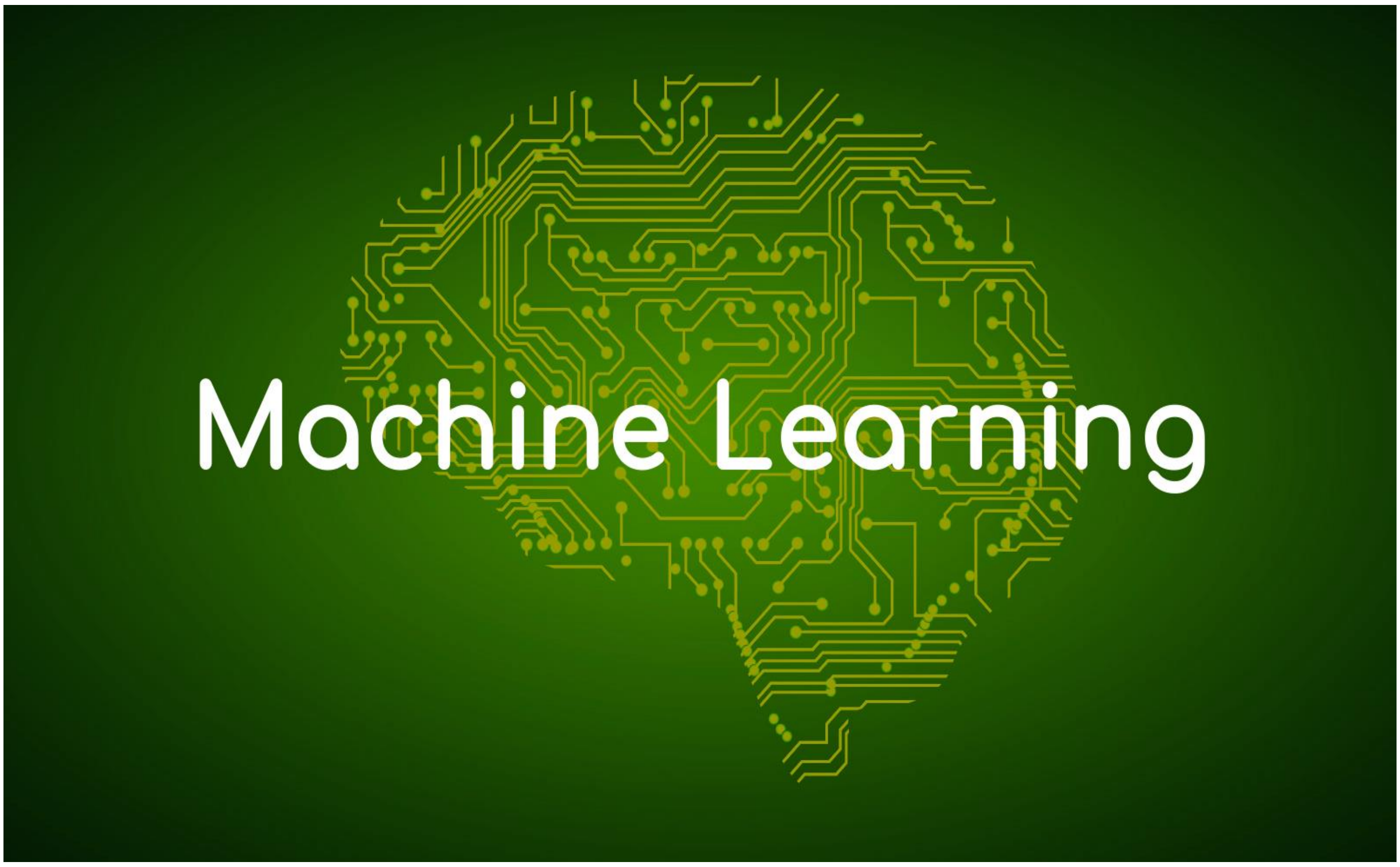


Machine Learning Hand Book

Part: 2

Edited By: Susmoy Barman

Source: GeeksForGeeks

Dimensionality Reduction :

Parameters for Feature Selection

Prerequisite : Introduction to Dimensionality Reduction

Dimensionality reduction is the process of reducing the number of random variables under consideration, by obtaining a set of principal variables. It can be divided into ***feature selection*** and ***feature extraction***.

Dimensionality Reduction is an important factor in predictive modeling. Various proposed methods have introduced different approaches to do so by either graphically or by various other methods like filtering, wrapping or embedding. However, most of these approaches are based on some threshold values and benchmark algorithms that determine the optimality of the features in the dataset.

One motivation for dimensionality reduction is that higher dimensional data sets increase the time complexity and also the space required will be more. Also, all the features in the dataset might not be useful. Some may contribute no information at all, while some may contribute similar information as the other features. Selecting the optimal set of features will help us hence reduce the space and time complexity as well as increase the accuracy or purity of classification (or regression) and clustering (or association) for supervised and unsupervised learning respectively.

Feature selection has four different approaches such as filter approach, wrapper approach, embedded approach, and hybrid approach.

1. **Wrapper approach :** This approach has high computational complexity. It uses a learning algorithm to evaluate the accuracy produced by the use of the selected features in classification. Wrapper methods can give high classification accuracy for particular classifiers.
2. **Filter approach :** A subset of features is selected by this approach without using any learning algorithm. Higher-dimensional datasets use this method and it is relatively faster than the wrapper-based approaches.
3. **Embedded approach :** The applied learning algorithms determine the specificity of this approach and it selects the features during the process of training the data set.
4. **Hybrid approach :** Both filter and wrapper-based methods are used in hybrid approach. This approach first selects the possible optimal feature set which is further tested by the wrapper approach. It hence uses the advantages of both filter and wrapper-based approach.

Parameters For Feature Selection :

The parameters are classified based on two factors –

The Similarity of information contributed by the features :

1. CORRELATION

The features are classified as associated or similar mostly based on their correlation factor. In the data set, we have many features which are correlated. Now the problem with having correlated features is that, if f1 and f2 are two correlated features of a data set, then the classifying or regression model including both f1 and f2 will give the same as the predictive model compared to the scenario where either f1 or f2 was included in the dataset. This is because both f1 and f2 are correlated and hence they contribute the same information regarding the model in the data set. There are various methods to calculate the correlation factor, however, Pearson's correlation coefficient is most widely used. The for

mula for Pearson's correlation coefficient(ρ) is:

$$\rho_{X,Y} = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y}$$

```
where
cov(X, Y) - covariance
sigma(X) - standard deviation of X
sigma(Y) - standard deviation of Y
```

Thus, the correlated features are irrelevant, as they all contribute similar information. Only one representative of the whole correlated or associated features would give the same classification or regression outcome. Hence these features are redundant and excluded for dimensionality reduction purposes after selecting a particular representative from each associated or correlated group of features using various algorithms.

Quantum of information contributed by the features :

1. ENTROPY

Entropy is the measure of the average information content. The higher the entropy, the higher is the information contribution by that feature. Entropy (H) can be formulated as:

$$\mathrm{H}(X) = \mathrm{E}[\mathrm{I}(X)] = \mathrm{E}[-\ln(\mathrm{P}(X))]$$

```
where
X - discrete random variable X
P(X) - probability mass function
E - expected value operator,
I - information content of X.
I(X) - a random variable.
```

In Data Science, entropy of a feature f1 is calculated by excluding feature f1 and then calculating the entropy of the rest of the features. Now, the lower the entropy value (excluding f1) the higher will be the information content of f1. In this manner the entropy of all the features is calculated. At the end, either a threshold value or further relevancy check determines the optimality of the features on the basis of which features are selected. Entropy is mostly used for Unsupervised Learning as we do have a class field in the dataset and hence entropy of the features can give substantial information.

2. MUTUAL INFORMATION
In information theory, mutual information I(X;Y) is the amount of uncertainty in X due to the knowledge of Y. Mathematically, mutual information is defined as

$$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} p(x,y) \log \left(\frac{p(x,y)}{p(x)\,p(y)} \right)$$

```
where
p(x, y) - joint probability function of X and Y,
p(x) - marginal probability distribution function of X
p(y) - marginal probability distribution function of Y
```

Mutual Information in Data science is mostly calculated to know the amount of information shared about the class by a feature. Hence is mostly used for dimensionality reduction in Supervised Learning. The features which have high mutual information value corresponding to the class in a supervised learning are considered optimal since they can influence the predictive model towards the right prediction and hence increase the accuracy of the model.

Introduction to Dimensionality Reduction

Machine Learning: As discussed in this article, machine learning is nothing but a field of study which allows computers to "learn" like humans without any need of explicit programming.

What is Predictive Modeling: Predictive modeling is a probabilistic process that allows us to forecast outcomes, on the basis of some predictors. These predictors are basically features that come into play when deciding the final result, i.e. the outcome of the model.

What is Dimensionality Reduction?

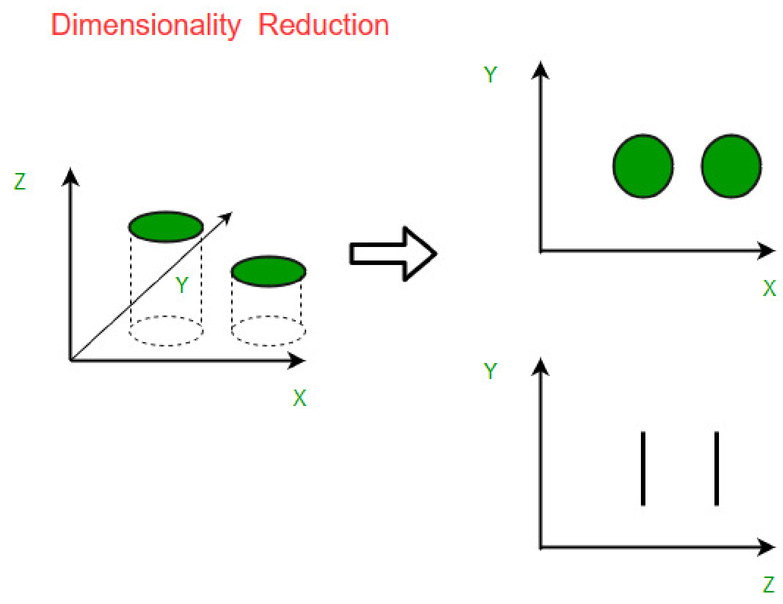
In machine learning classification problems, there are often too many factors on the basis of which the final classification is done. These factors are basically variables called features. The higher the number of features, the harder it gets to visualize the training set and then work on it. Sometimes, most of these features are correlated, and hence redundant. This is where dimensionality reduction algorithms come into play. Dimensionality reduction is the process of reducing the number of random variables under consideration, by obtaining a set of principal variables. It can be divided into feature selection and feature extraction.

Why is Dimensionality Reduction important in Machine Learning and Predictive Modeling?

An intuitive example of dimensionality reduction can be discussed through a simple e-mail classification problem, where we need to classify whether the e-mail is spam or not. This can involve a large number of features, such as whether or not the e-mail has a generic title, the content of the e-mail, whether the e-mail

uses a template, etc. However, some of these features may overlap. In another condition, a classification problem that relies on both humidity and rainfall can be collapsed into just one underlying feature, since both of the aforementioned are correlated to a high degree. Hence, we can reduce the number of features in such problems. A 3-D classification problem can be hard to visualize, whereas a 2-D one can be mapped to a simple 2 dimensional space, and a 1-D problem to a simple line. The below figure illustrates this concept,

where a 3-D feature space is split into two 1-D feature spaces, and later, if found to be correlated, the number of features can be reduced even further.



Components of Dimensionality Reduction

There are two components of dimensionality reduction:

- **Feature selection:** In this, we try to find a subset of the original set of variables, or features, to get a smaller subset which can be used to model the problem. It usually involves three ways:
  1. Filter
  2. Wrapper
  3. Embedded
- **Feature extraction:** This reduces the data in a high dimensional space to a lower dimension space, i.e. a space with lesser no. of dimensions.

Methods of Dimensionality Reduction

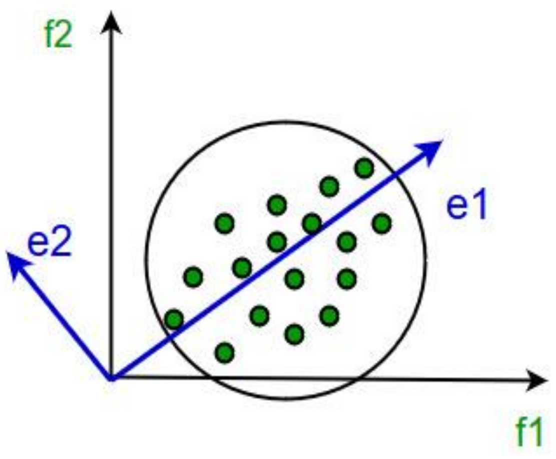
The various methods used for dimensionality reduction include:

- Principal Component Analysis (PCA)
- Linear Discriminant Analysis (LDA)
- Generalized Discriminant Analysis (GDA)

Dimensionality reduction may be both linear or non-linear, depending upon the method used. The prime linear method, called Principal Component Analysis, or PCA, is discussed below.

Principal Component Analysis

This method was introduced by Karl Pearson. It works on a condition that while the data in a higher dimensional space is mapped to data in a lower dimension space, the variance of the data in the lower dimensional space should be maximum.



It involves the following steps:

- Construct the covariance matrix of the data.
- Compute the eigenvectors of this matrix.
- Eigenvectors corresponding to the largest eigenvalues are used to reconstruct a large fraction of variance of the original data.

Hence, we are left with a lesser number of eigenvectors, and there might have been some data loss in the process. But, the most important variances should be retained by the remaining eigenvectors.

Advantages of Dimensionality Reduction

- It helps in data compression, and hence reduced storage space.
- It reduces computation time.
- It also helps remove redundant features, if any.

Disadvantages of Dimensionality Reduction

- It may lead to some amount of data loss.
- PCA tends to find linear correlations between variables, which is sometimes undesirable.
- PCA fails in cases where mean and covariance are not enough to define datasets.
- We may not know how many principal components to keep- in practice, some thumb rules are applied.

This article is contributed by **Anannya Uberoi**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Underfitting and Overfitting in Machine Learning

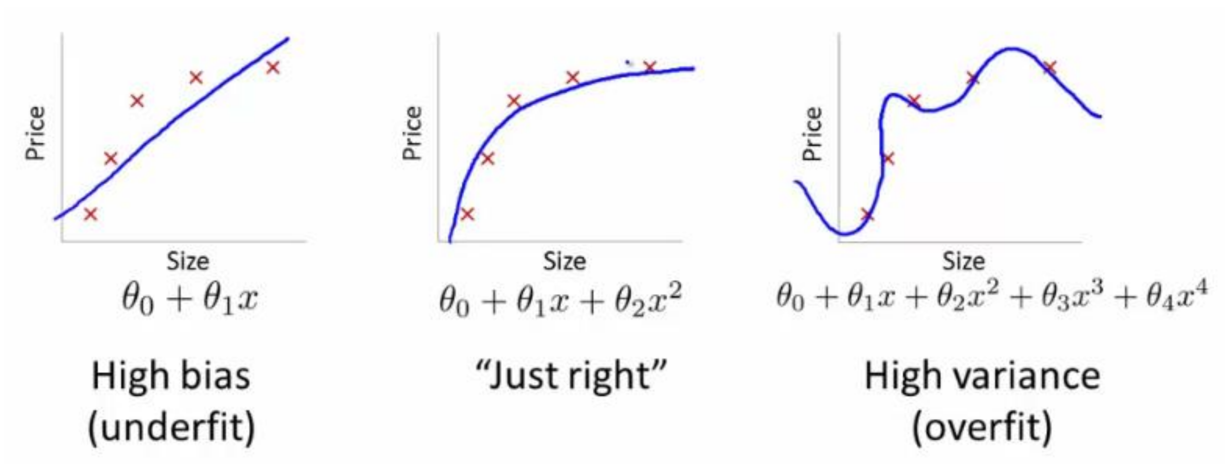
Let us consider that we are designing a machine learning model. A model is said to be a good machine learning model, if it generalizes any new input data from the problem domain in a proper way. This helps us to make predictions in the future data, that data model has never seen.

Now, suppose we want to check how well our machine learning model learns and generalizes to the new data. For that we have overfitting and underfitting, which are majorly responsible for the poor performances of the machine learning algorithms.

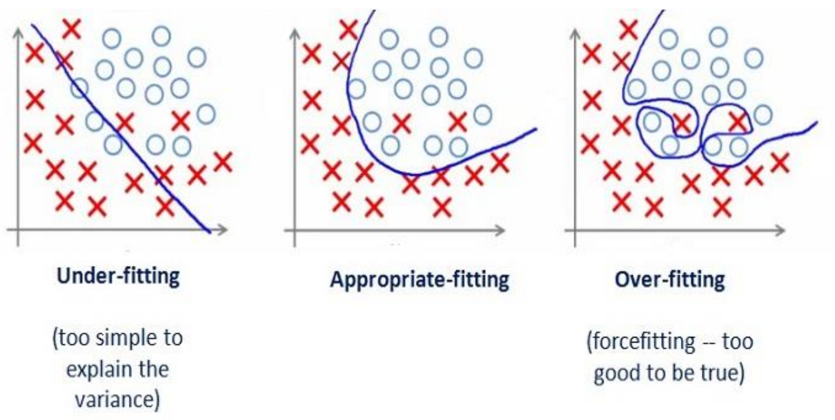
Underfitting:
A statistical model or a machine learning algorithm is said to have underfitting when it cannot capture the underlying trend of the data. *(It's just like trying to fit undersized pants!)* Underfitting destroys the accuracy of our machine learning model. Its occurrence simply means that our model or the algorithm does not fit the data well enough. It usually happens when we have less data to build an accurate model and also when we try to build a linear model with a non-linear data. In such cases the rules of the machine learning model are too easy and flexible to be applied on such a minimal data and therefore the model will probably make a lot of wrong predictions. Underfitting can be avoided by using more data and also reducing the features by feature selection.

Overfitting:
A statistical model is said to be overfitted, when we train it with a lot of data *(just like fitting ourselves in an oversized pants!)*. When a model gets trained with so much of data, it starts learning from the noise and inaccurate data entries in our data set. Then the model does not categorize the data correctly, because of too much of details and noise. The causes of overfitting are the non-parametric and non-linear methods because these types of machine learning algorithms have more freedom in building the model based on the dataset and therefore they can really build unrealistic models. A solution to avoid overfitting is using a linear algorithm if we have linear data or using the parameters like the maximal depth if we are using decision trees.

Examples:



Image_source: i.stack.imgur.com/t0zit.png


Under-fitting
(too simple to explain the variance)
Appropriate-fitting
Over-fitting
(forcefitting -- too good to be true)

Image_source: vitalflux.com/

How to avoid Overfitting:

The commonly used methodologies are:

- **Cross- Validation:** A standard way to find out-of-sample prediction error is to use 5-fold cross validation.
- **Early Stopping:** Its rules provide us the guidance as to how many iterations can be run before learner begins to over-fit.
- **Pruning:** Pruning is extensively used while building related models. It simply removes the nodes which add little predictive power for the problem in hand.
- **Regularization:** It introduces a cost term for bringing in more features with the objective function. Hence it tries to push the coefficients for many variables to zero and hence reduce cost term.

Good Fit in a Statistical Model:

Ideally, the case when the model makes the predictions with 0 error, is said to have a *good fit* on the data. This situation is achievable at a spot between overfitting and underfitting. In order to understand it we will have to look at the performance of our model with the passage of time, while it is learning from training dataset.

With the passage of time, our model will keep on learning and thus the error for the model on the training and testing data will keep on decreasing. If it will learn for too long, the model will become more prone to overfitting due to presence of noise and less useful details. Hence the performance of our model will decrease. In order to get a good fit, we will stop at a point just before where the error starts increasing. At this point the model is said to have good skills on training dataset as well our unseen testing dataset.

ML | Handling Missing Values

With this article be ready to get your hands dirty with ML algorithms, concepts, Maths and coding.

To work with ML code, libraries play a very important role in Python which we will study in details but let see a very brief description of the most important ones :

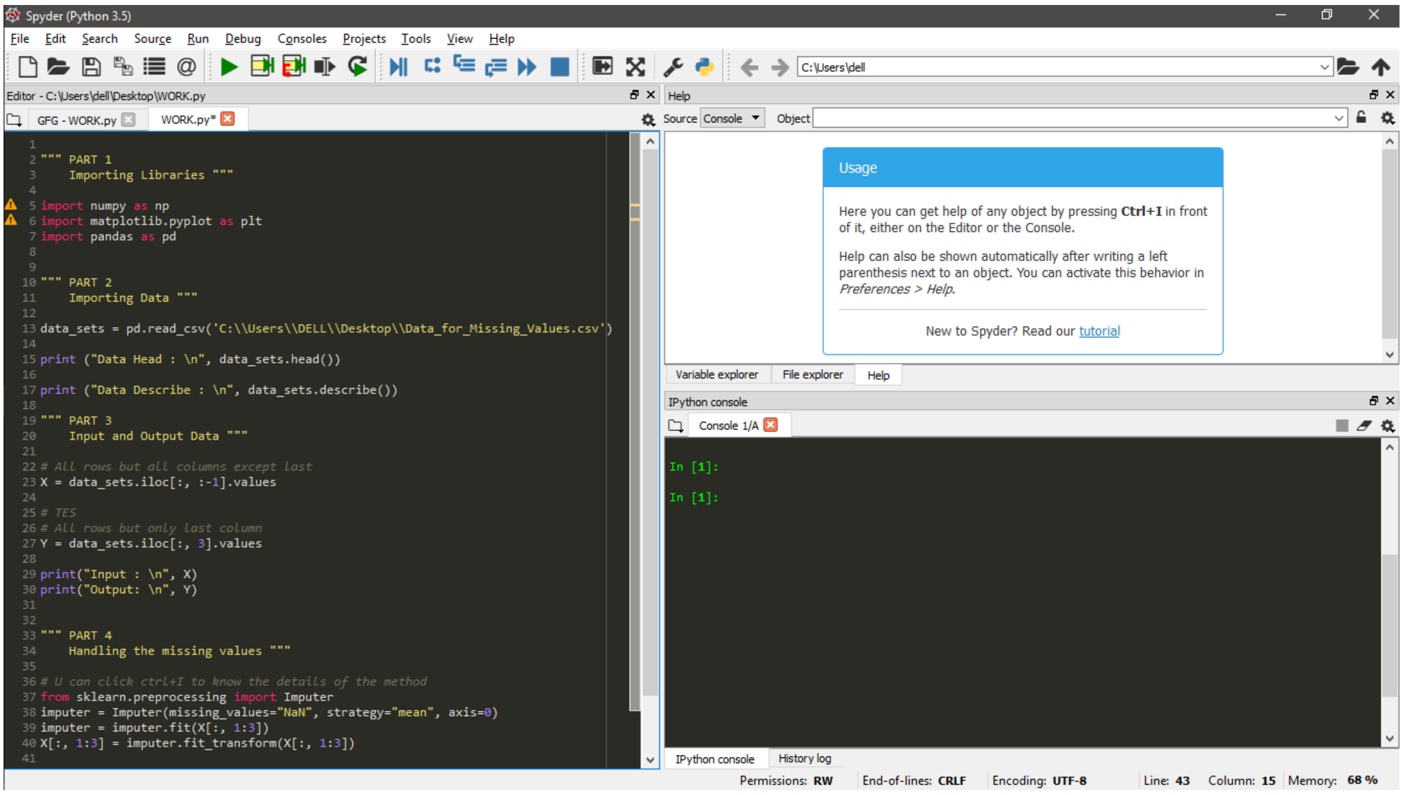
- **NumPy (Numerical Python) :** It is one of the greatest Scientific and Mathematical computing library for Python. Platforms like Keras, Tensorflow have embedded Numpy operations on Tensors. The feature we are concerned with its power and easy to handle and perform operation on Array.
- **Pandas :** This package is very useful when it comes to handle data. This makes it very easier to manipulate, aggregate and visualize data.
- **MatplotLib :** This library facilitates the task of powerful and very simple visualizations.

There are many more libraries but they have no use right now. So, let's begin.

Download the dataset :
Go to the link and download **Data_for_Missing_Values.csv**.

Anaconda :
I would suggest you guys to install [Anaconda]() on your systems. Launch Spyder our Jupyter on your system.

Reason behind suggesting is – Anaconda has all the basic Python Libraries pre installed in it.


Spyder (Python 3.5)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\dell
Editor - C:\Users\dell\Desktop\WORK.py
GFG - WORK.py
WORK.py*
Help
Source Console Object
Variable explorer File explorer Help
IPython console
Console 1/A
IPython console History log

```
""" PART 1
    Importing Libraries """

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


""" PART 2
    Importing Data """

data_sets = pd.read_csv('C:\\Users\\DELL\\Desktop\\Data_for_Missing_Values.csv')

print ("Data Head : \n", data_sets.head())

print ("Data Describe : \n", data_sets.describe())

""" PART 3
    Input and Output Data """

# All rows but all columns except last
X = data_sets.iloc[:, :-1].values

# TES
# All rows but only last column
Y = data_sets.iloc[:, 3].values

print("Input : \n", X)
print("Output: \n", Y)


""" PART 4
    Handling the missing values """

# U can click ctrl+I to know the details of the method
from sklearn.preprocessing import Imputer
imputer = Imputer(missing_values="NaN", strategy="mean", axis=0)
imputer = imputer.fit(X[:, 1:3])
X[:, 1:3] = imputer.fit_transform(X[:, 1:3])
```

Usage

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Help.*

New to Spyder? Read our tutorial

```
In [1]:

In [1]:
```

Permissions: RW End-of-lines: CRLF Encoding: UTF-8 Line: 43 Column: 15 Memory: 68 %

Below is the Python code :

```
# Python code explaining How to
# Handle Missing Value in Dataset

""" PART 1
    Importing Libraries """

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

```
""" PART 2
    Importing Data """

data_sets =
pd.read_csv('C:\\Users\\Admin\\Desktop\\Data_for_Missing_Values.csv')

print ("Data Head : \n", data_sets.head())

print ("\n\nData Describe : \n", data_sets.describe())
```

```
""" PART 3
    Input and Output Data """

# All rows but all columns except last

X = data_sets.iloc[:, :-1].values

# TES
# All rows but only last column
Y = data_sets.iloc[:, 3].values

print("\n\nInput : \n", X)
print("\n\nOutput: \n", Y)



""" PART 4
    Handling the missing values """

# We will use sklearn library >> preprocessing package
# Imputer class of that package
from sklearn.preprocessing import Imputer

# Using Imputer function to replace NaN
# values with mean of that parameter value
imputer = Imputer(missing_values = "NaN",
                  strategy = "mean", axis = 0)

# Fitting the data, function learns the stats
imputer = imputer.fit(X[:, 1:3])

# fit_transform() will execute those
# stats on the input ie. X[:, 1:3]
X[:, 1:3] = imputer.fit_transform(X[:, 1:3])

# filling the missing value with mean
print("\n\nNew Input with Mean Value for NaN : \n", X)
```

Ouput :

```
Data Head :
    Country   Age   Salary Purchased
0   France  44.0  72000.0        No
1    Spain  27.0  48000.0       Yes
```

```
2  Germany  30.0  54000.0        No
3    Spain  38.0  61000.0        No
4  Germany  40.0      NaN       Yes


Data Describe :
              Age        Salary
count   9.000000      9.000000
mean   38.777778  63777.777778
std     7.693793  12265.579662
```

```
min    27.000000  48000.000000
25%    35.000000  54000.000000
50%    38.000000  61000.000000
75%    44.000000  72000.000000
max    50.000000  83000.000000


Input :
 [['France' 44.0 72000.0]


 ['Spain' 27.0 48000.0]
 ['Germany' 30.0 54000.0]
 ['Spain' 38.0 61000.0]
 ['Germany' 40.0 nan]
 ['France' 35.0 58000.0]
 ['Spain' nan 52000.0]
 ['France' 48.0 79000.0]
 ['Germany' 50.0 83000.0]
 ['France' 37.0 67000.0]]


Output:
 ['No' 'Yes' 'No' 'No' 'Yes' 'Yes' 'No' 'Yes' 'No' 'Yes']


New Input with Mean Value for NaN :
 [['France' 44.0 72000.0]
 ['Spain' 27.0 48000.0]
 ['Germany' 30.0 54000.0]
 ['Spain' 38.0 61000.0]
 ['Germany' 40.0 63777.77777777778]
 ['France' 35.0 58000.0]
 ['Spain' 38.77777777777778 52000.0]
 ['France' 48.0 79000.0]
 ['Germany' 50.0 83000.0]
 ['France' 37.0 67000.0]]
```

CODE EXPLANATION :

- **Part 1 – Importing Libraries :** In the above code, imported numpy, pandas and matplotlib but we have used pandas only.
- **PART 2 – Importing Data :**
  - Import `Data_for_Missing_Values.csv` by giving the path to pandas read_csv function. Now “data_sets” is a DataFrame(Two-dimensional tabular data structure with labeled rows and columns).
  - Then print first 5 data-entries of the dataframe using **head()** function. Number of entries can be changed for e.g. for first 3 values we can use dataframe.head(3). Similarly, last values can also be gotten using **tail()** function.
  - Then used **describe()** function. It gives statistical summary of data which includes min, max, percentile (.25, .5, .75), mean and standard deviation for each parameter values.
- **PART 3 – Input and Output Data :** We split our dataframe to input and output.
- **PART 4 – Handling the missing values :** Using Imputer() function from sklearn.preprocessing package.

IMPUTER :

`Imputer(missing_values=’NaN’, strategy=’mean’, axis=0, verbose=0, copy=True)` is a function from Imputer class of sklearn.preprocessing package. It’s role is to transformer parameter value from missing values(NaN) to set strategic value.

```
Syntax : sklearn.preprocessing.Imputer()

Parameters :

-> missing_values  : integer or “NaN”
-> strategy        : What to impute - mean, median or most_frequent along axis
12 | Page

-> axis(default=0) : 0 means along column and 1 means along row
```

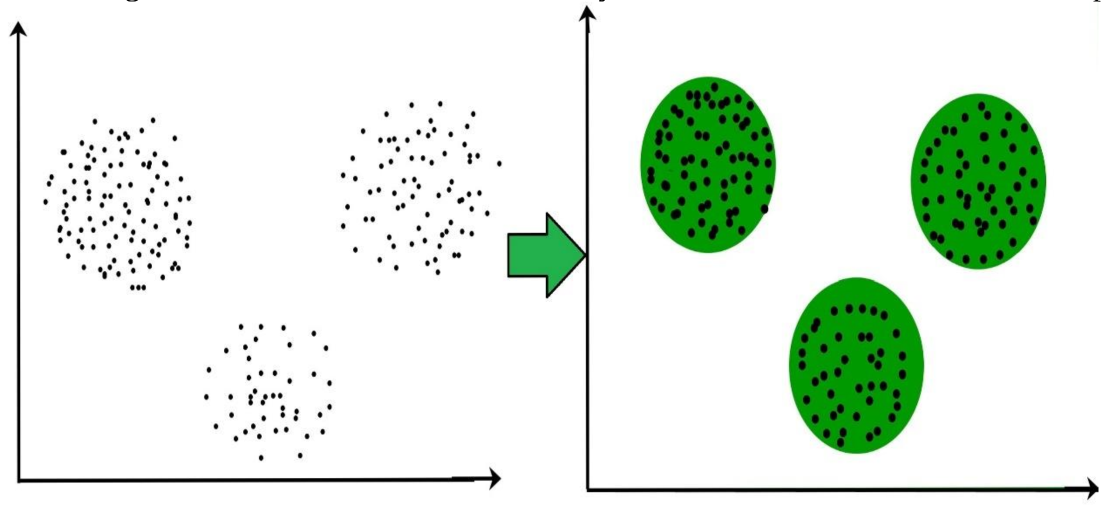
Clustering :

Clustering in Machine Learning

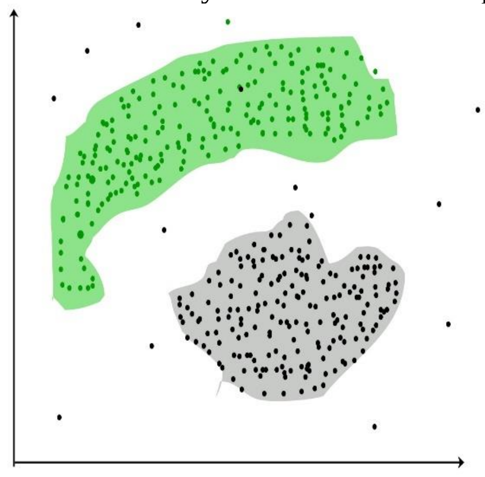
Introduction to Clustering

It is basically a type of *unsupervised learning method* . An unsupervised learning method is a method in which we draw references from datasets consisting of input data without labeled responses. Generally, it is used as a process to find meaningful structure, explanatory underlying processes, generative features, and groupings inherent in a set of examples.
Clustering is the task of dividing the population or data points into a number of groups such that data points in the same groups are more similar to other data points in the same group and dissimilar to the data points in other groups. It is basically a collection of objects on the basis of similarity and dissimilarity between them.

For ex– The data points in the graph below clustered together can be classified into one single group. We can distinguish the clusters, and we can identify that there are 3 clusters in the below picture.



It is not necessary for clusters to be a spherical. Such as :



DBSCAN Density data

These data points are clustered by using the basic concept that the data point lies within the given constraint from the cluster center. Various distance methods and techniques are used for calculation of the outliers.

Why Clustering ?

Clustering is very much important as it determines the intrinsic grouping among the unlabeled data present. There are no criteria for a good clustering. It depends on the user, what is the criteria they may use which satisfy their need. For instance, we could be interested in finding representatives for homogeneous groups (data reduction), in finding "natural clusters" and describe their unknown properties ("natural" data types), in finding useful and suitable groupings ("useful" data classes) or in finding unusual data objects (outlier detection). This algorithm must make some assumptions which constitute the similarity of points and each assumption make different and equally valid clusters.

Clustering Methods :

1. Density-Based Methods : These methods consider the clusters as the dense region having some similarity and different from the lower dense region of the space. These methods have good accuracy and ability to merge two clusters.Example *DBSCAN (Density-Based Spatial Clustering of Applications with Noise)* , *OPTICS (Ordering Points to Identify Clustering Structure)* etc.

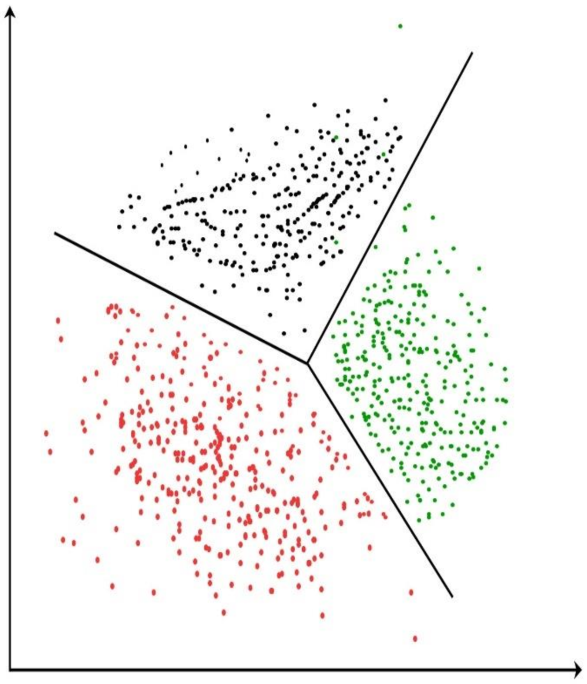
2. Hierarchical Based Methods : The clusters formed in this method forms a tree type structure based on the hierarchy. New clusters are formed using the previously formed one. It is divided into two category
-> Agglomerative (*bottom up approach*)
-> Divisive (*top down approach*) .
examples *CURE (Clustering Using Representatives), BIRCH (Balanced Iterative Reducing Clustering and using Hierarchies)* etc.

3. Partitioning Methods : These methods partition the objects into k clusters and each partition forms one cluster. This method is used to optimize an objective criterion similarity function such as when the distance is a major parameter example *K-means, CLARANS (Clustering Large Applications based upon randomized Search)* etc.

4. Grid-based Methods : In this method the data space are formulated into a finite number of cells that form a grid-like structure. All the clustering operation done on these grids are fast and independent of the number of data objects example *STING (Statistical Information Grid), wave cluster, CLIQUE (CLustering In Quest)* etc.

Clustering Algorithms :

K-means clustering algorithm – It is the simplest unsupervised learning algorithm that solves clustering problem.K-means algorithm partition n observations into k clusters where each observation belongs to the cluster with the nearest mean serving as a prototype of the cluster .



Applications of Clustering in different fields

1. **Marketing :** It can be used to characterize & discover customer segments for marketing purposes.
2. **Biology :** It can be used for classification among different species of plants and animals.
3. **Libraries :** It is used in clustering different books on the basis of topics and information.
4. **Insurance :** It is used to acknowledge the customers, their policies and identifying the frauds.
5. **City Planning :** It is used to make groups of houses and to study their values based on their geographical locations and other factors present.
6. **Earthquake studies :** By learning the earthquake affected areas we can determine the dangerous zones.

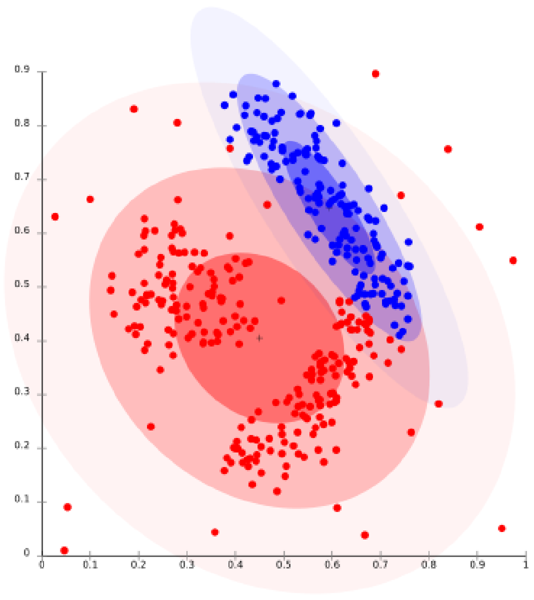
Different Types of Clustering Algorithm

The introduction to clustering is discussed in this article ans is advised to be understood first.

The clustering Algorithms are of many types. The following overview will only list the most prominent examples of clustering algorithms, as there are possibly over 100 published clustering algorithms. Not all provide models for their clusters and can thus not easily be categorized.

Distribution based methods

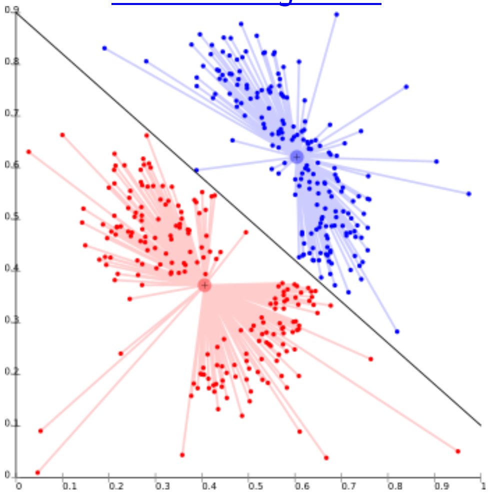
It is a clustering model in which we will fit the data on the probability that how it may belong to the same distribution. The grouping done may be *normal or gaussian* . Gaussian distribution is more prominent where we have fixed number of distributions and all the upcoming data is fitted into it such that the distribution of data may get maximized . This result in grouping which is shown in figure:-



This model works good on synthetic data and diversely sized clusters. But this model may have problem if the constraints are not used to limit model's complexity. Furthermore, Distribution-based clustering produces clusters which assume concisely defined mathematical models underlying the data, a rather strong assumption for some data distributions.
For Ex- *Expectation-maximization algorithm* which uses multivariate normal distributions is one of popular example of this algorithm .

Centroid based methods

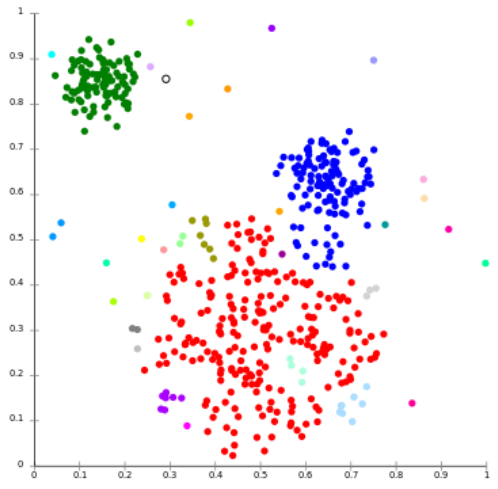
This is basically one of iterative clustering algorithm in which the clusters are formed by the closeness of data points to the *centroid* of clusters. Here , the cluster center i.e. *centroid* is formed such that the distance of data points is minimum with the center. This problem is basically one of NP- Hard problem and thus solutions are commonly approximated over a number of trials.
For Ex- *K – means algorithm* is one of popular example of this algorithm .



The biggest problem with this algorithm is that we need to specify K in advance. It also has problem in clustering density based distributions.

Connectivity based methods

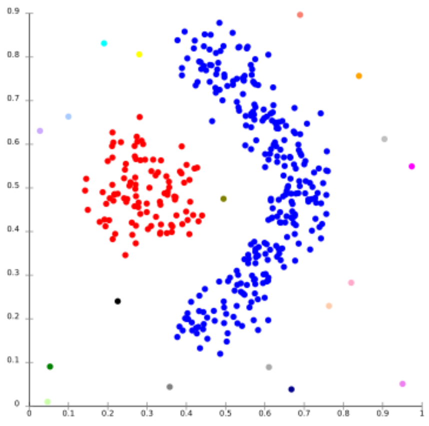
The core idea of connectivity based model is similar to Centroid based model which is basically defining clusters on the basis of closeness of data points .Here we work on a notion that the data points which are closer have similar behavior as compared to data points that are farther .
It is not a single partitioning of the data set , instead it provides an extensive hierarchy of clusters that merge with each other at certain distances. Here the choice of distance function is subjective. These models are very easy to interpret but it lacks scalability .



For Ex- *hierarchical algorithm* and it’s variants .

Density Models

In this clustering model there will be a searching of data space for areas of varied density of data points in the data space . It isolates various density regions based on different densities present in the data space .



For Ex- *DBSCAN and OPTI* *CS.*

Subspace clustering

Subspace clustering is an unsupervised learning problem that aims at grouping data points into multiple clusters so that data point at single cluster lie approximately on a low-dimensional linear subspace. Subspace clustering is an extension of feature selection just as with feature selection subspace clustering requires a search method and evaluation criteria but in addition subspace clustering limit the scope of evaluation criteria. Subspace clustering algorithm localize the search for relevant dimension and allow to them to find cluster that exist in multiple overlapping subspaces. Subspace clustering was originally purpose to solved very specific computer vision problem having a union of subspace structure in the data but it gains increasing attention in the statistic and machine learning community. People use this tool in social network, movie recommendation, and biological dataset. Subspace clustering raise the concern of data privacy as many such application involve dealing with sensitive information. Data points are assumed to be incoherentas it only protects the differential privacy of any feature of a user rather than the entire profile user of the database.
There are two branches of subspace clustering based on their search strategy.

- Top-down algorithms find an initial clustering in the full set of dimension and evaluate the subspace of each cluster.
- Bottom-up approach finds dense region in low dimensional space then combine to form clusters.

K means Clustering – Introduction

We are given a data set of items, with certain features, and values for these features (like a vector). The task is to categorize those items into groups. To achieve this, we will use the kMeans algorithm; an unsupervised learning algorithm.

Overview

(It will help if you think of items as points in an n-dimensional space). The algorithm will categorize the items into k groups of similarity. To calculate that similarity, we will use the euclidean distance as measurement.

The algorithm works as follows:

1. First we initialize k points, called means, randomly.
2. We categorize each item to its closest mean and we update the mean’s coordinates, which are the averages of the items categorized in that mean so far.
3. We repeat the process for a given number of iterations and at the end, we have our clusters.

The “points” mentioned above are called means, because they hold the mean values of the items categorized in it. To initialize these means, we have a lot of options. An intuitive method is to initialize the means at random items in the data set. Another method is to initialize the means at random values between the boundaries of the data set (if for a feature *x* the items have values in [0,3], we will initialize the means with values for *x* at [0,3]).

The above algorithm in pseudocode:

```
Initialize k means with random values

For a given number of iterations:
    Iterate through items:
        Find the mean closest to the item
        Assign item to mean
        Update mean
```

Read Data

We receive input as a text file ('data.txt'). Each line represents an item, and it contains numerical values (one for each feature) split by commas. You can find a sample data set here.

We will read the data from the file, saving it into a list. Each element of the list is another list containing the item values for the features. We do this with the following function:

```
def ReadData(fileName):

    # Read the file, splitting by lines
    f = open(fileName, 'r');
    lines = f.read().splitlines();
    f.close();

    items = [];

    for i in range(1, len(lines)):
        line = lines[i].split(',');
        itemFeatures = [];

        for j in range(len(line)-1):
            v = float(line[j]); # Convert feature value to float
            itemFeatures.append(v); # Add feature value to dict

        items.append(itemFeatures);

    shuffle(items);

    return items;
```

Initialize Means

We want to initialize each mean's values in the range of the feature values of the items. For that, we need to find the min and max for each feature. We accomplish that with the following function:

```
def FindColMinMax(items):
    n = len(items[0]);
    minima = [sys.maxint for i in range(n)];
    maxima = [-sys.maxint -1 for i in range(n)];

    for item in items:
        for f in range(len(item)):
            if (item[f] < minima[f]):
                minima[f] = item[f];

            if (item[f] > maxima[f]):
                maxima[f] = item[f];

return minima,maxima;
```

The variables *minima, maxima* are lists containing the min and max values of the items respectively. We initialize each mean's feature values randomly between the corresponding minimum and maximum in those above two lists:

```
def InitializeMeans(items, k, cMin, cMax):

    # Initialize means to random numbers between
    # the min and max of each column/feature
    f = len(items[0]); # number of features
    means = [[0 for i in range(f)] for j in range(k)];

    for mean in means:
        for i in range(len(mean)):

            # Set value to a random float
            # (adding +-1 to avoid a wide placement of a mean)
            mean[i] = uniform(cMin[i]+1, cMax[i]-1);

    return means;
```

Euclidean Distance

We will be using the euclidean distance as a metric of similarity for our data set (note: depending on your items, you can use another similarity metric).

```
def EuclideanDistance(x, y):
    S = 0; #  The sum of the squared differences of the elements
    for i in range(len(x)):
        S += math.pow(x[i]-y[i], 2);

    return math.sqrt(S); #The square root of the sum
```

Update Means

To update a mean, we need to find the average value for its feature, for all the items in the mean/cluster. We can do this by adding all the values and then dividing by the number of items, or we can use a more elegant solution. We will calculate the new average without having to re-add all the values, by doing the following:

```
m = (m*(n-1)+x)/n
```

where *m* is the mean value for a feature, *n* is the number of items in the cluster and *x* is the feature value for the added item. We do the above for each feature to get the new mean.

```
def UpdateMean(n,mean,item):
    for i in range(len(mean)):
        m = mean[i];
        m = (m*(n-1)+item[i])/float(n);
        mean[i] = round(m, 3);

    return mean;
```

Classify Items

Now we need to write a function to classify an item to a group/cluster. For the given item, we will find its similarity to each mean, and we will classify the item to the closest one.

```
def Classify(means,item):

    # Classify item to the mean with minimum distance
    minimum = sys.maxint;
    index = -1;

    for i in range(len(means)):

        # Find distance from item to mean
        dis = EuclideanDistance(item, means[i]);

        if (dis < minimum):
            minimum = dis;
            index = i;

    return index;
```

Find Means

To actually find the means, we will loop through all the items, classify them to their nearest cluster and update the cluster’s mean. We will repeat the process for some fixed number of iterations. If between two iterations no item changes classification, we stop the process as the algorithm has found the optimal solution.

The below function takes as input *k* (the number of desired clusters), the items and the number of maximum iterations, and returns the means and the clusters. The classification of an item is stored in the array *belongsTo* and the number of items in a cluster is stored in *clusterSizes*.

```
def CalculateMeans(k,items,maxIterations=100000):
    # Find the minima and maxima for columns
    cMin, cMax = FindColMinMax(items);

    # Initialize means at random points
    means = InitializeMeans(items,k,cMin,cMax);

    # Initialize clusters, the array to hold
    # the number of items in a class
    clusterSizes= [0 for i in range(len(means))];

    # An array to hold the cluster an item is in
    belongsTo = [0 for i in range(len(items))];

    # Calculate means
    for e in range(maxIterations):

        # If no change of cluster occurs, halt
        noChange = True;
        for i in range(len(items)):

            item = items[i];

            # Classify item into a cluster and update the
            # corresponding means.
            index = Classify(means,item);

            clusterSizes[index] += 1;
            cSize = clusterSizes[index];
            means[index] = UpdateMean(cSize,means[index],item);

            # Item changed cluster
            if(index != belongsTo[i]):
                noChange = False;

            belongsTo[i] = index;

        # Nothing changed, return
        if (noChange):
            break;

    return means;
```

Find Clusters

Finally we want to find the clusters, given the means. We will iterate through all the items and we will classify each item to its closest cluster.

```
def FindClusters(means,items):
    clusters = [[] for i in range(len(means))]; # Init clusters

    for item in items:
        # Classify item into a cluster
        index = Classify(means,item);

        # Add item to cluster
        clusters[index].append(item);

    return clusters;
```

Analysis of test data using K-Means Clustering in Python

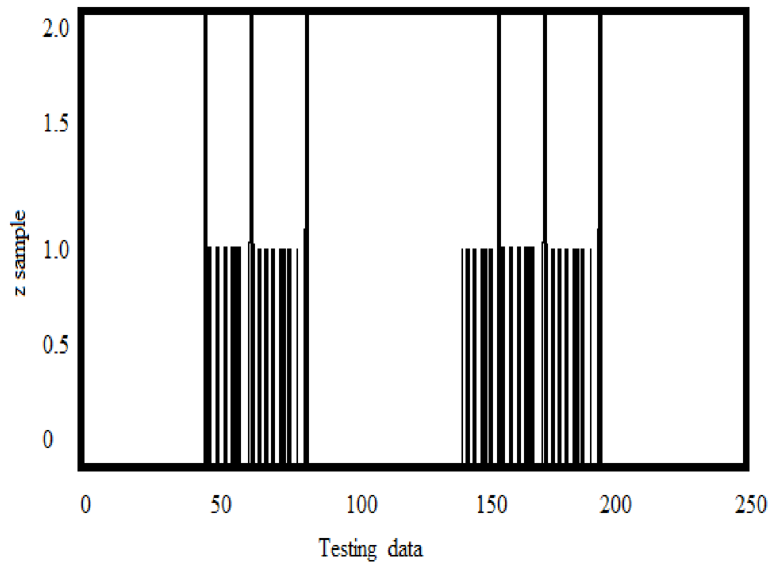
This article demonstrates an illustration of K-means clustering on a sample random data using open-cv library.

Pre-requisites: Numpy, OpenCV, matplot-lib
Let’s first visualize test data with Multiple Features using matplot-lib tool.

```
# importing required tools
import numpy as np
from matplotlib import pyplot as plt

# creating two test data
X = np.random.randint(10,35,(25,2))
Y = np.random.randint(55,70,(25,2))
Z = np.vstack((X,Y))
Z = Z.reshape((50,2))

# convert to np.float32
Z = np.float32(Z)

plt.xlabel('Test Data')
plt.ylabel('Z samples')

plt.hist(Z,256,[0,256])

plt.show()
```

Here ‘Z’ is an array of size 100, and values ranging from 0 to 255. Now, reshaped ‘z’ to a column vector. It will be more useful when more than one features are present. Then change the data to np.float32 type.

Output:



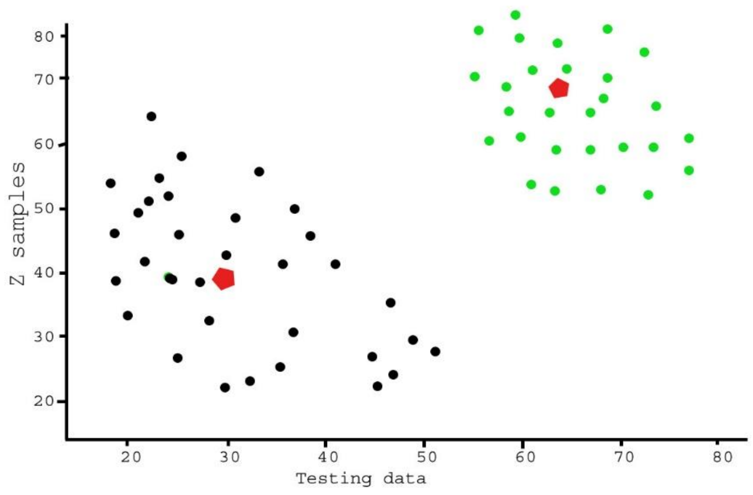
Now, apply the k-Means clustering algorithm to the same example as in the above test data and see its behavior.

Steps Involved:

1) First we need to set a test data.
2) Define criteria and apply kmeans().
3) Now separate the data.
4) Finally Plot the data.

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

X = np.random.randint(10,45,(25,2))
Y = np.random.randint(55,70,(25,2))
Z = np.vstack((X,Y))

# convert to np.float32
Z = np.float32(Z)

# define criteria and apply kmeans()
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
ret,label,center = cv2.kmeans(Z,2,None,criteria,10,cv2.KMEANS_RANDOM_CENTERS)

# Now separate the data
A = Z[label.ravel()==0]
B = Z[label.ravel()==1]

# Plot the data
plt.scatter(A[:,0],A[:,1])
plt.scatter(B[:,0],B[:,1],c = 'r')
plt.scatter(center[:,0],center[:,1],s = 80,c = 'y', marker = 's')
plt.xlabel('Test Data'),plt.ylabel('Z samples')
plt.show()
```

Output:



This example is meant to illustrate where k-means will produce intuitively possible clusters.

Applications:
1) Identifying Cancerous Data.
2) Prediction of Students' Academic Performance.
3) Drug Activity Prediction.

Gaussian Mixture Model

Suppose there are set of data points that needs to be grouped into several parts or clusters based on their similarity. In machine learning, this is known as Clustering.

There are several methods available for clustering like:

- K Means Clustering
- Hierarchical Clustering
- Gaussian Mixture Models

etc.

In this article, Gaussian Mixture Model will be discussed.

Normal or Gaussian Distribution

In real life, many datasets can be modeled by Gaussian Distribution (Univariate or Multivariate). So it is quite natural and intuitive to assume that the clusters come from different Gaussian Distributions. Or in other words, it is tried to model the dataset as a mixture of several Gaussian Distributions. This is the core idea of this model.

In one dimension the probability density function of a Gaussian Distribution is given by

$$G(X|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$$

where μ and σ^2 are respectively mean and variance of the distribution.

For Multivariate (let us say d-variate) Gaussian Distribution, the probability density function is given by

$$G(X|\mu,\Sigma) = \frac{1}{\sqrt{(2\pi)|\mathbf{\Sigma}|}} \exp\left(-\frac{1}{2}(X-\mu)^T\mathbf{\Sigma}^{-1}(X-\mu)\right)$$

Here μ is a d dimensional vector denoting the mean of the distribution and Σ is the d X d covariance matrix.

Gaussian Mixture Model

Suppose there are K clusters (For the sake of simplicity here it is assumed that the number of clusters is known and it is K). So μ and Σ is also estimated for each k. Had it been only one distribution, they would have been estimated by **maximum-likelihood method**. But since there are K such clusters and the probability density is defined as a linear function of densities of all these K distributions, i.e.

$$p(X) = \sum_{k=1}^{K} \pi_k G(X|\mu_k,\Sigma_k)$$

where π_k is the mixing coefficient for k-th distribution.

For estimating the parameters by maximum log-likelihood method, compute p(X|μ, Σ, π).

$$ln \; p(X|\mu,\Sigma,\pi) = \sum_{i=1}^{N} p(X_i) = \sum_{i=1}^{N} ln \sum_{k=1}^{K} \pi_k G(X_i|\mu_k,\Sigma_k)$$

Now define a random variable $\gamma_k(X)$ such that $\gamma_k(X)$=p(k|X).

From Bayes'theorem,

$$\gamma_k(X) = \frac{p(X|k)p(k)}{\sum_{k=1}^{K} p(k)p(X|k)} = \frac{p(X|k)\pi_k}{\sum_{k=1}^{K} \pi_k p(X|k)}$$

Now for the log likelihood function to be maximum, its derivative of $p(X|\mu,\Sigma,\pi)$ with respect to μ, Σ and π should be zero. So equaling the derivative of $p(X|\mu,\Sigma,\pi)$ with respect to μ to zero and rearranging the terms,

$$\mu_k = \frac{\sum_{n=1}^{N} \gamma_k(x_n)x_n}{\sum_{n=1}^{N} \gamma_k(x_n)}$$

Similarly taking derivative with respect to Σ and π respectively, one can obtain the following expressions.

$$\Sigma_k = \frac{\sum_{n=1}^{N} \gamma_k(x_n)(x_n-\mu_k)(x_n-\mu_k)^T}{\sum n=1^N \gamma_k(x_n)}$$

And

$$\pi_k = \frac{1}{N} \sum_{n=1}^{N} \gamma_k(x_n)$$

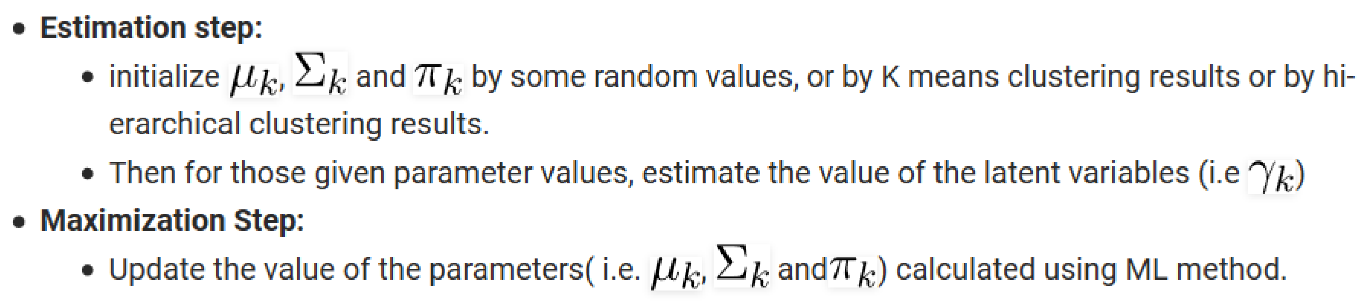
Note: $\sum_{n=1}^{N} \gamma_k(x_n)$ denotes the total number of sample points in the k-th cluster. Here it is assumed that there are total N number of samples and each sample containing d features are denoted by x_i.

So it can be clearly seen that the parameters cannot be estimated in closed form. This is where **Expectation-Maximization algorithm** is beneficial.

Expectation-Maximization (EM) Algorithm

The Expectation-Maximization (EM) algorithm is an iterative way to find maximum-likelihood estimates for model parameters when the data is incomplete or has some missing data points or has some hidden variables. EM chooses some random values for the missing data points and estimates a new set of data.

These new values are then recursively used to estimate a better first data, by filling up missing points, until the values get fixed.
These are the two basic steps of the EM algorithm, namely **E Step or Expectation Step or Estimation Step** and **M Step or Maximization Step**.



- **Estimation step:**
  - initialize μ_k, Σ_k and π_k by some random values, or by K means clustering results or by hierarchical clustering results.
  - Then for those given parameter values, estimate the value of the latent variables (i.e γ_k)
- **Maximization Step:**
  - Update the value of the parameters(i.e. μ_k, Σ_k andπ_k) calculated using ML method.

Algorithm:

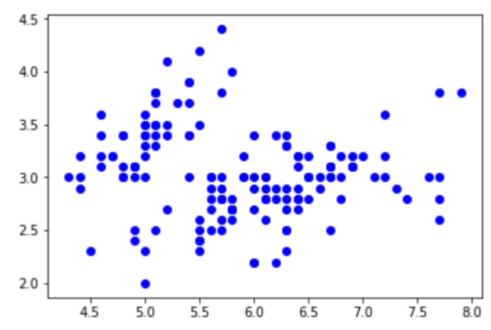
```
1. Initialize the mean    ,

2. the covariance matrix     and

3. the mixing coefficients
4. by some random values. (or other values)



5. Compute the     values for all k.



6. Again Estimate all the parameters

7. using the current     values.



8. Compute log-likelihood function.



9. Put some convergence criterion



10.  If the log-likelihood value converges to some value
11.  ( or if all the parameters converge to some values )
12.  then stop,
13.  else return to Step 2.
```

Example:
In this example, IRIS Dataset is taken. In Python there is a **GaussianMixture class** to implement GMM.

Note: This code might not run in an online compiler. Please use an offline ide.

1. Load the iris dataset from datasets package. To keep things simple, take only first two columns (i.e sepal length and sepal width respectively).
2. Now plot the dataset.

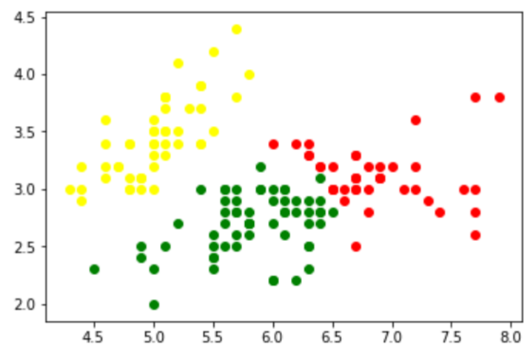
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pandas import DataFrame
from sklearn import datasets
from sklearn.mixture import GaussianMixture

# load the iris dataset
iris = datasets.load_iris()

# select first two columns
X = iris.data[:, :2]

# turn it into a dataframe
d = pd.DataFrame(X)

# plot the data
plt.scatter(d[0], d[1])
```



- Now fit the data as a mixture of 3 Gaussians.
- Then do the clustering, i.e assign a label to each observation. Also find the number of iterations needed for the log-likelihood function to converge and the converged log-likelihood value.

```
gmm = GaussianMixture(n_components = 3)

# Fit the GMM model for the dataset
# which expresses the dataset as a
# mixture of 3 Gaussian Distribution
gmm.fit(d)

# Assign a label to each sample
labels = gmm.predict(d)
d['labels']= labels
d0 = d[d['labels']== 0]
d1 = d[d['labels']== 1]
d2 = d[d['labels']== 2]
```

```
# plot three clusters in same plot
plt.scatter(d0[0], d0[1], c ='r')
plt.scatter(d1[0], d1[1], c ='yellow')
plt.scatter(d2[0], d2[1], c ='g')
```



- Print the converged log-likelihood value and no. of iterations needed for the model to converge

```
# print the converged log-likelihood value
print(gmm.lower_bound_)

# print the number of iterations needed
# for the log-likelihood value to converge
print(gmm.n_iter_)</div>
```

-

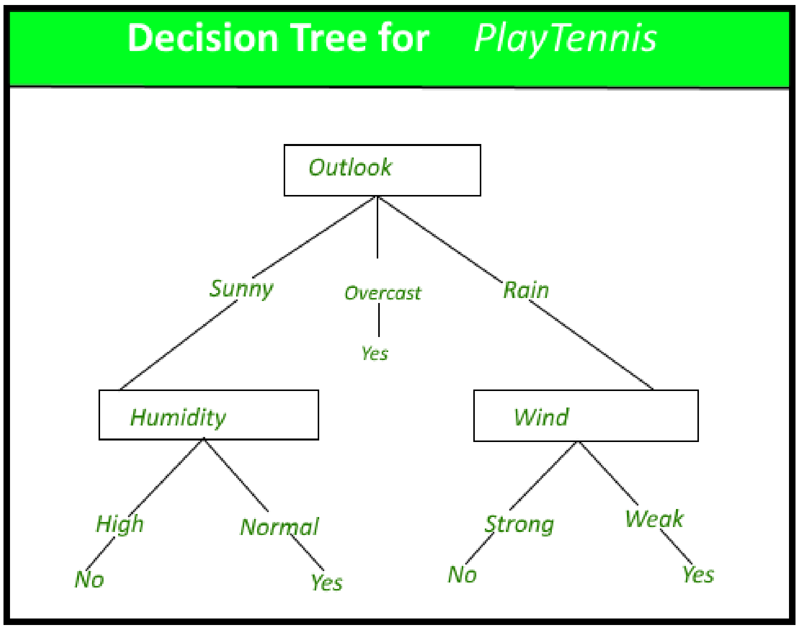
```
-1.4991548641719719
7
```

- Hence, it needed **7 iterations** for the log-likelihood to converge. If more iterations are performed, no appreciable change in the log-likelihood value, can be observed.

Non-parametric Methods :

Decision Tree

Decision Tree : Decision tree is the most powerful and popular tool for classification and prediction. A Decision tree is a flowchart like tree structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and eac



h leaf node (terminal node) holds a class label.

A decision tree for the concept PlayTennis.

Construction of Decision Tree :
A tree can be *"learned"* by splitting the source set into subsets based on an attribute value test. This process is repeated on each derived subset in a recursive manner called *recursive partitioning*. The recursion is completed when the subset at a node all has the same value of the target variable, or when splitting no longer adds value to the predictions. The construction of decision tree classifier does not require any domain knowledge or parameter setting, and therefore is appropriate for exploratory knowledge discovery. Decision trees can handle high dimensional data. In general decision tree classifier has good accuracy. Decision tree induction is a typical inductive approach to learn knowledge on classification.

Decision Tree Representation :
Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. An instance is classified by starting at the root node of the

tree,testing the attribute specified by this node,then moving down the tree branch corresponding to the value of the attribute as shown in the above figure.This process is then repeated for the subtree rooted at the new node.

The decision tree in above figure classifies a particular morning according to whether it is suitable for playing tennis and returning the classification associated with the particular leaf.(in this case Yes or No). For example,the instance

(Outlook = Rain, Temperature = Hot, Humidity = High, Wind = Strong)

would be sorted down the leftmost branch of this decision tree and would therefore be classified as a negative instance.

In other words we can say that decision tree represent a disjunction of conjunctions of constraints on the attribute values of instances.

(Outlook = Sunny ^ Humidity = Normal) v (Outllok = Overcast) v (Outlook = Rain ^ Wind = Weak)

Strengths and Weakness of Decision Tree approach
The strengths of decision tree methods are:

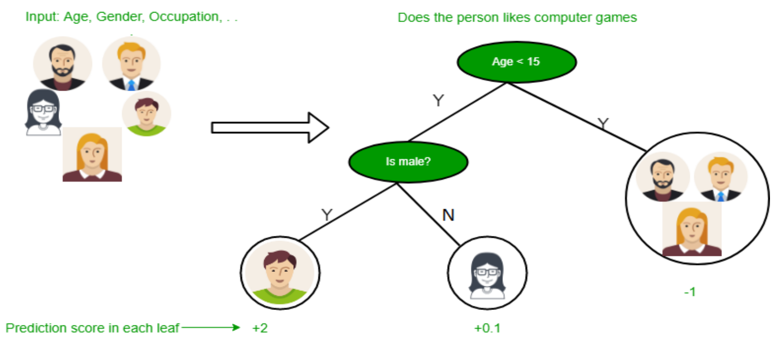
- Decision trees are able to generate understandable rules.
- Decision trees perform classification without requiring much computation.
- Decision trees are able to handle both continuous and categorical variables.
- Decision trees provide a clear indication of which fields are most important for prediction or classification.

The weaknesses of decision tree methods :

- Decision trees are less appropriate for estimation tasks where the goal is to predict the value of a continuous attribute.
- Decision trees are prone to errors in classification problems with many class and relatively small number of training examples.
- Decision tree can be computationally expensive to train. The process of growing a decision tree is computationally expensive. At each node, each candidate splitting field must be sorted before its best split can be found. In some algorithms, combinations of fields are used and a search must be made for optimal combining weights. Pruning algorithms can also be expensive since many candidate sub-trees must be formed and compared.

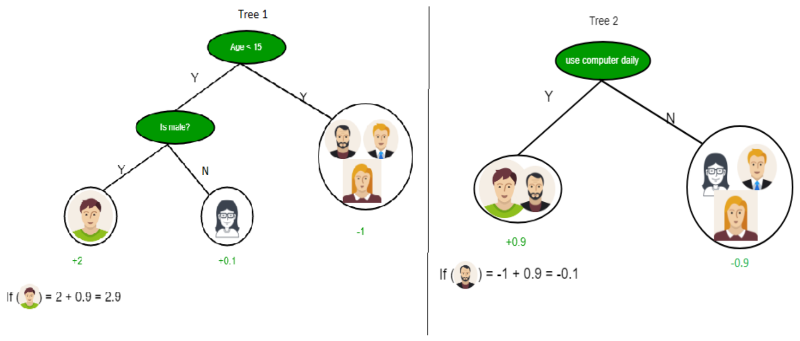
Decision Tree Introduction with example

- Decision tree algorithm falls under the category of the supervised learning. They can be used to solve both regression and classification problems.
- Decision tree uses the tree representation to solve the problem in which each leaf node corresponds to a class label and attributes are represented on the internal node of the tree.
- We can represent any boolean function on discrete attributes using the decision tree.


Input: Age, Gender, Occupation, . .
Does the person likes computer games
Age < 15
Y
Y
Is male?
Y
N
-1
Prediction score in each leaf
+2
+0.1

Below are some assumptions that we made while using decision tree:

- At the beginning, we consider the whole training set as the root.
- Feature values are preferred to be categorical. If the values are continuous then they are discretized prior to building the model.
- On the basis of attribute values records are distributed recursively.
- We use statistical methods for ordering attributes as root or the internal node.



As you can see from the above image that Decision Tree works on the Sum of Product form which is also knnown as *Disjunctive Normal Form*. In the above image we are predicting the use of computer in daily life of the people.

In Decision Tree the major challenge is to identification of the attribute for the root node in each level. This process is known as attribute selection. We have two popular attribute selection measures:

1. Information Gain
2. Gini Index

1. **Information Gain**
When we use a node in a decision tree to partition the training instances into smaller subsets the entropy changes. Information gain is a measure of this change in entropy.
Definition: *Suppose S is a set of instances, A is an attribute, S_v is the subset of S with A = v, and Values (A) is the set of all possible values of A, then*

$$Gain(S,A) = Entropy(S) - \sum_{v\epsilon Values(A)} \frac{|S_v|}{|S|}.Entropy(S_v)$$

Entropy

Entropy is the measure of uncertainty of a random variable, it characterizes the impurity of an arbitrary collection of examples. The higher the entropy more the information content.

***Definition**: Suppose S is a set of instances, A is an attribute, S_v is the subset of S with A = v, and Values (A) is the set of all possible values of A, then*

$$Gain(S,A) = Entropy(S) - \sum_{v\epsilon Values(A)} \frac{|S_v|}{|S|}.Entropy(S_v)$$

Example:

```
For the set X = {a,a,a,b,b,b,b,b}
Total intances: 8
Instances of b: 5
Instances of a: 3
```

$$EntropyH(X) = -\left[\left(\frac{3}{8}\right) log_2\frac{3}{8} + \left(\frac{5}{8}\right) log_2\frac{5}{8}\right]$$

```
              = -[0.375 * (-1.415) + 0.625 * (-0.678)]
              =-(-0.53-0.424)
              = 0.954
```

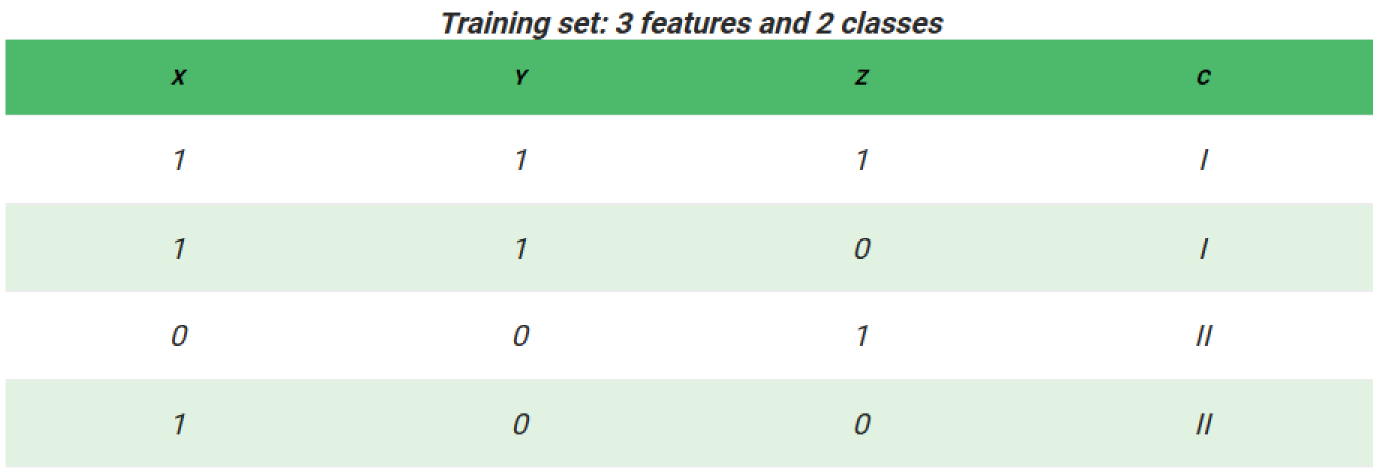
Building Decision Tree using Information Gain
The essentials:

- *Start with all training instances associated with the root node*
- *Use info gain to choose which attribute to label each node with*
- *Note: No root-to-leaf path should contain the same discrete attribute twice*
- *Recursively construct each subtree on the subset of training instances that would be classified down that path in the tree.*

The border cases:

- *If all positive or all negative training instances remain, label that node "yes" or "no" accordingly*
- *If no attributes remain, label with a majority vote of training instances left at that node*
- *If no instances remain, label with a majority vote of the parent's training instances*

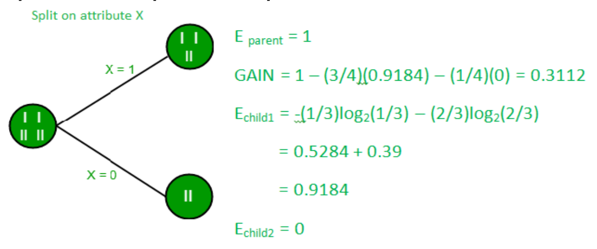
Example:
Now, lets draw a Decision Tree for the following data using Information gain.



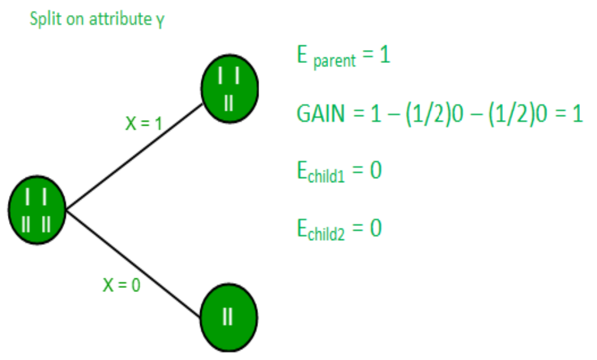
Training set: 3 features and 2 classes

X	Y	Z	C
1	1	1	I
1	1	0	I
0	0	1	II
1	0	0	II

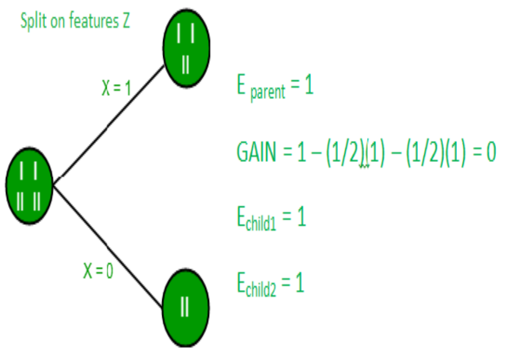
Here, we have 3 features and 2 output classes.
To build a decision tree using Information gain. We will take each of the feature and calculate the information for each feature.



Split on feature X



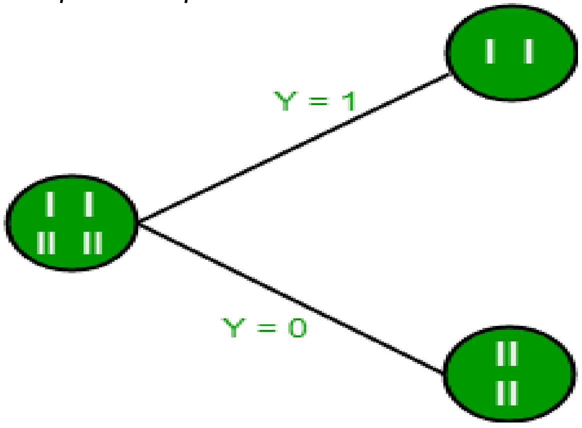
Split on feature Y



Split on feature Z

From the above images we can see that the information gain is maximum when we make a split on feature Y. So, for the root node best suited feature is feature Y. Now we can see that while spliting the dataset by feature Y, the child contains pure subset of the target variable. So we don't need to further split the dataset.

The final tree for the above dataset would be look like this:



2. Gini Index

- *Gini Index is a metric to measure how often a randomly chosen element would be incorrectly identified.*
- *It means an attribute with lower Gini index should be preferred.*
- *Sklearn supports "Gini" criteria for Gini Index and by default, it takes "gini" value.*
- *The Formula for the calculation of the of the Gini Index is given below.*

$$GiniIndex = 1 - \sum_j p_j^2$$

Example:
Lets consider the dataset in the image below and draw a decision tree using gini index.

Index	A	B	C	D	E
1	4.8	3.4	1.9	0.2	positive
2	5	3	1.6	1.2	positive
3	5	3.4	1.6	0.2	positive
4	5.2	3.5	1.5	0.2	positive
5	5.2	3.4	1.4	0.2	positive
6	4.7	3.2	1.6	0.2	positive
7	4.8	3.1	1.6	0.2	positive
8	5.4	3.4	1.5	0.4	positive
9	7	3.2	4.7	1.4	negative
10	6.4	3.2	4.7	1.5	negative
11	6.9	3.1	4.9	1.5	negative
12	5.5	2.3	4	1.3	negative
13	6.5	2.8	4.6	1.5	negative
14	5.7	2.8	4.5	1.3	negative
15	6.3	3.3	4.7	1.6	negative
16	4.9	2.4	3.3	1	negative

In the dataset above there are 5 attributes from which attribute E is the predicting feature which contains 2(Positive & Negitive) classes. We have equal proportion for both the classes.
In Gini Index, we have to choose some random values to categorize each attribute. These values for this dataset are:

```
   A        B         C         D
>= 5     >= 3.0    >= 4.2    >= 1.4
 < 5      < 3.0     < 4.2     < 1.4
```

Calculating Gini Index for Var A:
Value >= 5: 12

Attribute A >= 5 & class = positive: $\frac{5}{12}$

Attribute A >= 5 & class = negative: $\frac{7}{12}$

Gini(5, 7) = 1 – $\left[\left(\frac{5}{12}\right)^2 + \left(\frac{7}{12}\right)^2\right] = 0.4860$

Value < 5: 4

Attribute A < 5 & class = positive: $\frac{3}{4}$

Attribute A < 5 & class = negative: $\frac{1}{4}$

Gini(3, 1) = 1 – $\left[\left(\frac{3}{4}\right)^2 + \left(\frac{1}{4}\right)^2\right] = 0.375$

By adding weight and sum each of the gini indices:

$$gini(Target, A) = \left(\frac{12}{16}\right) * (0.486) + \left(\frac{4}{16}\right) * (0.375) = 0.45825$$

Calculating Gini Index for Var B:

Value >= 3: 12

Attribute B >= 3 & class = positive: $\frac{8}{12}$

Attribute B >= 5 & class = negative: $\frac{4}{12}$

Gini(5, 7) = 1 – $\left[\left(\frac{8}{12}\right)^2 + \left(\frac{4}{12}\right)^2\right] = 0.4460$

Value < 3: 4

Attribute A < 3 & class = positive: $\frac{0}{4}$

Attribute A < 3 & class = negative: $\frac{4}{4}$

Gini(3, 1) = 1 – $\left[\left(\frac{0}{4}\right)^2 + \left(\frac{4}{4}\right)^2\right] = 1$

By adding weight and sum each of the gini indices:

$$gini(Target, B) = \left(\frac{12}{16}\right) * (0.446) + \left(\frac{0}{16}\right) * (0) = 0.3345$$

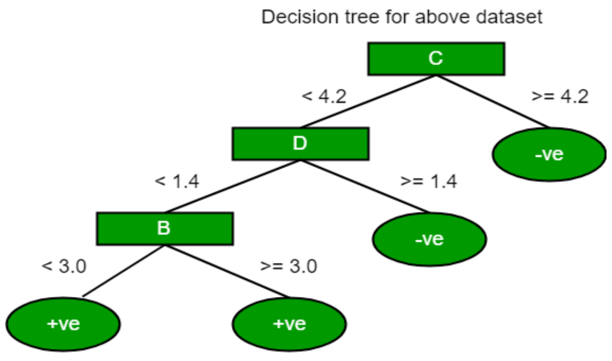
Using the same approach we can calculate the Gini index for C and D attributes.

```
              Positive    Negative
For A|>= 5.0     5          7
     |<5      3          1
Ginin Index of A = 0.45825
```

```
              Positive    Negative
For B|>= 3.0     8          4
     |< 3.0     0          4
Gini Index of B= 0.3345
```

```
              Positive    Negative
For C|>= 4.2     0          6
     |< 4.2     8          2
Gini Index of C= 0.2
```

```
              Positive    Negative
For D|>= 1.4     0          5
     |< 1.4     8          3
Gini Index of D= 0.273
```



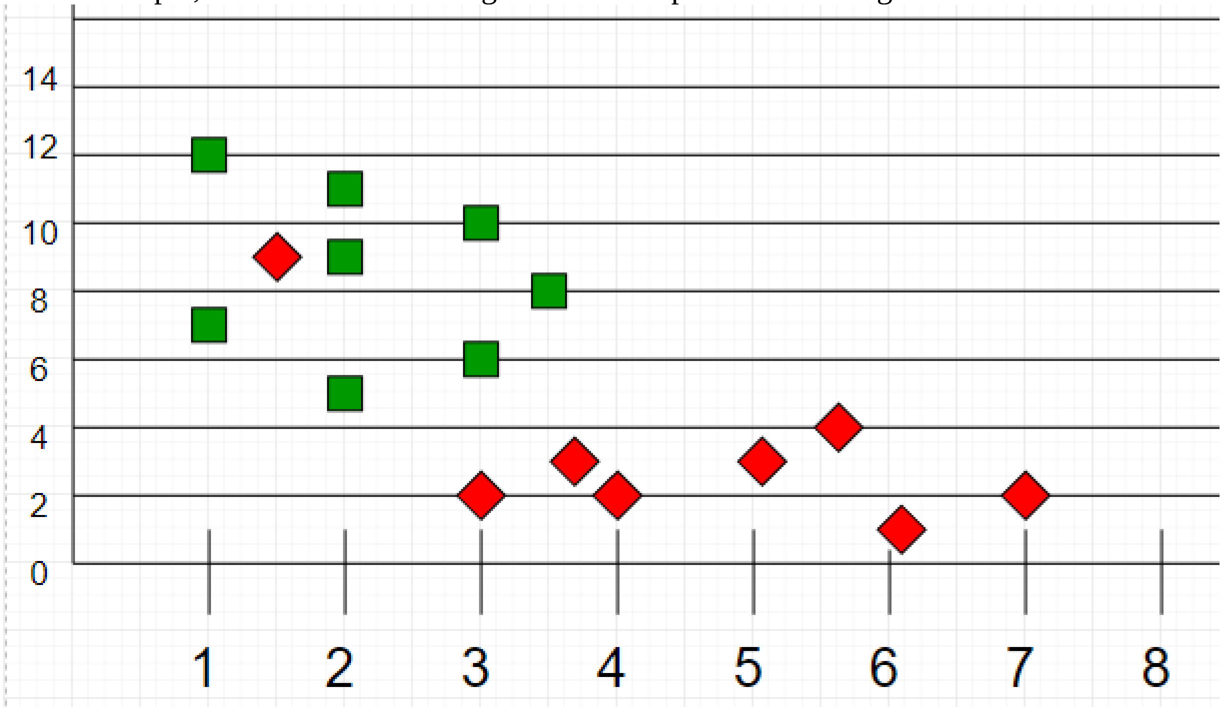
K-Nearest Neighbours

K-Nearest Neighbors is one of the most basic yet essential classification algorithms in Machine Learning. It belongs to the supervised learning domain and finds intense application in pattern recognition, data mining and intrusion detection.

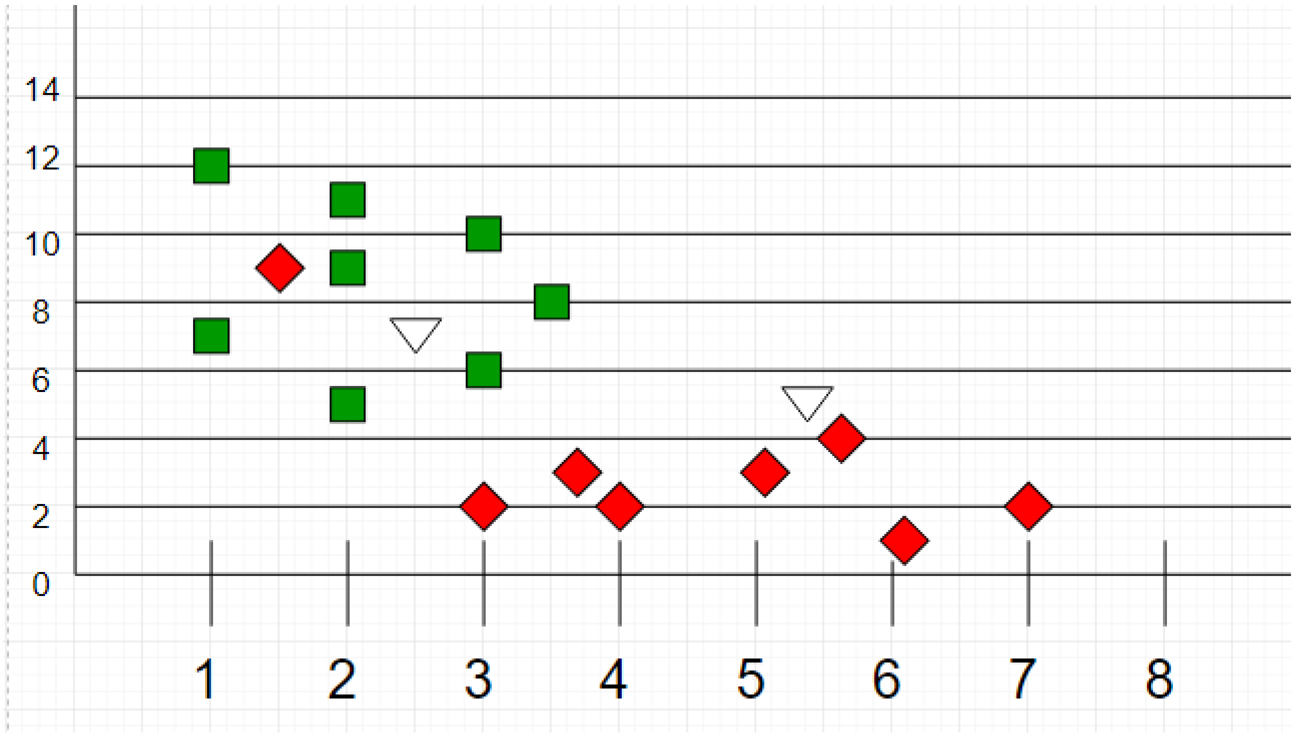
It is widely disposable in real-life scenarios since it is non-parametric, meaning, it does not make any underlying assumptions about the distribution of data (as opposed to other algorithms such as GMM, which assume a Gaussian distribution of the given data).

We are given some prior data (also called training data), which classifies coordinates into groups identified by an attribute.

As an example, consider the following table of data points containing two features:



Now, given another set of data points (also called testing data), allocate these points a group by analyzing the training set. Note that the unclassified points are marked as 'White'.



Intuition
If we plot these points on a graph, we may be able to locate some clusters, or groups. Now, given an unclassified point, we can assign it to a group by observing what group its nearest neighbours belong to. This means, a point close to a cluster of points classified as 'Red' has a higher probability of getting classified as 'Red'.

Intuitively, we can see that the first point (2.5, 7) should be classified as 'Green' and the second point (5.5, 4.5) should be classified as 'Red'.

Algorithm
Let m be the number of training data samples. Let p be an unknown point.

1. Store the training samples in an array of data points arr[]. This means each element of this array represents a tuple (x, y).
2. `for i=0 to m:`
3. `  Calculate Euclidean distance d(arr[i], p).`
4. Make set S of K smallest distances obtained. Each of these distances correspond to an already classified data point.
5. Return the majority label among S.
6. K can be kept as an odd number so that we can calculate a clear majority in the case where only two groups are possible (e.g. Red/Blue). With increasing K, we get smoother, more defined boundaries across different classifications. Also, the accuracy of the above classifier increases as we increase the number of data points in the training set.
7. **Example Program**
   Assume 0 and 1 as the two classifiers (groups).

```
# Python3 program to find groups of unknown
# Points using K nearest neighbour algorithm.

import math

def classifyAPoint(points,p,k=3):
    '''
     This function finds classification of p using
     k nearest neighbour algorithm. It assumes only two
     groups and returns 0 if p belongs to group 0, else
      1 (belongs to group 1).

      Parameters -
          points : Dictionary of training points having two keys - 0 and 1
                   Each key have a list of training data points belong to that

          p : A touple ,test data point of form (x,y)

          k : number of nearest neighbour to consider, default is 3
    '''

    distance=[]
    for group in points:
        for feature in points[group]:

            #calculate the euclidean distance of p from training points
```

```
                euclidean_distance = math.sqrt((feature[0]-p[0])**2 
        +(feature[1]-p[1])**2)

            # Add a touple of form (distance,group) in the distance list
            distance.append((euclidean_distance,group))

    # sort the distance list in ascending order
    # and select first k distances
    distance = sorted(distance)[:k]

    freq1 = 0 #frequency of group 0
    freq2 = 0 #frequency og group 1

    for d in distance:
        if d[1] == 0:
            freq1 += 1
        elif d[1] == 1:
            freq2 += 1

    return 0 if freq1>freq2 else 1

# driver function
def main():

    # Dictionary of training points having two keys - 0 and 1
    # key 0 have points belong to class 0
    # key 1 have points belong to class 1

    points = {0:[(1,12),(2,5),(3,6),(3,10),(3.5,8),(2,11),(2,9),(1,7)],
              1:[(5,3),(3,2),(1.5,9),(7,2),(6,1),(3.8,1),(5.6,4),(4,2),(2,5)]}

    # testing point p(x,y)
    p = (2.5,7)

    # Number of neighbours
    k = 3

    print("The value classified to unknown point is: {}".\
          format(classifyAPoint(points,p,k)))

if __name__ == '__main__':
    main()

# This code is contributed by Atul Kumar (www.fb.com/atul.kr.007)
```

Output:

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

Implementation of K Nearest Neighbors

Prerequisite : K nearest neighbours

Introduction

Say we are given a data set of items, each having numerically valued features (like Height, Weight, Age, etc). If the count of features is *n*, we can represent the items as points in an *n*-dimensional grid. Given a new item, we can calculate the distance from the item to every other item in the set. We pick the *k* closest neighbors and we see where most of these neighbors are classified in. We classify the new item there.

So the problem becomes **how we can calculate the distances between items.** The solution to this depends on the data set. If the values are real we usually use the Euclidean distance. If the values are categorical or binary, we usually use the Hamming distance.

Algorithm:

```
Given a new item:
    1. Find distances between new item and all other items
    2. Pick k shorter distances
    3. Pick the most common class in these k distances
    4. That class is where we will classify the new item
```

Reading Data

Let our input file be in the following format:

```
Height, Weight, Age, Class
1.70, 65, 20, Programmer
1.90, 85, 33, Builder
1.78, 76, 31, Builder
1.73, 74, 24, Programmer
1.81, 75, 35, Builder
1.73, 70, 75, Scientist
1.80, 71, 63, Scientist
1.75, 69, 25, Programmer
```

Each item is a line and under "Class" we see where the item is classified in. The values under the feature names ("Height" etc.) is the value the item has for that feature. All the values and features are separated by commas.

Place these data files in the working directory data2 and data. Choose one and paste the contents as is into a text file named *data*.

We will read from the file (named "data.txt") and we will split the input by lines:

```
f = open('data.txt', 'r');
lines = f.read().splitlines();
f.close();
```

The first line of the file holds the feature names, with the keyword "Class" at the end. We want to store the feature names into a list:

```
# Split the first line by commas,
# remove the first element and
# save the rest into a list. The
# list now holds the feature
# names of the data set.
features = lines[0].split(', ')[:-1];
```

Then we move onto the data set itself. We will save the items into a list, named *items*, whose elements are dictionaries (one for each item). The keys to these item-dictionaries are the feature names, plus "Class" to hold the item class. At the end, we want to shuffle the items in the list (this is a safety measure, in case the items are in a weird order).

```
items = [];

for i in range(1, len(lines)):

    line = lines[i].split(', ');

    itemFeatures = {"Class" : line[-1]};

    # Iterate through the features
    for j in range(len(features)):

        # Get the feature at index j
        f = features[j];

        # The first item in the line
        # is the class, skip it
        v = float(line[j]);

        # Add feature to dict
        itemFeatures[f] = v;

    # Append temp dict to items
    items.append(itemFeatures);

shuffle(items);
```

Classifying the data

With the data stored into *items*, we now start building our classifier. For the classifier, we will create a new function, *Classify*. It will take as input the item we want to classify, the items list and *k*, the number of the closest neighbors.

If *k* is greater than the length of the data set, we do not go ahead with the classifying, as we cannot have more closest neighbors than the total amount of items in the data set. (alternatively we could set k as the *items* length instead of returning an error message)

```
if(k > len(Items)):

        # k is larger than list
        # length, abort
        return "k larger than list length";
```

We want to calculate the distance between the item to be classified and all the items in the training set, in the end keeping the *k* shortest distances. To keep the current closest neighbors we use a list, called *neighbors*. Each element in the least holds two values, one for the distance from the item to be classified and another for the class the neighbor is in. We will calculate distance via the generalized Euclidean formula (for *n* dimensions). Then, we will pick the class that appears most of the time in *neighbors* and that will be our pick. In code:

```
def Classify(nItem, k, Items):
    if(k > len(Items)):

        # k is larger than list
        # length, abort
        return "k larger than list length";

    # Hold nearest neighbors.
    # First item is distance,
    # second class
    neighbors = [];

    for item in Items:

        # Find Euclidean Distance
        distance = EuclideanDistance(nItem, item);

        # Update neighbors, either adding
        # the current item in neighbors
        # or not.
        neighbors = UpdateNeighbors(neighbors, item, distance, k);

    # Count the number of each
    # class in neighbors
    count = CalculateNeighborsClass(neighbors, k);

    # Find the max in count, aka the
    # class with the most appearances.
    return FindMax(count);
```

The external functions we need to implement are *EuclideanDistance*, *UpdateNeighbors*, *CalculateNeighborsClass* and *FindMax*.

Finding Euclidean Distance

The generalized Euclidean formula for two vectors x and y is this:

```
distance = sqrt{(x_{1}-y_{1})^2 + (x_{2}-y_{2})^2 + ... + (x_{n}-y_{n})^2}
```

In code:

```
def EuclideanDistance(x, y):

    # The sum of the squared
    # differences of the elements
    S = 0;
    for key in x.keys():
        S += math.pow(x[key]-y[key], 2);

    # The square root of the sum
    return math.sqrt(S);
```

Updating Neighbors

We have our neighbors list (which should at most have a length of *k*) and we want to add an item to the list with a given distance. First we will check if *neighbors* has a length of *k*. If it has less, we add the item to it irregardless of the distance (as we need to fill the list up to *k* before we start rejecting items). If not, we will check if the item has a shorter distance than the item with the max distance in the list. If that is true, we will replace the item with max distance with the new item.

To find the max distance item more quickly, we will keep the list sorted in ascending order. So, the last item in the list will have the max distance. We will replace it with the new item and we will sort again.

To speed this process up, we can implement an Insertion Sort where we insert new items in the list without having to sort the entire list. The code for this though is rather long and, although simple, will bog the tutorial down.

```
def UpdateNeighbors(neighbors, item, distance, k):

    if(len(neighbors) > distance):

            # If yes, replace the last
            # element with new item
            neighbors[-1] = [distance, item["Class"]];
            neighbors = sorted(neighbors);

    return neighbors;
```

CalculateNeighborsClass

Here we will calculate the class that appears most often in *neighbors*. For that, we will use another dictionary, called *count*, where the keys are the class names appearing in *neighbors*. If a key doesn’t exist, we will add it, otherwise we will increment its value.

```
def CalculateNeighborsClass(neighbors, k):
    count = {};

    for i in range(k):

        if(neighbors[i][1] not in count):

            # The class at the ith index
            # is not in the count dict.
            # Initialize it to 1.
            count[neighbors[i][1]] = 1;
        else:

            # Found another item of class
            # c[i]. Increment its counter.
            count[neighbors[i][1]] += 1;

    return count;
```

FindMax

We will input to this function the dictionary *count* we build in *CalculateNeighborsClass* and we will return its max.

```
def FindMax(countList):

    # Hold the max
    maximum = -1;

    # Hold the classification
    classification = "";

    for key in countList.keys():

        if(countList[key] > maximum):
            maximum = countList[key];
            classification = key;

    return classification, maximum;
```

Conclusion

With that this kNN tutorial is finished.

You can now classify new items, setting *k* as you see fit. Usually for *k* an odd number is used, but that is not necessary. To classify a new item, you need to create a dictionary with keys the feature names and the values that characterize the item. An example of classification:

```
newItem = {'Height' : 1.74, 'Weight' : 67, 'Age' : 22};
print Classify(newItem, 3, items);
```

The complete code of the above approach is given below:-

```
# Python Program to illustrate
# KNN algorithm

# For pow and sqrt
import math
from random import shuffle

###_Reading_### def ReadData(fileName):

    # Read the file, splitting by lines
    f = open(fileName, 'r')
    lines = f.read().splitlines()
    f.close()

    # Split the first line by commas,
    # remove the first element and save
    # the rest into a list. The list
    # holds the feature names of the
    # data set.
    features = lines[0].split(', ')[:-1]

    items = []

    for i in range(1, len(lines)):

        line = lines[i].split(', ')

        itemFeatures = {'Class': line[-1]}

        for j in range(len(features)):

            # Get the feature at index j
            f = features[j]

            # Convert feature value to float
            v = float(line[j])

             # Add feature value to dict
            itemFeatures[f] = v

        items.append(itemFeatures)

    shuffle(items)

    return items


###_Auxiliary Function_### def EuclideanDistance(x, y):

    # The sum of the squared differences
    # of the elements
    S = 0

    for key in x.keys():
        S += math.pow(x[key] - y[key], 2)

    # The square root of the sum
    return math.sqrt(S)

def CalculateNeighborsClass(neighbors, k):
    count = {}
    for i in range(k):
        if neighbors[i][1] not in count:

            # The class at the ith index is
            # not in the count dict.
            # Initialize it to 1.
            count[neighbors[i][1]] = 1
        else:

            # Found another item of class
            # c[i]. Increment its counter.
            count[neighbors[i][1]] += 1

    return count

def FindMax(Dict):

    # Find max in dictionary, return
    # max value and max index
    maximum = -1
    classification = ''

    for key in Dict.keys():

        if Dict[key] > maximum:
            maximum = Dict[key]
            classification = key

    return (classification, maximum)



###_Core Functions_### def Classify(nItem, k, Items):

    # Hold nearest neighbours. First item
    # is distance, second class
    neighbors = []

    for item in Items:

        # Find Euclidean Distance
        distance = EuclideanDistance(nItem, item)

        # Update neighbors, either adding the
        # current item in neighbors or not.
        neighbors = UpdateNeighbors(neighbors, item, distance, k)

    # Count the number of each class
    # in neighbors
    count = CalculateNeighborsClass(neighbors, k)

    # Find the max in count, aka the
    # class with the most appearances
    return FindMax(count)
```

```
def UpdateNeighbors(neighbors, item, distance,
                                          k, ):
    if len(neighbors) < k:

        # List is not full, add
        # new item and sort
        neighbors.append([distance, item['Class']])
        neighbors = sorted(neighbors)
    else:

        # List is full Check if new
        # item should be entered
        if neighbors[-1][0] > distance:

            # If yes, replace the
            # last element with new item
            neighbors[-1] = [distance, item['Class']]
            neighbors = sorted(neighbors)

    return neighbors

###_Evaluation Functions_### def K_FoldValidation(K, k, Items):

    if K > len(Items):
        return -1

    # The number of correct classifications
    correct = 0

    # The total number of classifications
    total = len(Items) * (K - 1)

    # The length of a fold
    l = int(len(Items) / K)

    for i  in range(K):

        # Split data into training set
        # and test set
        trainingSet = Items[i * l:(i + 1) * l]
        testSet = Items[:i * l] + Items[(i + 1) * l:]

        for item in testSet:
            itemClass = item['Class']

            itemFeatures = {}

            # Get feature values
            for key in item:
                if key != 'Class':

                    # If key isn't "Class", add
                    # it to itemFeatures
                    itemFeatures[key] = item[key]
            # Categorize item based on
            # its feature values
            guess = Classify(itemFeatures, k, trainingSet)[0]

            if guess == itemClass:

                # Guessed correctly
                correct += 1

    accuracy = correct / float(total)
    return accuracy


def Evaluate(K, k, items, iterations):

    # Run algorithm the number of
    # iterations, pick average
    accuracy = 0

    for i  in range(iterations):
        shuffle(items)
        accuracy += K_FoldValidation(K, k, items)

    print accuracy / float(iterations)


###_Main_### def main():
    items = ReadData('data.txt')

    Evaluate(5, 5, items, 100)

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

Output:

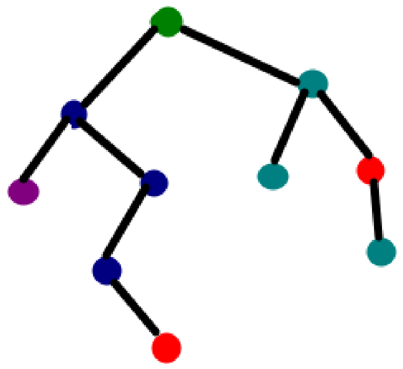
```
0.9375
```

The output can vary from machine to machine. The code includes a Fold Validation function, but it is unrelated to the algorithm, it is there for calculating the accuracy of the algorithm.

Decision tree implementation using Python

Prerequisites: Decision Tree, DecisionTreeClassifier, sklearn, numpy, pandas

Decision Tree is one of the most powerful and popular algorithm. Decision-tree algorithm falls under the category of supervised learning algorithms. It works for both continuous as well as categorical output variables.



In this article, We are going to implement a Decision tree algorithm on the *Balance Scale Weight & Distance Database* presented on the UCI.

Data-set Description :

```
Title          : Balance Scale Weight & Distance Database
Number of Instances: 625 (49 balanced, 288 left, 288 right)
Number of Attributes: 4 (numeric) + class name = 5
```
Attribute Information:

1. **Class Name (Target variable):** 3
   - L [balance scale tip to the left]
   - B [balance scale be balanced]
   - R [balance scale tip to the right]
2. **Left-Weight:** 5 (1, 2, 3, 4, 5)
3. **Left-Distance:** 5 (1, 2, 3, 4, 5)
4. **Right-Weight:** 5 (1, 2, 3, 4, 5)
5. **Right-Distance:** 5 (1, 2, 3, 4, 5)

Missing Attribute Values: None
Class Distribution:

1. 46.08 percent are L
2. 07.84 percent are B
3. 46.08 percent are R

You can find more details of the dataset here.

Used Python Packages :

1. **sklearn :**
   - In python, sklearn is a machine learning package which include a lot of ML algorithms.
   - Here, we are using some of its modules like train_test_split, DecisionTreeClassifier and accuracy_score.
2. **NumPy :**
   - It is a numeric python module which provides fast maths functions for calculations.

  - It is used to read data in numpy arrays and for manipulation purpose.

3. **Pandas :**
   - Used to read and write different files.
   - Data manipulation can be done easily with dataframes.

Installation of the packages :

In Python, sklearn is the package which contains all the required packages to implement Machine learning algorithm. You can install the sklearn package by following the commands given below.
using pip :

```
pip install -U scikit-learn
```

Before using the above command make sure you have *scipy* and *numpy* packages installed.

If you don't have pip. You can install it using

```
python get-pip.py
```

using conda :

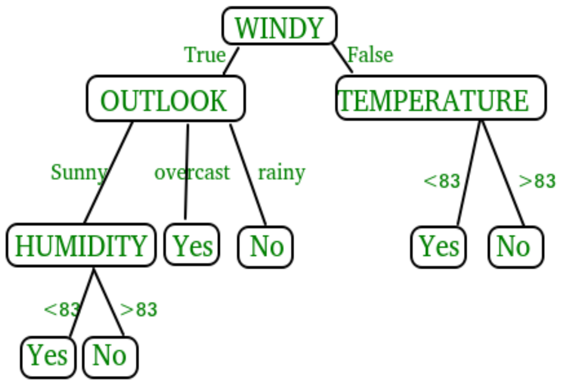
```
conda install scikit-learn
```

Assumptions we make while using Decision tree :

- At the beginning, we consider the whole training set as the root.
- Attributes are assumed to be categorical for information gain and for gini index, attributes are assumed to be continuous.
- On the basis of attribute values records are distributed recursively.
- We use statistical methods for ordering attributes as root or internal node.

Pseudocode :

```
1. Find the best attribute and place it on the root node of the tree.
2. Now, split the training set of the dataset into subsets. While making the subset make sure that each
   subset of training dataset should have the same value for an attribute.
3. Find leaf nodes in all branches by repeating 1 and 2 on each subset.
```



While implementing the decision tree we will go through the following two phases:

4. Building Phase
   - Preprocess the dataset.

  - Split the dataset from train and test using Python sklearn package.
  - Train the classifier.

5. Operational Phase
   - Make predictions.
   - Calculate the accuracy.

Data Import :

- To import and manipulate the data we are using the *pandas* package provided in python.
- Here, we are using a URL which is directly fetching the dataset from the UCI site no need to download the dataset. When you try to run this code on your system make sure the system should have an active Internet connection.
- As the dataset is separated by “,” so we have to pass the sep parameter’s value as “,”.
- Another thing is notice is that the dataset doesn’t contain the header so we will pass the Header parameter’s value as none. If we will not pass the header parameter then it will consider the first line of the dataset as the header.

Data Slicing :

- Before training the model we have to split the dataset into the training and testing dataset.
- To split the dataset for training and testing we are using the sklearn module *train_test_split*
- First of all we have to separate the target variable from the attributes in the dataset.

```
X = balance_data.values[:, 1:5]
Y = balance_data.values[:,0]
```

- Above are the lines from the code which sepearte the dataset. The variable X contains the attributes while the variable Y contains the target variable of the dataset.
- Next step is to split the dataset for training and testing purpose.

```
X_train, X_test, y_train, y_test = train_test_split(
          X, Y, test_size = 0.3, random_state = 100)
```

- Above line split the dataset for training and testing. As we are spliting the dataset in a ratio of 70:30 between training and testing so we are pass *test_size* parameter’s value as 0.3.
- *random_state* variable is a pseudo-random number generator state used for random sampling.

Terms used in code :

Gini index and information gain both of these methods are used to select from the *n* attributes of the dataset which attribute would be placed at the root node or the internal node.

Gini index

$$\text{Gini Index} = 1 - \sum_j P_j^2$$

- Gini Index is a metric to measure how often a randomly chosen element would be incorrectly identified.
- It means an attribute with lower gini index should be preferred.
- Sklearn supports “gini” criteria for Gini Index and by default, it takes “gini” value.

Entropy

if a random variable x can take N different value ,the i^{th} value x_i with probability $p(x_{ii})$,we can assocoate the folloeing entropy with x:

$$H(x) = -\sum_{i=1}^{N} p(x_i)\log_2 p(x_i)$$

- Entropy is the measure of uncertainty of a random variable, it characterizes the impurity of an arbitrary collection of examples. The higher the entropy the more the information content.

Information Gain

Definition: Suppose S is a set of instances, A is an attribute, S_v is the subset of s with A = v and Values(A) is the set of all possible of A,then

$$\text{Gain}(S,A) = \text{Entropy}(S) - \sum_{v\,:va(A)} \frac{|S_v|}{|S|}\,\text{Entropy}(S_v)$$

|S| denotes the size of set S

- The entropy typically changes when we use a node in a decision tree to partition the training instances into smaller subsets. Information gain is a measure of this change in entropy.
- Sklearn supports “entropy” criteria for Information Gain and if we want to use Information Gain method in sklearn then we have to mention it explicitly.

Accuracy score

- Accuracy score is used to calculate the accuracy of the trained classifier.

Confusion Matrix

- Confusion Matrix is used to understand the trained classifier behavior over the test dataset or validate dataset.

Below is the python code for the decision tree.

```
# Run this program on your local python
# interpreter, provided you have installed
# the required libraries.


# Importing the required packages
import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix
from sklearn.cross_validation import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
```

```
# Function importing Dataset
def importdata():
    balance_data = pd.read_csv(
'https://archive.ics.uci.edu/ml/machine-learning-'+
'databases/balance-scale/balance-scale.data',
    sep= ',', header = None)

    # Printing the dataswet shape
    print ("Dataset Lenght: ", len(balance_data))
    print ("Dataset Shape: ", balance_data.shape)

    # Printing the dataset obseravtions
    print ("Dataset: ",balance_data.head())
    return balance_data

# Function to split the dataset
def splitdataset(balance_data):

    # Seperating the target variable
    X = balance_data.values[:, 1:5]
    Y = balance_data.values[:, 0]

    # Spliting the dataset into train and test
    X_train, X_test, y_train, y_test = train_test_split(
    X, Y, test_size = 0.3, random_state = 100)

    return X, Y, X_train, X_test, y_train, y_test

# Function to perform training with giniIndex.
def train_using_gini(X_train, X_test, y_train):

    # Creating the classifier object
    clf_gini = DecisionTreeClassifier(criterion = "gini",
            random_state = 100,max_depth=3, min_samples_leaf=5)

    # Performing training
    clf_gini.fit(X_train, y_train)
    return clf_gini

# Function to perform training with entropy.
def tarin_using_entropy(X_train, X_test, y_train):

    # Decision tree with entropy
    clf_entropy = DecisionTreeClassifier(
            criterion = "entropy", random_state = 100,
            max_depth = 3, min_samples_leaf = 5)

    # Performing training
    clf_entropy.fit(X_train, y_train)
    return clf_entropy



# Function to make predictions
def prediction(X_test, clf_object):

    # Predicton on test with giniIndex
    y_pred = clf_object.predict(X_test)
    print("Predicted values:")
    print(y_pred)
    return y_pred

# Function to calculate accuracy
def cal_accuracy(y_test, y_pred):

    print("Confusion Matrix: ",
        confusion_matrix(y_test, y_pred))
    print ("Accuracy : ",
    accuracy_score(y_test,y_pred)*100)

    print("Report : ",
    classification_report(y_test, y_pred))

# Driver code
def main():

    # Building Phase
    data = importdata()
    X, Y, X_train, X_test, y_train, y_test = splitdataset(data)
    clf_gini = train_using_gini(X_train, X_test, y_train)
    clf_entropy = tarin_using_entropy(X_train, X_test, y_train)

    # Operational Phase
    print("Results Using Gini Index:")

    # Prediction using gini
    y_pred_gini = prediction(X_test, clf_gini)
    cal_accuracy(y_test, y_pred_gini)

    print("Results Using Entropy:")
    # Prediction using entropy
    y_pred_entropy = prediction(X_test, clf_entropy)
    cal_accuracy(y_test, y_pred_entropy)



# Calling main function
if __name__=="__main__":
    main()
```

Data Infomation:

```
Dataset Lenght:  625
Dataset Shape:  (625, 5)
Dataset:     0  1  2  3  4
0  B  1  1  1  1
1  R  1  1  1  2
2  R  1  1  1  3
3  R  1  1  1  4
4  R  1  1  1  5
```

Results Using Gini Index:

```
Predicted values:
['R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'R' 'L'
 'L' 'R' 'L' 'R' 'L' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'L'
 'L' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'R' 'L' 'R'
 'R' 'L' 'R' 'R' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'R'
 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'R' 'L'
 'R' 'R' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'R'
 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L'
 'L' 'L' 'L' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R'
 'L' 'L' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R'
 'L' 'L' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'R' 'R'
 'L' 'R' 'R' 'L' 'L' 'R' 'R' 'R']

Confusion Matrix:  [[ 0  6  7]
                    [ 0 67 18]
                    [ 0 19 71]]
Accuracy :  73.4042553191
Report :
       precision    recall  f1-score   support
  B       0.00      0.00      0.00        13
  L       0.73      0.79      0.76        85
  R       0.74      0.79      0.76        90
avg/total 0.68      0.73      0.71       188
```

Results Using Entropy:

```
Predicted values:
['R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'L'
 'L' 'R' 'L' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'L' 'L'
 'L' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'L' 'R' 'L' 'L' 'R' 'L' 'L'
 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'L' 'R' 'L' 'L' 'L' 'R'
 'R' 'L' 'R' 'L' 'R' 'R' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'R' 'R' 'L' 'R' 'L'
 'R' 'R' 'L' 'L' 'L' 'R' 'R' 'L' 'L' 'L' 'R' 'L' 'L' 'R' 'R' 'R' 'R' 'R'
 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L'
 'L' 'L' 'L' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'L' 'R'
 'L' 'R' 'R' 'L' 'L' 'R' 'L' 'R' 'R' 'R' 'R' 'R' 'L' 'R' 'R' 'R' 'R' 'R'
 'R' 'L' 'R' 'L' 'R' 'R' 'L' 'R' 'L' 'R' 'L' 'R' 'L' 'L' 'L' 'L' 'L' 'R'
 'R' 'R' 'L' 'L' 'L' 'R' 'R' 'R']

Confusion Matrix:  [[ 0  6  7]
                    [ 0 63 22]
                    [ 0 20 70]]
Accuracy :  70.7446808511
Report :
          precision    recall  f1-score   support
    B       0.00      0.00      0.00        13
    L       0.71      0.74      0.72        85
    R        0.71      0.78      0.74        90
avg / total 0.66      0.71      0.68       188
```