**Artificial Intelligence**

According to the father of Artificial Intelligence, John McCarthy, it is

“The science and engineering of making intelligent machines, especially intelligent computer programs”.

“Can a machine think and behave like humans do?”

**Intelligence?**

The ability of a system to calculate, reason, perceive relationships and analogies, learn from experience, store and retrieve information from memory, solve problems, comprehend complex ideas, use natural language fluently, classify, generalize, and adapt new situations.

**Why python for AI**

Simple syntax & less coding

* Python involves very less coding and simple syntax among other programming languages which can be used for developing AI applications. Due to this testing can be easier and we can focus more on programming.

Inbuilt libraries for AI Projects

* A major advantage for using Python for AI is that it comes with inbuild libraries. Python has libraries for almost all kinds of AI projects. For e.g. ***NumPy, SciPy, matplotlib, nltk, SimpleAI*** are some of the important inbuilt libraries of Python.
* Open source – Python is an open source programming language.
* Can be used for broad range of programming tasks like small shell script to enterprise web applications.

**Machine Learning**

Machine Learning Algorithms helps computer system learn without being explicitly programmed. These algorithms are categorized into **supervised and unsupervised**.

**Supervised machine learning algorithm**

Supervised ML is the process of learning from the training dataset can be **thought of as teacher supervising the learning process**. In this kind of algorithm, the possible outcomes already known, and training data also labeled with correct answers.

Suppose we have input variables **x** and an output variable **y** and we applied an algorithm to learn the mapping function from the input to output such as

Y= f(x)

Now, the main goal is to approximate the mapping function so well that when we have new input data

Mainly, supervised learning problems can be divided into following two kids of problems.

**Classification:** A problem is called classification problem when we have categorized output such as **“black”, ”teaching” , “non-teaching” , etc.**

**Regression**: A problem is called regression problem when we have the real value output such as **“distance” “kilogram”** etc.

Decision tree, random forest, knn, logistic regression are the examples of supervised learning algorithms.

**Unsupervised algorithms**

As the name suggests, these kinds of machine learning algorithms do not have any supervisor to provide any sort of guidance. That is why unsupervised machine learning algorithms are closely aligned with what some call true artificial intelligence. It can be understood as follow –

Suppose we have input variable x, then there will be no corresponding output variables as there is in supervised learning algorithms.

In simple words, we can say that in unsupervised learning there will be no correct answer and no teacher for the guidance. Algorithms help to discover interesting patterns in data.

Unsupervised learning problems can be divided into the following two kinds of problem –

**Clustering**: In clustering problems, we need to discover the inherent groupings in the data. For example, grouping customers by their purchasing behavior.

**Association**: A problem is called association problem because such kinds of problem require discovering the rules that describe large portions of out data. For example, finding the customers who buy both x and y.

**Reinforcement machine learning algorithms**

These kinds of machine learning algorithms are used very less. These algorithms train the systems to make specific decisions. Basically, the machine is exposed to an environment where it trains itself continually using the trail and error method. These algorithms learn from past experience and tries to capture the best possible knowledge to make accurate decisions.

**Most Common Machine Learning Algorithms**

**Linear Regression:** Mainly linear regression is a linear model that assumes a linear relationship between the input variables say **x** and the single output variable say **y**.

Its one of the most well-known algorithms in statistics and machine learning.

**Basic concept –** Mainly linear regression is a linear model that assumes a linear relationship between the input variables say x and single output variable say y. In other words, we can say that y can be calculated from a linear combination of the input variables x. The relationship between variables can be established by fitting a best line.

**Types of Linear Regression**

**Simple Linear Regression –** A linear regression algorithm is called simple linear regression if it is having only one independent variable.

**Multiple linear regression –** A linear regression algorithm is called multiple linear regression if it is having more than one independent variable.

**Linear regression is mainly used to estimate the real values based on continuous variable(s).** For example, the total sale of a shop in a day, based on real values, can be estimated by linear regression.

**Logistic Regression**

It is a classification algorithm and also known as logit regression.

Mainly logistic regression is a classification algorithm that issued to estimate the discrete values like 0 or 1, true or false, yes or no based on a given set of independent variable. Basically, it predicts the probability hence its output lies in between 0 and 1.

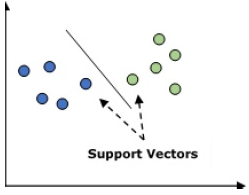
**Decision Tree**

Decision tree is a supervised learning algorithm that is mostly used for classification problem.

Basically, it is a classifier expressed as recursive partition based on the independent variables. Decision tree has nodes which form the rooted tree. Rooted tree is a directed tree with a node called “root”. Root does not have any incoming edges and all the other nodes have one incoming edge. These nodes are called leaves or decision nodes. For example, consider the following decision tree to see whether a person is fit or not.

**Support Vector Machine**

It is used for both classification and regression problems. But mainly it is used for classification problems. The main concept of SVM is to plot each data item as a point in n-dimensional space with the value of each feature being the value of a particular coordinate. Here n would be the features we would have. Following is a simple graphical representation to understand the concept of SVM



In the above diagram, we have two features hence we first need to plot these two variables in two dimensional space where each point has two co-ordinates called support vectors. The line splits the data into two different classified groups. This line would be the classifier.

**Naïve Bayes**

It is also a classification technique. The logic behind this classification technique is to use Bayes theorem for building classifiers. The assumption is that the predictors are independent. In simple words, it assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. Below is the equation for Bayes theorem—

$$P\left ( \frac{A}{B} \right ) = \frac{P\left ( \frac{B}{A} \right )P\left ( A \right )}{P\left ( B \right )}$$

The Naïve Bayes model is easy to build and particularly useful for large data sets.

**K-Nearest Neighbors (KNN)**

It is used for both classification and regression of the problem. It is widely used to solve classification problem. The main concept of this algorithm is that is used to store all the available cases and classifies new cases by majority votes of its kk neighbors. The case being then assigned to the class which is most common amongst its K-nearest neighbors, measured by a distance function. The distance function can be Euclidean, Minkowski and Hamming distance. Consider the following to use KNN –

* Computationally KNN are expensive than other algorithms used for classification problems.
* The normalization of variables needed otherwise higher range variables can bias it.
* In KNN, we need to work on pre-processing stage like noise removal.

**K-Means Clustering**

As the name suggests, it is used to solve the clustering problems. It is basically a type of unsupervised learning. The main logic of K-Means clustering algorithm is to classify the data set through a number of clusters. Follow these steps to form clusters by K-means

* K-means picks k number of points for each cluster known as centroids.
* Now each data point forms a cluster with the closest centroids. i.e. k clusters
* Now, it will find the centroids of each cluster based on existing cluster members.
* We need to repeat these until convergence occurs.

**Random Forest**

It is a supervised classification algorithm. The advantage of random forest algorithm is that it can be used for both classification and regression kind of problems. Basically, it is the collection of decision tress or you can say ensemble of the decision trees. The basic concept of random forest is that each tree gives a classification and the forest chooses the best classifications form them. Following are the advantages of Random Forest algorithm.

* Random forest classifier can be used for both classification and regression tasks
* They can handle missing values
* It won’t over fit the model even if we have more number of trees in the forest.

**Data Preparation**

In our daily life, we deal with lots of data but data is in raw form. To provide the data as the input of machine learning algorithms, we need to convert it into a meaningful data. That is where data preprocessing comes into picture. In other simple words, we can say that before providing the data to the machine learning algorithms we need preprocess the data.

**Data preprocessing steps:** Follow below steps in python for data preprocessing

**Step 1: Importing the useful packages -**

Import numpy as np

Import sklearn.preprocessing

**NumPy:** Basically NumPy is a general purpose array-processing package designed to efficiently manipulate large multi-dimensional arrays of arbitrary records without sacrificing too much speed for small multi-dimensional arrays.

**Sklearn.preprocessing:** This package provides many common utility functions and transformer classes to change raw feature vectors into a representation that is more suitable for machine learning algorithms.

**Step 2: Define sample data-**

After importing the packages, we need to define some sample data so that we can apply preprocessing techniques on that data. We will now define the following sample data-

Input\_data = np.array([2.1, -1.9, 5.5],

[-1.5, 2.4, 3.5],

[0.5, -7.9, 5.6],

[5.9, 2.3, -5.8])

**Step 3: Applying preprocessing technique**

In this step, in this step we need to apply any of the preprocessing techniques

**Techniques for Data preprocessing**

**Binarization:** This is the preprocessing technique which is used when we need to convert our numerical values into Boolean values. We can use an inbuilt method to binarize the input data say by using 0.5 as the threshold value in following way-

data\_binarized = preprocessing.Binarizer( threshold =0.5 ).transform(input\_data)

print(“\n Binarized data: \n” , data\_binarized)

Now, after running the above code we will get the following output, all the values greater than 0.5 will converted into 1 and all values less than 0.5 will be converted into 0.

**Binarized data**

[ [1. 0. 1.]

[0. 1. 1.]

[1. 1. 0.]

]

**Mean Removal**

It is used to eliminate the mean from feature vector so that every feature is centered on zero. We can also remove the bias from the features in the feature vector. For applying mean removal preprocessing technique on the sample data, we can write the Python code shown below. The code will display the Mean and Standard deviation of the input data.

Print(“ Mean = “, input\_data.mean(axis = 0))

Print(“Std deviation = “, input\_data.std(axis = 0))

The output of above two print statements

Mean = [ 1.75 -1.275 2.2]

Std deviation = [ 2.71431391 4.20022321 4.69414529]

Now, the code below will remove the Mean and Standard deviation of input data

data\_scaled = preprocessing.scale(input\_data)

print("Mean =", data\_scaled.mean(axis=0))

print("Std deviation =", data\_scaled.std(axis = 0))

We will get following output

Mean = [ 1.11022302e-16 0.00000000e+00 0.00000000e+00]

Std deviation = [ 1. 1. 1.]

Scaling

It is another data preprocessing technique that is used to scale the feature vectors. Scaling of feature vectors is needed because the values of every feature can vary between many random values. In other words we can say that scaling is important because we do not want any feature to be synthetically large or small. With the help of the following Python code, we can do the scaling of our input data, i.e., feature vector −

**# Min max scaling**

data\_scaler\_minmax = preprocessing.MinMaxScaler(feature\_range=(0,1))

data\_scaled\_minmax = data\_scaler\_minmax.fit\_transform(input\_data)

print ("\nMin max scaled data:\n", data\_scaled\_minmax)

We will get the following output after running the above lines of code −

**Min max scaled data**

[ [ 0.48648649 0.58252427 0.99122807]

[ 0. 1. 0.81578947]

[ 0.27027027 0. 1. ]

[ 1. 0. 99029126 0. ]]

**Normalization**

Used to modify the feature vectors. Such kind of modification is necessary to measure the feature vectors on a common scale. Followings are two types of normalization which can be used in machine learning

**L1 Normalization**

It is also referred to as Least Absolute Deviations. This kind of normalization modifies the values so that the sum of the absolute values is always up to 1 in each row. It can be implemented on the input data with the help of following Python code.

# Normalize data

data\_normalized\_l1 = preprocessing.normalize(input\_data, norm = 'l1')

print("\nL1 normalized data:\n", data\_normalized\_l1)

The above line of code generates the following output &miuns;

L1 normalized data:

[[ 0.22105263 -0.2 0.57894737]

[ -0.2027027 0.32432432 0.47297297]

[ 0.03571429 -0.56428571 0.4 ]

[ 0.42142857 0.16428571 -0.41428571]]

**L2 Normalization**

It is also referred to as **least squares**. This kind of normalization modifies the values so that the sum of the squares is always up to 1 in each row. It can be implemented on the input data with the help of the following Python code −

# Normalize data

data\_normalized\_l2 = preprocessing.normalize(input\_data, norm = 'l2')

print("\nL2 normalized data:\n", data\_normalized\_l2)

The above line of code will generate the following output −

L2 normalized data:

[[ 0.33946114 -0.30713151 0.88906489]

[ -0.33325106 0.53320169 0.7775858 ]

[ 0.05156558 -0.81473612 0.57753446]

[ 0.68706914 0.26784051 -0.6754239 ]]

**Labelling the Data**

We already know that data in a certain format is necessary for machine learning algorithms. Another important requirement is that the data must be labelled properly before sending it as the input of machine learning algorithms. For example, if we talk about classification, there are lot of labels on the data. Those labels are in the form of words, numbers, etc. Functions related to machine learning in **sklearn** expect that the data must have number labels. Hence, if the data is in other form then it must be converted to numbers. This process of transforming the word labels into numerical form is called label encoding.

### Label encoding steps

Follow these steps for encoding the data labels in Python −

**Step1 − Importing the useful packages**

If we are using Python then this would be first step for converting the data into certain format, i.e., preprocessing. It can be done as follows −

import numpy as np

from sklearn import preprocessing

**Step 2 − Defining sample labels**

After importing the packages, we need to define some sample labels so that we can create and train the label encoder. We will now define the following sample labels −

# Sample input labels

input\_labels = ['red','black','red','green','black','yellow','white']

**Step 3 − Creating & training of label encoder object**

In this step, we need to create the label encoder and train it. The following Python code will help in doing this −

# Creating the label encoder

encoder = preprocessing.LabelEncoder()

encoder.fit(input\_labels)

Following would be the output after running the above Python code −

LabelEncoder()

**Step4 − Checking the performance by encoding random ordered list**

This step can be used to check the performance by encoding the random ordered list. Following Python code can be written to do the same −

# encoding a set of labels

test\_labels = ['green','red','black']

encoded\_values = encoder.transform(test\_labels)

print("\nLabels =", test\_labels)

The labels would get printed as follows −

Labels = ['green', 'red', 'black']

Now, we can get the list of encoded values i.e. word labels converted to numbers as follows −

print("Encoded values =", list(encoded\_values))

The encoded values would get printed as follows −

Encoded values = [1, 2, 0]

**Step 5 − Checking the performance by decoding a random set of numbers −**

This step can be used to check the performance by decoding the random set of numbers. Following Python code can be written to do the same −

# decoding a set of values

encoded\_values = [3,0,4,1]

decoded\_list = encoder.inverse\_transform(encoded\_values)

print("\nEncoded values =", encoded\_values)

Now, Encoded values would get printed as follows −

Encoded values = [3, 0, 4, 1]

print("\nDecoded labels =", list(decoded\_list))

Now, decoded values would get printed as follows −

Decoded labels = ['white', 'black', 'yellow', 'green']

### Labeled v/s Unlabeled Data

Unlabeled data mainly consists of the samples of natural or human-created object that can easily be obtained from the world. They include, audio, video, photos, news articles, etc.

On the other hand, labeled data takes a set of unlabeled data and augments each piece of that unlabeled data with some tag or label or class that is meaningful. For example, if we have a photo then the label can be put based on the content of the photo, i.e., it is photo of a boy or girl or animal or anything else. Labeling the data needs human expertise or judgment about a given piece of unlabeled data.

There are many scenarios where unlabeled data is plentiful and easily obtained but labeled data often requires a human/expert to annotate. Semi-supervised learning attempts to combine labeled and unlabeled data to build better models.

**AI with Python Supervised Learning**

[Next Page](https://www.tutorialspoint.com/artificial_intelligence_with_python/artificial_intelligence_with_python_supervised_learning_regression.htm)

In this chapter, we will focus on implementing supervised learning − classification.

The classification technique or model attempts to get some conclusion from observed values. In classification problem, we have the categorized output such as “Black” or “white” or “Teaching” and “Non-Teaching”. While building the classification model, we need to have training dataset that contains data points and the corresponding labels. For example, if we want to check whether the image is of a car or not. For checking this, we will build a training dataset having the two classes related to “car” and “no car”. Then we need to train the model by using the training samples. The classification models are mainly used in face recognition, spam identification, etc.

## **Steps for Building a Classifier in Python**

For building a classifier in Python, we are going to use Python 3 and Scikit-learn which is a tool for machine learning. Follow these steps to build a classifier in Python −

### Step 1 − Import Scikit-learn

This would be very first step for building a classifier in Python. In this step, we will install a Python package called Scikit-learn which is one of the best machine learning modules in Python. The following command will help us import the package −

Import Sklearn

### Step 2 − Import Scikit-learn’s dataset

In this step, we can begin working with the dataset for our machine learning model. Here, we are going to use **the**[Breast Cancer Wisconsin Diagnostic Database.](http://scikit-learn.org/stable/datasets/index.html#breast-cancer-wisconsin-diagnostic-database) The dataset includes various information about breast cancer tumors, as well as classification labels of **malignant** or **benign**. The dataset has 569 instances, or data, on 569 tumors and includes information on 30 attributes, or features, such as the radius of the tumor, texture, smoothness, and area. With the help of the following command, we can import the Scikit-learn’s breast cancer dataset −

from sklearn.datasets import load\_breast\_cancer

Now, the following command will load the dataset.

data = load\_breast\_cancer()

Following is a list of important dictionary keys −

* Classification label names(target\_names)
* The actual labels(target)
* The attribute/feature names(feature\_names)
* The attribute (data)

Now, with the help of the following command, we can create new variables for each important set of information and assign the data. In other words, we can organize the data with the following commands −

label\_names = data['target\_names']

labels = data['target']

feature\_names = data['feature\_names']

features = data['data']

Now, to make it clearer we can print the class labels, the first data instance’s label, our feature names and the feature’s value with the help of the following commands −

print(label\_names)

The above command will print the class names which are malignant and benign respectively. It is shown as the output below −

['malignant' 'benign']

Now, the command below will show that they are mapped to binary values 0 and 1. Here 0 represents malignant cancer and 1 represents benign cancer. You will receive the following output −

print(labels[0])

0

The two commands given below will produce the feature names and feature values.

print(feature\_names[0])

mean radius

print(features[0])

[ 1.79900000e+01 1.03800000e+01 1.22800000e+02 1.00100000e+03

1.18400000e-01 2.77600000e-01 3.00100000e-01 1.47100000e-01

2.41900000e-01 7.87100000e-02 1.09500000e+00 9.05300000e-01

8.58900000e+00 1.53400000e+02 6.39900000e-03 4.90400000e-02

5.37300000e-02 1.58700000e-02 3.00300000e-02 6.19300000e-03

2.53800000e+01 1.73300000e+01 1.84600000e+02 2.01900000e+03

1.62200000e-01 6.65600000e-01 7.11900000e-01 2.65400000e-01

4.60100000e-01 1.18900000e-01]

From the above output, we can see that the first data instance is a malignant tumor the radius of which is 1.7990000e+01.

### Step 3 − Organizing data into sets

In this step, we will divide our data into two parts namely a training set and a test set. Splitting the data into these sets is very important because we have to test our model on the unseen data. To split the data into sets, sklearn has a function called the **train\_test\_split()** function. With the help of the following commands, we can split the data in these sets −

from sklearn.model\_selection import train\_test\_split

The above command will import the **train\_test\_split** function from sklearn and the command below will split the data into training and test data. In the example given below, we are using 40 % of the data for testing and the remaining data would be used for training the model.

train, test, train\_labels, test\_labels = train\_test\_split(features,labels,test\_size = 0.40, random\_state = 42)

### Step 4 − Building the model

In this step, we will be building our model. We are going to use Naïve Bayes algorithm for building the model. Following commands can be used to build the model −

from sklearn.naive\_bayes import GaussianNB

The above command will import the GaussianNB module. Now, the following command will help you initialize the model.

gnb = GaussianNB()

We will train the model by fitting it to the data by using gnb.fit().

model = gnb.fit(train, train\_labels)

### Step 5 − Evaluating the model and its accuracy

In this step, we are going to evaluate the model by making predictions on our test data. Then we will find out its accuracy also. For making predictions, we will use the predict() function. The following command will help you do this −

preds = gnb.predict(test)

print(preds)

[1 0 0 1 1 0 0 0 1 1 1 0 1 0 1 0 1 1 1 0 1 1 0 1 1 1 1 1 1

0 1 1 1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 0

0 1 1 0 0 1 1 1 0 0 1 1 0 0 1 0 1 1 1 1 1 1 0 1 1 0 0 0 0

0 1 1 1 1 1 1 1 1 0 0 1 0 0 1 0 0 1 1 1 0 1 1 0 1 1 0 0 0

1 1 1 0 0 1 1 0 1 0 0 1 1 0 0 0 1 1 1 0 1 1 0 0 1 0 1 1 0

1 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0

1 1 0 1 1 1 1 1 1 0 0 0 1 1 0 1 0 1 1 1 1 0 1 1 0 1 1 1 0

1 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1 0 0 1 1 0 1]

The above series of 0s and 1s are the predicted values for the tumor classes – malignant and benign.

Now, by comparing the two arrays namely **test\_labels** and **preds**, we can find out the accuracy of our model. We are going to use the **accuracy\_score()** function to determine the accuracy. Consider the following command for this −

from sklearn.metrics import accuracy\_score

print(accuracy\_score(test\_labels,preds))

0.951754385965

The result shows that the NaïveBayes classifier is 95.17% accurate.

In this way, with the help of the above steps we can build our classifier in Python.

## **Building Classifier in Python**

In this section, we will learn how to build a classifier in Python.

### Naïve Bayes Classifier

Naïve Bayes is a classification technique used to build classifier using the Bayes theorem. The assumption is that the predictors are independent. In simple words, it assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. For building Naïve Bayes classifier we need to use the python library called scikit learn. There are three types of Naïve Bayes models named **Gaussian, Multinomial and Bernoulli** under scikit learn package.

To build a Naïve Bayes machine learning classifier model, we need the following &minus

### Dataset

We are going to use the dataset named [Breast Cancer Wisconsin Diagnostic Database.](http://scikit-learn.org/stable/datasets/index.html#breast-cancer-wisconsin-diagnostic-database) The dataset includes various information about breast cancer tumors, as well as classification labels of **malignant** or **benign**. The dataset has 569 instances, or data, on 569 tumors and includes information on 30 attributes, or features, such as the radius of the tumor, texture, smoothness, and area. We can import this dataset from sklearn package.

### Naïve Bayes Model

For building Naïve Bayes classifier, we need a Naïve Bayes model. As told earlier, there are three types of Naïve Bayes models named **Gaussian, Multinomial** and **Bernoulli** under scikit learn package. Here, in the following example we are going to use the Gaussian Naïve Bayes model.

By using the above, we are going to build a Naïve Bayes machine learning model to use the tumor information to predict whether or not a tumor is malignant or benign.

To begin with, we need to install the sklearn module. It can be done with the help of the following command −

Import Sklearn

Now, we need to import the dataset named Breast Cancer Wisconsin Diagnostic Database.

from sklearn.datasets import load\_breast\_cancer

Now, the following command will load the dataset.

data = load\_breast\_cancer()

The data can be organized as follows −

label\_names = data['target\_names']

labels = data['target']

feature\_names = data['feature\_names']

features = data['data']

Now, to make it clearer we can print the class labels, the first data instance’s label, our feature names and the feature’s value with the help of following commands −

print(label\_names)

The above command will print the class names which are malignant and benign respectively. It is shown as the output below −

['malignant' 'benign']

Now, the command given below will show that they are mapped to binary values 0 and 1. Here 0 represents malignant cancer and 1 represents benign cancer. It is shown as the output below −

print(labels[0])

0

The following two commands will produce the feature names and feature values.

print(feature\_names[0])

mean radius

print(features[0])

[ 1.79900000e+01 1.03800000e+01 1.22800000e+02 1.00100000e+03

1.18400000e-01 2.77600000e-01 3.00100000e-01 1.47100000e-01

2.41900000e-01 7.87100000e-02 1.09500000e+00 9.05300000e-01

8.58900000e+00 1.53400000e+02 6.39900000e-03 4.90400000e-02

5.37300000e-02 1.58700000e-02 3.00300000e-02 6.19300000e-03

2.53800000e+01 1.73300000e+01 1.84600000e+02 2.01900000e+03

1.62200000e-01 6.65600000e-01 7.11900000e-01 2.65400000e-01

4.60100000e-01 1.18900000e-01]

From the above output, we can see that the first data instance is a malignant tumor the main radius of which is 1.7990000e+01.

For testing our model on unseen data, we need to split our data into training and testing data. It can be done with the help of the following code −

from sklearn.model\_selection import train\_test\_split

The above command will import the **train\_test\_split** function from sklearn and the command below will split the data into training and test data. In the below example, we are using 40 % of the data for testing and the remining data would be used for training the model.

train, test, train\_labels, test\_labels =

train\_test\_split(features,labels,test\_size = 0.40, random\_state = 42)

Now, we are building the model with the following commands −

from sklearn.naive\_bayes import GaussianNB

The above command will import the **GaussianNB** module. Now, with the command given below, we need to initialize the model.

gnb = GaussianNB()

We will train the model by fitting it to the data by using **gnb.fit()**.

model = gnb.fit(train, train\_labels)

Now, evaluate the model by making prediction on the test data and it can be done as follows −

preds = gnb.predict(test)

print(preds)

[1 0 0 1 1 0 0 0 1 1 1 0 1 0 1 0 1 1 1 0 1 1 0 1 1 1 1 1 1

0 1 1 1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 0

0 1 1 0 0 1 1 1 0 0 1 1 0 0 1 0 1 1 1 1 1 1 0 1 1 0 0 0 0

0 1 1 1 1 1 1 1 1 0 0 1 0 0 1 0 0 1 1 1 0 1 1 0 1 1 0 0 0

1 1 1 0 0 1 1 0 1 0 0 1 1 0 0 0 1 1 1 0 1 1 0 0 1 0 1 1 0

1 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0

1 1 0 1 1 1 1 1 1 0 0 0 1 1 0 1 0 1 1 1 1 0 1 1 0 1 1 1 0

1 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1 0 0 1 1 0 1]

The above series of 0s and 1s are the predicted values for the tumor classes i.e. malignant and benign.

Now, by comparing the two arrays namely **test\_labels** and **preds**, we can find out the accuracy of our model. We are going to use the **accuracy\_score()** function to determine the accuracy. Consider the following command −

from sklearn.metrics import accuracy\_score

print(accuracy\_score(test\_labels,preds))

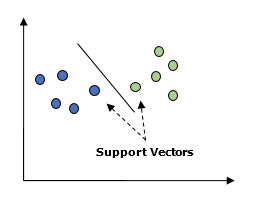
0.951754385965

The result shows that NaïveBayes classifier is 95.17% accurate.

That was machine learning classifier based on the Naïve Bayse Gaussian model.

### Support Vector Machines (SVM)

Basically, Support vector machine (SVM) is a supervised machine learning algorithm that can be used for both regression and classification. The main concept of SVM is to plot each data item as a point in n-dimensional space with the value of each feature being the value of a particular coordinate. Here n would be the features we would have. Following is a simple graphical representation to understand the concept of SVM −



In the above diagram, we have two features. Hence, we first need to plot these two variables in two dimensional space where each point has two co-ordinates, called support vectors. The line splits the data into two different classified groups. This line would be the classifier.

Here, we are going to build an SVM classifier by using scikit-learn and iris dataset. Scikitlearn library has the **sklearn.svm** module and provides sklearn.svm.svc for classification. The SVM classifier to predict the class of the iris plant based on 4 features are shown below.

### Dataset

We will use the iris dataset which contains 3 classes of 50 instances each, where each class refers to a type of iris plant. Each instance has the four features namely sepal length, sepal width, petal length and petal width. The SVM classifier to predict the class of the iris plant based on 4 features is shown below.

### Kernel

It is a technique used by SVM. Basically these are the functions which take low-dimensional input space and transform it to a higher dimensional space. It converts non-separable problem to separable problem. The kernel function can be any one among linear, polynomial, rbf and sigmoid. In this example, we will use the linear kernel.

Let us now import the following packages −

import pandas as pd

import numpy as np

from sklearn import svm, datasets

import matplotlib.pyplot as plt

Now, load the input data −

iris = datasets.load\_iris()

We are taking first two features −

X = iris.data[:, :2]

y = iris.target

We will plot the support vector machine boundaries with original data. We are creating a mesh to plot.

x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

h = (x\_max / x\_min)/100

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h),

np.arange(y\_min, y\_max, h))

X\_plot = np.c\_[xx.ravel(), yy.ravel()]

We need to give the value of regularization parameter.

C = 1.0

We need to create the SVM classifier object.

Svc\_classifier = svm\_classifier.SVC(kernel='linear',

C=C, decision\_function\_shape = 'ovr').fit(X, y)

Z = svc\_classifier.predict(X\_plot)

Z = Z.reshape(xx.shape)

plt.figure(figsize = (15, 5))

plt.subplot(121)

plt.contourf(xx, yy, Z, cmap = plt.cm.tab10, alpha = 0.3)

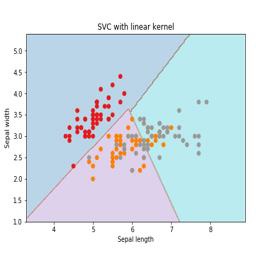
plt.scatter(X[:, 0], X[:, 1], c = y, cmap = plt.cm.Set1)

plt.xlabel('Sepal length')

plt.ylabel('Sepal width')

plt.xlim(xx.min(), xx.max())

plt.title('SVC with linear kernel')



## **Logistic Regression**

Basically, logistic regression model is one of the members of supervised classification algorithm family. Logistic regression measures the relationship between dependent variables and independent variables by estimating the probabilities using a logistic function.

Here, if we talk about dependent and independent variables then dependent variable is the target class variable we are going to predict and on the other side the independent variables are the features we are going to use to predict the target class.

In logistic regression, estimating the probabilities means to predict the likelihood occurrence of the event. For example, the shop owner would like to predict the customer who entered into the shop will buy the play station (for example) or not. There would be many features of customer − gender, age, etc. which would be observed by the shop keeper to predict the likelihood occurrence, i.e., buying a play station or not. The logistic function is the sigmoid curve that is used to build the function with various parameters.

### Prerequisites

Before building the classifier using logistic regression, we need to install the Tkinter package on our system. It can be installed from <https://docs.python.org/2/library/tkinter.html>.

Now, with the help of the code given below, we can create a classifier using logistic regression −

First, we will import some packages −

import numpy as np

from sklearn import linear\_model

import matplotlib.pyplot as plt

Now, we need to define the sample data which can be done as follows −

X = np.array([[2, 4.8], [2.9, 4.7], [2.5, 5], [3.2, 5.5], [6, 5], [7.6, 4],

[3.2, 0.9], [2.9, 1.9],[2.4, 3.5], [0.5, 3.4], [1, 4], [0.9, 5.9]])

y = np.array([0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3])

Next, we need to create the logistic regression classifier, which can be done as follows −

Classifier\_LR = linear\_model.LogisticRegression(solver = 'liblinear', C = 75)

Last but not the least, we need to train this classifier −

Classifier\_LR.fit(X, y)

Now, how we can visualize the output? It can be done by creating a function named Logistic\_visualize() −

Def Logistic\_visualize(Classifier\_LR, X, y):

min\_x, max\_x = X[:, 0].min() - 1.0, X[:, 0].max() + 1.0

min\_y, max\_y = X[:, 1].min() - 1.0, X[:, 1].max() + 1.0

In the above line, we defined the minimum and maximum values X and Y to be used in mesh grid. In addition, we will define the step size for plotting the mesh grid.

mesh\_step\_size = 0.02

Let us define the mesh grid of X and Y values as follows −

x\_vals, y\_vals = np.meshgrid(np.arange(min\_x, max\_x, mesh\_step\_size),

np.arange(min\_y, max\_y, mesh\_step\_size))

With the help of following code, we can run the classifier on the mesh grid −

output = classifier.predict(np.c\_[x\_vals.ravel(), y\_vals.ravel()])

output = output.reshape(x\_vals.shape)

plt.figure()

plt.pcolormesh(x\_vals, y\_vals, output, cmap = plt.cm.gray)

plt.scatter(X[:, 0], X[:, 1], c = y, s = 75, edgecolors = 'black',

linewidth=1, cmap = plt.cm.Paired)

The following line of code will specify the boundaries of the plot

plt.xlim(x\_vals.min(), x\_vals.max())

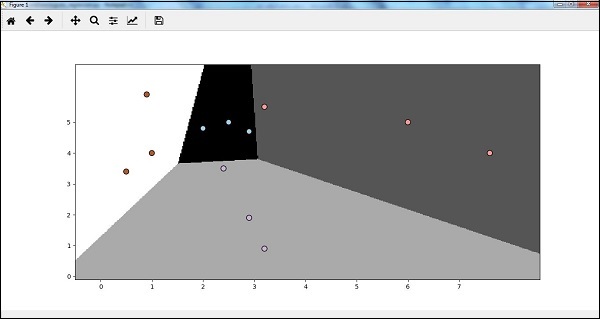
plt.ylim(y\_vals.min(), y\_vals.max())

plt.xticks((np.arange(int(X[:, 0].min() - 1), int(X[:, 0].max() + 1), 1.0)))

plt.yticks((np.arange(int(X[:, 1].min() - 1), int(X[:, 1].max() + 1), 1.0)))

plt.show()

Now, after running the code we will get the following output, logistic regression classifier −



## **Decision Tree Classifier**

A decision tree is basically a binary tree flowchart where each node splits a group of observations according to some feature variable.

Here, we are building a Decision Tree classifier for predicting male or female. We will take a very small data set having 19 samples. These samples would consist of two features – ‘height’ and ‘length of hair’.

### Prerequisite

For building the following classifier, we need to install **pydotplus** and **graphviz**. Basically, graphviz is a tool for drawing graphics using dot files and **pydotplus** is a module to Graphviz’s Dot language. It can be installed with the package manager or pip.

Now, we can build the decision tree classifier with the help of the following Python code −

To begin with, let us import some important libraries as follows −

import pydotplus

from sklearn import tree

from sklearn.datasets import load\_iris

from sklearn.metrics import classification\_report

from sklearn import cross\_validation

import collections

Now, we need to provide the dataset as follows −

X = [[165,19],[175,32],[136,35],[174,65],[141,28],[176,15],[131,32],

[166,6],[128,32],[179,10],[136,34],[186,2],[126,25],[176,28],[112,38],

[169,9],[171,36],[116,25],[196,25]]

Y = ['Man','Woman','Woman','Man','Woman','Man','Woman','Man','Woman',

'Man','Woman','Man','Woman','Woman','Woman','Man','Woman','Woman','Man']

data\_feature\_names = ['height','length of hair']

X\_train, X\_test, Y\_train, Y\_test = cross\_validation.train\_test\_split

(X, Y, test\_size=0.40, random\_state=5)

After providing the dataset, we need to fit the model which can be done as follows −

clf = tree.DecisionTreeClassifier()

clf = clf.fit(X,Y)

Prediction can be made with the help of the following Python code −

prediction = clf.predict([[133,37]])

print(prediction)

We can visualize the decision tree with the help of the following Python code −

dot\_data = tree.export\_graphviz(clf,feature\_names = data\_feature\_names,

out\_file = None,filled = True,rounded = True)

graph = pydotplus.graph\_from\_dot\_data(dot\_data)

colors = ('orange', 'yellow')

edges = collections.defaultdict(list)

for edge in graph.get\_edge\_list():

edges[edge.get\_source()].append(int(edge.get\_destination()))

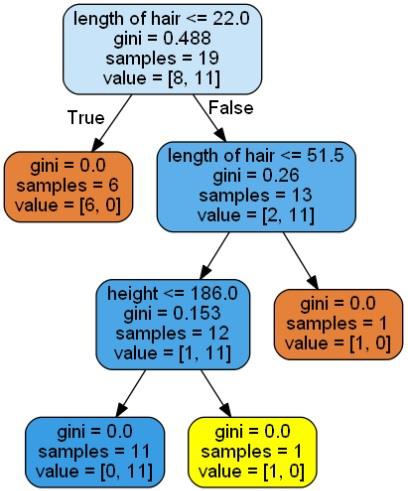
for edge in edges: edges[edge].sort()

for i in range(2):dest = graph.get\_node(str(edges[edge][i]))[0]

dest.set\_fillcolor(colors[i])

graph.write\_png('Decisiontree16.png')

It will give the prediction for the above code as **[‘Woman’]** and create the following decision tree −



We can change the values of features in prediction to test it.

## **Random Forest Classifier**

As we know that ensemble methods are the methods which combine machine learning models into a more powerful machine learning model. Random Forest, a collection of decision trees, is one of them. It is better than single decision tree because while retaining the predictive powers it can reduce over-fitting by averaging the results. Here, we are going to implement the random forest model on scikit learn cancer dataset.

Import the necessary packages −

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.datasets import load\_breast\_cancer

cancer = load\_breast\_cancer()

import matplotlib.pyplot as plt

import numpy as np

Now, we need to provide the dataset which can be done as follows &minus

cancer = load\_breast\_cancer()

X\_train, X\_test, y\_train,

y\_test = train\_test\_split(cancer.data, cancer.target, random\_state = 0)

After providing the dataset, we need to fit the model which can be done as follows −

forest = RandomForestClassifier(n\_estimators = 50, random\_state = 0)

forest.fit(X\_train,y\_train)

Now, get the accuracy on training as well as testing subset: if we will increase the number of estimators then, the accuracy of testing subset would also be increased.

print('Accuracy on the training subset:(:.3f)',format(forest.score(X\_train,y\_train)))

print('Accuracy on the training subset:(:.3f)',format(forest.score(X\_test,y\_test)))

### Output

Accuracy on the training subset:(:.3f) 1.0

Accuracy on the training subset:(:.3f) 0.965034965034965

Now, like the decision tree, random forest has the **feature\_importance** module which will provide a better view of feature weight than decision tree. It can be plot and visualize as follows −

n\_features = cancer.data.shape[1]

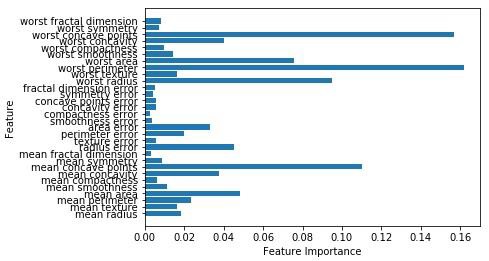
plt.barh(range(n\_features),forest.feature\_importances\_, align='center')

plt.yticks(np.arange(n\_features),cancer.feature\_names)

plt.xlabel('Feature Importance')

plt.ylabel('Feature')

plt.show()

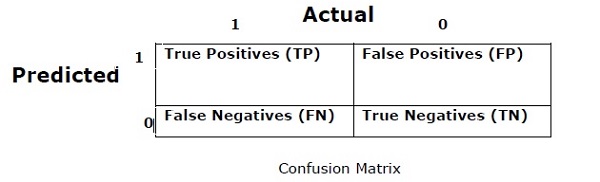


## **Performance of a classifier**

After implementing a machine learning algorithm, we need to find out how effective the model is. The criteria for measuring the effectiveness may be based upon datasets and metric. For evaluating different machine learning algorithms, we can use different performance metrics. For example, suppose if a classifier is used to distinguish between images of different objects, we can use the classification performance metrics such as average accuracy, AUC, etc. In one or other sense, the metric we choose to evaluate our machine learning model is very important because the choice of metrics influences how the performance of a machine learning algorithm is measured and compared. Following are some of the metrics −

### Confusion Matrix

Basically it is used for classification problem where the output can be of two or more types of classes. It is the easiest way to measure the performance of a classifier. A confusion matrix is basically a table with two dimensions namely “Actual” and “Predicted”. Both the dimensions have “True Positives (TP)”, “True Negatives (TN)”, “False Positives (FP)”, “False Negatives (FN)”.



In the confusion matrix above, 1 is for positive class and 0 is for negative class.

Following are the terms associated with Confusion matrix −

* **True Positives −** TPs are the cases when the actual class of data point was 1 and the predicted is also 1.
* **True Negatives −** TNs are the cases when the actual class of the data point was 0 and the predicted is also 0.
* **False Positives −** FPs are the cases when the actual class of data point was 0 and the predicted is also 1.
* **False Negatives −** FNs are the cases when the actual class of the data point was 1 and the predicted is also 0.

### Accuracy

The confusion matrix itself is not a performance measure as such but almost all the performance matrices are based on the confusion matrix. One of them is accuracy. In classification problems, it may be defined as the number of correct predictions made by the model over all kinds of predictions made. The formula for calculating the accuracy is as follows −

$$Accuracy = \frac{TP+TN}{TP+FP+FN+TN}$$

### Precision

It is mostly used in document retrievals. It may be defined as how many of the returned documents are correct. Following is the formula for calculating the precision −

$$Precision = \frac{TP}{TP+FP}$$

### Recall or Sensitivity

It may be defined as how many of the positives do the model return. Following is the formula for calculating the recall/sensitivity of the model −

$$Recall = \frac{TP}{TP+FN}$$

### Specificity

It may be defined as how many of the negatives do the model return. It is exactly opposite to recall. Following is the formula for calculating the specificity of the model −

$$Specificity = \frac{TN}{TN+FP}$$

## **Class Imbalance Problem**

Class imbalance is the scenario where the number of observations belonging to one class is significantly lower than those belonging to the other classes. For example, this problem is prominent in the scenario where we need to identify the rare diseases, fraudulent transactions in bank etc.

### Example of imbalanced classes

Let us consider an example of fraud detection data set to understand the concept of imbalanced class −

Total observations = 5000

Fraudulent Observations = 50

Non-Fraudulent Observations = 4950

Event Rate = 1%

### Solution

**Balancing the classes’** acts as a solution to imbalanced classes. The main objective of balancing the classes is to either increase the frequency of the minority class or decrease the frequency of the majority class. Following are the approaches to solve the issue of imbalances classes −

### Re-Sampling

Re-sampling is a series of methods used to reconstruct the sample data sets − both training sets and testing sets. Re-sampling is done to improve the accuracy of model. Following are some re-sampling techniques −

* **Random Under-Sampling** − This technique aims to balance class distribution by randomly eliminating majority class examples. This is done until the majority and minority class instances are balanced out.

Total observations = 5000

Fraudulent Observations = 50

Non-Fraudulent Observations = 4950

Event Rate = 1%

In this case, we are taking 10% samples without replacement from non-fraud instances and then combine them with the fraud instances −

Non-fraudulent observations after random under sampling = 10% of 4950 = 495

Total observations after combining them with fraudulent observations = 50+495 = 545

Hence now, the event rate for new dataset after under sampling = 9%

The main advantage of this technique is that it can reduce run time and improve storage. But on the other side, it can discard useful information while reducing the number of training data samples.

* **Random Over-Sampling** − This technique aims to balance class distribution by increasing the number of instances in the minority class by replicating them.

Total observations = 5000

Fraudulent Observations = 50

Non-Fraudulent Observations = 4950

Event Rate = 1%

In case we are replicating 50 fraudulent observations 30 times then fraudulent observations after replicating the minority class observations would be 1500. And then total observations in the new data after oversampling would be 4950+1500 = 6450. Hence the event rate for the new data set would be 1500/6450 = 23%.

The main advantage of this method is that there would be no loss of useful information. But on the other hand, it has the increased chances of over-fitting because it replicates the minority class events.

## **Ensemble Techniques**

This methodology basically is used to modify existing classification algorithms to make them appropriate for imbalanced data sets. In this approach we construct several two stage classifier from the original data and then aggregate their predictions. Random forest classifier is an example of ensemble based classifier.

Regression is one of the most important statistical and machine learning tools. We would not be wrong to say that the journey of machine learning starts from regression. It may be defined as the parametric technique that allows us to make decisions based upon data or in other words allows us to make predictions based upon data by learning the relationship between input and output variables. Here, the output variables dependent on the input variables, are continuous-valued real numbers. In regression, the relationship between input and output variables matters and it helps us in understanding how the value of the output variable changes with the change of input variable. Regression is frequently used for prediction of prices, economics, variations, and so on.

## **Building Regressors in Python**

In this section, we will learn how to build single as well as multivariable regressor.

### Linear Regressor/Single Variable Regressor

Let us important a few required packages −

import numpy as np

from sklearn import linear\_model

import sklearn.metrics as sm

import matplotlib.pyplot as plt

Now, we need to provide the input data and we have saved our data in the file named linear.txt.

input = 'D:/ProgramData/linear.txt'

We need to load this data by using the **np.loadtxt** function.

input\_data = np.loadtxt(input, delimiter=',')

X, y = input\_data[:, :-1], input\_data[:, -1]

The next step would be to train the model. Let us give training and testing samples.

training\_samples = int(0.6 \* len(X))

testing\_samples = len(X) - num\_training

X\_train, y\_train = X[:training\_samples], y[:training\_samples]

X\_test, y\_test = X[training\_samples:], y[training\_samples:]

Now, we need to create a linear regressor object.

reg\_linear = linear\_model.LinearRegression()

Train the object with the training samples.

reg\_linear.fit(X\_train, y\_train)

We need to do the prediction with the testing data.

y\_test\_pred = reg\_linear.predict(X\_test)

Now plot and visualize the data.

plt.scatter(X\_test, y\_test, color = 'red')

plt.plot(X\_test, y\_test\_pred, color = 'black', linewidth = 2)

plt.xticks(())

plt.yticks(())

plt.show()

### Output

Now, we can compute the performance of our linear regression as follows −

print("Performance of Linear regressor:")

print("Mean absolute error =", round(sm.mean\_absolute\_error(y\_test, y\_test\_pred), 2))

print("Mean squared error =", round(sm.mean\_squared\_error(y\_test, y\_test\_pred), 2))

print("Median absolute error =", round(sm.median\_absolute\_error(y\_test, y\_test\_pred), 2))

print("Explain variance score =", round(sm.explained\_variance\_score(y\_test, y\_test\_pred),

2))

print("R2 score =", round(sm.r2\_score(y\_test, y\_test\_pred), 2))

### Output

Performance of Linear Regressor −

Mean absolute error = 1.78

Mean squared error = 3.89

Median absolute error = 2.01

Explain variance score = -0.09

R2 score = -0.09

In the above code, we have used this small data. If you want some big dataset then you can use sklearn.dataset to import bigger dataset.

2,4.82.9,4.72.5,53.2,5.56,57.6,43.2,0.92.9,1.92.4,

3.50.5,3.41,40.9,5.91.2,2.583.2,5.65.1,1.54.5,

1.22.3,6.32.1,2.8

### Multivariable Regressor

First, let us import a few required packages −

import numpy as np

from sklearn import linear\_model

import sklearn.metrics as sm

import matplotlib.pyplot as plt

from sklearn.preprocessing import PolynomialFeatures

Now, we need to provide the input data and we have saved our data in the file named linear.txt.

input = 'D:/ProgramData/Mul\_linear.txt'

We will load this data by using the **np.loadtxt** function.

input\_data = np.loadtxt(input, delimiter=',')

X, y = input\_data[:, :-1], input\_data[:, -1]

The next step would be to train the model; we will give training and testing samples.

training\_samples = int(0.6 \* len(X))

testing\_samples = len(X) - num\_training

X\_train, y\_train = X[:training\_samples], y[:training\_samples]

X\_test, y\_test = X[training\_samples:], y[training\_samples:]

Now, we need to create a linear regressor object.

reg\_linear\_mul = linear\_model.LinearRegression()

Train the object with the training samples.

reg\_linear\_mul.fit(X\_train, y\_train)

Now, at last we need to do the prediction with the testing data.

y\_test\_pred = reg\_linear\_mul.predict(X\_test)

print("Performance of Linear regressor:")

print("Mean absolute error =", round(sm.mean\_absolute\_error(y\_test, y\_test\_pred), 2))

print("Mean squared error =", round(sm.mean\_squared\_error(y\_test, y\_test\_pred), 2))

print("Median absolute error =", round(sm.median\_absolute\_error(y\_test, y\_test\_pred), 2))

print("Explain variance score =", round(sm.explained\_variance\_score(y\_test, y\_test\_pred), 2))

print("R2 score =", round(sm.r2\_score(y\_test, y\_test\_pred), 2))

### Output

Performance of Linear Regressor −

Mean absolute error = 0.6

Mean squared error = 0.65

Median absolute error = 0.41

Explain variance score = 0.34

R2 score = 0.33

Now, we will create a polynomial of degree 10 and train the regressor. We will provide the sample data point.

polynomial = PolynomialFeatures(degree = 10)

X\_train\_transformed = polynomial.fit\_transform(X\_train)

datapoint = [[2.23, 1.35, 1.12]]

poly\_datapoint = polynomial.fit\_transform(datapoint)

poly\_linear\_model = linear\_model.LinearRegression()

poly\_linear\_model.fit(X\_train\_transformed, y\_train)

print("\nLinear regression:\n", reg\_linear\_mul.predict(datapoint))

print("\nPolynomial regression:\n", poly\_linear\_model.predict(poly\_datapoint))

### Output

Linear regression −

[2.40170462]

Polynomial regression −

[1.8697225]

In the above code, we have used this small data. If you want a big dataset then, you can use sklearn.dataset to import a bigger dataset.

2,4.8,1.2,3.22.9,4.7,1.5,3.62.5,5,2.8,23.2,5.5,3.5,2.16,5,

2,3.27.6,4,1.2,3.23.2,0.9,2.3,1.42.9,1.9,2.3,1.22.4,3.5,

2.8,3.60.5,3.4,1.8,2.91,4,3,2.50.9,5.9,5.6,0.81.2,2.58,

3.45,1.233.2,5.6,2,3.25.1,1.5,1.2,1.34.5,1.2,4.1,2.32.3,

6.3,2.5,3.22.1,2.8,1.2,3.6

**Logic Programming**

In this chapter, we will focus logic programming and how it helps in Artificial Intelligence.

We already know that logic is the study of principles of correct reasoning or in simple words it is the study of what comes after what. For example, if two statements are true then we can infer any third statement from it.

### Concept

Logic Programming is the combination of two words, logic and programming. Logic Programming is a programming paradigm in which the problems are expressed as facts and rules by program statements but within a system of formal logic. Just like other programming paradigms like object oriented, functional, declarative, and procedural, etc., it is also a particular way to approach programming.

## **How to Solve Problems with Logic Programming**

Logic Programming uses facts and rules for solving the problem. That is why they are called the building blocks of Logic Programming. A goal needs to be specified for every program in logic programming. To understand how a problem can be solved in logic programming, we need to know about the building blocks − Facts and Rules −

### Facts

Actually, every logic program needs facts to work with so that it can achieve the given goal. Facts basically are true statements about the program and data. For example, Delhi is the capital of India.

### Rules

Actually, rules are the constraints which allow us to make conclusions about the problem domain. Rules basically written as logical clauses to express various facts. For example, if we are building any game then all the rules must be defined.

Rules are very important to solve any problem in Logic Programming. Rules are basically logical conclusion which can express the facts. Following is the syntax of rule −

A∶− B1,B2,...,Bn.

Here, A is the head and B1, B2, ... Bn is the body.

For example − ancestor(X,Y) :- father(X,Y).

ancestor(X,Z) :- father(X,Y), ancestor(Y,Z).

This can be read as, for every X and Y, if X is the father of Y and Y is an ancestor of Z, X is the ancestor of Z. For every X and Y, X is the ancestor of Z, if X is the father of Y and Y is an ancestor of Z.

## **Installing Useful Packages**

For starting logic programming in Python, we need to install the following two packages −

### Kanren

It provides us a way to simplify the way we made code for business logic. It lets us express the logic in terms of rules and facts. The following command will help you install kanren −

pip install kanren

### SymPy

SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible. The following command will help you install SymPy −

pip install sympy

## **Examples of Logic Programming**

Followings are some examples which can be solved by logic programming −

### Matching mathematical expressions

Actually we can find the unknown values by using logic programming in a very effective way. The following Python code will help you match a mathematical expression −

Consider importing the following packages first −

from kanren import run, var, fact

from kanren.assoccomm import eq\_assoccomm as eq

from kanren.assoccomm import commutative, associative

We need to define the mathematical operations which we are going to use −

add = 'add'

mul = 'mul'

Both addition and multiplication are communicative processes. Hence, we need to specify it and this can be done as follows −

fact(commutative, mul)

fact(commutative, add)

fact(associative, mul)

fact(associative, add)

It is compulsory to define variables; this can be done as follows −

a, b = var('a'), var('b')

We need to match the expression with the original pattern. We have the following original pattern, which is basically (5+a)\*b −

Original\_pattern = (mul, (add, 5, a), b)

We have the following two expressions to match with the original pattern −

exp1 = (mul, 2, (add, 3, 1))

exp2 = (add,5,(mul,8,1))

Output can be printed with the following command −

print(run(0, (a,b), eq(original\_pattern, exp1)))

print(run(0, (a,b), eq(original\_pattern, exp2)))

After running this code, we will get the following output −

((3,2))

()

The first output represents the values for **a** and **b**. The first expression matched the original pattern and returned the values for **a** and **b** but the second expression did not match the original pattern hence nothing has been returned.

## **Checking for Prime Numbers**

With the help of logic programming, we can find the prime numbers from a list of numbers and can also generate prime numbers. The Python code given below will find the prime number from a list of numbers and will also generate the first 10 prime numbers.

Let us first consider importing the following packages −

from kanren import isvar, run, membero

from kanren.core import success, fail, goaleval, condeseq, eq, var

from sympy.ntheory.generate import prime, isprime

import itertools as it

Now, we will define a function called prime\_check which will check the prime numbers based on the given numbers as data.

def prime\_check(x):

if isvar(x):

return condeseq([(eq,x,p)] for p in map(prime, it.count(1)))

else:

return success if isprime(x) else fail

Now, we need to declare a variable which will be used −

x = var()

print((set(run(0,x,(membero,x,(12,14,15,19,20,21,22,23,29,30,41,44,52,62,65,85)),

(prime\_check,x)))))

print((run(10,x,prime\_check(x))))

The output of the above code will be as follows −

{19, 23, 29, 41}

(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)

## **Solving Puzzles**

Logic programming can be used to solve many problems like 8-puzzles, Zebra puzzle, Sudoku, N-queen, etc. Here we are taking an example of a variant of Zebra puzzle which is as follows −

There are five houses.

The English man lives in the red house.

The Swede has a dog.

The Dane drinks tea.

The green house is immediately to the left of the white house.

They drink coffee in the green house.

The man who smokes Pall Mall has birds.

In the yellow house they smoke Dunhill.

In the middle house they drink milk.

The Norwegian lives in the first house.

The man who smokes Blend lives in the house next to the house with cats.

In a house next to the house where they have a horse, they smoke Dunhill.

The man who smokes Blue Master drinks beer.

The German smokes Prince.

The Norwegian lives next to the blue house.

They drink water in a house next to the house where they smoke Blend.

We are solving it for the question **who owns zebra** with the help of Python.

Let us import the necessary packages −

from kanren import \*

from kanren.core import lall

import time

Now, we need to define two functions − **left()** and **next()** to check whose house is left or next to who’s house −

def left(q, p, list):

return membero((q,p), zip(list, list[1:]))

def next(q, p, list):

return conde([left(q, p, list)], [left(p, q, list)])

Now, we will declare a variable house as follows −

houses = var()

We need to define the rules with the help of lall package as follows.

There are 5 houses −

rules\_zebraproblem = lall(

(eq, (var(), var(), var(), var(), var()), houses),

(membero,('Englishman', var(), var(), var(), 'red'), houses),

(membero,('Swede', var(), var(), 'dog', var()), houses),

(membero,('Dane', var(), 'tea', var(), var()), houses),

(left,(var(), var(), var(), var(), 'green'),

(var(), var(), var(), var(), 'white'), houses),

(membero,(var(), var(), 'coffee', var(), 'green'), houses),

(membero,(var(), 'Pall Mall', var(), 'birds', var()), houses),

(membero,(var(), 'Dunhill', var(), var(), 'yellow'), houses),

(eq,(var(), var(), (var(), var(), 'milk', var(), var()), var(), var()), houses),

(eq,(('Norwegian', var(), var(), var(), var()), var(), var(), var(), var()), houses),

(next,(var(), 'Blend', var(), var(), var()),

(var(), var(), var(), 'cats', var()), houses),

(next,(var(), 'Dunhill', var(), var(), var()),

(var(), var(), var(), 'horse', var()), houses),

(membero,(var(), 'Blue Master', 'beer', var(), var()), houses),

(membero,('German', 'Prince', var(), var(), var()), houses),

(next,('Norwegian', var(), var(), var(), var()),

(var(), var(), var(), var(), 'blue'), houses),

(next,(var(), 'Blend', var(), var(), var()),

(var(), var(), 'water', var(), var()), houses),

(membero,(var(), var(), var(), 'zebra', var()), houses)

)

Now, run the solver with the preceding constraints −

solutions = run(0, houses, rules\_zebraproblem)

With the help of the following code, we can extract the output from the solver −

output\_zebra = [house for house in solutions[0] if 'zebra' in house][0][0]

The following code will help print the solution −

print ('\n'+ output\_zebra + 'owns zebra.')

The output of the above code would be as follows −

German owns zebra.

**Unsupervised machine learning Algorithms**

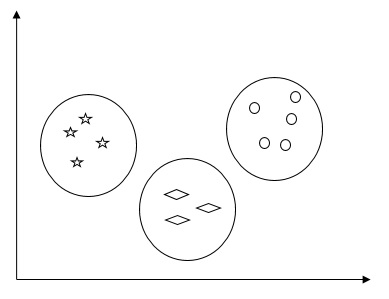
Unsupervised machine learning algorithms do not have any supervisor to provide any sort of guidance. That is why they are closely aligned with what some call true artificial intelligence.

In unsupervised learning, there would be no correct answer and no teacher for the guidance. Algorithms need to discover the interesting pattern in data for learning.

## **What is Clustering?**

Basically, it is a type of unsupervised learning method and a common technique for statistical data analysis used in many fields. Clustering mainly is a task of dividing the set of observations into subsets, called clusters, in such a way that observations in the same cluster are similar in one sense and they are dissimilar to the observations in other clusters. In simple words, we can say that the main goal of clustering is to group the data on the basis of similarity and dissimilarity.

For example, the following diagram shows similar kind of data in different clusters −



## **Algorithms for Clustering the Data**

Following are a few common algorithms for clustering the data −

### K-Means algorithm

K-means clustering algorithm is one of the well-known algorithms for clustering the data. We need to assume that the numbers of clusters are already known. This is also called flat clustering. It is an iterative clustering algorithm. The steps given below need to be followed for this algorithm −

**Step 1** − We need to specify the desired number of K subgroups.

**Step 2** − Fix the number of clusters and randomly assign each data point to a cluster. Or in other words we need to classify our data based on the number of clusters.

In this step, cluster centroids should be computed.

As this is an iterative algorithm, we need to update the locations of K centroids with every iteration until we find the global optima or in other words the centroids reach at their optimal locations.

The following code will help in implementing K-means clustering algorithm in Python. We are going to use the Scikit-learn module.

Let us import the necessary packages −

import matplotlib.pyplot as plt

import seaborn as sns; sns.set()

import numpy as np

from sklearn.cluster import KMeans

The following line of code will help in generating the two-dimensional dataset, containing four blobs, by using **make\_blob** from the **sklearn.dataset** package.

from sklearn.datasets.samples\_generator import make\_blobs

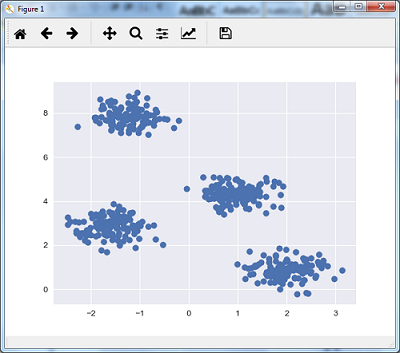
X, y\_true = make\_blobs(n\_samples = 500, centers = 4,

cluster\_std = 0.40, random\_state = 0)

We can visualize the dataset by using the following code −

plt.scatter(X[:, 0], X[:, 1], s = 50);

plt.show()



Here, we are initializing kmeans to be the KMeans algorithm, with the required parameter of how many clusters (n\_clusters).

kmeans = KMeans(n\_clusters = 4)

We need to train the K-means model with the input data.

kmeans.fit(X)

y\_kmeans = kmeans.predict(X)

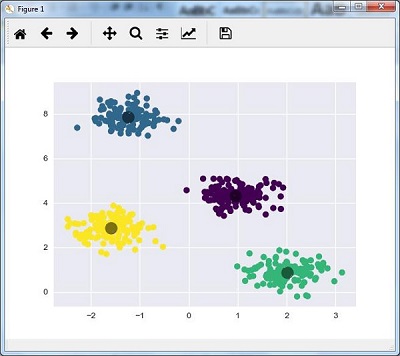
plt.scatter(X[:, 0], X[:, 1], c = y\_kmeans, s = 50, cmap = 'viridis')

centers = kmeans.cluster\_centers\_

The code given below will help us plot and visualize the machine's findings based on our data, and the fitment according to the number of clusters that are to be found.

plt.scatter(centers[:, 0], centers[:, 1], c = 'black', s = 200, alpha = 0.5);

plt.show()



### Mean Shift Algorithm

It is another popular and powerful clustering algorithm used in unsupervised learning. It does not make any assumptions hence it is a non-parametric algorithm. It is also called hierarchical clustering or mean shift cluster analysis. Followings would be the basic steps of this algorithm −

* First of all, we need to start with the data points assigned to a cluster of their own.
* Now, it computes the centroids and update the location of new centroids.
* By repeating this process, we move closer the peak of cluster i.e. towards the region of higher density.
* This algorithm stops at the stage where centroids do not move anymore.

With the help of following code we are implementing Mean Shift clustering algorithm in Python. We are going to use Scikit-learn module.

Let us import the necessary packages −

import numpy as np

from sklearn.cluster import MeanShift

import matplotlib.pyplot as plt

from matplotlib import style

style.use("ggplot")

The following code will help in generating the two-dimensional dataset, containing four blobs, by using **make\_blob** from the **sklearn.dataset** package.

from sklearn.datasets.samples\_generator import make\_blobs

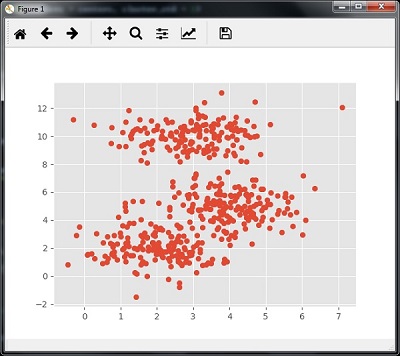
We can visualize the dataset with the following code

centers = [[2,2],[4,5],[3,10]]

X, \_ = make\_blobs(n\_samples = 500, centers = centers, cluster\_std = 1)

plt.scatter(X[:,0],X[:,1])

plt.show()



Now, we need to train the Mean Shift cluster model with the input data.

ms = MeanShift()

ms.fit(X)

labels = ms.labels\_

cluster\_centers = ms.cluster\_centers\_

The following code will print the cluster centers and the expected number of cluster as per the input data −

print(cluster\_centers)

n\_clusters\_ = len(np.unique(labels))

print("Estimated clusters:", n\_clusters\_)

[[ 3.23005036 3.84771893]

[ 3.02057451 9.88928991]]

Estimated clusters: 2

The code given below will help plot and visualize the machine's findings based on our data, and the fitment according to the number of clusters that are to be found.

colors = 10\*['r.','g.','b.','c.','k.','y.','m.']

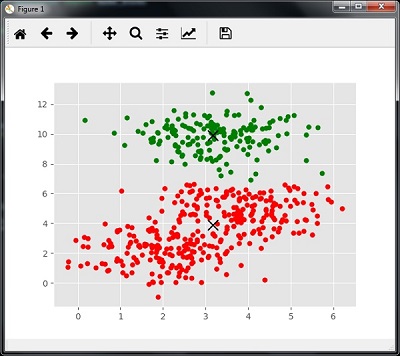
for i in range(len(X)):

plt.plot(X[i][0], X[i][1], colors[labels[i]], markersize = 10)

plt.scatter(cluster\_centers[:,0],cluster\_centers[:,1],

marker = "x",color = 'k', s = 150, linewidths = 5, zorder = 10)

plt.show()



## **Measuring the Clustering Performance**

The real world data is not naturally organized into number of distinctive clusters. Due to this reason, it is not easy to visualize and draw inferences. That is why we need to measure the clustering performance as well as its quality. It can be done with the help of silhouette analysis.

### Silhouette Analysis

This method can be used to check the quality of clustering by measuring the distance between the clusters. Basically, it provides a way to assess the parameters like number of clusters by giving a silhouette score. This score is a metric that measures how close each point in one cluster is to the points in the neighboring clusters.

### Analysis of silhouette score

The score has a range of [-1, 1]. Following is the analysis of this score −

* **Score of +1** − Score near +1 indicates that the sample is far away from the neighboring cluster.
* **Score of 0** − Score 0 indicates that the sample is on or very close to the decision boundary between two neighboring clusters.
* **Score of -1** − Negative score indicates that the samples have been assigned to the wrong clusters.

## **Calculating Silhouette Score**

In this section, we will learn how to calculate the silhouette score.

Silhouette score can be calculated by using the following formula −

$$silhouette score = \frac{\left ( p-q \right )}{max\left ( p,q \right )}$$

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

For finding the optimal number of clusters, we need to run the clustering algorithm again by importing the **metrics** module from the **sklearn** package. In the following example, we will run the K-means clustering algorithm to find the optimal number of clusters −

Import the necessary packages as shown −

import matplotlib.pyplot as plt

import seaborn as sns; sns.set()

import numpy as np

from sklearn.cluster import KMeans

With the help of the following code, we will generate the two-dimensional dataset, containing four blobs, by using **make\_blob** from the **sklearn.dataset** package.

from sklearn.datasets.samples\_generator import make\_blobs

X, y\_true = make\_blobs(n\_samples = 500, centers = 4, cluster\_std = 0.40, random\_state = 0)

Initialize the variables as shown −

scores = []

values = np.arange(2, 10)

We need to iterate the K-means model through all the values and also need to train it with the input data.

for num\_clusters in values:

kmeans = KMeans(init = 'k-means++', n\_clusters = num\_clusters, n\_init = 10)

kmeans.fit(X)

Now, estimate the silhouette score for the current clustering model using the Euclidean distance metric −

score = metrics.silhouette\_score(X, kmeans.labels\_,

metric = 'euclidean', sample\_size = len(X))

The following line of code will help in displaying the number of clusters as well as Silhouette score.

print("\nNumber of clusters =", num\_clusters)

print("Silhouette score =", score)

scores.append(score)

You will receive the following output −

Number of clusters = 9

Silhouette score = 0.340391138371

num\_clusters = np.argmax(scores) + values[0]

print('\nOptimal number of clusters =', num\_clusters)

Now, the output for optimal number of clusters would be as follows −

Optimal number of clusters = 2

## **Finding Nearest Neighbors**

If we want to build recommender systems such as a movie recommender system then we need to understand the concept of finding the nearest neighbors. It is because the recommender system utilizes the concept of nearest neighbors.

The **concept of finding nearest neighbors** may be defined as the process of finding the closest point to the input point from the given dataset. The main use of this KNN)K-nearest neighbors) algorithm is to build classification systems that classify a data point on the proximity of the input data point to various classes.

The Python code given below helps in finding the K-nearest neighbors of a given data set −

Import the necessary packages as shown below. Here, we are using the **NearestNeighbors** module from the **sklearn** package

import numpy as np

import matplotlib.pyplot as plt

from sklearn.neighbors import NearestNeighbors

Let us now define the input data −

A = np.array([[3.1, 2.3], [2.3, 4.2], [3.9, 3.5], [3.7, 6.4], [4.8, 1.9],

[8.3, 3.1], [5.2, 7.5], [4.8, 4.7], [3.5, 5.1], [4.4, 2.9],])

Now, we need to define the nearest neighbors −

k = 3

We also need to give the test data from which the nearest neighbors is to be found −

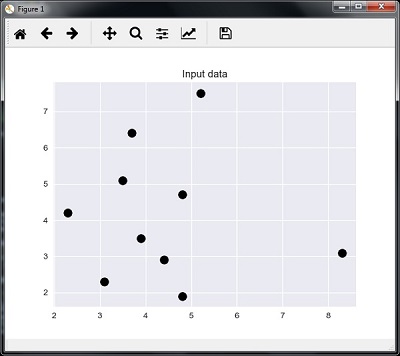
test\_data = [3.3, 2.9]

The following code can visualize and plot the input data defined by us −

plt.figure()

plt.title('Input data')

plt.scatter(A[:,0], A[:,1], marker = 'o', s = 100, color = 'black')



Now, we need to build the K Nearest Neighbor. The object also needs to be trained

knn\_model = NearestNeighbors(n\_neighbors = k, algorithm = 'auto').fit(X)

distances, indices = knn\_model.kneighbors([test\_data])

Now, we can print the K nearest neighbors as follows

print("\nK Nearest Neighbors:")

for rank, index in enumerate(indices[0][:k], start = 1):

print(str(rank) + " is", A[index])

We can visualize the nearest neighbors along with the test data point

plt.figure()

plt.title('Nearest neighbors')

plt.scatter(A[:, 0], X[:, 1], marker = 'o', s = 100, color = 'k')

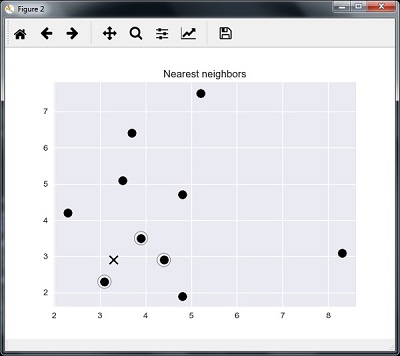
plt.scatter(A[indices][0][:][:, 0], A[indices][0][:][:, 1],

marker = 'o', s = 250, color = 'k', facecolors = 'none')

plt.scatter(test\_data[0], test\_data[1],

marker = 'x', s = 100, color = 'k')

plt.show()



### Output

**K Nearest Neighbors**

1 is [ 3.1 2.3]

2 is [ 3.9 3.5]

3 is [ 4.4 2.9]

## **K-Nearest Neighbors Classifier**

A K-Nearest Neighbors (KNN) classifier is a classification model that uses the nearest neighbors algorithm to classify a given data point. We have implemented the KNN algorithm in the last section, now we are going to build a KNN classifier using that algorithm.

### Concept of KNN Classifier

The basic concept of K-nearest neighbor classification is to find a predefined number, i.e., the 'k' − of training samples closest in distance to a new sample, which has to be classified. New samples will get their label from the neighbors itself. The KNN classifiers have a fixed user defined constant for the number of neighbors which have to be determined. For the distance, standard Euclidean distance is the most common choice. The KNN Classifier works directly on the learned samples rather than creating the rules for learning. The KNN algorithm is among the simplest of all machine learning algorithms. It has been quite successful in a large number of classification and regression problems, for example, character recognition or image analysis.

**Example**

We are building a KNN classifier to recognize digits. For this, we will use the MNIST dataset. We will write this code in the Jupyter Notebook.

Import the necessary packages as shown below.

Here we are using the **KNeighborsClassifier** module from the **sklearn.neighbors** package −

from sklearn.datasets import \*

import pandas as pd

%matplotlib inline

from sklearn.neighbors import KNeighborsClassifier

import matplotlib.pyplot as plt

import numpy as np

The following code will display the image of digit to verify what image we have to test −

def Image\_display(i):

plt.imshow(digit['images'][i],cmap = 'Greys\_r')

plt.show()

Now, we need to load the MNIST dataset. Actually there are total 1797 images but we are using the first 1600 images as training sample and the remaining 197 would be kept for testing purpose.

digit = load\_digits()

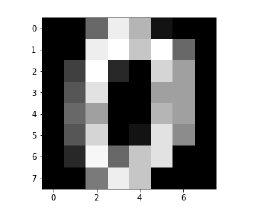
digit\_d = pd.DataFrame(digit['data'][0:1600])

Now, on displaying the images we can see the output as follows −

Image\_display(0)

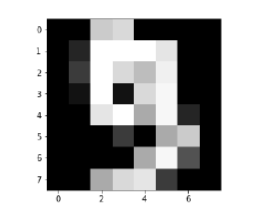
### Image\_display(0)

Image of 0 is displayed as follows −



### Image\_display(9)

Image of 9 is displayed as follows −



### digit.keys()

Now, we need to create the training and testing data set and supply testing data set to the KNN classifiers.

train\_x = digit['data'][:1600]

train\_y = digit['target'][:1600]

KNN = KNeighborsClassifier(20)

KNN.fit(train\_x,train\_y)

The following output will create the K nearest neighbor classifier constructor −

KNeighborsClassifier(algorithm = 'auto', leaf\_size = 30, metric = 'minkowski',

metric\_params = None, n\_jobs = 1, n\_neighbors = 20, p = 2,

weights = 'uniform')

We need to create the testing sample by providing any arbitrary number greater than 1600, which were the training samples.

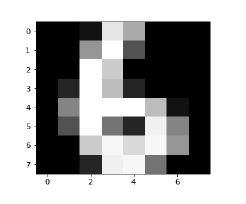
test = np.array(digit['data'][1725])

test1 = test.reshape(1,-1)

Image\_display(1725)

### Image\_display(6)

Image of 6 is displayed as follows −



Now we will predict the test data as follows −

KNN.predict(test1)

The above code will generate the following output −

array([6])

Now, consider the following −

digit['target\_names']

The above code will generate the following output −

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

**Natural Language Processing with AI**

Natural Language Processing (NLP) refers to AI method of communicating with intelligent systems using a natural language such as English.

Processing of Natural Language is required when you want an intelligent system like robot to perform as per your instructions, when you want to hear decision from a dialogue based clinical expert system, etc.

The field of NLP involves making computers erform useful tasks with the natural languages humans use. The input and output of an NLP system can be −

* Speech
* Written Text

## **Components of NLP**

In this section, we will learn about the different components of NLP. There are two components of NLP. The components are described below −

### Natural Language Understanding (NLU)

It involves the following tasks −

* Mapping the given input in natural language into useful representations.
* Analyzing different aspects of the language.

### Natural Language Generation (NLG)

It is the process of producing meaningful phrases and sentences in the form of natural language from some internal representation. It involves −

* **Text planning** − This includes retrieving the relevant content from the knowledge base.
* **Sentence planning** − This includes choosing the required words, forming meaningful phrases, setting tone of the sentence.
* **Text Realization** − This is mapping sentence plan into sentence structure.

## **Difficulties in NLU**

The NLU is very rich in form and structure; however, it is ambiguous. There can be different levels of ambiguity −

### Lexical ambiguity

It is at a very primitive level such as the word-level. For example, treating the word “board” as noun or verb?

### Syntax level ambiguity

A sentence can be parsed in different ways. For example, “He lifted the beetle with red cap.” − Did he use cap to lift the beetle or he lifted a beetle that had red cap?

### Referential ambiguity

Referring to something using pronouns. For example, Rima went to Gauri. She said, “I am tired.” − Exactly who is tired?

## **NLP Terminology**

Let us now see a few important terms in the NLP terminology.

* **Phonology** − It is study of organizing sound systematically.
* **Morphology** − It is a study of construction of words from primitive meaningful units.
* **Morpheme** − It is a primitive unit of meaning in a language.
* **Syntax** − It refers to arranging words to make a sentence. It also involves determining the structural role of words in the sentence and in phrases.
* **Semantics** − It is concerned with the meaning of words and how to combine words into meaningful phrases and sentences.
* **Pragmatics** − It deals with using and understanding sentences in different situations and how the interpretation of the sentence is affected.
* **Discourse** − It deals with how the immediately preceding sentence can affect the interpretation of the next sentence.
* **World Knowledge** − It includes the general knowledge about the world.

## **Steps in NLP**

This section shows the different steps in NLP.

### Lexical Analysis

It involves identifying and analyzing the structure of words. Lexicon of a language means the collection of words and phrases in a language. Lexical analysis is dividing the whole chunk of txt into paragraphs, sentences, and words.

### Syntactic Analysis (Parsing)

It involves analysis of words in the sentence for grammar and arranging words in a manner that shows the relationship among the words. The sentence such as “The school goes to boy” is rejected by English syntactic analyzer.

### Semantic Analysis

It draws the exact meaning or the dictionary meaning from the text. The text is checked for meaningfulness. It is done by mapping syntactic structures and objects in the task domain. The semantic analyzer disregards sentence such as “hot ice-cream”.

### Discourse Integration

The meaning of any sentence depends upon the meaning of the sentence just before it. In addition, it also brings about the meaning of immediately succeeding sentence.

### Pragmatic Analysis

During this, what was said is re-interpreted on what it actually meant. It involves deriving those aspects of language which require real world knowIn this chapter, we will learn how to get started with the Natural Language Toolkit Package.

### Prerequisite

If we want to build applications with Natural Language processing then the change in context makes it most difficult. The context factor influences how the machine understands a particular sentence. Hence, we need to develop Natural language applications by using machine learning approaches so that machine can also understand the way a human can understand the context.

To build such applications we will use the Python package called NLTK (Natural Language Toolkit Package).

## **Importing NLTK**

We need to install NLTK before using it. It can be installed with the help of the following command −

pip install nltk

To build a conda package for NLTK, use the following command −

conda install -c anaconda nltk

Now after installing the NLTK package, we need to import it through the python command prompt. We can import it by writing the following command on the Python command prompt −

>>> import nltk

## **Downloading NLTK’s Data**

Now after importing NLTK, we need to download the required data. It can be done with the help of the following command on the Python command prompt −

>>> nltk.download()

## **Installing Other Necessary Packages**

For building natural language processing applications by using NLTK, we need to install the necessary packages. The packages are as follows −

### gensim

It is a robust semantic modeling library that is useful for many applications. We can install it by executing the following command −

pip install gensim

### pattern

It is used to make **gensim** package work properly. We can install it by executing the following command

pip install pattern

## **Concept of Tokenization, Stemming, and Lemmatization**

In this section, we will understand what is tokenization, stemming, and lemmatization.

### Tokenization

It may be defined as the process of breaking the given text i.e. the character sequence into smaller units called tokens. The tokens may be the words, numbers or punctuation marks. It is also called word segmentation. Following is a simple example of tokenization −

**Input** − Mango, banana, pineapple and apple all are fruits.

**Output** −Tokenization

The process of breaking the given text can be done with the help of locating the word boundaries. The ending of a word and the beginning of a new word are called word boundaries. The writing system and the typographical structure of the words influence the boundaries.

In the Python NLTK module, we have different packages related to tokenization which we can use to divide the text into tokens as per our requirements. Some of the packages are as follows −

### sent\_tokenize package

As the name suggest, this package will divide the input text into sentences. We can import this package with the help of the following Python code −

from nltk.tokenize import sent\_tokenize

### word\_tokenize package

This package divides the input text into words. We can import this package with the help of the following Python code −

from nltk.tokenize import word\_tokenize

### WordPunctTokenizer package

This package divides the input text into words as well as the punctuation marks. We can import this package with the help of the following Python code −

from nltk.tokenize import WordPuncttokenizer

### Stemming

While working with words, we come across a lot of variations due to grammatical reasons. The concept of variations here means that we have to deal with different forms of the same words like ***democracy, democratic,*** and ***democratization***. It is very necessary for machines to understand that these different words have the same base form. In this way, it would be useful to extract the base forms of the words while we are analyzing the text.

We can achieve this by stemming. In this way, we can say that stemming is the heuristic process of extracting the base forms of the words by chopping off the ends of words.

In the Python NLTK module, we have different packages related to stemming. These packages can be used to get the base forms of word. These packages use algorithms. Some of the packages are as follows −

### PorterStemmer package

This Python package uses the Porter’s algorithm to extract the base form. We can import this package with the help of the following Python code −

from nltk.stem.porter import PorterStemmer

For example, if we will give the word **‘writing’** as the input to this stemmer them we will get the word **‘write’** after stemming.

### LancasterStemmer package

This Python package will use the Lancaster’s algorithm to extract the base form. We can import this package with the help of the following Python code −

from nltk.stem.lancaster import LancasterStemmer

For example, if we will give the word **‘writing’** as the input to this stemmer them we will get the word **‘write’** after stemming.

### SnowballStemmer package

This Python package will use the snowball’s algorithm to extract the base form. We can import this package with the help of the following Python code −

from nltk.stem.snowball import SnowballStemmer

For example, if we will give the word **‘writing’** as the input to this stemmer them we will get the word **‘write’** after stemming.

All of these algorithms have different level of strictness. If we compare these three stemmers then the Porter stemmers is the least strict and Lancaster is the strictest. Snowball stemmer is good to use in terms of speed as well as strictness.

### Lemmatization

We can also extract the base form of words by lemmatization. It basically does this task with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only. This kind of base form of any word is called lemma.

The main difference between stemming and lemmatization is the use of vocabulary and morphological analysis of the words. Another difference is that stemming most commonly collapses derivationally related words whereas lemmatization commonly only collapses the different inflectional forms of a lemma. For example, if we provide the word saw as the input word then stemming might return the word ‘s’ but lemmatization would attempt to return the word either see or saw depending on whether the use of the token was a verb or a noun.

In the Python NLTK module, we have the following package related to lemmatization process which we can use to get the base forms of word −

### WordNetLemmatizer package

This Python package will extract the base form of the word depending upon whether it is used as a noun or as a verb. We can import this package with the help of the following Python code −

from nltk.stem import WordNetLemmatizer

## **Chunking: Dividing Data into Chunks**

It is one of the important processes in natural language processing. The main job of chunking is to identify the parts of speech and short phrases like noun phrases. We have already studied the process of tokenization, the creation of tokens. Chunking basically is the labeling of those tokens. In other words, chunking will show us the structure of the sentence.

In the following section, we will learn about the different types of Chunking.

## **Types of chunking**

There are two types of chunking. The types are as follows −

### Chunking up

In this process of chunking, the object, things, etc. move towards being more general and the language gets more abstract. There are more chances of agreement. In this process, we zoom out. For example, if we will chunk up the question that “for what purpose cars are”? We may get the answer “transport”.

### Chunking down

In this process of chunking, the object, things, etc. move towards being more specific and the language gets more penetrated. The deeper structure would be examined in chunking down. In this process, we zoom in. For example, if we chunk down the question “Tell specifically about a car”? We will get smaller pieces of information about the car.

**Example**

In this example, we will do Noun-Phrase chunking, a category of chunking which will find the noun phrases chunks in the sentence, by using the NLTK module in Python −

**Follow these steps in python for implementing noun phrase chunking −**

**Step 1** − In this step, we need to define the grammar for chunking. It would consist of the rules which we need to follow.

**Step 2** − In this step, we need to create a chunk parser. It would parse the grammar and give the output.

**Step 3** − In this last step, the output is produced in a tree format.

Let us import the necessary NLTK package as follows −

import nltk

Now, we need to define the sentence. Here, DT means the determinant, VBP means the verb, JJ means the adjective, IN means the preposition and NN means the noun.

sentence=[("a","DT"),("clever","JJ"),("fox","NN"),("was","VBP"),

("jumping","VBP"),("over","IN"),("the","DT"),("wall","NN")]

Now, we need to give the grammar. Here, we will give the grammar in the form of regular expression.

grammar = "NP:{<DT>?<JJ>\*<NN>}"

We need to define a parser which will parse the grammar.

parser\_chunking = nltk.RegexpParser(grammar)

The parser parses the sentence as follows −

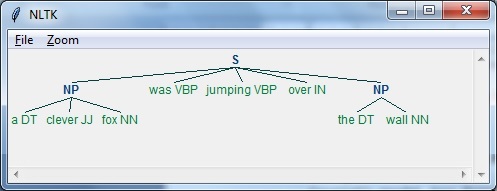
parser\_chunking.parse(sentence)

Next, we need to get the output. The output is generated in the simple variable called **output\_chunk**.

Output\_chunk = parser\_chunking.parse(sentence)

Upon execution of the following code, we can draw our output in the form of a tree.

output.draw()



## **Bag of Word (BoW) Model**

Bag of Word (BoW), a model in natural language processing, is basically used to extract the features from text so that the text can be used in modeling such that in machine learning algorithms.

Now the question arises that why we need to extract the features from text. It is because the machine learning algorithms cannot work with raw data and they need numeric data so that they can extract meaningful information out of it. The conversion of text data into numeric data is called feature extraction or feature encoding.

### How it works

This is very simple approach for extracting the features from text. Suppose we have a text document and we want to convert it into numeric data or say want to extract the features out of it then first of all this model extracts a vocabulary from all the words in the document. Then by using a document term matrix, it will build a model. In this way, BoW represents the document as a bag of words only. Any information about the order or structure of words in the document is discarded.

### Concept of document term matrix

The BoW algorithm builds a model by using the document term matrix. As the name suggests, the document term matrix is the matrix of various word counts that occur in the document. With the help of this matrix, the text document can be represented as a weighted combination of various words. By setting the threshold and choosing the words that are more meaningful, we can build a histogram of all the words in the documents that can be used as a feature vector. Following is an example to understand the concept of document term matrix −

**Example**

Suppose we have the following two sentences −

* **Sentence 1** − We are using the Bag of Words model.
* **Sentence 2** − Bag of Words model is used for extracting the features.

Now, by considering these two sentences, we have the following 13 distinct words −

* we
* are
* using
* the
* bag
* of
* words
* model
* is
* used
* for
* extracting
* features

Now, we need to build a histogram for each sentence by using the word count in each sentence −

* **Sentence 1** − [1,1,1,1,1,1,1,1,0,0,0,0,0]
* **Sentence 2** − [0,0,0,1,1,1,1,1,1,1,1,1,1]

In this way, we have the feature vectors that have been extracted. Each feature vector is 13-dimensional because we have 13 distinct words.

## **Concept of the Statistics**

The concept of the statistics is called TermFrequency-Inverse Document Frequency (tf-idf). Every word is important in the document. The statistics help us nderstand the importance of every word.

### Term Frequency(tf)

It is the measure of how frequently each word appears in a document. It can be obtained by dividing the count of each word by the total number of words in a given document.

### Inverse Document Frequency(idf)

It is the measure of how unique a word is to this document in the given set of documents. For calculating idf and formulating a distinctive feature vector, we need to reduce the weights of commonly occurring words like the and weigh up the rare words.

## **Building a Bag of Words Model in NLTK**

In this section, we will define a collection of strings by using CountVectorizer to create vectors from these sentences.

Let us import the necessary package −

from sklearn.feature\_extraction.text import CountVectorizer

Now define the set of sentences.

Sentences = ['We are using the Bag of Word model', 'Bag of Word model is

used for extracting the features.']

vectorizer\_count = CountVectorizer()

features\_text = vectorizer.fit\_transform(Sentences).todense()

**print(vectorizer.vocabulary\_)**

The above program generates the output as shown below. It shows that we have 13 distinct words in the above two sentences −

{'we': 11, 'are': 0, 'using': 10, 'the': 8, 'bag': 1, 'of': 7,

'word': 12, 'model': 6, 'is': 5, 'used': 9, 'for': 4, 'extracting': 2, 'features': 3}

These are the feature vectors (text to numeric form) which can be used for machine learning.

## **Solving Problems**

In this section, we will solve a few related problems.

### Category Prediction

In a set of documents, not only the words but the category of the words is also important; in which category of text a particular word falls. For example, we want to predict whether a given sentence belongs to the category email, news, sports, computer, etc. In the following example, we are going to use tf-idf to formulate a feature vector to find the category of documents. We will use the data from 20 newsgroup dataset of sklearn.

We need to import the necessary packages −

from sklearn.datasets import fetch\_20newsgroups

from sklearn.naive\_bayes import MultinomialNB

from sklearn.feature\_extraction.text import TfidfTransformer

from sklearn.feature\_extraction.text import CountVectorizer

Define the category map. We are using five different categories named Religion, Autos, Sports, Electronics and Space.

category\_map = {'talk.religion.misc':'Religion','rec.autos''Autos',

'rec.sport.hockey':'Hockey','sci.electronics':'Electronics', 'sci.space': 'Space'}

Create the training set −

training\_data = fetch\_20newsgroups(subset = 'train',

categories = category\_map.keys(), shuffle = True, random\_state = 5)

Build a count vectorizer and extract the term counts −

vectorizer\_count = CountVectorizer()

train\_tc = vectorizer\_count.fit\_transform(training\_data.data)

print("\nDimensions of training data:", train\_tc.shape)

The tf-idf transformer is created as follows −

tfidf = TfidfTransformer()

train\_tfidf = tfidf.fit\_transform(train\_tc)

Now, define the test data −

input\_data = [

'Discovery was a space shuttle',

'Hindu, Christian, Sikh all are religions',

'We must have to drive safely',

'Puck is a disk made of rubber',

'Television, Microwave, Refrigrated all uses electricity'

]

The above data will help us train a Multinomial Naive Bayes classifier −

classifier = MultinomialNB().fit(train\_tfidf, training\_data.target)

Transform the input data using the count vectorizer −

input\_tc = vectorizer\_count.transform(input\_data)

Now, we will transform the vectorized data using the tfidf transformer −

input\_tfidf = tfidf.transform(input\_tc)

We will predict the output categories −

predictions = classifier.predict(input\_tfidf)

The output is generated as follows −

for sent, category in zip(input\_data, predictions):

print('\nInput Data:', sent, '\n Category:', \

category\_map[training\_data.target\_names[category]])

The category predictor generates the following output −

Dimensions of training data: (2755, 39297)

Input Data: Discovery was a space shuttle

Category: Space

Input Data: Hindu, Christian, Sikh all are religions

Category: Religion

Input Data: We must have to drive safely

Category: Autos

Input Data: Puck is a disk made of rubber

Category: Hockey

Input Data: Television, Microwave, Refrigrated all uses electricity

Category: Electronics

### Gender Finder

In this problem statement, a classifier would be trained to find the gender (male or female) by providing the names. We need to use a heuristic to construct a feature vector and train the classifier. We will be using the labeled data from the scikit-learn package. Following is the Python code to build a gender finder −

Let us import the necessary packages −

import random

from nltk import NaiveBayesClassifier

from nltk.classify import accuracy as nltk\_accuracy

from nltk.corpus import names

Now we need to extract the last N letters from the input word. These letters will act as features −

def extract\_features(word, N = 2):

last\_n\_letters = word[-N:]

return {'feature': last\_n\_letters.lower()}

if \_\_name\_\_=='\_\_main\_\_':

Create the training data using labeled names (male as well as female) available in NLTK −

male\_list = [(name, 'male') for name in names.words('male.txt')]

female\_list = [(name, 'female') for name in names.words('female.txt')]

data = (male\_list + female\_list)

random.seed(5)

random.shuffle(data)

Now, test data will be created as follows −

namesInput = ['Rajesh', 'Gaurav', 'Swati', 'Shubha']

Define the number of samples used for train and test with the following code

train\_sample = int(0.8 \* len(data))

Now, we need to iterate through different lengths so that the accuracy can be compared −

for i in range(1, 6):

print('\nNumber of end letters:', i)

features = [(extract\_features(n, i), gender) for (n, gender) in data]

train\_data, test\_data = features[:train\_sample],

features[train\_sample:]

classifier = NaiveBayesClassifier.train(train\_data)

The accuracy of the classifier can be computed as follows −

accuracy\_classifier = round(100 \* nltk\_accuracy(classifier, test\_data), 2)

print('Accuracy = ' + str(accuracy\_classifier) + '%')

Now, we can predict the output −

for name in namesInput:

print(name, '==>', classifier.classify(extract\_features(name, i)))

The above program will generate the following output −

Number of end letters: 1

Accuracy = 74.7%

Rajesh -> female

Gaurav -> male

Swati -> female

Shubha -> female

Number of end letters: 2

Accuracy = 78.79%

Rajesh -> male

Gaurav -> male

Swati -> female

Shubha -> female

Number of end letters: 3

Accuracy = 77.22%

Rajesh -> male

Gaurav -> female

Swati -> female

Shubha -> female

Number of end letters: 4

Accuracy = 69.98%

Rajesh -> female

Gaurav -> female

Swati -> female

Shubha -> female

Number of end letters: 5

Accuracy = 64.63%

Rajesh -> female

Gaurav -> female

Swati -> female

Shubha -> female

In the above output, we can see that accuracy in maximum number of end letters are two and it is decreasing as the number of end letters are increasing.

## **Topic Modeling: Identifying Patterns in Text Data**

We know that generally documents are grouped into topics. Sometimes we need to identify the patterns in a text that correspond to a particular topic. The technique of doing this is called topic modeling. In other words, we can say that topic modeling is a technique to uncover abstract themes or hidden structure in the given set of documents.

We can use the topic modeling technique in the following scenarios −

### Text Classification

With the help of topic modeling, classification can be improved because it groups similar words together rather than using each word separately as a feature.

### Recommender Systems

With the help of topic modeling, we can build the recommender systems by using similarity measures.

## **Algorithms for Topic Modeling**

Topic modeling can be implemented by using algorithms. The algorithms are as follows −

### Latent Dirichlet Allocation(LDA)

This algorithm is the most popular for topic modeling. It uses the probabilistic graphical models for implementing topic modeling. We need to import gensim package in Python for using LDA slgorithm.

### Latent Semantic Analysis(LDA) or Latent Semantic Indexing(LSI)

This algorithm is based upon Linear Algebra. Basically it uses the concept of SVD (Singular Value Decomposition) on the document term matrix.

### Non-Negative Matrix Factorization (NMF)

It is also based upon Linear Algebra.

All of the above mentioned algorithms for topic modeling would have the **number of topics** as a parameter, **Document-Word Matrix** as an input and **WTM (Word Topic Matrix)** & **TDM (Topic Document Matrix)** as output.

**AI with python – Analyzing Time Series Data**

Predicting the next in a given input sequence is another important concept in machine learning. This chapter gives you a detailed explanation about analyzing time series data.

## **Introduction**

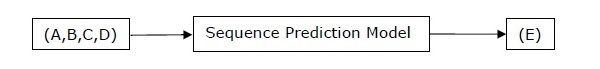
Time series data means the data that is in a series of particular time intervals. If we want to build sequence prediction in machine learning, then we have to deal with sequential data and time. Series data is an abstract of sequential data. Ordering of data is an important feature of sequential data.

### Basic Concept of Sequence Analysis or Time Series Analysis

Sequence analysis or time series analysis is to predict the next in a given input sequence based on the previously observed. The prediction can be of anything that may come next: a symbol, a number, next day weather, next term in speech etc. Sequence analysis can be very handy in applications such as stock market analysis, weather forecasting, and product recommendations.

**Example**

Consider the following example to understand sequence prediction. Here **A,B,C,D** are the given values and you have to predict the value **E** using a Sequence Prediction Model.



## **Installing Useful Packages**

For time series data analysis using Python, we need to install the following packages −

### Pandas

Pandas is an open source BSD-licensed library which provides high-performance, ease of data structure usage and data analysis tools for Python. You can install Pandas with the help of the following command −

pip install pandas

If you are using Anaconda and want to install by using the **conda** package manager, then you can use the following command −

conda install -c anaconda pandas

## **hmmlearn**

It is an open source BSD-licensed library which consists of simple algorithms and models to learn Hidden Markov Models(HMM) in Python. You can install it with the help of the following command −

pip install hmmlearn

If you are using Anaconda and want to install by using the **conda** package manager, then you can use the following command −

conda install -c omnia hmmlearn

### PyStruct

It is a structured learning and prediction library. Learning algorithms implemented in PyStruct have names such as conditional random fields(CRF), Maximum-Margin Markov Random Networks (M3N) or structural support vector machines. You can install it with the help of the following command −

pip install pystruct

### CVXOPT

It is used for convex optimization based on Python programming language. It is also a free software package. You can install it with the help of following command −

pip install cvxopt

If you are using Anaconda and want to install by using the **conda** package manager, then you can use the following command −

conda install -c anaconda cvdoxt

## **Pandas: Handling, Slicing and Extracting Statistic from Time Series Data**

Pandas is a very useful tool if you have to work with time series data. With the help of Pandas, you can perform the following −

* Create a range of dates by using the **pd.date\_range** package
* Index pandas with dates by using the **pd.Series** package
* Perform re-sampling by using the **ts.resample** package
* Change the frequency

### Example

The following example shows you handling and slicing the time series data by using Pandas. Note that here we are using the Monthly Arctic Oscillation data, which can be downloaded from [monthly.ao.index.b50.current.ascii](http://www.cpc.ncep.noaa.gov/products/precip/CWlink/daily_ao_index/monthly.ao.index.b50.current.ascii) and can be converted to text format for our use.

### Handling time series data

For handling time series data, you will have to perform the following steps −

The first step involves importing the following packages −

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

Next, define a function which will read the data from the input file, as shown in the code given below −

def read\_data(input\_file):

input\_data = np.loadtxt(input\_file, delimiter = None)

Now, convert this data to time series. For this, create the range of dates of our time series. In this example, we keep one month as frequency of data. Our file is having the data which starts from January 1950.

dates = pd.date\_range('1950-01', periods = input\_data.shape[0], freq = 'M')

In this step, we create the time series data with the help of Pandas Series, as shown below −

output = pd.Series(input\_data[:, index], index = dates)

return output

if \_\_name\_\_=='\_\_main\_\_':

Enter the path of the input file as shown here −

input\_file = "/Users/admin/AO.txt"

Now, convert the column to timeseries format, as shown here −

timeseries = read\_data(input\_file)

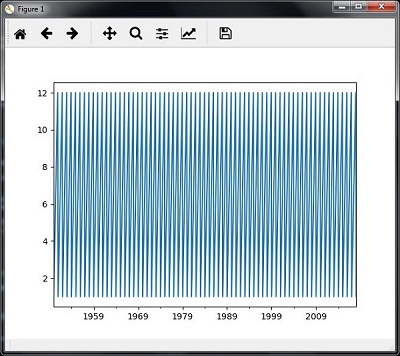
Finally, plot and visualize the data, using the commands shown −

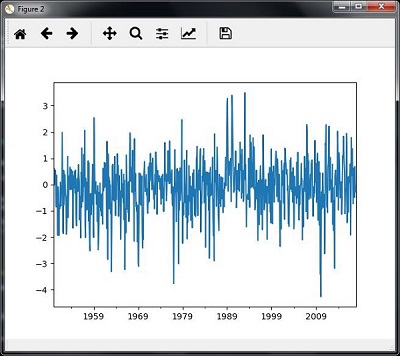
plt.figure()

timeseries.plot()

plt.show()

You will observe the plots as shown in the following images −





### Slicing time series data

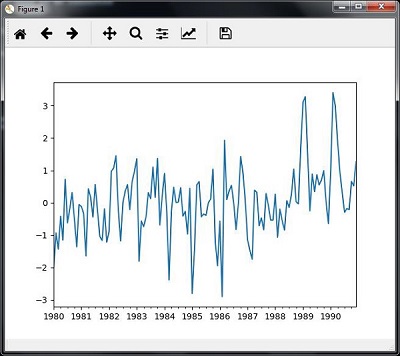
Slicing involves retrieving only some part of the time series data. As a part of the example, we are slicing the data only from 1980 to 1990. Observe the following code that performs this task −

timeseries['1980':'1990'].plot()

<matplotlib.axes.\_subplots.AxesSubplot at 0xa0e4b00>

plt.show()

When you run the code for slicing the time series data, you can observe the following graph as shown in the image here −



## **Extracting Statistic from Time Series Data**

You will have to extract some statistics from a given data, in cases where you need to draw some important conclusion. Mean, variance, correlation, maximum value, and minimum value are some of such statistics. You can use the following code if you want to extract such statistics from a given time series data −

### Mean

You can use the **mean()** function, for finding the mean, as shown here −

timeseries.mean()

Then the output that you will observe for the example discussed is −

-0.11143128165238671

### Maximum

You can use the **max()** function, for finding maximum, as shown here −

timeseries.max()

Then the output that you will observe for the example discussed is −

3.4952999999999999

### Minimum

You can use the min() function, for finding minimum, as shown here −

timeseries.min()

Then the output that you will observe for the example discussed is −

-4.2656999999999998

### Getting everything at once

If you want to calculate all statistics at a time, you can use the **describe()** function as shown here −

timeseries.describe()

Then the output that you will observe for the example discussed is −

count 817.000000

mean -0.111431

std 1.003151

min -4.265700

25% -0.649430

50% -0.042744

75% 0.475720

max 3.495300

dtype: float64

### Re-sampling

You can resample the data to a different time frequency. The two parameters for performing re-sampling are −

* Time period
* Method

### Re-sampling with mean()

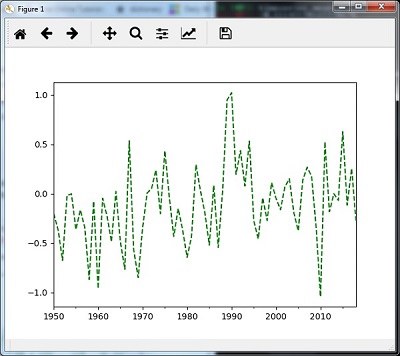
You can use the following code to resample the data with the mean()method, which is the default method −

timeseries\_mm = timeseries.resample("A").mean()

timeseries\_mm.plot(style = 'g--')

plt.show()

Then, you can observe the following graph as the output of resampling using mean() −



### Re-sampling with median()

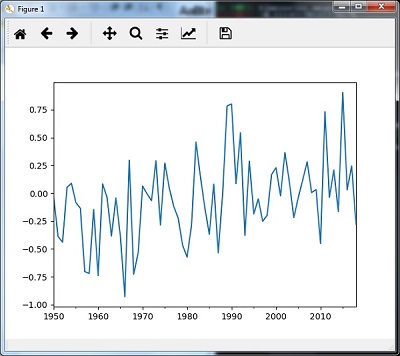
You can use the following code to resample the data using the **median()**method −

timeseries\_mm = timeseries.resample("A").median()

timeseries\_mm.plot()

plt.show()

Then, you can observe the following graph as the output of re-sampling with median() −



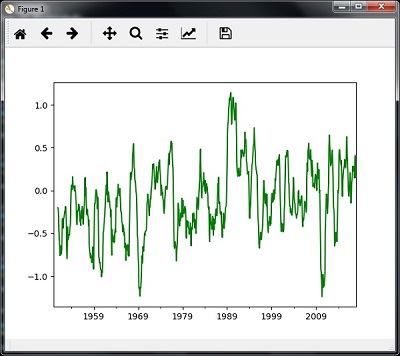
### Rolling Mean

You can use the following code to calculate the rolling (moving) mean −

timeseries.rolling(window = 12, center = False).mean().plot(style = '-g')

plt.show()

Then, you can observe the following graph as the output of the rolling (moving) mean −



## **Analyzing Sequential Data by Hidden Markov Model (HMM)**

HMM is a statistic model which is widely used for data having continuation and extensibility such as time series stock market analysis, health checkup, and speech recognition. This section deals in detail with analyzing sequential data using Hidden Markov Model (HMM).

### Hidden Markov Model (HMM)

HMM is a stochastic model which is built upon the concept of Markov chain based on the assumption that probability of future stats depends only on the current process state rather any state that preceded it. For example, when tossing a coin, we cannot say that the result of the fifth toss will be a head. This is because a coin does not have any memory and the next result does not depend on the previous result.

Mathematically, HMM consists of the following variables −

### States (S)

It is a set of hidden or latent states present in a HMM. It is denoted by S.

### Output symbols (O)

It is a set of possible output symbols present in a HMM. It is denoted by O.

### State Transition Probability Matrix (A)

It is the probability of making transition from one state to each of the other states. It is denoted by A.

### Observation Emission Probability Matrix (B)

It is the probability of emitting/observing a symbol at a particular state. It is denoted by B.

### Prior Probability Matrix (Π)

It is the probability of starting at a particular state from various states of the system. It is denoted by Π.

Hence, a HMM may be defined as **𝝀 = (S,O,A,B,𝝅)**,

where,

* **S = {s1,s2,…,sN}** is a set of N possible states,
* **O = {o1,o2,…,oM}** is a set of M possible observation symbols,
* A is an **N𝒙N** state Transition Probability Matrix (TPM),
* B is an **N𝒙M** observation or Emission Probability Matrix (EPM),
* π is an N dimensional initial state probability distribution vector.

## **Example: Analysis of Stock Market data**

In this example, we are going to analyze the data of stock market, step by step, to get an idea about how the HMM works with sequential or time series data. Please note that we are implementing this example in Python.

Import the necessary packages as shown below −

import datetime

import warnings

Now, use the stock market data from the **matpotlib.finance** package, as shown here −

import numpy as np

from matplotlib import cm, pyplot as plt

from matplotlib.dates import YearLocator, MonthLocator

try:

from matplotlib.finance import quotes\_historical\_yahoo\_och1

except ImportError:

from matplotlib.finance import (

quotes\_historical\_yahoo as quotes\_historical\_yahoo\_och1)

from hmmlearn.hmm import GaussianHMM

Load the data from a start date and end date, i.e., between two specific dates as shown here −

start\_date = datetime.date(1995, 10, 10)

end\_date = datetime.date(2015, 4, 25)

quotes = quotes\_historical\_yahoo\_och1('INTC', start\_date, end\_date)

In this step, we will extract the closing quotes every day. For this, use the following command −

closing\_quotes = np.array([quote[2] for quote in quotes])

Now, we will extract the volume of shares traded every day. For this, use the following command −

volumes = np.array([quote[5] for quote in quotes])[1:]

Here, take the percentage difference of closing stock prices, using the code shown below −

diff\_percentages = 100.0 \* np.diff(closing\_quotes) / closing\_quotes[:-]

dates = np.array([quote[0] for quote in quotes], dtype = np.int)[1:]

training\_data = np.column\_stack([diff\_percentages, volumes])

In this step, create and train the Gaussian HMM. For this, use the following code −

hmm = GaussianHMM(n\_components = 7, covariance\_type = 'diag', n\_iter = 1000)

with warnings.catch\_warnings():

warnings.simplefilter('ignore')

hmm.fit(training\_data)

Now, generate data using the HMM model, using the commands shown −

num\_samples = 300

samples, \_ = hmm.sample(num\_samples)

Finally, in this step, we plot and visualize the difference percentage and volume of shares traded as output in the form of graph.

Use the following code to plot and visualize the difference percentages −

plt.figure()

plt.title('Difference percentages')

plt.plot(np.arange(num\_samples), samples[:, 0], c = 'black')

Use the following code to plot and visualize the volume of shares traded −

plt.figure()

plt.title('Volume of shares')

plt.plot(np.arange(num\_samples), samples[:, 1], c = 'black')

plt.ylim(ymin = 0)

plt.show()