List and Sublist

**list:**

A list is a versatile, ordered collection of items that can be of any data type, including numbers, strings, and even other lists. Lists are defined using square brackets [], with elements separated by commas. For example, my\_list = [1, 2, 3, 'apple', 'banana'] creates a list containing integers and strings.

**Sublist:**

A sublist refers to a contiguous portion of a list. A sublist can be any segment of the original list, starting at some index and ending at another.

**Aim:**

To write a python program to check whether a list contains a sublist

**Algorithm:**

* Start the program
* Using input Method getting the input of list and sublist from User
* Checking if the list containing the sublist or not.
* If list contain sublist the output will be “list2 is a subset of list1”
* Else the output will be “list2 is Not a subset of list1”

**Program 1:**

if set(list2).issubset(set(list1)):

print('list2 is a subset of list1')

else:

print('list2 is not a subset of list1')

**Output:**

list1 element5 7 9 10

sublist elements7 9

list2 is a subset of list1

**Program 2:**

sub\_list=False;

for i in range(0,len(list1)):

j=0

while((i+j)<len(list1) and j<len(list2) and list1[i+j]==list2[j]):

j+=1

if j==len(list2):

sub\_list=True;

break;

if(sub\_list):

print("list2 is a subset of list1");

else:

print("list2 is not a subset of list1");

**Output:**

list1 element5 7 9 10

sublist elements7 8

list2 is not a subset of list1

**Program 3:**

list=[1,2,3,4,5,6,7];

print("The given list is ",str(list));

sub\_list=[1,4,3];

res=False;

for idx in range(len(list)-len(sub\_list)+1):

if(list[idx:idx+len(sub\_list)]==sub\_list):

res=True;

print("The given sub\_list is ",res)

**Output:**

The given list is [1, 2, 3, 4, 5, 6, 7]

The given sub\_list is False

list1 element

**Program 4:**

for i in list2:

if i in list1:

count=True

else:

count=False

if count:

print("list2 is a subset of list1");

else:

print("list2 is a not a subset of list1");

**Output:**

list1 element5 6 78 9 13

sublist elements5 6

list2 is a subset of list1

**Result:**

Thus the program has been executed successfully.

**2.Replace dictionary values with their average**

**Dictionary:**

A dictionary is a built-in data structure that stores data in key-value pairs. Each key is unique and maps to a specific value. Dictionaries are unordered collections, meaning they do not maintain the order of items, though as of Python 3.7, they do preserve insertion order. They are created using curly braces {} with key-value pairs separated by colons.

**Aim:**

To Develop a python program to replace dictionary values with their average

**Algorithm:**

* Start the Program
* Define the list of student dictionaries, each containing 'id', 'name', 'M1', and 'M2' keys.
* For each dictionary in the list:
* Retrieve the values associated with 'M1' and 'M2'.
* Calculate the average of 'M1' and 'M2'.
* Store the calculated average in a new key 'Marks[M1+M2]'.
* Output the modified list of dictionaries with the new key and value.
* Stop the Program

**Program:**

def dict\_avg\_val(list\_items):  
    for d in list\_items:  
        n1=d.pop('M1')  
        n2=d.pop('M2')  
        d['Marks[M1+M2]']=(n1+n2)/2  
    return list\_items  
Student\_list=[{'id':1,'name':"XXX",'M1':72,'M2':70},  
              {'id':2,'name':"YYY",'M1':80,'M2':80},  
              {'id':3,'name':"ZZZ",'M1':92,'M2':90}]  
print(dict\_avg\_val(Student\_list))

**Output:**

[{'id': 1, 'name': 'XXX', 'Marks[M1+M2]': 71.0}, {'id': 2, 'name': 'YYY', 'Marks[M1+M2]': 80.0}, {'id': 3, 'name': 'ZZZ', 'Marks[M1+M2]': 91.0}]

**Result:**

Thus the Program has been executed successfully

**3.Performing manipulation of tuple element.**

**Tuple:**

A tuple is an immutable, ordered collection of items. Unlike lists, tuples cannot be modified after their creation, which means you cannot add, remove, or change elements once a tuple is defined. Tuples are defined using parentheses () and can contain elements of any data type, including other tuples.

* **append() Method**

The append() method is used to add a single element to the end of a list. It modifies the list in place, meaning it updates the original list rather than creating a new one.

* **pop() Method**

The pop() method is used to remove and return an element from a list. By default, it removes the last element. Optionally, you can specify an index to remove an element from a specific position.

**Aim:**

To Develop a python program to perform manipulation of tuple element.

**Algorithm:**

* Start the Program
* Initialize Input Data
* Define a tuple named countries containing country names.
* Display Initial Tuple
* Print the initial tuple countries.
* Convert Tuple to List
* Convert the tuple countries to a list temp.
* Print the converted list.
* Add an Item to the List
* Append the item "Russia" to the list temp.
* Print the list after appending.
* Remove an Item from the List
* Remove the item at index 3 from the list temp.
* Print the list after removing the item.
* Change an Item in the List
* Change the item at index 2 in the list temp to "Finland".
* Print the list after changing the item.
* Convert List Back to Tuple
* Convert the modified list temp back to a tuple named countries.
* Print the final tuple countries.
* Stop the Program

**Program:**

countries = ("Spain", "Italy", "India", "England", "Germany")

print("Before manipulating the countries will be:")

print(countries)

temp = list(countries)

print("Countries: ",temp)

temp.append("Russia") #add item

print("After append the countries will be: ",temp)

temp.pop(3) #remove item

print("After removing the item in countries: ",temp)

temp[2] = "Finland" #change item

print("Changing the item in countries: ",temp)

countries = tuple(temp)

print("After manipulating the countries will be:")

print(countries)

**Output**:

Before manipulating the countries will be:

('Spain', 'Italy', 'India', 'England', 'Germany')

Countries: ['Spain', 'Italy', 'India', 'England', 'Germany']

After append the countries will be: ['Spain', 'Italy', 'India', 'England', 'Germany', 'Russia']

After removing the item in countries: ['Spain', 'Italy', 'India', 'Germany', 'Russia']

Changing the item in countries: ['Spain', 'Italy', 'Finland', 'Germany', 'Russia']

After manipulating the countries will be:

('Spain', 'Italy', 'Finland', 'Germany', 'Russia')

**Result:**

Thus the Program has been executed successfully

**4.Pandas and numpy to get the powers of an array values element-wise. First array elements raised to powers of second array element.**

**Pandas:**

Pandas is a popular open-source data manipulation and analysis library for Python. It provides data structures and functions needed to work with structured data seamlessly. The most important data structures in Pandas are:

1. **Series**: A one-dimensional labeled array capable of holding any data type (integers, strings, floating points, etc.). It can be thought of as a column in a spreadsheet.
2. **DataFrame**: A two-dimensional labeled data structure with columns of potentially different types. It’s like a table in a database or an Excel spreadsheet. DataFrames are the most commonly used objects in Pandas for data manipulation tasks.

**NumPy:**

NumPy (Numerical Python) is a fundamental library in Python used for numerical computing. It provides support for arrays, matrices, and a large collection of mathematical functions to operate on these data structures. NumPy is the foundation for many other scientific libraries in Python, such as Pandas, SciPy, and scikit-learn.

**Arrays:**

An array is a data structure that can store a collection of elements, typically of the same data type, in a contiguous block of memory. Arrays are used to store data in a structured and efficient way, making it easier to perform mathematical operations on large datasets.

**Aim:**

To Implement pandas and numpy to get the powers of an array values element-wise. First array elements raised to powers of second array element.

**Algorithm:**

* Start the program
* # Initialize numpy arrays
* Perform element-wise power operation using numpy
* Equivalent operation using numpy's \*\* operator
* Convert numpy arrays to pandas Series
* Perform element-wise power operation using pandas Series
* Stop the Program

**Program:**

import numpy as np

import pandas as pd

arr1 = np.array([[1, 2, 3], [4, 5, 6]])

arr2 = np.array([[1, 2, 3], [4, 5, 6]])

result\_numpy = np.power(arr1, arr2)

print("Numpy power result:")

print(result\_numpy)

print("Numpy \*\* operator result:")

print(arr1 \*\* arr2)

series1 = pd.Series(arr1.flatten())

series2 = pd.Series(arr2.flatten())

result\_pandas = series1.pow(series2)

print("Pandas power result:")

print(result\_pandas)

**Output:**

Numpy power result:

[[ 1 4 27]

[256 3125 7776]]

Numpy \*\* operator result:

[[ 1 4 27]

[256 3125 7776]]

Pandas power result:

0 1

1 4

2 27

3 256

4 3125

5 7776

dtype: int64

**Result:**

Thus the Program has been executed successfully

**5. Check whether a string is panagram or not**

**String:**

A string is a sequence of characters enclosed within either single quotes ('), double quotes ("), or triple quotes (''' or """). Strings are used to represent and manipulate textual data, such as words, sentences, or any combination of characters.

**Panagram:**

A **pangram** is a sentence that contains every letter of the alphabet at least once. The most famous example in English is:

**"The quick brown fox jumps over the lazy dog."**

This sentence includes every letter from A to Z and is often used for testing fonts, keyboards, and other types of text display.

**Aim:**

To Implement a python function to check whether a string is panagram or not

**Algorithm:**

* Start the Program
* Define the Function is\_pangram
* Initialize the Alphabet Set:Create a set containing all lowercase letters of the English alphabet.
* Process the Input Sentence:Convert the sentence to lowercase to ensure case insensitivity.Create a set of characters from the sentence, including only alphabetic characters.Compare the set of characters from the sentence with the alphabet set to determine if all alphabetic characters are present.
* Get User Input
* Prompt the user to enter a sentence.
* Check If the Sentence Is a Pangram
* Use the is\_pangram function to check if the entered sentence is a pangram.
* Print the result indicating whether the sentence is a pangram.
* Stop the Program

**Program:**

import string

def is\_pangram(sentence):

alphabet = set(string.ascii\_lowercase)

# Convert all characters in the sentence to lowercase and ignore non-alphabet characters

letters = set(char for char in sentence.lower() if char.isalpha())

return letters >= alphabet

# Example usage:

sentence1 = input("enter the string")

if is\_pangram(sentence1):

print(f'"{sentence1}" is a pangram.')

else:

print(f'"{sentence1}" is not a pangram.')

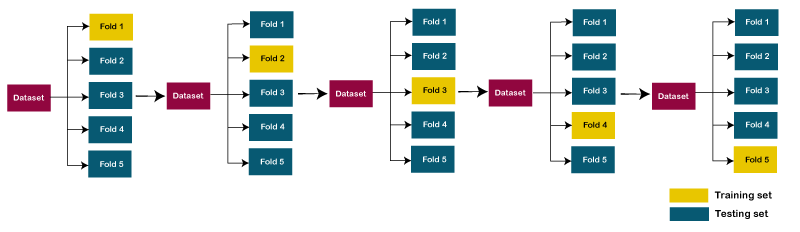
**Result:**

Thus the Program has been executed successfully

**6.k-fold cross validation Algorithm.**

**k-fold cross validation Algorithm:**

Cross-validation is a statistical method used in machine learning to evaluate the performance of a model. It involves partitioning the dataset into subsets, training the model on some subsets, and testing it on the remaining subsets. This process is repeated multiple times to ensure that the model's performance is consistent and not dependent on a particular partitioning of the data.



**Aim:**

To Implement k-fold cross validation Algorithm using python.

**Algorithm:**

1. Start the Program
2. Import Required Libraries
   * Import the datasets module from sklearn.
   * Import DecisionTreeClassifier from sklearn.tree.
   * Import KFold and cross\_val\_score from sklearn.model\_selection.
3. Load the Data
   * Load the Iris dataset using datasets.load\_iris().
   * Assign the feature matrix to X and the target vector to y.
4. Initialize the Model
   * Create an instance of DecisionTreeClassifier with a fixed random seed for reproducibility.
5. Set Up Cross-Validation
   * Create a KFold object with 5 splits for cross-validation.
6. Perform Cross-Validation
   * Use cross\_val\_score to evaluate the model using cross-validation with the defined folds.
   * Store the resulting scores in a variable.
7. Print Results
   * Print the cross-validation scores.
   * Print the average of the cross-validation scores.
   * Print the number of cross-validation scores used in the average.
8. Stop the Program

**Program**:

from sklearn import datasets

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import KFold, cross\_val\_score

X, y = datasets.load\_iris(return\_X\_y=True)

clf = DecisionTreeClassifier(random\_state=42)

k\_folds = KFold(n\_splits = 5)

scores = cross\_val\_score(clf, X, y, cv = k\_folds)

print("Cross Validation Scores: ", scores)

print("Average CV Score: ", scores.mean())

print("Number of CV Scores used in Average: ", len(scores))

Output:

Cross Validation Scores: [1. 1. 0.83333333 0.93333333 0.8 ]

Average CV Score: 0.9133333333333333

Number of CV Scores used in Average: 5

**Result:**

Thus the program has been executed successfully.

**7.To read each row a given CSV file and print a list of strings.**

**List:**

A list is a versatile, ordered collection of items that can be of any data type, including numbers, strings, and even other lists. Lists are defined using square brackets [], with elements separated by commas. For example, my\_list = [1, 2, 3, 'apple', 'banana'] creates a list containing integers and strings.

**String:**

A string is a sequence of characters enclosed within either single quotes ('), double quotes ("), or triple quotes (''' or """). Strings are used to represent and manipulate textual data, such as words, sentences, or any combination of characters.

**Aim:**

To Implement a python program to read each row a given CSV file and print a list of strings.

**Algorithm:**

1. Start the Program
2. Open the CSV File
   * Use the open function to open the file named 'File1.csv'.
   * Use newline='' to handle newline characters properly.
3. Read the CSV File
   * Create a csv.reader object to read the file.
   * Specify the delimiter and quote character.
4. Process Each Row
   * Iterate through each row of the CSV file.
   * Print each row with fields joined by a comma.
5. Close the File
   * The with statement automatically handles closing the file.
6. Stop the Program

**Program**:

import csv

with open('File1.csv', newline='') as csvfile:

data = csv.reader(csvfile, delimiter=' ', quotechar='|')

for row in data:

print(', '.join(row))

File1.csv:

RollNo,Name,Dept

23mca001,Abinaya,MCA

23mca002,Amuthavalli,MCA

23mca003,AnupShankar,MCA

23mca004,Ashika,MCA

23mca005,Bhuvaneshwari,MCA

Output:

RollNo,Name,Dept

23mca001,Abinaya,MCA

23mca002,Amuthavalli,MCA

23mca003,AnupShankar,MCA

23mca004,Ashika,MCA

23mca005,Bhuvaneshwari,MCA

**Result:**

Thus the program has been executed successfully.

**8. k-mean Algorithm.**

**k-mean Algorithm:**

K-Means is a popular clustering algorithm used in machine learning to partition a dataset into k distinct, non-overlapping groups or clusters. The main goal of the K-Means algorithm is to group data points that are similar to each other into clusters, while ensuring that the data points in different clusters are as different as possible.

**The** **working of the K-Means algorithm is explained in the below steps:**

**Step-1:** Select the number K to decide the number of clusters.

**Step-2:** Select random K points or centroids. (It can be other from the input dataset).

**Step-3:** Assign each data point to their closest centroid, which will form the predefined K clusters.

**Step-4:** Calculate the variance and place a new centroid of each cluster.

**Step-5:** Repeat the third steps, which means reassign each datapoint to the new closest centroid of each cluster.

**Step-6:** If any reassignment occurs, then go to step-4 else go to FINISH.

**Step-7**: The model is ready.

**Aim:**

To Implement k-mean Algorithm using python.

**Algorithm:**

* Begin execution of the program.
* Import matplotlib.pyplot as plt for creating visualizations.
* Import KMeans from sklearn.cluster to perform K-Means clustering.
* Define two lists x and y containing the coordinates of the data points.
* Combine these lists into a list of tuples called data, where each tuple represents a point in 2D space.
* Create an empty list inertias to store the inertia values for each K-Means model with different numbers of clusters.
* Loop through a range from 1 to the length of the data list (inclusive). For each iteration i:
  + **Create K-Means Model:** Initialize the KMeans object with n\_clusters=i and random\_state=42 for reproducibility.
  + **Fit Model:** Fit the K-Means model to the data.
  + **Calculate and Store Inertia:** Append the inertia\_ value (the sum of squared distances from each point to its nearest cluster center) to the inertias list.
* Use plt.plot() to plot the number of clusters (x-axis) against the inertia values (y-axis). Use markers to indicate individual points.
* Add a title to the plot ('Elbow Method').
* Label the x-axis ('Number of clusters') and the y-axis ('Inertia').
* Call plt.show() to display the plot on the screen.
* Complete the execution of the program and stop.

**Program**:

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

x = [1, 2, 3, 4, 5]

y = [1, 2, 3, 4, 5]

data = list(zip(x, y))

inertias = []

for i in range(1, len(data) + 1):

    kmeans = KMeans(n\_clusters=i, random\_state=42)

    kmeans.fit(data)

    inertias.append(kmeans.inertia\_)

plt.plot(range(1, len(data) + 1), inertias, marker='o')

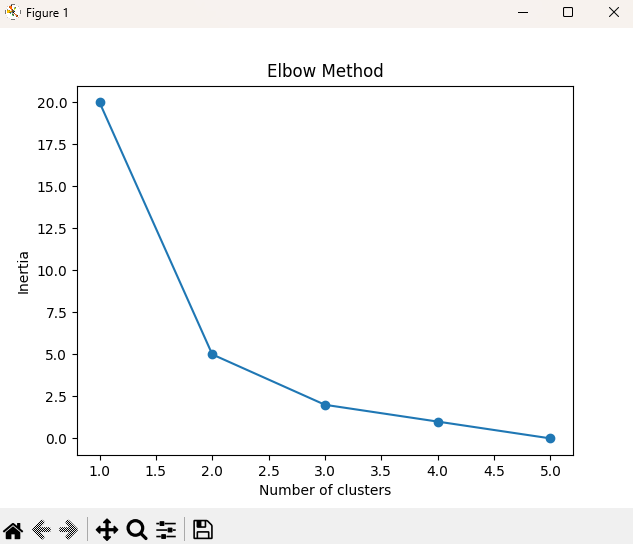
plt.title('Elbow Method')

plt.xlabel('Number of clusters')

plt.ylabel('Inertia')

plt.show()

Output:



**Result:**

Thus the program has been executed successfully.

**9.naïve bayes Algorithm.**

**naïve bayes Algorithm:**

The Naïve Bayes algorithm is a simple yet powerful probabilistic classifier based on applying Bayes' theorem with the "naïve" assumption that the features are independent of each other given the class label. Despite this strong assumption, Naïve Bayes works well in many real-world situations, particularly for text classification tasks like spam detection and sentiment analysis.

## **Naïve Bayes?**

The Naïve Bayes algorithm is comprised of two words Naïve and Bayes, Which can be described as:

* **Naïve**: It is called Naïve because it assumes that the occurrence of a certain feature is independent of the occurrence of other features. Such as if the fruit is identified on the bases of color, shape, and taste, then red, spherical, and sweet fruit is recognized as an apple. Hence each feature individually contributes to identify that it is an apple without depending on each other.
* **Bayes**: It is called Bayes because it depends on the principle of [Bayes' Theorem](https://www.javatpoint.com/bayes-theorem-in-artifical-intelligence).

## **Bayes' Theorem:**

* Bayes' theorem is also known as **Bayes' Rule** or **Bayes' law**, which is used to determine the probability of a hypothesis with prior knowledge. It depends on the conditional probability.
* The formula for Bayes' theorem is given as:

Naïve Bayes Classifier Algorithm

**Where,**

**P(A|B) is Posterior probability**: Probability of hypothesis A on the observed event B.

**P(B|A) is Likelihood probability**: Probability of the evidence given that the probability of a hypothesis is true.

**Aim:**

To Implement naïve bayes Algorithm using python.

**Algorithm:**

* Start the Program:
* Import numpy as np for handling arrays and numerical operations.
* Import train\_test\_split from sklearn.model\_selection to split the dataset.
* Import GaussianNB from sklearn.naive\_bayes to create the Naïve Bayes classifier.
* Import accuracy\_score from sklearn.metrics to evaluate the accuracy of the model.
* Define a 2D numpy array X representing the feature set, where each row is a data point with two features.
* Define a 1D numpy array y representing the labels, where each element corresponds to the class label of the respective data point in X.
* Use train\_test\_split() to divide X and y into training and testing sets:
* X\_train and y\_train: Subsets of X and y used for training the model.
* X\_test and y\_test: Subsets of X and y used for testing the model.
* Set test\_size=0.3 to allocate 30% of the data for testing.
* Use random\_state=42 to ensure reproducibility of the split.
* Create an instance of GaussianNB called model.
* Use the fit() method on model to train it using X\_train and y\_train.
* Use the trained model to predict the class labels for X\_test by calling model.predict().
* Store the predicted labels in y\_pred.
* Calculate the accuracy of the predictions by comparing y\_pred with the actual labels y\_test using accuracy\_score().
* Multiply the accuracy by 100 to get the percentage and print the result.
* Complete the execution of the program and stop.

**Program:**

import numpy as np

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

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score

X = np.array([[1, 2], [2, 3], [3, 4], [4, 5], [5, 6],

[6, 7], [7, 8], [8, 9], [9, 10], [10, 11]])

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

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

model = GaussianNB()

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

y\_pred = model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy \* 100:.2f}%")

**Output:**

**Accuracy: 100.00%**

**Result:**

Thus the program has been executed successfully.

**10.Logistic Regression(scikit-learn).**

**Logistic Regression(scikit-learn):**

Logistic regression is a supervised machine learning algorithm that accomplishes binary classification tasks by predicting the probability of an outcome, event, or observation. The model delivers a binary or dichotomous outcome limited to two possible outcomes: yes/no, 0/1, or true/false.

**Types of Logistic Regression:**

On the basis of the categories, Logistic Regression can be classified into three types:

Binomial: In binomial Logistic regression, there can be only two possible types of the dependent variables, such as 0 or 1, Pass or Fail, etc.

Multinomial: In multinomial Logistic regression, there can be 3 or more possible unordered types of the dependent variable, such as “cat”, “dogs”, or “sheep”

Ordinal: In ordinal Logistic regression, there can be 3 or more possible ordered types of dependent variables, such as “low”, “Medium”, or “High”.

**Aim:**

To Implement Logistic Regression using python(scikit-learn).

**Algorithm:**

* Start the program
* Load and prepare data
* Convert to a binary classification problem
* Split data into training and testing sets
* Initialize and train the model
* Make predictions
* Evaluate the model
* Optional: Visualize the decision boundary
* Stop the program

**Program:**

import numpy as np

import pandas as pd

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

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

data = load\_iris()

X = data.data

y = data.target

binary\_indices = np.where(y != 2)

X = X[binary\_indices]

y = y[binary\_indices]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

model = LogisticRegression()

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

y\_pred = model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f'Accuracy: {accuracy:.2f}')

print('Classification Report:')

print(classification\_report(y\_test, y\_pred))

print('Confusion Matrix:')

print(confusion\_matrix(y\_test, y\_pred))

X\_train\_2D = X\_train[:, :2]

X\_test\_2D = X\_test[:, :2]

model\_2D = LogisticRegression()

model\_2D.fit(X\_train\_2D, y\_train)

h = .02

x\_min, x\_max = X\_train\_2D[:, 0].min() - 1, X\_train\_2D[:, 0].max() + 1

y\_min, y\_max = X\_train\_2D[:, 1].min() - 1, X\_train\_2D[:, 1].max() + 1

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

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

Z = model\_2D.predict(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.8)

plt.scatter(X\_train\_2D[:, 0], X\_train\_2D[:, 1], c=y\_train, edgecolor='k', marker='o')

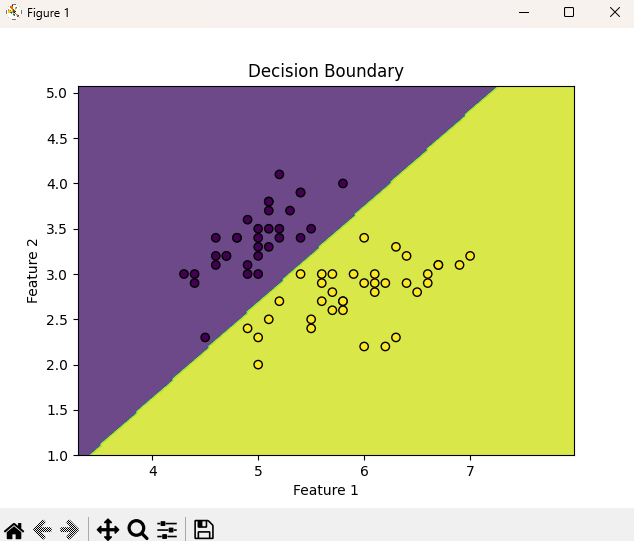
plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.title('Decision Boundary')

plt.show()

**Output:**

****

**Result:**

Thus the program has been executed successfully.

**11.stacked generalization(stacking).**

**Stacked generalization(stacking):**

**Stacked Generalization (Stacking)** is an ensemble learning technique used to improve the performance of machine learning models by combining the predictions of multiple base models. The idea is to leverage the strengths of different algorithms to create a more robust and accurate final model.

How stacking works?

We split the training data into K-folds just like K-fold cross-validation.

A base model is fitted on the K-1 parts and predictions are made for Kth part.

We do for each part of the training data.

The base model is then fitted on the whole train data set to calculate its performance on the test set.

We repeat the last 3 steps for other base models.

Predictions from the train set are used as features for the second level model.

Second level model is used to make a prediction on the test set.

**Aim:**

To Implement stacked generalization(stacking) with python.

**Algorithm:**

* Begin execution of the program.
* Import necessary libraries for data handling, model training, and evaluation.
* Load or create the dataset with features X and target labels y.
* Divide the dataset into training and testing sets.
* Initialize multiple base models that will be used in the stacking ensemble (e.g., Decision Trees, Logistic Regression, K-Nearest Neighbors).
* Train each base model on the training data.
* For each base model, generate predictions on the training data.
* Use these predictions as input features for the meta-model.
* Initialize the meta-model, which could be any machine learning model.
* Train the meta-model on the predictions generated by the base models.
* Use the trained meta-model to make predictions on the test data.
* Evaluate the performance of the stacking model using metrics such as accuracy, confusion matrix, and classification report.
* Complete the execution of the program and stop.

**Program:**

import numpy as np

from sklearn.datasets import load\_iris

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

from sklearn.ensemble import StackingClassifier

from sklearn.linear\_model import LogisticRegression

from sklearn.tree import DecisionTreeClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report

data = load\_iris()

X = data.data

y = data.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

base\_models = [

('decision\_tree', DecisionTreeClassifier()),

('knn', KNeighborsClassifier()),

('svc', SVC(probability=True))

]

meta\_model = LogisticRegression()

stacking\_model = StackingClassifier(estimators=base\_models, final\_estimator=meta\_model, cv=5)

stacking\_model.fit(X\_train, y\_train)

y\_pred = stacking\_model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy \* 100:.2f}%")

print("Confusion Matrix:")

print(confusion\_matrix(y\_test, y\_pred))

print("Classification Report:")

print(classification\_report(y\_test, y\_pred))

**Output:**

Accuracy: 100.00%

Confusion Matrix:

[[19 0 0]

[ 0 13 0]

[ 0 0 13]]

Classification Report:

precision recall f1-score support

0 1.00 1.00 1.00 19

1 1.00 1.00 1.00 13

2 1.00 1.00 1.00 13

accuracy 1.00 45

macro avg 1.00 1.00 1.00 45

weighted avg 1.00 1.00 1.00 45

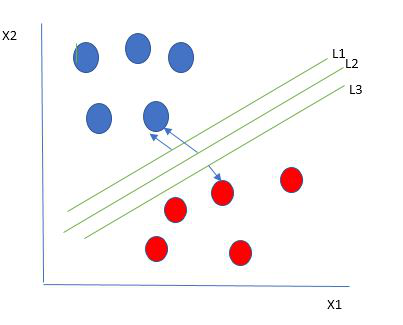
**Result:**

Thus the program has been executed successfully.

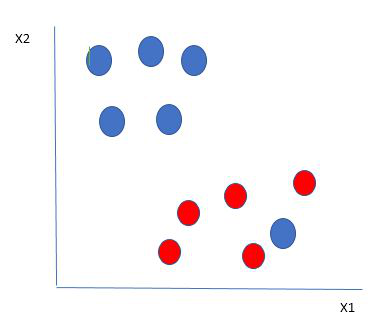
**12.Support vector machine.**

**Support Vector Machine (SVM):**

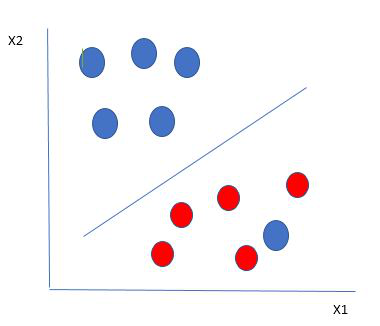
Support Vector Machine (SVM) is a [supervised machine learning](https://www.geeksforgeeks.org/supervised-unsupervised-learning/) algorithm used for both classification and regression. Though we say regression problems as well it’s best suited for classification. The main objective of the SVM algorithm is to find the optimal [hyperplane](https://www.geeksforgeeks.org/separating-hyperplanes-in-svm/) in an N-dimensional space that can separate the data points in different classes in the feature space. The hyperplane tries that the margin between the closest points of different classes should be as maximum as possible. The dimension of the hyperplane depends upon the number of features. If the number of input features is two, then the hyperplane is just a line. If the number of input features is three, then the hyperplane becomes a 2-D plane. It becomes difficult to imagine when the number of features exceeds three.



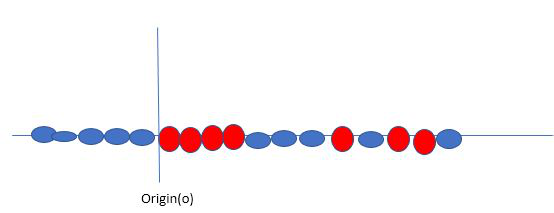
The hyperplane whose distance from it to the nearest data point on each side is maximized. If such a hyperplane exists it is known as the maximum-margin hyperplane/hard margin.



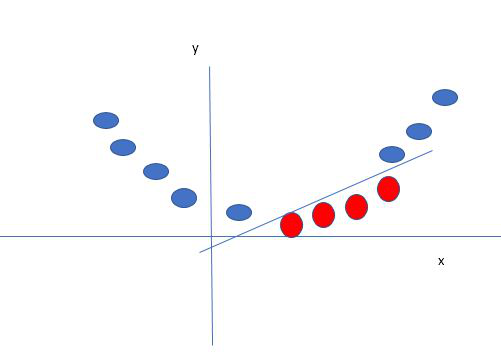
The blue ball in the boundary of red ones is an outlier of blue balls. The SVM algorithm has the characteristics to ignore the outlier and finds the best hyperplane that maximizes the margin. SVM is robust to outliers.



So in this type of data point what SVM does is, finds the maximum margin as done with previous data sets along with that it adds a penalty each time a point crosses the margin. So the margins in these types of cases are called soft margins. When there is a soft margin to the data set, the SVM tries to minimize (1/margin+∧(∑penalty)). Hinge loss is a commonly used penalty. If no violations no hinge loss.If violations hinge loss proportional to the distance of violation.



data is shown in the figure above. SVM solves this by creating a new variable using a kernel. We call a point xi on the line and we create a new variable yi as a function of distance from origin o.so if we plot this we get something like as shown below



the new variable y is created as a function of distance from the origin. A non-linear function that creates a new variable is referred to as a kernel.

**Aim:**

To Implement support vector machine using python.

**Algorithm:**

* Start the program
* Import necessary libraries: make\_blobs from sklearn.datasets, matplotlib.pyplot, SVC from sklearn.svm, numpy, and pandas.
* Use make\_blobs to create a synthetic dataset with 500 samples and 2 centers (classes).
* Plot the generated synthetic dataset to visualize the two classes.
* Initialize a SVC model with a linear kernel.
* Load the cancer dataset from a CSV file using pandas.
* Ensure the dataset contains the columns 'malignant' and 'benign'. If these columns exist:
* Extract the target variable y from the class column (assumed to be at index 30).
* Create a feature matrix x\_features using the 'malignant' and 'benign' columns.
* Fit the SVM model on x\_features and y.
* Predict the class for new data points [120, 990] and [85, 550].
* Print the predictions.
* Generate a range of values xfit for plotting decision boundaries.
* Plot the synthetic dataset again.
* Plot the decision boundaries and margins using the coefficients and intercepts defined in the model. (Note: This part assumes specific values for decision boundaries and margins; in practice, these should be derived from the trained model.)
* Display the scatter plot of the synthetic data.
* Display the scatter plot with decision boundaries.
* Stop the program

**Program:**

from sklearn.datasets import make\_blobs

import matplotlib.pyplot as plt

from sklearn.svm import SVC

import numpy as np

import pandas as pd

X, Y = make\_blobs(n\_samples=500, centers=2, random\_state=0, cluster\_std=0.40)

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

plt.title('Synthetic Data')

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.show()

clf = SVC(kernel='linear')

x = pd.read\_csv("path/to/cancer.csv")

if 'malignant' in x.columns and 'benign' in x.columns:

       y = x.iloc[:, 30].values  # or y = x['class\_column\_name'].values if you know the column name

    x\_features = np.column\_stack((x['malignant'], x['benign']))  # Ensure these column names are correct

    clf.fit(x\_features, y)

        prediction1 = clf.predict([[120, 990]])

    prediction2 = clf.predict([[85, 550]])

    print(f"Prediction for [120, 990]: {prediction1}")

    print(f"Prediction for [85, 550]: {prediction2}")

else:

    print("Columns 'malignant' and 'benign' not found in the CSV file")

xfit = np.linspace(-1, 3.5)

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

for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:

    yfit = m \* xfit + b

    plt.plot(xfit, yfit, '-k')

    plt.fill\_between(xfit, yfit - d, yfit + d, edgecolor='none', color='#AAAAAA', alpha=0.4)

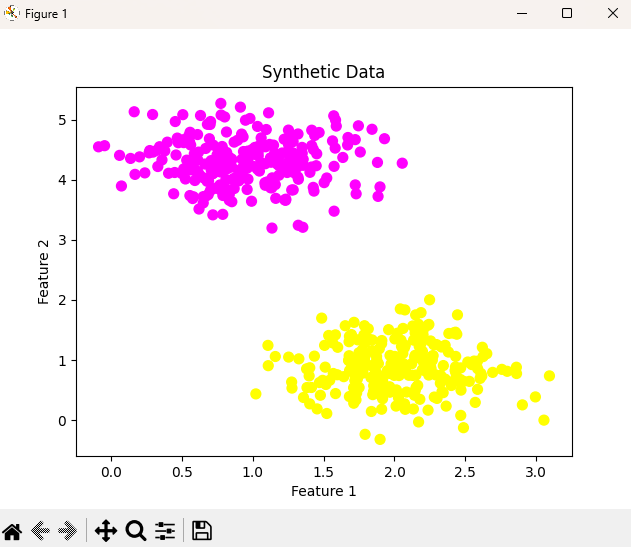
plt.xlim(-1, 3.5)

plt.title('Decision Boundaries')

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.show()

**Output:**

**Result:**

Thus the program has been executed successfully.

**12.Implement any deep learning python libraries.**

**Convolutional Neural Networks (CNNs)**

Definition: Convolutional Neural Networks (CNNs) are a class of deep learning models specifically designed to process and analyze grid-like data, such as images. CNNs use convolutional layers to automatically and adaptively learn spatial hierarchies of features from the input data. These layers apply convolutional filters to detect features like edges, textures, and shapes in images.

**Recurrent Neural Networks (RNNs)**

Definition: Recurrent Neural Networks (RNNs) are a class of neural networks designed to work with sequential data. Unlike traditional feedforward neural networks, RNNs have connections that form directed cycles, allowing them to maintain a state or memory of previous inputs. This makes them suitable for tasks where the context or sequence of data is important, such as time series analysis and natural language processing.

**Aim:**

To Implement any deep learning python libraries.

**Algorithm:**

* Start the program
* Import Libraries
* Import TensorFlow and relevant Keras modules.
* Load and Preprocess Data
* Load CIFAR-10 dataset.
* Normalize pixel values of training and test images.
* Define the CNN Model
* Create a Sequential model.
* Add Convolutional layers with ReLU activation.
* Add MaxPooling layers.
* Flatten the output and add Dense layers.
* Configure the final Dense layer with a softmax activation function for classification.
* Compile the Model
* Set optimizer (e.g., Adam).
* Define loss function (e.g., sparse categorical crossentropy).
* Specify metrics (e.g., accuracy).
* Train the Model
* Fit the model to the training data.
* Use validation data to monitor performance during training.
* Evaluate the Model
* Assess the model’s performance on the test data.
* Print the test accuracy.
* Algorithm for Stopping the Program
* End Training
* The program stops automatically once training is complete after the specified number of epochs.
* Model Evaluation
* After training, the model is evaluated on the test set.
* Output Results
* Print the test accuracy.
* Complete Execution
* The script ends after displaying the results.
* Stop the program

**Program:**

import tensorflow as tf

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0  # Normalize pixel values

model = Sequential([

    Conv2D(32, (3, 3), activation='relu', input\_shape=(32, 32, 3)),

    MaxPooling2D((2, 2)),

    Conv2D(64, (3, 3), activation='relu'),

    MaxPooling2D((2, 2)),

    Conv2D(64, (3, 3), activation='relu'),

    Flatten(),

    Dense(64, activation='relu'),

    Dense(10, activation='softmax')  # 10 classes for CIFAR-10

])

model.compile(optimizer='adam',

              loss='sparse\_categorical\_crossentropy',

              metrics=['accuracy'])

model.fit(x\_train, y\_train, epochs=10, validation\_data=(x\_test, y\_test))

test\_loss, test\_acc = model.evaluate(x\_test, y\_test)

print(f"Test accuracy: {test\_acc}")

**Algorithm:**

**Program:**

import tensorflow as tf

from tensorflow.keras.datasets import reuters

from tensorflow.keras.preprocessing.sequence import pad\_sequences

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Embedding, SimpleRNN, Dense

(x\_train, y\_train), (x\_test, y\_test) = reuters.load\_data(num\_words=10000)

x\_train = pad\_sequences(x\_train, maxlen=100)

x\_test = pad\_sequences(x\_test, maxlen=100)

model = Sequential([

    Embedding(10000, 64),

    SimpleRNN(64),

    Dense(46, activation='softmax')  # 46 classes for Reuters dataset

])

model.compile(optimizer='adam',

              loss='sparse\_categorical\_crossentropy',

              metrics=['accuracy'])

model.fit(x\_train, y\_train, epochs=5, validation\_data=(x\_test, y\_test))

test\_loss, test\_acc = model.evaluate(x\_test, y\_test)

print(f"Test accuracy: {test\_acc}")

**Algorithm:**

1. **Import Libraries**
   * Import TensorFlow and relevant Keras modules for handling the Reuters dataset and defining the RNN model.
2. **Load and Preprocess Data**
   * Load the Reuters dataset with a limit on the number of words to 10,000.
   * Pad sequences to ensure uniform input length (100 tokens).
3. **Define the RNN Model**
   * Create a Sequential model.
   * Add an Embedding layer to convert integer sequences into dense vectors of fixed size.
   * Add a SimpleRNN layer for processing the sequences.
   * Add a Dense layer with softmax activation for classification into 46 classes (Reuters dataset categories).
4. **Compile the Model**
   * Set optimizer (e.g., Adam).
   * Define loss function (e.g., sparse categorical crossentropy).
   * Specify metrics (e.g., accuracy).
5. **Train the Model**
   * Fit the model to the training data.
   * Use validation data to monitor performance during training.
6. **Evaluate the Model**
   * Assess the model’s performance on the test data.
   * Print the test accuracy.

**Output:**

**Result:**

Thus the program has been executed successfully.