

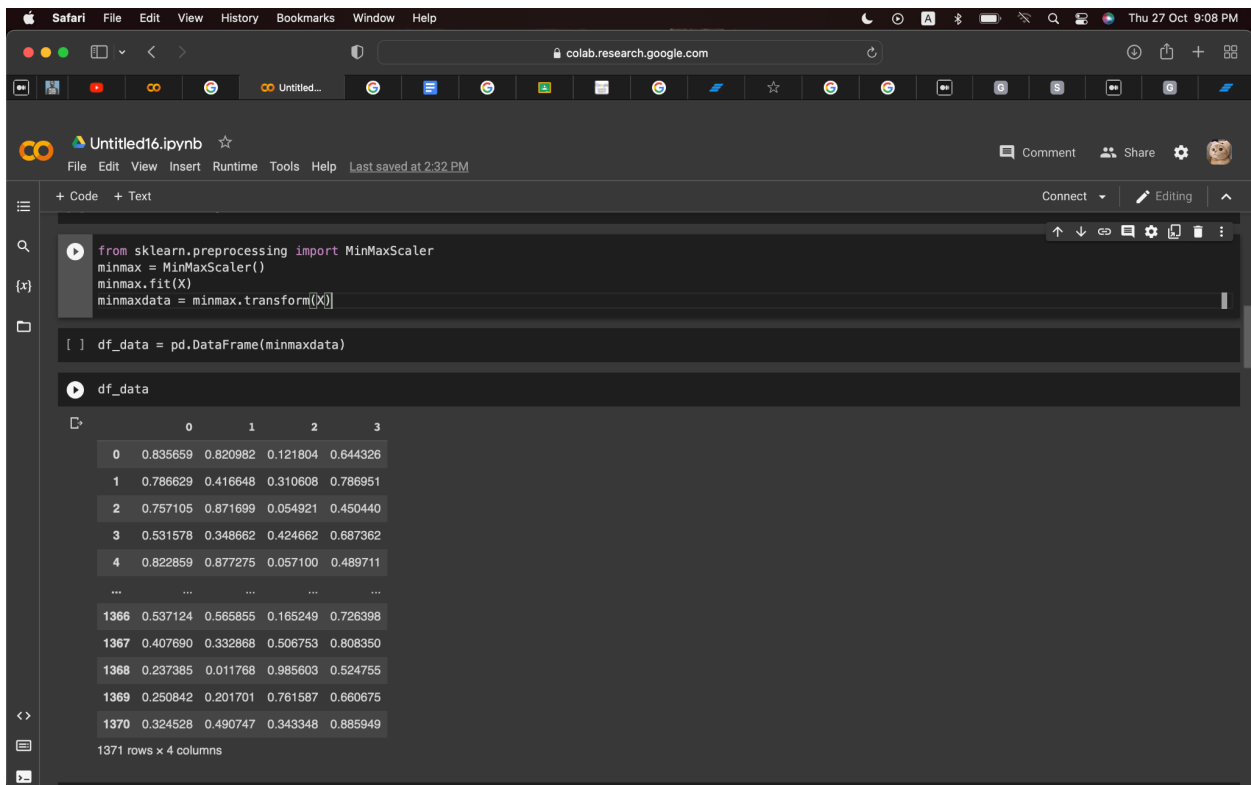
Smruti Dawale
LAB_09
B21BB007

Que 1)

I have removed the Nan- values by dropping it , using the line `data = data.dropna(axis=0)` .

Normalization scales each input variable separately to the range 0-1, which is the range for floating-point values where we have the most precision.

I have done normalization using **MinMaxScaler**, it basically scales values to the range 0 and 1.



The screenshot shows a Google Colab notebook with the following code and output:

```
from sklearn.preprocessing import MinMaxScaler
minmax = MinMaxScaler()
minmax.fit(X)
minmaxdata = minmax.transform(X)
```

```
[ ] df_data = pd.DataFrame(minmaxdata)
```

```
df_data
```

	0	1	2	3
0	0.835659	0.820982	0.121804	0.644326
1	0.786629	0.416648	0.310608	0.786951
2	0.757105	0.871699	0.054921	0.450440
3	0.531578	0.348662	0.424662	0.687362
4	0.822859	0.877275	0.057100	0.489711
...
1366	0.537124	0.565855	0.165249	0.726398
1367	0.407690	0.332868	0.506753	0.808350
1368	0.237385	0.011768	0.985603	0.524755
1369	0.250842	0.201701	0.761587	0.660675
1370	0.324528	0.490747	0.343348	0.885949

1371 rows x 4 columns

Train - test split

Train test split , here you basically split the data into 2 sets , where you use the training set for the training purpose rather than training the whole data.

For dividing the data into 3 parts , I first put the train and validation set together (main)and then the splitted the main into train and validation.

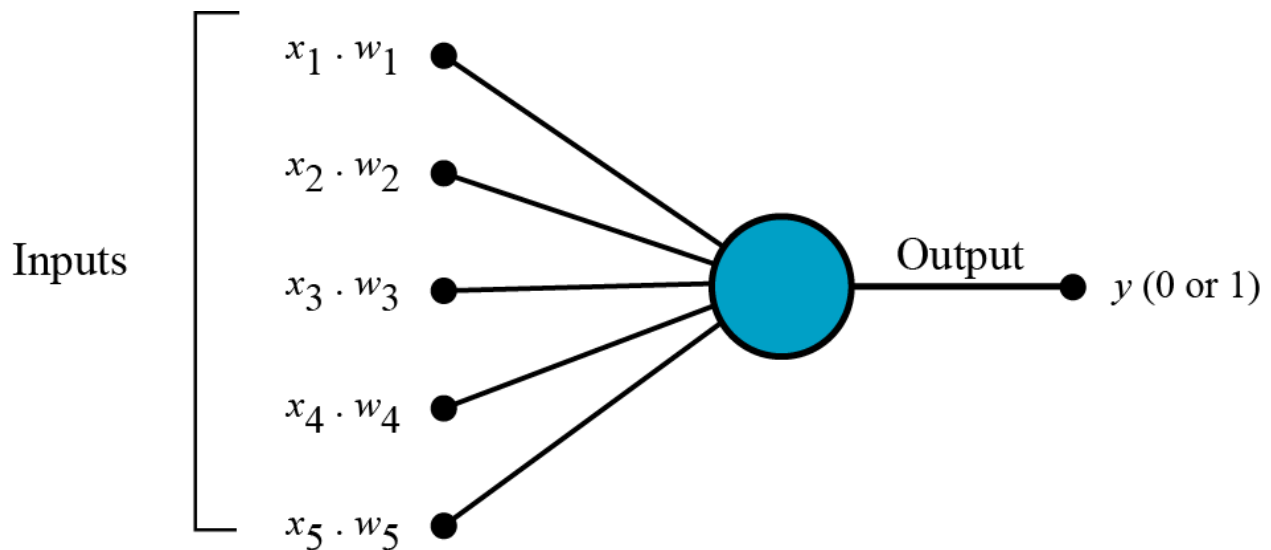
QUE 2)

Perceptron is usually used to classify the data into two parts. Therefore, it is also known as a linear binary classifier.

The perceptron consists of 4 parts.

1. Input values or One input layer
2. Weights and Bias
3. Net sum
4. Activation Function

a. All the inputs x are multiplied with their weights w . Let's call it k . (Weights show the strength of the particular node).



b. Add all the multiplied values and call them W .

c. Apply that weighted sum to the correct Activation Function. And you will get output as 0 or 1.

For Example: Unit Step Activation Function.

The general process of k-fold cross-validation for evaluating a model's performance is:

- The whole dataset is randomly split into independent k-folds without replacement.
- k-1 folds are used for the model training and one fold is used for performance evaluation.
- This procedure is repeated k times (iterations) so that we obtain k number of performance estimates for each iteration.
- Then we get the mean of k number of performance estimates.

The splitting process is done without replacement. So, each observation will be used for training and validation exactly once.

Good standard values for k in k-fold cross-validation are 5 and 10. However, the value of k depends on the size of the dataset. For small datasets, we can use higher values for k. However, larger values of k will also increase the runtime of the cross-validation algorithm and the computational cost.

Here , I have used $K = 5$, and got the following accuracy for each set.

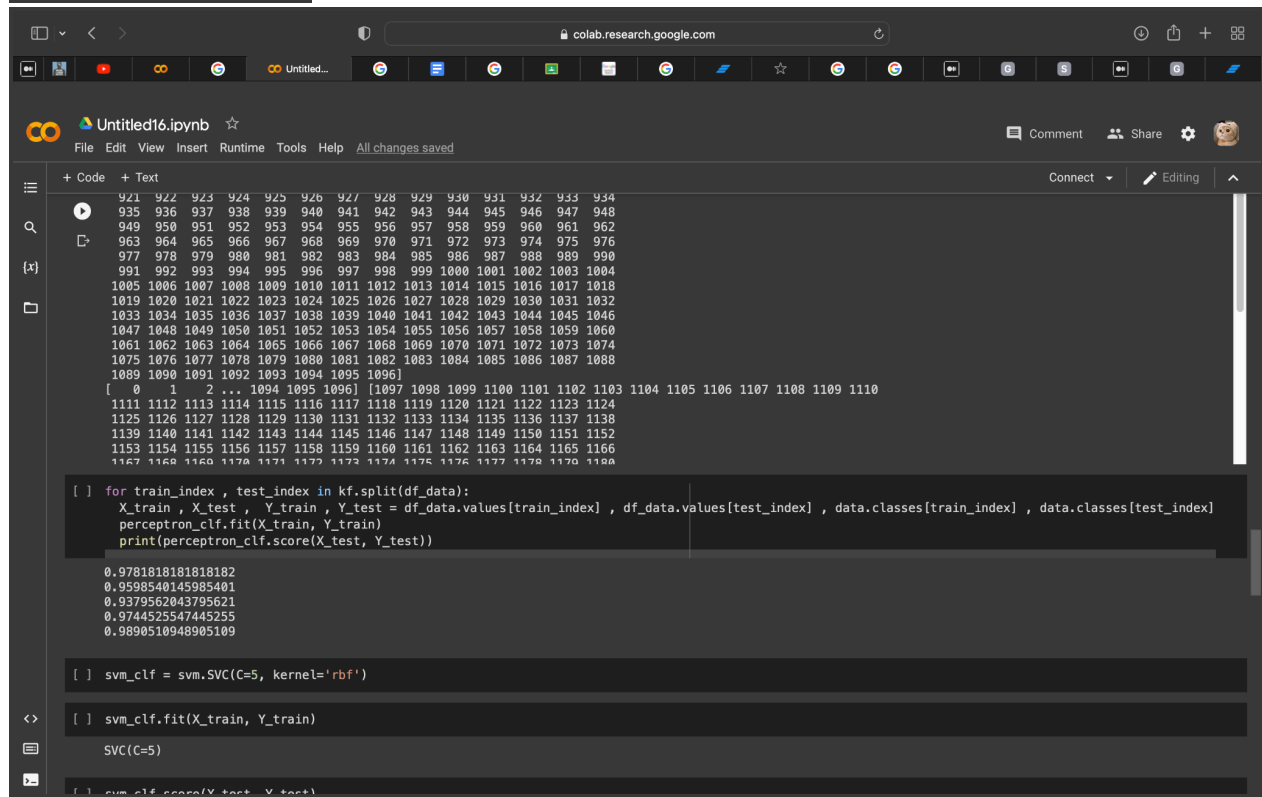
```
0.9781818181818182
```

```
0.9598540145985401
```

```
0.9379562043795621
```

```
0.9744525547445255
```

0.9890510948905109



The screenshot shows a Google Colab notebook interface. At the top, there's a browser address bar with 'colab.research.google.com'. Below it is a toolbar with various icons. The notebook title is 'Untitled16.ipynb'. The main area contains a grid of handwritten digits (MNIST style) and a code cell. The code cell contains the following Python code:

```
[ ] for train_index , test_index in kf.split(df_data):
    X_train , X_test , Y_train , Y_test = df_data.values[train_index] , df_data.values[test_index] , data.classes[train_index] , data.classes[test_index]
    perceptron_clf.fit(X_train, Y_train)
    print(perceptron_clf.score(X_test, Y_test))

0.9781818181818182
0.9598540145985401
0.9379562043795621
0.974452547445255
0.9890510948905109

[ ] svm_clf = svm.SVC(C=5, kernel='rbf')

[ ] svm_clf.fit(X_train, Y_train)

SVC(C=5)

[ ] svm_clf.score(X_test, Y_test)
```

Que 3) I have made a class MPerceptron in which i have defined various functions

First fit function , actually trains the data set , provides the updated bias and Weights and their additions

```
self.weights = self.weights + self.lr*( y[i] - y_predict)*x[i]

self.bias = self.bias + self.lr*( y[i] - y_predict)
```

Second activation function , if the sum is > than 0 , it gives the outcome 1 else the outcome remains 0.

Third function is the predict function which predicts the y-pred.

The screenshot shows a Jupyter Notebook titled 'Untitled16.ipynb' in a Safari browser window. The notebook contains the following Python code:

```
def __init__(self , learning_rate= 0.1 , n_iterations = 500 ):
    self.lr = learning_rate
    self.epochs = n_iterations
    self.weights = None
    self.bias = None

def fit(self , x , y):

    self.weights = np.zeros(x.shape[1])
    self.bias = 0

    for epoch in range(self.epochs):

        for i in range(x.shape[0]) :

            y_predict = self.ypre(np.dot(self.weights , x[i]) + self.bias )

            self.weights = self.weights + self.lr*( y[i] - y_predict)*x[i]
            self.bias = self.bias + self.lr*( y[i] - y_predict)

        print(self.weights)
        print(self.bias)

def ypre(self , ypre ):
    if ypre >0 :
        return 1
    else :
        return 0
```

The screenshot shows the same Jupyter Notebook with the following code added to the next cell:

```
def predict(self , x):

    y_pred = []
    for i in range(x.shape[0]):
        np.dot(self.weights ,x[i])

    y_pred.append(self.ypre(np.dot(self.weights , x[i]) + self.bias ))

    return np.array(y_pred)
```

Below the code, the notebook shows the execution of the following commands:

```
[ ] clf = MPerceptron()

[ ] clf.fit(x , y)

[-1.72706827 -1.38316408 -1.68563127  0.26124953]
2.2000000000000006

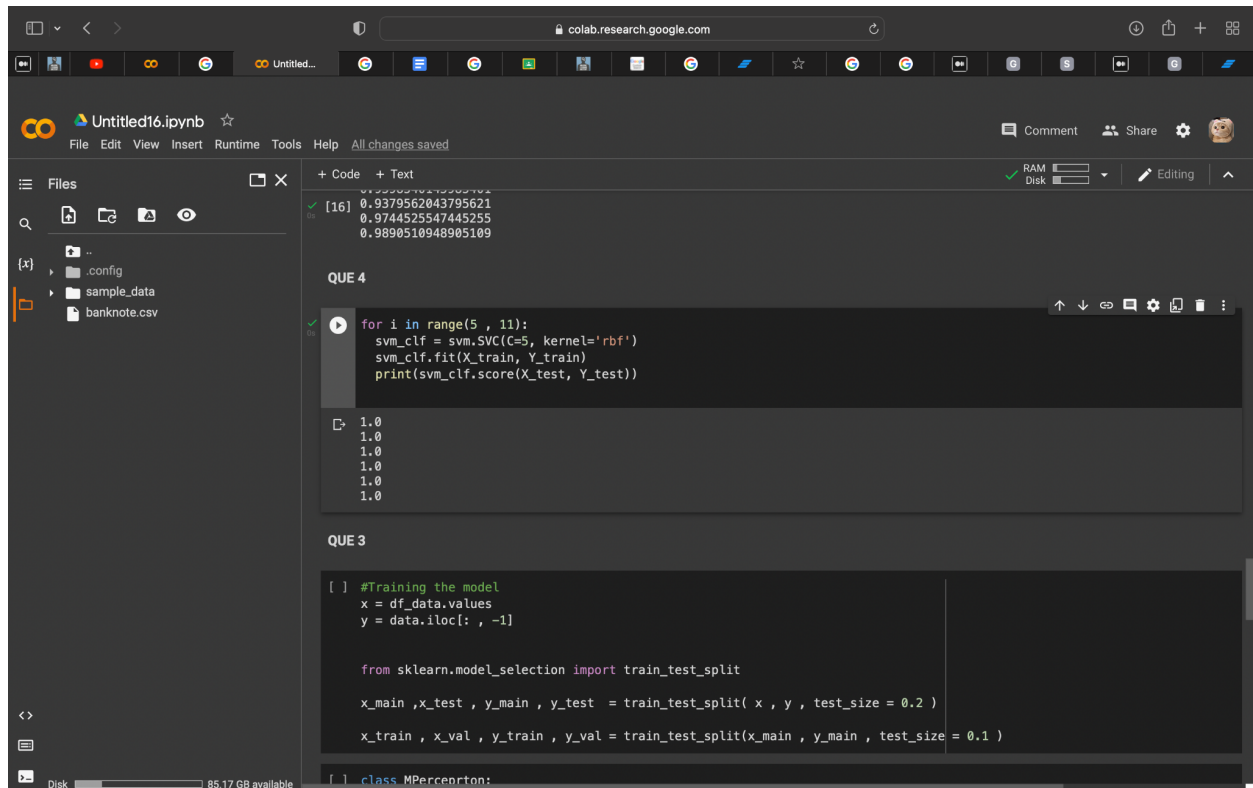
[ ] ypred = clf.predict(x_test)

[ ] print(accuracy_score(y_test , y_pred ) )
```

Que 4 .

A support vector machine is a Machine learning model that is able to generalize between two different classes if the set of labeled data is provided in the training set to the algorithm. The main function of the SVM is to check for that hyperplane that is able to distinguish between the two classes.

For different values of C the accuracy is 1 , this is because the data is not linearly separable.



The screenshot shows a Google Colab notebook titled 'Untitled16.ipynb'. The left sidebar displays a file explorer with a folder named 'sample_data' containing a file 'banknote.csv'. The main area shows two code cells. The top cell, labeled 'QUE 4', contains a loop that iterates over values of C (5, 11) and prints the score of an SVM model with kernel='rbf'. The output of this cell shows three lines of scores: 0.9379562043795621, 0.9744525547445255, and 0.9890510948905109. The bottom cell, labeled 'QUE 3', contains code for training a model using train_test_split and a class MPerceptron. The output of this cell shows the training process and the resulting model.

```
[16] 0.9379562043795621
      0.9744525547445255
      0.9890510948905109
```

QUE 4

```
for i in range(5, 11):
    svm_clf = svm.SVC(C=5, kernel='rbf')
    svm_clf.fit(X_train, Y_train)
    print(svm_clf.score(X_test, Y_test))
```

```
1.0
1.0
1.0
1.0
1.0
1.0
```

QUE 3

```
[ ] #Training the model
x = df_data.values
y = data.iloc[:, -1]

from sklearn.model_selection import train_test_split

x_main, x_test, y_main, y_test = train_test_split( x , y , test_size = 0.2 )

x_train, x_val, y_train, y_val = train_test_split(x_main , y_main , test_size = 0.1 )

[ ] class MPerceptron:
```