array([1., 2., 3., 4.])
```

In this example, the integer values are upcast to float values.

You can also create multidimensional arrays:

```
np.array([[1,2,3,4],[5,6,7,8]])
array([[1, 2, 3, 4],
    [5, 6, 7, 8]])
```

This creates a two-dimensional array containing values.

NOTE

The above example is a 2 x 4 matrix. A matrix is a rectangular array containing numbers. Its shape is N x M – N indicates how many rows there are, and M indicates how many columns there are.

An Array of Zeroes

You can also create arrays containing nothing but zeros. This is done with the np.zeros() method, ensuring you pass the shape of the array you want:

```
np.zeros(5)
array([0., 0., 0., 0., 0.])
```

Above, we have created a one-dimensional array, while the one below is a two-dimensional array:

```
np.zeros((2,3))
array([[0., 0., 0.],
    [0., 0., 0.]])
```

An Array of Ones

The np.ones() method is used to create an array containing 1s:

```
np.ones(5,dtype=np.int32)
array([1, 1, 1, 1, 1])
```

Using Random Numbers in a ndarray

The np.random.rand() method is commonly used to create ndarray's with a given shape containing random values from [0,1):

```
# random
np.random.rand(2,3)
array([[0.95580785, 0.98378873, 0.65133872],
    [0.38330437, 0.16033608, 0.13826526]])
```

Your Choice of Array

You can even create arrays with any value you want with the np.full() method, ensuring the shape of the array you want is passed in along with the desired value:

```
np.full((2,2),7)
array([[7, 7],
    [7, 7]])
```

NumPy IMatrix

The np.eye() method is another good one that returns 1s on the diagonal and 0s everywhere else. More formally known as an Identity Matrix, it is a square matric with an N x N shape, meaning there is the same number of rows as columns. Below is a 3 x 3 IMatrix:

```
# identity matrix
np.eye(3)
array([[1., 0., 0.],
    [0., 1., 0.],
    [0., 0., 1.]])
```

However, you can also change the diagonal where the values are 1s. It could be above the primary diagonal:

```
# not an identity matrix
np.eye(3,k=1)
array([[0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 0.]])
```

Or below it:

```
np.eye(3,k=-2)
array([[0., 0., 0.],
       [0., 0., 0.],
       [1., 0., 0.]])
```

NOTE

A matrix can only be an IMatrix when the 1s appear on the main diagonal, no other.

Evenly-Spaced ndarrays

The np.arange() method provides evenly-spaced number arrays:

```
np.arange(5)
array([0, 1, 2, 3, 4])
```

We can explicitly define the start and end and the interval step size by passing in an argument for each value.

NOTE

We define the interval as [start, end) and the final number is not added into the array:

```
np.arange(2,10,2)
array([2, 4, 6, 8])
```

Because we defined the step size as 2, we got the alternate elements as a result. The final element was 10, and this was not included.

A similar function is called np.linspace(), which takes an argument in the form of the number of samples we want from the interval.

NOTE

This time, the last number will be included in the result.

```
np.linspace(0,1,5)
array([0. , 0.25, 0.5 , 0.75, 1.  ])
```

That is how to create a variety of arrays with NumPy, but there is something else just as important – the array's shape.

Shaping and Reshaping a NumPy array

When your ndarray has been created, you need to check three things:

- How many axes the ndarray has
- The shape of the array
- The size of the array

NumPy Array Dimensions

Determining how many axes or dimensions the ndarray has is easily done using an attribute called ndims:

```
# number of axis
a = np.array([[5,10,15],[20,25,20]])
print('Array :','\n',a)
print('Dimensions :','\n',a.ndim)
Array :
[[ 5 10 15]
[20 25 20]]
Dimensions :
2
```

Here, we have a two-dimensional array with two rows and three columns.

NumPy Array Shape

The array shape is an array attribute showing the number of rows with elements

along each of the dimensions. The shape can be further indexed so the result shows the value on each dimension:

```
a = np.array([[1,2,3],[4,5,6]])
print('Array :','\n',a)
print('Shape :','\n',a.shape)
print('Rows = ',a.shape[0])
print('Columns = ',a.shape[1])
```

```
Array :
[[1 2 3]
[4 5 6]]
Shape :
(2, 3)
Rows =  2
Columns =  3
```

Array Size

The size attribute tells you the number of values in your array by multiplying the number of rows by columns:

```
# size of array
a = np.array([[5,10,15],[20,25,20]])
print('Size of array :',a.size)
print('Manual determination of size of array :',a.shape[0]*a.shape[1])
```

```
Size of array : 6
Manual determination of size of array : 6
```

Reshaping an Array

You can use the np.reshape() method to reshape your ndarray without changing any of the data in it:

```
# reshape
a = np.array([3,6,9,12])
```

```
np.reshape(a,(2,2))
array([[ 3,  6],
       [ 9, 12]])
```

We reshaped this from a one-dimensional to a two-dimensional array.

When you reshape your array and are not sure of any of the axes shapes, input -1. When NumPy sees -1, it will calculate the shape automatically:

```
a = np.array([3,6,9,12,18,24])
print('Three rows :','\n',np.reshape(a,(3,-1)))
print('Three columns :','\n',np.reshape(a,(-1,3)))
Three rows :
[[ 3  6]
[ 9 12]
[18 24]]
Three columns :
[[ 3  6  9]
[12 18 24]]
```

Flattening Arrays

There may be times when you want to collapse a multidimensional array into a single-dimensional array. There are two methods you can use for this – flatten() or ravel():

```
a = np.ones((2,2))
b = a.flatten()
c = a.ravel()
print('Original shape :', a.shape)
print('Array :','\n', a)
print('Shape after flatten :',b.shape)
print('Array :','\n', b)
print('Shape after ravel :',c.shape)
```

```
print('Array :','\n', c)
Original shape : (2, 2)
Array :
[[1. 1.]
[1. 1.]]
Shape after flatten : (4,)
Array :
[1. 1. 1. 1.]
Shape after ravel : (4,)
Array :
 [1. 1. 1. 1.]
```

However, one critical difference exists between the two methods – flatten() will return a copy of the original while ravel() only returns a reference. If there are any changes to the array revel() returns, the original array will reflect them, but this doesn't happen with the flatten() method.

```
b[0] = 0
print(a)
[[1. 1.]
[1. 1.]]
```

The original array did not show the changes.

```
c[0] = 0
print(a)
[[0. 1.]
[1. 1.]]
```

But this one did show the changes.

A deep copy of the ndarray is created by flatten(), while a shallow copy is created by ravel().

A new ndarray is created in a deep copy, stored in memory, with the object

flatten() returns pointing to that location in memory. That is why changes will not reflect in the original.

A reference to the original location is returned by a shallow copy which means the object ravel() returns points to the memory location where the original object is stored. That means changes will be reflected in the original.

Transposing a ndarray

The transpose() method is a reshaping method that takes the input array, swapping the row values with the columns and vice versa:

```
a = np.array([[1,2,3],
[4,5,6]])
b = np.transpose(a)
print('Original','\n','Shape',a.shape,'\n',a)
print('Expand along columns:','\n','Shape',b.shape,'\n',b)
Original
Shape (2, 3)
[[1 2 3]
[4 5 6]]
Expand along columns:
Shape (3, 2)
[[1 4]
[2 5]
[3 6]]
```

When you transpose a 2 x 3 array, the result is a 3 x 2 array.

Expand and Squeeze NumPy Arrays

Expanding a ndarray involves adding new axes. This is done with the method called expand_dims(), passing the array and the axis you want to expand along:

```
# expand dimensions
a = np.array([1,2,3])
```

```
b = np.expand_dims(a,axis=0)
c = np.expand_dims(a,axis=1)
print('Original:','\n','Shape',a.shape,'\n',a)
print('Expand along columns:','\n','Shape',b.shape,'\n',b)
print('Expand along rows:','\n','Shape',c.shape,'\n',c)
Original:
Shape (3,)
[1 2 3]
Expand along columns:
Shape (1, 3)
[[1 2 3]]
Expand along rows:
Shape (3, 1)
[[1]
[2]
[3]]
```

However, if you want the axis reduced, you would use the squeeze() method. This will remove an axis with one entry so, where you have a matrix of 2 x 2 x 1, the third dimension is removed.

```
# squeeze
a = np.array([[[1,2,3],
[4,5,6]]])
b = np.squeeze(a, axis=0)
print('Original','\n','Shape',a.shape,'\n',a)
print('Squeeze array:','\n','Shape',b.shape,'\n',b)
Original
Shape (1, 2, 3)
```

```
[[[1 2 3]
  [4 5 6]]]
Squeeze array:
Shape (2, 3)
[[1 2 3]
[4 5 6]]
```

Where you have a 2 x 2 matrix and you try to use squeeze() you would get an error:

```
# squeeze
a = np.array([[1,2,3],
[4,5,6]])
b = np.squeeze(a, axis=0)
print('Original','\n','Shape',a.shape,'\n',a)
print('Squeeze array:','\n','Shape',b.shape,'\n',b)cv
```

The error message will point to line 3 being the issue, providing a message saying:

```
ValueError:   cannot select an axis to squeeze out which has size not equal to one
```

Index and Slice a NumPy Array

We know how to create NumPy arrays and how to change their shape and size. Now, we will look at using indexing and slicing to extract values.

Slicing a One-Dimensional Array

When you slice an array, you retrieve elements from a specified "slice" of the array. You must provide the start and end points like this [start: end], but you can also take things a little further and provide a step size. Let's say you alternative element in the array printed. In that case, your step size would be defined as 2, which means you want the element 2 steps in from the current index. This would look something like [start: end:step-size]:

```
a = np.array([1,2,3,4,5,6])
```

```
print(a[1:5:2])
```

```
[2 4]
```

Note the final element wasn't considered – when you slice an array, only the start index is included, not the end index.

You can get around this by writing the next higher value to your final value:

```
a = np.array([1,2,3,4,5,6])
print(a[1:6:2])
```

```
[2 4 6]
```

If the start or end index isn’t specified, it will default to 0 for the start and the array size for the end index, with a default step size of 1:

```
a = np.array([1,2,3,4,5,6])
print(a[:6:2])
print(a[1::2])
print(a[1:6:])
```

```
[1 3 5]
[2 4 6]
[2 3 4 5 6]
```

Slicing a Two-Dimensional Array

Two-dimensional arrays have both columns and rows so slicing them is not particularly easy. However, once you understand how it's done, you will be able to slice any array you want.

Before we slice a two-dimensional array, we need to understand how to retrieve elements from it:

```
a = np.array([[1,2,3],
[4,5,6]])
print(a[0,0])
print(a[1,2])
print(a[1,0])
```

```
1
6
4
```

Here, we identified the relevant element by supplying the row and column values. We only need to provide the column value in a one-dimensional array because the array only has a single row.

Slicing a two-dimensional array requires that a slice for both the column and the row are mentioned:

```
a = np.array([[1,2,3],[4,5,6]])
# print first row values
print('First row values :','\n',a[0:1,:])
# with step-size for columns
print('Alternate values from first row:','\n',a[0:1,::2])
#
print('Second column values :','\n',a[:,1::2])
print('Arbitrary values :','\n',a[0:1,1:3])
First row values :
[[1 2 3]]
Alternate values from first row:
[[1 3]]
Second column values :
[[2]
[5]]
Arbitrary values :
[[2 3]]
```

Slicing a Three-Dimensional Array

We haven't looked at three-dimensional arrays yet, so let's see what one looks like:

```
a = np.array([[[1,2],[3,4],[5,6]],# first axis array
[[7,8],[9,10],[11,12]],# second axis array
[[13,14],[15,16],[17,18]]])# third axis array
# 3-D array
print(a)
[[[ 1  2]
  [ 3  4]
  [ 5  6]]

 [[ 7  8]
  [ 9 10]
  [11 12]]

 [[13 14]
  [15 16]
  [17 18]]]
```

Three-dimensional arrays don't just have rows and columns. They also have depth axes, which is where a two-dimensional array is stacked behind another one. When a three-dimensional array is sliced, you must specify the two-dimensional array you want to be sliced. This would typically be listed first in the index:

```
# value
print('First array, first row, first column value :','\n',a[0,0,0])
print('First array last column :','\n',a[0,:,1])
print('First two rows for second and third arrays :','\n',a[1:,0:2,0:2])
First array, first row, first column value :
1
First array last column :
```

```
[2 4 6]
```

First two rows for second and third arrays :

```
[[[ 7  8]
  [ 9 10]]


 [[13 14]
  [15 16]]]
```

If you wanted the values listed as a one-dimensional array, the flatten() method can be used:

```
print('Printing as a single array :','\n',a[1:,0:2,0:2].flatten())
```

Printing as a single array :

```
[ 7  8  9 10 13 14 15 16]
```

Negative Slicing

You can also use negative slicing on your array. This prints the elements from the end, instead of starting at the beginning:

```
a = np.array([[1,2,3,4,5],
[6,7,8,9,10]])
print(a[:,-1])
```

```
[ 5 10]
```

Each row's final values were printed, but if we wanted the values extracted from the end, a negative step-by-step would need to be provided. If not, you get an empty list.

```
print(a[:,-1:-3:-1])
```

```
[[ 5  4]
[10  9]]
```

That said, slicing logic is the same – you won't get the end index in the output.

You can also use negative slicing if you want the original array reversed:

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

```
[6,7,8,9,10]])
print('Original array :','\n',a)
print('Reversed array :','\n',a[::-1,::-1])
Original array :
[[ 1  2  3  4  5]
[ 6  7  8  9 10]]
Reversed array :
[[10  9  8  7  6]
[ 5  4  3  2  1]]
```

And a ndarray can be reversed using the flip() method:

```
a = np.array([[1,2,3,4,5],
[6,7,8,9,10]])
print('Original array :','\n',a)
print('Reversed array vertically :','\n',np.flip(a,axis=1))
print('Reversed array horizontally :','\n',np.flip(a,axis=0))
Original array :
[[ 1  2  3  4  5]
[ 6  7  8  9 10]]
Reversed array vertically :
[[ 5  4  3  2  1]
[10  9  8  7  6]]
Reversed array horizontally :
[[ 6  7  8  9 10]
[ 1  2  3  4  5]]
```

Stack and Concatenate NumPy Arrays

Existing arrays can be combined to create new arrays, and this can be done in two ways:

- Vertically combine arrays along the rows. This is done with the vstack() method and increases the number of rows in the array that gets returned.
- Horizontally combine arrays along the columns. This is done using the hstack() method and increases the number of columns in the array that gets returned.

```
a = np.arange(0,5)

b = np.arange(5,10)

print('Array 1 :','\n',a)

print('Array 2 :','\n',b)

print('Vertical stacking :','\n',np.vstack((a,b)))

print('Horizontal stacking :','\n',np.hstack((a,b)))
```

```
Array 1 :

[0 1 2 3 4]

Array 2 :

[5 6 7 8 9]

Vertical stacking :

[[0 1 2 3 4]

[5 6 7 8 9]]

Horizontal stacking :

[0 1 2 3 4 5 6 7 8 9]
```

Worth noting is that the axis along which the array is combined must have the same size. If it doesn’t, an error is thrown:

```
a = np.arange(0,5)

b = np.arange(5,9)

print('Array 1 :','\n',a)

print('Array 2 :','\n',b)

print('Vertical stacking :','\n',np.vstack((a,b)))

print('Horizontal stacking :','\n',np.hstack((a,b)))
```

The error message points to line 5 as where the error is and reads:

> ValueError: all the input array dimensions for the concatenation axis must match exactly, but along dimension 1, the array at index 0 has size 5 and the array at index 1 has size 4.

You can also use the dstack() method to combine arrays by combing the elements one index at a time and stacking them on the depth axis.

```
a = [[1,2],[3,4]]
b = [[5,6],[7,8]]
c = np.dstack((a,b))
print('Array 1 :','\n',a)
print('Array 2 :','\n',b)
print('Dstack :','\n',c)
print(c.shape)
```

```
Array 1 :
[[1, 2], [3, 4]]
Array 2 :
[[5, 6], [7, 8]]
Dstack :
[[[1 5]
  [2 6]]

 [[3 7]
  [4 8]]]
(2, 2, 2)
```

You can stack arrays to combine old ones into a new one, but you can also join passed arrays on an existing axis using the concatenate() method:

```
a = np.arange(0,5).reshape(1,5)
b = np.arange(5,10).reshape(1,5)
```

```
print('Array 1 :','\n',a)
print('Array 2 :','\n',b)
print('Concatenate along rows :','\n',np.concatenate((a,b),axis=0))
print('Concatenate along columns :','\n',np.concatenate((a,b),axis=1))
Array 1 :
[[0 1 2 3 4]]
Array 2 :
[[5 6 7 8 9]]
Concatenate along rows :
[[0 1 2 3 4]
[5 6 7 8 9]]
Concatenate along columns :
[[0 1 2 3 4 5 6 7 8 9]]
```

However, this method has one primary drawback – the original array must contain the axis along which you are combining, or else you will get an error.

The append() method can be used to add new elements at the end of a ndarray. This is quite useful if you want new values added to an existing ndarray:

```
# append values to ndarray
a = np.array([[1,2],
              [3,4]])
np.append(a,[[5,6]], axis=0)
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Broadcasting in NumPy Arrays

While ndarrays have many good features, one of the best is undoubtedly broadcasting. It allows mathematical operations between different sized ndarrays or a simple number and ndarray.

In essence, broadcasting stretches smaller ndarrays to match the larger one’s shape.

```
a = np.arange(10,20,2)

b = np.array([[2],[2]])

print('Adding two different size arrays :','\n',a+b)

print('Multiplying a ndarray and a number :',a*2)
```

Adding two different size arrays :

[[12 14 16 18 20]

[12 14 16 18 20]]

Multiplying a ndarray and a number : [20 24 28 32 36]

Think of it as being similar to making a copy of or stretching the number or scalar [2, 2, 2] to match the ndarray shape and then do an elementwise operation on it. However, no actual copies are made – it is just the best way to think of how broadcasting works.

Why is this so useful?

Because multiplying an array containing a scalar value is more efficient than multiplying another array.

NOTE

A pair of ndarrays must be compatible to broadcast together. This compatibility happens when:

- Both ndarrays have identical dimensions
- One ndarray has a dimension of 1, which is then broadcast to meet the larger one's size requirements

Should they not be compatible, an error is thrown:

```
a = np.ones((3,3))

b = np.array([2])

a+b

array([[3., 3., 3.],
       [3., 3., 3.],
```

[3., 3., 3.]])

In this case, we hypothetically stretched the second array to a shape of 3 x 3 before calculating the result.

NumPy Ufuncs

If you know anything about Python, you know it is dynamically typed. That means that Python doesn't need to know its data type at the time a variable is assigned because it will determine it automatically at runtime. However, while this ensures a cleaner code, it does slow things down a bit.

This tends to worsen when Python needs to do loads of operations over and again, such as adding two arrays. Python must check each element's data type whenever an operation has to be performed, slowing things down, but we can get over this by using ufuncs.

NumPy speeds things up with something called vectorization. On a ndarray in compiled code, vectorization will perform one operation on each element in turn. In this way, there is no need to determine the elements' data types each time, thus making things faster.

A ufunc is a universal function, nothing more than a mathematical function that performs high-speed element-by-element operations. When you do simple math operations on an array, the ufunc is automatically called because the arrays are ufuncs wrappers.

For example, when you use the + operator to add NumPy arrays, a ufunc called add() is called automatically under the hood and does what it has to do quietly. This is how a Python list would look:

```
a = [1,2,3,4,5]

b = [6,7,8,9,10]

%timeit a+b
```

While this is the same operation as a NumPy array:

```
a = np.arange(1,6)

b = np.arange(6,11)

%timeit a+b
```

As you can see, using ufuncs, the array addition takes much less time.

Doing Math with NumPy Arrays

Math operations are common in Python, never more so than in NumPy arrays, and these are some of the more useful ones you will perform:

Basic Arithmetic

Those with Python experience will know all about arithmetic operations and how easy they are to perform. And they are just as easy to do on NumPy arrays. All you really need to remember is that the operation symbols play the role of ufuncs wrappers:

```
print('Subtract :',a-5)
print('Multiply :',a*5)
print('Divide :',a/5)
print('Power :',a**2)
print('Remainder :',a%5)
```

```
Subtract : [-4 -3 -2 -1 0]
Multiply : [ 5 10 15 20 25]
Divide : [0.2 0.4 0.6 0.8 1. ]
Power : [ 1 4 9 16 25]
Remainder : [1 2 3 4 0]
```

Mean, Median and Standard Deviation

Finding the mean, median and standard deviation on NumPy arrays requires the use of three methods – mean(), median(), and std():

```
a = np.arange(5,15,2)
print('Mean :',np.mean(a))
print('Standard deviation :',np.std(a))
print('Median :',np.median(a))
```

```
Mean : 9.0
Standard deviation : 2.8284271247461903
Median : 9.0
```

Min-Max Values and Their Indexes

In a ndarray, we can use the min() and max() methods to find the min and max values:

```
a = np.array([[1,6],
[4,3]])
# minimum along a column
print('Min :',np.min(a,axis=0))
# maximum along a row
print('Max :',np.max(a,axis=1))
Min : [1 3]
Max : [6 4]
```

You can also use the argmin() and argmax() methods to get the minimum or maximum value indexes along a specified axis:

```
a = np.array([[1,6,5],
[4,3,7]])
# minimum along a column
print('Min :',np.argmin(a,axis=0))
# maximum along a row
print('Max :',np.argmax(a,axis=1))
Min : [0 1 0]
Max : [1 2]
```

Breaking down the output, the first column's minimum value is the column's first element. The second column's minimum is the second element, while it's the first element for the third column.

Similarly, the outputs can be determined for the maximum values.

The Sorting Operation

Any good programmer will tell you that the most important thing to consider is an algorithm's time complexity. One basic yet important operation you are likely

to use daily as a data scientist is sorting. For that reason, a good sorting algorithm must be used, and it must have the minimum time complexity.

The NumPy library is top of the pile when it comes to sorting an array's elements, with a decent range of sorting functions on offer. And when you use the sort() method, you can also use mergesort, heapsort, and timesort, all available in NumPy, under the hood.

```
a = np.array([1,4,2,5,3,6,8,7,9])
np.sort(a, kind='quicksort')
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

It is also possible to sort arrays on any axis you want:

```
a = np.array([[5,6,7,4],
        [9,2,3,7]])# sort along the column
print('Sort along column :','\n',np.sort(a, kind='mergresort',axis=1))
# sort along the row
print('Sort along row :','\n',np.sort(a, kind='mergresort',axis=0))
Sort along column :
[[4 5 6 7]
[2 3 7 9]]
Sort along row :
[[5 2 3 4]
[9 6 7 7]]
```

NumPy Arrays and Images

NumPy arrays are widely used for storing image data and manipulating it, but what is image data?

Every image consists of pixels, and these are stored in arrays. Every pixel has a value of somewhere between 0 and 255, with 0 being a black pixel and 255 being a white pixel. Colored images comprise three two-dimensional arrays for the three color channels (RGB – Red, Green, Blue. These are back-to-back and make up a three-dimensional array. Each of the array's values stands for a pixel value so, the array size is dependent on how many pixels are on each dimension.

Python uses the scipy.misc.imread() method from the SciPy library to read images as arrays. The output is a three-dimensional array comprised of the pixel values:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import misc

# read image
im = misc.imread('./original.jpg')
# image
im
array([[[115, 106,  67],
        [113, 104,  65],
        [112, 103,  64],
        ...,
        [160, 138,  37],
        [160, 138,  37],
        [160, 138,  37]],

       [[117, 108,  69],
        [115, 106,  67],
        [114, 105,  66],
        ...,
        [157, 135,  36],
        [157, 135,  34],
        [158, 136,  37]],
```

```
[[120, 110,  74],
 [118, 108,  72],
 [117, 107,  71],
 ...,
```

Checking the type and shape of the array is done like this:

```
print(im.shape)
print(type(type))
(561, 997, 3)
numpy.ndarray
```

Because images are nothing more than arrays, we can use array functions to manipulate them. For example, the image could be horizontally flipped using a method called np.flip():

```
# flip
plt.imshow(np.flip(im, axis=1))
```

Or the pixels value range could be changed or normalized, especially if you need faster computation:

```
im/255
array([[[0.45098039, 0.41568627, 0.2627451 ],
        [0.44313725, 0.40784314, 0.25490196],
        [0.43921569, 0.40392157, 0.25098039],
        ...,
        [0.62745098, 0.54117647, 0.14509804],
        [0.62745098, 0.54117647, 0.14509804],
        [0.62745098, 0.54117647, 0.14509804]],

       [[0.45882353, 0.42352941, 0.27058824],
        [0.45098039, 0.41568627, 0.2627451 ],
        [0.44705882, 0.41176471, 0.25882353],
```

```
       ...,
       [0.61568627, 0.52941176, 0.14117647],
       [0.61568627, 0.52941176, 0.13333333],
       [0.61960784, 0.53333333, 0.14509804]],

      [[0.47058824, 0.43137255, 0.29019608],
       [0.4627451 , 0.42352941, 0.28235294],
       [0.45882353, 0.41960784, 0.27843137],
       ...,
       [0.6       , 0.52156863, 0.14117647],
       [0.6       , 0.52156863, 0.13333333],
       [0.6       , 0.52156863, 0.14117647]],

      ...,
```

That is as much of an introduction as I can give you to NumPy, showing you the basic methods and operations you are likely to use as a data scientist. Next, we discover how to manipulate data using Pandas.

Part Three – Data Manipulation with Pandas

Pandas is another incredibly popular Python data science package, and there is a good reason for that. It offers users flexible, expressive, and powerful data structures that make data analysis and manipulation dead simple. One of those structures is the DataFrame.

In this part, we will look at Pandas DataFrames, including fundamental manipulation, up to the more advanced operations. To do that, we will answer the top 11 questions asked to provide you with the answers you need.

What Is a DataFrame?

First, let's look at what a DataFrame is.

If you have any experience using the R language, you will already know that a data frame provides rectangular grids to store data so it can be looked at easily. The rows in the grids all correspond to a measurement or an instance value, and the columns are vectors with data relating to a specific variable. So, there is no need for the rows to contain identical value types, although they can if needed. They can store any type of value, such as logical, character, numeric, etc.

Python DataFrames are much the same. They are included in the Pandas library, defined as two-dimensional data structures, labeled, and with columns containing possible different types.

It would be fair to say that there are three main components in a Pandas DataFrames – data, index, and columns.

First, we can store the following type of data in a DataFrame:

- A Pandas DataFrame
- A Pandas Series, a labeled one-dimensional array that can hold any data type with an index or axis label. A simple example would a DataFrame column
- A NumPy ndarray, which could be a structure or record
- A two-dimensional ndarray
- A dictionary containing lists, Series, dictionaries or one-dimensional ndarrays

NOTE

Do not confuse np.ndarray with np.array(). The first is a data type, and the second is a function that uses other data structures to make arrays.

With a structured array, a user can manipulate data via fields, each named differently. The example below shows a structured array being created containing three tuples. Each tuple's first element is called foo and is the int type. The second element is a float called bar.

The same example shows a record array. These are used to expand the properties of a structured array, and users can access the fields by attribute and not index. As you can see in the example, we access the foo values in the record array called r2:

```
# A structured array
my_array = np.ones(3, dtype=([('foo', int), ('bar', float
)]))
# Print the structured array
_____(my_array['foo'])
# A record array
my_array2 = my_array.view(np.recarray)
# Print the record array
_____(my_array2.foo)
```

Second, the index and column names can also be specified for the DataFrame. The index provides the row differences, while the column name indicates the column differences. Later, you will see that these are handy DataFrame components for data manipulation.

If you haven't already got Pandas or NumPy on your system, you can import them with the following import command:

```
import numpy as np
import pandas as pd
```

Now you know what DataFrames are and what you can use them for, let's learn all about them by answering those top 11 questions.

Question One: How Do I Create a Pandas DataFrame?

The first step when it comes to data munging or manipulation is to create your DataFrame, and you can do this in two ways – convert an existing one or create

a brand new one. We'll only be looking at converting existing structures in this part, but we will discuss creating new ones later on.

NumPy ndarrays are just one thing that you can use as a DataFrame input. Creating a DataFrame from a NumPy array is pretty simple. All you need to do is pass the array to the DataFrame() function inside the data argument:

```
data = np.array([['','Col1','Col2'],
                 ['Row1',1,2],
                 ['Row2',3,4]])


print(pd.DataFrame(data=data[1:,1:],
                   index=data[1:,0],
                   columns=data[0,1:]))
```

One thing you need to pay attention to is how the DataFrame is constructed from NumPy array elements. First, the values from lists starting with Row1 and Row2 are selected. Then the row numbers or index, Row1 or Row2, are selected, followed by the column names of Col1 and Col2.

Next, we print a small data selection. As you can see, this is much the same as when you subset and two-dimensional NumPy array – the row is indicated first, where you want your data looked for, followed by the column. Don't forget that, in Python, indices begin at 0. In the above example, we look in the rows in index 1 to the end and choose all elements after index 1. The result is 1, 2, 3, and 4 being selected.

This is the same approach to creating a DataFrame as for any structure taken as an input by DataFrame().

Try it for yourself on the next example:

```
# Take a 2D array as input to your DataFrame
my_2darray = np.array([[1, 2, 3], [4, 5, 6]])
print(_____________________)
# Take a dictionary as input to your DataFrame
my_dict = {1: ['1', '3'], 2: ['1', '2'], 3: ['2', '4']}
```

```
print(_____)
# Take a DataFrame as input to your DataFrame
my_df = pd.DataFrame(data=[4,5,6,7], index=range(0,4),
columns=['A'])
print(_____)
# Take a Series as input to your DataFrame
my_series = pd.Series({"Belgium":"Brussels", "India":"New
Delhi", "United Kingdom":"London", "United States"
:"Washington"})
print(_____)
```

Your Series and DataFrame index has the original dictionary keys, but in a sorted manner – index 0 is Belgium, while index 3 is the United States. Once your DataFrame is created, you can find out some things about it. For example, the len() function or shape property can be used with the .index property:

```
df = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6]]))
# Use the `shape` property
print(_____)
# Or use the `len()` function with the `index` property
print(_____)
```

These options provide differing DataFrame information. We get the DataFrame dimensions using the shape property, which means the height and width of the DataFrame. And, if you use the index property and len() function together, you will only get the DataFrame height.

The df[0].count() function could also be used to find out more information about the DataFrame height but, should there be any NaN values, this won't provide them, which is why .count() is not always the best option.

Okay, now we know how to create a DataFrame from an existing structure, it's time to get down to the real work and start looking at some of the basic operations.

Question Two – How Do I Select a Column or Index from a

DataFrame?

Before we can begin with basic operations, such as deleting, adding, or renaming, we need to learn how to select the elements. It isn't difficult to select a value, column, or index from a DataFrame. In fact, it's much the same as it is in other data analysis languages. Here's an example of the values in the DataFrame:

```
   A  B  C
0  1  2  3
1  4  5  6
2  7  8  9
```

We want to get access to the value in column A at index 0 and we have several options to do that:

```
# Using `iloc[]`
print(df.iloc[0][0])
# Using `loc[]`
print(df.loc[0]['A'])
# Using `at[]`
print(df.at[0,'A'])
# Using `iat[]`
print(df.iat[0,0])
```

There are two of these that you must remember - .iloc[] and .loc[]. There are some differences between these, but we'll discuss those shortly.

Now let's look at how to select some values. If you wanted rows and columns, you would do this:

```
# Use `iloc[]` to select row `0`
print(df.iloc[_])
# Use `loc[]` to select column `'A'`
print(df.loc[:,'_'])
```

Right now, it's enough for you to know that values can be accessed by using their label name or their index or column position.

Question Three: How Do I Add a Row, Column, or Index to a DataFrame?

Now you know how to select values, it's time to understand how a row, column, or index is added.

Adding an Index

When a DataFrame is created, you want to ensure that you get the right index so you can add some input to the index argument. If this is not specified, by default, the DataFrame's index will be of numerical value and will start with 0, running until the last row in the DataFrame.

However, even if your index is automatically specified, you can still re-use a column, turning it into your index. We call set_index() on the DataFrame to do this:

```
# Print out your DataFrame `df` to check it out

print(__)

# Set 'C' as the index of your DataFrame

df._____('C')
```

Adding a Row

Before reaching the solution, we should first look at the concept of loc and how different it is from .iloc, .ix, and other indexing attributes:

- **.loc[] –** works on your index labels. For example, if you have loc[2], you are looking for DataFrame values with an index labeled 2.
- **.iloc[] –** works on your index positions. For example, if you have iloc[2], you are looking for DataFrame values with an index labeled 2.
- **.ix[] –** this is a little more complex. If you have an integer-based index, a label is passed to .ix[]. In that case, .ix[2] means you are looking for DataFrame values with an index labeled 2. This is similar to .loc[] but, where you don't have an index that is purely integer-bases, .ix also works with positions.

Let’s make this a little simpler with the help of an example:

```
df = pd.DataFrame(data=np.array([[1, 2, 3], [4, 5, 6], [7

, 8, 9]]), index= [2, 'A', 4], columns=[48, 49, 50])
```

```
# Pass `2` to `loc`
print(df.loc[_])
# Pass `2` to `iloc`
print(df.iloc[_])
# Pass `2` to `ix`
print(df.ix[_])
```

Note that we used a DataFrame not based solely on integers, just to show you the differences between them. As you can clearly see, you don't get the same result when you pass 2 to .loc[] or .iloc[]/.ix[].

.loc[] will examine those values labeled 2, and you might get something like the following returned:

```
48  1
49  2
50  3
```

.iloc[] looks at the index position, and passing 2 will give you something like this:

```
48  7
49  8
50  9
```

Because we don't just have integers in the index, .ix[] behaves the same way as .iloc[] and examines the index positions. In that case, you get the same results that .iloc[] returned.

Understanding the difference between those three is a critical part of adding rows to a DataFrame. You should also have realized that the recommendation is to use .loc[] when you insert rows. If you were to use df.ix[], you might find you are referencing numerically indexed values, and you could overwrite an existing DataFrame row, all be it accidentally.

Once again, have a look at the example below"

```
df = pd._____(data=np.array([[1, 2, 3], [4, 5, 6], [7
, 8, 9]]), index= [2.5, 12.6, 4.8], columns=[48, 49, 50])
```

```
# There's no index labeled `2`, so you will change the
index at position `2`
df.ix[2] = [60, 50, 40]
print(df)
# This will make an index labeled `2` and add the new
values
df.loc[2] = [11, 12, 13]
print(df)
```

It’s easy to see how this is all confusing.

Adding a Column

Sometimes, you may want your index to be a part of a DataFrame, and this can be done by assigning a DataFrame column or referencing and assigning one that you haven't created yet. This is done like this:

```
df = pd._____(data=np.array([[1, 2, 3], [4, 5, 6], [7,
8, 9]]), columns=['A', 'B', 'C'])
# Use `.index`
df['D'] = df.index
# Print `df`
print(__)
```

What you are doing here is telling the DataFrame that the index should be column A.

However, appending columns could be done in the same way as adding an index - .loc[] or .iloc[] is used but, this time, a Series is added to the DataFrame using .loc[]:

```
# Study the DataFrame `df`
print(__)
# Append a column to `df`
df.loc[:, 4] = pd.Series(['5', '6'], index=df.index)
```

```
# Print out `df` again to see the changes
_____(__)
```

Don't forget that a Series object is similar to a DataFrame's column, which explains why it is easy to add them to existing DataFrames. Also, note that the previous observation we made about .loc[] remains valid, even when adding columns.

Resetting the Index

If your index doesn't look quite how you want it, you can reset it using reset_index(). However, be aware when you are doing this because it is possible to pass arguments that can break your reset.

```
# Check out the strange index of your DataFrame
print(df)
# Use `reset_index()` to reset the values.
df_reset = df._____(level=0, drop=True)
# Print `df_reset`
print(df_reset)
```

Use the code above but replace drop with inplace and see what would happen.

The drop argument is useful for indicating that you want the existing index eliminated, but if you use inplace, the original index will be added, with floats, as an additional column.

Question Four: How Do I Delete Indices, Rows, or Columns From a Data Frame?

Now you know how to add rows, columns, and indices, let's look at how to remove them from the data structure.

Deleting an Index

Removing an index from a DataFrame isn't always a wise idea because Series and DataFrames should always have one. However, you can try these instead:

- Reset your DataFrame index, as we did earlier
- If your index has a name, use del df.index.name to remove it

- Remove any duplicate values – to do this, reset the index, drop the index column duplicates, and reinstate the column with no duplicates as the index:

  df = pd.DataFrame(data=np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [40, 50, 60], [23, 35, 37]]),

  index= [2.5, 12.6, 4.8, 4.8, 2.5],

  columns=[48, 49, 50])

  df._____.drop_duplicates(subset='index', keep='last').set_index('index')

- Finally, you can remove an index with a row – more about this later.

Deleting a Column

To remove one or more columns, the drop() method is used:

```
# Check out the DataFrame `df`
print(__)
# Drop the column with label 'A'
df.____('A', axis=1, inplace=True)
# Drop the column at position 1
df.____(df.columns[[1]], axis=1)
```

This might look too straightforward, and that's because the drop() method has got some extra arguments:

- When the axis argument indicates rows, it is 0, and when it drops columns, it is 1
- Inplace can be set to true, thus deleting the column without needing the DataFrame to be reassigned.

Removing a Row

You can use df.drop_duplicates() to remove duplicate rows but you can also do it by considering only the column’s values:

```
# Check out your DataFrame `df`
```

```
print(__)
# Drop the duplicates in `df`
df._____([48], keep='last')
```

If you don’t need to add any unique criteria to your deletion, the drop() method is ideal, using the index property for specifying the index for the row you want removed:

```
# Check out the DataFrame `df`
print(__)
# Drop the index at position 1
df.____(df.index[_])
```

After this, you can reset your index.

Question Five: How Do I Rename the Columns or Index of a DataFrame?

If you want to change the index or column values in your DataFrame different values, you should use a method called .rename():

```
# Check out your DataFrame `df`
print(__)
# Define the new names of your columns
newcols = {
    'A': 'new_column_1',
    'B': 'new_column_2',
    'C': 'new_column_3'
}
# Use `rename()` to rename your columns
df._____(columns=newcols, inplace=True)
# Use `rename()` to your index
df._____(index={1: 'a'})
```

If you were to change the inplace argument value to False, you would see that

the DataFrame didn't get reassigned when the columns were renamed. This would result in the second part of the example taking an input of the original argument rather than the one returned from the rename() operation in the first part.

Question Six: How Do I Format the DataFrame Data?

On occasion, you may also want to do operations on the DataFrame values. We're going to look at a few of the most important methods to formatting DataFrame values.

Replacing All String Occurrences

The replace() method can easily be used when you want to replace specific strings in the DataFrame. All you have to do is pass in the values you want to be changed and the new, replacement values, like this:

```
# Study the DataFrame `df` first
_____(df)
# Replace the strings by numerical values (0-4)
df._______(['Awful', 'Poor', 'OK', 'Acceptable',
'Perfect'], [0, 1, 2, 3, 4])
```

You can also use the regex argument to help you out with odd combinations of strings:

```
# Check out your DataFrame `df`
print(df)
# Replace strings by others with `regex`
df.replace({'\n': '<br>'}, regex=True)
```

Put simply, replace() is the best method to deal with replacing strings or values in the DataFrame.

Removing Bits of Strings from DataFrame Cells

It isn't easy to remove bits of strings and it can be a somewhat cumbersome task. However, there is a relatively easy solution:

```
# Check out your DataFrame
_____(df)
```

```
# Delete unwanted parts from the strings in the `result`
column
df['result'] = df['result'].map(lambda x: x.lstrip('+-'
).rstrip('aAbBcC'))
# Check out the result again
df
```

To apply the lambda function element-wise or over individual elements in the column, we use map() on the column's result. The map() function will take the string value and removes the + or – on the left and one or more of the aABbcC you see on the right.

Splitting Column Text into Several Rows

This is a bit more difficult but the example below walks you through what needs to be done:

```
# Inspect your DataFrame `df`
print(__)
# Split out the two values in the third row
# Make it a Series
# Stack the values
ticket_series = df['Ticket'].str.split(' ').apply(pd
.Series, 1).stack()
# Get rid of the stack:
# Drop the level to line up with the DataFrame
ticket_series.index = ticket_series.index.droplevel(-1)
# Make your `ticket_series` a dataframe
ticketdf = pd._____(ticket_series)
# Delete the `Ticket` column from your DataFrame
del df['Ticket']
# Join the `ticketdf` DataFrame to `df`
```

```
df.____(ticketdef)

# Check out the new `df`

df
```

What you are doing is:

- Inspecting the DataFrame. The values in the last column and the last row are long. It looks like we have two tickets because one guest took a partner with them to the concert.
- The Ticket column is removed from the DataFrame df and the strings on a space. This ensures that both tickets are in separate rows. These four values, which are the ticket numbers, are placed into a Series object:

```
     0         1
0  23:44:55       NaN
1  66:77:88       NaN
2  43:68:05  56:34:12
```

- Something still isn't right because we can see NaN values. The Series must be stacked to avoid NaN values in the result.
- Next, we see the stacked Series:

```
0  0    23:44:55
1  0    66:77:88
2  0    43:68:05
   1    56:34:12
```

- That doesn't really work, either, and this why the level is dropped, so it lines up with the DataFrame:

```
0   23:44:55
1    66:77:88
2    43:68:05
2    56:34:12
dtype: object
```

- Now that is what you really want to see.

- Next, your Series is transformed into a DataFrame so it can be rejoined to the initial DataFrame. However, we don't want duplicates, so the original Ticket column must be deleted.

Applying Functions to Rows or Columns

You can apply functions to data in the DataFrame to change it. First, we make a lambda function:

```
doubler = lambda x: x*2
```

Next, we apply the function:

```
# Study the `df` DataFrame

_____(__)

# Apply the `doubler` function to the `A` DataFrame column

df['A'].apply(doubler)
```

Note the DataFrame row can also be selected and the doubler lambda function applied to it. You know how to select rows – by using .loc[] or .iloc[].

Next, something like the following would be executed, depending on whether your index is selected based on position or label:

```
df.loc[0].apply(doubler)
```

Here, the apply() function is only relevant to the doubler function on the DataFrame axis. This means you target the columns or the index – a row or a column in simple terms.

However, if you wanted it applied element-wise, the map() function can be used. Simply replace apply() with map(), and don't forget that the doubler still has to be passed to map() to ensure the values are multiplied by 2.

Let's assume this doubling function is to be applied to the entire DataFrame, not just the A column. In that case, the doubler function is applied used applymap() to every element in your DataFrame:

```
doubled_df = df.applymap(doubler)

print(doubled_df)
```

Here, we've been working with anonymous functions create at runtime or lambda functions but you can create your own:

```
def doubler(x):
    if x % 2 == 0:
        return x
    else:
        return x * 2
# Use `applymap()` to apply `doubler()` to your DataFrame
doubled_df = df.applymap(doubler)
# Check the DataFrame
print(doubled_df)
```

Question Seven: How Do I Create an Empty DataFrame?

We will use the DataFrame() function for this, passing in the data we want to be included, along with the columns and indices. Remember, you don't have to use homogenous data for this – it can be of any type.

There are a few ways to use the DataFrame() function to create empty DataFrames. First, numpy.nan can be used to initialize it with NaNs. The numpy.nan function has a float data type:

```
df = pd.DataFrame(np.nan, index=[0,1,2,3], columns=['A'])
print(df)
```

At the moment, this DataFrame's data type is inferred. This is the default and, because numpy.nan has the float data type, that is the type of values the DataFrame will contain. However, the DataFrame can also be forced to be a specific type by using the dtype attribute and supplying the type you want, like this:

```
df = pd.DataFrame(index=range(0,4),columns=['A'], dtype
='float')
print(df)
```

If the index or axis labels are not specified, the input data is used to construct them, using common sense rules.

Question Eight: When I Import Data, Will Pandas Recognize

Dates?

Yes, Pandas can recognize dates but only with a bit of help. When you read the data in, for example, from a CSV file, the parse_dates argument should be added:

```
import pandas as pd
pd.read_csv('yourFile', parse_dates=True)


# or this option:
pd.read_csv('yourFile', parse_dates=['columnName'])
```

However, there is always the chance that you will come across odd date-time formats. There is nothing to worry about as it's simple enough to construct a parser to handle this. For example, you could create a lambda function to take the DateTime and use a format string to control it:

```
import pandas as pd
dateparser = lambda x: pd.datetime.strptime(x, '%Y-%m-%d
%H:%M:%S')


# Which makes your read command:
pd.read_csv(infile, parse_dates=['columnName'], date_parser=dateparse)


# Or combine two columns into a single DateTime column
pd.read_csv(infile, parse_dates={'datetime': ['date', 'time']},
date_parser=dateparse)
```

Question Nine: When Should a DataFrame Be Reshaped? Why and How?

When you reshape a DataFrame, you transform it to ensure that the structure you get better fits your data analysis. We can infer from this that reshaping is less concerned with formatting the DataFrame values and more concerned with transforming the DataFrame shape.

So that tells you when and why you should reshape, but how do you do it?

You have a choice of three ways, and we'll look at each of them in turn:

Pivoting

The pivot() function is used for creating a derived table from the original. Three arguments can be passed with the function:

- **values –** lets you specify the values from the original DataFrame to bring into your pivot table.
- **columns –** whatever this argument is passed to becomes a column in the table.
- **index –** whatever this argument is passed to becomes an index.

```
# Import pandas
import _____ as pd
products = pd.DataFrame({'category': ['Cleaning',
'Cleaning', 'Entertainment', 'Entertainment', 'Tech',
'Tech'],
                'store': ['Walmart', 'Dia',
'Walmart', 'Fnac', 'Dia','Walmart'],
                'price':[11.42, 23.50, 19.99, 15
.95, 55.75, 111.55],
                'testscore': [4, 3, 5, 7, 5, 8]})
# Use `pivot()` to pivot the DataFrame
pivot_products = products._____(index='category', columns
='store', values='price')
# Check out the result
print(pivot_products)
```

If you don’t specify the values you want on the table, you will end up pivoting by several columns:

```
# Import the Pandas library
import _____ as pd
```

```
# Construct the DataFrame
products = pd.DataFrame({'category': ['Cleaning',
'Cleaning', 'Entertainment', 'Entertainment', 'Tech',
'Tech'],
                'store': ['Walmart', 'Dia',
'Walmart', 'Fnac', 'Dia','Walmart'],
                'price':[11.42, 23.50, 19.99, 15
.95, 55.75, 111.55],
                'testscore': [4, 3, 5, 7, 5, 8]})
# Use `pivot()` to pivot your DataFrame
pivot_products = products._____(index='category', columns
='store')
# Check out the results
print(pivot_products)
```

The data may not have any rows containing duplicate values for the specified columns. If they do, an error message is thrown. If your data cannot be unique, you must use a different method, pivot_table():

```
# Import the Pandas library
import ______ as pd
# Your DataFrame
products = pd.DataFrame({'category': ['Cleaning',
'Cleaning', 'Entertainment', 'Entertainment', 'Tech',
'Tech'],
                'store': ['Walmart', 'Dia',
'Walmart', 'Fnac', 'Dia','Walmart'],
                'price':[11.42, 23.50, 19.99, 15
.95, 19.99, 111.55],
```

```
                    'testscore': [4, 3, 5, 7, 5, 8]})
# Pivot your `products` DataFrame with `pivot_table()`
pivot_products = products._____(index='category',
columns='store', values='price', aggfunc='mean')
# Check out the results
print(pivot_products)
```

We passed an extra argument to the pivot_table method, called aggfunc. This indicates that multiple values are combined with an aggregation function. In our example, we used the mean function.

Reshaping a DataFrame Using Stack() and Unstack()

If you refer back to question five, you will see we already used a stacking example. When a DataFrame is stacked, it is made taller. The innermost column index is moved to become the innermost row index, thus returning a DataFrame with an index containing a new innermost row labels level.

Conversely, we can use unstack() to make the innermost row index into the innermost column index.

Using melt() to Reshape a DataFrame

Melting is useful when your data has at least one column that is an identifier variable, and all others are measured variables.

Measured variables are considered as unpivoted to the row axis. This means that, where the measured variables were scattered over the DataFrame's width, melt() will ensure they are placed to its height. So, instead of becoming wider, the DataFrame becomes longer. This results in a pair of non-identifier columns called value and variable. Here's an example to show you what this means:

```
# The `people` DataFrame
people = pd.DataFrame({'FirstName' : ['John', 'Jane'],
                       'LastName' : ['Doe', 'Austen'],
                       'BloodType' : ['A-', 'B+'],
                       'Weight' : [90, 64]})
# Use `melt()` on the `people` DataFrame
```

```
print(pd.____(people, id_vars=['FirstName', 'LastName'],
var_name='measurements'))
```

Question Ten: How Do I Iterate Over a DataFrame?

Iterating over the DataFrame rows is done with a for loop and calling iterrows() on the DataFrame:

```
df = pd.DataFrame(data=np.array([[1, 2, 3], [4, 5, 6], [7,
8, 9]]), columns=['A', 'B', 'C'])
for index, row in df.iterrows() :
    print(row['A'], row['B'])
```

Using iterrows() lets you loop efficiently over the rows as pairs – index, Series. In simple terms, the result will be (index, row) tuples.

Question Eleven: How Do I Write a DataFrame to a File?

When you have finished doing all you want to do, you will probably want to export your DataFrame into another format. We'll look at two ways to do this – Excel or CSV file.

Outputting to a CSV File

Writing to a CSV file requires the use of to_csv():

```
import pandas as pd
df.to_csv('myDataFrame.csv')
```

That looks like a simple code but, for many people, it's where things get difficult. This is because everyone will have their own requirements for the data output. You may not want to use commas as the delimiter or want a specific type of encoding.

We can get around this with specific arguments passed to to_csv() to ensure you get the output in the format you want:

- Delimiting a tab requires the sep argument:

  ```
  import pandas as pd
  df.to_csv('myDataFrame.csv', sep='\t')
  ```

- The encoding argument allows you to choose the character encoding you

want:

```
import pandas as pd

df.to_csv('myDataFrame.csv', sep='\t', encoding='utf-8')
```

- You can even specify how to represent missing or NaN values – you may or may not want the header outputted, you may or may not want the row names written, you may want compression, and so on. You can find out how to do all that by reading this page .

Outputting to an Excel File

For this one, you use to_excel() but it isn't quite as straightforward as the CSV file:

```
import pandas as pd

writer = pd.ExcelWriter('myDataFrame.xlsx')

df.to_excel(writer, 'DataFrame')

writer.save()
```

However, it's worth noting that you require many additional arguments, just like you do with to_csv, to ensure your data is output as you want. These include startrow, startcol, and more, and you can find out about them on this page .

More than Just DataFrames

While that was a pretty basic Pandas tutorial, it gives you a good idea of using it. Those 11 questions we answered were the most commonly asked ones, and they represent the fundamental skills you need to import your data, clean it and manipulate it.

In the next chapter, we'll take a brief look at data visualization with Matplotlib and Seaborn.

Part Four – Data Visualization with Matplotlib and Seaborn

Data visualization is one of the most important parts of communicating your results to others. It doesn't matter whether it is in the form of pie charts, scatter plots, bar charts, histograms, or one of the many other forms of visualization; getting it right is key to unlocking useful insights from the data. Thankfully, data visualization is easy with Python.

It's fair to say that data visualization is a key part of analytical tasks, like exploratory data analysis, data summarization, model output analysis, and more. Python offers plenty of libraries and tools to help us gain good insights from data, and the most commonly used is Matplotlib. This library enables us to generate all different visualizations, all from the same code if you want.

Another useful library is Seaborn, built on Matplotlib, providing highly aesthetic data visualizations that are sophisticated, statistically speaking. Understanding these libraries is critical for data scientists to make the most out of their data analysis.

Using Matplotlib to Generate Histograms

Matplotlib is packed with tools that help provide quick data visualization. For example, researchers who want to analyze new data sets often want to look at how values are distributed over a set of columns, and this is best done using a histogram.

A histogram generates approximations to distributions by using a set range to select results, placing each value set in a bucket or bin. Using Matplotlib makes this kind of visualization straightforward.

We'll be using the FIFA19 dataset, which you can download from here . Run these codes to see the outputs for yourself.

To start, if you haven't already done so, import Pandas:

```
import pandas as pd
```

Next, the pyplot module from Matplotlib is required, and the customary way of importing it is as plt:

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

Next, the data needs to be read into a DataFrame. We'll use the set_option() method from Pandas to relax the display limit of rows and columns:

```
df = pd.read_csv("fifa19.csv")


pd.set_option('display.max_columns', None)
```

```
pd.set_option('display.max_rows', None)
```

Now we can print the top five rows of the data, and this is done with the head() method:

```
print(df.head())
```

A histogram can be generated for any numerical column. The hist() method is called on the plt object, and the selected DataFrame column is passed in. We'll try this on the Overall column because it corresponds to the overall player rating:

```
plt.hist(df['Overall'])
```

The a and y axes can also be labeled, along with the plot title, using three methods:

- xlabel()
- ylabel()
- title()

```
plt.xlabel('Overall')


plt.ylabel('Frequency')


plt.title('Histogram of Overall Rating')


plt.show()
```

This histogram is one of the best ways to see the distribution of values and see which ones occur the most and the least.

Using Matplotlib to Generate Scatter Plots

Scatterplots are another useful visualization that shows variable dependence. For example, if we wanted to see if a positive relationship existed between wage and overall player rating (if the wage increases, does the rating go up?), we can use the scatterplot to find it.

Before generating the scatterplot, we need the wage column converted from string to floating-point, which is a numerical column. To do this, a new column is created, named wage_euro:

```
df['wage_euro'] = df['Wage'].str.strip('€')

df['wage_euro'] = df['wage_euro'].str.strip('K')

df['wage_euro'] = df['wage_euro'].astype(float)*1000.0
```

Now, let’s display our new column wage_euro and the overall column:

```
print(df[['Overall', 'wage_euro']].head())
```

The result would look like this:

```
  Overall wage_euro
0 94   565000.0
1 94   405000.0
2 92   290000.0
3 91   260000.0
4 91   355000.0
```

Generating a Matplotlib scatter plot is as simple as using scatter() on the plt object. We can also label each axis and provide the plot with a title:

```
plt.scatter(df['Overall'], df['wage_euro'])

plt.title('Overall vs. Wage')

plt.ylabel('Wage')

plt.xlabel('Overall')

plt.show()
```

Using Matplotlib to Generate Bar Charts

Another good visualization tool is the bar chart, useful for analyzing data

categories. For example, using our FIFA19 data set, we might want to look at the most common nationalities, and bar charts can help us with this. Visualizing categorical columns requires the values to be counted first. We can generate a dictionary of count values for every one of the categorical column categories using the count method. We'll do that for the nationality column:

```
from collections import Counter


print(Counter(df[‘Nationality’]))
```

The result would be a long list of nationalities, starting with the most values and ending with the least.

This dictionary can be filtered using a method called most_common. We'll ask for the ten most common nationalities in this case but, if you wanted to see the least common, you could use the least_common method:

```
print(dict(Counter(df[‘Nationality’]).most_common(10)))
```

Again, the result would be a dictionary containing the ten most common nationalities in order of the most values to the least.

Finally, the bar chart can be generated by calling the bar method on the plt object, passing the dictionary keys and values in:

```
nationality_dict = dict(Counter(df['Nationality']).most_common(10))


plt.bar(nationality_dict.keys(), nationality_dict.values())


plt.xlabel('Nationality')


plt.ylabel('Frequency')


plt.title('Bar Plot of Ten Most Common Nationalities')


plt.xticks(rotation=90)
```

```
plt.show()
```

You can see from the resulting bar chart that the x-axis values overlap, making them difficult to see. If we want to rotate the values, we use the xticks() method, like this:

```
plt.xticks(rotation=90)
```

Using Matplotlib to Generate Pie Charts

Pie charts are an excellent way of visualizing proportions in the data. Using our FIFA19 dataset, we could visualize the player proportions for three countries – England, Germany, and Spain – using a pie chart.

First, we create four new columns, one each for England, Germany, Spain, and Other, which contains all the other nationalities:

```
df.loc[df.Nationality =='England',  'Nationality2'] = 'England'


df.loc[df.Nationality =='Spain',  'Nationality2'] = 'Spain'


df.loc[df.Nationality =='Germany',  'Nationality2'] = 'Germany'


df.loc[~df.Nationality.isin(['England', 'German', 'Spain']),  'Nationality2'] =
'Other'
```

Next, let's create a dictionary that contains the proportion values for each of these:

```
prop = dict(Counter(df['Nationality2']))


for key, values in prop.items():


   prop[key] = (values)/len(df)*100
```

```
print(prop)
```

The result shows you a dictionary or list showing Other, Spain, and England, with their player proportions. From here, we can use the Matplotlib pie method to create a pie chart from the dictionary:

```
fig1, ax1 = plt.subplots()


ax1.pie(prop.values(), labels=prop.keys(), autopct='%1.1f%%',


    shadow=True, startangle=90)


ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.


plt.show()
```

As you can see, these methods all of us very powerful ways of visualizing category proportions in the data.

Visualizing Data with Seaborn

Seaborn is another useful library, built on Matplotlib and offering a powerful plot formatting method. Once you have got to grips with Matplotlib, you should move on to Seaborn to use more complex visualizations.

For example, we can use the set() method from Seaborn to improve how our Matplotlib plots look.

First, Seaborn needs to be imported, and all the figures generated earlier need to be reformatted. Add the following to the top of your script and run it:

```
import seaborn as sns


sns.set()
```

Using Seaborn to Generate Histograms

We can use Seaborn to generate the same visualizations we generated with

Matplotlib. First, we'll generate the histogram showing the overall column. This is done on the Seaborn object with distplot:

```
sns.distplot(df['Overall'])
```

We can also reuse our plt object to make additional formats on the axis and set the title:

```
plt.xlabel('Overall')


plt.ylabel('Frequency')


plt.title('Histogram of Overall Rating')


plt.show()
```

As you can see, this is much better looking than the plot generated with Matplotlib.

Using Seaborn to Generate Scatter Plots

We can also use Seaborn to generate scatter plots in a straightforward way, so let's recreate our earlier one:

```
sns.scatterplot(df['Overall'], df['wage_euro'])


plt.title('Overall vs. Wage')


plt.ylabel('Wage')


plt.xlabel('Overall')


plt.show( )
```

Using Seaborn to Generate Heatmaps

Seaborn is perhaps best known for its ability to create correlation heatmaps.

These are used for identifying variable independence and, before generating them, the correlation between a specified set of numerical columns must be calculated. We'll do this for four columns:

- age
- overall
- wage_euro
- skill moves

```
corr = df[['Overall', 'Age', 'wage_euro', 'Skill Moves']].corr()


sns.heatmap(corr, annot=True)


plt.title('Heatmap of Overall, Age, wage_euro, and Skill Moves')


plt.show()
```

If we want to see the correlation values, annot can be set to true:

```
sns.heatmap(corr, annot=True)
```

Using Seaborn to Generate Pairs Plot

This is the final tool we'll look at – a method called pairplot – which allows for a matrix of distributions to be generated, along with a scatter plot for a specified numerical feature set. We'll do this with three columns:

- age
- overall
- potential

```
data = df[['Overall', 'Age', 'Potential',]]


sns.pairplot(data)


plt.show()
```

This proves to be one of the quickest and easiest ways to visualize the relationships between the variables and the numerical value distributions using

scatter plots.

Matplotlib and Seaborn are both excellent tools, very valuable in the data science world. Matplotlib helps you label, title, and format graphs, one of the most important factors in effectively communicating your results. It also gives many of the fundamental tools we need to produce visualizations, such as scatter plots, histograms, bar charts, and pie charts.

Seaborn is also a very important library because it offers in-depth statistical tools and stunning visuals. As you saw, the Seaborn-generated visuals were much more aesthetically pleasing to look at than the Matplotlib visuals. The Seaborn tools also allow for more sophisticated visuals and analyses.

Both libraries are popular in data visualization, both allowing quick data visualization to tell stories with the data. There is some overlap in their use cases, but it is wise to learn both to ensure you do the best job with your data as you can.

Now buckle up and settle in. The next part is by far the longest as we dive deep into the most important subset of data science – machine learning.

Part Five – An In-Depth Guide to Machine Learning

The term “machine learning” was first coined in 1959 by Arthur Samuel, an AI and computer gaming pioneer who defined it as “a field of study that provides computers with the capability of learning without explicit programming.”

That means that machine learning is an AI application that allows software to learn by experience, continually improving without explicit programming. For example, think about writing a program to identify vegetables based on properties such as shape, size, color, etc.

You could hardcode it all, add a few rules which you use to identify the vegetables. To some, this might seem to be the only way that will work, but there isn’t one programmer who can write the perfect rules that will work for all cases. That’s where machine learning comes in, without any rules – this makes it more practical and robust. Throughout this chapter, you will see how machine learning can be used for this task.

So, we could say that machine learning is a study in making machines exhibit human behavior in terms of decision making and behavior by allowing them to learn with little human intervention – which means no explicit programming. This begs the question, how does a machine gain experience, and where does it learn from? The answer to that is simple – data, the fuel that powers machine learning. Without it, there is no machine learning.

One more question – if machine learning was first mentioned can k in 1959, why has it taken so long to become mainstream? The main reason for this is the significant amount of computing power machine learning requires, not to mention the hardware capable of storing vast amounts of data. It’s only in recent years that we have achieved the requirements needed to practice machine learning.

How Do Machine Learning and Traditional Programming Differ?

Traditional programming involves input data being fed into a machine, together with a clean, tested program, to generate some output. With machine learning, the input and the output data are fed to the machine during the initial phase, known as the learning phase. From there, the machine will work the programming out for itself.

Don’t worry if this doesn’t make sense right now. As you work through this in-depth chapter, it will all come together.

Why We Need Machine Learning

These days, machine learning has more attention than you would think possible. One of its primary uses is to automate tasks that require human intelligence to perform. Machines can only replicate this intelligence via machine learning.

Businesses use machine learning to automate tasks and automate and create data analysis models. Some industries are heavily reliant on huge amounts of data which they use to optimize operations and make the right decisions. Machine learning makes it possible to create models to process and analyze complex data in vast amounts and provide accurate results. Those models are scalable and precise, and they function quickly, allowing businesses to leverage their huge power and the opportunities they provide while avoiding risks, both known and unknown.

Some of the many uses of machine learning in the real world include text generation, image recognition, and so on, this increasing the scope, ensuring machine learning experts become highly sought after individuals.

How Does Machine Learning Work?

Machine learning models are fed historical data, known as training data. They learn from that and then build a prediction algorithm that predicts an output for a completely different set of data, known as testing data, that is given as an input to the system. How accurate the models are is entirely dependent on the quality of the data and how much of it is provided – the more quality data there is, the better the results and accuracy are.

Let’s say we have a highly complex issue that requires predictions. Rather than writing complex code, we could give the data to machine learning algorithms. These develop logic and provide predictions, and we’ll be discussing the different types of algorithms shortly.

Machine Learning Past to Present

These days, we see plenty of evidence of machine learning in action, such as Natural Language Processing and self-drive vehicles, to name but two. Machine learning has been in existence for more than 70 years, though, all beginning in 1943. Warren McCulloch, a neuropsychologist, and Water Pitts, a mathematician, produced a paper about neurons and the way they work. They went on to create a model with an electrical circuit, producing the first-ever neural network.

Alan Turing created the Turing test in 1950. This test was used to determine whether computers possessed real intelligence, and passing the test required the computer to fool a human into believing it was a human.

Arthur Samuel wrote the very first computer learning program in 1952, a game of checkers that the computer improved at the more it played. It did this by studying the moves, determining which ones came into the winning strategies and then using those moves in its own program.

In 1957, Frank Rosenblatt designed the perceptron, a neural network for computers that simulates human brain thought processes. Ten years later, an algorithm was created called “Nearest Neighbor.” This algorithm allowed computers to begin using pattern recognition, albeit very basic at the time. This was ideal for salesmen to map routes, ensuring all their visits could be completed within the shortest time.

It was in the 1990s that the biggest changes became apparent. Work was moving from an approach driven almost entirely by knowledge to a more data-driven one. Scientists started creating programs that allowed computers to analyze huge swathes of data, learning from the results to draw conclusions.

In 1997, Deep Blue, created by IBM, was the first computer-based chess player to beat a world champion at his own game. The program searched through potential moves, using its computing power to choose the best ones. And just ten years later, Geoffrey Hinton coined the term “deep” learning to describe the new algorithms that allowed computers to recognize text and objects in images and videos.

In 2012, Alex Krizhevzky published a paper with Ilya Sutskever and Geoffrey Hinton. An influential paper, it described a computing model that could reduce the error rate significantly in image recognition systems. At the same time, Google’s X Lab came up with an algorithm that could browse YouTube videos autonomously, choosing those that had cats in them.

In 2016, Google DeepMind researchers created AlphaGo to play the ancient Chinese game, Go, and pitted it against the world champion, Lee Sedol. AlphaGo beat the reigning world champion four times out of five.

Finally, in 2020, GPT-3 was released, arguably the most powerful language model in the world. Released by OpenAI, the model could generate fully

functioning code, write creative fiction, write good business memos, and so much more, with its use cases only limited by imagination.

Machine Learning Features

Machine learning offers plenty of features, with the most prominent being the following:

- **Automation –** most email accounts have a spam folder where all your spam emails appear. You might question how your email provider knows which emails are spam, and the answer is machine learning. The process is automated after the algorithm is taught to recognize spam emails.

  Being able to automate repetitive tasks is one of machine learning's best features. Many businesses worldwide are using it to deal with email and paperwork automation. Take the financial sector, for example. Every day, lots of data-heavy, repetitive tasks need to be performed, and machine learning takes the brunt of this work, automating it to speed it up and freeing up valuable staff hours for more important tasks.

- **Better Customer Experience –** good customer relationships are critical to businesses, helping to promote brand loyalty and drive engagement. And the best way for a customer to do this is to provide better services and food customer experiences. Machine learning comes into play with both of these. Think about when you last saw ads on the internet or visited a shopping website. Most of the time, the ads are relevant to things you have searched for, and the shopping sites recommend products based on previous searches or purchases. Machine learning is behind these incredibly accurate recommendation systems and ensures that we can tailor experiences to each user.

  In terms of services, most businesses use chatbots to ensure that they are available 24/7. And some are so intelligent that many customers don’t even realize they are not chatting to a real human being.

- **Automated Data Visualization –** these days, vast amounts of data are being generated daily by individuals and businesses, for example, Google, Facebook, and Twitter. Think about the sheer amount of data each of those must be generating daily. That data can be used to

visualize relationships, enabling businesses to make decisions that benefit them and their customers. Using automated data visualization platforms, businesses can gain useful insights into the data and use them to improve customer experiences and customer service.

- **BI or Business Intelligence –** When machine learning characteristics are merged with data analytics, particularly big data, companies find it easier to find solutions to issues and use them to grow their business and increase their profit. Pretty much every industry uses this type of machine learning to increase operations.

With Python, you get the flexibility to choose between scripting and object-oriented programming. You don't need to recompile any code – the changes can be implemented, and the results are shown instantly. It is the most versatile of all the programming languages and is multi-platforms, which is why it works so well for machine learning and data science.

Different Types of Machine Learning

Machine learning falls into three primary categories:

- Supervised learning
- Unsupervised learning
- Reinforcement learning

We'll look briefly at each type before we turn our attention to the popular machine learning algorithms.

Supervised Learning

We'll start with the easiest of examples to explain supervised learning. Let's say you are trying to teach your child how to tell the difference between a cat and a dog. How do you do it?

You point out a dog and tell your child, "that's a dog." Then you do the same with a cat. Show them enough and, eventually, they will be able to tell the difference. They may even begin to recognize different breeds in time, even if they haven't seen them before.

In the same way, supervised learning works with two sets of variables. One is a target variable or labels, which is the variable we are predicting, while the other is the features variable, which are the ones that help us make the

predictions. The model is shown the features and the labels that go with them and will then find patterns in the data it looks at. To clear this up, below is an example taken from a dataset about house prices. We want to predict a house price based on its size. The target variable is the price, and the feature it depends on is the size.

Number of rooms Price

1 $100

3 $300

5 $500

There are thousands of rows and multiple features in real datasets, such as the size, number of floors, location, etc.

So, the easiest way to explain supervised learning is to say that the model has x, which is a set of input variables, and y, which is the output variable. An algorithm is used to find the mapping function between x and y, and the relationship is $y = f(x)$.

It is called supervised learning because we already know what the output will be, and we provide it to the machine. Each time it runs, the algorithm is corrected, and this continues until we get the best possible results. The data set is used to train the algorithm to an optimal outcome.

Supervised learning problems can be grouped as follows:

- **Regression –** predicts future values. Historical data is used to train the model, and the prediction would be the future value of a property, for example.
- **Classification –** the algorithm is trained on various labels to identify things in a particular category, i.e., cats or dogs, oranges or apples, etc.

Unsupervised Learning

Unsupervised learning does not involve any target variables. The machine is given only the features to learn from, which it does alone and finds structures in the data. The idea is to find the distribution that underlies the data to learn more about the data.

Unsupervised learning can be grouped as follows:

- **Clustering –** the input variables sharing characteristics are put

together, i.e., all users that search for the same things on the internet

- **Association –** the rules governing meaningful data associations are discovered, i.e., if a person watches this, they are likely to watch that too.

Reinforcement Learning

In reinforcement learning, models are trained to provide decisions based on a reward/feedback system. They are left to learn how to achieve a goal in complex situations and, whenever they achieve the goal during the learning, they are rewarded.

It differs from supervised learning in that there isn't an available answer, so the agent determines what steps are needed to do the task. The machine uses its own experiences to learn when no training data is provided.

Common Machine Learning Algorithms

There are far too many machine learning algorithms to possibly mention them all, but we can talk about the most common ones you are likely to use. Later, we'll look at how to use some of these algorithms in machine learning examples, but for now, here's an overview of the most common ones.

Naïve Bayes Classifier Algorithm

The Naïve Bayes classifier is not just one algorithm. It is a family of classification algorithms, all based on the Bayes Theorem. They all share one common principle – each features pair being classified is independent of one another.

Let's dive straight into this with an example.

Let's use a fictional dataset describing weather conditions for deciding whether to play a golf game or not. Given a specific condition, the classification is Yes (fit to play) or No (unfit to play.)

The dataset looks something like this:

Outlook Temperature Humidity Windy Play Golf

0 Rainy Hot High False No

1 Rainy Hot High True No

2 Overcast Hot High False Yes

3 Sunny Mild High False Yes

4 Sunny Cool Normal False Yes

5 Sunny Cool Normal True No

6 Overcast Cool Normal True Yes

7 Rainy Mild High False No

8 Rainy Cool Normal False Yes

9 Sunny Mild Normal False Yes

10 Rainy Mild Normal True Yes

11 Overcast Mild High True Yes

12 Overcast Hot Normal False Yes

13 Sunny Mild High True No

This dataset is split in two:

- **Feature matrix –** where all the dataset rows (vectors) are stored – the vectors contain the dependent feature values. In our dataset, the four features are Outlook, Temperature, Windy, and Humidity.
- **Response vector –** where all the class variable values, i.e., prediction or output, are stored for each feature matrix row. In our dataset, the class variable name is called Play Golf.

Assumption

The Naïve Bayes classifiers share an assumption that all features will provide the outcome with an equal and independent contribution. As far as our dataset is concerned, we can understand this as:

- An assumption that there is no dependence between any features pair. For example, a temperature classified as Hot is independent of humidity, and a Rainy outlook is not related to the wind. As such, we assume the features are independent.
- Each feature has the same importance (weight). For example, if you only know the humidity and the temperature, you cannot provide an accurate prediction. No attribute is considered irrelevant, and all play an equal part in determining the outcome.

Bayes' Theorem

The Bayes' Theorem uses the probability of an event that happened already to determine the probability of another one happening. The Theorem is mathematically stated as:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

A and B are both events and P(B)?0:

- We want to predict the probability of event A, should event B equal true. Event B is also called Evidence.
- P(A) is the prior probability of A, i.e., the probability an event will occur before any evidence is seen. That evidence consists of an unknown instance's attribute value – in our case, it is event B.
- P(A|B) is called a posteriori probability of B, or the probability an event will occur once the evidence is seen.

We can apply the Theorem to our dataset like this:

$$P(y|X) = \frac{P(X|y)P(y)}{P(X)}$$

Where the class variable is y and the dependent features vector, size n, is X, where

$$X = (x_1, x_2, x_3, ..., x_n)$$

Just to clear this up, a feature vector and its corresponding class variable example is:

X = (Rainy, Hot, High, False)

y = No

Here, P(y|X) indicates the probability of not playing, given a set of weather conditions as follows:

- Rainy outlook
- Hot temperature
- High humidity
- No wind

Naïve Assumption

Giving the Bayes Theorem a naïve assumption means providing independence between the features. The evidence can now be split into its independent parts. If any two events, A and B, are independent:

P(A,B) = P(A)P(B)

The result is:

$$P(y|x_1,...,x_n) = \frac{P(x_1|y)P(x_2|y)...P(x_n|y)P(y)}{P(x_1)P(x_2)...P(x_n)}$$

And this is expressed as:

$$P(y|x_1,...,x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i|y)}{P(x_1)P(x_2)...P(x_n)}$$

For a given input, the denominator stays constant so that term can be removed:

$$P(y|x_1,...,x_n) \propto P(y)\prod_{i=1}^{n} P(x_i|y)$$

The next step is creating a classifier model. We will determine the probability of a given input set for all class variable y's possible values, choosing the output that offers the maximum probability. This is mathematically expressed as:

$$y = argmax_y P(y)\prod_{i=1}^{n} P(x_i|y)$$

That leaves us having to calculate P(y) and P(x_i | y).

P(y) is also known as the class probability, and P(x_i | y) is the conditional probability.

There are a few Naïve Bayes classifiers, and their main difference is via the assumptions about the P(x_i | y) distribution.

Let's apply this formula to our weather dataset manually. We will need to carry out some precomputations, i.e., for each x_i in X and y_i in y, we must find P(X=y_i). These calculations can be seen in the following tables:

Outlook

	Yes	No	P(yes)	P(no)
Sunny	2	3	2/9	3/5
Overcast	4	0	4/9	0/5
Rainy	3	2	3/9	2/5
Total	9	5	100%	100%

Temperature

	Yes	No	P(yes)	P(no)
Hot	2	2	2/9	2/5
Mild	4	2	4/9	2/5
Cool	3	1	3/9	1/5
Total	9	5	100%	100%

Humidity

	Yes	No	P(yes)	P(no)
High	3	4	3/9	4/5
Normal	6	1	6/9	1/5
Total	9	5	100%	100%

Wind

	Yes	No	P(yes)	P(no)
False	6	2	6/9	2/5
True	3	3	3/9	3/5
Total	9	5	100%	100%

Play		P(Yes)/P(No)
Yes	9	9/14
No	5	5/14
Total	14	100%

$P(X=x_i \mid y_i)$ is calculated for each x_i in X and y_i in y manually in the above ta
For example, we calculate the probability of playing given a cool temperature – P(temp. = cool | play gold = yes = 3/9.

We also need the (P(y)) class probabilities, calculated in the final table above.

For example, the probabilities are calculated as P(play gold = yes) = 9/14.

Now the classifier is ready to go, so we'll test it on a feature set we'll call 'today.'

today = (Sunny, Hot, Normal, False)

Given this, the probability of playing is:

$$P(Yes|today) = \frac{P(SunnyOutlook|Yes)P(HotTemperature|Yes)P(NormalHumidity|Yes)P(NoWind|Yes)P(Yes)}{P(today)}$$

And not playing is:

$$P(No|today) = \frac{P(SunnyOutlook|No)P(HotTemperature|No)P(NormalHumidity|No)P(NoWind|No)P(No)}{P(today)}$$

P(today) appears in both of the probabilities, so we can ignore it and look for the proportional probabilities:

$$P(Yes|today) \propto \frac{2}{9}.\frac{2}{9}.\frac{6}{9}.\frac{6}{9}.\frac{9}{14} \approx 0.0141$$

and

$$P(No|today) \propto \frac{3}{5}.\frac{2}{5}.\frac{1}{5}.\frac{2}{5}.\frac{5}{14} \approx 0.0068$$

Because

$$P(Yes|today) + P(No|today) = 1$$

We can make the sum equal to 1 to convert the numbers into a probability – this is called normalization:

$$P(Yes|today) = \frac{0.0141}{0.0141+0.0068} = 0.67$$

and

$$P(No|today) = \frac{0.0068}{0.0141+0.0068} = 0.33$$

Because

$$P(Yes|today) > P(No|today)$$

In that case, we predict that Yes, golf will be played.

We've discussed a method that we can use for discrete data. Should we be working with continuous data, assumptions need to be made about each feature's value distribution.

We'll talk about one of the popular Naïve Bayes classifiers.

Gaussian Naive Bayes classifier

In this classifier, the assumption is that each feature's associated continuous

values are distributed via a Gaussian distribution, also known as a Normal distribution. When this is plotted, you see a curve shaped like a bell, symmetric around the feature values' mean.

The assumption is that the likelihood of the features is Gaussian, which means the conditional probability is provided via:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} exp\left(-\frac{(x_i-\mu_y)^2}{2\sigma_y^2}\right)$$

Let's see what a Python implementation of this classifier looks like using Scikit-learn:

```
# load the iris dataset
from sklearn.datasets import load_iris
iris = load_iris()

# store the feature matrix (X) and response vector (y)
X = iris.data
y = iris.target

# splitting X and y into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,
random_state=1)

# training the model on training set
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
gnb.fit(X_train, y_train)

# making predictions on the testing set
y_pred = gnb.predict(X_test)
```

```
# comparing actual response values (y_test) with predicted response values (y_pred)

from sklearn import metrics

print("Gaussian Naive Bayes model accuracy(in %):", metrics.accuracy_score(y_test, y_pred)*100)
```

And the output is:

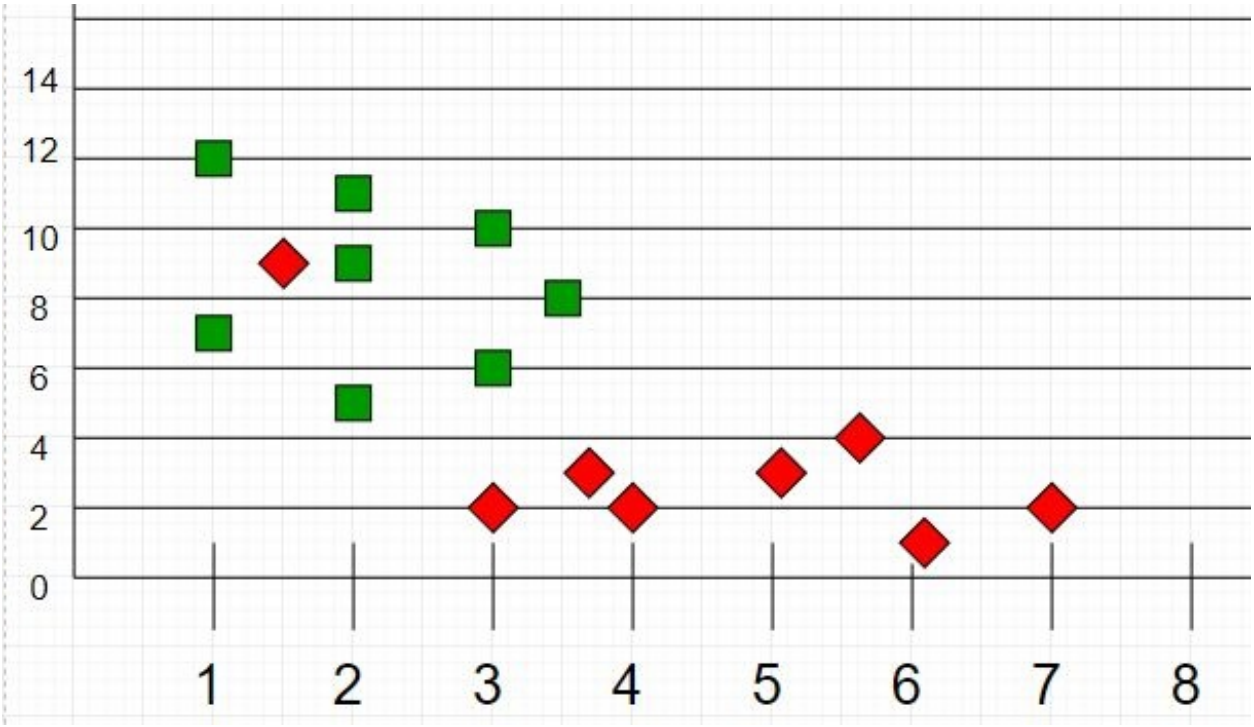
```
Gaussian Naive Bayes model accuracy(in %): 95.0
```

K-Nearest Neighbors

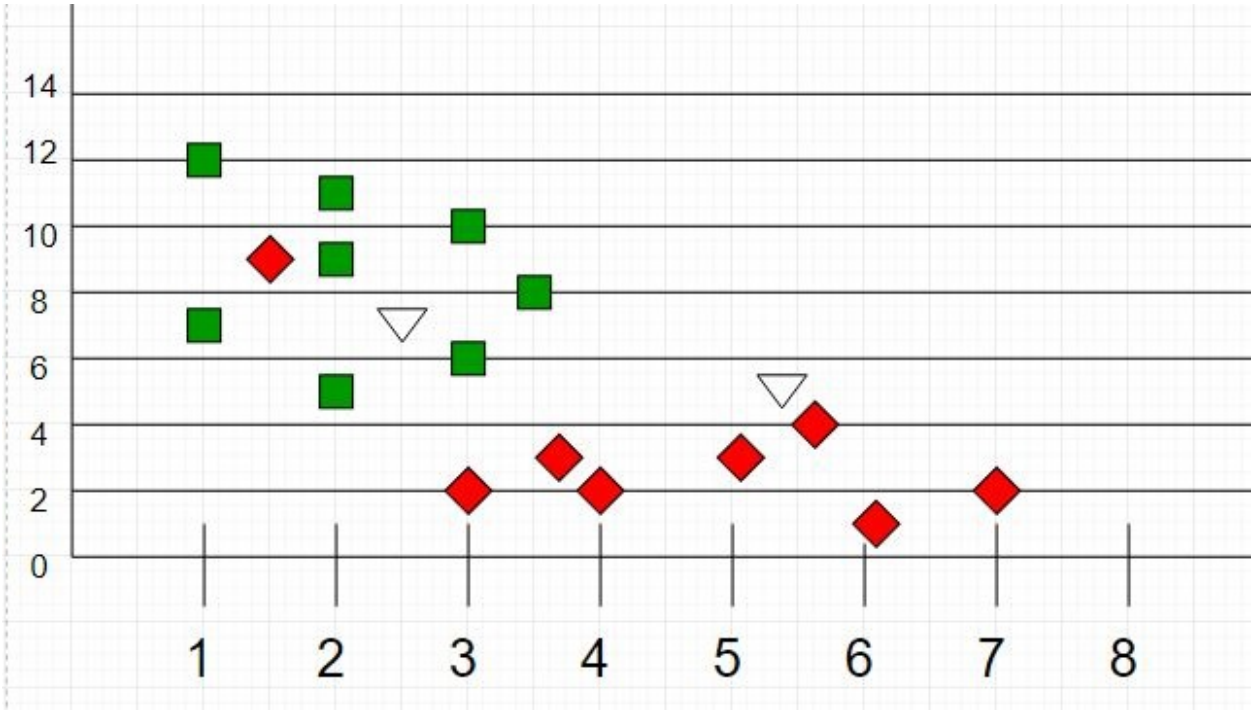
K-Nearest Neighbors, otherwise known as K-NN, is a critical yet simple classification algorithm. It is a supervised learning algorithm used for data mining, pattern detection, and intrusion detection.

In real-life scenarios, it is considered disposable because it is non-parametric, which means it makes no underlying assumptions about the data distribution. Some prior data, the training data, is provided, and this uses attributes to classify the coordinates into specific groups.

Here is an example showing a table with some data points and two features:



With another set of data points, called the testing data, we can analyze the data and allocate the points into a group. Unclassified points on the table are marked ‘White.”



Intuition

If those points were plotted on a graph, we might see some groups or clusters. Unclassified points can be assigned to a group just by seeing which group the unclassified point’s nearest neighbors are in. That means a point near a cluster classified ‘Red’ is more likely to be classified as ‘Red.’

Using intuition, it’s clear that the first point (2.5, 7) should have a classification of ‘Green’ while the second one (5.5, 4.5) should be ‘Red.’

Algorithm

Here, m indicates how many training samples we have, and p is an unknown point. Here’s what we want to do:

1. The training samples are stored in arr[], which is an array containing data points. Each of the array elements represents (x, y), which is a tuple.

For i=0 to m

2. Calculate d(arr[i], p), which is the Euclidean distance

3. Set S of K should be the smallest distances retrieved. Each distance is correspondent to a data point we already classified.
4. The majority label among S is returned.

We can keep K as an odd number, making it easier to work out the clear majority where there are only two possible groups. As K increases, the boundaries across the classifications become more defined and smoother. And, as the number of data points increases in the training set, the better the classifier accuracy becomes.

Here's an example program, written in C++ and Python, where 0 and 1 are the classifiers::

```
// C++ program to find groups of unknown
// Points using K nearest neighbor algorithm.
#include <bits/stdc++.h>
using namespace std;

struct Point
{
   int val;     // Group of point
   double x, y;     // Co-ordinate of point
   double distance; // Distance from test point
};

// Used to sort an array of points by increasing
// order of distance
bool comparison(Point a, Point b)
{
   return (a.distance < b.distance);
}
```

```
// This function finds classification of point p using
// k nearest neighbor algorithm. It assumes only two
// groups and returns 0 if p belongs to group 0, else
// 1 (belongs to group 1).
int classifyAPoint(Point arr[], int n, int k, Point p)
{
    // Fill distances of all points from p
    for (int i = 0; i < n; i++)
        arr[i].distance =
            sqrt((arr[i].x - p.x) * (arr[i].x - p.x) +
                 (arr[i].y - p.y) * (arr[i].y - p.y));

    // Sort the Points by distance from p
    sort(arr, arr+n, comparison);

    // Now consider the first k elements and only
    // two groups
    int freq1 = 0;     // Frequency of group 0
    int freq2 = 0;     // Frequency of group 1
    for (int i = 0; i < k; i++)
    {
        if (arr[i].val == 0)
            freq1++;
        else if (arr[i].val == 1)
            freq2++;
    }
```

```
    return (freq1 > freq2 ? 0 : 1);
}

// Driver code
int main()
{
    int n = 17; // Number of data points
    Point arr[n];

    arr[0].x = 1;
    arr[0].y = 12;
    arr[0].val = 0;

    arr[1].x = 2;
    arr[1].y = 5;
    arr[1].val = 0;

    arr[2].x = 5;
    arr[2].y = 3;
    arr[2].val = 1;

    arr[3].x = 3;
    arr[3].y = 2;
    arr[3].val = 1;

    arr[4].x = 3;
    arr[4].y = 6;
```

```
arr[4].val = 0;

arr[5].x = 1.5;
arr[5].y = 9;
arr[5].val = 1;

arr[6].x = 7;
arr[6].y = 2;
arr[6].val = 1;

arr[7].x = 6;
arr[7].y = 1;
arr[7].val = 1;

arr[8].x = 3.8;
arr[8].y = 3;
arr[8].val = 1;

arr[9].x = 3;
arr[9].y = 10;
arr[9].val = 0;

arr[10].x = 5.6;
arr[10].y = 4;
arr[10].val = 1;

arr[11].x = 4;
```

```
arr[11].y = 2;
arr[11].val = 1;

arr[12].x = 3.5;
arr[12].y = 8;
arr[12].val = 0;

arr[13].x = 2;
arr[13].y = 11;
arr[13].val = 0;

arr[14].x = 2;
arr[14].y = 5;
arr[14].val = 1;

arr[15].x = 2;
arr[15].y = 9;
arr[15].val = 0;

arr[16].x = 1;
arr[16].y = 7;
arr[16].val = 0;

/*Testing Point*/
Point p;
p.x = 2.5;
p.y = 7;
```

```
    // Parameter to decide group of the testing point
    int k = 3;
    printf ("The value classified to unknown point"
          " is %d.\n", classifyAPoint(arr, n, k, p));
    return 0;
}
```

And the output is:

```
The value classified to the unknown point is 0.
```

Support Vector Machine Learning Algorithm

Support vector machines or SVMs are grouped under the supervised learning algorithms used to analyze the data used in regression and classification analysis. The SVM is classed as a discriminative classifier, defined formally by a separating hyperplane. In simple terms, with labeled training data, the supervised learning part, the output will be an optimal hyperplace that takes new examples and categorizes them.

SVMs represent examples as points in space. Each is mapped so that the separate categories have a wide, clear gap separating them. As well as being used for linear classification, an SVM can also be used for efficient non-linear classification, with the inputs implicitly mapped into high-dimensional feature spaces.

So, what does an SVM do?

When you provide an SVM algorithm with training examples, each one labeled as belonging to one of the two categories, it will build a model that will assign newly provided examples to one of the categories. This makes it a 'non-probabilistic binary linear classification.'

To illustrate this, we'll use an SVM model that uses ML tools, like Scikit-learn, to classify cancer UCI datasets. For this, you need to have NumPy, Pandas, Matplotlib, and Scikit-learn.

First, the dataset must be created:

```
# importing scikit learn with make_blobs
```

```
from sklearn.datasets.samples_generator import make_blobs

# creating datasets X containing n_samples
# Y containing two classes
X, Y = make_blobs(n_samples=500, centers=2,
                  random_state=0, cluster_std=0.40)
import matplotlib.pyplot as plt
# plotting scatters
plt.scatter(X[:, 0], X[:, 1], c=Y, s=50, cmap='spring');
plt.show()
```

Support vector machines don't just draw lines between classes. They also consider a region around the line of a specified width. Here's an example:

```
# creating line space between -1 to 3.5
xfit = np.linspace(-1, 3.5)

# plotting scatter
plt.scatter(X[:, 0], X[:, 1], c=Y, s=50, cmap='spring')

# plot a line between the different sets of data
for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:
    yfit = m * xfit + b
    plt.plot(xfit, yfit, '-k')
    plt.fill_between(xfit, yfit - d, yfit + d, edgecolor='none',
    color='#AAAAAA', alpha=0.4)

plt.xlim(-1, 3.5);
plt.show()
```

Importing Datasets

The SVMs intuition is optimizing linear discriminant models representing the distance between the datasets, which is a perpendicular distance. Let's use our training data to train the model. Before we do that, the cancer datasets need to be imported as a CSV file, and we will train two of all the features:

```
# importing required libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


# reading csv file and extracting class column to y.
x = pd.read_csv("C:\...\cancer.csv")
a = np.array(x)
y  = a[:,30] # classes having 0 and 1


# extracting two features
x = np.column_stack((x.malignant,x.benign))


# 569 samples and 2 features
x.shape


print (x),(y)
```

The output would be:

```
[[  122.8    1001.  ]
[  132.9    1326.  ]
[  130.    1203.  ]
...,
[  108.3    858.1 ]
```

```
[  140.1   1265.  ]
[   47.92   181.  ]]


array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  1.,  1.,  1.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  1.,  1.,  1.,  1.,
        1.,  0.,  0.,  1.,  0.,  0.,  1.,  1.,  1.,  1.,  0.,  1., ....,
        1.])
```

Fitting Support Vector Machines

Next, we need to fit the classifier to the points.

```
# import support vector classifier
# "Support Vector Classifier"
from sklearn.svm import SVC
clf = SVC(kernel='linear')


# fitting x samples and y classes
clf.fit(x, y)
```

Once the model is fitted, it can be used to make predictions on new values:

```
clf.predict([[120, 990]])


clf.predict([[85, 550]])
array([ 0.])
array([ 1.])
```

The data and the pre-processing methods are analyzed, and Matplotlib is used to produce optimal hyperplanes.

Linear Regression Machine Learning Algorithm

No doubt you have heard of linear regression, as it is one of the most popular machine learning algorithms. It is a statistical method used to model relationships between one dependent variable and a specified set of independent variables. For this section, the dependent variable is referred to as response and the independent variables as features.

Simple Linear Regression

This is the most basic form of linear regression and is used to predict responses from single features. Here, the assumption is that there is a linear relationship between two variables. As such, we want a linear function that can predict the response (y) value as accurately as it can as a function of the independent (x) variable or feature.

Let's look at a dataset with a response value for each feature:

x	0	1	2	3	4	5	6	7	8	9
y	1	3	2	5	7	8	8	9	10	12

We define the following for generality:

- x is defined as a feature vector, i.e. x = [x_1, x_2, …., x_n]
- y is defined as a response vector, i.e. y = [y_1, y_2, …., y_n]

These are defined for n observations, i.e. n=10.

The model's task is to fund the best fitting line to predict responses for new feature values, i.e., those features not in the dataset already. The line is known as a regression line, and its equation is represented as:

$h(x_i) = \beta_0 + \beta_1 x_i$

Here:

- h(x_1) is representing the ith observation's predicted response
- b_0 and b_1 are both coefficients and are representing the y-intercept and the regression line slope, respectively.

Creating the model requires that we estimate or learn the coefficients' values, and once they have been estimated, the model can predict response.

We will be making use of the principle of Least Squares, so consider the following:

$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i = h(x_i) + \varepsilon_i \Rightarrow \varepsilon_i = y_i - h(x_i)$

In this, e_i is the ith observation's residual error and we want to reduce the total residual error.

The cost function or squared error is defined as:

$J(\beta_0, \beta_1) = \frac{1}{2n} \sum_{i=1}^{n} \varepsilon_i^2$

We want to find the b_0 and b_1 values, where J(b_1, b_1) is the minimum.

Without drawing you through all the mathematical details, this is the result:

$\beta_1 = \frac{SS_{xy}}{SS_{xx}}$

$\beta_0 = \bar{y} - \beta_1 \bar{x}$

Here, SS_xy represents the sum of y and x (the cross deviations:

$SS_{xy} = \sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) = \sum_{i=1}^{n} y_i x_i - n\bar{x}\bar{y}$

While SS_xx is representing the sum of x, the squared deviations:

$SS_{xx} = \sum_{i=1}^{n}(x_i - \bar{x})^2 = \sum_{i=1}^{n} x_i^2 - n(\bar{x})^2$

Let's look at the Python implementation of our dataset:

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


def estimate_coef(x, y):
    # number of observations/points
    n = np.size(x)


    # mean of x and y vector
    m_x = np.mean(x)
    m_y = np.mean(y)


    # calculating cross-deviation and deviation about x
    SS_xy = np.sum(y*x) - n*m_y*m_x
    SS_xx = np.sum(x*x) - n*m_x*m_x
```

```
    # calculating regression coefficients
    b_1 = SS_xy / SS_xx
    b_0 = m_y - b_1*m_x

    return (b_0, b_1)

def plot_regression_line(x, y, b):
    # plotting the actual points as scatter plot
    plt.scatter(x, y, color = "m",
                marker = "o", s = 30)

    # predicted response vector
    y_pred = b[0] + b[1]*x

    # plotting the regression line
    plt.plot(x, y_pred, color = "g")

    # putting labels
    plt.xlabel('x')
    plt.ylabel('y')

    # function to show plot
    plt.show()

def main():
    # observations / data
    x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
    y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])

    # estimating coefficients
    b = estimate_coef(x, y)
    print("Estimated coefficients:\nb_0 = {}  \
        \nb_1 = {}".format(b[0], b[1]))

    # plotting regression line
    plot_regression_line(x, y, b)

if __name__ == "__main__":
    main()
```

And the output is:

```
Estimated coefficients:
b_0 = -0.0586206896552
b_1 = 1.45747126437
```

Multiple Linear Regression

Multiple linear regression tries to model relationships between at least two features and one response, and to do this, a linear equation is fit to the observed data. It's nothing more complex than an extension of simple linear regression.

Let's say we have a dataset containing p independent variables or features and one dependent variable or response. The dataset has n rows (observations).

We define the following:

- X (feature matrix) = a matrix of n X p size where x_{ij} is used to denote the values of the ith observation's jth feature.

So

$$\begin{pmatrix} x_{11} & \cdots & x_{1p} \\ x_{21} & \cdots & x_{2p} \\ \vdots & \ddots & \vdots \\ x_{n1} & \vdots & x_{np} \end{pmatrix}$$

And

- y (response vector) = a vector of n size, where y_{i} denotes the ith observation's response value

$$y = \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ y_n \end{bmatrix}$$

The p features regression line is represented as:

$$h(x_i) = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip}$$

Where h(x_i) is the ith observation's predicted response, and the regression coefficients are b_0, b_1, …., b_p.

We can also write:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip} + \varepsilon_i$$
or
$$y_i = h(x_i) + \varepsilon_i \Rightarrow \varepsilon_i = y_i - h(x_i)$$

Where the ith observation's residual error is represented by e_i.

By presenting the matrix features as the following, the linear model can be generalized a bit further:

$$X = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1p} \\ 1 & x_{21} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{pmatrix}$$

And the linear model can now be expressed as the following in terms of matrices:

$$y = X\beta + \varepsilon$$

where

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ . \\ . \\ \beta_p \end{bmatrix}$$

and

$$\varepsilon = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ . \\ . \\ \varepsilon_n \end{bmatrix}$$

The estimate of b can now be determined using the Least Squares method, which is used to determine b where the total residual errors are minimized. The result is:

$\hat{\beta} = (X'X)^{-1}X'y$

Where ' represents the matrix transpose, and the inverse is represented by -1.

Now that the least-squares estimate, b' is known, we can estimate the multiple linear regression models as:

$\hat{y} = X\hat{\beta}$

Where the estimated response vector is y.

Here's a Python example of multiple linear regression on the Boston house price dataset:

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


# load the boston dataset
boston = datasets.load_boston(return_X_y=False)


# defining feature matrix(X) and response vector(y)
X = boston.data
y = boston.target
```

```
# splitting X and y into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,
                                                    random_state=1)

# create linear regression object
reg = linear_model.LinearRegression()

# train the model using the training sets
reg.fit(X_train, y_train)

# regression coefficients
print('Coefficients: ', reg.coef_)

# variance score: 1 means perfect prediction
print('Variance score: {}'.format(reg.score(X_test, y_test)))

# plot for residual error

## setting plot style
plt.style.use('fivethirtyeight')

## plotting residual errors in training data
plt.scatter(reg.predict(X_train), reg.predict(X_train) - y_train,
            color = "green", s = 10, label = 'Train data')
```

```
## plotting residual errors in test data
plt.scatter(reg.predict(X_test), reg.predict(X_test) - y_test,
        color = "blue", s = 10, label = 'Test data')

## plotting line for zero residual error
plt.hlines(y = 0, xmin = 0, xmax = 50, linewidth = 2)

## plotting legend
plt.legend(loc = 'upper right')

## plot title
plt.title("Residual errors")

## method call for showing the plot
plt.show()
```

And the output is:

```
Coefficients:
[ -8.80740828e-02   6.72507352e-02   5.10280463e-02   2.18879172e+00
-1.72283734e+01   3.62985243e+00   2.13933641e-03  -1.36531300e+00
2.88788067e-01  -1.22618657e-02  -8.36014969e-01   9.53058061e-03
-5.05036163e-01]
Variance score: 0.720898784611
```

In this example, the Explained Variance Score is used to determine the accuracy score.

We defined:

explained_variance_score = 1 – Var{y – y'}/Var{y}

In this, the estimated target output is y', the correct target output is y, and the

variance is VAR, which is the standard deviation square. 1.0 is the best score, while anything lower is worse.

Assumptions

These are the assumptions made by a linear regression model on the dataset it is applied to:

- **Linear relationship –** there should be a linear relationship between the response and the feature variables. Scatter plots can be used to test the linearity assumption.
- **Little to no multicollinearity** – this occurs when the independent variables are not independent of one another.
- **Little to no auto - correlation** – this occurs when there is no independence between the residual errors.
- **Homoscedasticity** – this is when the error term is the same across all the independent variables. This is defined as the random disturbance or noise in the relationship between the independent and dependent variables.

Logistic Regression Machine Learning Algorithm

Logistic regression is another machine learning technique from the field of statistics and is the go-to for solving binary classification, where there are two class values. It is named after the function at the very center of the method, known as the logistic function.

You may also know it as the sigmoid function. It was developed to describe the population growth properties in ecology, quickly rising and maxing out at the environment's carrying capacity. An S curve, it takes numbers (real values) and maps them to values between but not exactly at 0 and 1.

1 / (1 + e^-value)

E is the natural logarithm base, while value is the numerical value you want to be transformed. Let's see how the logistic function is used in logistic regression.

Representation

The representation used by logistic regression is an equation similar to linear regression. Input values, x, are linearly combined using coefficient values or weights to predict y, the output value. It differs from linear regression in that the modeled output value is binary, i.e., 0 or 1, and not a numeric value.

Here’s an example of a logistic regression algorithm:

*y = e^(b0 + b1*x) / (1 + e^(b0 + b1*x))*

The predicted output is y, the intercept or bias is b0, and the coefficient for x, the input value, is b1. There is a b coefficient associated with each input data column. This coefficient is a constant real value that the model learns from the training data.

The coefficients you see in the equation are the actual model representation stored in a file or in memory.

Probability Prediction

Logistic regression is used to model the default class's probability. For example, let's say we are modeling sex (male or female) based on a person's height. In that case, the default class might be male, and the model is written for the probability of a male, given a certain height. More formally, that would be:

P(sex=male|height)

We could write this another way and say that the model is for the probability of X, an input belonging to Y=1, the default class. This could be written formally as:

P(X) = P(Y=1|X)

Be aware that the prediction needs to be transformed to binary values, 0 or 1, actually to make the prediction.

Logistic regression may be linear, but the logistic function is used to transform the predictions. This means that the predictions can no longer be understood as linear combinations of inputs like we do with linear regression. Continuing from earlier, we can state this model as:

*p(X) = e^(b0 + b1*X) / (1 + e^(b0 + b1*X))*

Without going too deeply into the math, the above equation can be turned around as follows, not forgetting that e can be removed from one side with the addition of ln, a natural logarithm, on the other side:

*ln(p(X) / 1 – p(X)) = b0 + b1 * X*

This is useful because it allows us to see that the output calculation on the right side is linear, while the left-side input is a log of the default class's probability.

The right-side ratio is known as the default class's odds rather than probability. We calculate odds as a ratio of the event's probability, divided by the probability

of the event not happening, for example, 0.8/(1-0.8). The odds of this are 4, so we could write the following instead:

*ln(odds) = b0 + b1 * X*

As the odds are log-transformed, the left-side is called the probit or log-odds.

The exponent can be shifted to the right again and written as:

*odds = e^(b0 + b1 * X)*

This helps us see that our model is still linear in its output combination, but the combination relates to the default class's log-odds.

A Logistic Regression Model

You should use your training data to estimate the logistic regression algorithm's coefficients (Beta values b). This is done with the maximum-likelihood estimation. This is one of the more common learning algorithms that many other algorithms use, but it is worth noting that it makes assumptions about your data distribution.

The very best coefficients would give us a model that predicts values as close as possible to 1 for our default class and as close as possible to 0 for the other class. In this case, the maximum-likelihood's intuition for the logistic regression is that a search procedure seeks the coefficient's values that minimize the error in the predicted probabilities to the data. For example, if the data is the primary class, we get a probability of 1.

The math behind maximum-likelihood is out of the scope of this book. Still, it is sufficient to say that minimization algorithms are used to optimize the coefficient's best values for the training data.

Making Predictions with Logistic Regression

This is as easy as inputting the numbers to the equation and calculating your result. Here's an example to make this clearer.

Let's assume we have a model that uses height to predict whether a person is male or female – we'll use a base height of 150cm.

The coefficients have been learned – b0 = -100 and b1 = 1.6. Using the above equation, the probability is calculated that we have a male given a height of 150cm. More formally, this would be

P(male|height=150)

EXP() is used for e:

*y = e^(b0 + b1*X) / (1 + e^(b0 + b1*X))*

*y = exp(-100 + 0.6*150) / (1 + EXP(-100 + 0.6*X))*

y = 0.0000453978687

This gives us a near-zero probability of a person being male.

In practice, the probabilities can be used directly. However, we want a solid answer from this classification problem, so the probabilities can be snapped to binary class values, i.e.

0 if p(male) < 0.5

1 if p(male) >= 0.5

The next step is to prepare the data.

Prepare Data for Logistic Regression

Logistic regression makes assumptions about data distribution and relationships which are similar to those made by linear regression. Much work has gone into the definition of these assumptions, and they are written in precise statistical and probabilistic language. My advice? Use these as rules of thumb or guidelines, and don't be afraid to experiment with preparation schemes.

Here’s an example of logistic regression in Python code:

We’ll use the Pima Indian Diabetes dataset for this, which you can download from here . First, we load the data using the read CSV function in Pandas:

```
#import pandas

import pandas as pd

col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi', 'pedigree',
'age', 'label']

# load dataset

pima = pd.read_csv("pima-indians-diabetes.csv", header=None,
names=col_names)

pima.head()
```

Next, we want to select the features. The columns can be divided into two – the dependent variable (the target) and the independent variables (the features.)

```
#split dataset in features and target variable

feature_cols = ['pregnant', 'insulin', 'bmi', 'age','glucose','bp','pedigree']

X = pima[feature_cols] # Features

y = pima.label # Target variable
```

The dataset must be divided into a training and test dataset. We do that with the train_test_split function, passing three parameters – features, target, test_set_size. You can also use random_state to choose records randomly:

```
# split X and y into training and testing sets

from sklearn.cross_validation import train_test_split

X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.25,random_s

/home/admin/.local/lib/python3.5/site-
packages/sklearn/cross_validation.py:41: DeprecationWarning: This
module was deprecated in version 0.18 in favor of the model_selection
module into which all the refactored classes and functions are moved.
Also, note that the interface of the new CV iterators is different from that
of this module. This module will be removed in 0.20.

  "This module will be removed in 0.20.", DeprecationWarning)
```

We used a ratio of 75:25 to break the dataset – 75% training data and 25% testing data.

Developing the Model and Making the Prediction

First, import the logistic regression model and use the LogisticRegression() function to create a classifier object. Then you can fit the model using fit() on the training set and use predict() on the test set to make the predictions:

```
# import the class

from sklearn.linear_model import LogisticRegression


# instantiate the model (using the default parameters)

logreg = LogisticRegression()
```

```
# fit the model with data
logreg.fit(X_train,y_train)


#
y_pred=logreg.predict(X_test)
```

Using the Confusion Matrix to Evaluate the Model

A confusion matrix is a table we use when we want the performance of our classification model evaluated. You can also use it to visualize an algorithm's performance. A confusion matrix shows a class-wise summing up of the correct and incorrect predictions:

```
# import the metrics class
from sklearn import metrics
cnf_matrix = metrics.confusion_matrix(y_test, y_pred)
cnf_matrix
array([[119,  11],
       [ 26,  36]])
```

The result is a confusion matrix as an array object, and the matrix dimension is 2*2 because this was a binary classification. There are two classes, 0 and 1. Accurate predictions are represented by the diagonal values, while the non-diagonals represent the inaccurate predictions. The output, in this case, is 119 and 36 as the actual predictions and 26 and 11 as the inaccurate predictions.

Using Heatmap to Visualize the Confusion Matrix

This uses Seaborn and Matplotlib to visualize the results in the confusion matrix – we'll be using Heatmap:

```
# import required modules
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

```
%matplotlib inline
class_names=[0,1] # name  of classes
fig, ax = plt.subplots()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names)
plt.yticks(tick_marks, class_names)
# create heatmap
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu"
,fmt='g')
ax.xaxis.set_label_position("top")
plt.tight_layout()
plt.title('Confusion matrix', y=1.1)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
Text(0.5,257.44,'Predicted label')
```

Evaluation Metrics

Now we can evaluate the model using precision, recall, and accuracy evaluation metrics:

```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
print("Precision:",metrics.precision_score(y_test, y_pred))
print("Recall:",metrics.recall_score(y_test, y_pred))
```

The result is:

```
Accuracy: 0.8072916666666666
Precision: 0.7659574468085106
Recall: 0.5806451612903226
```

We got a classification accuracy of 80%, which is considered pretty good.

The precision metric is about precision, as you would expect. When your model makes predictions, you want to know how often it makes the right one. In our

case, the model made a 76% prediction accuracy that people would have diabetes.

In terms of recall, the model can correctly identify those with diabetes in the test set 58% of the time.

ROC Curve

The ROC or Receiver Operating Characteristic curve plots the true against the false positive rate and shows the tradeoff between specificity and sensitivity.

```
y_pred_proba = logreg.predict_proba(X_test)[::,1]
fpr, tpr, _ = metrics.roc_curve(y_test,  y_pred_proba)
auc = metrics.roc_auc_score(y_test, y_pred_proba)
plt.plot(fpr,tpr,label="data 1, auc="+str(auc))
plt.legend(loc=4)
plt.show()
```

The AUC score is 0.86. A score of 0.5 indicates a poor classifier, while a score of 1 indicates a perfect classifier. 0.86 is as near to perfect as we can get.

Your ultimate focus in predictive modeling is making predictions that are as accurate as possible, rather than analyzing the results. As such, some assumptions can be broken, so long as you have a strong model that performs well:

- **Binary Output Variable** : you might think this is obvious, given that we mentioned it earlier. Logistic regression is designed for binary classification, and it will predict probabilities of instances belonging to the first or default class and snapping them to a 0 or 1 classification.
- **Removing Noise** : one assumption is that the output variable, y, has no error. You should remove outliers and potentially misclassified data from your training data right from the start.
- **Gaussian Distribution** : because logistic regression is linear but with the output being transformed non-linearly, an assumption is made of a linear relationship between the input and output variables. You get a more accurate model when you transform the input variables to expose the relationship better. For example, univariate transforms, such as Box-cox, log, and root, can do this.

- **Removing Correlated Inputs** : in the same way as linear regression, your model can end up overfitting if you have too many highly correlated inputs. Think about removing the highly correlated inputs and calculating pairwise correlations.
- **Failure to Converge** : it may happen that the likelihood estimation learning the coefficients doesn't converge. This could be because of too many highly correlated inputs in the data or sparse data, which means the input data contains many zeros.

Decision Tree Machine Learning Algorithm

The decision tree is a popular algorithm, undoubtedly one of the most popular, and it comes under supervised learning. It is used successfully with categorical and continuous output variables.

To show you how the decision tree algorithm works, we're going to walk through implementing one on the UCI database, Balance Scale Weight & Distance. The database contains 625 instances – 49 balanced and 288 each to the left and right. There are four numeric attributes and the class name.

Python Packages

You need three Python packages for this:

- **Sklearn –** a popular ML package containing many algorithms, and we'll be using some popular modules, such as train_test_split, accuracy_score, and DecisionTreeClassifier.
- **NumPy –** a numeric module containing useful math functions, NumPy is used to manipulate data and read data in NumPy arrays.
- **Pandas –** commonly used to read/write files and uses DataFrames to manipulate data.

Sklearn is where all the packages are that help us implement the algorithm, and this is installed with the following commands:

using pip :

pip install -U scikit-learn

Before you dive into this, ensure that NumPy and Scipy are installed and that you have pip installed too. If pip is not installed, use the following commands to get it:

```
python get-pip.py
```

If you are using conda, use this command:

```
conda install scikit-learn
```

Decision Tree Assumptions

The following are the assumptions made for this algorithm:

- The entire training set is considered as the root at the start of the process.
- It is assumed that the attributes are categorical for information gain and continuous for the Gini index.
- Records are recursively distributed, based on attribute values
- Statistical methods are used to order the attributes as internal node or root.

Pseudocode:

- The best attribute must be found and placed on the tree's root node
- The training dataset needs to be divided into subsets, ensuring that each one has the same attribute value
- Leaf nodes must be found in every branch by repeating steps one and two on every subset

The implementation of the algorithm requires the following two phases:

- **The Building Phase –** the dataset is preprocessed, then sklearn is used to split the dataset into test and train sets before training the classifier.
- **The Operational Phase –** predictions are made and the accuracy calculated

Importing the data

Data import and manipulation are done using the Pandas package. The URL we use fetches the dataset directly from the UCI website, so you don't need to download it at the start. That means you will need a good internet connection when you run the code. Because "," is used to separate the dataset, we need to pass that as the sep value parameter. Lastly, because there is no header in the dataset, the Header's parameter must be passed as none. If no header parameter is passed, the first line in the dataset will be used as the Header.

Slicing the Data

Before we train our model, the dataset must be split into two – the training and testing sets. To do this, the train_test_split module from sklearn is used. First, the

target variable must be split from the dataset attributes:

```
X = balance_data.values[:, 1:5]

Y = balance_data.values[:,0]
```

Those two lines of code will separate the dataset. X and Y are both variables – X stores the attributes, and Y stores the dataset's target variable. Next, we need to split the dataset into the training and testing sets:

```
X_train, X_test, y_train, y_test = train_test_split(

X, Y, test_size = 0.3, random_state = 100)
```

These code lines split the dataset into a ratio of 70:30 – 70% for training and 30% for testing. The parameter value for test_size is passed as 0.3. The variable called random_state is a pseudo-random number generator typically used in random sampling cases.

Code Terminology:

Both the information gain and Gini Index methods select the right attribute to place at the internal node or root node.

Gini Index:

- This is a metric used to measure how often elements chosen at random would be identified wrongly.
- Attributes with lower Gini Indexes are preferred
- Sklearn provides support for the "gini" criteria and takes "gini" value by default.

Entropy:

- This is a measure of a random variable's uncertainty, characterizing impurity in an arbitrary example collection.
- The higher entropy is, the more information it contains.

Information Gain:

- Typically, entropy will change when a node is used to divide training instances in a decision tree into smaller sets. Information gain is one measure of change.
- Sklearn has support for the Information Gain's "entropy" criteria. If we need the Information Gain method from sklearn, it must be explicitly mentioned.

Accuracy Score:

- This is used to calculate the trained classifier's accuracy score.

Confusion Matrix:

- This is used to help understand how the trained classifier behaves over the test dataset or help validate the dataset.

Let's get into the code:

```
# Run this program on your local python
# interpreter, provided you have installed
# the required libraries.

# Importing the required packages
import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix
from sklearn.cross_validation import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report

# Function importing Dataset
def importdata():
    balance_data = pd.read_csv(
'https://archive.ics.uci.edu/ml/machine-learning-'+
'databases/balance-scale/balance-scale.data',
    sep= ',', header = None)

    # Printing the dataswet shape
```

```
    print ("Dataset Length: ", len(balance_data))
    print ("Dataset Shape: ", balance_data.shape)

    # Printing the dataset obseravtions
    print ("Dataset: ",balance_data.head())
    return balance_data

# Function to split the dataset
def splitdataset(balance_data):

    # Separating the target variable
    X = balance_data.values[:, 1:5]
    Y = balance_data.values[:, 0]

    # Splitting the dataset into train and test
    X_train, X_test, y_train, y_test = train_test_split(
    X, Y, test_size = 0.3, random_state = 100)

    return X, Y, X_train, X_test, y_train, y_test

# Function to perform training with giniIndex.
def train_using_gini(X_train, X_test, y_train):

    # Creating the classifier object
    clf_gini = DecisionTreeClassifier(criterion = "gini",
            random_state = 100,max_depth=3, min_samples_leaf=5)
```

```
    # Performing training
    clf_gini.fit(X_train, y_train)
    return clf_gini

# Function to perform training with entropy.
def tarin_using_entropy(X_train, X_test, y_train):

    # Decision tree with entropy
    clf_entropy = DecisionTreeClassifier(
            criterion = "entropy", random_state = 100,
            max_depth = 3, min_samples_leaf = 5)

    # Performing training
    clf_entropy.fit(X_train, y_train)
    return clf_entropy



# Function to make predictions
def prediction(X_test, clf_object):

    # Prediction on test with giniIndex
    y_pred = clf_object.predict(X_test)
    print("Predicted values:")
    print(y_pred)
    return y_pred

# Function to calculate accuracy
```

```
def cal_accuracy(y_test, y_pred):

    print("Confusion Matrix: ",
        confusion_matrix(y_test, y_pred))

    print ("Accuracy : ",
    accuracy_score(y_test,y_pred)*100)

    print("Report : ",
    classification_report(y_test, y_pred))

# Driver code
def main():

    # Building Phase
    data = importdata()
    X, Y, X_train, X_test, y_train, y_test = splitdataset(data)
    clf_gini = train_using_gini(X_train, X_test, y_train)
    clf_entropy = tarin_using_entropy(X_train, X_test, y_train)

    # Operational Phase
    print("Results Using Gini Index:")

    # Prediction using gini
    y_pred_gini = prediction(X_test, clf_gini)
    cal_accuracy(y_test, y_pred_gini)
```

```
    print("Results Using Entropy:")
    # Prediction using entropy
    y_pred_entropy = prediction(X_test, clf_entropy)
    cal_accuracy(y_test, y_pred_entropy)


# Calling main function
if __name__=="__main__":
    main()
```

Here’s the data information:

```
Dataset Length:  625
Dataset Shape: (625, 5)
Dataset:     0  1  2  3  4
0  B  1  1  1  1
1  R  1  1  1  2
2  R  1  1  1  3
3  R  1  1  1  4
4  R  1  1  1  5
```

Using the Gini Index, the results are:

Predicted values:

```
['R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'R' 'L'
'L' 'R' 'L' 'R' 'L' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'L'
'L' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'R' 'L' 'R'
'R' 'L' 'R' 'R' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'R'
'R' 'L' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'R' 'L'
'R' 'R' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'R'
'R' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L'
```

'L' 'L' 'L' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R'

'L' 'L' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R'

'L' 'L' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'R' 'R'

'L' 'R' 'R' 'L' 'L' 'R' 'R' 'R']

The confusion matrix is:

[[0 6 7]

[0 67 18]

[0 19 71]]

The accuracy is:

73.4042553191

The report:

	precision	recall	f1-score	support
B	0.00	0.00	0.00	13
L	0.73	0.79	0.76	85
R	0.74	0.79	0.76	90
avg/total	0.68	0.73	0.71	188

Using Entropy, the result are:

Predicted values:

['R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'L'

'L' 'R' 'L' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'L' 'L'

'L' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'L' 'R' 'L' 'L' 'R' 'L' 'L'

'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'L' 'R' 'L' 'L' 'L' 'R'

'R' 'L' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'R' 'L'

'R' 'R' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'R' 'R' 'R' 'R' 'R'

'R' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L'

'L' 'L' 'L' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R'

'L' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R'

'R' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'L' 'R'

'R' 'R' 'L' 'L' 'L' 'R' 'R' 'R']

The confusion matrix is:

[[0 6 7]

[0 63 22]

[0 20 70]]

The accuracy is

70.7446808511

And the report is:

	precision	recall	f1-score	support
B	0.00	0.00	0.00	13
L	0.71	0.74	0.72	85
R	0.71	0.78	0.74	90
avg / total	0.66	0.71	0.68	188

Random Forest Machine Learning Algorithm

The random forest classifier is another supervised machine learning algorithm that can be used for regression, classification, and other decision tree tasks. The random forest creates sets of decision trees from random training data subsets. It then makes its decision using votes from the trees.

To demonstrate how this works, we'll use the well-known iris flower dataset to train the model and test it. The model will be built to determine the flower classifications.

First, we load the dataset:

```
# importing required libraries

# importing Scikit-learn library and datasets package

from sklearn import datasets


# Loading the iris plants dataset (classification)
```

iris = datasets.load_iris()

Code: checking our dataset content and features names present in it.

print(iris.target_names)

Output:

[‘setosa’ ‘versicolor’ ‘virginica’]

Code:

print(iris.feature_names)

Output:

[‘sepal length (cm)’, ’sepal width (cm)’, ’petal length (cm)’, ’petal width (cm)’]

Code:

dividing the datasets into two parts i.e. training datasets and test datasets

X, y = datasets.load_iris(return_X_y = True)

Splitting arrays or matrices into random train and test subsets

from sklearn.model_selection import train_test_split

i.e. 80 % training dataset and 30 % test datasets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.70)

Code: Importing required libraries and random forest classifier module.

importing random forest classifier from assemble module

from sklearn.ensemble import RandomForestClassifier

import pandas as pd

creating dataframe of IRIS dataset

data = pd.DataFrame({‘sepallength’: iris.data[:, 0], ’sepalwidth’: iris.data[:, 1],

```
'petallength': iris.data[:, 2], 'petalwidth': iris.data[:, 3],
'species': iris.target})
```

Code: Looking at a dataset

```
# printing the top 5 datasets in iris dataset
print(data.head())
```

The output of this is:

	sepallength	sepalwidth	petallength	petalwidth	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

Next, we create the Random Forest Classifier, train it and then test it:

```
# creating a RF classifier
clf = RandomForestClassifier(n_estimators = 100)

# Training the model on the training dataset
# fit function is used to train the model using the training sets as
parameters
clf.fit(X_train, y_train)

# performing predictions on the test dataset
```

```
y_pred = clf.predict(X_test)


# metrics are used to find accuracy or error
from sklearn import metrics
print()


# using metrics module for accuracy calculation
print("ACCURACY OF THE MODEL: ", metrics.accuracy_score(y_test, y_pred))
```

Output:

ACCURACY OF THE MODEL: 0.9238095238095239

Code: predicting the type of flower from the data set

```
# predicting which type of flower it is.
clf.predict([[3, 3, 2, 2]])
```

And the output is:

```
array([0])
```

The result tells us the flower classification is the setosa type. Note that the dataset contains three types – Setosa, Vericolor, and Virginia.

Next, we want to find the important features in the dataset:

```
# importing random forest classifier from assemble module
from sklearn.ensemble import RandomForestClassifier
# Create a Random forest Classifier
clf = RandomForestClassifier(n_estimators = 100)
```

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

Code: Calculating feature importance

```
# using the feature importance variable
import pandas as pd
feature_imp = pd.Series(clf.feature_importances_, index =
iris.feature_names).sort_values(ascending = False)
feature_imp
```

The output is:

```
petal width (cm)     0.458607
petal length (cm)    0.413859
sepal length (cm)    0.103600
sepal width (cm)     0.023933
dtype: float64
```

Each decision tree created by the random forest algorithm has a high variance. However, when they are all combined in parallel, the variance is low. That's because the individual trees are trained perfectly on the specified sample data, and the output is not dependent on just one tree but all of them. In classification problems, a majority voting classifier is used to get the final output. In regression problems, the output is the result of aggregation – the mean of all outputs.

Random forest uses the aggregation and bootstrap technique, also called bagging. The idea is to combine several decision trees to determine the output rather than relying on the individual ones.

The bootstrap part of the technique is where random row and feature sampling forms the sample datasets for each model. The regression technique must be approached as we would any ML technique:

- A specific data or question is designed, and the source is obtained to work out what data we need

- The data must be in an accessible format, or it must be converted to an accessible format
- All noticeable anomalies must be specified, along with missing data points that might be needed to get the right data
- The machine learning model is created, and a baseline model set that you want to be achieved
- The machine learning model is trained on the data
- The test data is used to test the model
- The performance metrics from the test and predicted data are compared
- If the model doesn't do what it should, work on improving the model, changing the data, or using a different modeling technique.
- At the end, the data is interpreted and reported.

Here's another example of how random forest works.

First, the libraries are imported:

```
# Importing the libraries

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd
```

Next, the dataset is imported and printed:

```
data = pd.read_csv('Salaries.csv')

print(data)
```

Then all the rows and column 1 are selected from the dataset to x and every row and column 2 as y :

```
x = data.iloc[:, 1:2].values

print(x)

y = data.iloc[:, 2].values
```

The random forest aggressor is fitted to the dataset:

```
# Fitting Random Forest Regression to the dataset

# import the regressor

from sklearn.ensemble import RandomForestRegressor
```

```
# create regressor object
regressor = RandomForestRegressor(n_estimators = 100, random_state =
0)


# fit the regressor with x and y data
regressor.fit(x, y)
```

The result is predicted:

```
Y_pred = regressor.predict(np.array([6.5]).reshape(1, 1))  # test the output
by changing values
```

The result is visualized:

```
# Visualizing the Random Forest Regression results


# arange for creating a range of values
# from min value of x to max
# value of x with a difference of 0.01
# between two consecutive values
X_grid = np.arange(min(x), max(x), 0.01)


# reshape for reshaping the data into a len(X_grid)*1 array,
# i.e. to make a column out of the X_grid value
X_grid = X_grid.reshape((len(X_grid), 1))


# Scatter plot for original data
plt.scatter(x, y, color = 'blue')


# plot predicted data
```

```
plt.plot(X_grid, regressor.predict(X_grid),
        color = 'green')
plt.title('Random Forest Regression')
plt.xlabel('Position level')
plt.ylabel('Salary')
plt.show()
```

Artificial Neural Networks Machine Learning Algorithm

Artificial Neural Networks, or ANNs, are paradigms to process information based on how the human brain works. Like a human, an ANN learns by example. They are configured for particular applications, like data classification or pattern recognition, through a learning process that mostly involves adjusting the connections between the neurons.

The human brain is filled with billions and billions of neurons connected to one another via synapses. These are the connections by which neurons send impulses to other neurons. When one neuron sends a signal to another, the signal gets added to the neuron's inputs. Once a specified threshold is exceeded, the target neuron fires a signal forward. That is pretty much how the internal thinking process happens.

In data science, this process is modeled by networks created on computers using matrices. We can understand these networks as neuron abstractions with none of the biological complexities. We could get very complex here, but we won't. We'll create a simple neural network that has two layers and can solve a linear classification problem

Let's say we want to predict the output, using specified inputs and outputs as training examples.

The training process is:

1. **Forward Propagation**

The inputs are multiplied by the weights:

Let Y = WiIi = W1I1+W2I2+W3I3

The result is passed through a sigmoid formula so the neuron's output can be calculated. This function normalizes the result so it falls between 0 and 1.

1/1 + e-y

2. **Back Propagation**

The error is calculated. This is the difference between the actual and expected output. Depending on what the error is, the weights can be adjusted. To do this, multiply the error by the input and then do it again with sigmoid curve gradient:

Weight += Error Input Output (1-Output) ,here Output (1-Output) is derivative of sigmoid curve.

The entire process must be repeated for however many iterations there are, potentially thousands.

Let's see what the whole thing looks like coded in Python, using NumPy to do the matrices calculations. Make sure NumPy is already installed and then go ahead with the following code:

```
from joblib.numpy_pickle_utils import xrange
from numpy import *



class NeuralNet(object):
    def __init__(self):
        # Generate random numbers
        random.seed(1)


        # Assign random weights to a 3 x 1 matrix,
        self.synaptic_weights = 2 * random.random((3, 1)) - 1


    # The Sigmoid function
    def __sigmoid(self, x):
        return 1 / (1 + exp(-x))
```

```
    # The derivative of the Sigmoid function.
    # This is the gradient of the Sigmoid curve.
    def __sigmoid_derivative(self, x):
        return x * (1 - x)

    # Train the neural network and adjust the weights each time.
    def train(self, inputs, outputs, training_iterations):
        for iteration in xrange(training_iterations):
            # Pass the training set through the network.
            output = self.learn(inputs)

            # Calculate the error
            error = outputs - output

            # Adjust the weights by a factor
            factor = dot(inputs.T, error * self.__sigmoid_derivative(output))
            self.synaptic_weights += factor

        # The neural network thinks.

    def learn(self, inputs):
        return self.__sigmoid(dot(inputs, self.synaptic_weights))


if __name__ == "__main__":
    # Initialize
    neural_network = NeuralNet()
```

```
# The training set.
inputs = array([[0, 1, 1], [1, 0, 0], [1, 0, 1]])
outputs = array([[1, 0, 1]]).T

# Train the neural network
neural_network.train(inputs, outputs, 10000)

# Test the neural network with a test example.
print(neural_network.learn(array([1, 0, 1])))
```

The Output

Once 10 iterations have been completed, the NN should predict a value of 0.6598.921. This isn't what we want as the answer should be 1. If the number of iterations is increased to 100, the output is 0.87680541. That means the network is improving the more iterations it does. So, for 10,000 iterations, we get as near as we can with an output of 0.9897704.

Machine Learning Steps

If only machine learning were as simple as just applying an algorithm to the data and obtaining the predictions. Sadly, it isn’t, and it involves many more steps for each project you do:

Step One – Gather the Data

This is the first and most important step in machine learning and is by far the most time-consuming. It involves collecting all the data needed to solve the problem and enough of it. For example, let’s say we wanted to build a model that predicted house prices. In that case, we need a dataset with information about previous sales, which we can use to form a tabular structure. We’ll solve something similar when we show you how to implement a regression problem later.

Step Two – Preparing the Data

When we have the data, it needs to be put into the right format and pre-

processed. Pre-processing involves different steps, including cleaning the data. Should there be abnormal values, such as strings instead of numbers, or empty values, you need to understand how you will deal with them. The simplest way is to drop those rows containing empty values, although there are a few other ways you can do it. Sometimes, you may also have columns that will have no bearing on the results; those columns can also be removed. Usually, data visualization will be done to see the data in the form of charts, graphs, and diagrams. Once these have been analyzed, we can determine the important features.

Step Three – Choosing Your Model

Now the data is ready, and we need a machine-learning algorithm to feed it to. Some people use "machine learning algorithm" interchangeably with "machine learning model," but they are not the same. The model is the algorithm's output. When the algorithm is implemented on the data, we get an output containing the numbers, rules, and other data structures related to the algorithm and needed to make the predictions. For example, we would get an equation showing the best-fit line if we did linear regression on the data. That equation is the model. If we didn't go down the route of tuning the hyperparameters, your next step would be training the model.

Step Four – Tuning the Hyperparameters

Hyperparameters are critical because they are responsible for controlling how a machine learning model behaves. The goal is to find the best hyperparameter combination to give us the best results. But what are they? Take the K-NN algorithm and the K variable. When K is given different values, the results are predictably different. K's best value isn't defined, and each dataset will have its own value. There is also no easy way of knowing what the best values are; all you can do is try different ones and see what results you get. In this case, K is a hyperparameter, and its values need to be tuned to get the right results.

Step Five – Evaluating the Model

Next, we want to know how well the model has performed. The best way to know that is to test it on more data. This is the testing data, and it must be completely different from the data the algorithm was trained on. When you train a model, you do not intend it to learn the values in the training data. Instead, you want it to learn the data's underlying pattern and use that to

make predictions on previously unseen data. We'll discuss this more in the next section.

Step Six – Making Predictions

The final step is to use the model on real-world problems in the hopes that it will work as intended.

Evaluating a Machine Learning Model

Training a machine learning model is an important step in machine learning. Equally important is evaluating how the model works on previously unseen data and is something you should consider with every pipeline you build. You need to know if your model works and, if it does, whether its predictions can be trusted or not. Is the model memorizing the data you feed it, or is it learning by example? If it is only memorizing data, its predictions on unseen data will be next to useless.

We're going to take some time out to look at some of the techniques you can use to evaluate your model to see how well it works. We'll also look at some of the common evaluation metrics for regression and classification methods with Python.

Model Evaluation Techniques

Handling how your machine learning model performs requires evaluation, an integral part of any data science project you are involved in. When you evaluate the model, you aim to estimate how accurate the model's generalization is on unseen data.

Model evaluation methods fall into two distinct categories – cross-validation and holdout. Both use test sets, which are datasets of completely new, previously unseen data, to evaluate the performance. As you already know by now, using the same data you train a model on to test it is not recommended because the model will not learn anything – it will simply memorize what you trained it on. The predictions will always be correct anywhere in the training set. This is called overfitting.

Holdout

Holdout evaluation tests models on different data to the training data, providing an unbiased learning performance estimate. The dataset will be divided into three random subsets:

- **The training set –** this is a subset of the full dataset using in predictive model building
- **The validation set –** this subset of the dataset is used to assess the model's performance. It gives us a test platform that allows us to fine-tune the parameters for our model and choose the model that performs the best. Be aware that you won't need a validation set for every single algorithm.
- **The test set –** this is the unseen data, a subset of the primary dataset used to tell us how the model may perform in the future. If you find the model fitting the training set better than it does the test set, you probably have a case of overfitting on your hands.

The holdout technique is useful because it is simple, flexible, and fast. However, it does come with an association to high variance. This is because the differences between the training and set datasets can provide significant differences in terms of the accuracy estimation.

Cross-Validation

The cross-validation technique will partition the primary dataset into two – a training set to train the model and another independent set for evaluating the analysis. Perhaps the best-known cross-validation technique is k-fold, which partitions the primary dataset into k number of subsamples, equal in size. Each subsample is called a fold, and the k defines a specified number – typically, 5 or 10 are the preferred options. This process is repeated k number of times, and each time a k subset is used to validate, while the remaining subsets are combined into a training set. The model effectiveness is drawn from an average estimation of all the k trials.

For example, when you perform ten-fold cross-validation, the data is split into ten parts, roughly equal in size. Models are then trained in sequence – the first fold is used as the test set, while the other nine are used for training. This is repeated for each part, and accuracy estimation is averaged from all ten trials to see how effective the model is.

Each data point is used as the test set once and is part of the training set k number of times, in our case, nine. The reduces the bias significantly as almost all the data is used for fitting. It also reduces variance significantly because almost all the data is used in the test set. This method's effectiveness is increased because the test and training sets are interchanged.

Model Evaluation Metrics

Quantifying the model performance requires the use of evaluation metrics, and which ones you use depend on what kind of task you are doing – regression, classification, clustering, and so on. Some metrics are good for multiple tasks, such as precision-recall. Classification, regression, and other supervised learning tasks make up the largest number of machine learning applications, so we will concentrate on regression and classification

Classification Metrics

We'll take a quick look at the most common metrics for classification problems, including:

- Classification Accuracy
- Confusion matrix
- Logarithmic Loss
- AUC – Area Under Curve
- F-Measure

Classification Accuracy

This is one of the more common metrics used in classification and indicates the correct predictions as a ratio of all predictions. The sklearn module is used for computing classification accuracy, as in the example below:

```
#import modules
import warnings
import pandas as pd
import numpy as np
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.metrics import accuracy_score
#ignore warnings
warnings.filterwarnings('ignore')
```

```
# Load digits dataset
iris = datasets.load_iris()
# # Create feature matrix
X = iris.data
# Create target vector
y = iris.target
#test size
test_size = 0.33
#generate the same set of random numbers
seed = 7
#cross-validation settings
kfold = model_selection.KFold(n_splits=10, random_state=seed)
#Model instance
model = LogisticRegression()
#Evaluate model performance
scoring = 'accuracy'
results = model_selection.cross_val_score(model, X, y, cv=kfold, scoring=scoring)
print('Accuracy -val set: %.2f%% (%.2f)' % (results.mean()*100, results.std()))


#split data
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=test_size, random_state=seed)
#fit model
model.fit(X_train, y_train)
#accuracy on test set
result = model.score(X_test, y_test)
```

```
print("Accuracy - test set: %.2f%%" % (result*100.0))
```

In this case, the accuracy is 88%.

Cross-validation allows us to test the model while training, checking for overfitting, and looking to see how the model will generalize on the test data. Cross-validation can also compare how different models perform on the same data set and helps us choose the best parameter values to ensure the maximum possible model accuracy. This is also known as parameter tuning.

Confusion Matrix

Confusion matrices provide more detail in the breakdown of right and wrong classifications for each individual class. In this case, we will use the Iris dataset for classification and computation of the confusion matrix:

```
#import modules

import warnings

import pandas as pd

from sklearn import model_selection

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import confusion_matrix

import matplotlib.pyplot as plt

%matplotlib inline


#ignore warnings

warnings.filterwarnings('ignore')

# Load digits dataset

url = "http://archive.ics.uci.edu/ml/machine-learning-
databases/iris/iris.data"

df = pd.read_csv(url)

# df = df.values

X = df.iloc[:,0:4]

y = df.iloc[:,4]
```

```
# print (y.unique())
#test size
test_size = 0.33
#generate the same set of random numbers
seed = 7
#Split data into train and test set.
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
test_size=test_size, random_state=seed)
#Train Model
model = LogisticRegression()
model.fit(X_train, y_train)
pred = model.predict(X_test)

#Construct the Confusion Matrix
labels = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
cm = confusion_matrix(y_test, pred, labels)
print(cm)
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(cm)
plt.title('Confusion matrix of the classifier')
fig.colorbar(cax)
ax.set_xticklabels([''] + labels)
ax.set_yticklabels([''] + labels)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

The short way of explaining how a confusion matrix should be interpreted is this:

The elements on the diagonal represent how many points for which the predicted label and true label are equal. Anything not on the diagonal has been mislabeled. The higher the values on the diagonal, the better because this indicates a higher number of correct predictions.

Our classifier was correct in predicting all 13 Setosa and 18 Virginica iris plants in the test data, but it was wrong in classifying four Versicolor plants as Virginica.

Logarithmic Loss

Logarithmic loss is also called logloss, and it is used to measure a classification model's performance where a probability value between 0 and 1 is used as the prediction input. With a divergence between the predicted probability and the actual label, we saw an increase in log loss. That value needs to be minimized by the machine learning model – the smaller the logloss, the better, and the best models have a log loss of 0.

```
#Classification LogLoss
import warnings
import pandas
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss


warnings.filterwarnings('ignore')
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv"
dataframe = pandas.read_csv(url)
dat = dataframe.values
X = dat[:,:-1]
y = dat[:,-1]
seed = 7
#split data
```

```
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
test_size=test_size, random_state=seed)
model.fit(X_train, y_train)
#predict and compute logloss
pred = model.predict(X_test)
accuracy = log_loss(y_test, pred)
print("Logloss: %.2f" % (accuracy))
```

Area Under Curve (AUC)

This performance metric measures how well binary classifiers discriminate between positive classes and negative ones.:

```
#Classification Area under curve
import warnings
import pandas
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score, roc_curve

warnings.filterwarnings('ignore')

url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-
indians-diabetes.data.csv"
dataframe = pandas.read_csv(url)
dat = dataframe.values
X = dat[:,:-1]
y = dat[:,-1]
seed = 7
#split data
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
```

```
test_size=test_size, random_state=seed)
model.fit(X_train, y_train)

# predict probabilities
probs = model.predict_proba(X_test)
# keep probabilities for the positive outcome only
probs = probs[:, 1]

auc = roc_auc_score(y_test, probs)
print('AUC - Test Set: %.2f%%' % (auc*100))

# calculate roc curve
fpr, tpr, thresholds = roc_curve(y_test, probs)
# plot no skill
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
plt.plot(fpr, tpr, marker='.')
# show the plot
plt.show()
```

In this example, we have an AUC greater than 0.5 and close to 1. The perfect classifier has a ROC curve along the y-axis and then along the x-axis.

F-Measure

Also called F-Score, this measures the accuracy of a test, using the test's recall and precision to come up with the score. Precision indicates correct positive results divided by the total number of predicted positive observations. Recall indicates correct positive results divided by the total actual positives or the total of all the relevant samples:

```
import warnings
```

```
import pandas
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss
from sklearn.metrics import precision_recall_fscore_support as score, precision_score, recall_score, f1_score


warnings.filterwarnings('ignore')


url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv"
dataframe = pandas.read_csv(url)
dat = dataframe.values
X = dat[:,:-1]
y = dat[:,-1]
test_size = 0.33
seed = 7


model = LogisticRegression()
#split data
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=test_size, random_state=seed)
model.fit(X_train, y_train)
precision = precision_score(y_test, pred)
print('Precision: %f' % precision)
# recall: tp / (tp + fn)
recall = recall_score(y_test, pred)
print('Recall: %f' % recall)
```

```
# f1: tp / (tp + fp + fn)
f1 = f1_score(y_test, pred)
print('F1 score: %f' % f1)
```

Regression Metrics

There are two primary metrics used to evaluate regression problems – Root Mean Squared Error and Mean Absolute Error.

Mean Absolute Error, also called MAE, tells us the sum of the absolute differences between the actual and predicted values. Root Mean Squared Error, also called RMSE, measures an average of the error magnitude by "taking the square root of the average of squared differences between prediction and actual observation."

Here’s how we implement both metrics:

```
import pandas
from sklearn import model_selection
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error
from math import sqrt
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/housing.data
dataframe = pandas.read_csv(url, delim_whitespace=True)
df = dataframe.values
X = df[:,:-1]
y = df[:,-1]
seed = 7
model = LinearRegression()
#split data
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
test_size=test_size, random_state=seed)
model.fit(X_train, y_train)
```

```
#predict
pred = model.predict(X_test)
print("MAE test score:", mean_absolute_error(y_test, pred))
print("RMSE test score:", sqrt(mean_squared_error(y_test, pred)))
```

In an ideal world, a model's estimated performance will tell us if the model performs well on unseen data or not. Most of the time, we will be building models to predict on future data and, before a metric is chosen, you must understand the context – each model will use different datasets and different objectives to solve the problem.

Implementing Machine Learning Algorithms with Python

By now, you should already have all the software and libraries you need on your machine – we will mostly be using Scikit-learn to solve two problems – regression and classification.

Implementing a Regression Problem

We want to solve the problem of predicting house prices given specific features, like the house size, how many rooms there are, etc.

- **Gather the Data –** we don't need to collect the required data manually because someone has already done it for us. All we need to do is import that dataset. It's worth noting that not all available datasets are free, but the ones we are using are.

We want the Boston Housing dataset, which contains records describing suburbs or towns in Boston. The data came from the Boston SMSA (Standard Metropolitan Statistical Area) in 1970, and the attributes have been defined as:

1. CRIM: the per capita crime rate in each town
2. ZN: the proportion of residential land zoned for lots of more than 25,000 sq.ft.
3. INDUS: the proportion of non-retail business acres in each town
4. CHAS: the Charles River dummy variable (= 1 if the tract bounds river; 0 if it doesn't)

5. NOX: the nitric oxide concentrations in parts per 10 million
6. RM: the average number of rooms in each property
7. AGE: the proportion of owner-occupied units built before 1940
8. DIS: the weighted distances to five employment centers in Boston
9. RAD: the index of accessibility to radial highways
10. TAX: the full-value property tax rate per $10,000
11. PTRATIO: the pupil-teacher ratio by town
12. B: 1000(Bk−0.63)2 where Bk is the proportion of black people in each town
13. LSTAT: % lower status of the population
14. MEDV: Median value of owner-occupied homes in $1000s

You can download the dataset here .

Download the file and open it to see the data on the house sales. Note that the dataset is not in tabular forms, and there are no names for the columns. Commas separate all the values.

The first step is to place the data in tabular form, and Pandas will help us do that. We give Pandas a list with all the column names and a delimiter of '\s+,' meaning that every entry can be differentiated when single or multiple spaces are encountered.

We'll start by importing the libraries we need and then the dataset in CSV format. All this is imported into a Pandas DataFrame.

```
import numpy as np

import pandas as pd

column_names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX','PTRATIO', 'B', 'LSTAT', 'MEDV']

bos1 = pd.read_csv('housing.csv', delimiter=r"\s+", names=column_names)
```

- **Preprocess the Data**

Next, the data must be pre-processed. When we look at the dataset, we can

immediately see that it doesn't contain any NaN values. We also see that the data is in numbers, not strings, so we don't need to worry about errors when we train the model.

We'll divide the data into two – 70% for training and 30% for testing.

```
bos1.isna().sum()


from sklearn.model_selection import train_test_split
X=np.array(bos1.iloc[:,0:13])
Y=np.array(bos1["MEDV"])
#testing data size is of 30% of entire data
x_train, x_test, y_train, y_test =train_test_split(X,Y, test_size = 0.30, random_state =5)
```

- **Choose Your Model**

To solve this problem, we want two supervised learning algorithms for solving regression. Later, the results from both will be compared. The first is K-NN and the second is linear regression, both of which were demonstrated earlier in this part.

```
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsRegressor
#load our first model
lr = LinearRegression()
#train the model on training data
lr.fit(x_train,y_train)
#predict the testing data so that we can later evaluate the model
pred_lr = lr.predict(x_test)
#load the second model
Nn=KNeighborsRegressor(3)
Nn.fit(x_train,y_train)
```

```
pred_Nn = Nn.predict(x_test)
```

- **Tune the Hyperparameters**

All we are going to do here is tune the ok K value in K-NN – we could do more, but this is just for starters, so I don't want you to get too overwhelmed. We'll use a for loop and check on the results from k, from 1 to 50. K-NN works fast on small datasets, so this won't take too much time.

```
import sklearn
for i in range(1,50):
   model=KNeighborsRegressor(i)
   model.fit(x_train,y_train)
   pred_y = model.predict(x_test)
   mse = sklearn.metrics.mean_squared_error(y_test,
pred_y,squared=False)
   print("{} error for k = {}".format(mse,i))
```

The output will be a list of errors for k = 1 right through to k = 50. The least error is for k = 3, justifying why k = 3 was used when the model was being trained.

- **Evaluate the Model**

We will use MSE (mean_squared_error) method in Scikit-learn to evaluate our model. The 'squared' parameter must be set as False, otherwise you won’t get the RMSE error:

```
#error for linear regression
mse_lr= sklearn.metrics.mean_squared_error(y_test,
pred_lr,squared=False)
print("error for Linear Regression = {}".format(mse_lr))
#error for linear regression
mse_Nn= sklearn.metrics.mean_squared_error(y_test,
pred_Nn,squared=False)
print("error for K-NN = {}".format(mse_Nn))
```

The output is:

error for Linear Regression = 4.627293724648145

error for K-NN = 6.173717334583693

We can conclude from this that Linear Regression is much better than K-NN on this dataset at any rate. This won't always be the case as it is dependent on what data you are working with.

- **Make the Prediction**

At last, we can predict the house prices using our models and the predict function. Ensure that you get all the features that were in the data you used to train the model.

Here’s the entire program from start to finish:

```
import numpy as np

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.neighbors import KNeighborsRegressor

column_names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']

bos1 = pd.read_csv('housing.csv', delimiter=r"\s+", names=column_names)

X=np.array(bos1.iloc[:,0:13])

Y=np.array(bos1["MEDV"])

#testing data size is of 30% of entire data

x_train, x_test, y_train, y_test =train_test_split(X,Y, test_size = 0.30, random_state =54)

#load our first model

lr = LinearRegression()

#train the model on training data

lr.fit(x_train,y_train)

#predict the testing data so that we can later evaluate the model
```

```
pred_lr = lr.predict(x_test)
#load the second model
Nn=KNeighborsRegressor(12)
Nn.fit(x_train,y_train)
pred_Nn = Nn.predict(x_test)
#error for linear regression
mse_lr= sklearn.metrics.mean_squared_error(y_test, pred_lr,squared=False)
print("error for Linear Regression = {}".format(mse_lr))
#error for linear regression
mse_Nn= sklearn.metrics.mean_squared_error(y_test, pred_Nn,squared=False)
print("error for K-NN = {}".format(mse_Nn))
```

Implementing a Classification Problem

The problem we are solving here is the Iris population classification problem. The Iris dataset is one of the most popular and common for beginners. It has 50 samples of each of three Iris species, along with other properties of the flowers. One is linearly separable from two species, but those two are not linearly separable from one another. This dataset has the following columns:

Different species of iris

- SepalLengthCm
- SepalWidthCm
- PetalLengthCm
- PetalWidthCm
- Species

This dataset is already included in Scikit-learn, so all we need to do is import it:

```
from sklearn.datasets import load_iris
iris = load_iris()
```

```
X=iris.data
Y=iris.target
print(X)
print(Y)
```

The output is a list of features with four items – these are the features. The bottom part is a list of labels, all transformed to numbers because the model is unable to understand strings – each name must be coded as a number.

Here's the whole thing:

```
from sklearn.model_selection import train_test_split
#testing data size is of 30% of entire data
x_train, x_test, y_train, y_test =train_test_split(X,Y, test_size = 0.3, random_state =5)
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
#fitting our model to train and test
Nn = KNeighborsClassifier(8)
Nn.fit(x_train,y_train)
#the score() method calculates the accuracy of model.
print("Accuracy for K-NN is ",Nn.score(x_test,y_test))
Lr = LogisticRegression()
Lr.fit(x_train,y_train)
print("Accuracy for Logistic Regression is ",Lr.score(x_test,y_test))
```

The output is:

Accuracy for K-NN is 1.0

Accuracy for Logistic Regression is 0.9777777777777777

Advantages and Disadvantages of Machine Learning

Everything has its advantages and disadvantages, and machine learning is no different:

Advantages

- **Makes Identifying Trends and Patterns in the Data Easy**

With machine learning, huge amounts of data can be reviewed, looking for trends and patterns the human eye wouldn't see. For example, Flipkart, Amazon, and other e-commerce websites use machine learning to help understand how their customers browse and what they buy to send them relevant deals, product details, and reminders. The machine learning results are used to show relevant ads to each customer based on their browsing and purchasing history.

- **Continuous Improvement**

New data is continuously being generated, and providing machine learning models with the new data helps them upgrade over time, become more accurate, and continuously improve their performance. This leads to continually better decisions being made.

- **They can Handle Multi-Variety and Multidimensional Data**

Machine learning algorithms can handle multi-variety and multidimensional data easily in uncertain or dynamic environments.

- **They are Widely Used**

It doesn't matter what industry you are in, machine learning can work for you. Wherever it is applied, it can give businesses the means and answers they need to make better decisions and provide their customers with a more personal user experience.

Disadvantages

- **It Needs Huge Amounts of Data**

Machine learning models need vast amounts of data to learn, but the quantity of day is only a small part. It must be high-quality, unbiased data, and there must be enough of it. At times, you may even need to wait until new data has been generated before you can train the model.

- **It Takes Time and Resources**

Machin learning needs sufficient time for the algorithms to learn and develop to ensure they can do their job with a significant level of relevancy and accuracy. They also require a high level of resources which can mean using

more computer power than you might have available.

- **Interpretation of Results**

One more challenge is being able to interpret the results that the algorithms' output accurately. You must also carefully choose the algorithms based on the task at hand and what it is intended to do. You won't always choose the right algorithm for the first time, even though your analysis points you to a specific one.

- **Susceptible to Errors**

Although machine learning is autonomous, it does have a high susceptibility to errors. Let's say you train an algorithm with such small datasets that they are not inclusive. The result would be biased predictions because your training set is biased. This would lead to a business showing their customers' irrelevant ads or purchase choices. With machine learning, this type of thing can kick-start a whole list of errors that may not be detected for some time and, when they are finally detected, its time consuming to find where the issues started and put it right.

Conclusion

Thank you for taking the time to read my guide. Data science is one of the most used buzzwords of the current time, which will not change. With the sheer amount of data being generated daily, companies rely on data science to help them move their business forward, interact intelligently with their customers, and ensure they provide exactly that their customers need when they need it.

Machine learning is another buzzword, a subset of data science that has only really become popular in the last couple of decades. Machine learning is all about training machines to think like humans, to make decisions like a human would and, although true human thought is still some way off for machines, it is improving by the day.

We discussed several data science techniques and machine learning algorithms that can help you work with data and draw meaningful insights from it. However, we've really only scraped the surface, and there is so much more to learn. Use this book as your starting point and, once you've mastered what's here, you can take your learning further. Data science and machine learning are two of the most coveted job opportunities in the world these days and, with the rise in data, there will only be more positions available. It makes sense to learn something that can potentially lead you to a lucrative and highly satisfying career.

Thank you once again for choosing my guide, and I wish you luck in your data science journey.

References

“A Comprehensive Guide to Python Data Visualization with Matplotlib and Seaborn.” *Built In* , builtin.com/data-science/data-visualization-tutorial.

“A Guide to Machine Learning Algorithms and Their Applications.” *Sas.com* , 2019, www.sas.com/en_gb/insights/articles/analytics/machine-learning-algorithms.html.

Analytics Vidhya. “Scikit-Learn in Python - Important Machine Learning Tool.” *Analytics Vidhya* , 5 Jan. 2015, www.analyticsvidhya.com/blog/2015/01/scikit-learn-python-machine-learning-tool/.

“Difference between Data Science and Machine Learning: All You Need to Know in 2021.” *Jigsaw Academy* , 9 Apr. 2021, www.jigsawacademy.com/what-is-the-difference-between-data-science-and-machine-learning/.

Leong, Nicholas. “Python for Data Science — a Guide to Pandas.” *Medium* , 11 June 2020, towardsdatascience.com/python-for-data-science-basics-of-pandas-5f8d9680617e.

“Machine Learning Algorithms with Python.” *Data Science | Machine Learning | Python | C++ | Coding | Programming | JavaScript* , 27 Nov. 2020, thecleverprogrammer.com/2020/11/27/machine-learning-algorithms-with-python/.

Mujtaba, Hussain. “Machine Learning Tutorial for Beginners | Machine Learning with Python.” *GreatLearning* , 18 Sept. 2020, www.mygreatlearning.com/blog/machine-learning-tutorial/.

Mutuvi, Steve. “Introduction to Machine Learning Model Evaluation.” *Medium* , Heartbeat, 16 Apr. 2019, heartbeat.fritz.ai/introduction-to-machine-learning-model-evaluation-fa859e1b2d7f.

Python, Real. “Pandas for Data Science (Learning Path) – Real Python.” *Realpython.com* , realpython.com/learning-paths/pandas-data-science/.

“The Ultimate NumPy Tutorial for Data Science Beginners.” *Analytics Vidhya* , 27 Apr. 2020, www.analyticsvidhya.com/blog/2020/04/the-ultimate-numpy-tutorial-for-data-science-beginners/.