NumPy

NumPy is a Python library used for working with arrays.

NumPy stands for Numerical Python. In Python we have lists that serve the purpose of arrays, but they are slow to process.

NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.

NumPy Faster Than Lists

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

This behavior is called locality of reference in computer science.

This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

Import NumPy

Once NumPy is installed, import it in your applications by adding the `import` keyword:

```
import numpy
```

Now NumPy is imported and ready to use.

Example

```
import numpy

arr = numpy.array([1, 2, 3, 4, 5])

print(arr)
```

```
import numpy as np
```

Now the NumPy package can be referred to as `np` instead of `numpy`.

Example

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)
```

Various Operations that can be applied on Pandas DataFrames:

Pandas uses the function concatenation—**concat()**, aka concat. But it's easier to understand if you think of these are **inner joins** (intersection) and **outer joins** (union) of sets, which is how I refer to it below.
(This tutorial is part of our Pandas Guide. Use the right-hand menu to navigate.)

Concatenation (Outer join)

Think of concatenation like an outer join. The result is the same.

Suppose we have dataframes A and B with common elements among the indexes and columns.

Dataframe A

	column1	column2	column3	column4
1	A	F	J	N
2	C	G	K	O
3	D	H	L	P
4	E	I	M	Q

Dataframe B

	column3	column5	column6	column7
3	R	V	Z	σ
4	S	W	α	χ
5	T	X	β	ι
6	U	Y	υ	κ

Now concatenate. It's not an append. (There is an append() function for that.) This concat() operation creates a superset of both sets a and b but combines the common rows. It's not an inner join, either, since it lists all rows even those for which there is no common index.

Notice the missing values **NaN**. This is where there are no corresponding dataframe indexes in Dataframe B with the index in Dataframe A.
For example, index 3 is in both dataframes. So, Pandas copies the 4 columns from the first dataframe and the 4 columns from the second dataframe to the newly constructed dataframe. Similarly, index 5 is in Dataframe B but not Dataframe A for columns 1,2, 3. So those columns are marked as missing (NaN).

Copy

```
a = pd.DataFrame({'column1': ['A', 'C', 'D', 'E'],
'column2': ['F', 'G', 'H', 'I'],
'column3': ['J', 'K', 'L', 'M'],
'column4': ['N', 'O', 'P', 'Q']},
index=[1,2,3,4])
b = pd.DataFrame({'column3': ['R', 'S', 'T', 'U'],
'column5': ['V', 'W', 'X', 'Y'],
'column6': ['Z', 'α', 'β', 'υ'],
'column7': ['σ', 'χ', 'ι', 'κ']},
index=[3,4,5,6])
```

```
result = pd.concat([a, b], axis=1)
```
Results in:

	column1	column2	column3	column4	column3	column5	column6	column7
1	A	F	J	N	NaN	NaN	NaN	NaN
2	C	G	K	O	NaN	NaN	NaN	NaN
3	D	H	L	P	R	V	Z	σ
4	E	I	M	Q	S	W	α	χ
5	NaN	NaN	NaN	NaN	T	X	β	ι
6	NaN	NaN	NaN	NaN	U	Y	υ	κ

Outer join

Here we do an outer join, which, in terms of sets, means the union of two sets. So, all rows are added. An outer join here does not create the intersection of common indexes. But still, for those for which the column does not exist in the set of all columns the value is NaN. That has to be the case since not all columns exist for all rows. So, you have to list all of them but mark some of them as empty.

```
result = pd.concat([a, b], join='outer')
```

Inner join along the 0 axis (Row)

	column1	column2	column3	column4	column5	column6	column7
1	A	F	J	N	NaN	NaN	NaN
2	C	G	K	O	NaN	NaN	NaN
3	D	H	L	P	NaN	NaN	NaN
4	E	I	M	Q	NaN	NaN	NaN
3	NaN	NaN	R	NaN	V	Z	σ
4	NaN	NaN	S	NaN	W	α	χ
5	NaN	NaN	T	NaN	X	β	ι
6	NaN	NaN	U	NaN	Y	υ	κ

We can do an inner join by index or column. An inner join finds the intersection of two sets.

Let’s join along the 0 axis (row). Only indices 3 and 4 are in both dataframes. So, an inner join takes all columns from only those two rows.

```
result = pd.concat([a, b], axis=1,join='inner')
```

	column1	column2	column3	column4	column3	column5	column6	column7
3	D	H	L	P	R	V	Z	σ
4	E	I	M	Q	S	W	α	χ

Inner join along the 1 axis (Column)

Column3 is the only column common to both dataframe. So, we concatenate all the rows from A with the rows in B and select only the common column, i.e., an inner join along the column axis.

```
result = pd.concat([a, b], axis=0,join='inner')
```

	column3
1	J
2	K
3	L
4	M
3	R
4	S
5	T
6	U

Merge

A merge is like an inner join, except we tell it what column to merge on.

Here, make the first column name (i.e., key value in the dictionary) some common name “key”. Then we merge on that.

```
a = pd.DataFrame({'key': ['A', 'C', 'D', 'E'],
'column2': ['F', 'G', 'H', 'I'],
'column3': ['J', 'K', 'L', 'M'],
'column4': ['N', 'O', 'P', 'Q']},
index=[1,2,3,4])
b = pd.DataFrame({'key': ['C', 'D', 'T', 'U'],
'column5': ['V', 'W', 'X', 'Y'],
'column6': ['Z', 'α', 'β', 'υ'],
'column7': ['σ', 'χ', 'ι', 'κ']},
index=[3,4,5,6])
result=pd.merge(a, b, on='key')
```

The resulting dataframe is the one which has a common key value from the array **key=[]**.You see that only C and D are common to both data frames.

	column2	column3	column4	key	column5	column6	column7
0	G	K	O	C	V	Z	σ
1	H	L	P	D	W	α	χ

Append

This simply appends one dataframe onto another. Here is a Series, which is a DataFrame with only one column. The result is all rows from Dataframe A added to Dataframe B to create Dataframe C.

```
import pandas as pd
a=pd.DataFrame([1,2,3])
b=pd.DataFrame([4,5,6])
c=a.append(b)
c
```

	0
0	1
1	2
2	3
0	4
1	5
2	6

Python Try Except

The `try` block lets you test a block of code for errors.

The `except` block lets you handle the error.

The `else` block lets you execute code when there is no error.

The `finally` block lets you execute code, regardless of the result of the try- and except blocks.

Exception Handling: When an error occurs, or exception as we call it, Python will normally stop and generate an error message.These exceptions can be handled using the `try` statement:

Example

The `try` block will generate an exception, because `x` is not defined:

```
try:
  print(x)
except:
  print("An exception occurred")
```

Since the try block raises an error, the except block will be executed.

Without the try block, the program will crash and raise an error:

Example

This statement will raise an error, because `x` is not defined:

```
print(x)
```

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example

Print one message if the try block raises a `NameError` and another for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

Python try...finally

The `try` statement in Python can have an optional `finally` clause. This clause is executed no matter what, and is generally used to release external resources.

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")

finally:
  print("Program did not run")
```

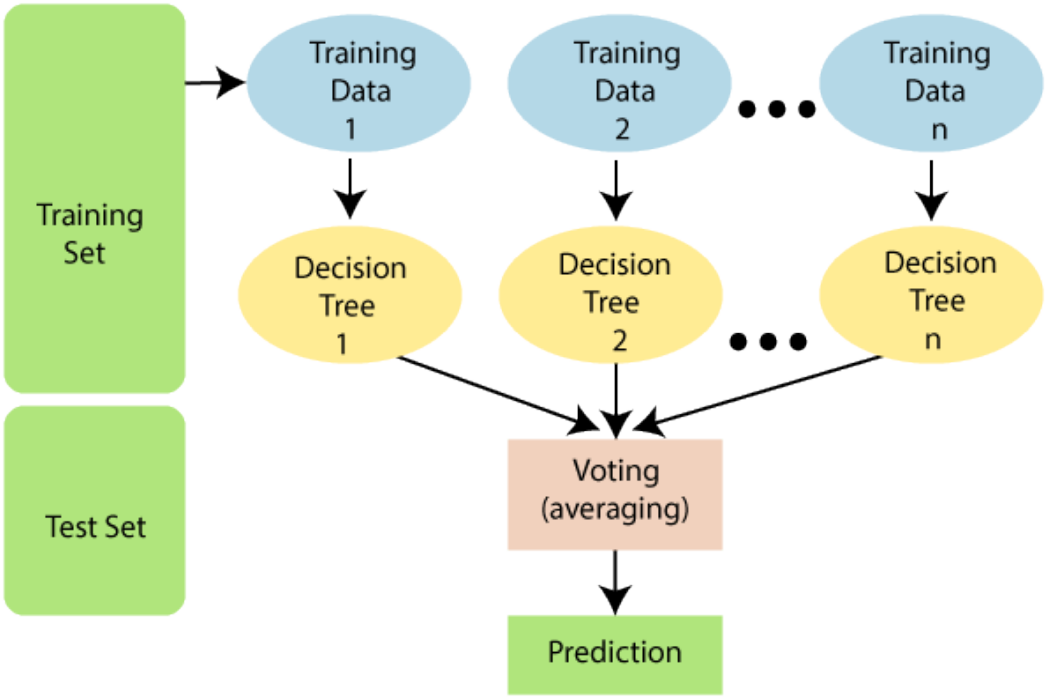
Random Forest Algorithm

Random Forest is a popular machine learning algorithm that belongs to the supervised learning technique. It can be used for both Classification and Regression problems in ML. It is based on the concept of ensemble learning, which is a process of *combining multiple classifiers to solve a complex problem and to improve the performance of the model.*

As the name suggests, *"Random Forest is a classifier that contains a number of decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset."* Instead of relying on one decision tree, the random forest takes the prediction from each tree and based on the majority votes of predictions, and it predicts the final output.

The greater number of trees in the forest leads to higher accuracy and prevents the problem of overfitting.

The below diagram explains the working of the Random Forest algorithm:



Assumptions for Random Forest

Since the random forest combines multiple trees to predict the class of the dataset, it is possible that some decision trees may predict the correct output, while others may not. But together, all the trees predict the correct output. Therefore, below are two assumptions for a better Random forest classifier:

- There should be some actual values in the feature variable of the dataset so that the classifier can predict accurate results rather than a guessed result.
- The predictions from each tree must have very low correlations.

Why use Random Forest?

Below are some points that explain why we should use the Random Forest algorithm:

- It takes less training time as compared to other algorithms.
- It predicts output with high accuracy, even for the large dataset it runs efficiently.
- It can also maintain accuracy when a large proportion of data is missing.

How does Random Forest algorithm work?

Random Forest works in two-phase first is to create the random forest by combining N decision tree, and second is to make predictions for each tree created in the first phase.

The Working process can be explained in the below steps and diagram:

Step-1: Select random K data points from the training set.

Step-2: Build the decision trees associated with the selected data points (Subsets).

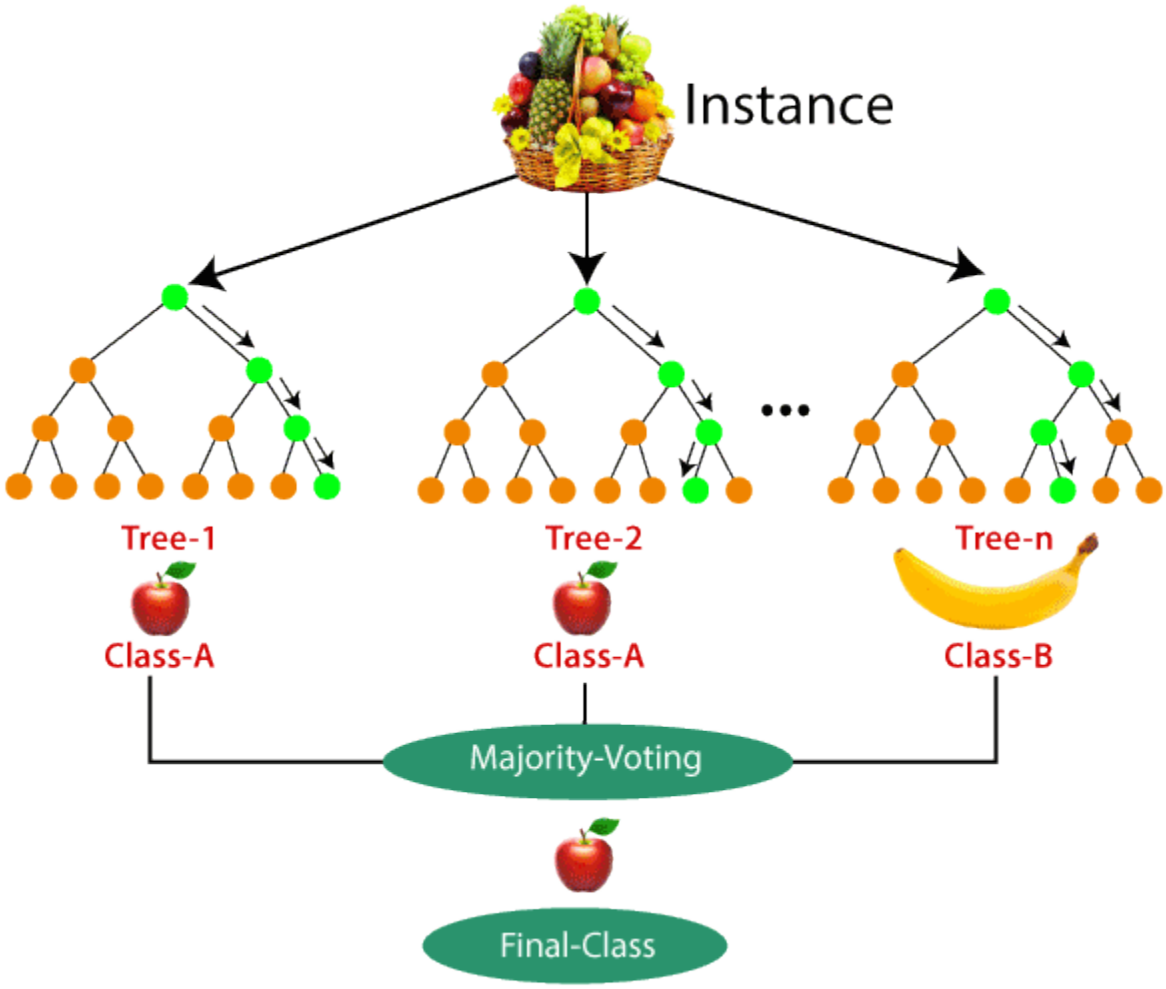
Step-3: Choose the number N for decision trees that you want to build.

Step-4: Repeat Step 1 & 2.

Step-5: For new data points, find the predictions of each decision tree, and assign the new data points to the category that wins the majority votes.

The working of the algorithm can be better understood by the below example:

Example: Suppose there is a dataset that contains multiple fruit images. So, this dataset is given to the Random forest classifier. The dataset is divided into subsets and given to each decision tree. During the training phase, each decision tree produces a prediction result, and when a new data point occurs, then based on the majority of results, the Random Forest classifier predicts the final decision. Consider the below image:



Applications of Random Forest

There are mainly four sectors where Random forest mostly used:

1. Banking: Banking sector mostly uses this algorithm for the identification of loan risk.
2. Medicine: With the help of this algorithm, disease trends and risks of the disease can be identified.
3. Land Use: We can identify the areas of similar land use by this algorithm.
4. Marketing: Marketing trends can be identified using this algorithm.

Advantages of Random Forest

- Random Forest is capable of performing both Classification and Regression tasks.
- It is capable of handling large datasets with high dimensionality.
- It enhances the accuracy of the model and prevents the overfitting issue.

Disadvantages of Random Forest

- Although random forest can be used for both classification and regression tasks, it is not more suitable for Regression tasks.

Python Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Creating a Function

In Python a function is defined using the def keyword:

Example

```
def my_function():
    print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

Example:

Call by Value:

```
def greet():
    print(“hi”)
    print(“welcome”)
// Two line breaks
greet()
```

Call by Reference:

```
def greet(first_name , last_name ):
    print(“hi”)
    print(“welcome”)
// Two line breaks
greet(“Meera”, “ajay”)
```

Pandas:

Pandas are a Python library.

Pandas are used to analyze data.

Pandas are a Python library used for working with data sets.

It has functions for analyzing, cleaning, exploring, and manipulating data.

The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Why Use Pandas?

Pandas allow us to analyze big data and make conclusions based on statistical theories.

Pandas can clean messy data sets, and make them readable and relevant.

Relevant data is very important in data science.

```
import pandas

mydataset = {
  'cars': ["BMW", "Volvo", "Ford"],
  'passings': [3, 7, 2]
}

myvar = pandas.DataFrame(mydataset)

print(myvar)
```

O/P: cars passings

0 BMW 3

1 Volvo 7

2 Ford 2

A Pandas Series is like a column in a table.

It is a one-dimensional array holding data of any type.

Example

Create a simple Pandas Series from a list:

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a)

print(myvar)
```

Create Labels

With the index argument, you can name your own labels.

Example

Create you own labels:

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a, index = ["x", "y", "z"])

print(myvar)
```

O/P

x 1

y 7

z 2

DataFrames

Data sets in Pandas are usually multi-dimensional tables, called Data Frames.

Series is like a column, a Data Frame is the whole table.

Example

Create a Data Frame from two Series:

```
import pandas as pd

data = {
```

```
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}

myvar = pd.DataFrame(data)

print(myvar)
```

o/p

	calories	duration
0	**420**	**50**
1	**380**	**40**
2	**390**	**45**

Pandas - Analyzing DataFrames

Viewing the Data

One of the most used method for getting a quick overview of the DataFrame, is the head() method.

The head() method returns the headers and a specified number of rows, starting from the top.

Example

Get a quick overview by printing the first 10 rows of the DataFrame:

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.head(10))
```

There is also a tail() method for viewing the *last* rows of the DataFrame.

The tail() method returns the headers and a specified number of rows, starting from the bottom.

Example

Print the last 5 rows of the DataFrame:

```
print(df.tail())
```

Support Vector Machine Algorithm

Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane.

SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called as support vectors, and hence algorithm is termed as Support Vector Machine

SVM algorithm can be used for Face detection, image classification, text categorization, etc.

Types of SVM

SVM can be of two types:

- Linear SVM: Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.
- Non-linear SVM: Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data and classifier used is called as Non-linear SVM classifier.

Hyperplane and Support Vectors in the SVM algorithm:

Hyperplane: There can be multiple lines/decision boundaries to segregate the classes in n-dimensional space, but we need to find out the best decision boundary that helps to classify the data points. This best boundary is known as the hyperplane of SVM.

The dimensions of the hyperplane depend on the features present in the dataset, which means if there are 2 features (as shown in image), then hyperplane will be a straight line. And if there are 3 features, then hyperplane will be a 2-dimension plane.

We always create a hyperplane that has a maximum margin, which means the maximum distance between the data points.

Support Vectors:

The data points or vectors that are the closest to the hyperplane and which affect the position of the hyperplane are termed as Support Vector. Since these vectors support the hyperplane, hence called a Support vector.

Example: SVM can be understood with the example that we have used in the KNN classifier. Suppose we see a strange cat that also has some features of dogs, so if we want a model that can accurately identify whether it is a cat or dog, so such a model can be created by using the SVM algorithm. We will first train our model with lots of images of cats and dogs so that it can learn about different features of cats and dogs, and then we test it with this strange creature. So as support vector creates a decision boundary between these two data (cat and dog) and choose extreme cases (support vectors), it will see the extreme case of cat and dog. On the basis of the support vectors, it will classify it as a cat.

Decision Tree Classification Algorithm

- Decision Tree is a Supervised learning technique that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. It is a tree-structured classifier, where internal nodes represent the features of a dataset, branches represent the decision rules and each leaf node represents the outcome.
- In a Decision tree, there are two nodes, which are the Decision Node and Leaf Node. Decision nodes are used to make any decision and have multiple branches, whereas Leaf nodes are the output of those decisions and do not contain any further branches.
- The decisions or the test are performed on the basis of features of the given dataset.
- *It is a graphical representation for getting all the possible solutions to a problem/decision based on given conditions.*
- It is called a decision tree because, similar to a tree, it starts with the root node, which expands on further branches and constructs a tree-like structure.
- In order to build a tree, we use the CART algorithm, which stands for Classification and Regression Tree algorithm.
- A decision tree simply asks a question, and based on the answer (Yes/No), it further split the tree into subtrees.
- Below diagram explains the general structure of a decision tree:

Decision Tree algorithm

In a decision tree, for predicting the class of the given dataset, the algorithm starts from the root node of the tree. This algorithm compares the values of root attribute with the record (real dataset) attribute and, based on the comparison, follows the branch and jumps to the next node.

For the next node, the algorithm again compares the attribute value with the other sub-nodes and move further. It continues the process until it reaches the leaf node of the tree. The complete process can be better understood using the below algorithm:

- Step-1: Begin the tree with the root node, says S, which contains the complete dataset.
- Step-2: Find the best attribute in the dataset using Attribute Selection Measure (ASM).
- Step-3: Divide the S into subsets that contains possible values for the best attributes.
- Step-4: Generate the decision tree node, which contains the best attribute.
- Step-5: Recursively make new decision trees using the subsets of the dataset created in step -3. Continue this process until a stage is reached where you cannot further classify the nodes and called the final node as a leaf node.

Advantages of the Decision Tree

- It is simple to understand as it follows the same process which a human follow while making any decision in real-life.
- It can be very useful for solving decision-related problems.
- It helps to think about all the possible outcomes for a problem.
- There is less requirement of data cleaning compared to other algorithms.

Disadvantages of the Decision Tree

- The decision tree contains lots of layers, which makes it complex.
- It may have an overfitting issue, which can be resolved using the Random Forest algorithm.
- For more class labels, the computational complexity of the decision tree may increase.