DATA SCIENCES

PYTHON MACHINE LEARNING

Practical Guide for Beginners

FRANÇOIS DUVAL

Data Sciences

PYTHON MACHINE LEARNING

Practical Guide for Beginners

François Duval

How to contact us

If you find editing issues or any others issues in this book, please immediately notify our customer service by email at:

customer_service@datasciences-book.com

If you have any questions or suggestions, you can also contact the author directly at:

Francois_Duval@datasciences-book.com

Our goal is to provide you high-quality books for your technical learning in computer science subjects.

Thank you so much for buying this book.

Preface

“Python has been an important part of Google since the beginning, and remains so as the system grows and evolves. Today dozens of Google engineers use Python, and we’re looking for more people with skills in this language.”

— Peter Norvig, director of search quality at Google, Inc.

The overall aim of this book is to help you understand the application of machine learning techniques with python.

Machine learning is a field of Artificial Intelligence that uses algorithms to learn from data and make predictions. This means that we can feed data into an algorithm, and use it to make predictions about what might happen in the future.

This book is a practical guide through the basic principles of machine learning and how to get started with machine learning using Python based on libraries that make machine learning easy to get started with.

Book Objectives

If you are interested in learning more about machine learning with practical examples and its applications with python, then this book is exactly what you need.

This book will help you:

- Have an appreciation for machine learning and an understanding of their fundamental principles.
- Have an elementary grasp of machine learning concepts and algorithms.
- Achieve a technical background in machine learning and also deep learning

Target Users

The book is designed for a variety of target audiences. The most suitable users would include:

- Beginners who want to use machine learning approaches, but are too afraid of complex math to start
- Newbies in computer science techniques and machine learning
- Professionals in data science and social sciences
- Professors, lecturers or tutors who are looking to find better ways to explain the content to their students in the simplest and easiest way
- Students and academicians, especially those focusing on machine learning and deep learning

Is this book for me?

If you want to smash machine learning problems with Python and TensorFlow, this book is for you. Little programming experience is required. If you already wrote a few lines of code and recognize basic programming statements, you would be OK.

First Printing, 2017

Edited by Davies

Ebook Converted and Cover by Pixel Studio

Publised by CreateSpace Publishing

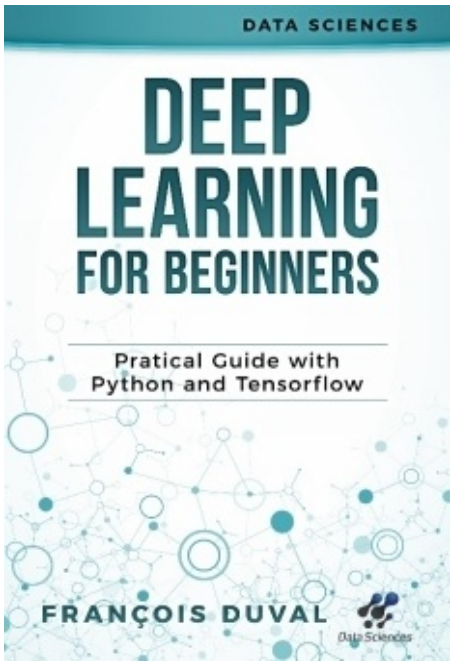
ISBN-13: 978-1985670969

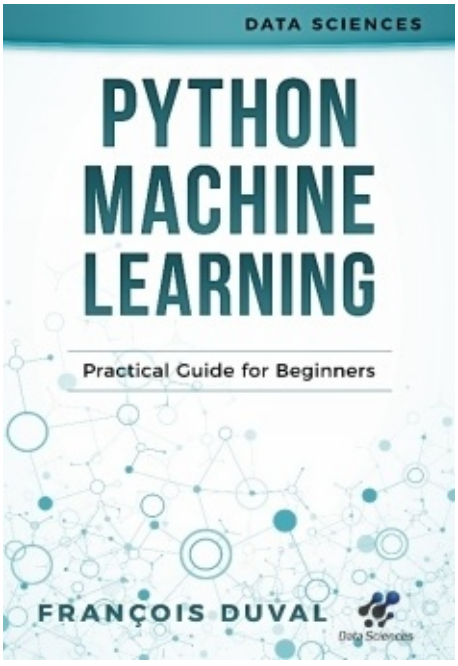
ISBN-10: 1985670968

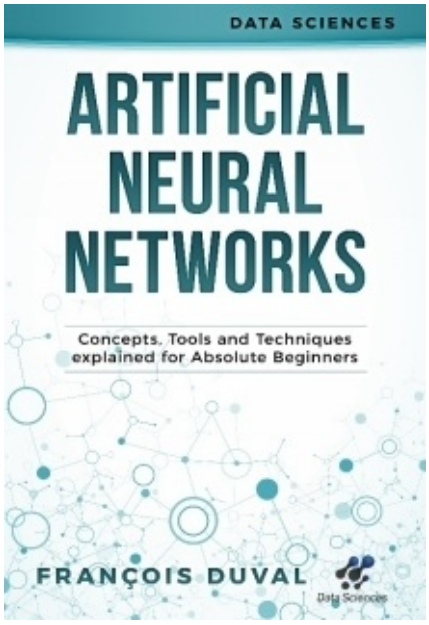
To my wife Melanie and my two children Mariane and Thomas

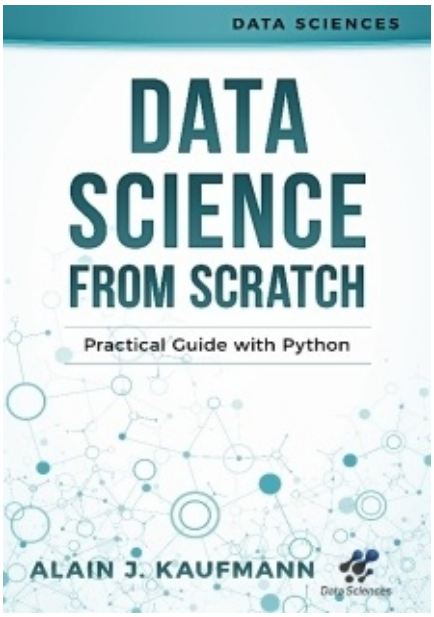
You are my life and I love you so much!

In the same Series: Data Sciences









Data Sciences

Author Biography

François Duval is a seasoned data science expert and emerging author who has quickly earned the reputation as an innovative leader in the information technology and data analysis space. Throughout the course of the past decade, he has gained extensive firsthand experience in his field.

Currently, François is proudly serving as a scientific data consultant. He is also working on series of books in statistics and computer science.

Prior to his present venture, he was on the IBM team for two years and is the founder of Intelligence Data Consulting, which is a prominent company that provides data analysis consulting solutions in France and throughout Europe. He has also worked as an engineer across multiple industries, including but not limited to finance (KPMG within the Financial Advisory Department in London) as well as environmental and healthcare research (INERIS and Mines ParisTech in France).

No matter what venture he takes on, François is on a lifelong mission to impart data analysis newbies with the knowledge they need to better grasp various industry techniques. By effectively simplifying such methodologies, up-and-coming professionals are empowered to create positive changes in the world of computer science that will continue to have a ripple effect in the entire field as we know it.

Furthermore, François holds a PhD in Quantitative Analysis and Stochastic Calculus and an Undergraduate Degree in Statistics & Data Analysis.

When he isn’t immersed in his multifaceted career, François Duval enjoys spending quality family time with his loved ones. Today, he happily resides in London, England and is the proud father of two beautiful children.

Table of Contents

Introduction

Objective of this Chapter

At the end of this chapter, the reader should have learnt:

- Definition of main concepts and terminology
- Importance of Python in the machine learning

Definition

Machine learning has been around for a long time now and every single social media user, eventually in time, have been buyers of Machine learning innovation. One of the normal illustrations is facial recognition software, which has the capacity to recognize whether a digital photo includes a given person. Today, Facebook clients can see automatic suggestions to tag their friends in digital photos that are uploaded on Facebook.

How about we begin with defining what Machine realizing is. There are numerous specialized definitions for Machine learning, one is given below:

“A branch of Artificial Intelligence in which a Computer produces rules based on the raw data that has been fed into the computer and the computer can take a decision on the basis of that data.”

Beyond these definitions, a solitary term or definition for Machine learning is the key to facilitate the definition of a problem-solving platform. Essentially, it is a mechanism for pattern search and building intelligence into a machine to have the capacity to learn, suggesting that it will have the capacity to improve the situation later on from its own particular experience.

Drilling down somewhat more into what an example ordinarily is, pattern search or pattern recognition is the study of how machines react to the certain environment, learn to discriminate behavior of interest from the rest and have the capacity to take sensible choices about categorizing the behavior. People more frequently perform all these. The objective is to encourage exactness, speed, and stay away from the possibility of inappropriate use of the system.

Machine learning algorithms that are developed along these lines to handle building intelligence. Basically, machines understand information similarly like people do.

The essential objective of a Machine learning usage is to build up a general-purpose algorithm that takes care of practical and focused problem. A portion of the viewpoints that are imperative and should be considered in this procedure incorporate information, time and space requirements. Above all, with the capacity to be connected to a wide class of learning problems, the objective of a learning algorithm is to deliver an outcome that is a rule and is as exact as could be expected under the circumstances.

What is learning?

Now, let us have a look at what the meaning of “learning” really is with regards to Machine learning. In basic terms, historical data or observations are used so that machine can predict or machine can derive actionable tasks. Clearly, one mandate for an intelligent system is its capacity to learn. Given below are a few considerations that define a learning issue:

- Provides the definition of what the learner really wants to learn and the need for learning.
- Secondly defines the data requirements and the original source of the data.
- Thirdly defines if the learner ought to work on the entire dataset or a subset will do.

Before we dive into understandings of the internals of each learning type in the following section, you must have to understand the basic and simple process that is followed to solve the learning problems, which includes building and validating models that solves any issue with most accurate precision.

In machine learning generally there are fundamentally two sorts of datasets required. The first type of dataset is manually prepared where the input data

and expected output data are already available and manually prepared. It is very important that each piece of input data has an expected output data point accessible as this will be used in a supervised manner to build the rule. The second type of dataset is totally opposite to first type because in this dataset we have input data and instead of manually prepared output data we have to predict the expected output data.

As an initial step, the given information is isolated into three datasets: training, validation and testing. There is no one hard rule on what percentage of data should be training, validation, and testing datasets. It can be 70-10-20, 60-30-10, 50-25-25, or some other values.

The training of the datasets refers to data examples that are used to learn or build a classifier, for instance. The validation of the dataset refers to the data examples that are checked against the built classifier and can help tune the accuracy of the output.

The testing dataset refers to the data examples that help assess the performance of the classifier.

There are normally three stages for performing Machine learning:

• Phase 1—Training Phase:

This is where training data is utilized to train the model by paring the given input with the expected output. The purpose of this stage is simply the training of the model.

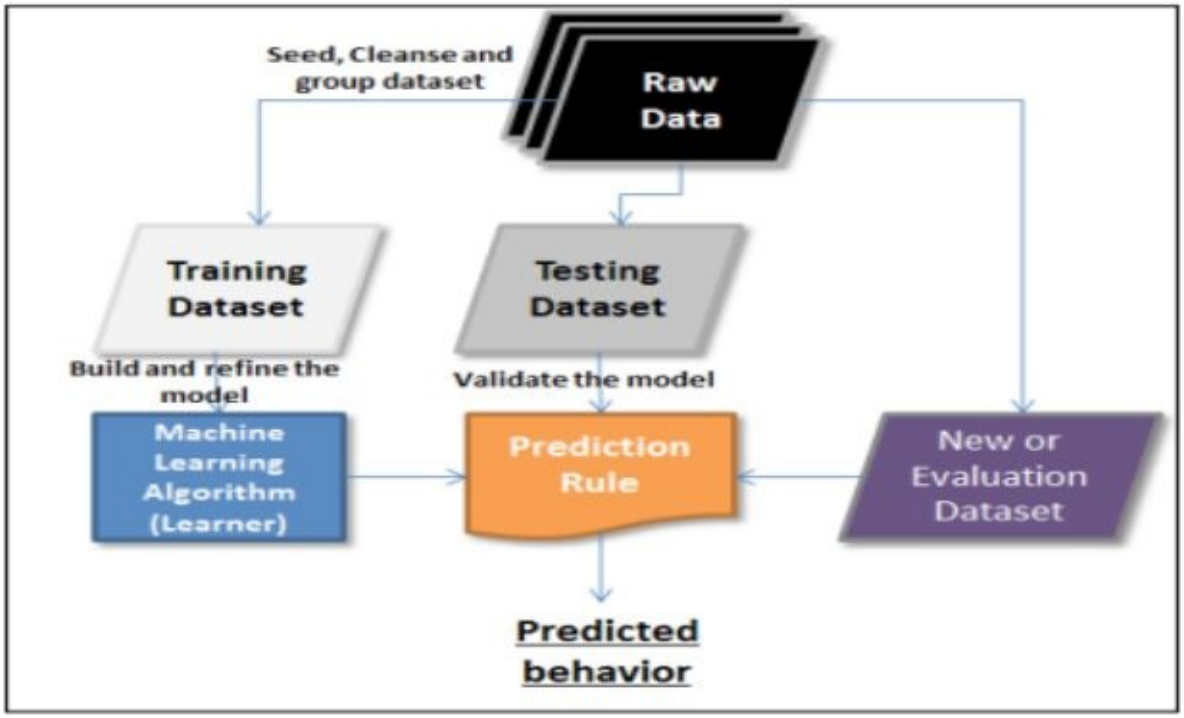
• Phase 2—Validation and Test Phase:

This stage is to quantify how great the training model, that has been trained, is and estimate the model properties, for example, mistake measures, review,

exactness, and others. This stage utilizes a validation dataset, and the output is a modern learning model.

• Phase 3—Application Phase:

In this stage, the model is subject to the real-world data for which the outcomes should be inferred.



Core Concepts and Terminology

At the core of Machine learning is knowing and utilizing the information properly.

This incorporates gathering the correct information, purifying the information, handling the information, utilizing learning algorithms iteratively to build models, utilizing certain key features of information, and in view of the theories from these models, making predictions.

In this chapter, we will cover the standard nomenclature or terminology utilized in machine learning, beginning from how to describe data and

particular machine learning tasks

Data

In Machine learning data forms the source of the learning. The data referenced here can be in any format, can be gotten at any frequency, and can be of any size. With regards to dealing with substantial datasets in the Machine learning context, there are some new procedures that have advanced and are being explored using different avenues.

There are likewise more enormous data aspects, including parallel processing, distributed storage, and execution. More on the huge scale, parts of data will be secured in the following part, including some extraordinary differentiators.

When we consider data, dimensions ring the bell of mind. To begin with, we have columns and rows with respect to organized and unstructured data. This book will cover taking care of both organized and unstructured data in the machine learning context.

In this section of data, we will cover the terminology related to data in the Machine learning context.

Term	Purpose or meaning in the context of Machine learning
Feature, attribute, field, or variable	This is a single column of data being referenced by the learning algorithms. Some features can be input to the learning algorithm, and some can be the outputs.
Instance	This is a single row of data in the dataset.
Feature vector or tuple	This is a list of features.
Dimension	This is a subset of attributes used to describe a property of data. For example, a date dimension consists of three attributes: day, month, and year.
Dataset	A collection of rows or instances is called a dataset. In the context of Machine learning, there are different types of datasets that are meant to be used for different purposes. An algorithm is run on different datasets at different stages to measure the accuracy of the model. There are three types of dataset: training, testing, and evaluation datasets. Any given comprehensive dataset is split into three categories of datasets and is usually in the following proportions: 60% training, 30% testing, and 10% evaluation.
a. Training Dataset	The training dataset is the dataset that is the base dataset against which the model is built or trained.
b. Testing Dataset	The testing dataset is the dataset that is used to validate the model built. This dataset is also referred to as a validating dataset.
c. Evaluation Dataset	The evaluation dataset is the dataset that is used for final verification of the model (and can be treated more as user acceptance testing).
Data Types	Attributes or features can have different data types. Some of the data types are listed here: • Categorical (for example: young, old). • Ordinal (for example: 0, 1). • Numeric (for example: 1.3, 2.1, 3.2, and so on).
Coverage	The percentage of a dataset for which a prediction is made or the model is covered. This determines the confidence of the prediction model.

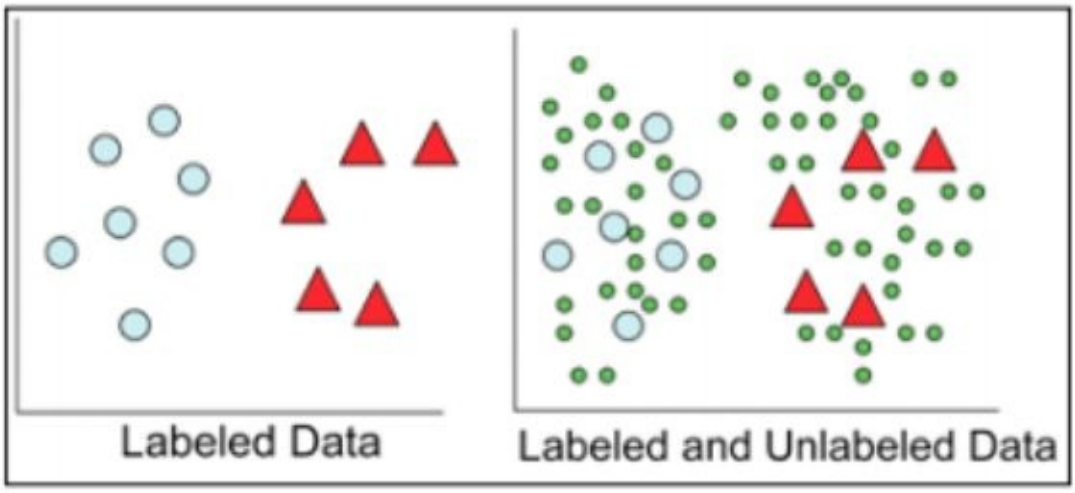
Labeled and unlabeled Data

Data in the machine learning context can either be labeled or unlabeled. Before we go deeper into the Machine learning basics, you have to understand this classification, what data is utilized when, as this term will be utilized through out this book.

Unlabeled data is generally the crude type of the data. It comprises of sample of natural or then again human-made antiques. This classification of data is effectively accessible in abundance.

For instance, video streams, sound, photographs, and tweets among others. This type of data most often has no explanation and no meaning attached. When a meaning is attached to the unlabeled data, it becomes labeled data. This is the main difference between them. Here, we are looking at connecting a “tag” or “label” that is required, and is obligatory, to translate and characterize the importance. For instance, labels for a photograph can be the points of interest like a sound record, a political gathering, a goodbye gathering, et cetera. All the more regularly, the names are mapped or characterized by people and are fundamentally costlier to get than the unlabeled raw data.

The learning models can be connected to both labeled and unlabeled data. We can infer more precise models utilizing a mix of labeled and unlabeled datasets. The accompanying chart speaks about labeled and unlabeled information. The two triangles and greater circles represent the labeled data and little circles represent the unlabeled data.



The use of labeled and unlabeled information is talked about in more detail in the following segments. You will see that supervised learning embraces labeled data and unsupervised learning embraces unlabeled data. Semi-supervised learning and deep learning methods apply a blend of labeled and unlabeled data in an assortment of approaches to construct exact models.

Tasks

Basically, the core concept of the task in terms of Machine Learning is a problem that the Machine learning algorithm is built to solve. It is essential that we measure the performance of a task when it is performing. The expression “performance” in this unique situation is only the degree of certainty with which problem is solved. Different Types of algorithms when runs on various datasets produces a different model.

It is essential that the models hence generated are not compared, and rather, the consistency of the model comes about with various datasets and diverse models is estimated.

Algorithms

Once you have the complete understanding of what really a machine learning problem is, the focus is around what data and algorithms are important or applicable. There are a few algorithms which are accessible. These algorithms are either assembled by the learning subfields (for example, managed, unsupervised, reinforcement, semi-supervised, or deep) or the problem classifications, (for example, Classification, Regression, Clustering or Optimization). These algorithms are connected iteratively on various datasets, and output models that develop with new data are caught. Some of the Machine learning algorithms with python code are given below:

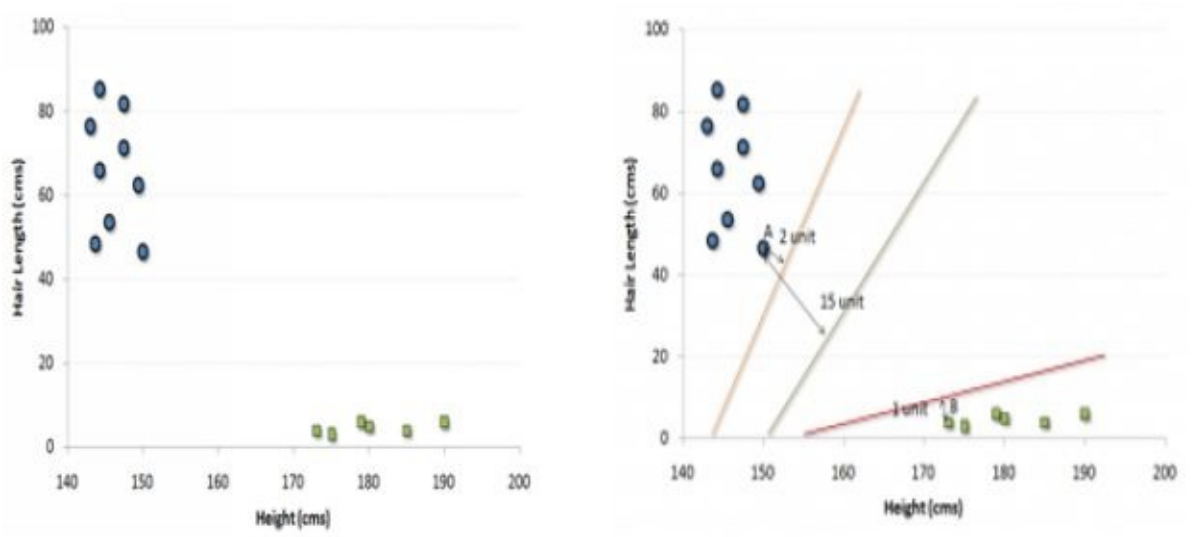
- Support Vector Machine (SVM)
- K- Nearest neighbor (KNN)

Support Vector Machine (SVM)

It is a classification method. In this algorithm, we plot each data section as a point in n-dimensional space (where n is the number of attributes you have) where the value of each attribute is the value of an exacting coordinate.

For example, in the event that we just had two characteristics like Height and Hair length of an element, we would first plot these two factors in two-dimensional space where each point has two co-ordinates (these co-ordinates

are known as Support Vectors.)



In the figure above, the line which parts the data into two diverse classified gatherings is the dark line, given that the two nearest focuses are the most distant separated from the line. This line is our classifier. At that point, depending upon where the testing data arrives on either side of the line, that is the thing that class we can characterize the new information as.

SVM Python code

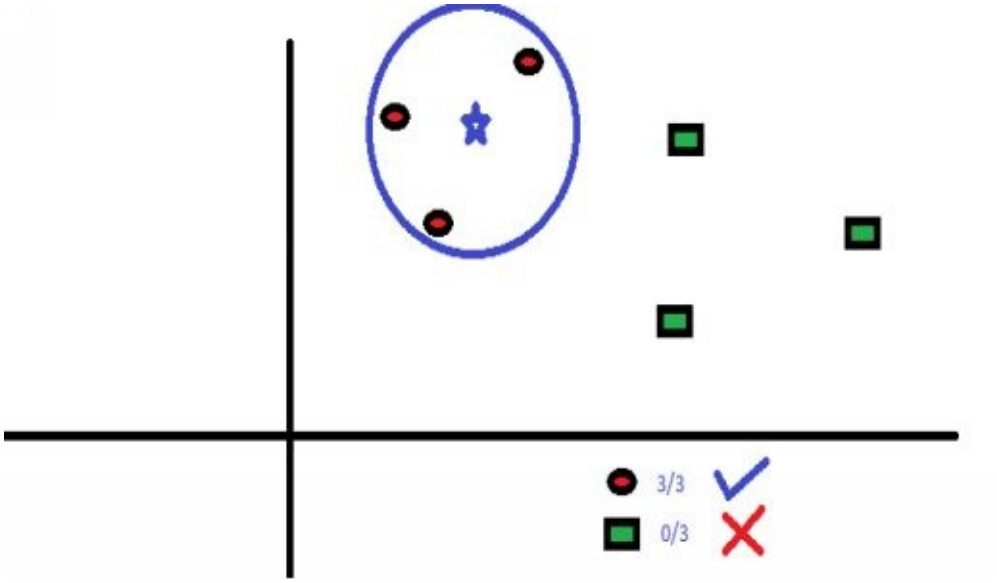
```
#Import Library
from sklearn import svm

#Assume that you have X (predictor) and Y (target) for training data set and x_test(predictor) for test_dataset

# Create SVM classification object
model = svm.svc() # there are various options associated with it, this is simple for classification.

# Train the model using the training sets and check the score
model.fit(X, y)
model.score(X, y)
```

```
#Predict an Output
predicted= model.predict(x_test)
```

K- Nearest neighbor (KNN)

It can be useful for both classification and regression cases. Then again, it is most broadly utilized as a part of grouping issues in the organization. K closest neighbors is a simple algorithm that stores every single accessible case and groups new cases by a larger part vote of its k neighbors. The case being appointed to the class is most basic among its K closest neighbors calculated by a distance function.

These separation capacities can be Euclidean, Manhattan, Minkowski and Hamming separation. Initial three capacities are used for continuous function and fourth one (Hamming) for categorical variables. On the off chance that K = 1, at that point the case is just allocated to the class of its closest neighbor. Now and again, picking K ends up being a test while performing KNN.



KNN Python code

```
#Import Library
```

```
from sklearn.neighbors import KNeighborsClassifier

#Assume that you have, X (predictor) and Y (target) for training data set and x_test(predictor) for test_dataset

# Create KNeighbors classifier object model
KNeighborsClassifier(n_neighbors=6) # default value for n_neighbors is 5

# Train the model using the training sets and check the score
model.fit(X, y)
```

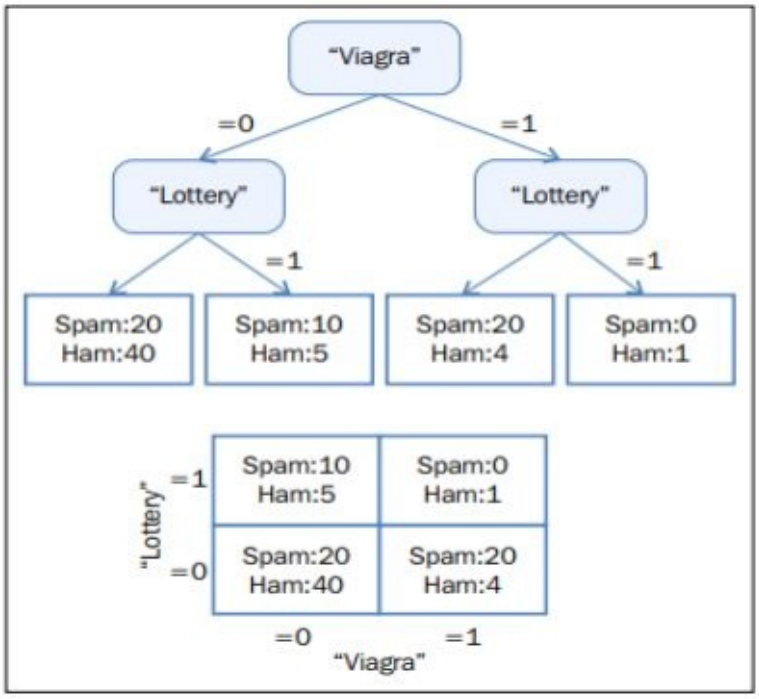
Models

Models are a core part to any Machine learning implementation. A model depicts data that is seen in a system. Models are the output of algorithms connected to a dataset. By and large, these models are connected to new datasets that assist the models learn new conduct and furthermore anticipate them. There is an immense range of machine learning algorithms that can be connected to a given issue. At an abnormal state, models are classified as the accompanying:

- Logical models
- Geometric models
- Probabilistic models

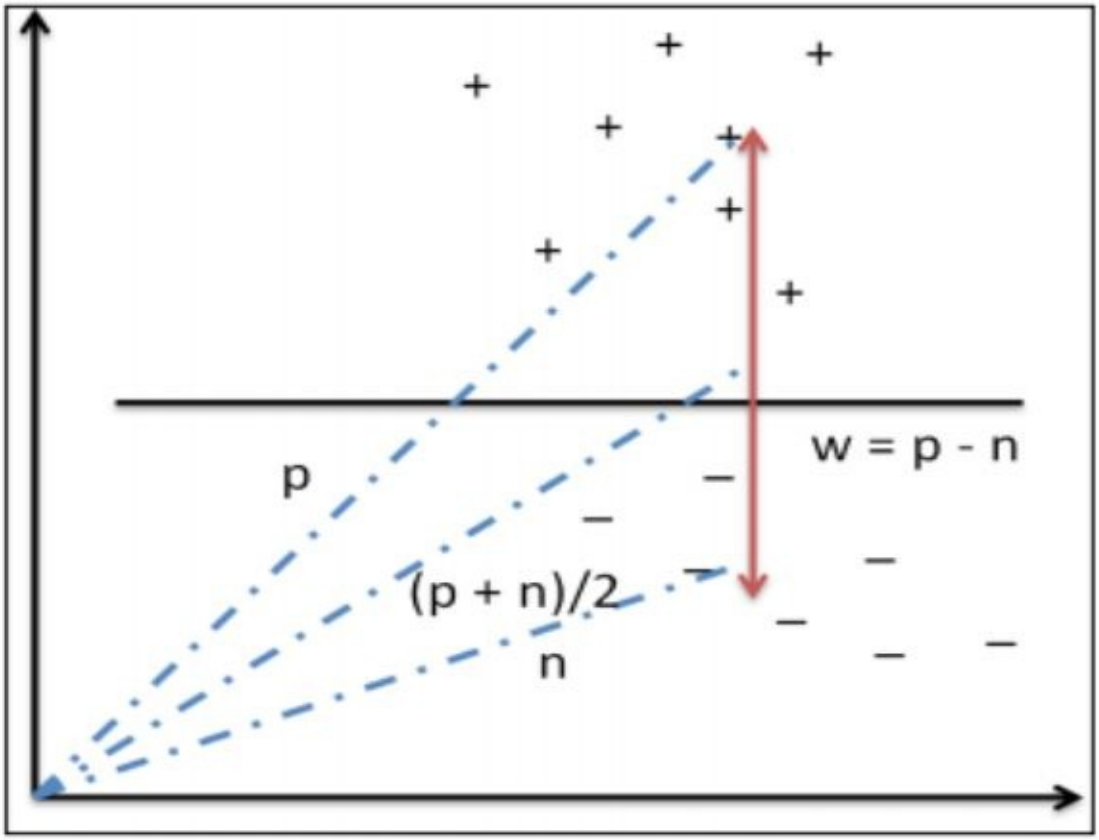
Logical models

The nature of Logical models is more algorithmic and they help us derive the set of rules by running the algorithm iteratively on the given data set. A Decision Tree is one such example of logical models.



Geometric models

Geometric models are basically of geometric type and uses the geometric concepts such as lines, planes and distances. These Models usually operate, or can operate on very high volumes of data. Basically, linear transformation helps us to compare different machine learning methods.



Probabilistic models

Probabilistic models are statistical models that utilize statistical procedures. These models depend on a methodology that characterizes the connection between two variables. This relationship can be inferred without a doubt as this includes using the random background process. As a rule, a subset of the general information can be considered for preparing:

Viagra	Lottery	P(Y= Spam (Viagra, lottery))	P(Y= ham (Viagra, lottery))
0	0	0.31	0.69
0	1	0.65	0.35
1	0	0.80	0.20
1	1	0.40	0.60

Why Python for Machine Learning

Python is easy, rich, predictable, and math-like. Python code has been portrayed as comprehensible pseudo code.

It is easy to pick it up due to its consistent syntax and the way it mirrors human language as well as their mathematical counterparts. The most important thing is that there are libraries such as Numpy that is something one will acknowledge if he somehow happens to implement a machine learning algorithm of which the core is likely simply mathematical optimization.

Essential Libraries and their Installation

Objective of this Chapter

At the end of this chapter, the reader should have learned:

- Essential Python libraries
- Python installation
- Virtualenvs and Autoenv installation
- Creating Python Environment

Essential python libraries

There are a couple of open source libraries that are available that allow the formation of deep neural nets in Python without having to explicitly make the code without preparation. The most frequently used are: Keras, Theano, TensorFlow, Caffe, and Torch. In this book we will give illustrations using the library TensorFlow, which would all have the capacity to be used as a piece of Python.

TensorFlow

TensorFlow is the second machine learning framework that Google made and is utilized to configure, manufacture, and get ready deep learning models. You can use the TensorFlow library to do numerical computations, which itself doesn't seem, by all accounts, to be extremely uncommon, yet these calculations are done with data stream graphs. In these graphs, hubs address numerical tasks, while the edges address the data, which as a rule are multidimensional data bunches or tensors that are passed on between these edges.

TensorFlow works correspondingly to Theano, and in TensorFlow, calculations are moreover communicated as outlines or charts. A TensorFlow graph is thusly a portrayal of calculations. In TensorFlow, you don't need to unequivocally require the use of your GPU; rather, TensorFlow will subsequently endeavor to use your GPU in case you have one, in any case if you have more than one GPU you ought to assign tasks to each GPU explicitly, or simply the first one will be used. To do this, you basically need to give the line:

With tensorflow.device ("/gpu: 1"):

Following devices are to be defined

• "/cpu: 0": main CPU of your machine

• "/gpu: 0": first GPU of your machine, if one exists

• “/gpu: 1”: second GPU of your machine, if it exists

• “/gpu: 2”: third GPU of your machine, if it exists, and so on

Starting with TensorFlow

Once you have done the proper installation you can import tensorflow library on your laptops and computers without any trouble, few websites links are given in case of any problem you face:

(https://www.tensorflow.org/install/install_windows)
(https://www.tensorflow.org/install/install_mac)

import tensorflow as tf

First step is that you have import tensorflow library under the alias tf. Initialize two factors that are really constant. Pass an array of four numbers to the constants() function. As tensors are about arrays, next you can utilize multiply() to multiply your two variables. Store the outcome in the outcome variable. In conclusion, print out the outcome with the assistance of the print() function.

```
# Import `tensorflow`
import tensorflow as tf

# Initialize two constants
x1 = tf.constant([1,2,3,4])
x2 = tf.constant([5,6,7,8])

# Multiply
result = tf.multiply(x1, x2)
# Print the result
print(result)

config=tf.ConfigProto(log_device_placement=True)
```

At this point we should sign in to the GPU or CPU gadget.

Since you have TensorFlow brought and imported into your workspace and you have encountered the nuts and bolts of working with this package, this is a perfect chance to learn some basics of building deep networks.

Python Installation

As Python 3 has various awesome features, it enables you to simply introduce the latest variation of Python 3 from www.python.org/downloads first. Make a beeline for the connection at that point, tap on the download v3.6 and after that, it is presented.

After a while you should have this yield in the terminal:

```
$ python3 —version
Python 3.6.0
```

Installing Virtualenvs

This virtualenvs is exceptionally recommended as it empowers different endeavors on your PC to have particular Python condition (http://docs.python-guide.org/en/most recent/dev/virtualenvs/), or we can simply include:

```
$ brew install autoenv
$ echo “source $(brew —prefix autoenv)/activate.sh” >> ~/.bash_profile
```

Installing autoenv

This autoenv instrument is similarly exceptionally prescribed as it subsequently switches the correct Python condition for you when you album

into an envelope. It uses a config record called an .env archive in your wander file to switch the shell Python condition to needed condition. Look at (https://github.com/kennethreitz/autoenv) or essentially endeavor to present it along these lines:

```
$ brew install autoenv
$ echo “source $(brew —prefix autoenv)/activate.sh” >> ~/.bash_profile
```

Variational Auto-encoder in TensorFlow

Now let us have a look at how to do fundamentals imports, stack the information (MNIST), and characterize some partner capacities. You just have to copy and paste the code in python tool once you have installed and configured this library:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
%matplotlib inline
np.random.seed (0)
tf.set_random_seed (0)

# Load MNIST data in a format suited for tensorflow.
# The script input_data is available under this URL:
”””
https://raw.githubusercontent.com/tensorflow/tensorflow/master/tensorflow/examples/tutorials/mnist/inp
”””
import input_data
mnist = input_data.read_data_sets(‘MNIST_data’, one_hot=True)
n_samples = mnist.train.num_examples
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
```

```
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

```
def xavier_init(fan_in, fan_out, constant=1):
  """ Xavier initialization of network weights"""
  # https://stackoverflow.com/questions/33640581/how-to-do-xavier-initialization-on-tensorflow
  low = -constant*np.sqrt(6.0/(fan_in + fan_out))
  high = constant*np.sqrt(6.0/(fan_in + fan_out))
  return tf.random_uniform((fan_in, fan_out),
     minval=low, maxval=high,
     dtype=tf.float32)
```

This class “Variational autoencoder” is defined with a border which can be trained using incremental approach with mini-batches using partial fit. The purpose of this trained model is that it is used in reconstruction of unseen input, which can generate new samples and to map inputs to the latent space.

class VariationalAutoencoder(object):

""" Variation Autoencoder (VAE) with an sklearn-like interface implemented using TensorFlow.

This implementation uses probabilistic encoders and decoders using Gaussian distributions and realized by multi-layer perceptrons. The VAE can be learned end-to-end.

See “Auto-Encoding Variational Bayes” by Kingma and Welling for more details.

"""

```
  def __init__(self, network_architecture, transfer_fct=tf.nn.softplus,
     learning_rate=0.001, batch_size=100):
     self.network_architecture = network_architecture
```

```
        self.transfer_fct = transfer_fct
        self.learning_rate = learning_rate
        self.batch_size = batch_size

        # tf Graph input
        self.x = tf.placeholder(tf.float32, [None, network_architecture[“n_input”]])

        # Create autoencoder network
        self._create_network()
        # Define loss function based variational upper-bound and
        # corresponding optimizer
        self._create_loss_optimizer()

        # Initializing the tensor flow variables
        init = tf.global_variables_initializer()

        # Launch the session
        self.sess = tf.InteractiveSession()
        self.sess.run(init)

    def _create_network(self):
        # Initialize autoencode network weights and biases
        network_weights = self._initialize_weights(**self.network_architecture)

        # Use recognition network to determine mean and
        # (log) variance of Gaussian distribution in latent
        # space
        self.z_mean, self.z_log_sigma_sq = \
        self._recognition_network(network_weights[“weights_recog”],
        network_weights[“biases_recog”])

        # Draw one sample z from Gaussian distribution
        n_z = self.network_architecture[“n_z”]
        eps = tf.random_normal((self.batch_size, n_z), 0, 1,
        dtype=tf.float32)
        # z = mu + sigma*epsilon
```

```
    self.z = tf.add(self.z_mean,
    tf.mul(tf.sqrt(tf.exp(self.z_log_sigma_sq)), eps))

    # Use generator to determine mean of
    # Bernoulli distribution of reconstructed input
    self.x_reconstr_mean = \
    self._generator_network(network_weights[“weights_gener”],
    network_weights[“biases_gener”])

  def _initialize_weights(self, n_hidden_recog_1, n_hidden_recog_2,
    n_hidden_gener_1, n_hidden_gener_2,
    n_input, n_z):
    all_weights = dict()
    all_weights[‘weights_recog’] = {
    ‘h1’: tf.Variable(xavier_init(n_input, n_hidden_recog_1)),
    ‘h2’: tf.Variable(xavier_init(n_hidden_recog_1, n_hidden_recog_2)),
    ‘out_mean’: tf.Variable(xavier_init(n_hidden_recog_2, n_z)),
    ‘out_log_sigma’: tf.Variable(xavier_init(n_hidden_recog_2, n_z))}
    all_weights[‘biases_recog’] = {
    ‘b1’: tf.Variable(tf.zeros([n_hidden_recog_1], dtype=tf.float32)),
    ‘b2’: tf.Variable(tf.zeros([n_hidden_recog_2], dtype=tf.float32)),
    ‘out_mean’: tf.Variable(tf.zeros([n_z], dtype=tf.float32)),
    ‘out_log_sigma’: tf.Variable(tf.zeros([n_z], dtype=tf.float32))}
    all_weights[‘weights_gener’] = {
    ‘h1’: tf.Variable(xavier_init(n_z, n_hidden_gener_1)),
    ‘h2’: tf.Variable(xavier_init(n_hidden_gener_1, n_hidden_gener_2)),
    ‘out_mean’: tf.Variable(xavier_init(n_hidden_gener_2, n_input)),
    ‘out_log_sigma’: tf.Variable(xavier_init(n_hidden_gener_2, n_input))}
    all_weights[‘biases_gener’] = {
    ‘b1’: tf.Variable(tf.zeros([n_hidden_gener_1], dtype=tf.float32)),
    ‘b2’: tf.Variable(tf.zeros([n_hidden_gener_2], dtype=tf.float32)),
    ‘out_mean’: tf.Variable(tf.zeros([n_input], dtype=tf.float32)),
    ‘out_log_sigma’: tf.Variable(tf.zeros([n_input], dtype=tf.float32))}
    return all_weights

  def _recognition_network(self, weights, biases):
```

```
    # Generate probabilistic encoder (recognition network), which
    # maps inputs onto a normal distribution in latent space.
    # The transformation is parametrized and can be learned.
    layer_1 = self.transfer_fct(tf.add(tf.matmul(self.x, weights['h1']),
    biases['b1']))
    layer_2 = self.transfer_fct(tf.add(tf.matmul(layer_1, weights['h2']),
    biases['b2']))
    z_mean = tf.add(tf.matmul(layer_2, weights['out_mean']),
    biases['out_mean'])
    z_log_sigma_sq = \
    tf.add(tf.matmul(layer_2, weights['out_log_sigma']),
    biases['out_log_sigma'])
    return (z_mean, z_log_sigma_sq)

  def _generator_network(self, weights, biases):
    # Generate probabilistic decoder (decoder network), which
    # maps points in latent space onto a Bernoulli distribution in data space.
    # The transformation is parametrized and can be learned.
    layer_1 = self.transfer_fct(tf.add(tf.matmul(self.z, weights['h1']),
    biases['b1']))
    layer_2 = self.transfer_fct(tf.add(tf.matmul(layer_1, weights['h2']),
    biases['b2']))
    x_reconstr_mean = \
    tf.nn.sigmoid(tf.add(tf.matmul(layer_2, weights['out_mean']),
    biases['out_mean']))
    return x_reconstr_mean

  def _create_loss_optimizer(self):
    # The loss is composed of two terms:
    # 1.) The reconstruction loss (the negative log probability
    #    of the input under the reconstructed Bernoulli distribution
    #    induced by the decoder in the data space).
    #    This can be interpreted as the number of "nats" required
    #    for reconstructing the input when the activation in latent
    #    is given.
    # Adding 1e-10 to avoid evaluation of log(0.0)
```

```
    reconstr_loss = \
    -tf.reduce_sum(self.x * tf.log(1e-10 + self.x_reconstr_mean)
    + (1-self.x) * tf.log(1e-10 + 1 - self.x_reconstr_mean),
    1)
    # 2.) The latent loss, which is defined as the Kullback Leibler divergence
    ##  between the distribution in latent space induced by the encoder on
    #   the data and some prior. This acts as a kind of regularizer.
    #   This can be interpreted as the number of “nats” required
    #   for transmitting the latent space distribution given
    #   the prior.
    latent_loss = -0.5 * tf.reduce_sum(1 + self.z_log_sigma_sq
    - tf.square(self.z_mean)
    - tf.exp(self.z_log_sigma_sq), 1)
    self.cost = tf.reduce_mean(reconstr_loss + latent_loss)  # average over batch
    # Use ADAM optimizer
    self.optimizer = \
    tf.train.AdamOptimizer(learning_rate=self.learning_rate).minimize(self.cost)

  def partial_fit(self, X):
    “””Train model based on mini-batch of input data.

    Return cost of mini-batch.
    ”””
    opt, cost = self.sess.run((self.optimizer, self.cost),
    feed_dict={self.x: X})
    return cost

  def transform(self, X):
    “””Transform data by mapping it into the latent space.”””
    # Note: This maps to mean of distribution, we could alternatively
    # sample from a Gaussian distribution
    return self.sess.run(self.z_mean, feed_dict={self.x: X})

  def generate(self, z_mu=None):
    ””” Generate data by sampling from latent space.
```

```
    If z_mu is not None, data for this point in latent space is
    generated. Otherwise, z_mu is drawn from prior in latent
    space.
    """
    if z_mu is None:
    z_mu = np.random.normal(size=self.network_architecture["n_z"])
    # Note: This maps to mean of distribution, we could alternatively
    # sample from Gaussian distribution
    return self.sess.run(self.x_reconstr_mean,
    feed_dict={self.z: z_mu})

  def reconstruct(self, X):
    """ Use VAE to reconstruct given data. """
    return self.sess.run(self.x_reconstr_mean,
    feed_dict={self.x: X})

def train(network_architecture, learning_rate=0.001,
    batch_size=100, training_epochs=10, display_step=5):
  vae = VariationalAutoencoder(network_architecture,
    learning_rate=learning_rate,
    batch_size=batch_size)
  # Training cycle
  for epoch in range(training_epochs):
    avg_cost = 0.
    total_batch = int(n_samples / batch_size)
    # Loop over all batches
    for i in range(total_batch):
    batch_xs, _ = mnist.train.next_batch(batch_size)
    # Fit training using batch data
    cost = vae.partial_fit(batch_xs)
    # Compute average loss
    avg_cost += cost / n_samples * batch_size

    # Display logs per epoch step
    if epoch % display_step == 0:
    print("Epoch:", '%04d' % (epoch+1),
```

```
    "cost=", "{:.9f}".format(avg_cost))
  return vae
```

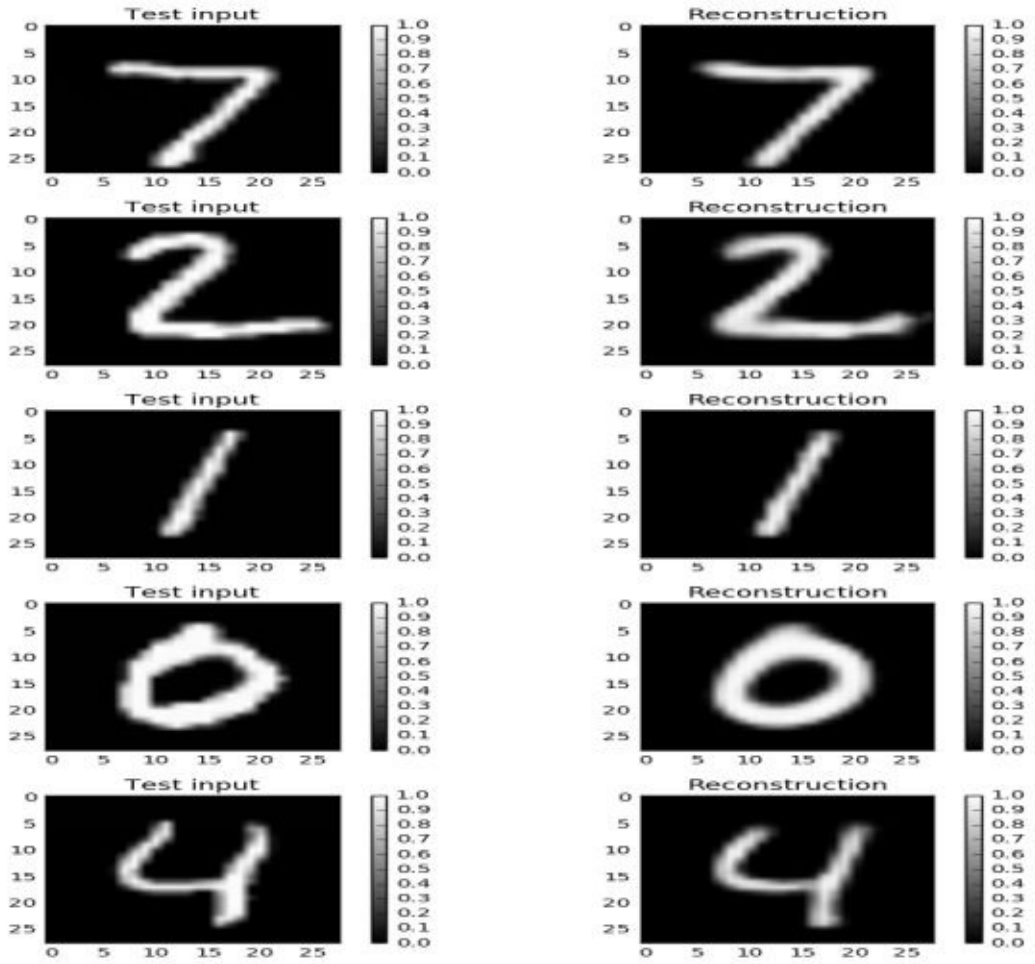
We can now train a VAE on MNIST by merely specifying the network topology. We start with training a VAE with a 20-dimensional latent space.

```
network_architecture = \
  dict(n_hidden_recog_1=500, # 1st layer encoder neurons
    n_hidden_recog_2=500, # 2nd layer encoder neurons
    n_hidden_gener_1=500, # 1st layer decoder neurons
    n_hidden_gener_2=500, # 2nd layer decoder neurons
    n_input=784, # MNIST data input (img shape: 28*28)
    n_z=20) # dimensionality of latent space


vae = train(network_architecture, training_epochs=75)
```

```
Epoch: 0001 cost= 181.437451200
Epoch: 0006 cost= 110.449812497
Epoch: 0011 cost= 106.030304274
Epoch: 0016 cost= 103.363558572
Epoch: 0021 cost= 101.201389451
Epoch: 0026 cost= 99.697997270
Epoch: 0031 cost= 98.750394565
Epoch: 0036 cost= 97.979149531
Epoch: 0041 cost= 97.337145219
Epoch: 0046 cost= 96.880868322
Epoch: 0051 cost= 96.415262562
Epoch: 0056 cost= 96.061289187
Epoch: 0061 cost= 95.761952903
Epoch: 0066 cost= 95.480659388
Epoch: 0071 cost= 95.174087233
```

With the help of this we can check some tests inputs and check how auto-encoders can remake those already test inputs.

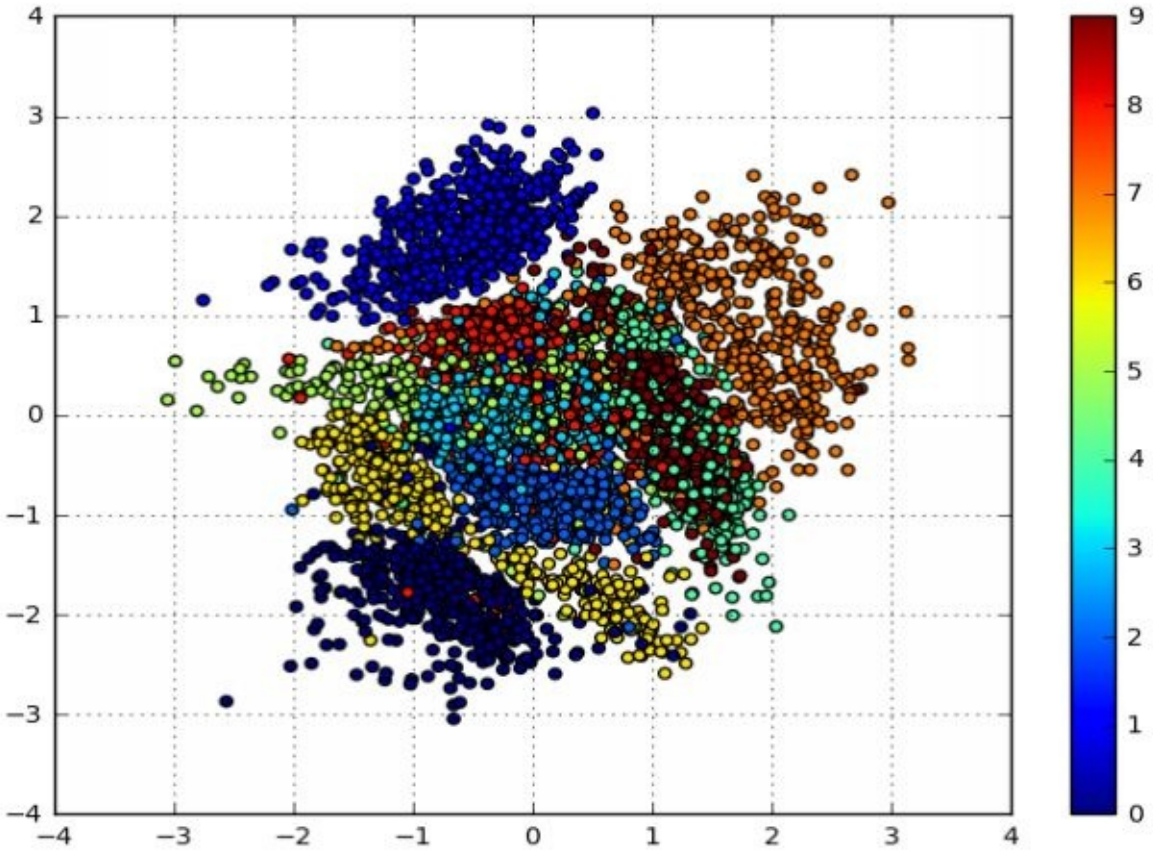
```
x_sample = mnist.test.next_batch(100)[0]
x_reconstruct = vae.reconstruct(x_sample)

plt.figure(figsize=(8, 12))
for i in range(5):

  plt.subplot(5, 2, 2*i + 1)
  plt.imshow(x_sample[i].reshape(28, 28), vmin=0, vmax=1, cmap="gray")
  plt.title("Test input")
  plt.colorbar()
  plt.subplot(5, 2, 2*i + 2)
  plt.imshow(x_reconstruct[i].reshape(28, 28), vmin=0, vmax=1, cmap="gray")
  plt.title("Reconstruction")
  plt.colorbar()
plt.tight_layout()
```




```
network_architecture = \
  dict(n_hidden_recog_1=500, # 1st layer encoder neurons
       n_hidden_recog_2=500, # 2nd layer encoder neurons
       n_hidden_gener_1=500, # 1st layer decoder neurons
       n_hidden_gener_2=500, # 2nd layer decoder neurons
       n_input=784, # MNIST data input (img shape: 28*28)
       n_z=2) # dimensionality of latent space

vae_2d = train(network_architecture, training_epochs=75)

Epoch: 0001 cost= 187.989159047
Epoch: 0006 cost= 153.522320945
Epoch: 0011 cost= 148.390766352
Epoch: 0016 cost= 145.729972423
Epoch: 0021 cost= 144.193653717
Epoch: 0026 cost= 143.046030051
```

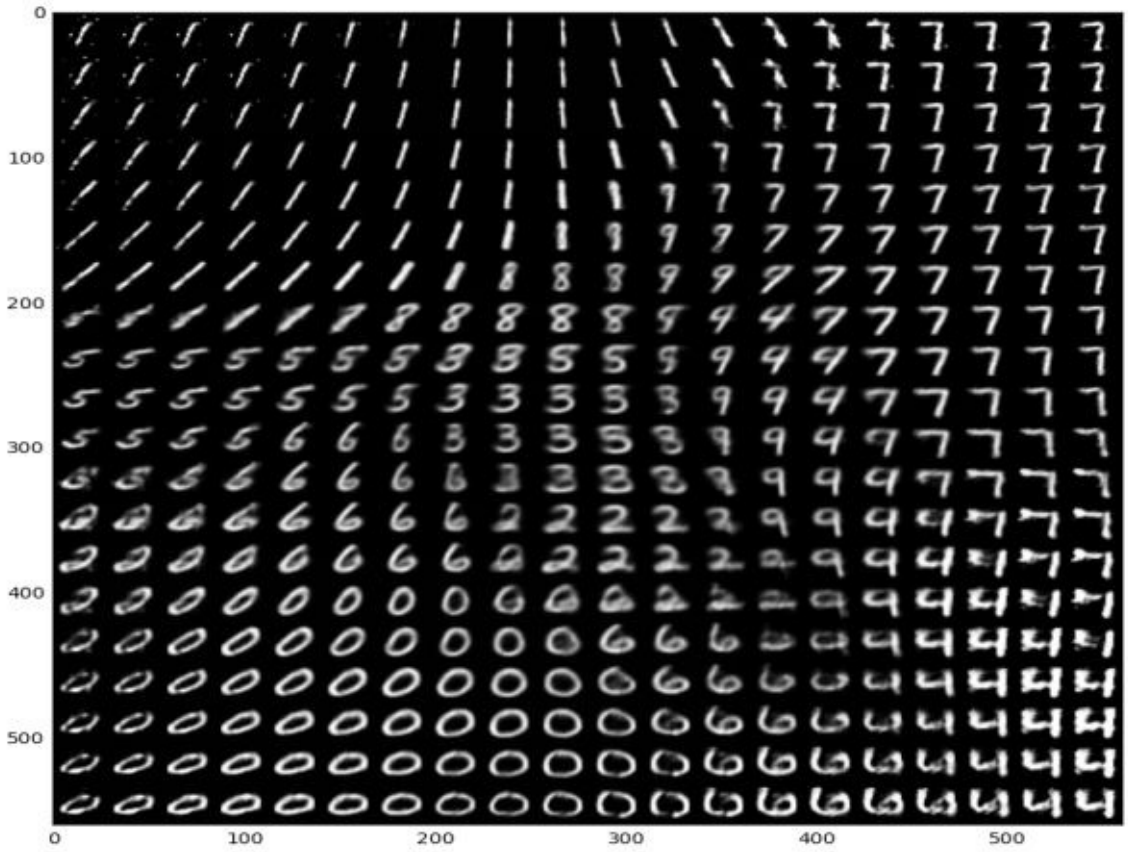
```
Epoch: 0031 cost= 142.209316600
Epoch: 0036 cost= 141.562141529
Epoch: 0041 cost= 140.994664584
Epoch: 0046 cost= 140.519218500
Epoch: 0051 cost= 140.079906838
Epoch: 0056 cost= 139.805822754
Epoch: 0061 cost= 139.486802576
Epoch: 0066 cost= 139.185289986
Epoch: 0071 cost= 138.955365503
```

```
x_sample, y_sample = mnist.test.next_batch(5000)
z_mu = vae_2d.transform(x_sample)
plt.figure(figsize=(8, 6))
plt.scatter(z_mu[:, 0], z_mu[:, 1], c=np.argmax(y_sample, 1))
plt.colorbar()
plt.grid()
```




```
nx = ny = 20
x_values = np.linspace(-3, 3, nx)
```

```
y_values = np.linspace(-3, 3, ny)

canvas = np.empty((28*ny, 28*nx))
for i, yi in enumerate(x_values):
  for j, xi in enumerate(y_values):
     z_mu = np.array([[xi, yi]]*vae.batch_size)
     x_mean = vae_2d.generate(z_mu)
     canvas[(nx-i-1)*28:(nx-i)*28, j*28:(j+1)*28] = x_mean[0].reshape(28, 28)

plt.figure(figsize=(8, 10))
Xi, Yi = np.meshgrid(x_values, y_values)
plt.imshow(canvas, origin="upper", cmap="gray")
plt.tight_layout()
```



Creating a Python Environment

Great, you have almost reached there and we have arranged everything; close every one of the shells you have opened and open another (which will guarantee the whole past foundation is associated). By then make yourself a venture envelope for continuing with. I will call my tf-learn (you can name it whatever you need), and in that coordinator, we make a virtual area called tf-venv (yet again, you can name it whatever you need).

```
$ cd tf-learn
$ virtualenv -p python3 tf-venv
```

At that point, make an .env record to make autoenv work for you.

```
$ echo source `pwd`/tf-venv/bin/activate > .env
$ cat .env
```

Some output is shown. At that point, just attempt a cd one more level up and back once more.

```
$ cd ..
$ cd tf-learn
autoenv:
autoenv: WARNING:
autoenv: This is the first time you are about to source /Users/fuyang/tf-learn/.env:
autoenv:    source /Users/fuyang/tf-learn/tf-venv/bin/activate$
```


```
$ cd tf-learn
$ virtualenv -p python3 tf-venv
```

At that point, make an .env record to make autoenv work for you.

```
$ echo source `pwd`/tf-venv/bin/activate > .env
$ cat .env
```

Some output is shown. At that point, just attempt a cd one more level up and

back once more.

```
$ cd ..
$ cd tf-learn
autoenv:
autoenv: WARNING:
autoenv: This is the first time you are about to source /Users/fuyang/tf-learn/.env:
autoenv:    source /Users/fuyang/tf-learn/tf-venv/bin/activate$
```

Essentially input y and enter. By then you should see (tf-env) show up before your bash note. (Remember you can use command $ deactivate to deactivate nature at whatever point you are inside it. Check these commands:

```
$ which python
/Users/fuyang/tf-learn/tf-venv/bin/python

$ which pip
/Users/fuyang/tf-learn/tf-venv/bin/pip

$ python —version
Python 3.6.0
```

If you have information that shows up like that, perfect.

You can upgrade Tensorflow using these lines:

```
$ pip install —upgrade https://storage.googleapis.com/tensorflow/mac/cpu/tensorflow-1.0.1-py3-none-any.whl
$ pip show tensorflow
```

Example:

Next, we investigate an exceptionally straightforward case. Let us name it

“example-1”.

import tensorflow as tf

```
a = tf.placeholder(“float”)
b = tf.placeholder(“float”)
y = tf.multiply(a, b)
with tf.Session() as sess:
  result = sess.run(y, feed_dict={a: 6, b: 7})
  print(result)
```

By running it, you should see a yield of 42. Now we have a super essential Tensorflow program working!

Basic of Python Language in Machine Learning

Objective of this Chapter

At the end of this chapter, the reader should have learned:

- Machine Learning application with Python
- Starting with SciPy
- Load data in Python
- Making basics data analysis
- Making predictions

The point is to take a newcomer from negligible information of machine learning in Python all the way to the knowledgeable practitioner. The prime target of this section is to enable you to swim through the various free choices that are accessible; there are many, no doubt, yet which are the best. Which complement each other? What is the best order in which to utilize chosen assets?

- Advancing, I make the presumption that you are not a specialist in:
- Machine learning
- Python

Beginners Need a Small End-to-End Project

Python is a prominent and effective interpreted language. Python is an entire language and platform that you can use for both innovative work and create production systems.

There is likewise a considerable measure of modules and libraries to look over, giving various approaches to do each task. It can feel overpowering.

The ideal approach to begin utilizing Python for machine learning is to finish a task.

- It will constrain you to install and begin the Python interpreter (in any event).
- It will give you a superior perspective of how to advance through a little project.
- It will give you certainty, possibly to go ahead with your own little tasks.

The learning from Books and courses you will be frustrated within a week.

They give you heaps of formulas and bits, yet you never get the chance to perceive how they all fit together.

When you are applying machine learning to your own datasets, you are dealing with a project.

A machine learning project may not be direct, but rather it has various understood advances:

- Define Problem
- Get the Data ready
- Evaluate Algorithms
- Enhance Results
- Presents Results

The ideal approach to truly deal with any new platform or tool is to work through a machine learning project end-to-end and cover the key factors. In particular, from data loading, data summarizing, implementing algorithms and making a few predictions.

On the off chance that you can do that, you have a format that you can use on dataset after dataset. You can fill in the gaps; for example, facilitate information planning and enhancing result undertakings later, once you have more confidence.

Hello World of Machine Learning

The best little task to begin with on new tool is the classification of iris flowers (e.g. the iris dataset).

This is a decent task since it is so surely knowing.

- Attributes are numeric so you need to make sense of how to load and handle data.

- It is an order issue, enabling you to practice with maybe a less demanding kind of supervised learning algorithm.
- It is a multi-class classification issue (multi-ostensible) that may require some particular dealing with.
- It just has 4 attributes and 150 rows, which means it is little and effortlessly fits into memory (and a screen or A4 page).

The greater part of the numeric characteristics is in similar units and a similar scale, not requiring any unique scaling or changes to begin.

How about we begin with your hello world machine learning project in Python.

Machine Learning in Python: Step-By-Step

In this segment, we will work on a little machine learning project end-to-end.

Here is an outline of what we will cover:

- Installing the Python and SciPy platform.
- Loading the dataset.
- Summarizing the dataset.
- Visualizing the dataset.
- Evaluating some algorithms.
- Making some predictions

Take your time. Work through each step.

Try to type in the commands yourself or copy-and-paste the commands to speed things up.

Downloading, Installing and Starting Python SciPy

We have already discussed the installation procedure of python in the previous section. So we will not be discussing this again. There is no difficulty in installing python on your laptops especially if you are a developer. There are many tutorials available on YouTube if you don't understand from the previous section because it might be tough for a beginner who knows nothing about programming, but if you are familiar with it then it won't be a problem for you.

Install SciPy Libraries

We recommend you to install Python version 2.7 or 3.5. Because the example we have discussed in this section uses these versions.

Before proceeding forward, you need to install these five libraries of SciPy because we are using these libraries to solve this machine learning small project. Here is the list of these libraries:

- scipy
- numpy
- matplotlib
- pandas
- sklearn

There are numerous approaches to install these libraries. My best counsel is to pick one strategy and be confident in introducing every library that way.

- The scipy establishment page gives magnificent guidelines for introducing the above libraries on numerous diverse stages, for example, Linux, macintosh OS X and Windows
- On Linux, you can utilize your package manager, for example, yum on Fedora to introduce RPMs.4

If you are on Windows or you are not certain, I would prescribe introducing the free form of Anaconda that incorporates all that you require.

Start Python and Check Versions

It is a smart thought to ensure your Python environment was installed effectively and in working condition

The code below will enable you to test out your environment. It imports every library required in this instructional exercise and prints the version.

Open a charge line and begin the python interpreter:

Being a beginner, I will recommend you work directly on python interpreter and writing your scripts and running them on the command line rather than using any big editors and IDE's. You need to focus on keeping the simple in machine learning while in a learning stage, not the toolchain.

It is up to whether you type the script by yourself and copy paste it.

```
# Check the versions of libraries

# Python version
import sys
print('Python: {}'.format(sys.version))
# scipy
import scipy
print('scipy: {}'.format(scipy.__version__))
# numpy
import numpy
print('numpy: {}'.format(numpy.__version__))
# matplotlib
import matplotlib
```

```
print('matplotlib: {}'.format(matplotlib.__version__))
# pandas
import pandas
print('pandas: {}'.format(pandas.__version__))
# scikit-learn
import sklearn
print('sklearn: {}'.format(sklearn.__version__))
```

The output of the above given code is as follows

```
Python: 2.7.11 (default, Mar  1 2016, 18:40:10)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)]
scipy: 0.17.0
numpy: 1.10.4
matplotlib: 1.5.1
pandas: 0.17.1
sklearn: 0.18.1
```

Compare the above output to your versions

If you have installed python successfully, your version should match or be more recent. The APIs do not change rapidly, so do not be excessively concerned in the event that you are a couple of versions behind, everything in this section exercise will work for you.

On the off chance that you get a mistake, stop. Right now, is an ideal opportunity to settle it. You have to install python first else you will not be able to proceed, so fix the installation procedure then move on to further in this section.

Load the Data

We will utilize the iris blossoms dataset. This dataset is renowned in light of the fact that it is utilized as the "hello world" dataset in machine learning and statistics by practically everybody.

The dataset contains 150 perceptions of iris flowers. There are four columns of measurements of the flowers in centimeters. The fifth section is the types of the flowers watched. Every observed flower has a place with one of three species

In this progression, we will load the iris information from CSV document URL.

To start with, we should import the greater part of the modules, functions, and objects we will use in this project.

```
# Load libraries
import pandas
from pandas.plotting import scatter_matrix
import matplotlib.pyplot as plt
from sklearn import model_selection
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
```

Everything should load without mistake. In the event that you have a blunder, stop. You require a working SciPy environment before proceeding.

We can load the data specifically from the UCI Machine Learning repository.

We are utilizing pandas to load the information. We will likewise utilize pandas beside investigate the data both with descriptive statistics and data visualization.

Note that we are indicating the names of every segment when loading the information. This will enable later us to investigate the information.

```
# Load dataset
url = “https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data”
names = [‘sepal-length’, ‘sepal-width’, ‘petal-length’, ‘petal-width’, ‘class’]
dataset = pandas.read_csv(url, names=names)
```

The dataset should load without any error

On the off chance that you do have network issues, you can download the iris.data record into your working registry and load it utilizing a similar technique, changing URL to the local file name.

Summarize the Dataset

Once the date is uploaded now it’s the time to look at the data. In this step, we will look at the data in different ways that are given below.

- Dimensions of the dataset.
- Peek at the data itself.
- Statistical summary of all attributes.
- Breakdown of the data by the class variable.

You need not worry about it, each look at the data is in one command. These commands are very useful that you can use in the future for different machine learning projects.

We can get a rough idea of how many rows and columns our data contains with flower shapes properly.

```
# shape
print(dataset.shape)
```

You should see 150 instances and 5 attributes

```
(150, 5)
```

It is also a good practice to peek at your data. You will see the first 20 rows of the data by using this command.

```
# head
print(dataset.head(20))
```

You should see the first 20 rows of the data

	sepal-length	sepal-width	petal-length	petal-width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa
10	5.4	3.7	1.5	0.2	Iris-setosa
11	4.8	3.4	1.6	0.2	Iris-setosa
12	4.8	3.0	1.4	0.1	Iris-setosa
13	4.3	3.0	1.1	0.1	Iris-setosa

14	5.8	4.0	1.2	0.2	Iris-setosa
15	5.7	4.4	1.5	0.4	Iris-setosa
16	5.4	3.9	1.3	0.4	Iris-setosa
17	5.1	3.5	1.4	0.3	Iris-setosa
18	5.7	3.8	1.7	0.3	Iris-setosa
19	5.1	3.8	1.5	0.3	Iris-setosa

Now let’s have a look at the summary of each attribute. By summary, we mean that it will include count, mean, min, max and percentiles of the given data.

```
# descriptions
print(dataset.describe())
```

We can see that all of the numerical values have the same scale (centimeters) and similar ranges between 0 and 8 centimeters

	sepal-length	sepal-width	petal-length	petal-width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

After viewing the summary of the data let’s have a look at the number of rows belongs to each class.

```
# class distribution
```

```
print(dataset.groupby('class').size())
```

We can see that each class has the same number of instances (50 or 33% of the dataset

```
class
Iris-setosa       50
Iris-versicolor   50
Iris-virginica    50
```

Data Visualization

Now we have a basic idea about the data we have. Now we can extend the data with some visualization. For this, we are going to have a look at the two types of plots.

Univariate plots to better understand each attribute

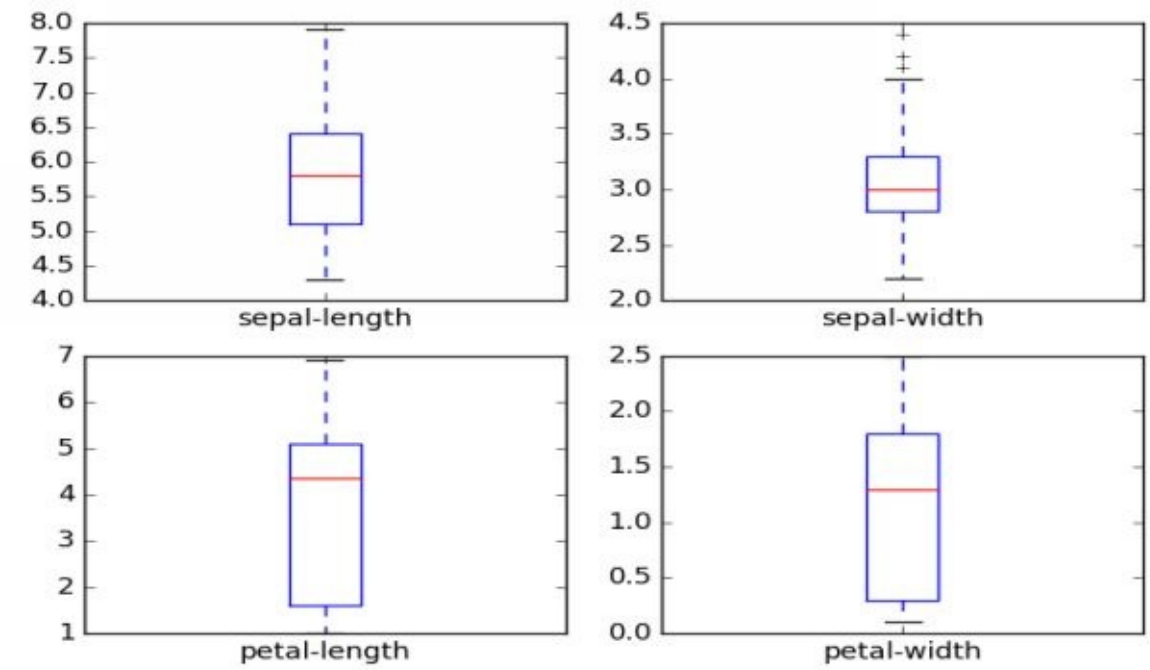
Multivariate plots to better understand the relationships between attributes.

Univariate Plots

We will start with this type of plot that will show the plot of each individual

```
# box and whisker plots
dataset.plot(kind='box', subplots=True, layout=(2,2), sharex=False, sharey=False)
plt.show()
```

This gives us a much clear idea of the distribution of the input attributes:

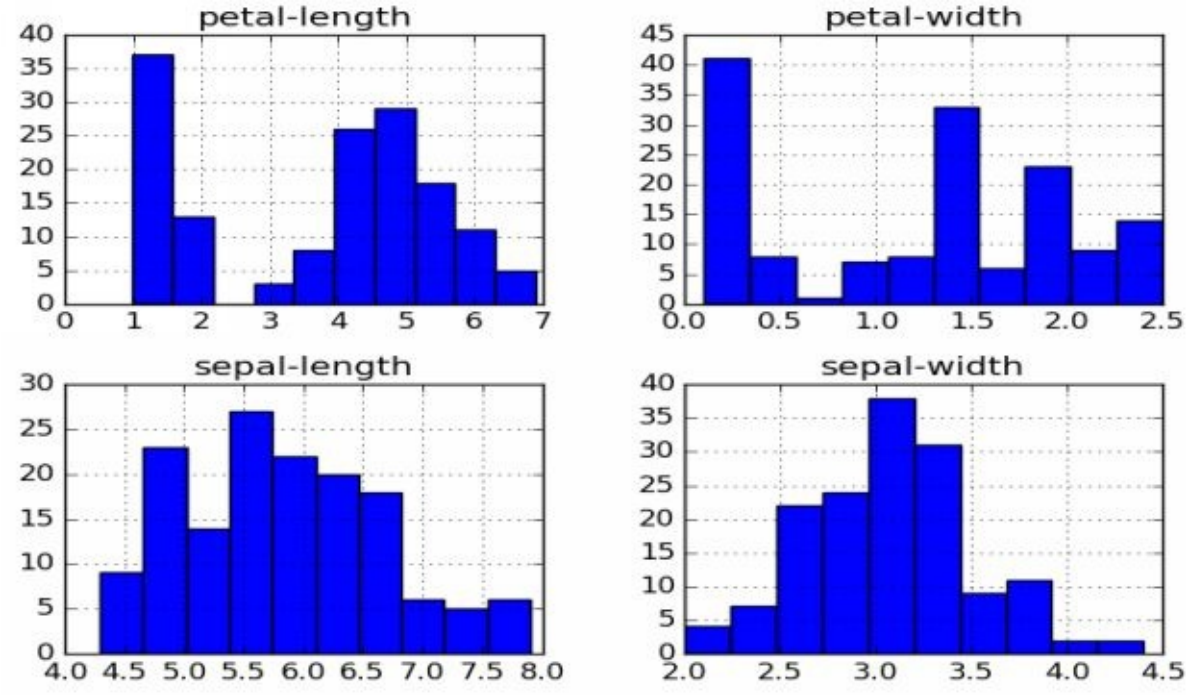


Box and Whisker Plots

We can also create a histogram of each input variable to get an idea of the distribution.

```
# histograms
dataset.hist()
plt.show()
```

It looks like perhaps two of the input variables have a Gaussian distribution. This is useful to note as we can use algorithms that can exploit this assumption.

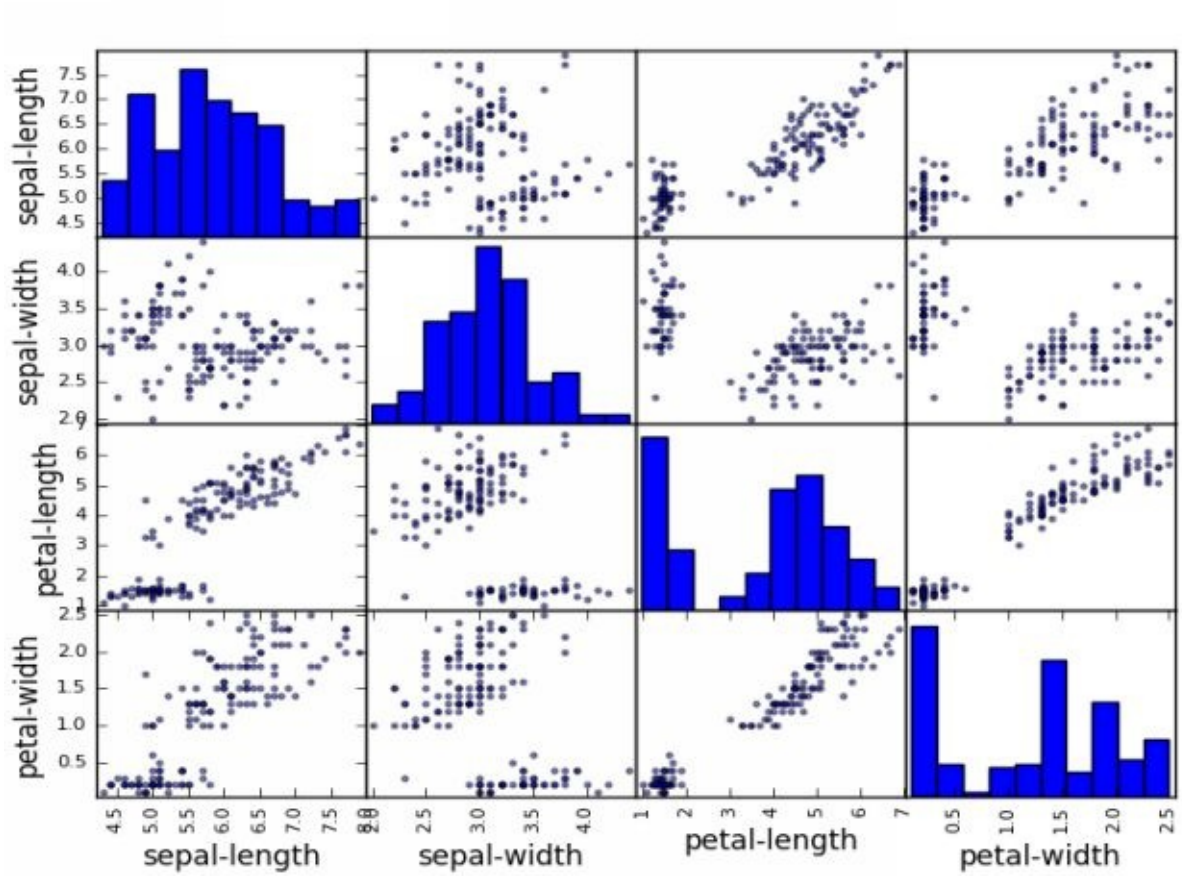


Multivariate Plots

In this type of plot, we will look at the interaction between the variables. For this, we need to have a look at scatterplots of all pairs of attributes we have. This can be helpful to spot structured relationships between input variables we have in our dataset.

```
# scatter plot matrix
scatter_matrix(dataset)
plt.show()
```

Note the diagonal grouping of some pairs of attributes. This suggests a high correlation and a predictable relationship.



Evaluate Some Algorithms

Now we have almost at the end of this project for beginners. It is the time to build some models for our data and check their accuracy on unseen data. To do this we need to follow following steps.

- Separate out a validation dataset.
- Set-up the test harness to use 10-fold cross validation.
- Build 5 different models to predict species from flower measurements
- Select the best model.

We have to realize that the model we made is any great.

Afterward, we will utilize factual strategies to estimate the exactness of the models that we make on unseen data. We additionally need a more concrete estimate of the precision of the best model on unseen data by assessing it on real unseen data.

That is, where we are going to hold back and some of the data that the algorithms will not get to see and we will then use this data to get a second and independent idea of how accurate the best model might actually be.

We will split the loaded dataset into two, 80% of which we will use to prepare our models and 20% that we will keep down as an approval dataset.

```
# Split-out validation dataset
array = dataset.values
X = array[:,0:4]
Y = array[:,4]
validation_size = 0.20
seed = 7
X_train, X_validation, Y_train, Y_validation = model_selection.train_test_split(X, Y, test_size=validation_size, random_state=seed)
```

You now have training data in the X_train and Y_train for preparing models and X_validation and Y_validation sets that we can use later.

To estimate the accuracy of our data we will use 10-fold cross validation. This will split our data into 10 parts means to train on 9 and test on 1 and repeat for all the combinations of train test splits.

```
# Test options and evaluation metric
seed = 7
scoring = ‘accuracy’
```

We are using the metric of ‘*accuracy*’ to evaluate models. This is a ratio of the number of correctly predicted instances in divided by the total number of instances in the dataset multiplied by 100 to give a percentage (e.g. 95% accurate). We will be using the *scoring* variable when we run build and evaluate each model next.

Now we do not know which algorithm will be good for this problem or what configuration do we have to use. We have an idea from the plots we made. Now let's evaluate 6 different algorithms.

- Logistic Regression (LR)
- Linear Discriminant Analysis (LDA)
- K-Nearest Neighbors (KNN).
- Classification and Regression Trees (CART).
- Gaussian Naive Bayes (NB).
- Support Vector Machines (SVM).

This is a good mixture of basic linear (LR and LDA), nonlinear (KNN, CART, NB, and SVM) calculations. We reset the irregular number seed before each run to guarantee that the assessment of every calculation is performed utilizing the very same information parts. It guarantees the outcomes are specifically directly comparable.

We should construct and assess our five models:

```
# Spot Check Algorithms
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC()))
# evaluate each model in turn
results = []
names = []
for name, model in models:
        kfold = model_selection.KFold(n_splits=10, random_state=seed)
```

```
        cv_results = model_selection.cross_val_score(model, X_train, Y_train, cv=kfold,
scoring=scoring)
        results.append(cv_results)
        names.append(name)
        msg = “%s: %f (%f)” % (name, cv_results.mean(), cv_results.std())
        print(msg)
```

Now we are almost there and we have six models and accuracy estimation of each model now is the time to select the model out of six models. We need to compare all six models and pick the best one out.

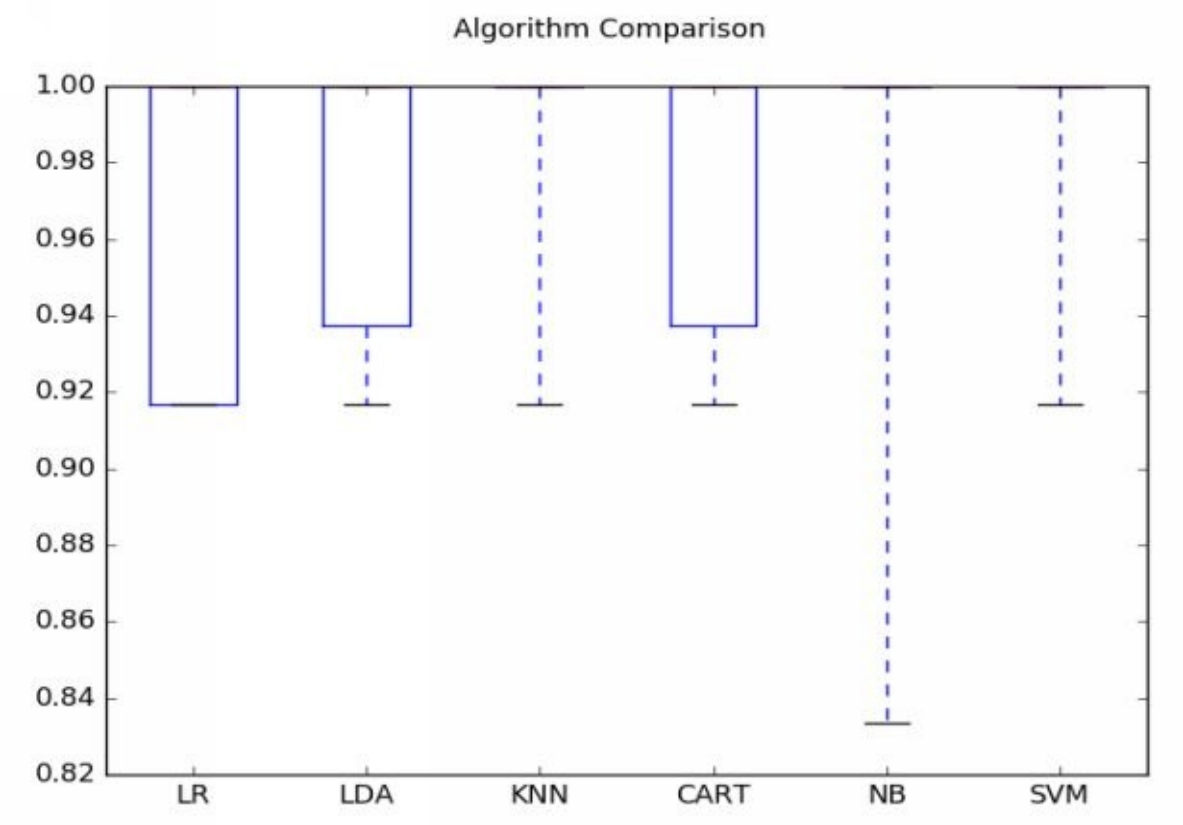
```
LR: 0.966667 (0.040825)
LDA: 0.975000 (0.038188)
KNN: 0.983333 (0.033333)
CART: 0.975000 (0.038188)
NB: 0.975000 (0.053359)
SVM: 0.981667 (0.025000)
```

We can see that it looks like KNN has the largest estimated accuracy score.

We can also create the plot of each model and compare the spread and the mean accuracy of all six models.

```
# Compare Algorithms
fig = plt.figure()
fig.suptitle(‘Algorithm Comparison’)
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```

You can see that the box and whisker plots are squashed at the top of the range, with many samples achieving 100% accuracy.



Make Predictions

The KNN calculation was the most precise model that we tried. Presently we need to get an idea of the accuracy of the model on our validation set.

This will give us independent final check on the estimation of the best model. It is profitable to keep an approval set just in the event that you influenced a slip amid training, for example, overfitting to the training set or a data leak. Both will result in an overly optimistic result.

We can run the KNN display specifically on the approval set and condense the outcomes as a last exactness score, a confusion matrix, and a classification report.

```
# Make predictions on validation dataset
knn = KNeighborsClassifier()
knn.fit(X_train, Y_train)
```

```
predictions = knn.predict(X_validation)
print(accuracy_score(Y_validation, predictions))
print(confusion_matrix(Y_validation, predictions))
print(classification_report(Y_validation, predictions))
```

We can see that the precision is 0.9 or 90%. The confusion matrix gives a sign of the three blunders made. At long last, the order report gives a breakdown of each class by exactness, review, f1-score and bolster demonstrating brilliant outcomes (allowed the validation dataset was little).

```
0.9

[[ 7  0  0]
[ 0 11  1]
[ 0  2  9]]

                 precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00         7
Iris-versicolor       0.85      0.92      0.88        12
 Iris-virginica       0.90      0.82      0.86        11

avg / total       0.90      0.90      0.90        30
```

You Can Do Machine Learning in Python

Through the instructional exercise end-to-end that we have done to get an outcome. You don’t have to understand everything on the primary pass. Rundown down your inquiries as you go. Make heavy use of the help (“Function Name”) help syntax in Python to learn about all of the functions that you’re using.

You don’t have to know how the algorithms work. It is essential to think about the confinements and how to arrange machine learning algorithms. Be that as it may, finding out about algorithms can come later. You have to develop this algorithm learning gradually over a long period of time. Today, start off by getting comfortable with the platform.

You don’t need to be a Python software engineer. The sentence structure of the Python language can be intuitive in the event that you are unfamiliar with it. Just like other languages, focus on function calls (e.g. function ()) and assignments (e.g. a = “b”). This will get you more than halfway. You are a designer, you know how to get the nuts and bolts of a language genuine quick. Simply begin and jump into the points of interest later.

You shouldn’t be a machine learning master. You can find out about the advantages and constraints of different algorithms later, and there are a lot of posts that you can read later to look over the means of a machine learning project and the significance of assessing precision utilizing cross approval.

Shouldn’t something be said about the different steps in a machine learning project? We didn’t cover the greater part of a machine learning project since this is your first task and we have to focus on the key features. Namely, loading data, looking at the data, evaluating some algorithms and making some predictions.

Data and Inconsistencies in Machine Learning

Objective of this Chapter

This chapter will cover all the data inconsistencies and what the learning problems we face while implementing machine learning projects. There are several types of data inconsistencies given below:

- Under-fitting
- Over-fitting
- Data instability
- Unpredictable future
- Model accuracy

Under-fitting

This model is said to be under fitting when it does not take into consideration enough information to accurately model the actual data for instance, if we just have two points on an exponential curve that are mapped, this potentially turns into a linear representation, however, there could be where a pattern does not exist. In cases like these, we will see increasing errors and often an inaccurate model.

Likewise, in situations where the classifier is excessively inflexible or is not complex enough, under-fitting is caused not only due to an absence of information, yet can likewise be an aftereffect of an inaccurate model. For instance, if there are two classes namely shape and concentric circles and we attempt to fit a linear model, expecting they were linearly separable, this could conceivably result in under-fitting.

The exactness of the model is controlled by a measure called “power” in the statistical world. In the event that the dataset measure is too little, we can never focus on an optimal solution.

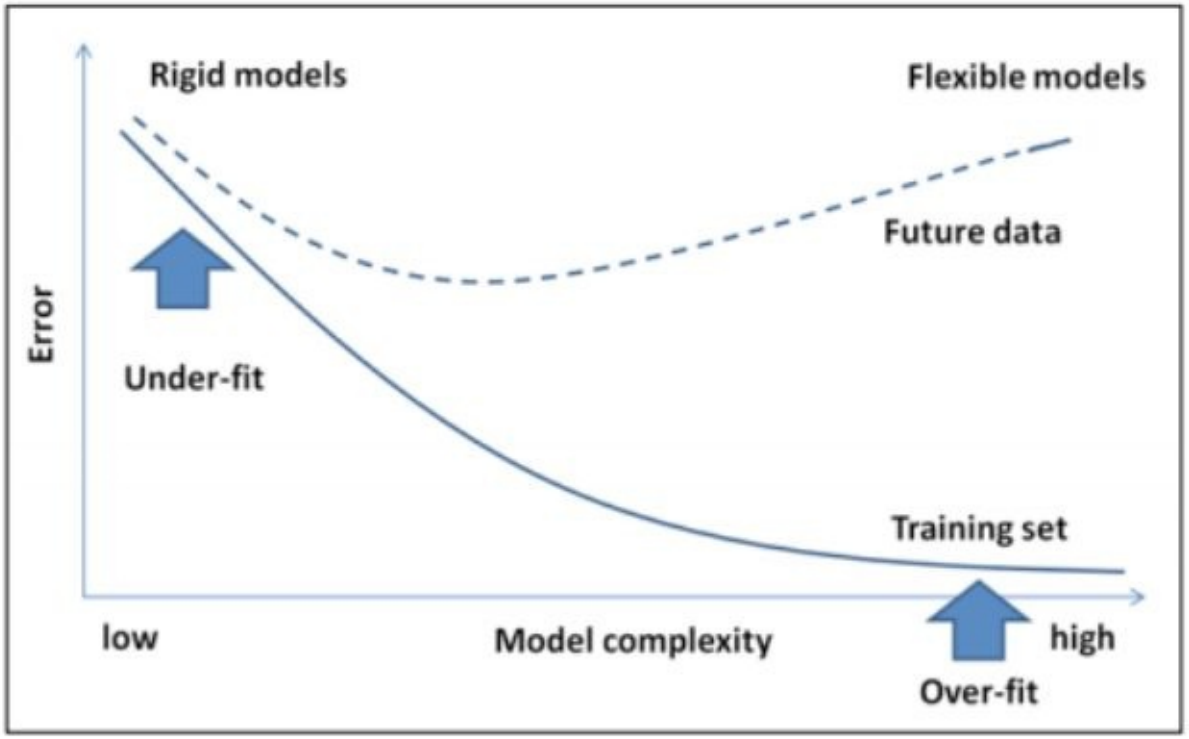
Over-fitting

This case is totally opposite from the under-fitting case that we have discussed before in this section. While a small sample is not appropriate to provide the optimal solution, the larger datasets have a risk of having the model overfit the data. Over-fitting normally happens at the point when the statistical model portrays noise instead of describing the relationships

Explaining on the previous case in this specific circumstance, suppose we have 500,000 data points. On the off chance that the model winds up taking into account oblige every one of the 500,000 data points, this ends up as overfitting. This will actually mean that the model is memorizing the data. This model functions admirably as long as the dataset does not have points outside the curve.

A model that is over-fit exhibits poor performance as minor fluctuation in information has a tendency to be misrepresented. The essential the reason behind over fitting likewise could be that the standard used to train the model is not the same as the standard used to judge the accuracy of the model. In basic terms, if the model remembers the training information instead of learning it, this circumstance apparently occurs more frequently.

Presently, during the time spent alleviating the issue of under-fitting the information, by giving it more information, this can in itself be a hazard and wind up in over fitting. Thinking about that more data can mean greater complexity and noise, we could possibly wind up with a solution model that fits the present data close by and that’s it, which makes it unusable. In the accompanying chart, with the increasing model complexity and errors, the conditions for over-fit and under-fit are pointed out:



Data instability

Machine learning calculations are normally robust to noise in the data. An issue will happen if the exceptions are because of manual blunder or misinterpretation of the data. This will bring about a skewing of the data, which will at last wind up as an incorrect model.

Accordingly, there is a solid need for a process to correct or handle human mistakes that can bring about building an incorrect model.

Unpredictable data formats

Machine learning is intended to work with new data continually coming into the system, what's more, learning from that data. Many-sided quality will sneak in when the new data entering the system comes in designs that are not supposed by the machine learning system. It is presently hard to state if our models function admirably for the new data given the instability in the formats that we receive the data unless there is a mechanism built to handle this.

Practical Machine learning examples

In this segment, we should investigate some genuine machine learning applications. We discussed different examples in the introductory section, we will now cover some domain-specific examples with a brief description of each problem.

Problem / problem Domain	Description
Spam detection	The problem statement here is to identify which e-mails are "spam". A Machine learning algorithm can categorize an e-mail to be marked as spam based on some rules that it builds using some key features of e-mail data. Once an e-mail is marked as spam, that e-mail is then moved to the spam folder and the rest are left in the inbox.
Credit card fraud detection	This is one of the recent problems that credit card firms need a solution for. Based on the usage patterns of the credit card by the consumer and the purchase behavior of the customer, the need is to identify any transaction that is not potentially made by the customer and mark them as fraudulent for necessary action to be taken.
Digit recognition	This is a very simple use case that requires the ability to group posts based on the zip code. This includes the need to interpret a handwritten numeric accurately and bucket the posts based on the zip code for faster processing.
Speech recognition	Automated call centers need this capability where a user's request on the phone is interpreted and mapped to one of the tasks for execution. The moment the user request can be mapped to a task, its execution can be automated. A model of this problem will allow a program to understand and make an attempt to fulfill that request. The iPhone with Siri has this capability.

Problem / problem Domain	Description
Face detection	This is one of the key features that today's social media websites provide. This feature provides an ability to tag a person across many digital photographs. This gives aptitude to a group or categorizes the photographs by a person. Some cameras and software such as iPhoto have this capability.
Product recommendation or customer segmentation	This capability is found in almost all of the top online shopping websites today. Given a purchase history for a customer and a large inventory of products, the idea is to identify those products that the customer will most likely be interested in buying, thus motivating more product purchases. There are many online shopping and social websites that support this feature (for example: Amazon, Facebook, Google+, and many others). There are other cases like the ability to predict whether a trial version customer opts for the paid version of the product.
Stock trading	This means predicting stock performance based on the current past stock movement. This task is critical to financial analysts and helps provide decision support when buying and selling stocks.
Sentiment analysis	Many times, we find that the customers make decisions based on opinions shared by others. For example, we buy a product because it has received positive feedback from the majority of its users. Not only in commercial businesses as detailed earlier, but sentiment analysis is also being used by political strategists to gauge public opinion on policy announcements or campaign messages.

Building intelligent machines to transform data into knowledge

In this time of current innovation, there is one asset that we have in plenitude: a lot of organized and unstructured information. In the second 50% of the twentieth century, machine learning developed as a subfield of Artificial Intelligence (AI) that included self-learning in an algorithm that got data from data with a specific end goal to make predictions. Rather than expecting people to physically determine principles and manufacture models from breaking down a lot of information, machine learning offers a more productive option for catching the information in information to bit by bit enhance the execution of prescient models and settle on information-driven choices. Not only is machine learning becoming increasingly important in computer science research, but it also plays an ever-greater role in our everyday lives.

In the training stage, the algorithm is prepared and balanced until the point when it creates the ideal outcome. On account of the most recent variant of Syngo via, such Deep Learning calculations automatically detect and segment related structures - the heart, the fundamental corridor or the lungs - without time-consuming, manual processing. The basic Deep Learning algorithms were trained with a variety of computed tomography data sets. Subsequently, photorealistic images are created based on the cinematic volume rendering technique. This permits notwithstanding for complex information to be changed over into an effectively reasonable visual language, which is particularly for the correspondence with referrers and patients helpful.

The more the data is available for training, the better the system can solve up the issue, and the more exact and strong the result can be. Today, enough medical data is available to implement complex, multi-layer networks based on extremely high computing capacities. The analytic procedure produces a gigantic volume of subjective picture information, lab results, and pathological and radiological findings.

Intelligent assistants for Radiology

This topic can be about expanding significance for radiologists since they too confront real difficulties. Consistently, the Royal College of Radiologists performs reviews of all radiology units in Britain's National Health Service. Between 2013 and 2016, the association recorded an expansion of more than 30 percent in computed tomography (CT) and magnetic resonance imaging (MRI) examinations – three times as high as the development in Radiology workforce. A similar report demonstrates that exclusive three percent of healing facility radiology divisions are equipped for managing all their patient outputs amid typical working hours.

Another examination demonstrated that the mistake rate for the taking an interest, radiologists were 10%, in view of a normal translation time of ten minutes and nine seconds. If they had to interpret comparable CT images in half that time, the error rate expanded to 26.6 percent.

Siemens Health engineers are constantly working on intelligent algorithms to help the healthcare providers remain competitive in the face of increasing patient numbers and a shortage of skilled staff to ensure consistent diagnoses – regardless of the patient, the technologist or the individual assessing the image. The understanding and prioritization of routine pictures – which may likewise be liable to a lower repayment rate – is exactly where AI can enable radiologists to accomplish, for instance, a quicker assessment so they can give more opportunity to more perplexing cases

Making predictions about the future with supervised learning

We do not know what will happen and what will not, future is uncertain. Fortunately, by utilizing some machine learning ‘black magic’ we will investigate approaches to predict future. Estimating the future has been one of man’s interests for quite a long time. From antiquated Babylon’s forecasters, meteorologists detailing the figure on tomorrow’s climate, forecasting future changes in the stock markets predicting the future has become part of our human ‘evolution’.

In this topic, we will focus on anticipating from information, which is referred as Time series data. Time series is data changes over the time and is caught in (time, value) or (x, y) sets with x being time. The examples time series data include stock prices, yearly rainfall, inflation, profits, etc. The area in which I am interested in, is the load on a server in the cloud.

Classical Machine learning

Before moving on to how to predict on time series, let us initially examine machine learning. Classical machine learning expects to group information into classifications named classes. An outstanding case is one where we have data points gathered from flowers that have a place with the Iris. For each flower sample, the following was measured: the length and width of the sepals and petals and these four estimations form what machine learning refers to as features. Utilizing these features, we prepare a classifier (a scientific model) with a portion of the data gathered, in each preparation case, telling the model what species the specific example has a place with. After the classifier-display has been trained, it is tried by giving another example the label unknown and makes a prediction on which animal categories the new example has a place. This is called statistical classification with supervised learning.

Time Series

Classical machine learning utilizes features, similar to the petal and sepal lengths of flowers or the side effects displayed by a patient or components in a wine in order to make some decision on a new sample. Conversely, time series do not contain a gathering of features, yet rather a value at each time step. Time series prediction accepts that the future relies upon the past and utilizing these assumptions it sees each time step back; x[t-1], x[t-2], … as features. Time series regularly likewise has at least one of the accompanying qualities; it contains randomness, follows a trend and has seasonal and/or cyclical behavior. Predicting or forecasting a value (or set of qualities) into

what has to come is the ‘decision’ the machine-learning model must make. The question now is how does one approach a time series prediction problem?

Steps in Forecasting

The mentioned below are the following generic steps in forecasting time series:

At first, we have to define the problem to be solved. It is very important to consider the final result of the forecast will be and how the forecast final result will be used.

- **Gathering of Data**.

  The famous saying is “There is no data like more data”, more information empowers better preparing, testing, and approval.

- **Examine your Data.**

  Utilize statistical tools to plot distinctive data plots and decide the qualities of your data; does it contain patterns, is it excessively random or does it follow a trend

- **Pick models and fit them.**

  This can be basic models, for example, utilizing the mean of past data points or complex methodologies, for example, neural systems. I will give a short overview of models as of now connected to time series, beneath.

- **Assess your model accuracy.**

  This involves estimating error metrics and optimizing model parameters.

I will now examine stages 3-5 in more depth. Stage 1 and 2 are direct and depends on the particular issue you are confronting.

Examine your Data

By examining the data our aims are to get inside the data that means discovering the structure of data, behaviors of data, patterns in data, outliers and other information to help us choosing a forecasting model. A key assumption of time series prediction is that the data is Stationary. In stationary process joint probability, the distribution does not change over time. The tools to determine the stationarity of data include calculating the Auto-correlation of the data or using the Augmented Dickey–Fuller test. Another important factor in data, which is to explore, is to determine if our data follows a trend

or contains a seasonal behavior. Some models that we have are not capable enough to accurately predicting the data, which contains a trend, or periodic pattern and thus one will typically remove these behaviors through transforming the data.

Pick models and fit them

The least difficult ‘model’ is calculating a mean in the light of the past few samples and utilizing this as a prediction. One-step up is to plot a Histogram of your data and utilize the averages of the examples in the most significant bin. Moving Average (MA) models computes averages of the examples in a sliding window after some time, with Exponential Smoothing weighting each lagged sample exponentially less Auto-Regressive (AR) models intend to fit a function f(t) to a prediction y(t) = f(t) + e, where e catches the irregular error in the data. Joining MA and AR models, give ARMA (p,q) models with arrange (p,q) defining the order of AR(p) and MA(q) individually.

More developed models incorporate Markov chains and neural systems. The thought is that one can segregate a time series value into M bin, which at that point means M states and utilizing a Markov model, predict the probability of transitioning with one state then onto the next for each time step. The Neural system intends to learn patterns from past values and predicts another value or a probability distribution for a set of values into the future. Other advanced methods include combining different models or vectorizing a collection of time series data in order to improve model accuracy.

Assess your model accuracy

Once we have chosen which model is fit and forecasting some values it is very important to be able to measure the accuracy of the model we have chosen and predictions we have made.

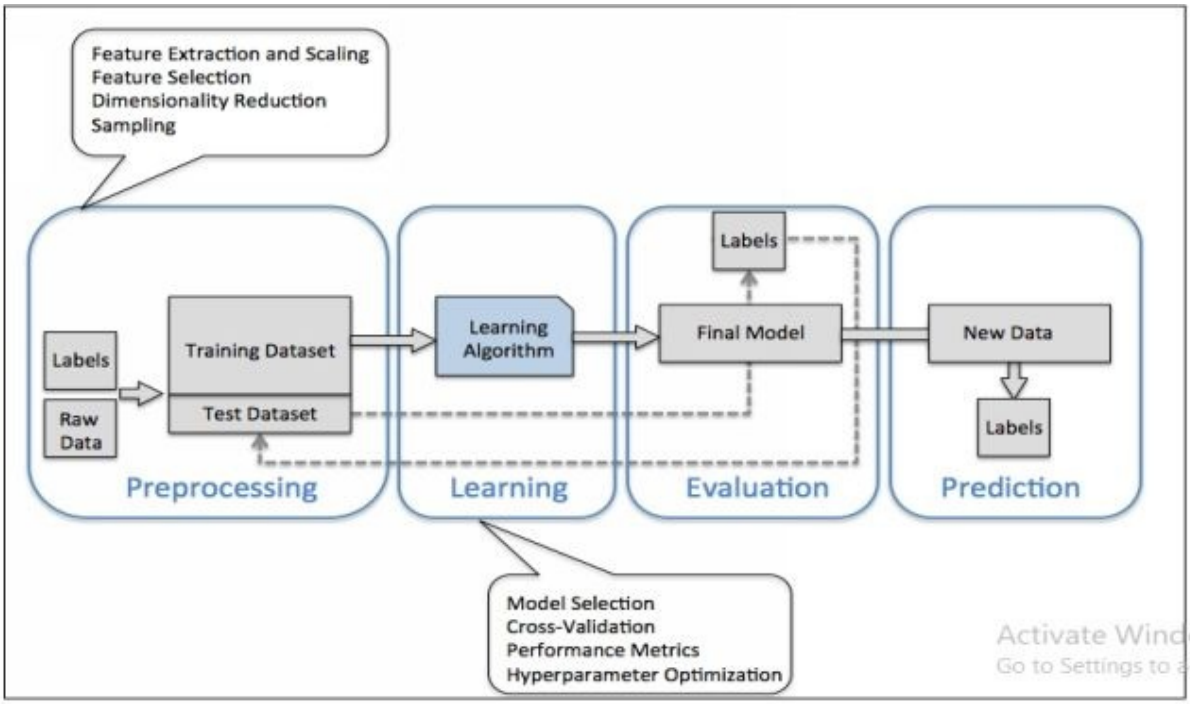
The basic metric to use is to calculate the Root Mean Squared of the Error (RMSE) also referred to as the Root Mean Squared Deviation (RMSD); calculated by: $\mathrm{RMSD} = \sqrt{\frac{\sum_{t=1}^{n}(\hat{y}_t - y_t)^2}{n}}$. With $\hat{y}_t$ being the predicted value and y the actual value at time *t.*

The Future is still uncertain because we just make predictions and check their accuracy and that doesn’t mean nothing will happen against our prediction, however utilizing Forecasting models in machine learning, I conjured some ‘magic’ to empower us to foresee what’s to come. This topic in-short covered classical machine learning, namely classification with supervised learning and the focus was on the steps should take when forecasting on time series data. We have discussed the popular models currently being applied to time series and also mentioned advanced approaches using Markov models or neural networks.

A Roadmap for building Machine Learning Systems

Objective of this Chapter

In this chapter, we will examine other important parts of a machine learning system starting with the learning algorithm. The graph beneath demonstrates an average work process outline for utilizing machine learning in predictive modeling, which we will discuss in the following subsections:


Feature Extraction and Scaling
Feature Selection
Dimensionality Reduction
Sampling
Labels
Raw Data
Training Dataset
Test Dataset
Learning Algorithm
Labels
Final Model
New Data
Labels
Preprocessing
Learning
Evaluation
Prediction
Model Selection
Cross-Validation
Performance Metrics
Hyperparameter Optimization

Preprocessing – getting data into shape

The raw data infrequently comes in the frame and shape that is fundamental for the optimal performance of a learning algorithm. In this way, the preprocessing of the data stands out amongst the essential step in any machine learning application. If we take the Iris flower dataset from the previous section, for instance, we could think about the raw data as a series of flowers pictures from which we need to extricate significant features. Valuable features could be the color, the hue, the intensity of flowers, the tallness, and the flower lengths and widths. Numerous machine learning algorithms likewise require that the chosen features are on a similar scale for ideal execution, which is frequently accomplished by changing the features in the range [0, 1] or a standard typical distribution with zero mean and unit variance.

It might be possible that selected features might be much related and in this way repetitive to a specific degree. In those cases, dimensionality reduction techniques are valuable for compressing the features onto a lower dimensional subspace. Reducing the dimensionality of our feature space has the favorable position that less storage room is required, and the learning algorithm can run significantly quicker.

To decide if our machine learning algorithm performs well on the training set also generalize well to new data, we likewise need a random partition of the dataset into a different training and test set. We utilize the training set to train and optimize our machine learning model, while we keep the test set until the very end to assess the last model.

Training and selecting a predictive model

As we will see in later in this book, a wide range of machine learning algorithms have been created to take care of various problems we face? A vital point that can be summarized from David Wolpert’s popular No Free Lunch Theorems is that we can’t get learning “for free” (The Lack of A Priori Distinctions Between Learning Algorithms, D.H. Wolpert 1996; No Free Lunch Theorems for Optimization, D.H. Wolpert and W.G. Macready, 1997). Naturally, we can relate this idea to the popular saying, “I assume it is tempting, if only the tool you have is a hammer, to treat everything as though

it were a nail" (Abraham Maslow, 1966). For instance, every classification calculation has its natural biases, and no classification model enjoys superiority if we don't make any assumptions related to the task. By and by, it is accordingly basic to analyze no less than a handful features of different algorithms keeping in mind the end goal to train and select the best performing model. In any case, before we can think about various models, we initially need to settle on a metric to quantify execution. One regularly utilized metric is classification accuracy, which is defined as the proportion of correctly classified instances.

One honest question to ask is: how would we know which model performs well on the final test dataset and real-world data on the off chance that we don't utilize this test set for the model selection, however, keep it for the final model assessment? Keeping in mind the end goal to address the problem discussed in this question, diverse cross-validation strategies can be utilized where the training dataset is additionally isolated into training and validation subsets in order to estimate the generalization performance of the model. At long last, we additionally can't expect that the default parameters of the distinctive learning algorithms gave by programming libraries are ideal for our particular problem. So, we will make frequent utilization of hyper parameter optimization techniques that help us to fine-tune the performance of our model. Naturally, we can think about those hyper parameters as parameters that are not learned from the data but rather speak to the knobs of a model that we can use to enhance its execution, which will turn out to be much clearer when we see real illustrations.

Evaluating models and predicting unseen data instances

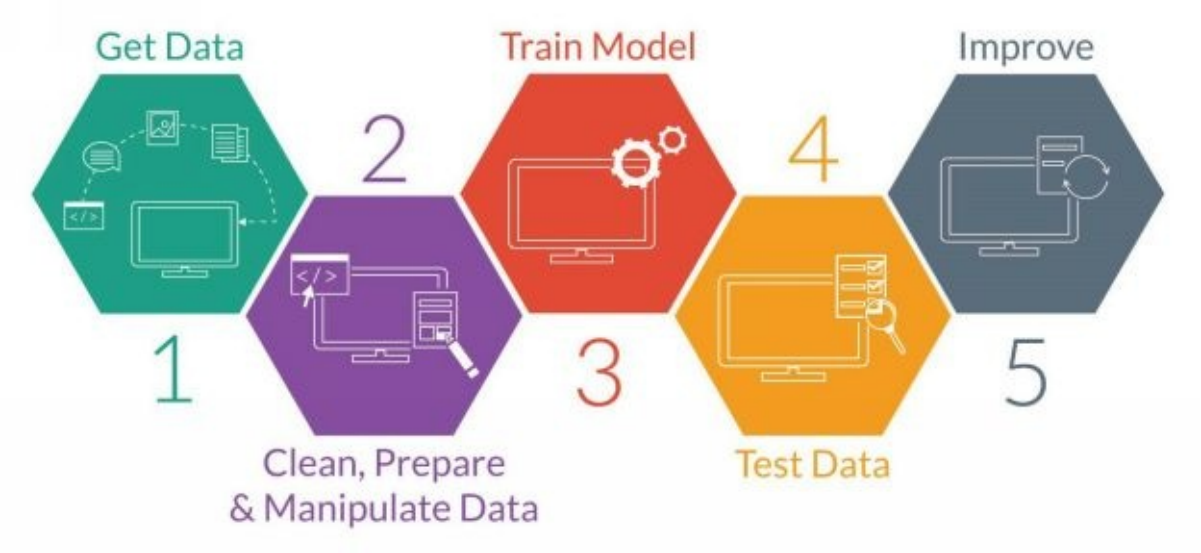
After we have chosen a model that has been fitted on the training dataset, we can utilize the test dataset to check how well it performs on this inconspicuous unseen data to assess the general mistake. If we are happy with its execution, we would now be able to utilize this model to predict new, future data. Note that the parameters for the previously mentioned procedures —likewise feature scaling and dimensionality reduction—are solely obtained from the training dataset, and the same parameters are later re-applied to transform the test dataset, as well as any new data samples—the performance measured on the test data may be overoptimistic otherwise.

Data Cleaning and Preparation

Objective of this Chapter

The more disciplined you are in your handling of the data, the more steady and better outcomes you are like likely to achieve the procedure for preparing data for a machine learning algorithms can be compressed in three stages:

- Step 1: Select Data
- Step 2: Preprocess Data
- Step 3: Transform Data


Get Data
1
Clean, Prepare
& Manipulate Data
2
Train Model
3
Test Data
4
Improve
5

Select Data

In this step, we choose the subset of every single accessible data that you will work with. There is always a strong desire for including all data that is accessible, that the proverb “more is better” will hold. This may or may not be true

You have to consider what data you really need to address the inquiry or issue you are dealing with. Make a few suppositions about the data you require and be mindful so as to record those suspicions with the goal that you can test them later if necessary.

Below are some questions to help you think through this process:

- What is the degree of accessibility of the data you have? For instance, through time, database tables, associated systems. Guarantee you have a reasonable picture of everything that you can utilize.
- What data isn’t accessible that you wish you had accessible? For instance, data that is not recorded or cannot be recorded. You might have the to derive or simulate this data
- What data don’t you have to address the issue? Excluding data is quite often simpler than including data. Note down which data you rejected and why.

It is just in little issues, similar to a competition or toy datasets where the data has just been chosen for you.

Preprocess Data

After you have chosen the data, you have to consider how you will utilize the data. This preprocessing step is tied in with getting the chose data into a shape that you can work.

Three normal data preprocessing steps are organizing, cleaning and examining:

Arranging:

The data you have chosen may not be in a format that is reasonable for you to

work with. The data might be in a relational database and you might want it in a flat file, or the data might be in proprietary file format and you might want it in a relational database or a text file.

Cleaning:

Cleaning data is the removal or fixing of missing data. There might be data cases that are inadequate and do not convey the data you trust, you have to address this issue. These examples may be expelled. Moreover, there might be sensitive data in some of the attributes and these attributes may be anonymized or expelled from the data.

Sampling:

There might be significantly more chosen data accessible than you have to work with. More data results in much longer running time for algorithms and bigger computational and memory necessities. You can also take a smaller representative sample of the chosen data which will be considerably speedier for investigating and prototyping arrangements before considering the entire dataset.

It is likely that the machine learning tools you use on the data will affect the preprocessing you will be required to perform. You will probably return to this step.

So much data

Transform Data

The last step is to transform the process data. The particular algorithm you are working with and the knowledge of the problem domain will influence this step and you will have to revisit different transformations of your preprocessed data as you work on your problem.

Three normal data transformations are attribute decompositions and attribute aggregations. All of them are discussed below:

The last step is to transform the process data. The particular algorithm you are working with and the knowledge of the problem domain will influence this step and you will have to revisit different transformations of your preprocessed data as you work on your problem.

Three normal data transformations are attribute decompositions and attribute aggregations. All of them are discussed below:

Scaling:

The preprocessed information may contain attributes with a mixture of scales for different amounts, for example, dollars, kilograms and sales volume. Numerous machine learning strategies like data attributes have a similar scale, for example, in between 0 and 1 for the littlest and biggest feature for a given component. Consider any feature for which you need to perform scaling

Decomposition:

There might be features that speak to a complex idea that might be more valuable to a machine learning strategy when split into the constituent parts. A case is a date that may have day and time segments that thusly could be parted out further. Maybe just the hour of the day is applicable to the issue being solved. Consider what feature decompositions you can perform.

Aggregation:

There might be features that can be aggregated into a single feature that would be more important to the issue you are attempting to solve. For instance, there might be a data examples for each time a client signed into a system that could be aggregated into a count for the number of logins allowing the additional instances to be discarded. Consider what kind of features aggregations could perform.

You can invest a ton of energy designing features from your data and it can be

exceptionally advantageous to the execution of an algorithm. Begin little and expand the abilities you learn.

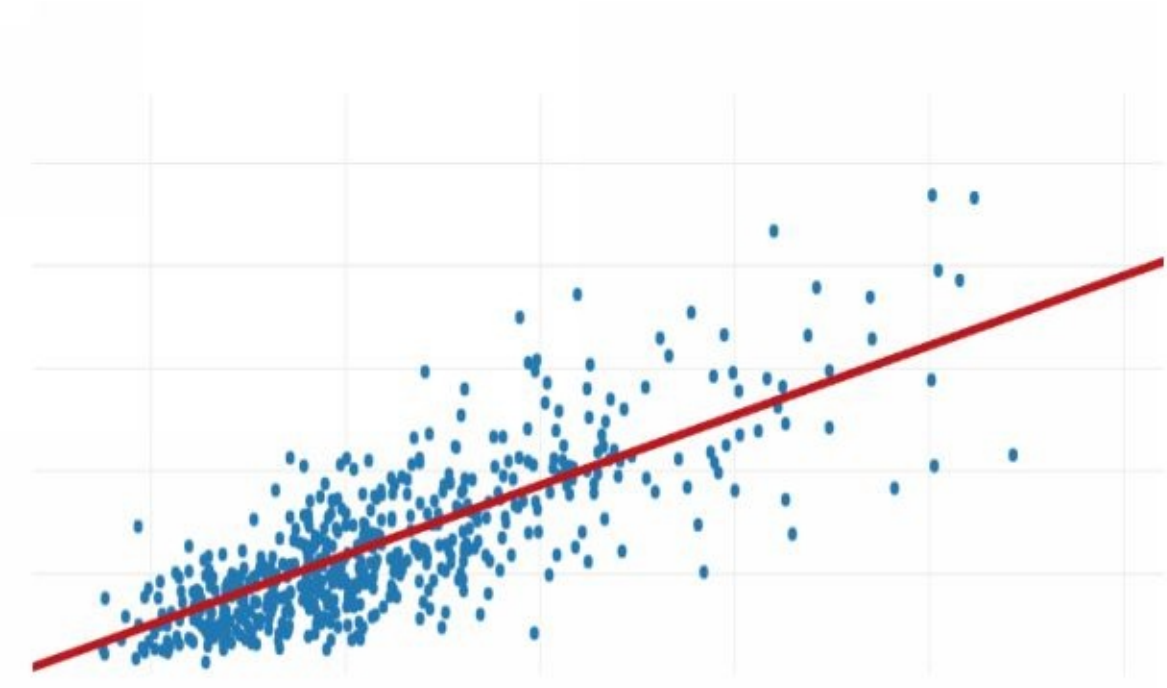
Application of Supervised Learning Techniques with Python

Objective of this Chapter

At the end of this chapter, the reader should have learned:

- Classification application with Python
- Regression application with Python
- Decision Tree application with Python

The fate of planet Earth is Artificial Intelligence/Machine Learning. Any individual who does not understand it will soon get themselves abandoned. Awakening in this world full of innovation feels more like a magic. There are numerous sorts of executions and methods to do Artificial Intelligence and Machine Learning to take care of constant issues, out of which Supervised Learning is a standout amongst the most utilized methodologies.



In supervised learning, we begin with bringing in a dataset containing training attributes and the target attributes. The Supervised Learning calculation will take in the connection between training examples and their related target variables and apply that learned relationship to characterize altogether new sources of input

Classification

Consider the case of a medical scientist who needs to examine breast cancer data to foresee which one of every three particular medications a patient ought to get. This data examination task is called Classification, where a model or classifier is developed to predict class labels, for example,

“treatment A,” “treatment B” or “treatment C.”

Basically, classification is a prediction problem that predicts the straight-out class names which are discrete and unordered. It is a two-step process, comprising of a learning step and a classification step.

The most commonly used algorithms to solve classification are listed below.

- K - Nearest Neighbor
- Decision Trees
- Naïve Bayes

Support Vector Machines

The first step the is learning step in which our classifier models build the classifier by completely analyzing the training set. Now the next step is the classification step in which class labels for our given data are predicted. The tuples of our dataset and their related class labels under analysis are split into two set training set and test set.

The test set is utilized to evaluate the predictive exactness of a classifier. The precision of a classifier is the level of test tuples that are effectively ordered by the classifier. To accomplish higher exactness, the ideal path is to test out various calculations and attempting diverse parameters inside every calculation too. As well as can be expected to be chosen by cross-validation.

Initial steps, to apply our machine learning algorithms we have to understand and investigate the given dataset. In this illustration, we utilize IRIS dataset which is imported from the scikit-learn package. Now we should jump into the code and investigate the IRIS dataset.

First of all, make sure python is installed on your computers and also install these packages on your computers.

```
pip install pandas pip install matplotlib pip install scikit-learn
```

```
from sklearn import datasets
import pandas as pd
import matplotlib.pyplot as plt


# Loading IRIS dataset from scikit-learn object into iris variable.
iris = datasets.load_iris()


# Prints the type/type object of iris
print(type(iris))
# <class ‘sklearn.datasets.base.Bunch’>


# prints the dictionary keys of iris data
print(iris.keys())


# prints the type/type object of given attributes
print(type(iris.data), type(iris.target))


# prints the no of rows and columns in the dataset
print(iris.data.shape)


# prints the target set of the data
print(iris.target_names)


# Load iris training dataset
X = iris.data


# Load iris target set
Y = iris.target
```

```
# Convert datasets' type into dataframe
df = pd.DataFrame(X, columns=iris.feature_names)


# Print the first five tuples of dataframe.
print(df.head())
```

Output:

```
<class 'sklearn.datasets.base.Bunch'>
dict_keys(['data', 'target', 'target_names', 'DESCR', 'feature_names'])]
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
(150, 4)
['setosa' 'versicolor' 'virginica']
sepal length (cm) sepal width (cm) petal length (cm) petal width  (cm)
0   5.1   3.5   1.4  0.2
1   4.9   3.0   1.4  0.2
2   4.7   3.2   1.3  0.2
3   4.6   3.1   1.5  0.2
4   5.0   3.6   1.4  0.2
```

```
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier


# Load iris dataset from sklearn
iris = datasets.load_iris()


# Declare an of the KNN classifier class with the value with neighbors.
knn = KNeighborsClassifier(n_neighbors=6)


# Fit the model with training data and target values
knn.fit(iris['data'], iris['target'])
```

```
# Provide data whose class labels are to be predicted
X = [
    [5.9, 1.0, 5.1, 1.8],
    [3.4, 2.0, 1.1, 4.8],
]


# Prints the data provided
print(X)


# Store predicted class labels of X
prediction = knn.predict(X)


# Prints the predicted class labels of X
print(prediction)
```

Output:

[1 1]

Regression

Regression is typically named as deciding connections between at least two variables. For instance, think of you as the need to predict the salary of a man, in light of the given data X.

Here, target variable means the unknown variable we think about predicting, and continuous means there aren’t gaps (discontinuities) in the value that Y can go up against.

Predicting salary is a classic regression issue. Your information ought to have all the data (known as features) about the person that can foresee pay, for

example, his working hours, training background, work title, a place he lives.

There are most commonly used regression models that are:

- Linear Regression
- Logistic Regression
- Polynomial Regression

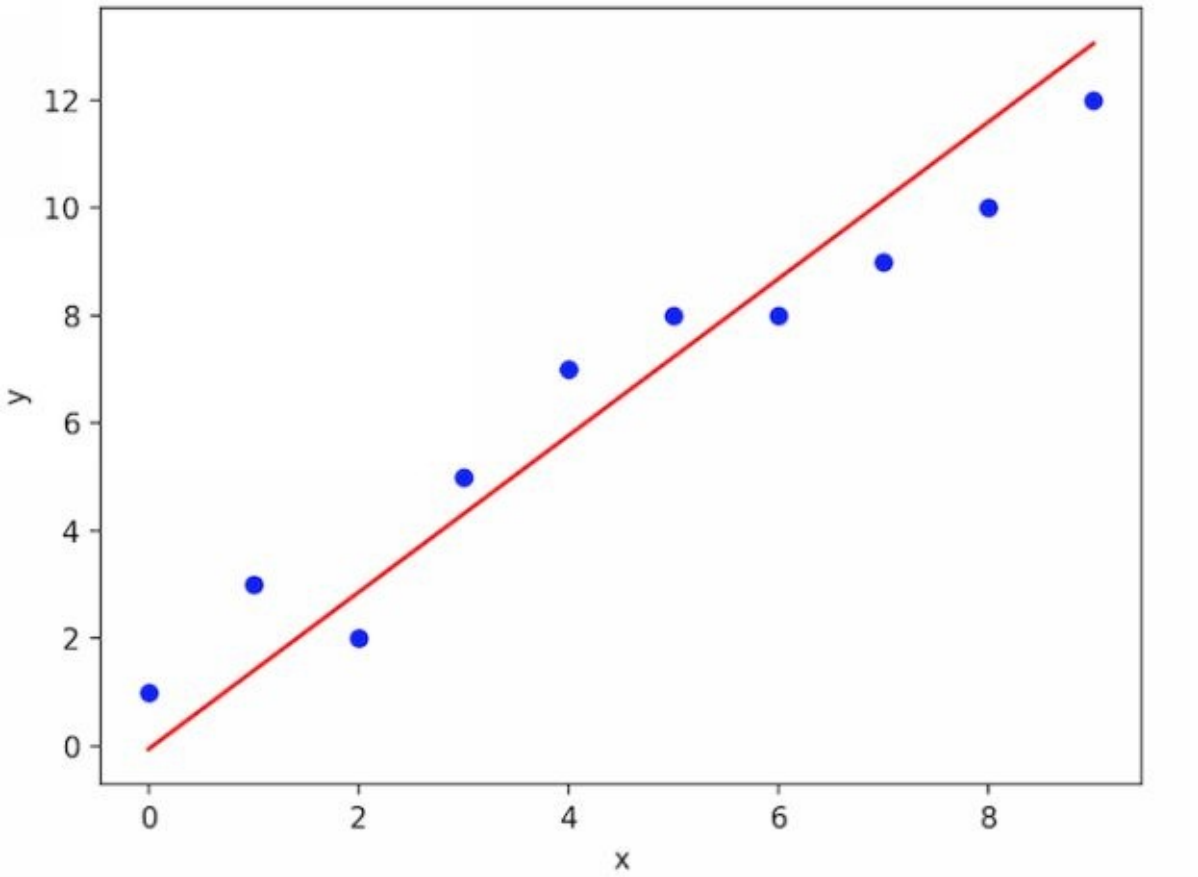
Linear Regression

In this regression model it develops a relationship between a dependent variable (y) and one or more independent variable (x) using the best fit straight line called regression line.

Mathematically it is represented as:

$$\mathbf{h(xi) = \beta o + \beta 1 * xi + e}$$

where βo is the intercept, β1 is the slope of the line and e is the error term.



Logistic Regression

This regression model is an algorithm and is used where the response variable is categorical. The basic concept of this type of regression is to find a relationship between features and probability of particular outcomes.

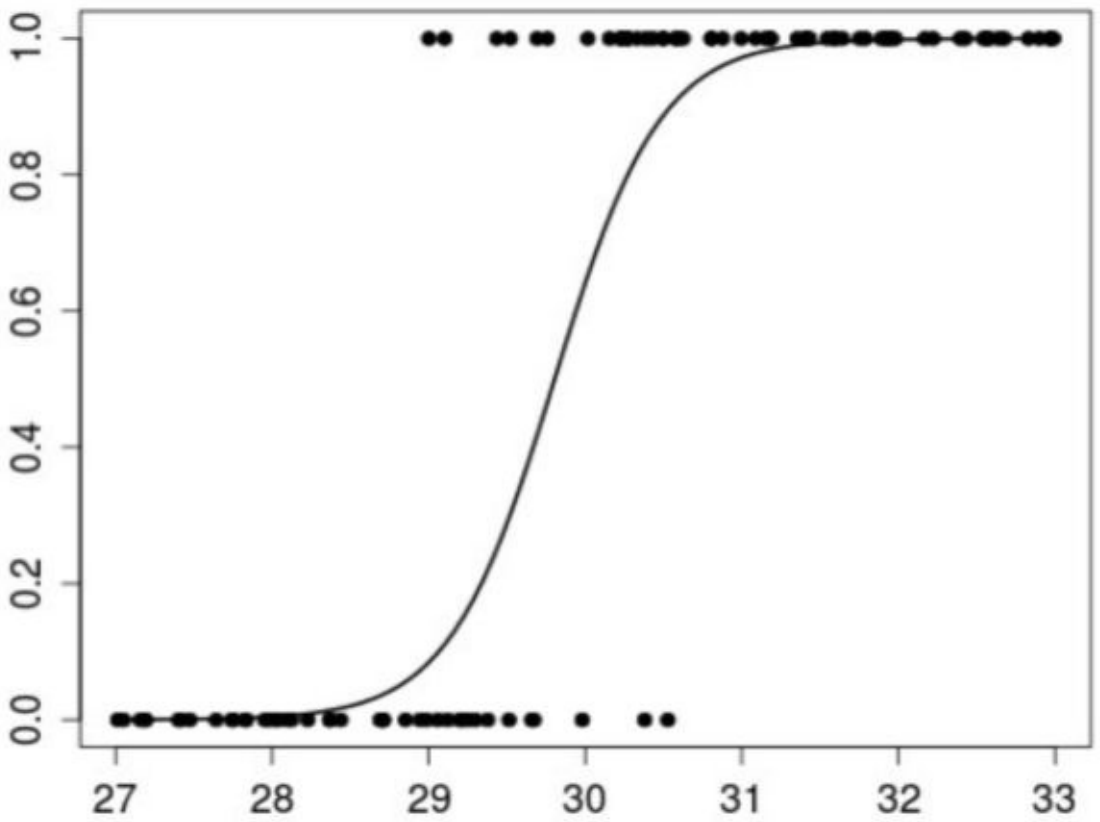
Mathematically it is represented as:

$$p(X) = \beta o + \beta 1 * X$$

Where

$$p(x) = p(y = 1 \mid x)$$

While graphically it is represented as:



Polynomial Regression

The model of regression in which relationship between independent variable x and dependent variable y is modeled as nth degree polynomial in x

Implementation of Linear Regression in scikit-learn

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


# Load the diabetes dataset
diabetes = datasets.load_diabetes()
```

```
# Use only one feature for training
diabetes_X = diabetes.data[:, np.newaxis, 2]


# Split the data into training/testing sets
diabetes_X_train = diabetes_X[:-20]
diabetes_X_test = diabetes_X[-20:]


# Split the targets into training/testing sets
diabetes_y_train = diabetes.target[:-20]
diabetes_y_test = diabetes.target[-20:]


# Create linear regression object
regr = linear_model.LinearRegression()


# Train the model using the training sets
regr.fit(diabetes_X_train, diabetes_y_train)


# Input data
print('Input Values')
print(diabetes_X_test)


# Make predictions using the testing set
diabetes_y_pred = regr.predict(diabetes_X_test)


# Predicted Data
print("Predicted Output Values")
print(diabetes_y_pred)
```

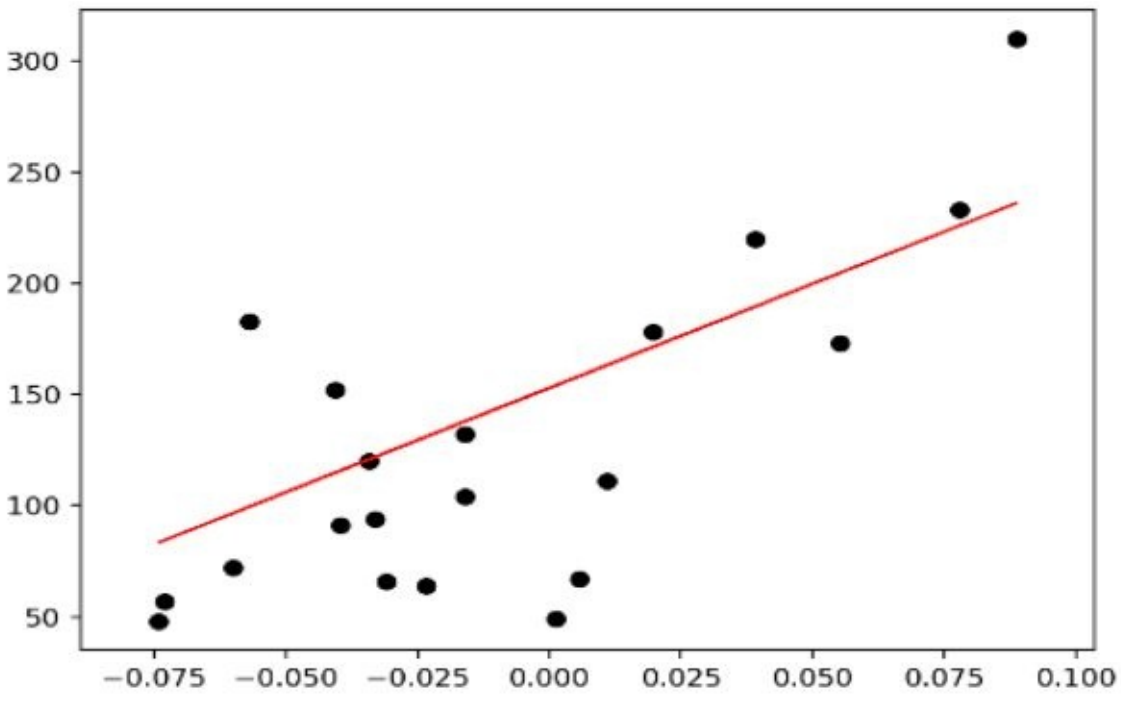
```
# Plot outputs
plt.scatter(diabetes_X_test, diabetes_y_test, color='black')
plt.plot(diabetes_X_test, diabetes_y_pred, color='red', linewidth=1)


plt.show()
```

Output:

```
Input Values[[ 0.07786339]  [-0.03961813]  [ 0.01103904]  [-0.04069594]    [-0.03422907]  [
0.00564998]  [ 0.08864151]  [-0.03315126] [-0.05686312]  [-0.03099563]  [ 0.05522933]
[-0.06009656] [ 0.00133873]  [-0.02345095]  [-0.07410811]  [ 0.01966154][-0.01590626]
[-0.01590626]  [ 0.03906215]  [-0.0730303 ]]
```

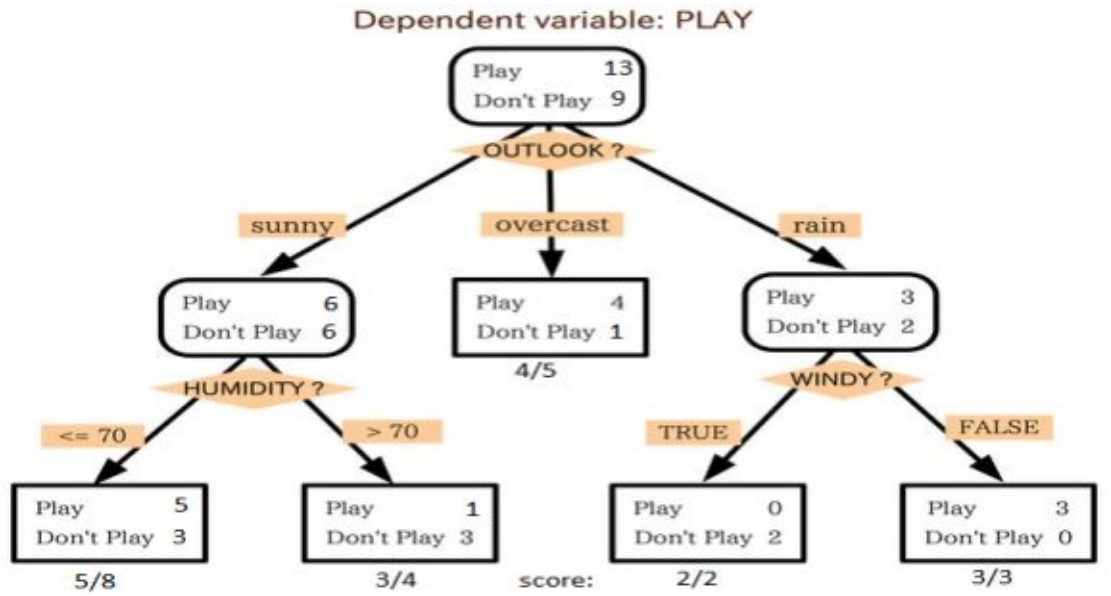
Predicted Output Values

```
[225.9732401   115.74763374  163.27610621  114.73638965   120.80385422  158.21988574
236.08568105  121.81509832  99.56772822   123.83758651  204.73711411   96.53399594
154.17490936  130.91629517   83.3878227   171.36605897 137.99500384  137.99500384
189.56845268   84.3990668 ]
```



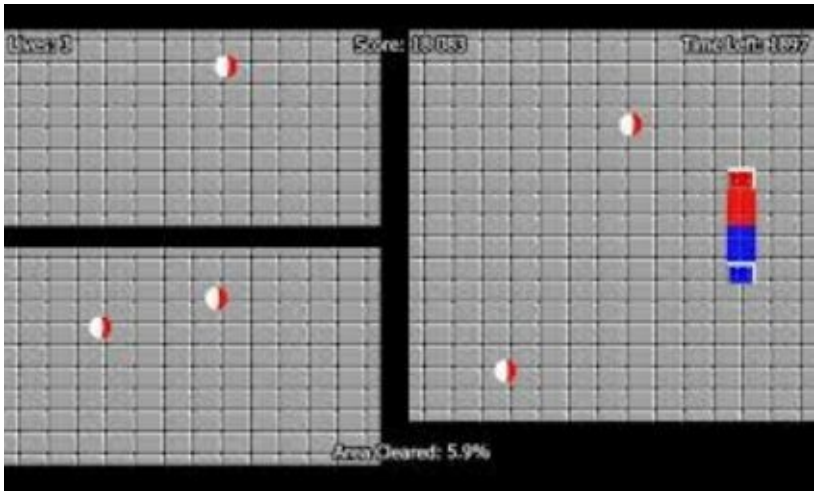
Decision tree

This is one of the easiest algorithms and is utilized frequently. It is a sort of supervised learning algorithm that is frequently used for classification troubles. Out of the blue, it works for both categorical and continuous dependent variables. In this algorithm, we split the populace into two or additional homogeneous sets. This is finished in the view of most considerable attributes/ independent variables to make as particular gatherings as could reasonably be expected.



In the photo above, you can see that populace is ordered into four disparate gatherings in light of various credits to perceive ‘in the event that they will play or not’. To part the populace into unique heterogeneous gatherings, it utilizes an assortment of procedures like Gini, Information Gain, Chi-square, entropy.

The most brilliant approach to know how choice tree functions is to play Jezzball – an exemplary amusement from Microsoft (picture underneath 1.10). In a general sense, you have a life with moving dividers and you have to make dividers to such an extent that greatest territory gets tidied up without the balls.



Along these lines, every minute in time you separate the life with a divider, you are attempting to make 2 divergent populaces inside a similar room. Decision trees work in a like design by partitioning a populace in as various gatherings as would be possible.

```
#Import Library

#Import other necessary libraries like pandas, numpy…

from sklearn import tree


#Assumed you have, X (predictor) and Y (target) for training data set and x_test(predictor) of test_dataset


# Create tree object

model = tree.DecisionTreeClassifier(criterion='gini') # for classification, here you can change the algorithm as gini or entropy (information gain) by default it is gini


# model = tree.DecisionTreeRegressor() for regression

# Train the model using the training sets and check score


model.fit(X, y)

model.score(X, y)

#Predict Output
```

Applications of unsupervised learning Techniques with python

Objective of this Chapter

At the end of this chapter, the reader should have learned:

- Hierarchical & Partitioned Clustering application with Python
- K-means Clustering application with Python
- Neural Networks application with Python

Unsupervised learning is the place you just have input data (X) and no related output factors. The objective of unsupervised learning is to display the hidden structure or dispersion in the data keeping in mind the end goal to take in more about the data.

These are called unsupervised learning because unlike supervised learning above, there are no correct answers and there is no teacher. Algorithms are left to their own particular devices to find and present the fascinating structure in the data.

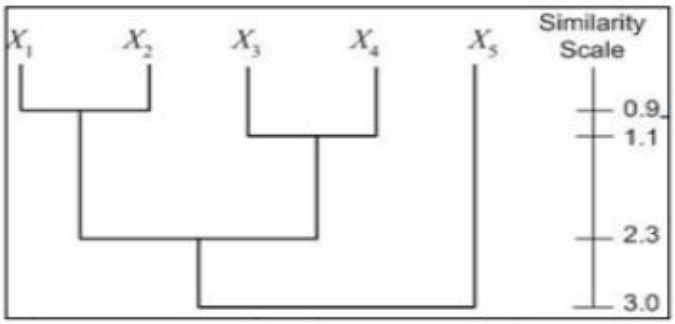
Clustering

Clustering based learning is an unsupervised learning procedure and in this manner, works without a concrete definition of the target attribute. You will learn the basics and the high-level concepts of this procedure, what's more, get hands-on implementation guidance in utilizing Apache Mahout, R, Julia, Apache Spark, and Python to execute the k-means clustering algorithm.

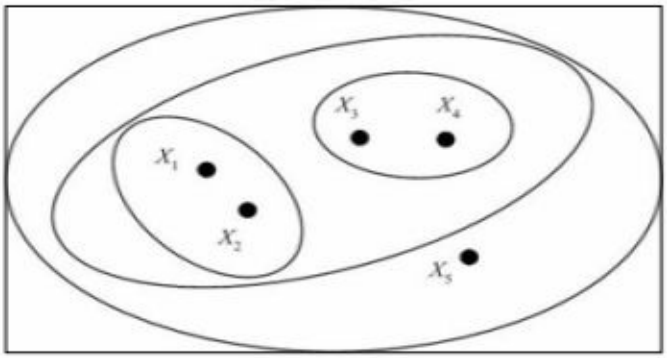
There are two types of Clustering that are given below:

Hierarchical clustering

The Hierarchical clustering is tied in with characterizing clusters that have a hierarchy, and this is done either by iteratively merging smaller clusters into a bigger cluster, or partitioning a bigger group into little clusters. This hierarchy of clusters that are created by a clustering algorithm is known as a dendogram. A dendogram is one of the routes in which the various hierarchical clusters can be represented, and the user can realize different clustering based on the level at which the dendogram is defined. It utilizes a similarity scale that represents the distance between the clusters that were gathered from the bigger bunch. The accompanying graph portrays a dendogram portrayal for the Hierarchical clusters:



There is another simple way of representing the Hierarchical clusters; that is, the Venn diagram. In this representation, we circle the data points that are a part of the cluster. The following diagram depicts a Venn representation for five data points:



Partitioned clustering

Partitioned clustering algorithms are diverse in contrast with the Hierarchical clustering algorithms as the clusters or partitions are created and assessed utilizing a particular predefined criterion that is area particular. Since each group shaped is totally unrelated, there can never be a various leveled connection between the clusters. Actually, every example can be put in one and just a single of the k clusters. The quantity of clusters (k) to be formed is the contribution to this algorithm, and this one set of k clusters is the output of the partitioned cluster algorithms A standout amongst the most generally utilized partitioned cluster algorithms that we will cover in this part is the k-mean clustering algorithms.

K-means clustering

The basic execution of K-mean algorithms is given below. Now we move towards the implementation of this algorithm

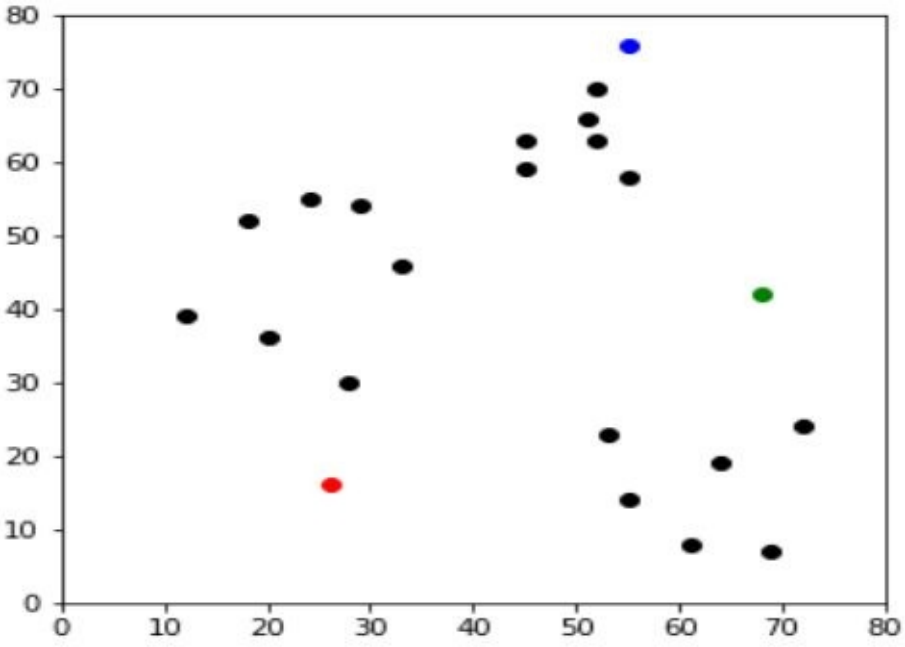
- Identify the k data points as the initial centroids (cluster centers).

- Repeat step 1.
- For each data point x ϵ D dox
- Compute the distance from x to the centroid.
- Assign x to the closest centroid (a centroid represents a cluster).
- endfor
- Re-compute the centroids using the current cluster memberships until the stopping criterion is met

Implementation of k-means Clustering in Python

```
# Initialisation

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

df = pd.DataFrame({
    'x': [12, 20, 28, 18, 29, 33, 24, 45, 45, 52, 51, 52, 55, 53, 55, 61, 64, 69, 72],
    'y': [39, 36, 30, 52, 54, 46, 55, 59, 63, 70, 66, 63, 58, 23, 14, 8, 19, 7, 24]
})


np.random.seed(200)
k = 3
# centroids[i] = [x, y]
centroids = {
    i+1: [np.random.randint(0, 80), np.random.randint(0, 80)]
    for i in range(k)
}

fig = plt.figure(figsize=(5, 5))
plt.scatter(df['x'], df['y'], color='k')
colmap = {1: 'r', 2: 'g', 3: 'b'}
```

```
for i in centroids.keys():
    plt.scatter(*centroids[i], color=colmap[i])
plt.xlim(0, 80)
plt.ylim(0, 80)
plt.show()
```

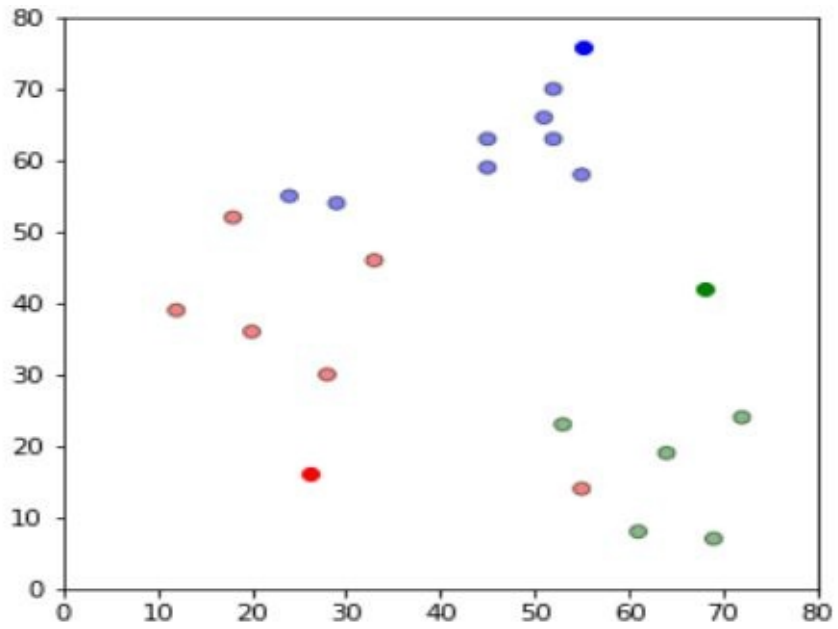


Assignment Stage

```
def assignment(df, centroids):
    for i in centroids.keys():
    # sqrt((x1 - x2)^2 - (y1 - y2)^2)
    df['distance_from_{}'.format(i)] = (
    np.sqrt(
    (df['x'] - centroids[i][0]) ** 2
    + (df['y'] - centroids[i][1]) ** 2
    )
    )
    centroid_distance_cols = ['distance_from_{}'.format(i) for i in centroids.keys()]
    df['closest'] = df.loc[:, centroid_distance_cols].idxmin(axis=1)
    df['closest'] = df['closest'].map(lambda x: int(x.lstrip('distance_from_')))
    df['color'] = df['closest'].map(lambda x: colmap[x])
    return df
```

```
df = assignment(df, centroids)
print(df.head())

fig = plt.figure(figsize=(5, 5))
plt.scatter(df['x'], df['y'], color=df['color'], alpha=0.5, edgecolor='k')
for i in centroids.keys():
    plt.scatter(*centroids[i], color=colmap[i])
plt.xlim(0, 80)
plt.ylim(0, 80)
plt.show()
```

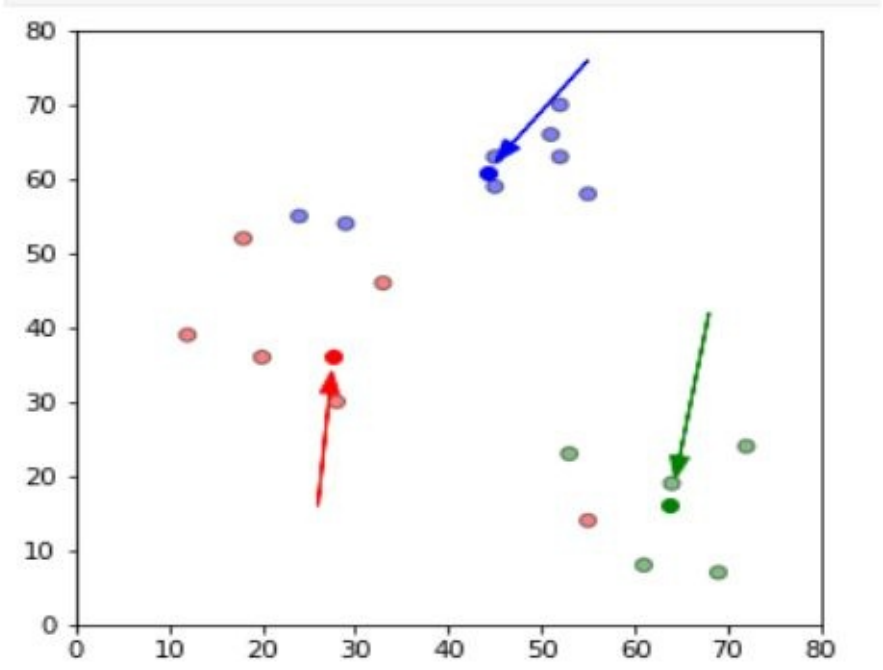
	x	y	distance_from_1	distance_from_2	distance_from_3	closest	color
0	12	39	26.925824	56.080300	56.727418	1	r
1	20	36	20.880613	48.373546	53.150729	1	r
2	28	30	14.142136	41.761226	53.338541	1	r
3	18	52	36.878178	50.990195	44.102154	1	r
4	29	54	38.118237	40.804412	34.058773	3	b



Update Stage

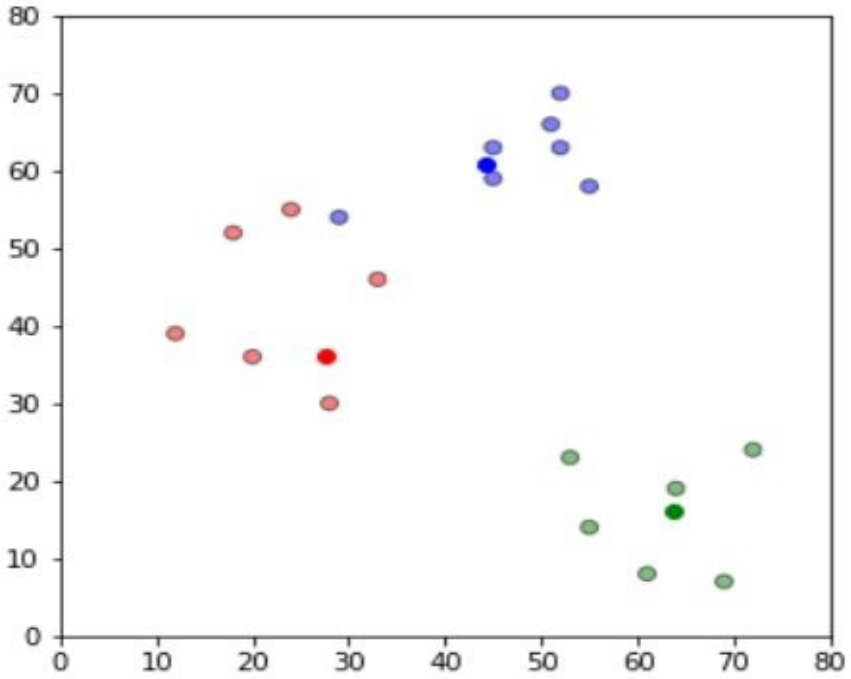
```
import copy
```

```
old_centroids = copy.deepcopy(centroids)


def update(k):
    for i in centroids.keys():
    centroids[i][0] = np.mean(df[df[‘closest’] == i][‘x’])
    centroids[i][1] = np.mean(df[df[‘closest’] == i][‘y’])
    return k


centroids = update(centroids)

fig = plt.figure(figsize=(5, 5))
ax = plt.axes()
plt.scatter(df[‘x’], df[‘y’], color=df[‘color’], alpha=0.5, edgecolor=‘k’)
for i in centroids.keys():
    plt.scatter(*centroids[i], color=colmap[i])
plt.xlim(0, 80)
plt.ylim(0, 80)
for i in old_centroids.keys():
    old_x = old_centroids[i][0]
    old_y = old_centroids[i][1]
    dx = (centroids[i][0] - old_centroids[i][0]) * 0.75
    dy = (centroids[i][1] - old_centroids[i][1]) * 0.75
    ax.arrow(old_x, old_y, dx, dy, head_width=2, head_length=3, fc=colmap[i], ec=colmap[i])
plt.show()
```

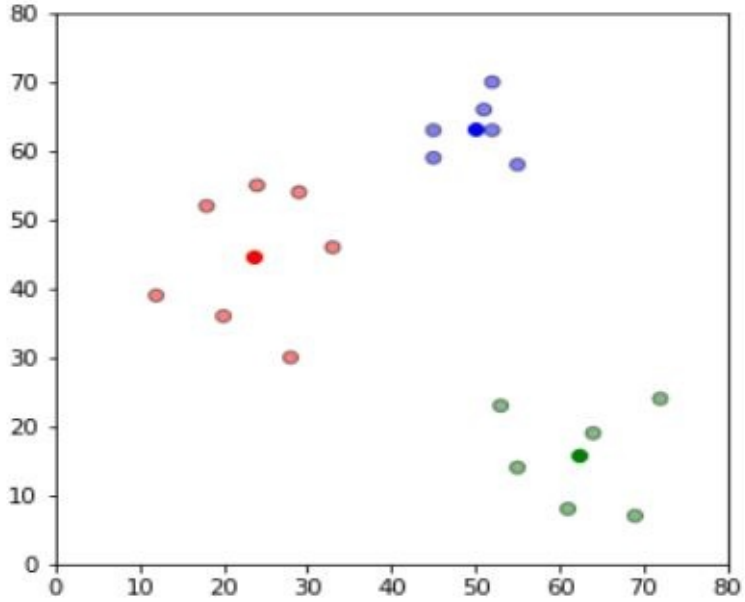


Repeat Assignment Stage

```
df = assignment(df, centroids)

# Plot results
fig = plt.figure(figsize=(5, 5))
plt.scatter(df[‘x’], df[‘y’], color=df[‘color’], alpha=0.5, edgecolor=‘k’)
for i in centroids.keys():
    plt.scatter(*centroids[i], color=colmap[i])
plt.xlim(0, 80)
plt.ylim(0, 80)
plt.show()
```

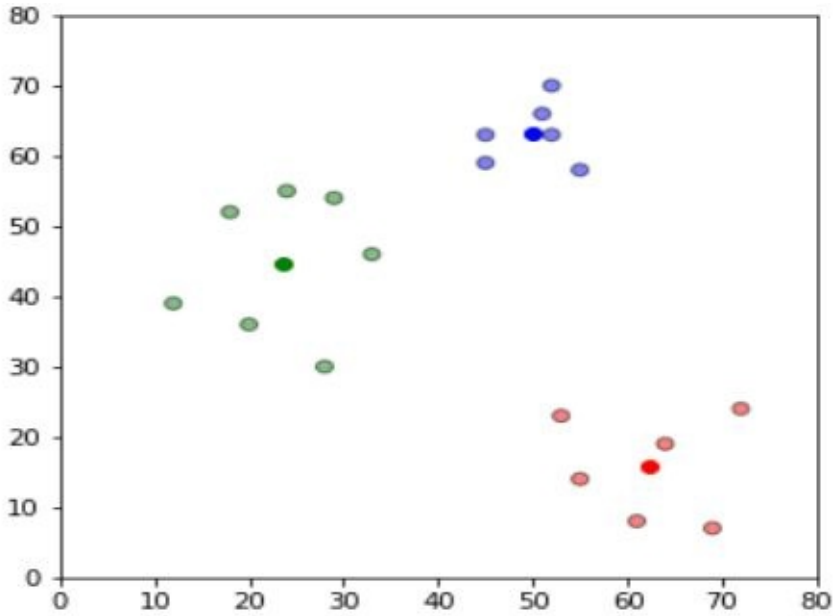



```
# Continue until all assigned categories don’t change any more
while True:
    closest_centroids = df[‘closest’].copy(deep=True)
    centroids = update(centroids)
    df = assignment(df, centroids)
    if closest_centroids.equals(df[‘closest’]):
    break

fig = plt.figure(figsize=(5, 5))
plt.scatter(df[‘x’], df[‘y’], color=df[‘color’], alpha=0.5, edgecolor=‘k’)
for i in centroids.keys():
    plt.scatter(*centroids[i], color=colmap[i])
plt.xlim(0, 80)
plt.ylim(0, 80)
plt.show()
```



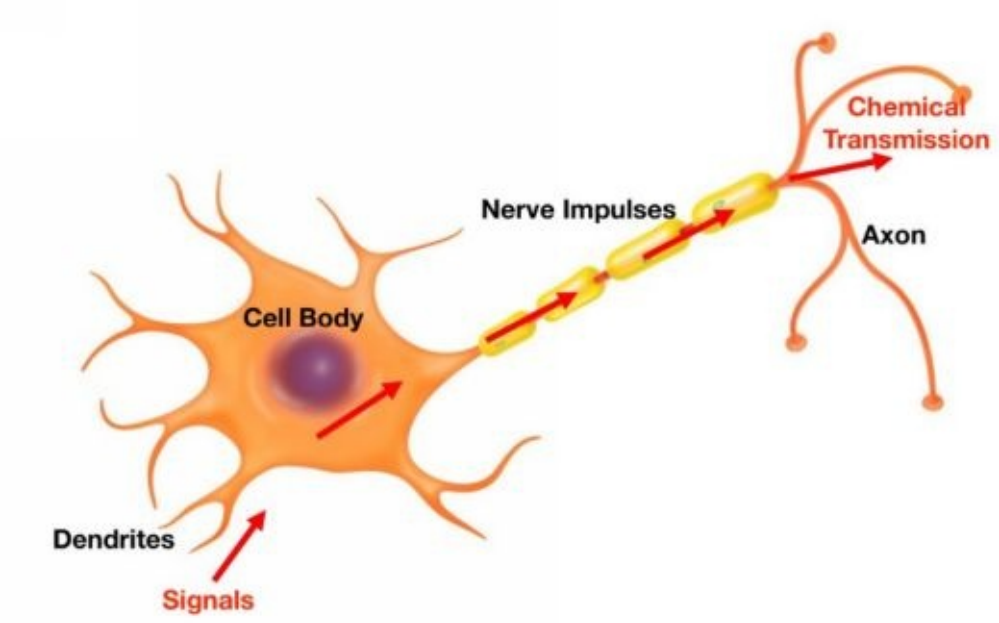

```
df = pd.DataFrame({
    'x': [12, 20, 28, 18, 29, 33, 24, 45, 45, 52, 51, 52, 55, 53, 55, 61, 64, 69, 72],
    'y': [39, 36, 30, 52, 54, 46, 55, 59, 63, 70, 66, 63, 58, 23, 14, 8, 19, 7, 24]
})

from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=3)
kmeans.fit(df)
Out[6]:
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
    n_clusters=3, n_init=10, n_jobs=1, precompute_distances='auto',
    random_state=None, tol=0.0001, verbose=0)
Then we learn the labels
In [7]:
labels = kmeans.predict(df)
centroids = kmeans.cluster_centers_
In [8]:
fig = plt.figure(figsize=(5, 5))

colors = map(lambda x: colmap[x+1], labels)
```

```
plt.scatter(df[‘x’], df[‘y’], color=colors, alpha=0.5, edgecolor=‘k’)
for idx, centroid in enumerate(centroids):
    plt.scatter(*centroid, color=colmap[idx+1])
plt.xlim(0, 80)
plt.ylim(0, 80)
plt.show()
```



Neural Networks

Study of the artificial neural network is generally referred as a neural network. This study has been motivated by the human brain that is a complete and successful biological system working in a different way from the traditional digital computer. The brain is highly complex, non-linear and parallel processor. It consists of numerous nerve cells with significant quality to work massively in parallel and has the ability to learn. There are nearly 100 billion neurons in the human brain. A typical neuron consists of a cell body, dendrites, and axon.



The building blocks of neural networks are neurons. In the technical system, we also refer to them as units or nodes.

Each neuron:

- receives input from many other neurons,
- changes its internal state (activation) based on the current input,
- Sends one output signal to many other neurons, possibly including its input neurons (recurrent network).

Data is transmitted as a progression of electric motivations, supposed spikes. The recurrence and period of these spikes encode the data. In natural frameworks, one neuron can be associated with upwards of 10,000 different neurons. More often than not, a neuron gets its data from different neurons in a kept territory, its alleged open field.

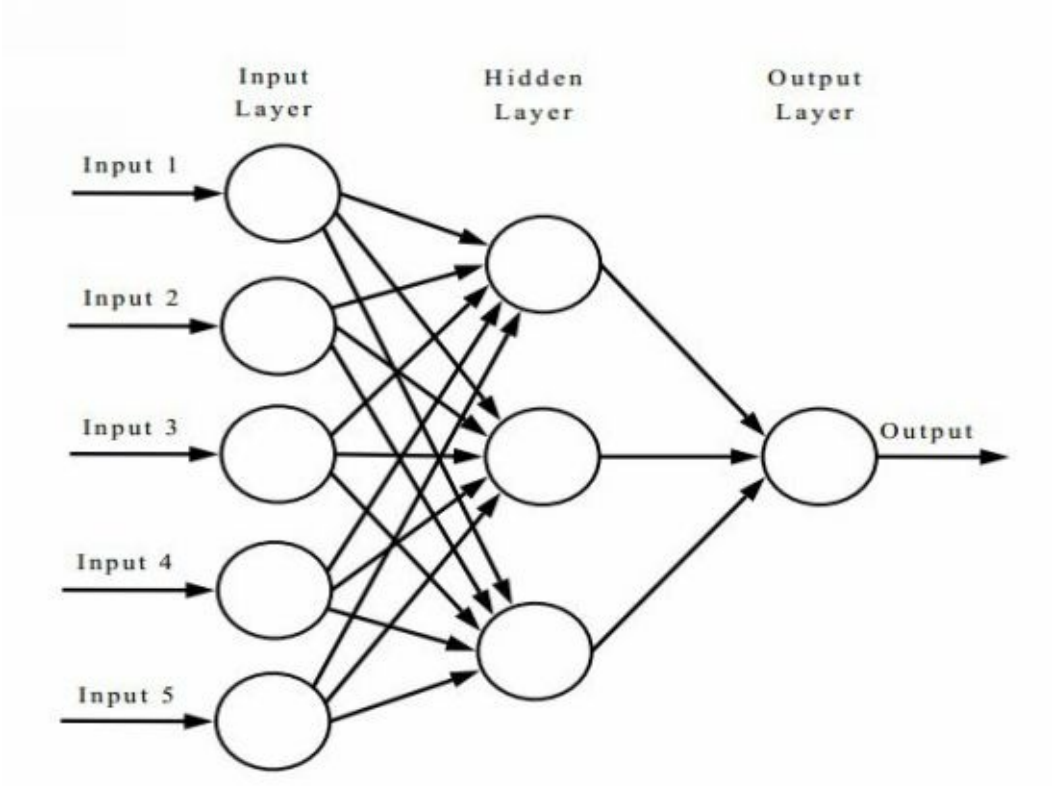
NNs (Neural Networks) are able to learn by adapting their connectivity patterns so that the organism improves its behavior in terms of reaching certain (evolutionary) goals. The power of a connection, or whether it is excitatory or inhibitory, depends on the state of a receiving neuron’s synapses the neural network achieves learning by appropriately adapting the states of

its synapses.

Artificial Neural Network Layers

The Artificial Neural network is typically organized in layers. Layers are being made up of many interconnected ‘nodes’ which contain an ‘activation function’. A neural network may contain the following 3 layers:

Patterns are presented to the network via the ‘input layer’, which communicates with one or more ‘hidden layers’ where the actual processing is done via a system of weighted ‘connections’. The hidden layers then link to an ‘output layer’ where the answer is output as shown in the figure below.



Input layer:

The purpose of the input layer is to receive as input the values of the explanatory attributes for each observation. Usually, the number of input

nodes in an input layer is equal to the number of explanatory variables. ‘Input layer’ presents the patterns to the network, which communicates with one or more ‘hidden layers’.

The nodes of the input layer are passive, meaning they do not change the data. They receive a single value on their input and duplicate the value to their many outputs. From the input layer, it duplicates each value and sent to all the hidden nodes.

Hidden layer

The Hidden layers apply given transformations to the input values inside the network. In this, incoming arcs that go from other hidden nodes or from input nodes connected to each node. It connects with outgoing arcs to output nodes or to other hidden nodes. In hidden layer, the actual processing is done via a system of weighted ‘connections’. There may be one or more hidden layers. The values entering a hidden node are multiplied by their corresponding weights, a set of predetermined numbers stored in the program. The weighted inputs are then added to produce a single number.

Output layer

The hidden layers then link to an ‘output layer‘. Output layer receives connections from hidden layers or from input layer. It returns an output value that corresponds to the prediction of the response variable. In classification problems, there is usually only one output node. The active nodes of the output layer combine and change the data to produce the output values.

The ability of the neural network to provide useful data manipulation lies in the proper selection of the weights. This is different from conventional information processing.

Structure of a Neural Network

A neural network has at least two physical components, namely, the processing elements and the connections between them. The processing elements are called neurons, and the connections between the neurons are known as links.

The structure of a neural network is also referred to as its ‘architecture’ or ‘topology’. It consists of the number of layers, Elementary units. It also consists of an interconnected weight adjustment mechanism. The choice of the structure determines the results which are going to obtain. It is the most critical part of the implementation of a neural network.

The simplest structure is the one in which units are distributed in two layers: An input layer and an output layer. Each unit in the input layer has a single input and a single output which is equal to the input. The output unit has all the units of the input layer connected to its input, with a combination function and a transfer function. There may be more than 1 output unit. In this case, resulting model is a linear or logistic regression. This is depending on whether transfer function is linear or logistics. The weights of the network are regression coefficients.

By adding 1 or more hidden layers between the input and output layers and units in this layer the predictive power of neural network increases. But a number of hidden layers should be as small as possible. This ensures that neural network does not store all information from learning set, but can generalize it to avoid over fitting.

Over fitting can occur. It occurs when weights make the system learn details of learning set instead of discovering structures. This happens when the size of learning set is too small in relation to the complexity of the model.

A hidden layer is present or not, the output layer of the network can sometimes have many units when there are many classes to predict.

Applications of Neural Network

Neural Network Applications can be grouped into following categories:

- **Clustering:**

A clustering algorithm explores the similarity between patterns and places similar patterns in a cluster. Best known applications include data compression and data mining.

- **Classification/Pattern recognition:**

The undertaking of example acknowledgment is to dole out an info design (like a manually written image) to one of many classes. This classification incorporates algorithmic usage, for example, acquainted memory.

- **Function approximation:**

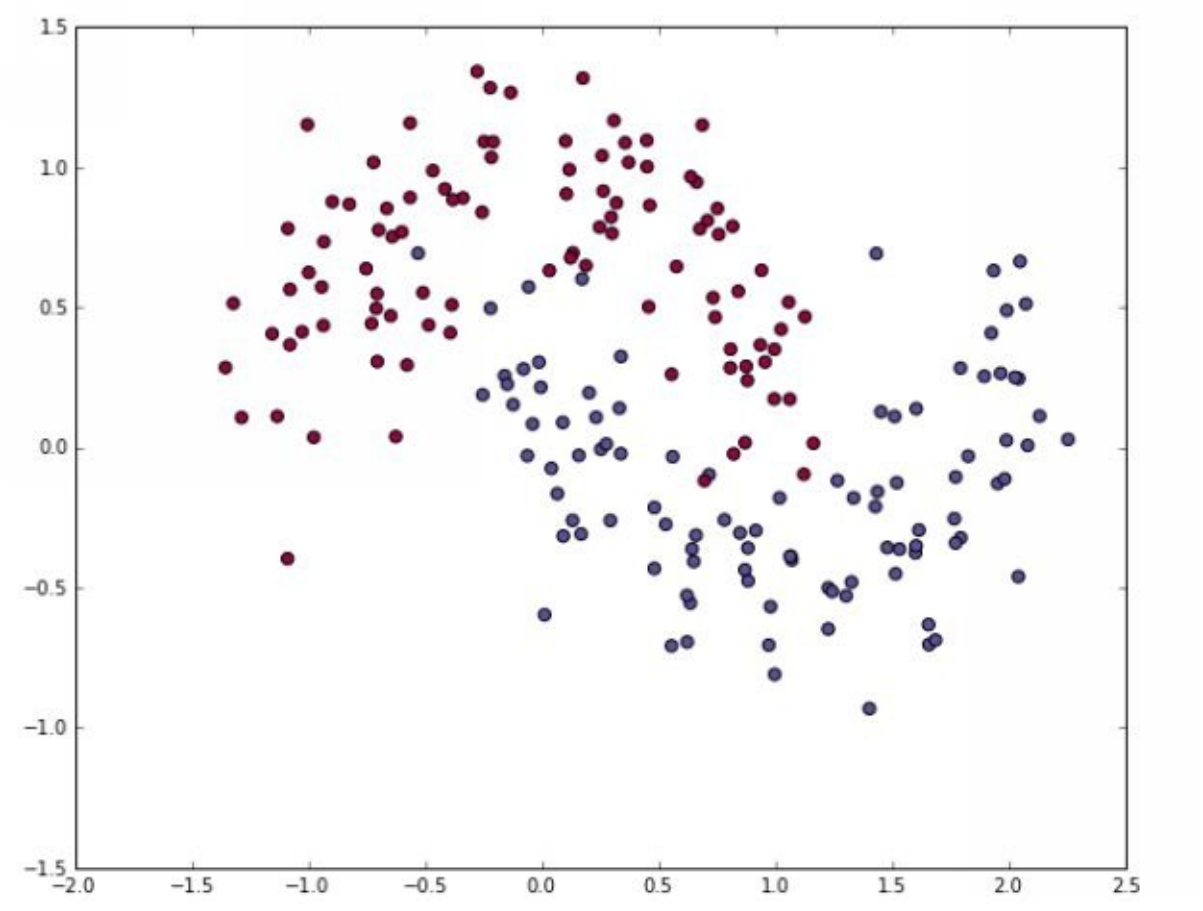
The tasks of function approximation are to find an estimate of the unknown function f() subject to noise. Various engineering and scientific disciplines require function approximation.

- **Prediction/Dynamical Systems:**

The task is to forecast some future values of a time-sequenced data. Prediction significantly affects the choice of emotionally supportive networks. Forecast contrasts from the function estimate by considering the time factor. Here, the framework is dynamic and may deliver distinctive outcomes for similar info information in light of framework state (time).

Implementation in Python

```
# Generate a dataset and plot it
np.random.seed(0)
X, y = sklearn.datasets.make_moons(200, noise=0.20)
plt.scatter(X[:,0], X[:,1], s=40, c=y, cmap=plt.cm.Spectral)
```




```
num_examples = len(X) # training set size
nn_input_dim = 2 # input layer dimensionality
nn_output_dim = 2 # output layer dimensionality

# Gradient descent parameters (I picked these by hand)
epsilon = 0.01 # learning rate for gradient descent
reg_lambda = 0.01 # regularization strength
```

First let’s implement the loss function we defined above. We use this to evaluate how well our model is doing:

```
# Helper function to evaluate the total loss on the dataset
def calculate_loss(model):
    W1, b1, W2, b2 = model[‘W1’], model[‘b1’], model[‘W2’], model[‘b2’]
    # Forward propagation to calculate our predictions
    z1 = X.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2
    exp_scores = np.exp(z2)
```

```
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    # Calculating the loss
    corect_logprobs = -np.log(probs[range(num_examples), y])
    data_loss = np.sum(corect_logprobs)
    # Add regulatization term to loss (optional)
    data_loss += reg_lambda/2 * (np.sum(np.square(W1)) + np.sum(np.square(W2)))
    return 1./num_examples * data_loss
```

We also implement a helper function to calculate the output of the network. It does forward propagation as defined above and returns the class with the highest probability.

```
# Helper function to predict an output (0 or 1)
def predict (model, x):
    W1, b1, W2, b2 = model['W1'], model['b1'], model['W2'], model['b2']
    # Forward propagation
    z1 = x.dot(W1) + b1
    a1 = np.tanh(z1)
    z2 = a1.dot(W2) + b2
    exp_scores = np.exp(z2)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    return np.argmax(probs, axis=1)
```

Finally, here comes the function to train our Neural Network. It implements batch gradient descent using the backpropagation derivates we found above.

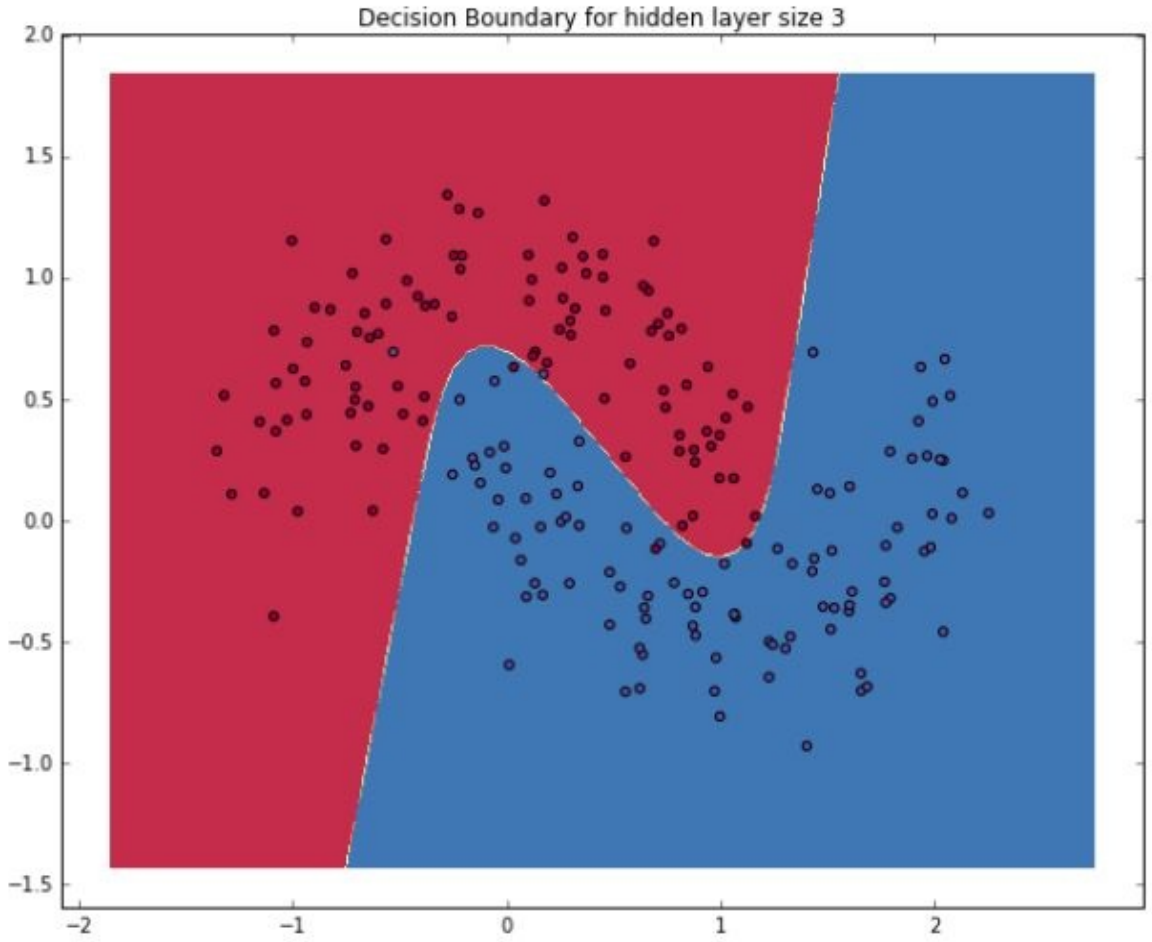
```
# This function learns parameters for the neural network and returns the model.
# - nn_hdim: Number of nodes in the hidden layer
# - num_passes: Number of passes through the training data for gradient descent
# - print_loss: If True, print the loss every 1000 iterations
def build_model(nn_hdim, num_passes=20000, print_loss=False):

    # Initialize the parameters to random values. We need to learn these.
    np.random.seed(0)
    W1 = np.random.randn(nn_input_dim, nn_hdim) / np.sqrt(nn_input_dim)
    b1 = np.zeros((1, nn_hdim))
    W2 = np.random.randn(nn_hdim, nn_output_dim) / np.sqrt(nn_hdim)
    b2 = np.zeros((1, nn_output_dim))

    # This is what we return at the end
    model = {}
```

```
# Gradient descent. For each batch…
for i in xrange(0, num_passes):

# Forward propagation
z1 = X.dot(W1) + b1
a1 = np.tanh(z1)
z2 = a1.dot(W2) + b2
exp_scores = np.exp(z2)
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

# Backpropagation
delta3 = probs
delta3[range(num_examples), y] -= 1
dW2 = (a1.T).dot(delta3)
db2 = np.sum(delta3, axis=0, keepdims=True)
delta2 = delta3.dot(W2.T) * (1 - np.power(a1, 2))
dW1 = np.dot(X.T, delta2)
db1 = np.sum(delta2, axis=0)

# Add regularization terms (b1 and b2 don’t have regularization terms)
dW2 += reg_lambda * W2
dW1 += reg_lambda * W1

# Gradient descent parameter update
W1 += -epsilon * dW1
b1 += -epsilon * db1
W2 += -epsilon * dW2
b2 += -epsilon * db2

# Assign new parameters to the model
model = { ‘W1’: W1, ‘b1’: b1, ‘W2’: W2, ‘b2’: b2}

# Optionally print the loss.
# This is expensive because it uses the whole dataset, so we don’t want to do it too often.
if print_loss and i % 1000 == 0:
print “Loss after iteration %i: %f” %(i, calculate_loss(model))
```

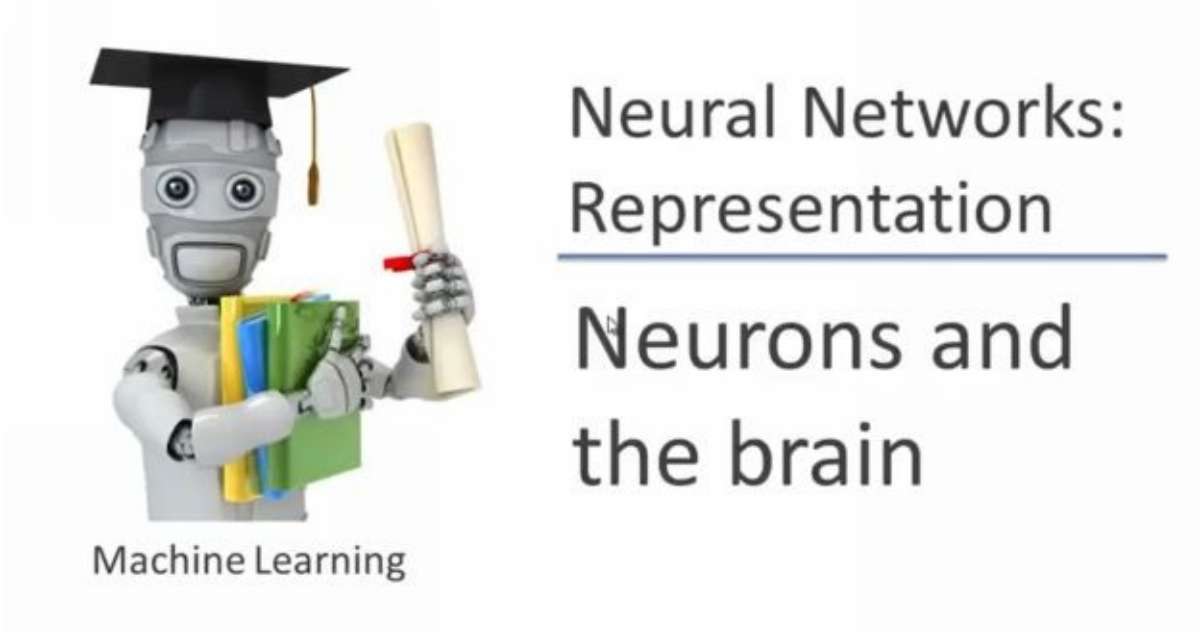
```
    return model
```

Let’s see what happens if we train a network with a hidden layer size of 3.

```
# Build a model with a 3-dimensional hidden layer
model = build_model(3, print_loss=True)

# Plot the decision boundary
plot_decision_boundary(lambda x: predict(model, x))
plt.title(“Decision Boundary for hidden layer size 3”)
```



Training Machine Learning Algorithms



Objective of this Chapter

The purpose of this chapter is to provide the introduction of the perceptron algorithms and the sonar dataset and we will discuss how to implement them in python.

Perceptron Algorithm

The Perceptron is motivated by the data processing of a single neural cell called a neuron.

A neuron acknowledges input signals through its dendrites, which pass the electrical signal down to the cell body.

Also, the Perceptron gets input signals from examples of training data that we weight and combined in a linear equation called the activation. After this activation is then transformed into an output value or prediction using a transfer function such as step transfer function

```
activation = sum(weight_i * x_i) + bias
```

```
prediction = 1.0 if activation >= 0.0 else 0.0
```

Sonar Dataset

This is a dataset that depicts sonar chirp returns bouncing off different services. The input factors which are 60 are the strength of the returns at different angles. It is a binary classification that requires a model to separate rocks from metal barrels.

It is surely a known dataset. The greater part is variables are consistent and generally in the range of 0 to 1. Thusly, we won't need to standardize the information, which is frequently a decent practice with the Perceptron algorithm. The output variable is a string "M" for mine and "R" for rock, which should be changed over to whole numbers 1 and 0.

By foreseeing the class with the most perceptions in the dataset (M or mines) the Zero Rule Algorithm can accomplish an exactness of 53%.

The three steps need to be followed for implementing perceptron in python:

Making Predictions

This will be required both in the assessment of candidate's weight value in sonar dataset and after the model is settled and we wish to begin making a prediction on test data or new data.

The following is a function named predict () that predicts an output of a row for a column given an arrangement of weights.

The principal weight is dependably the bias as it is independent and not in charge of a particular info value.

```
# Make a prediction with weights
def predict(row, weights):
        activation = weights[0]
        for i in range(len(row)-1):
        activation += weights[i + 1] * row[i]
        return 1.0 if activation >= 0.0 else 0.0
```

You can also contrive a small dataset to test our prediction function.

X1	X2	Y
2.7810836	2.550537003	0
1.465489372	2.362125076	0
3.396561688	4.400293529	0
1.38807019	1.850220317	0
3.06407232	3.005305973	0
7.627531214	2.759262235	1
5.332441248	2.088626775	1
6.922596716	1.77106367	1

8.675418651	-0.242068655	1
7.673756466	3.508563011	1

The previously prepared weights can also be used to make predictions for this dataset.

Now put all this together to test our predict () function below.

```
# Make a prediction with weights
def predict(row, weights):
        activation = weights[0]
        for i in range(len(row)-1):
                activation += weights[i + 1] * row[i]
        return 1.0 if activation >= 0.0 else 0.0

# test predictions
dataset = [[2.7810836,2.550537003,0],
        [1.465489372,2.362125076,0],
        [3.396561688,4.400293529,0],
        [1.38807019,1.850220317,0],
        [3.06407232,3.005305973,0],
        [7.627531214,2.759262235,1],
        [5.332441248,2.088626775,1],
        [6.922596716,1.77106367,1],
        [8.675418651,-0.242068655,1],
        [7.673756466,3.508563011,1]]
weights = [-0.1, 0.20653640140000007, -0.23418117710000003]
for row in dataset:
        prediction = predict(row, weights)
        print("Expected=%d, Predicted=%d" % (row[-1], prediction))
```

```
activation = (w1 * X1) + (w2 * X2) + bias
```

```
activation = (0.206 * X1) + (-0.234 * X2) + -0.1
```

Running this function, we get predictions

```
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=1, Predicted=1
Expected=1, Predicted=1
Expected=1, Predicted=1
Expected=1, Predicted=1
Expected=1, Predicted=1
```

Training Network Weights

We can use sonar dataset to estimate out weight values for training data. There are loops we need to perform in this function to get the desired results:

- Loop over each epoch.
- Loop over each row in the training data for an epoch.
- Loop over each weight and update it for a row in an epoch.

Now you can see after applying these loops we update each weight for each row in the training data, each epoch.

```
w(t+1)= w(t) + learning_rate * (expected(t) - predicted(t)) * x(t)
bias(t+1) = bias(t) + learning_rate * (expected(t) - predicted(t))
```

Now we can put all of this together. Below is a function named train_weights() that calculates weight values for a training dataset

using stochastic gradient descent.

```
# Estimate Perceptron weights using stochastic gradient descent
def train_weights(train, l_rate, n_epoch):
	weights = [0.0 for i in range(len(train[0]))]
	for epoch in range(n_epoch):
		sum_error = 0.0
		for row in train:
			prediction = predict(row, weights)
			error = row[-1] - prediction
			sum_error += error**2
			weights[0] = weights[0] + l_rate * error
			for i in range(len(row)-1):
				weights[i + 1] = weights[i + 1] + l_rate * error * row[i]
		print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))
	return weights
```

We can test this function on the same small contrived dataset from above.

```
# Make a prediction with weights
def predict(row, weights):
	activation = weights[0]
	for i in range(len(row)-1):
		activation += weights[i + 1] * row[i]
	return 1.0 if activation >= 0.0 else 0.0

# Estimate Perceptron weights using stochastic gradient descent
def train_weights(train, l_rate, n_epoch):
	weights = [0.0 for i in range(len(train[0]))]
	for epoch in range(n_epoch):
```

```
                sum_error = 0.0
                for row in train:
                        prediction = predict(row, weights)
                        error = row[-1] - prediction
                        sum_error += error**2
                        weights[0] = weights[0] + l_rate * error
                        for i in range(len(row)-1):
                                weights[i + 1] = weights[i + 1] + l_rate * error * row[i]
                print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))
        return weights

# Calculate weights
dataset = [[2.7810836,2.550537003,0],
        [1.465489372,2.362125076,0],
        [3.396561688,4.400293529,0],
        [1.38807019,1.850220317,0],
        [3.06407232,3.005305973,0],
        [7.627531214,2.759262235,1],
        [5.332441248,2.088626775,1],
        [6.922596716,1.77106367,1],
        [8.675418651,-0.242068655,1],
        [7.673756466,3.508563011,1]]
l_rate = 0.1
n_epoch = 5
weights = train_weights(dataset, l_rate, n_epoch)
print(weights)
```

Running the example prints a message each epoch with the sum squared error for that epoch and the final set of weights.

```
>epoch=0, lrate=0.100, error=2.000
>epoch=1, lrate=0.100, error=1.000
>epoch=2, lrate=0.100, error=0.000
>epoch=3, lrate=0.100, error=0.000
>epoch=4, lrate=0.100, error=0.000
[-0.1, 0.20653640140000007, -0.23418117710000003]
```

You can see how the problem is learned very quickly by the algorithm.

Now, let's apply this algorithm on a real dataset.

Modeling the Sonar Dataset

We will use the predict () and train_weights() functions created above to train the model and a new perceptron() function to tie them together.

Below is the complete example.

```
# Perceptron Algorithm on the Sonar Dataset
from random import seed
from random import randrange
from csv import reader

# Load a CSV file
def load_csv(filename):
	dataset = list()
	with open(filename, 'r') as file:
		csv_reader = reader(file)
		for row in csv_reader:
			if not row:
				continue
			dataset.append(row)
	return dataset
```

```
# Convert string column to float
def str_column_to_float(dataset, column):
	for row in dataset:
		row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
	class_values = [row[column] for row in dataset]
	unique = set(class_values)
	lookup = dict()
	for i, value in enumerate(unique):
		lookup[value] = i
	for row in dataset:
		row[column] = lookup[row[column]]
	return lookup

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
	dataset_split = list()
	dataset_copy = list(dataset)
	fold_size = int(len(dataset) / n_folds)
	for i in range(n_folds):
		fold = list()
		while len(fold) < fold_size:
			index = randrange(len(dataset_copy))
			fold.append(dataset_copy.pop(index))
		dataset_split.append(fold)
	return dataset_split

# Calculate accuracy percentage
```

```
def accuracy_metric(actual, predicted):
	correct = 0
	for i in range(len(actual)):
		if actual[i] == predicted[i]:
			correct += 1
	return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
	folds = cross_validation_split(dataset, n_folds)
	scores = list()
	for fold in folds:
		train_set = list(folds)
		train_set.remove(fold)
		train_set = sum(train_set, [])
		test_set = list()
		for row in fold:
			row_copy = list(row)
			test_set.append(row_copy)
			row_copy[-1] = None
		predicted = algorithm(train_set, test_set, *args)
		actual = [row[-1] for row in fold]
		accuracy = accuracy_metric(actual, predicted)
		scores.append(accuracy)
	return scores

# Make a prediction with weights
def predict(row, weights):
	activation = weights[0]
	for i in range(len(row)-1):
		activation += weights[i + 1] * row[i]
```

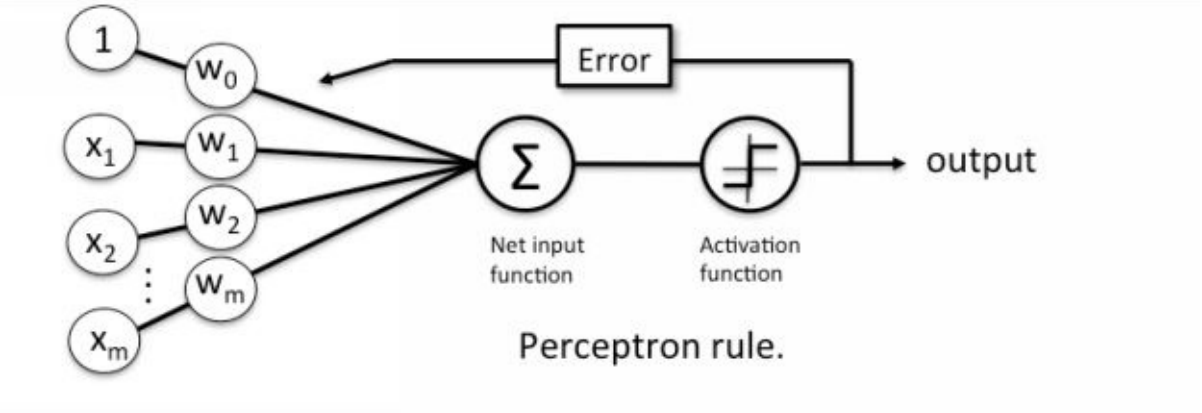
```
        return 1.0 if activation >= 0.0 else 0.0

# Estimate Perceptron weights using stochastic gradient descent
def train_weights(train, l_rate, n_epoch):
        weights = [0.0 for i in range(len(train[0]))]
        for epoch in range(n_epoch):
                for row in train:
                        prediction = predict(row, weights)
                        error = row[-1] - prediction
                        weights[0] = weights[0] + l_rate * error
                        for i in range(len(row)-1):
                                weights[i + 1] = weights[i + 1] + l_rate * error * row[i]
        return weights

# Perceptron Algorithm With Stochastic Gradient Descent
def perceptron(train, test, l_rate, n_epoch):
        predictions = list()
        weights = train_weights(train, l_rate, n_epoch)
        for row in test:
                prediction = predict(row, weights)
                predictions.append(prediction)
        return(predictions)

# Test the Perceptron algorithm on the sonar dataset
seed(1)
# load and prepare data
filename = 'sonar.all-data.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])-1):
        str_column_to_float(dataset, i)
# convert string class to integers
str_column_to_int(dataset, len(dataset[0])-1)
```

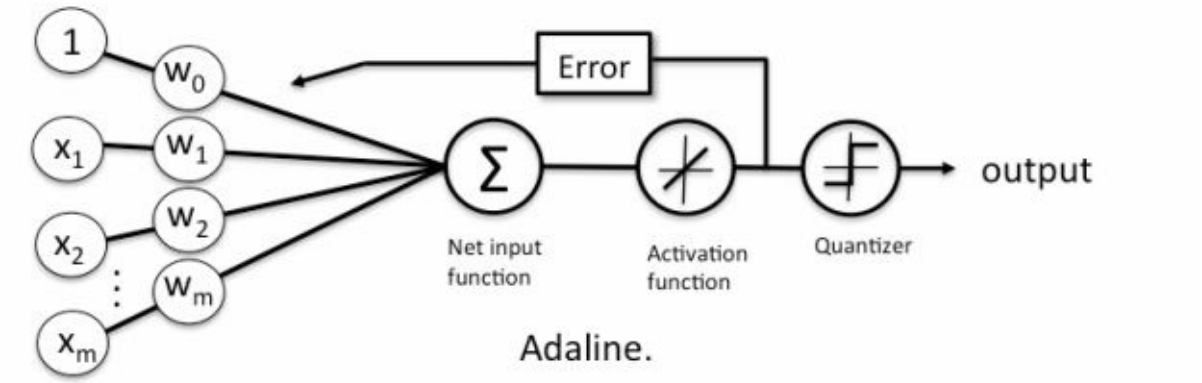
```
# evaluate algorithm
n_folds = 3
l_rate = 0.01
n_epoch = 500
scores = evaluate_algorithm(dataset, perceptron, n_folds, l_rate, n_epoch)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

Implementing an Adaptive Linear Neuron in Python

The perceptron most likely was extremely well known at the time of its discovery, be that as it may, it just took a couple of years until Bernard Widrow and his doctoral understudy Tedd Hoff proposed the possibility of the Adaptive Linear Neuron(ADALINE)

As opposed to the perceptron control, the delta rule of the Adaline (otherwise called Widrow-Hoff" run or Adaline govern) refreshes the weights in light of a linear activation function instead of a unit step function;this linear activation function g(z) is only the identity function of the new information g(wTx)=wTx. In the following area, we will perceive any reason why this direct initiation is a change over the perceptron refresh and where the name "delta rule" originates from.





Gradient Descent

Being a continuous function, one of the greatest focal points of the linear activation function over the unit step function is that it is differentiable. This property enables us to characterize a cost function J(w) that we can limit with a specific end goal to refresh our weights. On account of the linear activation function, we can characterize the cost function J(w) as the sum of squared errors (SSE), which is like the cost function that is limited in ordinary least squares (OLS) linear regression.

$J(w)=12\sum i(target(i)-output(i))2output(i) \in R$

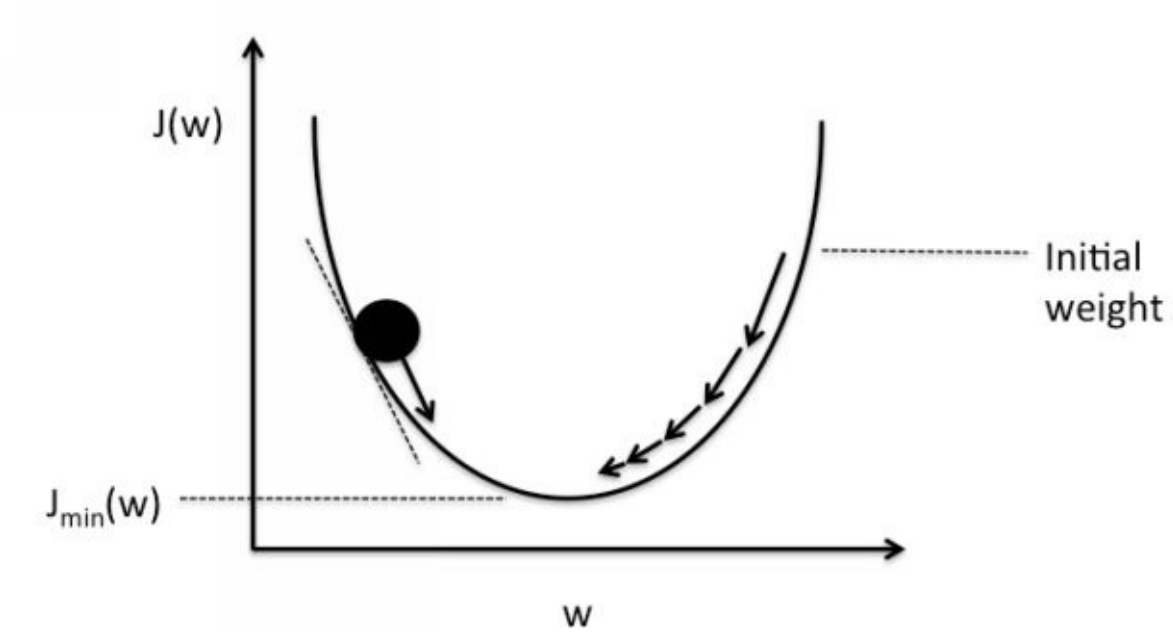
Being a continuous function, one of the greatest focal points of the linear activation function over the unit step function is that it is differentiable. This property enables us to characterize a cost function J(w) that we can limit with a specific end goal to refresh our weights. On account of the linear activation function, we can characterize the cost function J(w) as the sum of squared errors (SSE), which is like the cost function that is limited in ordinary least squares (OLS) linear regression.

$J(w)=12\sum i(target(i)-output(i))2output(i) \in R$

(The fraction 12 is just used for convenience to derive the gradient as we will see in the next paragraphs.)

With a specific end goal to limit the SSE cost function, we will utilize gradient descent, a basic yet valuable optimization algorithm that is frequently utilized as a part of machine learning out how to locate the neighborhood least of linear frameworks.

Before we get to the fun part (analytics), let us consider a convex cost function for one single weight. As represented in the figure beneath, we can portray the rule of gradient descent as “moving down a slope” until the point that a nearby or global minimum comes. At each progression, we step into the other way of the inclination, and the progression measure is controlled by the estimation of the learning rate and also the slant of the slope.

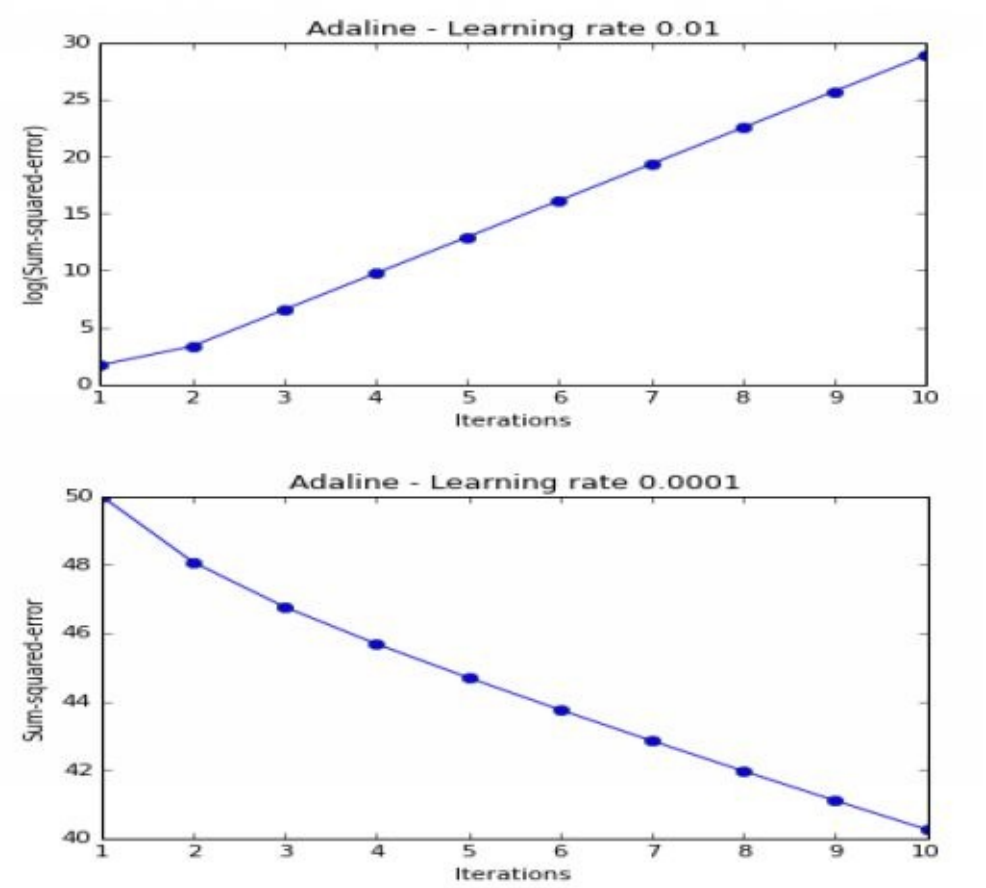


Now, it is time to implement the gradient descent rule in Python.

```
import numpy as np

class AdalineGD(object):

    def __init__(self, eta=0.01, epochs=50):
```

```
        self.eta = eta
        self.epochs = epochs

        def train(self, X, y):

        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []

        for i in range(self.epochs):
        output = self.net_input(X)
        errors = (y - output)
        self.w_[1:] += self.eta * X.T.dot(errors)
        self.w_[0] += self.eta * errors.sum()
        cost = (errors**2).sum() / 2.0
        self.cost_.append(cost)
        return self

        def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]

        def activation(self, X):
        return self.net_input(X)

        def predict(self, X):
        return np.where(self.activation(X) >= 0.0, 1, -1)
ada = AdalineGD(epochs=10, eta=0.01).train(X, y)
plt.plot(range(1, len(ada.cost_)+1), np.log10(ada.cost_), marker='o')
plt.xlabel('Iterations')
plt.ylabel('log(Sum-squared-error)')
plt.title('Adaline - Learning rate 0.01')
plt.show()

ada = AdalineGD(epochs=10, eta=0.0001).train(X, y)
plt.plot(range(1, len(ada.cost_)+1), ada.cost_, marker='o')
plt.xlabel('Iterations')
plt.ylabel('Sum-squared-error')
```

```
plt.title('Adaline - Learning rate 0.0001')
plt.show()
```

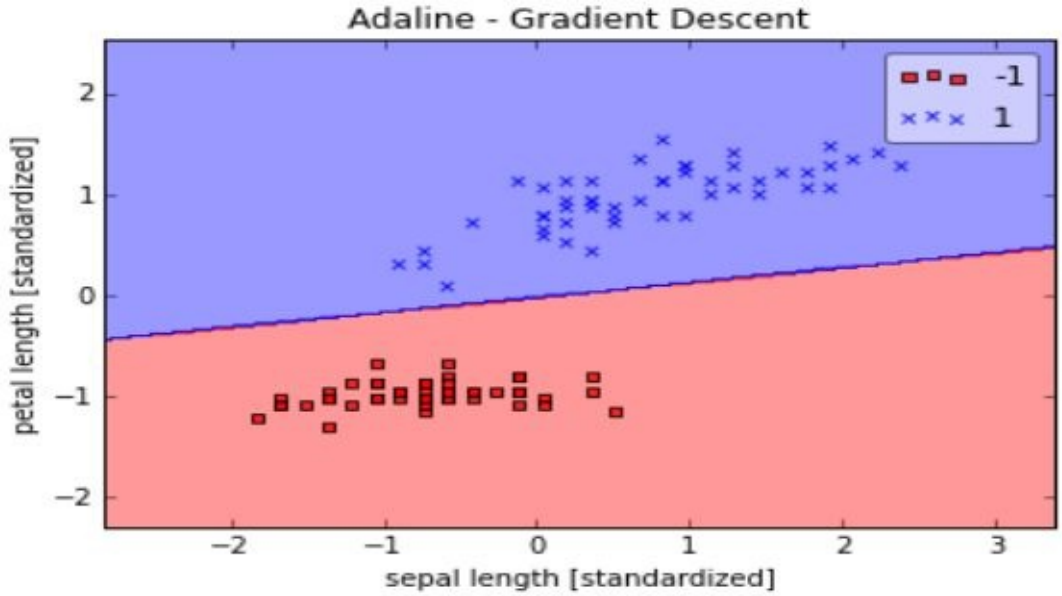


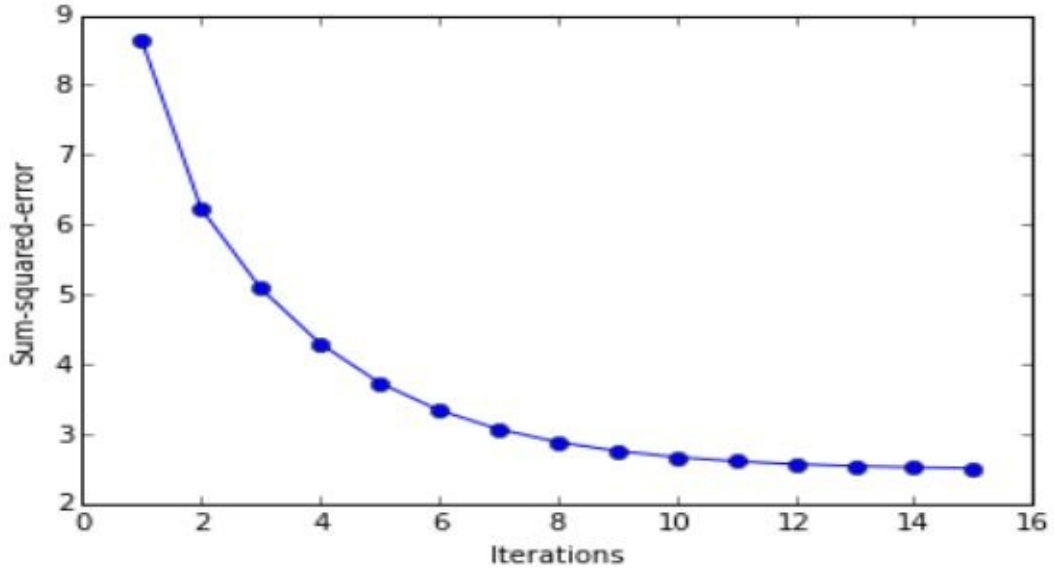
Standardize features

```
X_std = np.copy(X)
X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
%matplotlib inline
import matplotlib.pyplot as plt
from mlxtend.plotting import plot_decision_regions

ada = AdalineGD(epochs=15, eta=0.01)

ada.train(X_std, y)
plot_decision_regions(X_std, y, clf=ada)
plt.title('Adaline - Gradient Descent')
```

```
plt.xlabel(‘sepal length [standardized]’)
plt.ylabel(‘petal length [standardized]’)
plt.show()

plt.plot(range(1, len( ada.cost_)+1), ada.cost_, marker=‘o’)
plt.xlabel(‘Iterations’)
plt.ylabel(‘Sum-squared-error’)
plt.show()

import numpy as np

class AdalineSGD(object):

    def __init__(self, eta=0.01, epochs=50):
    self.eta = eta
    self.epochs = epochs

    def train(self, X, y, reinitialize_weights=True):

    if reinitialize_weights:
    self.w_ = np.zeros(1 + X.shape[1])
    self.cost_ = []

    for i in range(self.epochs):
    for xi, target in zip(X, y):
    output = self.net_input(xi)
    error = (target - output)
    self.w_[1:] += self.eta * xi.dot(error)
    self.w_[0] += self.eta * error

    cost = ((y - self.activation(X))**2).sum() / 2.0
    self.cost_.append(cost)
    return self

    def net_input(self, X):
    return np.dot(X, self.w_[1:]) + self.w_[0]
```

```
    def activation(self, X):
    return self.net_input(X)

    def predict(self, X):
    return np.where(self.activation(X) >= 0.0, 1, -1)
ada = AdalineSGD(epochs=15, eta=0.01)

# shuffle data
np.random.seed(123)
idx = np.random.permutation(len(y))
X_shuffled, y_shuffled =  X_std[idx], y[idx]

# train and adaline and plot decision regions
ada.train(X_shuffled, y_shuffled)
plot_decision_regions(X_shuffled, y_shuffled, clf=ada)
plt.title('Adaline - Gradient Descent')
plt.xlabel('sepal length [standardized]')
plt.ylabel('petal length [standardized]')
plt.show()

plt.plot(range(1, len(ada.cost_)+1), ada.cost_, marker='o')
plt.xlabel('Iterations')
plt.ylabel('Sum-squared-error')
plt.show()
```


Adaline - Gradient Descent
-1
1
petal length [standardized]
sepal length [standardized]
2
1
0
−1
−2
−2
−1
0
1
2
3


Sum-squared-error
Iterations
9
8
7
6
5
4
3
2
0
2
4
6
8
10
12
14
16

Combining Different Models for ensemble learning

Objective of this Chapter

At the end of this chapter, the reader should have learned:

- Majority Vote Classifier application with Python
- Bagging Technique

The objective behind ensemble techniques is to combine diverse classifiers into a meta-classifier that has a better execution than every individual classifier alone. For instance, accepting that we gathered expectations from 10 specialists, ensemble methods would enable us to deliberately join these predictions by the 10 specialists to think of a prediction that is more exact and hearty than the predictions by every individual master. As we will see later in this section, there are a few diverse methodologies for making an ensemble of classifiers. In this area, we will present a basic perception of how ensembles work and why they are commonly perceived as yielding a better execution.

Implementing a simple majority vote classifier

After the basic concept of ensemble learning in the past segment, we should begin with a warm-up exercise and execute a basic ensemble classifier for majority voting in Python. Despite the fact that the accompanying algorithm likewise sums up to multi-class settings by means of majority voting, we will utilize the term greater part voting in favor of straightforwardness as is additionally regularly done in writing.

The algorithm that we will execute will enable us to join distinctive classification algorithms related to singular weights for certainty. We will probably assemble a more grounded meta-classifier that adjusts out the individual classifiers' shortcomings on a specific dataset. In more exact scientific terms, we can compose the weighted majority vote as follows:

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j \chi_A \left(C_j(\boldsymbol{x}) = i \right)$$

Here, Wj is a weight associated with a base classifier, Cj is the predicted class label of the ensemble, X_A (Greek Chi) is the characteristic function and A is the set of unique class labels.

```
from sklearn.base import BaseEstimator
from sklearn.base import ClassifierMixin
import numpy as np
import operator
```

```
class EnsembleClassifier(BaseEstimator, ClassifierMixin):
    """
    Ensemble classifier for scikit-learn estimators.

    Parameters
    ----------

    clf : `iterable`
    A list of scikit-learn classifier objects.
    weights : `list` (default: `None`)
    If `None`, the majority rule voting will be applied to the predicted class labels.
    If a list of weights (`float` or `int`) is provided, the averaged raw probabilities (via `predict_proba`)
    will be used to determine the most confident class label.

    """
    def __init__(self, clfs, weights=None):
    self.clfs = clfs
    self.weights = weights

    def fit(self, X, y):
    """
    Fit the scikit-learn estimators.

    Parameters
    ----------

    X : numpy array, shape = [n_samples, n_features]
    Training data
    y : list or numpy array, shape = [n_samples]
    Class labels

    """
    for clf in self.clfs:
    clf.fit(X, y)
```

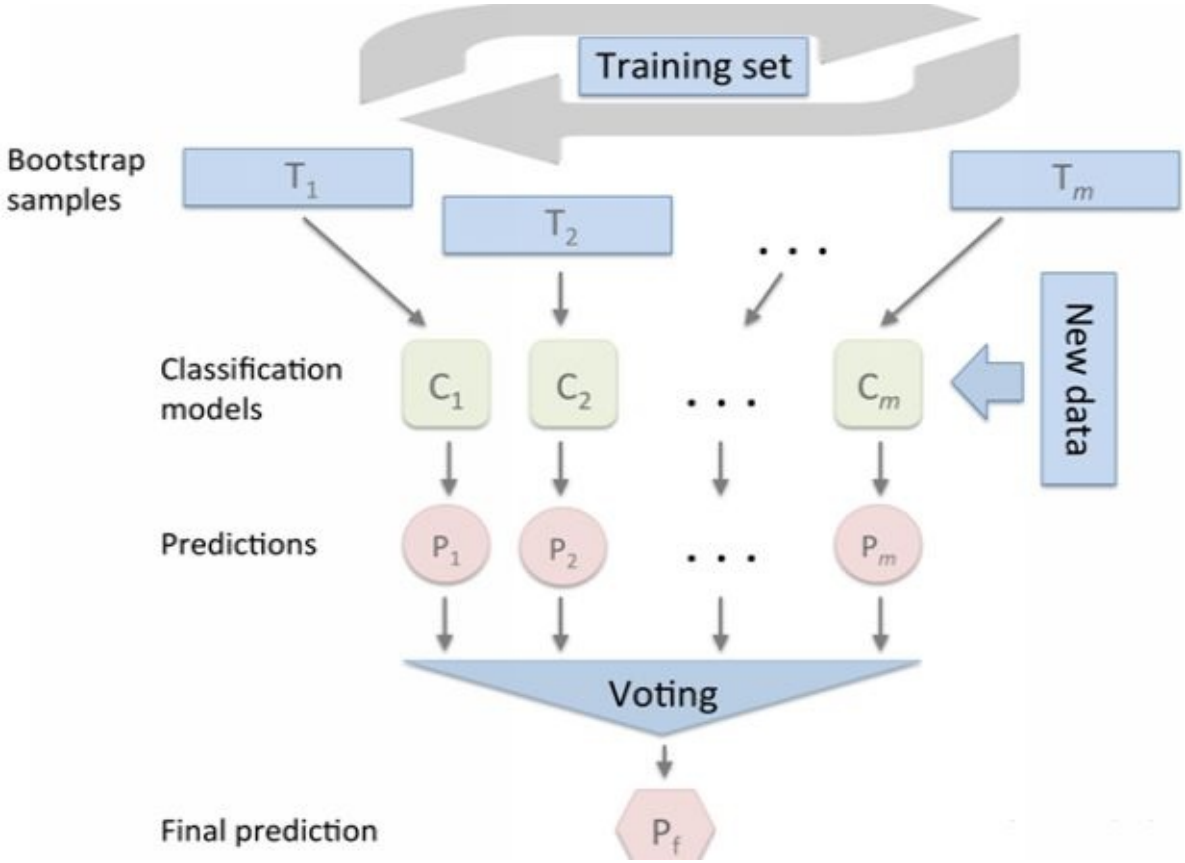
```
    def predict(self, X):
    """
    Parameters
    ----------

    X : numpy array, shape = [n_samples, n_features]

    Returns
    ----------

    maj : list or numpy array, shape = [n_samples]
    Predicted class labels by majority rule

    """

    self.classes_ = np.asarray([clf.predict(X) for clf in self.clfs])
    if self.weights:
    avg = self.predict_proba(X)

    maj = np.apply_along_axis(lambda x: max(enumerate(x), key=operator.itemgetter(1))[0], axis=1,
arr=avg)

    else:
    maj = np.asarray([np.argmax(np.bincount(self.classes_[:,c])) for c in range(self.classes_.shape[1])])

    return maj

    def predict_proba(self, X):

    """
    Parameters
    ----------

    X : numpy array, shape = [n_samples, n_features]

    Returns
```

```
    ——-

    avg : list or numpy array, shape = [n_samples, n_probabilities]
    Weighted average probability for each class per sample.

    """
    self.probas_ = [clf.predict_proba(X) for clf in self.clfs]
    avg = np.average(self.probas_, axis=0, weights=self.weights)

    return avg
np.random.seed(123)
eclf = EnsembleClassifier(clfs=[clf1, clf2, clf3], weights=[1,1,1])

for clf, label in zip([clf1, clf2, clf3, eclf], ['Logistic Regression', 'Random Forest', 'naive Bayes',
'Ensemble']):

    scores = cross_validation.cross_val_score(clf, X, y, cv=5, scoring='accuracy')
    print("Accuracy: %0.2f (+/- %0.2f) [%s]" % (scores.mean(), scores.std(), label))
```

```
Accuracy: 0.90 (+/- 0.05) [Logistic Regression]
Accuracy: 0.92 (+/- 0.05) [Random Forest]
Accuracy: 0.91 (+/- 0.04) [naive Bayes]
Accuracy: 0.95 (+/- 0.03) [Ensemble]
```

Bagging

This is an ensemble learning method that is closely related to MajorityVoteClassifier that we have implemented above and this method is illustrated with the help of diagram:



In this method instead of using same training set to fit individual classifiers in ensemble, we draw bootstrap sample from the starting training set.

Sample indices	Bagging round 1	Bagging round 2	...
1	2	7	...
2	2	3	...
3	1	2	...
4	3	1	...
5	7	1	...
6	2	7	...
7	4	7	...
	C_1	C_2	C_m

Conclusion

Overall, here we are! The end of the book. While you most likely do not have the same depth of understanding in machine learning, I trust you have picked up something. In particular, I seek you’ve built up a point of view after moving toward issues that machine learning works so well at solving. I solidly trust that utilizing tests is the main way that we can adequately utilize the scientific method. Modern world exists, and it helps us to become much better at writing code. Obviously, you cannot compose a test for everything, except the mentality matters. Furthermore, ideally, you have taken in somewhat about how you can apply that mentality to machine learning.

Thank you !

Thank you for buying this book! It is intended to help you understanding machine learning with Python. If you enjoyed this book and felt that it added value to your life, we ask that you please take the time to review it.

Your honest feedback would be greatly appreciated. It really does make a difference.

We are a very small publishing company and our survival depends on your reviews. Please, take a minute to write us your review.