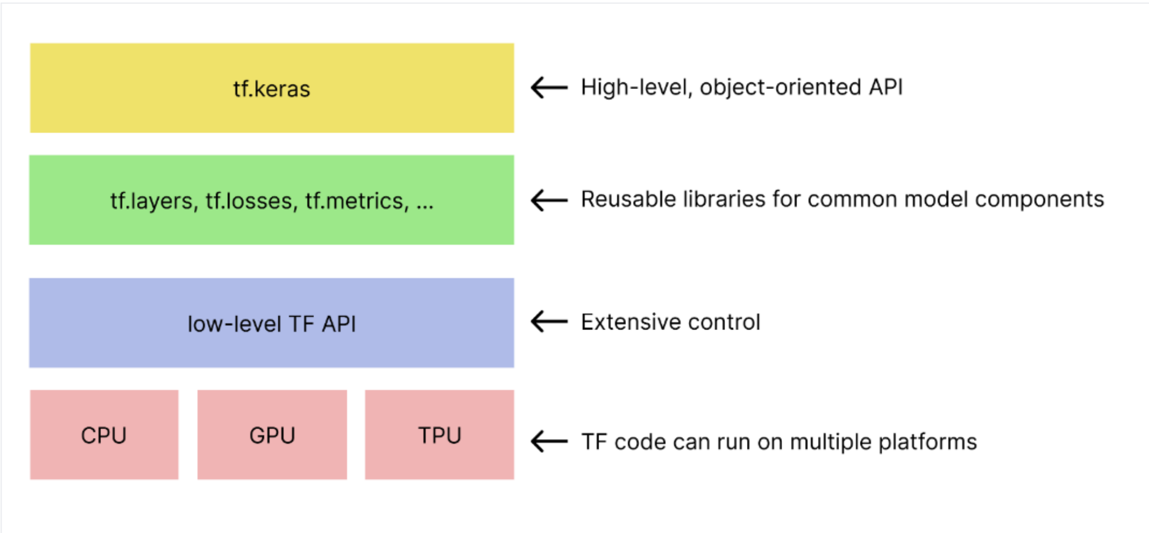
PRACTICAL: 1

Practical: 1

Aim: Prepare a study on environment set up for Tensor Flow and Google Colab. Also implement the basic python commands related to Machine Learning.

TensorFlow:

- TensorFlow is an end-to-end open source platform for machine learning.
- TensorFlow is a rich system for managing all aspects of a machine learning system; however, this class focuses on using a particular TensorFlow API to develop and train machine learning models.
- TensorFlow APIs are arranged hierarchically, with the high-level APIs built on the low-level APIs.
- Machine learning researchers use the low-level APIs to create and explore new machine learning algorithms.



- In this class, you will use a high-level API named tf.keras to define and train machine learning models and to make predictions.
- tf.keras is the TensorFlow variant of the open-source Keras API.

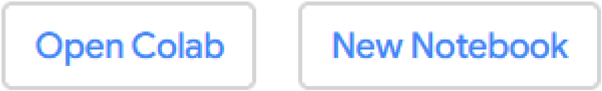
Google Colab :

- Full form of Google Colab is **Google Colaboratory.**

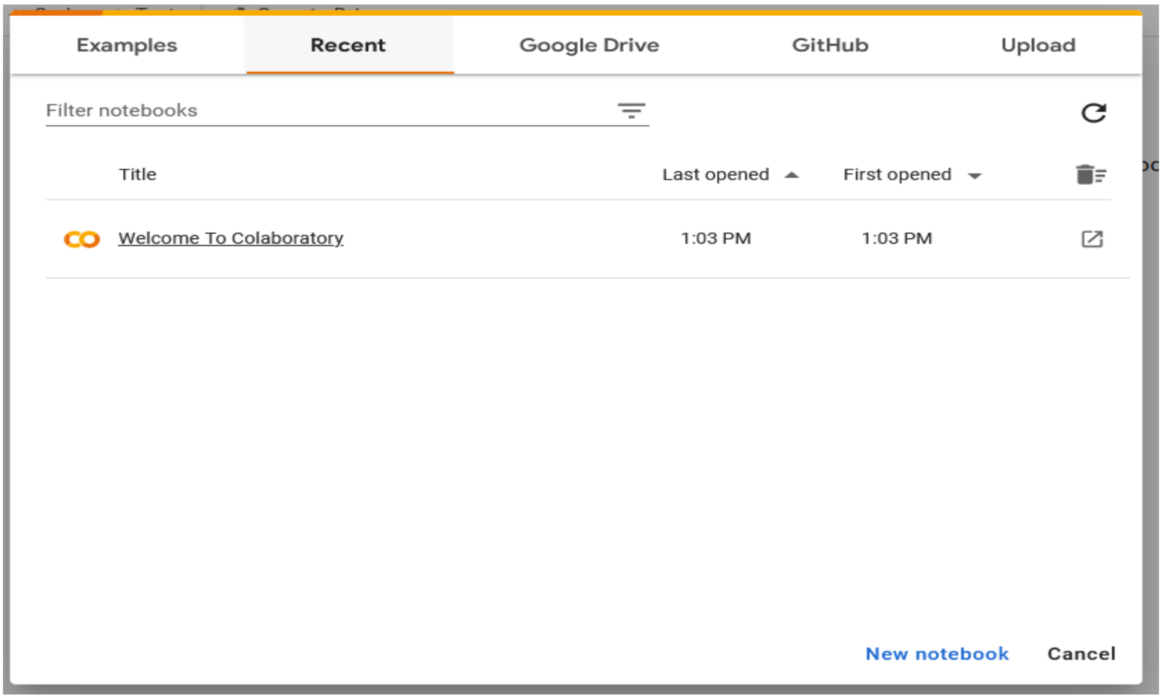
- Colab is a hosted Jupyter Notebook service that requires no setup to use and provides free access to computing resources, including GPUs and TPUs.
- Colab is especially well suited to machine learning, data science, and education.
- Colaboratory, or "Colab" for short, is a product from Google Research. Colab allows anybody to write and execute arbitrary python code through the browser, and is especially well suited to machine learning, data analysis and education.
- More technically, Colab is a hosted Jupyter notebook service that requires no setup to use, while providing access free of charge to computing resources including GPUs.

How to Open Google Colab :

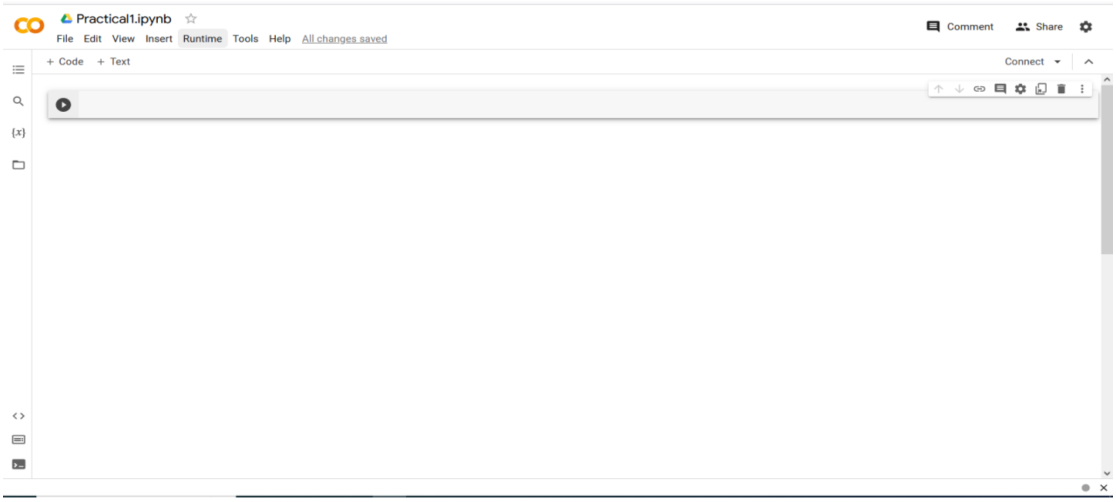
Step 1 : Search google Colab and click on open colab.



Step 2 : show this window and click on New Notebook.



Step 3 : Give a File Name and execute Command.



Output:

Basic Python Commands for ML:

1) Input command :

```
name = input("Enter Your Name: ")
print("Entered Name is: %s" %name)
```

```
Enter Your Name: Jayda Patel
Entered Name is: Jayda Patel
```

2) Type Conversion :

```
c1 = int(2.7)
c2 = float(5)
c3 = float(False)
c4 = float(True)
print(c1, c2, c3, c4)
```

```
2 5.0 0.0 1.0
```

3) String & String Slicing :

```
string = "Machine Learning"
string
```

```
'Machine Learning'
```

```
print(string[8:])
```

```
Learning
```

4) Python Structure:
   - List :

```
list = [1,2,3,4,5]
print(list)
```

```
[1, 2, 3, 4, 5]
```

- Tuple:

```
tuple = (1, 2, 3, 4, 5)
print(tuple)
```

```
(1, 2, 3, 4, 5)
```

- Set:

```
set = {"jayda", "jayshree", "jayda", "tulsi", "twisha", "twisha", "zeel"}
set
```

```
{'jayda', 'jayshree', 'tulsi', 'twisha', 'zeel'}
```

- Dictionary:

```
dict = {"item1": 1, "item2" : [2, 3, 4], "item3" : "5"}
dict
```

```
{'item1': 1, 'item2': [2, 3, 4], 'item3': '5'}
```

5) rang() function:

```
numbers = list(range(1, 6))
print(numbers)
```

```
[1, 2, 3, 4, 5]
```

PRACTICAL: 2

Practical: 2

Aim: Implement the following data manipulation commands/functions:

a) **Loading a CSV file.**
b) **Save data from CSV file to Dataframe.**
c) **Calculation of mean, median, variance, quartiles and inter-quartile range.**

a) Loading a CSV file :

b) Save data from CSV file to Dataframe.

```
import pandas as pd

df=pd.read_csv("auto-mpg.csv")

print(df)
```

```
      mpg  cylinders  displacement  horsepower  weight  acceleration  \
0    18.0          8         307.0       130.0    3504          12.0
1    15.0          8         350.0       165.0    3693          11.5
2    18.0          8         318.0       150.0    3436          11.0
3    16.0          8         304.0       150.0    3433          12.0
4    17.0          8         302.0       140.0    3449          10.5
..    ...        ...           ...         ...     ...           ...
393  27.0          4         140.0        86.0    2790          15.6
394  44.0          4          97.0        52.0    2130          24.6
395  32.0          4         135.0        84.0    2295          11.6
396  28.0          4         120.0        79.0    2625          18.6
397  31.0          4         119.0        82.0    2720          19.4

     model-year
0            70
1            70
2            70
3            70
4            70
..          ...
393          82
394          82
395          82
396          82
397          82

[398 rows x 7 columns]
```

C) Calculation of mean, median, variance, quartiles and inter-quartile range.

	mpg	cylinders	displacement	horsepower	weight	acceleration	model-year
count	398.000000	398.000000	398.000000	396.000000	398.000000	398.000000	398.000000
mean	23.514573	5.454774	193.425879	104.189394	2970.424623	15.568090	76.010050
std	7.815984	1.701004	104.269838	38.402030	846.841774	2.757689	3.697627
min	9.000000	3.000000	68.000000	46.000000	1613.000000	8.000000	70.000000
25%	17.500000	4.000000	104.250000	75.000000	2223.750000	13.825000	73.000000
50%	23.000000	4.000000	148.500000	92.000000	2803.500000	15.500000	76.000000
75%	29.000000	8.000000	262.000000	125.000000	3608.000000	17.175000	79.000000
max	46.600000	8.000000	455.000000	230.000000	5140.000000	24.800000	82.000000

PRACTICAL: 3

Practical: 3

Aim: Write a program to implement the naïve Bayesian classifier for Cancer data set stored as a .CSV file. Compute the accuracy of the classifier.

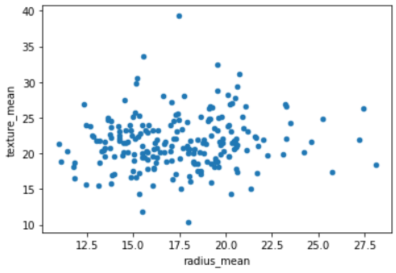
```
In [2]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
```

```
In [4]: dataset = pd.read_csv("data.csv")
        dataset.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 33 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   id                       569 non-null    int64
 1   diagnosis                569 non-null    object
 2   radius_mean              569 non-null    float64
 3   texture_mean             569 non-null    float64
 4   perimeter_mean           569 non-null    float64
 5   area_mean                569 non-null    float64
 6   smoothness_mean          569 non-null    float64
 7   compactness_mean         569 non-null    float64
 8   concavity_mean           569 non-null    float64
 9   concave points_mean      569 non-null    float64
 10  symmetry_mean            569 non-null    float64
 11  fractal_dimension_mean   569 non-null    float64
 12  radius_se                569 non-null    float64
 13  texture_se               569 non-null    float64
 14  perimeter_se             569 non-null    float64
 15  area_se                  569 non-null    float64
 16  smoothness_se            569 non-null    float64
 17  compactness_se           569 non-null    float64
 18  concavity_se             569 non-null    float64
 19  concave points_se        569 non-null    float64
 20  symmetry_se              569 non-null    float64
 21  fractal_dimension_se     569 non-null    float64
 22  radius_worst             569 non-null    float64
 23  texture_worst            569 non-null    float64
 24  perimeter_worst          569 non-null    float64
 25  area_worst               569 non-null    float64
 26  smoothness_worst         569 non-null    float64
 27  compactness_worst        569 non-null    float64
```

```
In [5]: dataset = dataset.drop(["id"], axis = 1)
        dataset = dataset.drop(["Unnamed: 32"], axis = 1)
```

```
In [6]: M = dataset[dataset.diagnosis == "M"]
        B = dataset[dataset.diagnosis == "B"]
```

```
In [15]: #plt.title("Malignant vs Benign Tumor")
         #plt.xlabel("Radius Mean")
         #plt.ylabel("Texture Mean")
         #plt.scatter(M.radius_mean, M.texture_mean, color = "red", label = "Malignant", alpha = 0.3)
         #plt.scatter(B.radius_mean, B.texture_mean, color = "lime", label = "Benign", alpha = 0.3)
         #plt.legend()
         #plt.show()
         g1 = dataset.loc[dataset.diagnosis=='M',:]
         # dataframe.plot.scatter() method
         g1.plot.scatter('radius_mean','texture_mean');
```




```
In [ ]: dataset.diagnosis = [1 if i== "M" else 0 for i in dataset.diagnosis]
```

```
In [ ]: x = dataset.drop(["diagnosis"], axis = 1)
        y = dataset.diagnosis.values
```

```
In [ ]: from sklearn.model_selection import train_test_split
        x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3, random_state = 42)
```

```
In [ ]: from sklearn.naive_bayes import GaussianNB
        nb = GaussianNB()
        nb.fit(x_train, y_train)
        GaussianNB()
        print("Naive Bayes score: ",nb.score(x_test, y_test))

        Naive Bayes score:  0.9415204678362573
```

PRACTICAL: 4

Practical: 4

Aim: Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Compute the accuracy of the classifier.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier
```

```
[3] #read file
    data = pd.read_csv('iris.csv')
    data.head()
```

	sepal.length	sepal.width	petal.length	petal.width	variety
0	5.1	3.5	1.4	0.2	Setosa
1	4.9	3.0	1.4	0.2	Setosa
2	4.7	3.2	1.3	0.2	Setosa
3	4.6	3.1	1.5	0.2	Setosa
4	5.0	3.6	1.4	0.2	Setosa

```
[4] predictors = data.iloc[:,0:4] #predict Value
    target = data.iloc[:,4] #target value
```

```
[10] predictors_train, predictors_test, target_train, target_test = train_test_split(predictors,target,test_size=0.3,random_state=123)
```

```
[6] #model with 3 neighbors
    nn = KNeighborsClassifier(n_neighbors=3)
```

```
[7] #train model
    model = nn.fit(predictors_train,target_train)
```

```
[9] result = nn.predict([[4,5,2,1],])
    print(result)
```

```
['Setosa']
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:439: UserWarning: X does not have valid feature names, but KNeighborsClassifier was fitted
  warnings.warn(
```

```
[11] #check prediction accuracy
     nn.score(predictors_test, target_test)

     0.9555555555555556
```

PRACTICAL: 5

Practical: 5

Aim: Write a program to demonstrate the working of the decision tree algorithm. Use Cancer data set for building the decision tree and apply this knowledge to classify a new sample.

0s
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier
```

0s
```
[4] #read data
    data = pd.read_csv('Breast-Cancer-Wisconsin-Diagnostic-DataSet.csv')
    data.head()
```

0s
```
[4] #read data
    data = pd.read_csv('Breast-Cancer-Wisconsin-Diagnostic-DataSet.csv')
    data.head()
```


	id	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	symmetry_mean	...	texture_worst	perimeter_worst	area
0	842302	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	...	17.33	184.60	
1	842517	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	...	23.41	158.80	
2	84300903	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	...	25.53	152.50	
3	84348301	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	...	26.50	98.87	
4	84358402	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	...	16.67	152.20	

5 rows × 32 columns

0s
```
[5] predictors = data.iloc[:,0:31] #predictors variable
    target = data.iloc[:,31] #target variable
```

0s
```
[6] predictors_train, predictors_test, target_train, target_test = train_test_split(predictors, target, test_size=0.3, random_state=123)
```

```
[7] #Decision tree Classifier
    dtree_entropy = DecisionTreeClassifier(criterion="entropy", random_state=100, max_depth=3, min_samples_leaf=5)
```

```
[9] #train model
    model = dtree_entropy.fit(predictors_train, target_train)
```

```
[12] prediction = dtree_entropy.predict(predictors_test)
```

```
accuracy_score(target_test, prediction, normalize=True)
```

```
0.9532163742690059
```

PRACTICAL: 6

Practical: 6

Aim: Write a program to implement Random Forest Algorithm to classify Cancer data set.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.ensemble import RandomForestClassifier
```

```
[2] #read data
    data = pd.read_csv('Breast-Cancer-Wisconsin-Diagnostic-DataSet.csv')
    data.head()
```

	id	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	symmetry_mean	...	texture_worst	perimeter_worst	area
0	842302	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	...	17.33	184.60	
1	842517	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	...	23.41	158.80	
2	84300903	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	...	25.53	152.50	
3	84348301	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	...	26.50	98.87	
4	84358402	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	...	16.67	152.20	

5 rows × 32 columns

```
[3] predictors = data.iloc[:,0:31] #predictors variable
    target = data.iloc[:,31] #target variable
```

```
[4] predictors_train, predictors_test, target_train, target_test = train_test_split(predictors, target, test_size=0.3, random_state=123)
```

```
[15] #Decision tree Classifier
     r1 = RandomForestClassifier()
```

```
[16] #train model
     model = r1.fit(predictors_train, target_train)
```

```
prediction = r1.predict(predictors_test)
prediction
```

```
array(['B', 'B', 'M', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'M',
       'B', 'M', 'B', 'B', 'B', 'B', 'B', 'M', 'M', 'B', 'B', 'B', 'M',
       'M', 'B', 'M', 'B', 'M', 'B', 'B', 'B', 'M', 'B', 'B', 'B', 'B',
       'M', 'M', 'B', 'M', 'B', 'M', 'B', 'M', 'M', 'M', 'M', 'M', 'M',
       'B', 'B', 'B', 'M', 'B', 'M', 'M', 'B', 'M', 'B', 'B', 'B', 'B',
       'M', 'B', 'B', 'B', 'M', 'B', 'B', 'M', 'B', 'M', 'B', 'B', 'M',
       'M', 'M', 'B', 'M', 'M', 'B', 'B', 'B', 'M', 'B', 'M', 'B', 'M',
       'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
       'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'M', 'M', 'B',
       'M', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'M', 'M', 'B', 'B',
       'M', 'B', 'B', 'M', 'M', 'B', 'B', 'B', 'M', 'M', 'M', 'B', 'M',
       'B', 'M', 'B', 'M', 'M', 'B', 'M', 'B', 'M', 'B', 'B', 'M', 'B',
       'M', 'M', 'M', 'B', 'B', 'B', 'M', 'M', 'B', 'B', 'B', 'M', 'M',
       'M', 'B'], dtype=object)
```

```
[18] accuracy_score(target_test, prediction, normalize=True)
```

```
0.9824561403508771
```

PRACTICAL: 7

Practical: 7

Aim: Write a program to implement Support Vector Machine Algorithm to classify Cancer data set.

```
[1] import pandas as pd
    from sklearn.model_selection import train_test_split
    from sklearn.metrics import accuracy_score
    from sklearn import svm
    from sklearn.preprocessing import MinMaxScaler
    from sklearn.metrics import confusion_matrix, classification_report
```

```
[3] #read data
    data = pd.read_csv('Breast-Cancer-Wisconsin-Diagnostic-DataSet.csv')
    data.head()
```

	id	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	symmetry_mean	...	texture_worst	perimeter_worst	area
0	842302	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	...	17.33	184.60	
1	842517	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	...	23.41	158.80	
2	84300903	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	...	25.53	152.50	
3	84348301	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	...	26.50	98.87	
4	84358402	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	...	16.67	152.20	

5 rows × 32 columns

```
[4] predictors = data.iloc[:,0:31] #predictors variable
    target = data.iloc[:,31] #target variable
```

```
[5] predictors_train, predictors_test, target_train, target_test = train_test_split(predictors, target, test_size=0.3, random_state=123)
```

```
[7] #train model
    svm = svm.LinearSVC(C=100)
    scaler = MinMaxScaler()
    scaler.fit(predictors_train)
    predictors_train = scaler.transform(predictors_train)
    predictors_test = scaler.transform(predictors_test)
    svm.fit(predictors_train,target_train)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/svm/_base.py:1244: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
▾ LinearSVC
LinearSVC(C=100)
```

```
[10] prediction = svm.predict(predictors_test)
     prediction
```

```
array(['B', 'B', 'M', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'M',
       'B', 'M', 'B', 'B', 'B', 'B', 'B', 'M', 'M', 'B', 'B', 'B', 'M',
       'M', 'B', 'M', 'B', 'M', 'B', 'B', 'B', 'M', 'B', 'B', 'B', 'B',
       'M', 'M', 'B', 'M', 'M', 'M', 'B', 'M', 'M', 'B', 'M', 'M', 'M',
       'B', 'B', 'B', 'M', 'B', 'M', 'M', 'B', 'M', 'B', 'B', 'B', 'B',
       'M', 'B', 'B', 'B', 'M', 'B', 'B', 'M', 'B', 'M', 'B', 'B', 'M',
       'M', 'M', 'B', 'M', 'M', 'B', 'B', 'B', 'M', 'B', 'M', 'B', 'M',
       'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
       'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'M', 'B',
       'M', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'M', 'M', 'B', 'B',
       'M', 'B', 'B', 'M', 'M', 'B', 'B', 'B', 'M', 'M', 'M', 'B', 'M',
       'B', 'M', 'B', 'M', 'M', 'B', 'M', 'B', 'M', 'B', 'B', 'M', 'B',
       'M', 'M', 'M', 'B', 'B', 'B', 'M', 'B', 'B', 'B', 'B', 'M', 'M',
       'M', 'M'], dtype=object)
```

```
[11] accuracy_score(target_test, prediction, normalize=True)

0.9766081871345029
```

```
[13] confusion = confusion_matrix(target_test, prediction)
     print('confusion matrix: \n{}'.format(confusion))
```

```
confusion matrix:
[[102   1]
 [  3  65]]
```

```
report = classification_report(target_test, prediction, target_names=['malignant', 'benign'])
print(report)
```

```
              precision    recall  f1-score   support

   malignant       0.97      0.99      0.98       103
      benign       0.98      0.96      0.97        68

    accuracy                           0.98       171
   macro avg       0.98      0.97      0.98       171
weighted avg       0.98      0.98      0.98       171
```

PRACTICAL: 8

Practical: 8

Aim: Write a program to implement K-means Clustering.

```
import numpy as np
import pandas as pd
import sklearn
```

```
from sklearn.datasets import load_digits
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score, homogeneity_score, completeness_score
from scipy.stats import mode
```

```
digits = load_digits()
digits_data =  digits.data/255
```

```
kmeans = KMeans(n_clusters=10, random_state=0)
digits_kmeans = kmeans.fit_predict(digits_data)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: F
  warnings.warn(
```

```
# get_cluster_accuracy(digits.target, digits_kmeans, 10)
labels = np.zeros_like(digits_kmeans)
for i in range(10):
    mask = (digits_kmeans == i)
    labels[mask] = mode(digits.target[mask])[0]
print("Accuracy {0} \n Homogeneity {1} \n Completeness {2}".format(
    accuracy_score(digits.target, labels), homogeneity_score(digits.target, labels),
completeness_score(digits.target, labels)))
```

```
Accuracy 0.7935447968836951
 Homogeneity 0.7423769268336259
 Completeness 0.7514312243853245
```

PRACTICAL: 9

Practical: 9

Aim: Write a program to implement Apriori algorithm for association rule learning.

```
import pandas as pd
import numpy as np
from mlxtend.frequent_patterns import apriori, association_rules
```

```
df = pd.read_csv('GroceryStoreDataSet.csv', names = ['products'], sep = ',')
df.head()
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipykernel.py:283: DeprecationWarning: `shoul
  and should_run_async(code)
```

	products
0	MILK,BREAD,BISCUIT
1	BREAD,MILK,BISCUIT,CORNFLAKES
2	BREAD,TEA,BOURNVITA
3	JAM,MAGGI,BREAD,MILK
4	MAGGI,TEA,BISCUIT

```
df.shape
```

```
/usr/local/lib/python3.10/dist-packages/ipyke
  and should_run_async(code)
(20, 1)
```

```
data = list(df["products"].apply(lambda x:x.split(",") ))
data
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_ru
  and should_run_async(code)
[['MILK', 'BREAD', 'BISCUIT'],
 ['BREAD', 'MILK', 'BISCUIT', 'CORNFLAKES'],
 ['BREAD', 'TEA', 'BOURNVITA'],
 ['JAM', 'MAGGI', 'BREAD', 'MILK'],
 ['MAGGI', 'TEA', 'BISCUIT'],
 ['BREAD', 'TEA', 'BOURNVITA'],
 ['MAGGI', 'TEA', 'CORNFLAKES'],
 ['MAGGI', 'BREAD', 'TEA', 'BISCUIT'],
 ['JAM', 'MAGGI', 'BREAD', 'TEA'],
 ['BREAD', 'MILK'],
 ['COFFEE', 'COCK', 'BISCUIT', 'CORNFLAKES'],
 ['COFFEE', 'COCK', 'BISCUIT', 'CORNFLAKES'],
 ['COFFEE', 'SUGER', 'BOURNVITA'],
 ['BREAD', 'COFFEE', 'COCK'],
 ['BREAD', 'SUGER', 'BISCUIT'],
 ['COFFEE', 'SUGER', 'CORNFLAKES'],
 ['BREAD', 'SUGER', 'BOURNVITA'],
 ['BREAD', 'COFFEE', 'SUGER'],
 ['BREAD', 'COFFEE', 'SUGER'],
 ['TEA', 'MILK', 'COFFEE', 'CORNFLAKES']]
```

```
#Let's transform the list, with one-hot encoding
from mlxtend.preprocessing import TransactionEncoder
a = TransactionEncoder()
a_data = a.fit(data).transform(data)
df = pd.DataFrame(a_data,columns=a.columns_)
df = df.replace(False,0)
df
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_asy
  and should_run_async(code)
```

	BISCUIT	BOURNVITA	BREAD	COCK	COFFEE	CORNFLAKES	JAM	MAGGI	MILK	SUGER	TEA
0	True	0	True	0	0	0	0	0	True	0	0
1	True	0	True	0	0	True	0	0	True	0	0
2	0	True	True	0	0	0	0	0	0	0	True
3	0	0	True	0	0	0	True	True	True	0	0
4	True	0	0	0	0	0	0	True	0	0	True
5	0	True	True	0	0	0	0	0	0	0	True
6	0	0	0	0	0	True	0	True	0	0	True
7	True	0	True	0	0	0	0	True	0	0	True
8	0	0	True	0	0	0	True	True	0	0	True
9	0	0	True	0	0	0	0	0	True	0	0
10	True	0	0	True	True	True	0	0	0	0	0
11	True	0	0	True	True	True	0	0	0	0	0
12	0	True	0	0	True	0	0	0	0	True	0

```
#set a threshold value for the support value and calculate the support value.
df = apriori(df, min_support = 0.2, use_colnames = True, verbose = 1)
df
```

```
Processing 42 combinations | Sampling itemset size 3
/usr/local/lib/python3.10/dist-packages/ipykernel/ipykernel.py:283: DeprecationWar
  and should_run_async(code)
/usr/local/lib/python3.10/dist-packages/mlxtend/frequent_patterns/fpcommon.py:110
  warnings.warn(
```

	support	itemsets
0	0.35	(BISCUIT)
1	0.2	(BOURNVITA)
2	0.65	(BREAD)
3	0.4	(COFFEE)
4	0.3	(CORNFLAKES)
5	0.25	(MAGGI)
6	0.25	(MILK)
7	0.3	(SUGER)
8	0.35	(TEA)
9	0.2	(BREAD, BISCUIT)
10	0.2	(MILK, BREAD)
11	0.2	(BREAD, SUGER)
12	0.2	(TEA, BREAD)

```
#Let's view our interpretation values using the Associan rule function.
df_ar = association_rules(df, metric = "confidence", min_threshold = 0.6)
df_ar
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipykernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell` automatical
  and should_run_async(code)
```

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction	zhangs_metric
0	(MILK)	(BREAD)	0.25	0.65	0.2	0.800000	1.230769	0.0375	1.75	0.250000
1	(SUGER)	(BREAD)	0.30	0.65	0.2	0.666667	1.025641	0.0050	1.05	0.035714
2	(CORNFLAKES)	(COFFEE)	0.30	0.40	0.2	0.666667	1.666667	0.0800	1.80	0.571429
3	(SUGER)	(COFFEE)	0.30	0.40	0.2	0.666667	1.666667	0.0800	1.80	0.571429
4	(MAGGI)	(TEA)	0.25	0.35	0.2	0.800000	2.285714	0.1125	3.25	0.750000

PRACTICAL: 10

Practical: 10

Aim: Write a program for prediction using Linear Regression on Boston Housing Dataset.

```
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model, metrics
import pandas as pd
```

```
# load the boston dataset
data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
X = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
y = raw_df.values[1::2, 2]
```

```
# splitting X and y into training and testing sets
X_train, X_test,\
    y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=1)
```

```
# create linear regression object
reg = linear_model.LinearRegression()

# train the model using the training sets
reg.fit(X_train, y_train)
```

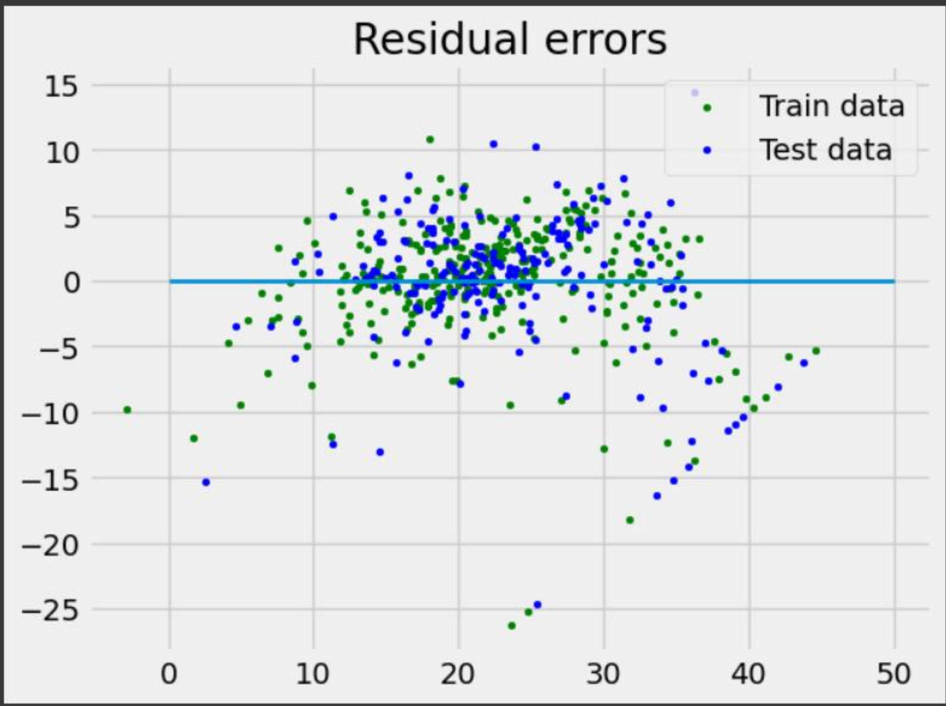
```
▾ LinearRegression
LinearRegression()
```

```
# regression coefficients
print('Coefficients: ', reg.coef_)

# variance score: 1 means perfect prediction
print('Variance score: {}'.format(reg.score(X_test, y_test)))
```

```
Coefficients:  [-8.95714048e-02  6.73132853e-02  5.04649248e-02  2.18579583e+00
 -1.72053975e+01  3.63606995e+00  2.05579939e-03 -1.36602886e+00
  2.89576718e-01 -1.22700072e-02 -8.34881849e-01  9.40360790e-03
 -5.04008320e-01]
Variance score: 0.720905667266174
```

```
# plot for residual error

# setting plot style
plt.style.use('fivethirtyeight')

# plotting residual errors in training data
plt.scatter(reg.predict(X_train),
            reg.predict(X_train) - y_train,
            color="green", s=10,
            label='Train data')

# plotting residual errors in test data
plt.scatter(reg.predict(X_test),
            reg.predict(X_test) - y_test,
            color="blue", s=10,
            label='Test data')

# plotting line for zero residual error
plt.hlines(y=0, xmin=0, xmax=50, linewidth=2)

# plotting legend
plt.legend(loc='upper right')

# plot title
plt.title("Residual errors")

# method call for showing the plot
plt.show()
```

```
plt.show()
```


Residual errors
Train data
Test data
15
10
5
0
−5
−10
−15
−20
−25
0
10
20
30
40
50

PRACTICAL: 11

Practical: 11

Aim: Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.

```
import random
from math import exp
from random import seed

# Initialize a network

def initialize_network(n_inputs, n_hidden, n_outputs):
    network = list()
    hidden_layer = [{'weights':[random.uniform(-0.5,0.5) for i in range(n_inputs + 1)]} for i in range(n_hidden)]
    network.append(hidden_layer)
    output_layer = [{'weights':[random.uniform(-0.5,0.5) for i in range(n_hidden + 1)]} for i in range(n_outputs)]
    network.append(output_layer)
    i= 1
    print("\n The initialised Neural Network:\n")
    for layer in network:
        j=1
        for sub in layer:
            print("\n Layer[%d] Node[%d]:\n" %(i,j),sub)
            j=j+1
        i=i+1
    return network

# Calculate neuron activation (net) for an input

def activate(weights, inputs):
    activation = weights[-1]
    for i in range(len(weights)-1):
        activation += weights[i] * inputs[i]
    return activation

# Transfer neuron activation to sigmoid function
```

```
# Transfer neuron activation to sigmoid function
def transfer(activation):
    return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output
def forward_propagate(network, row):
    inputs = row
    for layer in network:
        new_inputs = []
        for neuron in layer:
            activation = activate(neuron['weights'], inputs)
            neuron['output'] = transfer(activation)
            new_inputs.append(neuron['output'])
        inputs = new_inputs
    return inputs

# Calculate the derivative of an neuron output
def transfer_derivative(output):
    return output * (1.0 - output)

# Backpropagate error and store in neurons
def backward_propagate_error(network, expected):
    for i in reversed(range(len(network))):
        layer = network[i]
        errors = list()

        if i != len(network)-1:
            for j in range(len(layer)):
                error = 0.0
                for neuron in network[i + 1]:
                    error += (neuron['weights'][j] * neuron['delta'])
                errors.append(error)
```

```
                errors.append(error)
        else:
            for j in range(len(layer)):
                neuron = layer[j]
                errors.append(expected[j] - neuron['output'])

        for j in range(len(layer)):
            neuron = layer[j]
            neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])


# Update network weights with error
def update_weights(network, row, l_rate):
    for i in range(len(network)):
        inputs = row[:-1]
        if i != 0:
            inputs = [neuron['output'] for neuron in network[i - 1]]
        for neuron in network[i]:
            for j in range(len(inputs)):
                neuron['weights'][j] += l_rate * neuron['delta'] * inputs[j]
            neuron['weights'][-1] += l_rate * neuron['delta']


# Train a network for a fixed number of epochs
def train_network(network, train, l_rate, n_epoch, n_outputs):

    print("\n Network Training Begins:\n")

    for epoch in range(n_epoch):
        sum_error = 0
        for row in train:
            outputs = forward_propagate(network, row)
            expected = [0 for i in range(n_outputs)]
```

```
            expected = [0 for i in range(n_outputs)]
            expected[row[-1]] = 1
            sum_error += sum([(expected[i]-outputs[i])**2 for i in range(len(expected))])
            backward_propagate_error(network, expected)
            update_weights(network, row, l_rate)
        print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))

    print("\n Network Training Ends:\n")


#Test training backprop algorithm
seed(2)
dataset = [[2.7810836,2.550537003,0],
    [1.465489372,2.362125076,0],
    [3.396561688,4.400293529,0],
    [1.38807019,1.850220317,0],
    [3.06407232,3.005305973,0],
    [7.627531214,2.759262235,1],
    [5.332441248,2.088626775,1],
    [6.922596716,1.77106367,1],
    [8.675418651,-0.242068655,1],
    [7.673756466,3.508563011,1]]

print("\n The input Data Set :\n",dataset)
n_inputs = len(dataset[0]) - 1
print("\n Number of Inputs :\n",n_inputs)
n_outputs = len(set([row[-1] for row in dataset]))
print("\n Number of Outputs :\n",n_outputs)

#Network Initialization
network = initialize_network(n_inputs, 2, n_outputs)

# Training the Network
```

```
# Training the Network
train_network(network, dataset, 0.5, 20, n_outputs)


print("\n Final Neural Network :")

i= 1
for layer in network:
    j=1
    for sub in layer:
        print("\n Layer[%d] Node[%d]:\n" %(i,j),sub)
        j=j+1
    i=i+1
```

```
Number of Outputs :
2
```

```
Number of Outputs :
2

The initialised Neural Network:


Layer[1] Node[1]:
{'weights': [0.4560342718892494, 0.4478274870593494, -0.4434486322731913]}

Layer[1] Node[2]:
{'weights': [-0.41512800484107837, 0.33549887812944956, 0.2359699890685233]}

Layer[2] Node[1]:
{'weights': [0.1697304014402209, -0.1918635424108558, 0.10594416567846243]}

Layer[2] Node[2]:
{'weights': [0.10680173364083789, 0.08120401711200309, -0.3416171297451944]}

Network Training Begins:

>epoch=0, lrate=0.500, error=5.278
>epoch=1, lrate=0.500, error=5.122
>epoch=2, lrate=0.500, error=5.006
>epoch=3, lrate=0.500, error=4.875
>epoch=4, lrate=0.500, error=4.700
>epoch=5, lrate=0.500, error=4.466
>epoch=6, lrate=0.500, error=4.176
>epoch=7, lrate=0.500, error=3.838
>epoch=8, lrate=0.500, error=3.469
>epoch=9, lrate=0.500, error=3.089
>epoch=10, lrate=0.500, error=2.716
>epoch=11, lrate=0.500, error=2.367
>epoch=12, lrate=0.500, error=2.054
>epoch=13, lrate=0.500, error=1.780
>epoch=14, lrate=0.500, error=1.546
>epoch=15, lrate=0.500, error=1.349
```

```
>epoch=15, lrate=0.500, error=1.349
>epoch=16, lrate=0.500, error=1.184
>epoch=17, lrate=0.500, error=1.045
>epoch=18, lrate=0.500, error=0.929
>epoch=19, lrate=0.500, error=0.831

 Network Training Ends:


 Final Neural Network :

 Layer[1] Node[1]:
 {'weights': [0.8642508164347664, -0.8497601716670761, -0.8668929014392035], 'output': 0.9295587965836384, 'delta': 0.005645382825629247}

 Layer[1] Node[2]:
 {'weights': [-1.2934302410111027, 1.7109363237151511, 0.7125327507327331], 'output': 0.04760703296164143, 'delta': -0.005928559978815065}

 Layer[2] Node[1]:
 {'weights': [-1.3098359335096292, 2.16462207144596, -0.3079052288835877], 'output': 0.1989556395205846, 'delta': -0.03170801648036036}

 Layer[2] Node[2]:
 {'weights': [1.5506793402414165, -2.11315950446121, 0.1333585709422027], 'output': 0.8095042653312078, 'delta': 0.029375796661413225}
```

Predict

```
from math import exp

# Calculate neuron activation for an input
def activate(weights, inputs):
    activation = weights[-1]
    for i in range(len(weights)-1):
        activation += weights[i] * inputs[i]
    return activation

# Transfer neuron activation
def transfer(activation):
    return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output
def forward_propagate(network, row):
    inputs = row
    for layer in network:
        new_inputs = []
        for neuron in layer:
            activation = activate(neuron['weights'], inputs)
            neuron['output'] = transfer(activation)
            new_inputs.append(neuron['output'])
        inputs = new_inputs
    return inputs

# Make a prediction with a network
def predict(network, row):
    outputs = forward_propagate(network, row)
```

```python
# Make a prediction with a network
def predict(network, row):
    outputs = forward_propagate(network, row)
    return outputs.index(max(outputs))

# Test making predictions with the network
dataset = [[2.7810836,2.550537003,0],
    [1.465489372,2.362125076,0],
    [3.396561688,4.400293529,0],
    [1.38807019,1.850220317,0],
    [3.06407232,3.005305973,0],
    [7.627531214,2.759262235,1],
    [5.332441248,2.088626775,1],
    [6.922596716,1.77106367,1],
    [8.675418651,-0.242068655,1],
    [7.673756466,3.508563011,1]]
#network = [[{'weights': [-1.482313569067226, 1.8308790073202204, 1.078381922048799]}, {'weights': [0.23244990332399884, 0.3621998343835864, 0.40289821191094327]}],
#    [{'weights': [2.5001872433501404, 0.7887233511355132, -1.1026649757805829]}, {'weights': [-2.429350576245497, 0.8357651039198697, 1.0699217181280656]}]]
for row in dataset:
    prediction = predict(network, row)
    print('Expected=%d, Got=%d' % (row[-1], prediction))
```

```
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
```