**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Pattern Recognition Lab Assignment-2**

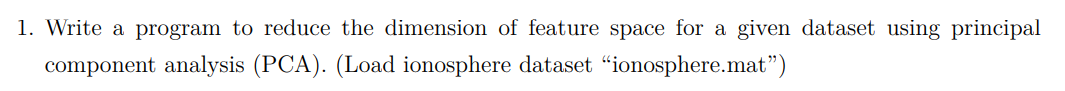
**Submitted by: Yukti Khurana, 2017UCP1234**

**Submitted to: Dr Deepak Ranjan Nayak**

**Semester-8 2021**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**DAY-5**



**CODE**

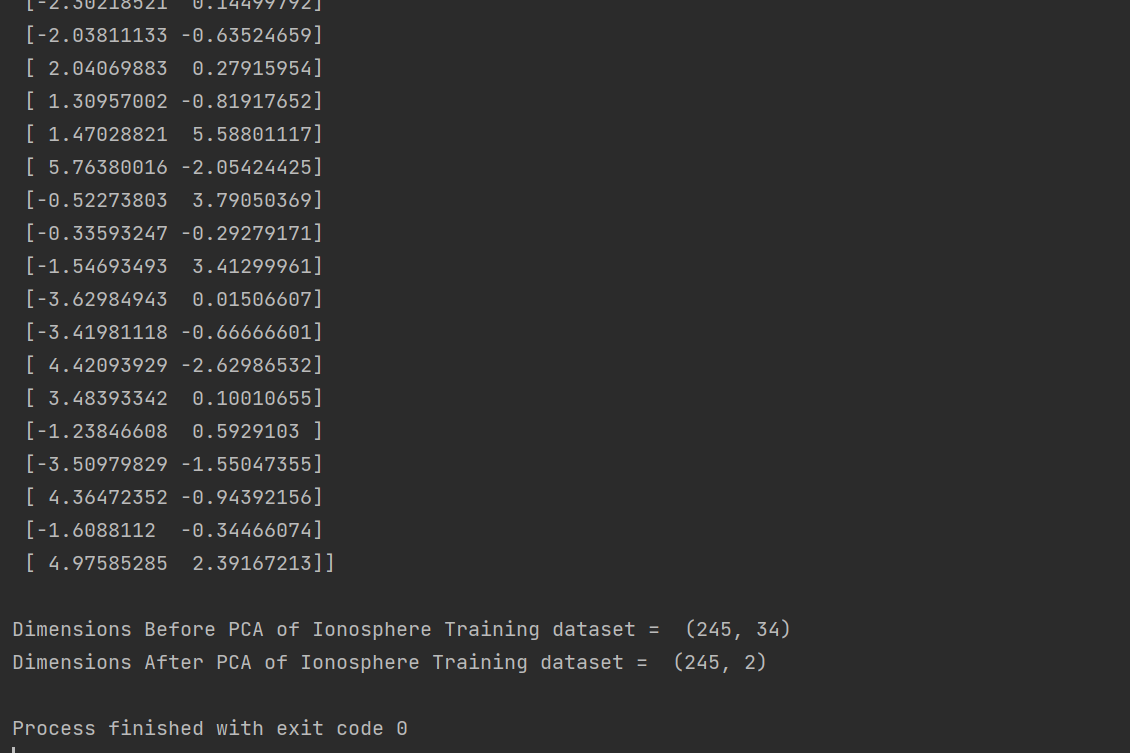
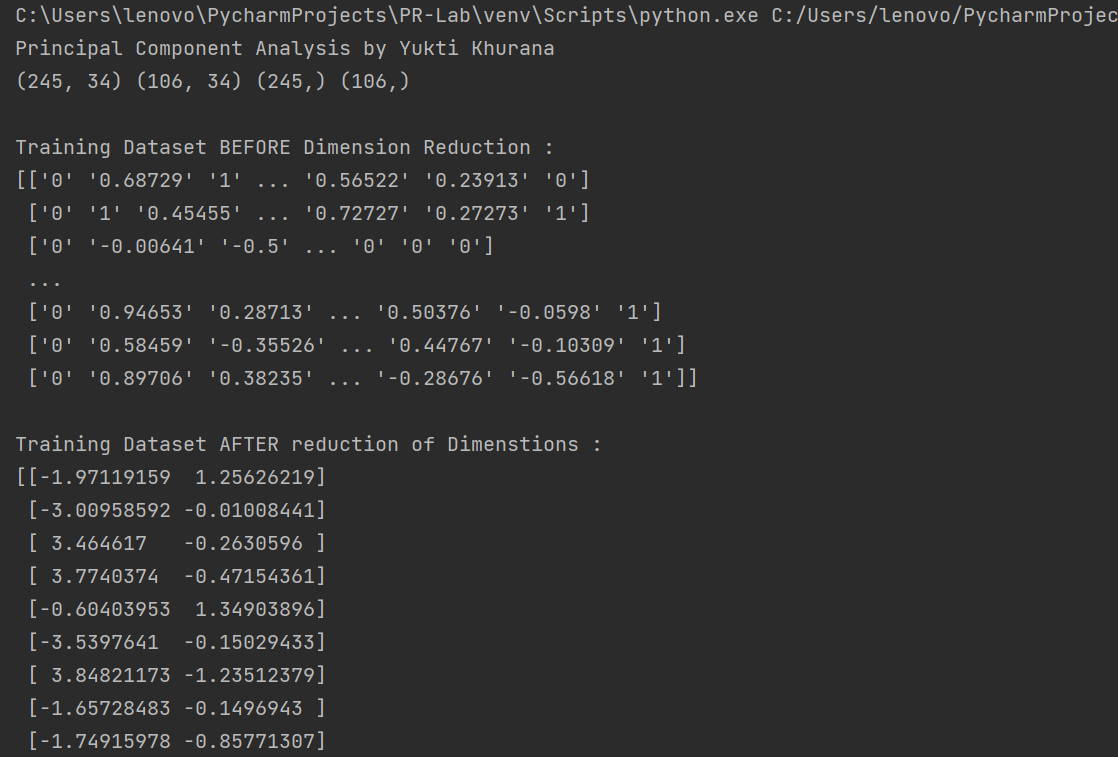
**PCA.py**

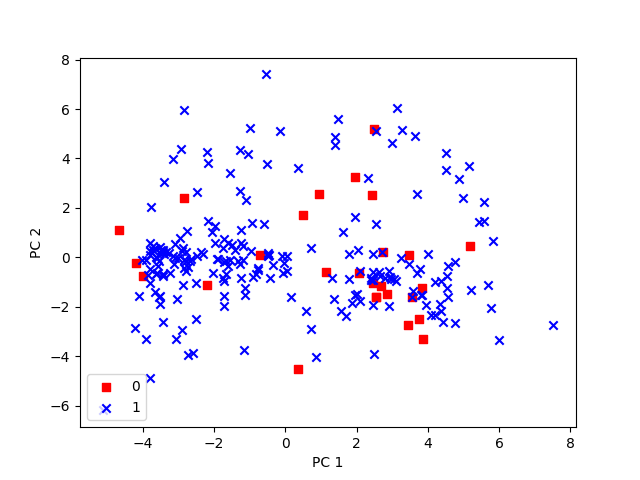
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
from sklearn.model\_selection import train\_test\_split  
from sklearn.preprocessing import StandardScaler  
import matplotlib.pyplot as plt  
  
print("Principal Component Analysis by Yukti Khurana")  
df = pd.read\_csv("Ionosphere.csv",header=None)  
df = df[1:]  
# split into training and testing sets  
X = df.iloc[:, 1:].values  
y = df.iloc[:, 0].values  
# Splitting the training and testing data   
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, train\_size = 0.70, random\_state = 41, stratify = y)  
print(X\_train.shape, X\_test.shape, y\_train.shape, y\_test.shape)  
  
# Standardize the features  
sc = StandardScaler()  
X\_train\_std = sc.fit\_transform(X\_train)  
X\_test\_std = sc.transform(X\_test)  
  
# finding the Covariance Matrix of training data - X  
cov\_mat = np.cov(X\_train\_std.T)  
# Eigen values and Eigen vectors using the covariance matrix  
eigen\_vals, eigen\_vecs = np.linalg.eig(cov\_mat)  
  
# calculate cumulative sum of explained variances  
total\_evals = sum(eigen\_vals)  
var\_exp = [(i / total\_evals) for i in sorted(eigen\_vals, reverse=True)]  
cum\_var\_exp = np.cumsum(var\_exp)  
  
# Feature Extraction  
# Make a list of (eigenvalue, eigenvector) tuples  
eigen\_pairs = [(np.abs(eigen\_vals[i]), eigen\_vecs[:, i]) for i in range(len(eigen\_vals))]  
  
# Sort the (eigenvalue, eigenvector) tuples from high to low (in decreasing order of eigen values)  
eigen\_pairs.sort(key=lambda k: k[0], reverse=True)  
#print("Sorted Eigen Value and Eigen vector pairs")  
#print(eigen\_pairs)  
# we collect the two eigenvectors that correspond to the two largest eigenvalues,  
w = np.hstack((eigen\_pairs[0][1][:, np.newaxis], eigen\_pairs[1][1][:, np.newaxis]))  
#print('Matrix W:\n', w)  
  
# Using the projection matrix, we can now transform a sample x (represented as a 1 x 34-dimensional row vector)   
# onto the PCA subspace (the principal components one and two) obtaining x′  
# now a two-dimensional sample vector consisting of two new features  
# x' = xW  
X\_train\_std[0].dot(w)  
  
# we can transform the entire 245 X 34-dimensional training dataset   
# onto the two principal components by calculating the matrix dot product  
# X' = XW  
X\_train\_pca = X\_train\_std.dot(w)  
print()  
print("Training Dataset BEFORE Dimension Reduction : ")  
print(X\_train)  
print()  
print("Training Dataset AFTER reduction of Dimenstions : ")  
print(X\_train\_pca)  
# Hence the dimensions of dataset were reduced from 34 to 2  
# visualizing the reduced Ionosphere training dataset  
colors = ['r', 'b', 'g']  
markers = ['s', 'x', 'o']  
for l, c, m in zip(np.unique(y\_train), colors, markers):  
 plt.scatter(X\_train\_pca[y\_train==l, 0],   
 X\_train\_pca[y\_train==l, 1],   
 c=c, label=l, marker=m)   
plt.xlabel('PC 1')  
plt.ylabel('PC 2')  
plt.legend(loc='lower left')  
plt.show()  
  
print()  
print("Dimensions Before PCA of Ionosphere Training dataset = ", X\_train.shape)  
print("Dimensions After PCA of Ionosphere Training dataset = ", X\_train\_pca.shape)

**INPUT**

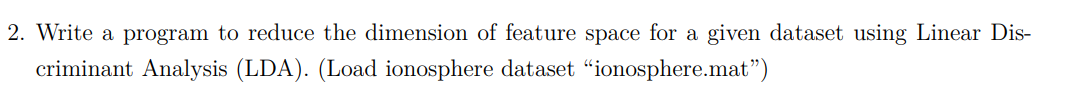
* Ionosphere dataset is used. Any other dataset can also be used.
* The dimensions of the dataset are reduced from 34 to 4.

**OUTPUT**





For Visualization of Dataset features after Dimension reduction by PCA

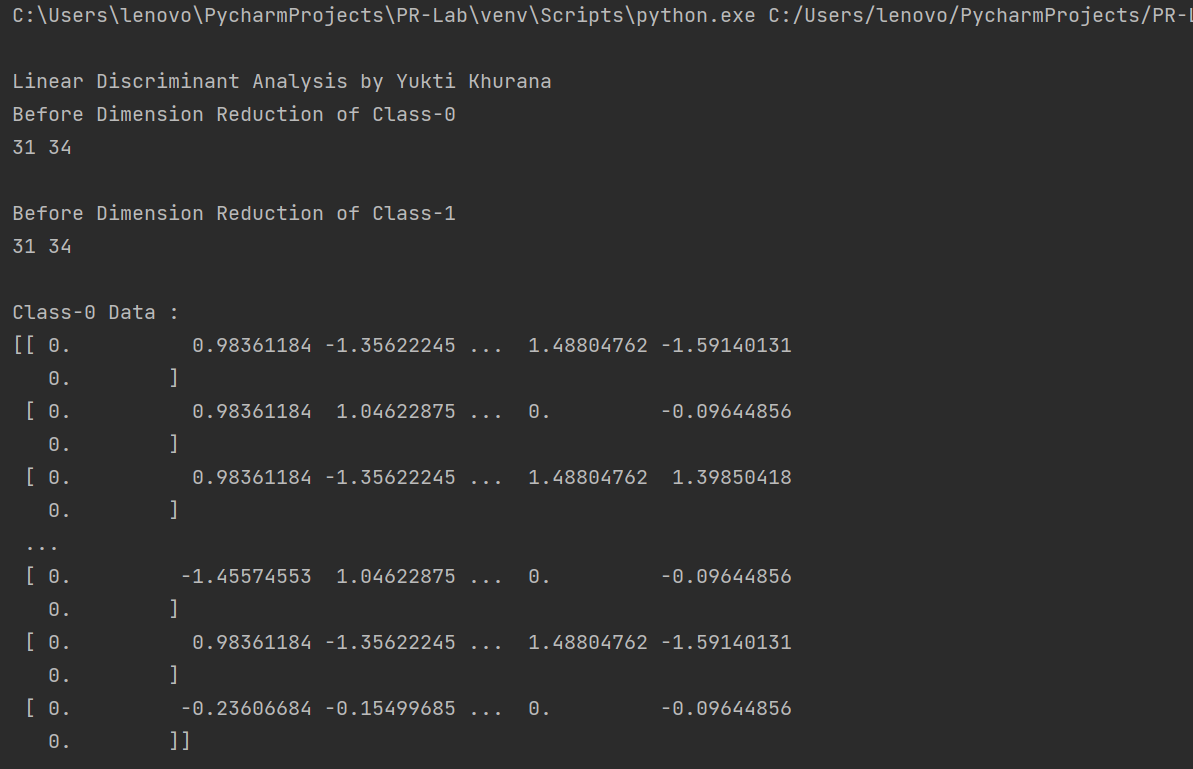


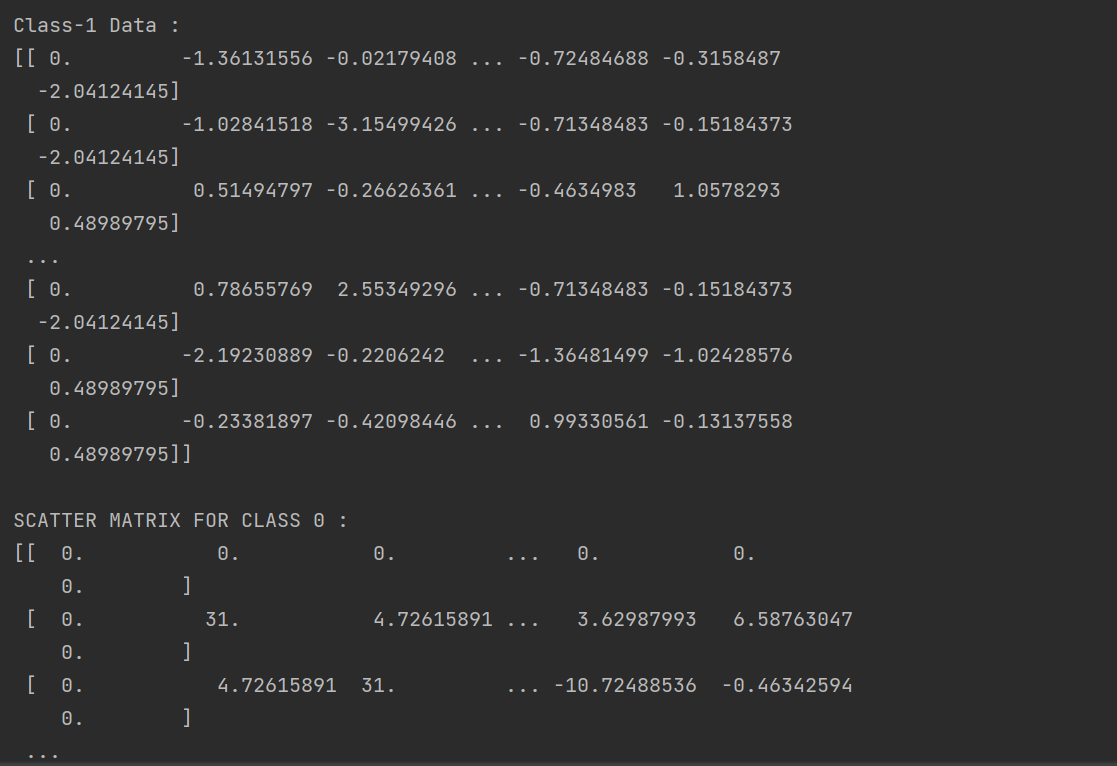
**CODE**

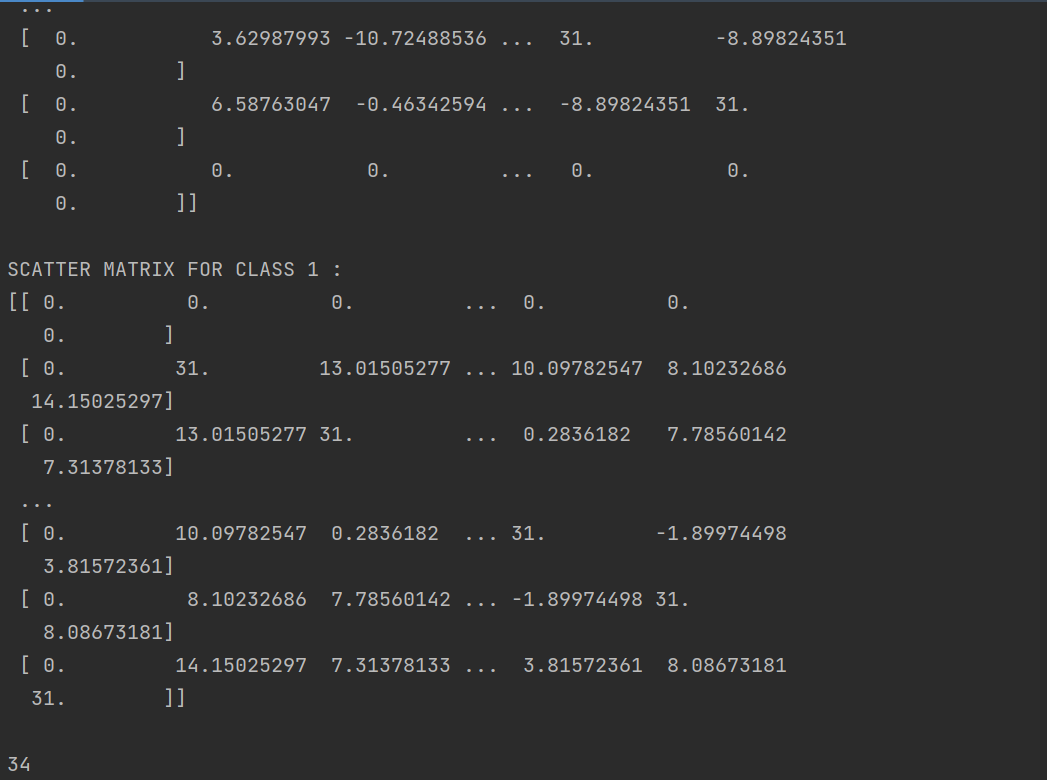
import numpy as np  
import pandas as pd  
from sklearn.preprocessing import StandardScaler  
from sklearn.model\_selection import train\_test\_split  
  
print("\nLinear Discriminant Analysis by Yukti Khurana")  
df = pd.read\_csv("Ionosphere.csv",header=None)  
df = df[1:]  
  
#grouping based on the class of the classification :  
df\_grouped = df.groupby(0)  
  
df\_group0 = df\_grouped.get\_group("0")  
df\_group1 = df\_grouped.get\_group("1")  
  
X\_group0 = df\_group0.iloc[:, 1:].values  
y\_group0 = df\_group0.iloc[:, 0].values  
  
X\_group1 = df\_group1.iloc[:, 1:].values  
y\_group1 = df\_group1.iloc[:, 0].values  
  
X\_train1, X\_test1, y\_train1, y\_test1 = train\_test\_split(X\_group1, y\_group1, train\_size = 0.10, random\_state = 41, stratify = y\_group1)  
X\_train0, X\_test0, y\_train0, y\_test0 = train\_test\_split(X\_group0, y\_group0, train\_size = 0.84, random\_state = 41, stratify = y\_group0)  
  
sc = StandardScaler()  
X\_train\_std1 = sc.fit\_transform(X\_train1)  
X\_test\_std1 = sc.transform(X\_test1)  
  
X\_train\_std0 = sc.fit\_transform(X\_train0)  
X\_test\_std0 = sc.transform(X\_test0)  
  
mean1 = np.mean(X\_train\_std1, axis=0)  
mean0 = np.mean(X\_train\_std0, axis=0)  
  
def Mean(samples):  
 s = np.array(samples)  
 m = np.mean(s, axis=0)  
 return np.array(m)  
  
def getZ(samples, mean):  
 return np.array(samples) - np.array(mean)  
  
print("Before Dimension Reduction of Class-0")  
print(np.size(X\_train\_std0,0), np.size(X\_train\_std0,1))  
  
print("\nBefore Dimension Reduction of Class-1")  
print(np.size(X\_train\_std1,0), np.size(X\_train\_std1,1))  
print()  
  
u0 = getZ(X\_train\_std0, mean0)  
u1 = getZ(X\_train\_std1, mean1)  
  
print("Class-0 Data : ")  
print(u0)  
print("\nClass-1 Data : ")  
print(u1)  
print()  
  
def getScatterMat(Z):  
 return np.dot(np.array(Z).T, np.array(Z))  
  
  
m0 = getScatterMat(u0)  
m1 = getScatterMat(u1)  
print("SCATTER MATRIX FOR CLASS 0 : ")  
print(m0)  
print("\nSCATTER MATRIX FOR CLASS 1 : ")  
print(m1)  
print()  
  
#Sw is the sum of the scatter matrix for both the classes  
sw = np.add(np.array(m0), np.array(m1))  
  
t1 = np.subtract(np.array(mean0), np.array(mean1))  
t2 = t1[np.newaxis]  
print(np.size(t2,1))  
t3 = t2.T  
print(np.size(t3,0))  
  
#Sb is the between class scatter calculated using the multiplation of difference transpose and difference of the mean matrices  
sb = np.matmul(t3, t2)  
print(sb)  
  
#Sb-1Sw for eigen vector calculation  
req = np.matmul(np.linalg.inv(sw), sb)  
  
#Calculation of eigen values and vectors  
w, v = np.linalg.eig(req)  
eigen\_pairs = [(np.abs(w[i]), v[:, i]) for i in range(len(w))]  
  
#sorting the eigen vectors based on eigen values  
eigen\_pairs.sort(key=lambda k: k[0], reverse=True)  
  
#taking reduced dimensionality to 4  
matw = np.hstack((eigen\_pairs[0][1][:, np.newaxis], eigen\_pairs[1][1][:, np.newaxis], eigen\_pairs[2][1][:, np.newaxis], eigen\_pairs[3][1][:, np.newaxis]))  
  
data0 = np.matmul(matw.T, u0.T)  
data1 = np.matmul(np.array(matw).T, u1.T)  
  
print("After Dimension Reduction of Class-0 ")  
print(data0)  
print("\nAfter Dimension Reduction of Class-1 ")  
print(data1)  
print("\nDimensions After LDA of Ionosphere dataset = ", np.size(data1,0))

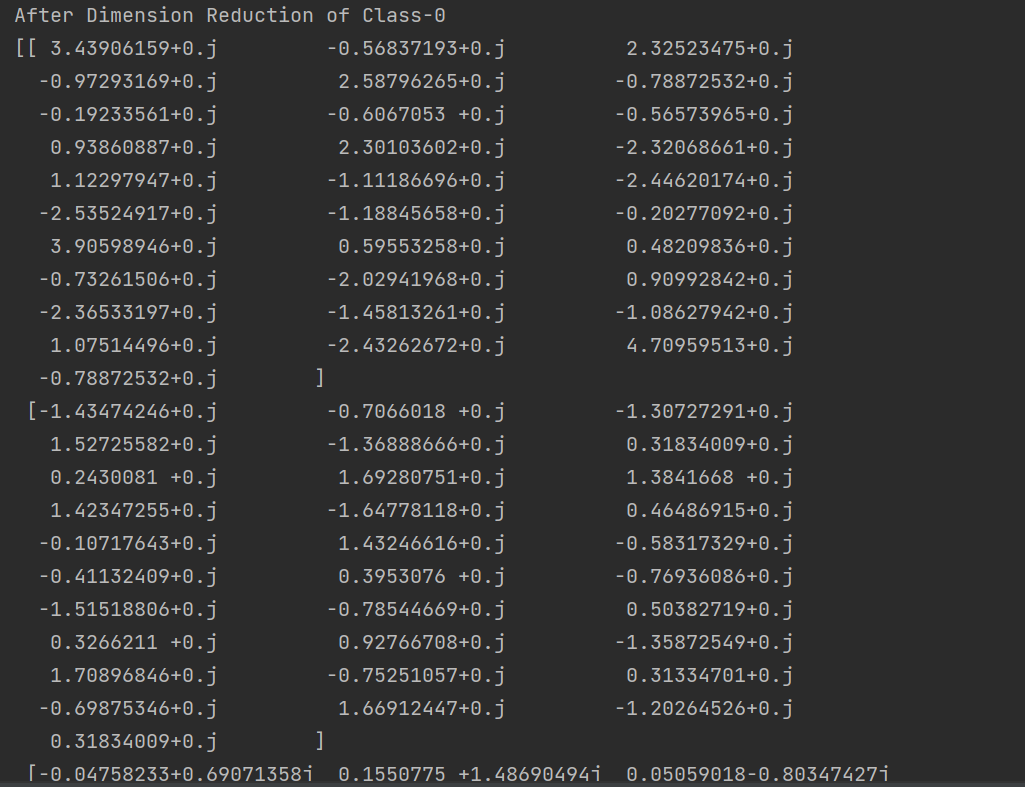
**INPUT**

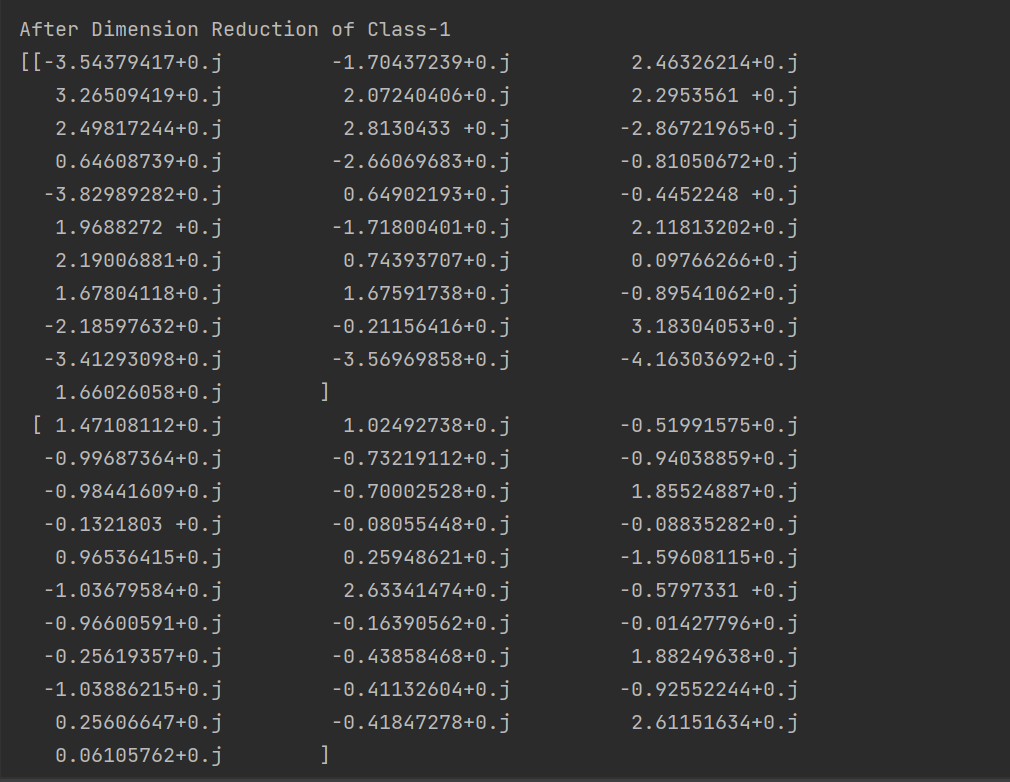
* Ionosphere dataset is used. Any other dataset can also be used.
* The dimensions of the dataset are reduced from 34 to 4.

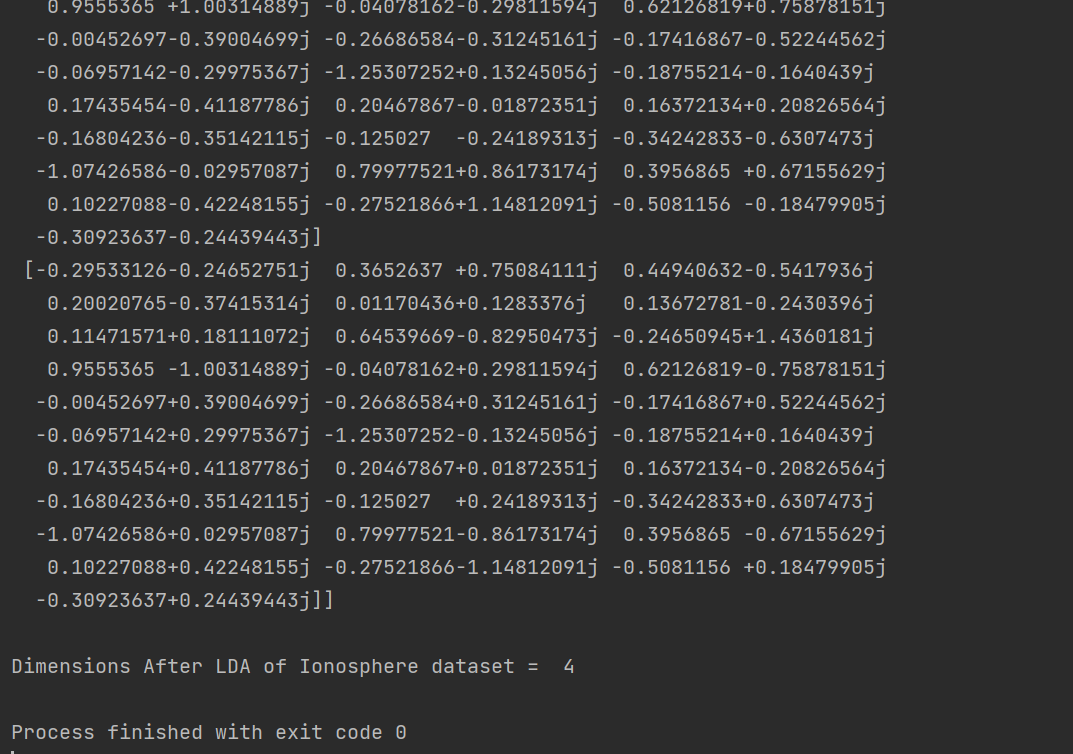
**OUTPUT**



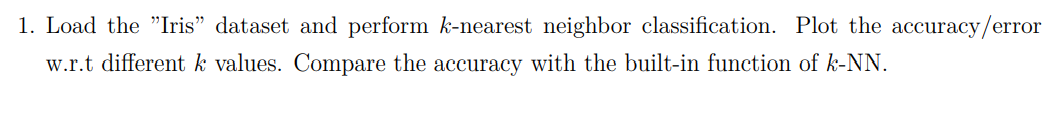








**DAY-6**



**CODE**

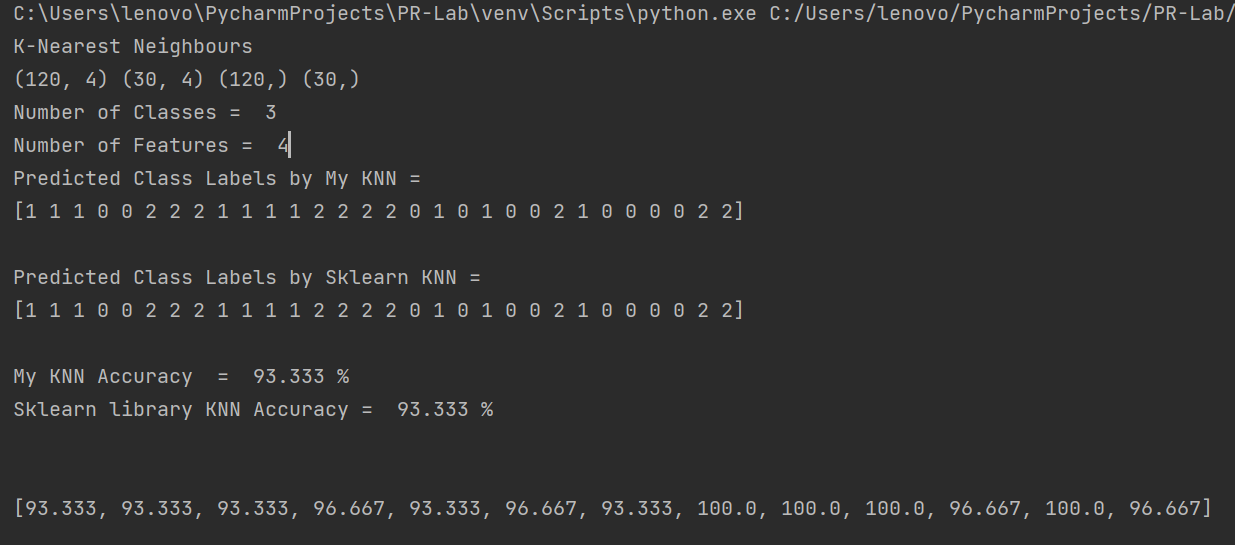
import numpy as np  
from sklearn import datasets  
from sklearn.model\_selection import train\_test\_split  
from sklearn.neighbors import KNeighborsClassifier  
from collections import Counter  
import matplotlib.pyplot as plt  
  
print("K-Nearest Neighbours ")  
# HELPER FUNCTIONS  
  
def Euclidean\_distance(a, b):  
 # just in case, if the instances are lists or tuples:  
 a = np.array(a)  
 b = np.array(b)  
 #ed = np.sqrt(np.sum((a - b)\*\*2)) # or ed = np.linalg.norm(a - b)  
 ed = np.linalg.norm(a - b)  
 return ed  
  
def get\_K\_neighbours(X\_train, y\_train, test\_sample, k):  
 # calculate the distances of the test sample from each training sample  
 distances = [Euclidean\_distance(x, test\_sample) for x in X\_train]  
  
 # sort the distances and get the k neighbours with smallest distance  
 k\_nearest = np.argsort(distances)[:k]  
  
 # get the class labels of all k-nearest neighbours  
 k\_nearest\_labels = [y\_train[i] for i in k\_nearest]  
 return k\_nearest\_labels  
  
def get\_majority\_class(k\_nearest\_labels):  
 # select the most frequent class labels amongst the k-nearest neighbours of the test sample  
 most\_common\_class = Counter(k\_nearest\_labels).most\_common(1)[0][0]  
 return most\_common\_class  
  
def knn\_predict(X\_train, y\_train, X\_test, k=3):  
 y\_pred = []  
 for test\_sample in X\_test:  
 k\_nearest\_labels = get\_K\_neighbours(X\_train, y\_train,test\_sample, k)  
 prediction = get\_majority\_class(k\_nearest\_labels)  
 y\_pred.append(prediction)  
 return np.array(y\_pred)  
  
def sklearn\_knn\_algo(X\_train, y\_train, X\_test, k=3):  
 # using the sklearn library for comparison  
 knn = KNeighborsClassifier(n\_neighbors=k)  
 knn.fit(X\_train, y\_train)  
 y\_pred = knn.predict(X\_test)  
 return y\_pred  
  
def accuracy(actual, predicted):  
 correct = 0  
 for i in range(len(actual)):  
 if actual[i] == predicted[i]:  
 correct += 1  
 a = correct / float(len(actual)) \* 100.0  
 return round(a,3)  
  
  
  
# Loading the Dataset- iris  
dataset = datasets.load\_iris()  
target\_iris\_names = list(dataset.target\_names)  
X = dataset.data # input features  
y = dataset.target # target features  
  
# Stratified split of training and testing data  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, train\_size = 0.80, random\_state = 41, stratify = y)  
print(X\_train.shape, X\_test.shape, y\_train.shape, y\_test.shape)  
  
# Initializing the count of each of all classes of flowers - Versicolor,Setosa,Virginica  
# let the total number of classes be 'C'  
C = len(np.unique(np.array(y\_test)))  
  
# set the total number of features/ dimensions  
M = X.shape[1]  
  
print("Number of Classes = ",C)  
print("Number of Features = ",M)  
  
# K-NEAREST NEIGHBOURS ALGORITHM  
# set the value of k  
k = 3  
# predicting the class labels for test data  
y\_pred = knn\_predict(X\_train, y\_train, X\_test, k)  
my\_accuracy = accuracy(y\_test, y\_pred)  
  
# Running Knn sklearn library function for comparison  
sklearn\_ypred = sklearn\_knn\_algo(X\_train, y\_train, X\_test, k)  
sklearn\_accuracy = accuracy(y\_test, sklearn\_ypred)  
  
# Comparing Results  
print("Predicted Class Labels by My KNN = \n" + str(y\_pred))  
print()  
print("Predicted Class Labels by Sklearn KNN = \n" + str(sklearn\_ypred))  
print()  
print("My KNN Accuracy = ",my\_accuracy, "%")  
print("Sklearn library KNN Accuracy = ", sklearn\_accuracy, "%")  
print("\n")  
  
"""  
for i in range(len(y\_pred)):  
 print("Test data = ", X\_test[i])  
 print("Class label = ", y\_pred[i])  
 print(i," Predicted Value = iris ",target\_iris\_names[int(y\_pred[i])])  
"""  
  
# plotting the accuracy vs k values  
k\_values = [i for i in range(2,15)]  
y\_preds = []  
accuracies = []  
for i in range(2,15):  
 cur\_pred = knn\_predict(X\_train, y\_train, X\_test, i)  
 y\_preds.append(cur\_pred)  
 a = accuracy(y\_test, cur\_pred)  
 accuracies.append(a)  
print(accuracies)  
plt.plot(k\_values, accuracies)

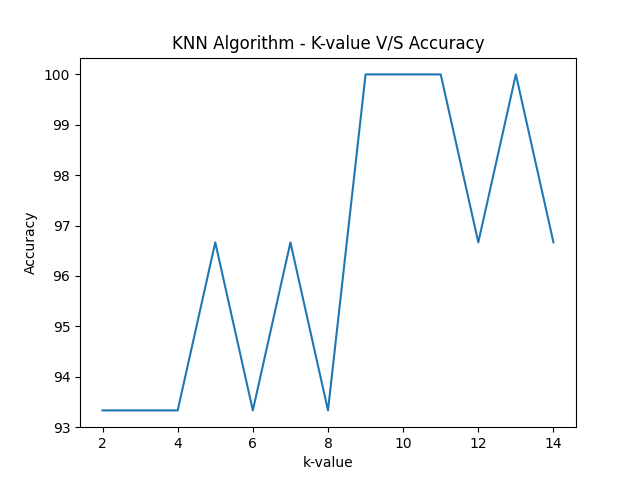
plt.title("KNN Algorithm - K-value V/S Accuracy")  
plt.xlabel("k-value")  
plt.ylabel("Accuracy")  
plt.show()

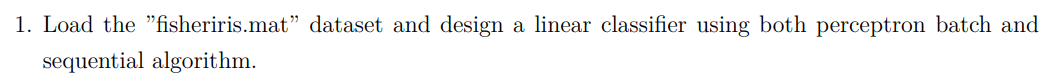
**INPUT**

K-value for running algorithm = 3 (by default)

K-values from 2 to 15 to plot the graph between k and accuracy of algorithm

**OUTPUT**



**DAY-7**

**CODE**

**Perceptron\_sequential.py**

import numpy as np  
from matplotlib import pyplot as plt  
import pandas as pd  
from matplotlib.colors import ListedColormap  
  
print("Perceptron Implementation using Python by Yukti Khurana")  
# loading dataset  
df = pd.read\_csv("Iris.csv", usecols=['SepalLengthCm','SepalWidthCm','PetalLengthCm','PetalWidthCm','Species'])  
df.columns = range(df.shape[1])  
X = df.iloc[0:100, [0,2]].values  
  
# Visualizing the dataset  
plt.scatter(X[:50, 0], X[:50, 1], label = 'setosa',marker='x', color='purple')  
plt.scatter(X[50:100, 0], X[50:100, 1], label = 'versicolor',color='green')  
plt.title("Iris Dataset Visualization by Yukti Khurana")  
plt.xlabel("SepalLengthCm")  
plt.ylabel("PetalLengthCm")  
plt.show()  
  
y = df.iloc[0:100, 4].values  
# class label for setosa flower = -1  
# class label for versicolor flower = 1  
y = np.where(y == 'Iris-setosa', -1, 1)  
  
# Initialising the model parameters  
learn\_rate = 0.001  
epochs = 50  
errors = []  
# initializing an array for weights which will get updated in each iteration  
weights = np.zeros(1 + X.shape[1])  
  
# function for summing the given matrix inputs and their corresponding weights.  
def net\_input(x, weights):  
 return np.dot(x, weights[1:]) + weights[0]  
  
# prediction function  
def predict(x, weights):  
 return np.where(net\_input(x, weights) >= 0.0, 1, -1)  
  
# model fitting  
# creating an numpy array of the size of train data   
weights = np.zeros(1 + X.shape[1])  
for i in range(epochs):  
 err = 0  
 for xi, target in zip(X,y):  
 update = learn\_rate \* (target - predict(xi, weights))  
 weights[1:] += update\*xi  
 weights[0] += update  
 err += int(update != 0)  
 errors.append(err)  
print(errors)  
  
# observing the drop in misclassification error after each epoch  
plt.plot(range(1, len(errors) + 1), errors, marker='o')  
plt.xlabel('Epochs')  
plt.ylabel('Number of misclassifications')  
plt.show()  
  
# Visualization of the perceptron model  
resolution = 0.02  
# setup marker generator and color map  
markers = ('x', 'o')  
colors = ('purple', 'green')  
cmap = ListedColormap(colors[:len(np.unique(y))])  
  
# plot the decision surface  
x1\_min, x1\_max = X[:, 0].min() - 1, X[:, 0].max() + 1  
x2\_min, x2\_max = X[:, 1].min() - 1, X[:, 1].max() + 1  
  
xx1, xx2 = np.meshgrid(np.arange(x1\_min, x1\_max, resolution), np.arange(x2\_min, x2\_max, resolution))  
  
Z = predict(np.array([xx1.ravel(), xx2.ravel()]).T, weights)  
Z = Z.reshape(xx1.shape)  
  
plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)  
plt.xlim(xx1.min(), xx1.max())  
plt.ylim(xx2.min(), xx2.max())  
  
class\_names = ['setosa','versicolor']  
# plot class samples  
for idx, cl in enumerate(np.unique(y)):  
 plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],alpha=0.7, c=cmap(idx),marker=markers[idx], label=class\_names[idx])  
  
plt.title("Perceptron Model on Iris dataset")  
plt.xlabel("Sepal Length (cm)")  
plt.ylabel("Petal Length (cm)")  
plt.legend()  
plt.show()

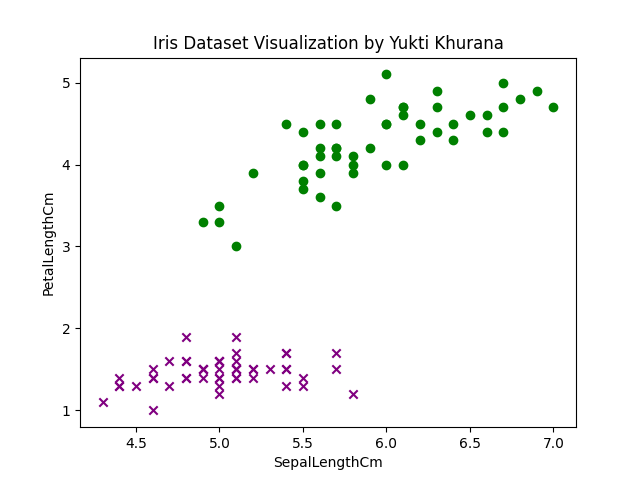
**Perceptron\_batch.py**

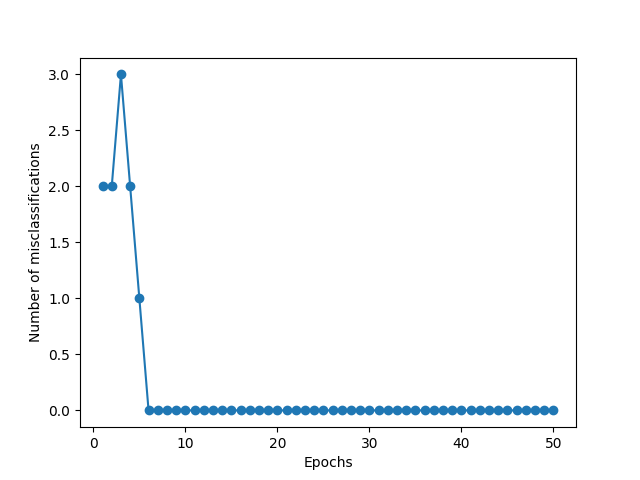
import numpy as np  
from matplotlib import pyplot as plt  
import pandas as pd  
from matplotlib.colors import ListedColormap  
  
print("Perceptron Implementation using Python by Yukti Khurana")  
# loading dataset  
df = pd.read\_csv("Iris.csv", usecols=['SepalLengthCm','SepalWidthCm','PetalLengthCm','PetalWidthCm','Species'])  
df.columns = range(df.shape[1])  
X = df.iloc[0:100, [0,2]].values  
  
# Visualizing the dataset  
plt.scatter(X[:50, 0], X[:50, 1], label = 'setosa',marker='x', color='purple')  
plt.scatter(X[50:100, 0], X[50:100, 1], label = 'versicolor',color='green')  
plt.title("Iris Dataset Visualization by Yukti Khurana")  
plt.xlabel("SepalLengthCm")  
plt.ylabel("PetalLengthCm")  
plt.show()  
  
y = df.iloc[0:100, 4].values  
# class label for setosa flower = -1  
# class label for versicolor flower = 1  
y = np.where(y == 'Iris-setosa', -1, 1)  
  
# Initialising the model parameters  
learn\_rate = 0.001  
epochs = 40  
errors = []  
# initializing an array for weights which will get updated in each iteration  
weights = np.zeros(1 + X.shape[1])  
  
# function for summing the given matrix inputs and their corresponding weights.  
def net\_input(x, weights):  
 return np.dot(x, weights[1:]) + weights[0]  
  
# prediction function  
def predict(x, weights):  
 return np.where(net\_input(x, weights) >= 0.0, 1, -1)  
  
# model fitting  
# creating an numpy array of the size of train data  
weights = np.zeros(1 + X.shape[1])  
errors = []  
for i in range(epochs):  
 error = y - net\_input(X, weights)  
 weights[1:] += learn\_rate \* X.T.dot(error)  
 weights[0] += learn\_rate \* error.sum()  
 cost = (error \*\* 2).sum() / 2.0  
 errors.append(cost)  
print(errors)  
  
# observing the drop in misclassification error after each epoch  
plt.plot(range(1, len(errors) + 1), errors, marker='o')  
plt.xlabel('Epochs')  
plt.ylabel('Number of misclassifications')  
plt.show()  
  
# Visualization of the perceptron model  
resolution = 0.02  
# setup marker generator and color map  
markers = ('x', 'o')  
colors = ('purple', 'green')  
cmap = ListedColormap(colors[:len(np.unique(y))])  
  
# plot the decision surface  
x1\_min, x1\_max = X[:, 0].min() - 1, X[:, 0].max() + 1  
x2\_min, x2\_max = X[:, 1].min() - 1, X[:, 1].max() + 1  
  
xx1, xx2 = np.meshgrid(np.arange(x1\_min, x1\_max, resolution), np.arange(x2\_min, x2\_max, resolution))  
  
Z = predict(np.array([xx1.ravel(), xx2.ravel()]).T, weights)  
Z = Z.reshape(xx1.shape)  
  
plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)  
plt.xlim(xx1.min(), xx1.max())  
plt.ylim(xx2.min(), xx2.max())  
  
class\_names = ['setosa','versicolor']  
# plot class samples  
for idx, cl in enumerate(np.unique(y)):  
 plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],alpha=0.7, c=cmap(idx),marker=markers[idx], label=class\_names[idx])  
  
plt.title("Perceptron Model on Iris dataset")  
plt.xlabel("Sepal Length (cm)")  
plt.ylabel("Petal Length (cm)")  
plt.legend()  
plt.show()

**INPUT**

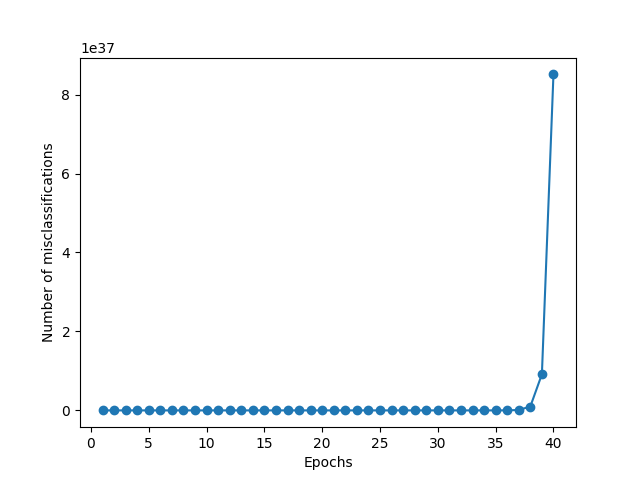
Iris dataset is used. Any other dataset can also be used.

**OUTPUT**

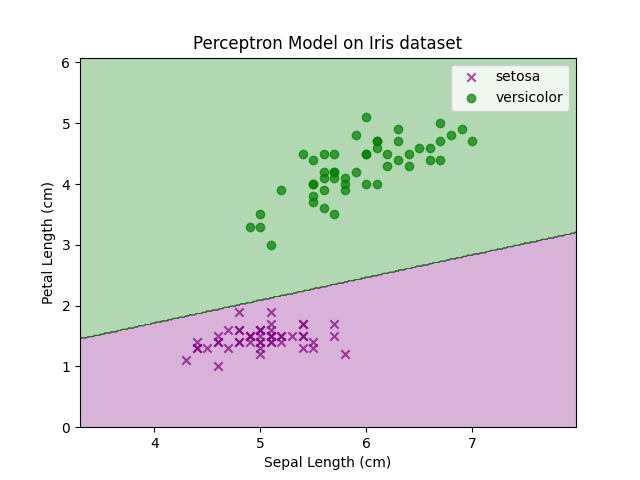




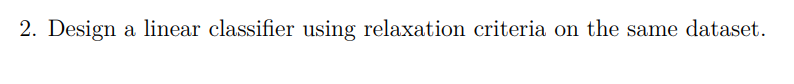
**For Perceptron sequential.py**



**For Perceptron batch.py**



**Visualization of Classification of Iris dataset using Perceptron**



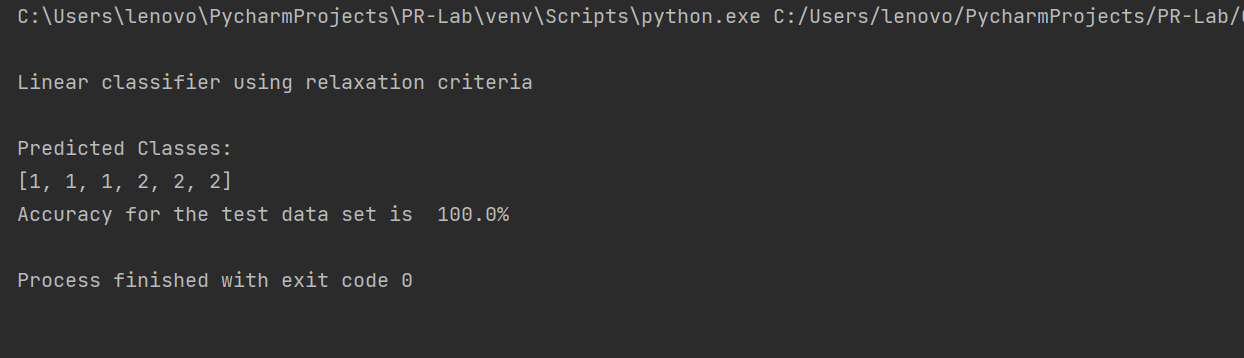
**CODE**

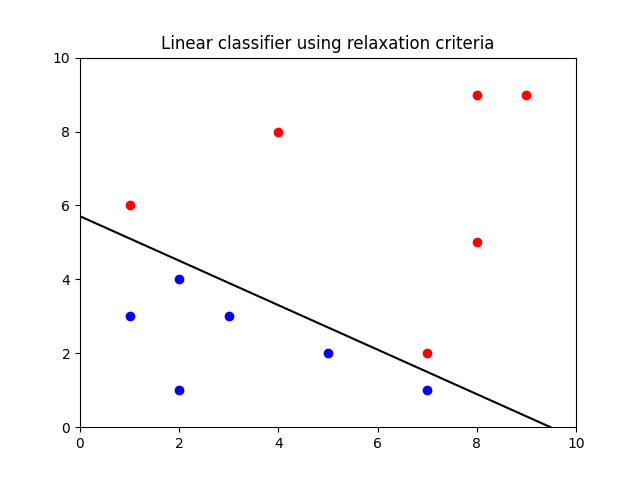
import matplotlib.pyplot as plt  
import numpy as np  
  
def augment\_vector(X):  
 bias = np.ones((len(X), 1))  
 return np.hstack((X, bias))  
  
def normalise\_train\_set(X, Y):  
 X = np.asarray(X)  
 Y = np.asarray(Y)  
 train\_set = augment\_vector(X)  
 labels\_ = np.unique(Y)  
 idx = Y == labels\_[1] # Select one class as negative class  
 train\_set[idx] = -train\_set[idx] # Make the x coordinate of selected class as negative  
 dim = train\_set.shape[1] # Return Dimensionality of feature space // train set  
 return X, Y, train\_set, dim  
  
def linear\_relaxation\_algo(X, Y, learning\_rate, margin):  
 # Obtained the values as required (Augmented and negated)  
 X, Y, train\_set, dim = normalise\_train\_set(X, Y)  
 weights = [0, 0, 1]  
 k = -1  
 i = 0  
 count = 0  
 while i != len(train\_set):  
 k = (k + 1) % len(train\_set)  
 if np.dot(train\_set[k], weights) <= margin:  
 i = 0  
 temp1 = (margin - np.dot(train\_set[k], weights))  
 temp2 = np.dot(train\_set[k], train\_set[k])  
 temp3 = float((float(temp1) / float(temp2)) \* 2)  
 temp = np.dot(temp3, train\_set[k])  
 weights = weights + (learning\_rate \* temp)  
 else:  
 i += 1  
 count += 1  
  
 plot\_boundary(weights, train\_set, X, Y)  
 return weights  
  
  
def plot\_boundary(weights, train\_set, X, Y):  
 # plot data-points  
 x\_points = X[:, 0]  
 y\_points = X[:, 1]  
 length = len(x\_points)  
 length = int(length)  
 x\_points\_1 = x\_points[0:int(length / 2)]  
 x\_points\_2 = x\_points[int(length / 2):length]  
 y\_points\_1 = y\_points[0:int(length / 2)]  
 y\_points\_2 = y\_points[int(length / 2):length]  
  
 plt.plot(x\_points\_1, y\_points\_1, 'ro');  
 plt.axis([0, 10, 0, 10])  
 plt.plot(x\_points\_2, y\_points\_2, 'bo');  
  
 a, b, c = weights  
 xchord\_1 = 0  
 xchord\_2 = -(float(c)) / (float(a))  
 ychord\_2 = 0  
 ychord\_1 = -(float(c)) / (float(b))  
 plt.title("Linear classifier using relaxation criteria")  
 plt.plot([xchord\_1, xchord\_2], [ychord\_1, ychord\_2], 'black')  
 plt.show()  
  
  
def predict(test\_set, weights):  
 test\_set = augment\_vector(test\_set)  
 pred\_list = []  
 for i in range(len(test\_set)):  
 if np.dot(test\_set[i], weights) < 0:  
 pred\_list.append(2)  
 elif np.dot(test\_set[i], weights) > 0:  
 pred\_list.append(1)  
 return pred\_list  
  
  
def compute\_accuracy(pred\_labels\_, Y\_test):  
 count = 0  
 length = len(pred\_labels\_)  
 for k in range(len(pred\_labels\_)):  
 if pred\_labels\_[k] == Y\_test[k]:  
 count = count + 1  
 accuracy = (float(count) / float(length)) \* 100  
 return round(accuracy,3)  
  
def main():  
 print("\nLinear classifier using relaxation criteria \n")  
 X = [(1, 6), (7, 2), (8, 9), (9, 9), (4, 8), (8, 5), (2, 1), (3, 3), (2, 4), (7, 1), (1, 3), (5, 2)]  
 Y = [1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2]  
 testset = [(0, 1), (8, 1), (2, 6), (2, 4.5), (6, 1.5), (4, 3)]  
 test = [(6, 4), (6, 6), (9, 4), (0, 0), (0, -2), (1, 1)]  
 Y\_test = [1, 1, 1, 2, 2, 2]  
 Xnew = [(1, 6), (7, 6), (8, 9), (9, 9), (4, 8), (8, 5), (2, 1), (3, 3), (2, 4), (7, 1), (1, 3), (5, 2)]  
 X\_no\_sep = [(2, 1), (7, 2), (2, 4), (9, 9), (4, 8), (5, 2), (1, 6), (3, 3), (8, 9), (7, 1), (1, 3), (8, 5)]  
  
 learning\_rate = 1  
 margin = 1.0  
 weights = linear\_relaxation\_algo(X, Y, learning\_rate, margin)  
 pred\_labels = predict(test, weights)  
 print("Predicted Classes: ")  
 print(pred\_labels)  
 accuracy = compute\_accuracy(pred\_labels, Y\_test)  
 print('Accuracy for the test data set is {}%'.format(accuracy))  
  
  
main()

**INPUT**

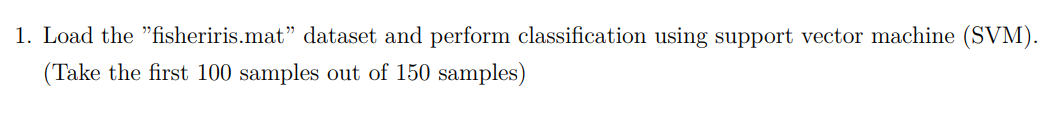
Given Dataset is used. Any other may also be used.

**OUTPUT**





**DAY-8**



**CODE**

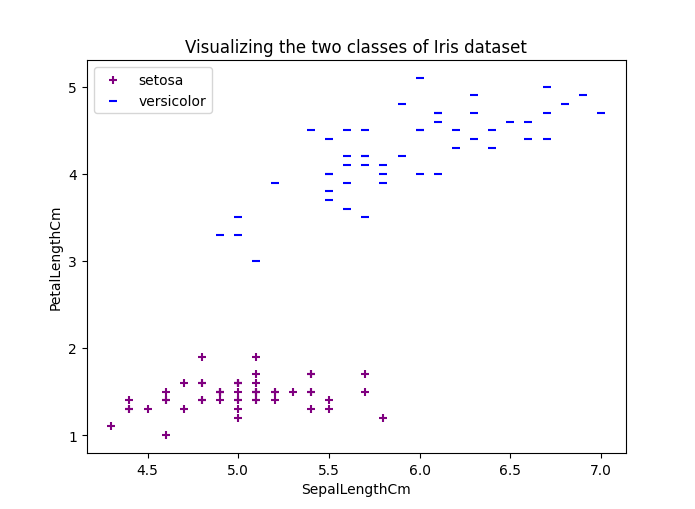
**SVM.py**

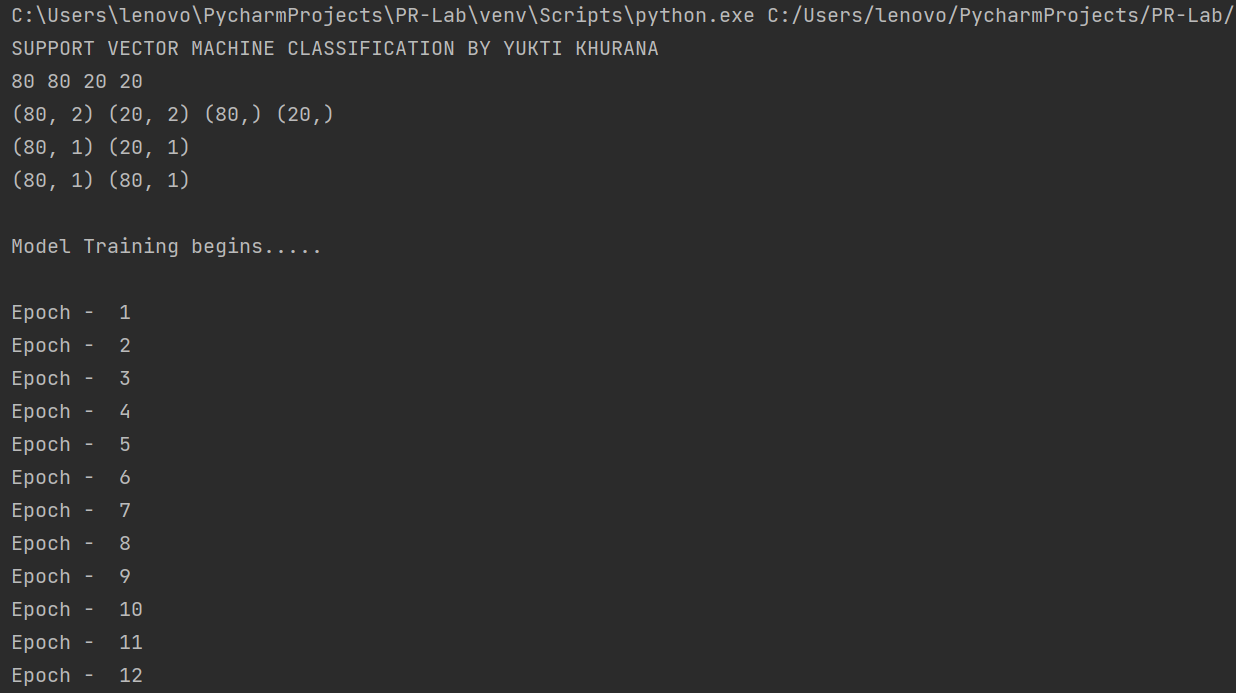
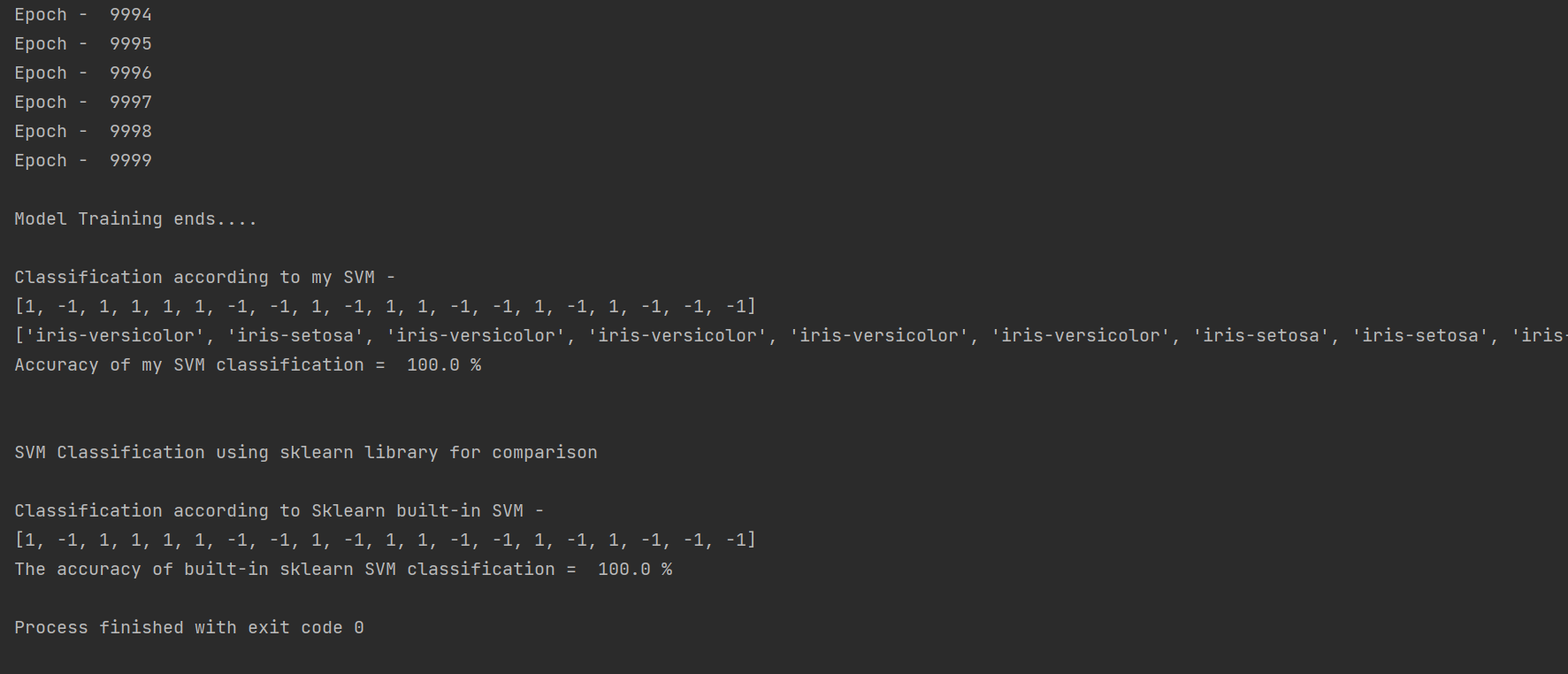
import pandas as pd  
import matplotlib.pyplot as plt  
from sklearn.svm import SVC  
from sklearn.model\_selection import train\_test\_split  
import numpy as np  
  
print("SUPPORT VECTOR MACHINE CLASSIFICATION BY YUKTI KHURANA")  
  
df = pd.read\_csv("Iris.csv")  
df = df.drop(['Id'], axis=1)  
# taking the first 100 samples out of 150 samples  
# converting this into a binary classification problem by considering only two classes  
# two classes - setosa and versicolor  
df = df[0:100]  
# visualising the two classes  
x = df['SepalLengthCm']  
y = df['PetalLengthCm']  
setosa\_x = x[:50]  
setosa\_y = y[:50]  
versicolor\_x = x[50:]  
versicolor\_y = y[50:]  
plt.figure(figsize=(8, 6))  
plt.scatter(setosa\_x, setosa\_y, label = 'setosa',marker='+', color='purple')  
plt.scatter(versicolor\_x, versicolor\_y,label='versicolor', marker='\_', color='blue')  
plt.xlabel("SepalLengthCm")  
plt.ylabel("PetalLengthCm")  
plt.title("Visualizing the two classes of Iris dataset")  
plt.legend()  
plt.show()  
  
# Drop unnecessary features  
df = df.drop(['SepalWidthCm', 'PetalWidthCm'], axis=1)  
# constructing the class labels  
y = []  
target = df['Species']  
for val in target:  
 if (val == 'Iris-setosa'):  
 y.append(-1)  
 else:  
 y.append(1)  
  
df = df.drop(['Species'], axis=1)  
X = df.values.tolist()  
split\_size = 0.80  
# Splitting the training and testing data  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, train\_size=split\_size, random\_state=41, stratify=y)  
print(len(X\_train), len(y\_train), len(X\_test), len(y\_test))  
  
X\_train, X\_test, y\_train, y\_test = np.array(X\_train), np.array(X\_test), np.array(y\_train), np.array(y\_test)  
print(X\_train.shape, X\_test.shape, y\_train.shape, y\_test.shape)  
  
y\_train = y\_train.reshape(X\_train.shape[0], 1)  
y\_test = y\_test.reshape(X\_test.shape[0], 1)  
print(y\_train.shape, y\_test.shape)  
  
# extracting the training features  
# feature 1  
train\_f1 = X\_train[:, 0]  
# feature 2  
train\_f2 = X\_train[:, 1]  
  
  
# extracting the test data features  
test\_f1 = X\_test[:, 0]  
test\_f2 = X\_test[:, 1]  
  
test\_f1 = test\_f1.reshape(X\_test.shape[0], 1)  
test\_f2 = test\_f2.reshape(X\_test.shape[0], 1)  
  
train\_f1 = train\_f1.reshape(X\_train.shape[0], 1)  
train\_f2 = train\_f2.reshape(X\_train.shape[0], 1)  
print(train\_f1.shape, train\_f2.shape)  
  
w1 = np.zeros((X\_train.shape[0], 1))  
w2 = np.zeros((X\_train.shape[0], 1))  
  
# no of epochs  
epochs = 1  
# learning rate  
alpha = 0.0001  
  
# the regularization parameter λ is set to 1/epochs.  
# Therefore, the regularizing value reduces the number of epochs increases.  
print("\nModel Training begins.....\n")  
while (epochs < 10000):  
 y = w1 \* train\_f1 + w2 \* train\_f2  
 prod = y \* y\_train  
 count = 0  
 print("Epoch - ", epochs)  
 for val in prod:  
 if (val >= 1):  
 # no misclassification  
 # cost will be zero  
 # w = w - alpha \* (2\*lambda\*w)  
 cost = 0  
 w1 = w1 - alpha \* (2 \* 1 / epochs \* w1)  
 w2 = w2 - alpha \* (2 \* 1 / epochs \* w2)  
 # When there is a misclassification, i.e our model make a mistake on the prediction of the class of our data point, we include the loss along with the regularization parameter to perform gradient update.  
 else:  
 # misclassification occurs  
 # so gradient updation will involve cost  
 # w = w+ alpha \* (yi \* xi - (2\*lambda\*w))  
 cost = 1 - val  
 w1 = w1 + alpha \* (train\_f1[count] \* y\_train[count] - 2 \* 1 / epochs \* w1)  
 w2 = w2 + alpha \* (train\_f2[count] \* y\_train[count] - 2 \* 1 / epochs \* w2)  
 count += 1  
 epochs += 1  
print("\nModel Training ends....\n")  
  
def accuracy(actual, predicted):  
 correct = 0  
 for i in range(len(actual)):  
 if actual[i] == predicted[i]:  
 correct += 1  
 return correct / float(len(actual)) \* 100.0  
  
  
# Clipping the weights  
index = list(range(X\_test.shape[0], X\_train.shape[0]))  
w1 = np.delete(w1, index)  
w2 = np.delete(w2, index)  
  
# reshaping the weights  
w1 = w1.reshape(X\_test.shape[0], 1)  
w2 = w2.reshape(X\_test.shape[0], 1)  
  
# Prediction  
y\_pred = w1 \* test\_f1 + w2 \* test\_f2  
predictions = []  
for val in y\_pred:  
 if (val > 1):  
 predictions.append(1)  
 else:  
 predictions.append(-1)  
  
pred\_flower = []  
for i in predictions:  
 if i == -1:  
 pred\_flower.append("iris-setosa")  
 else:  
 pred\_flower.append("iris-versicolor")  
  
print("Classification according to my SVM - ")  
print(predictions)  
print(pred\_flower)  
print("Accuracy of my SVM classification = ", accuracy(y\_test, predictions), "%")  
print("\n")  
  
  
print("SVM Classification using sklearn library for comparison\n")  
svc\_clf = SVC(kernel='linear')  
svc\_clf.fit(X\_train,np.ravel(y\_train,order='C'))  
sklearn\_ypred = svc\_clf.predict(X\_test)  
print("Classification according to Sklearn built-in SVM -")  
print(sklearn\_ypred.tolist())  
print("The accuracy of built-in sklearn SVM classification = ",accuracy(y\_test,sklearn\_ypred), "%")

**INPUT**

* Iris Dataset has been used for SVM implementation. Any other dataset may also be used
* epochs = 10000
* learning rate = 0.0001

**OUTPUT**

s



Classification according to my SVM -

[1, -1, 1, 1, 1, 1, -1, -1, 1, -1, 1, 1, -1, -1, 1, -1, 1, -1, -1, -1]

['iris-versicolor', 'iris-setosa', 'iris-versicolor', 'iris-versicolor', 'iris-versicolor', 'iris-versicolor', 'iris-setosa', 'iris-setosa', 'iris-versicolor', 'iris-setosa', 'iris-versicolor', 'iris-versicolor', 'iris-setosa', 'iris-setosa', 'iris-versicolor', 'iris-setosa', 'iris-versicolor', 'iris-setosa', 'iris-setosa', 'iris-setosa']

Accuracy of my SVM classification = 100.0 %

SVM Classification using sklearn library for comparison

Classification according to Sklearn built-in SVM -

[1, -1, 1, 1, 1, 1, -1, -1, 1, -1, 1, 1, -1, -1, 1, -1, 1, -1, -1, -1]

The accuracy of built-in sklearn SVM classification = 100.0 %