**Analysis Report**

**1(a).**

**Generic K-Fold Split Function():**

def NfoldSplit(self,Data,n):

      NewData = []

      size = (Data.shape[0]//n)+1

      for i in range(n):

        T = Data[i\*size:i\*size+size]

        NewData.append(T)

      return NewData

**Predict() code scratch :**

def predict(self,X\_test):

      X\_test = X\_test.to\_numpy()

      y\_predicted =(np.sum(X\_test\*self.Coff,axis = 1)) # self.Coff stores the weights

      return y\_predicted

**Scratch Code for Mean Squared Error :**

def MSE(self,ytrue,ypred):

      MSE = np.square(ytrue-ypred)

      MSE = np.sum(MSE)/len(ytrue)

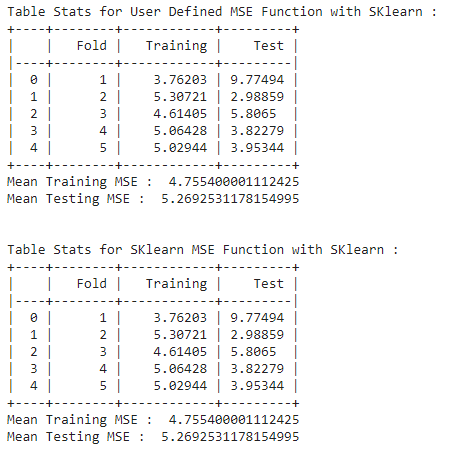
      return MSE

**1(b).**

**Approach :**

* Load the data using the loadmat() function.
* Divide the dataset into a list of folds(as per the number of folds)
* For each fold :
* Train the model using : LinearRegression(fit\_intercept=False).fit(Train[X], Train[Y])
* Predict the values for the training and testing data using the User-defined Predict Method.
* Calculate the MSE over training and testing data using the User Defined Method.
* Calculate the MSE over training and testing data using Sklearn Method.
* Store the above values for each fold.
* Print Table Stats with mean training and testing MSE.

**Output :**



**Observation :**

* The MSE function of Sklearn and user-defined functions give the same results.

**1(c).**

**Approach :**

* Load the data using the loadmat() function.
* Divide the dataset into a list of folds(as per the number of folds)
* For each fold :
* Calculate the regression coefficients using :

Coff = np.matmul(TrainX\_Tr,TrainX)

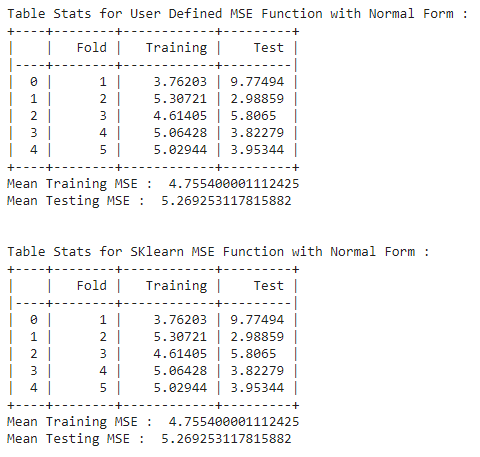
Coff = np.linalg.inv(Coff)

Coff = np.matmul(Coff,TrainX\_Tr)

Coff = np.matmul(Coff,TrainY)

* Predict the values for the training and testing data using the User-defined Predict Method and above calculated coefficients.
* Calculate the MSE over training and testing data using the User Defined Method.
* Calculate the MSE over training and testing data using Sklearn Method.
* Store the above values for each fold.
* Print Table Stats with mean training and testing MSE.

**Output :**



**Observation :**

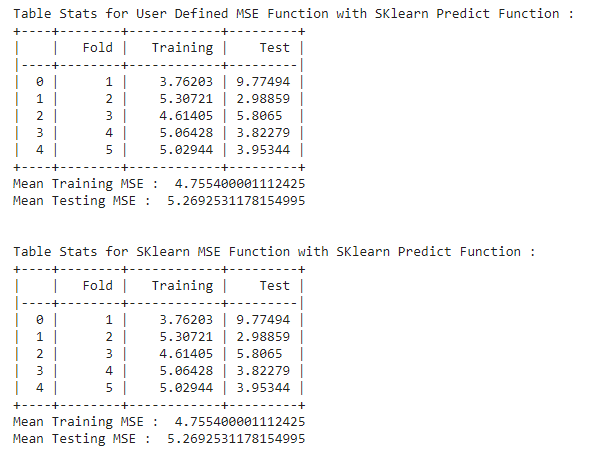
* The performance of both MSE functions is the same.

**1(d).**

**Approach :**

* Load the data using the loadmat() function.
* Divide the dataset into a list of folds(as per the number of folds)
* For each fold :
* Train the model using : LinearRegression(fit\_intercept=False).fit(Train[X], Train[Y])
* Predict the values for the training and testing data using Sklearn Predict Method.
* Calculate the MSE over training and testing data using the User Defined Method.
* Calculate the MSE over training and testing data using Sklearn Method.
* Store the above values for each fold.
* Print Table Stats with mean training and testing MSE.

**Output :**



**Observation :**

* The performance of both methods is the same.
* There is no performance deviation in the three approaches.

**2(a).**

**Scatter Plotting :**

def Scatter\_plot(self,arg):

      Data = self.Load\_DataSet(arg)

      Data = pd.DataFrame(Data, columns =['D1','D2','Label'])

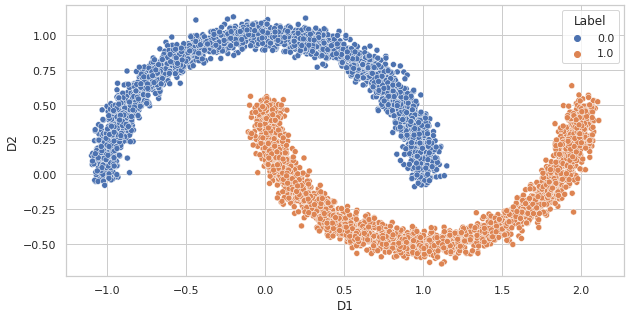
      plt.figure(figsize=(10,5))

      sn.set(style = 'whitegrid')

      sn.scatterplot(data = Data ,x='D1',y='D2',hue='Label')

      plt.show()

**Output :**



**Observation :**

* Class 1 belongs to the data points that follow Concave Distribution.
* Class 0 belongs to the data points that follow Concave Distribution.
* The classes are not linearly separable.

**2(c).**

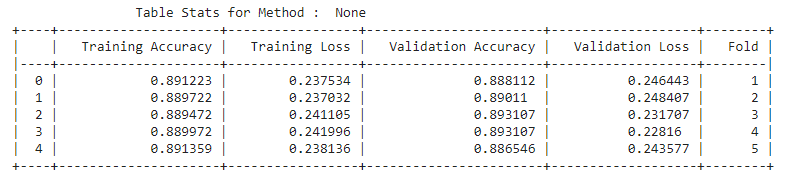
**Approach :**

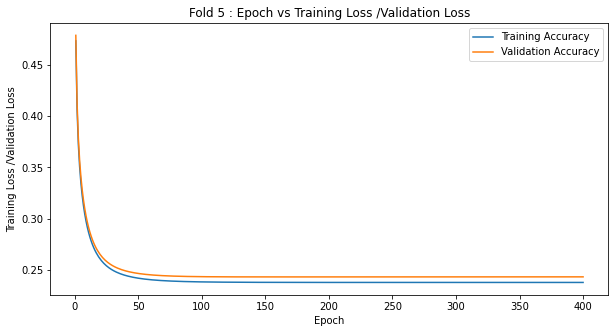
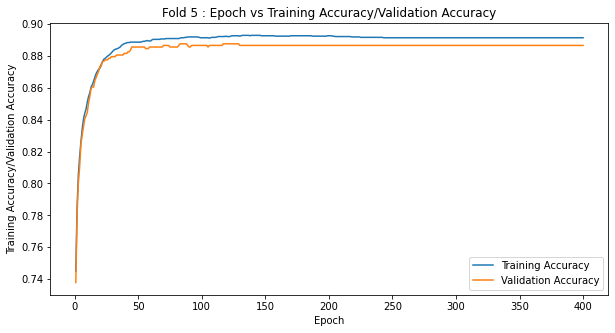
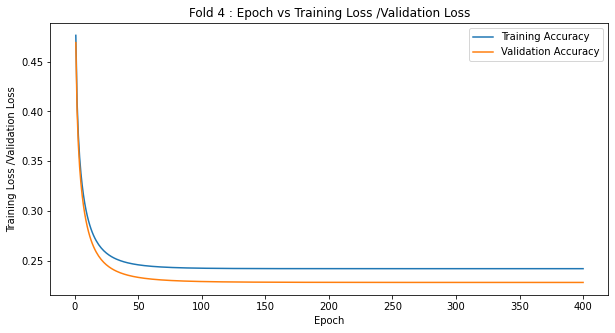
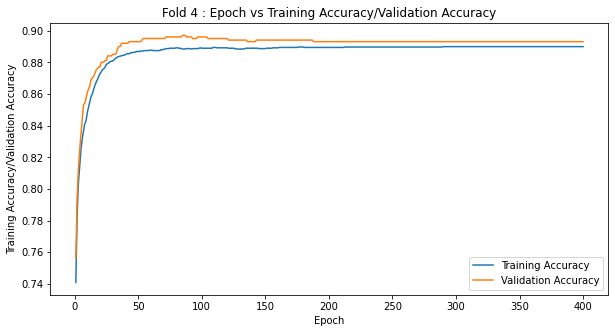
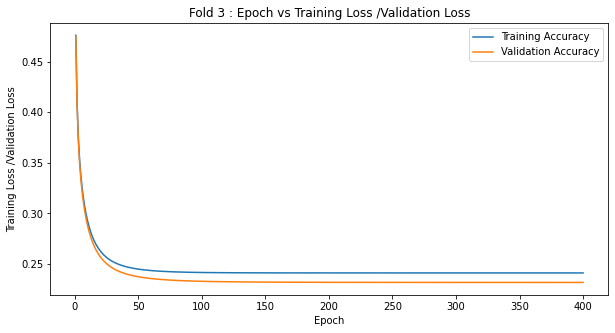
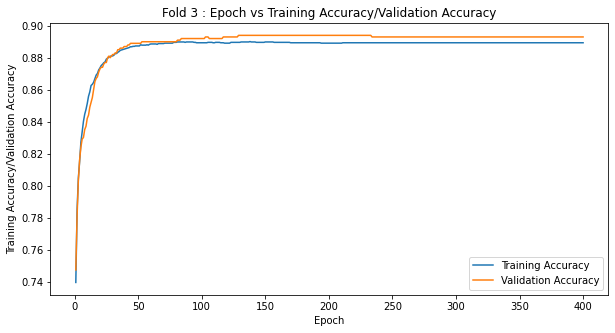
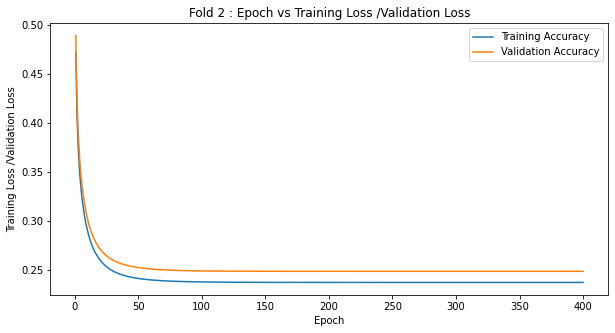
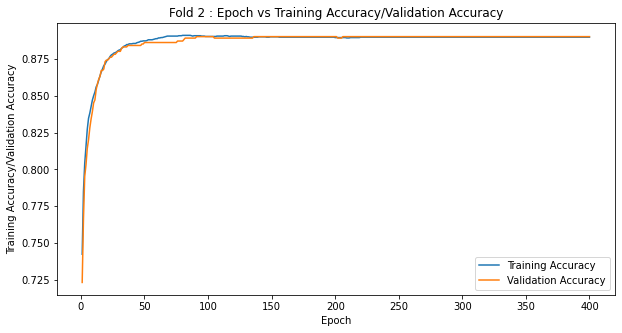
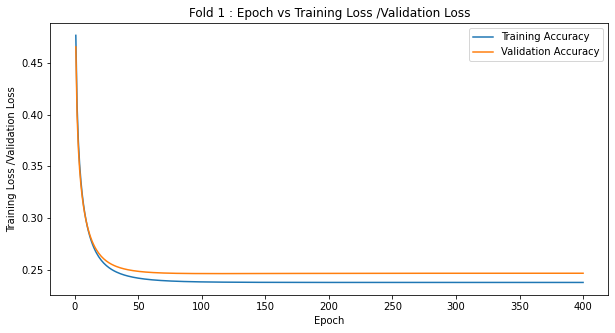
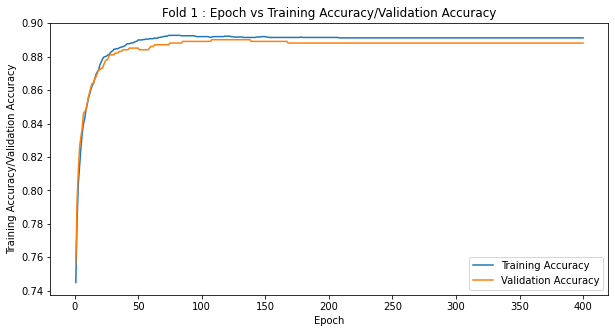
* Load the data using the loadmat() function.
* Divide the dataset into a list of folds(as per the number of folds)
* For each fold :
* Update weights and bias for the specified number of epochs.
* For each epoch, calculate the training and testing accuracy.
* For each epoch, calculate the training and loss accuracy.
* Store the above values for each fold.
* Plot result for all the folds.
* Print Table Stats.

**Hyperparameters Value :**

* Learning Rate = 0.0001
* Epoch = 400

**Output :**



****

**Observation :**

* Consistency of different stats values across different folds.
* From the plots of accuracy and loss curves, it can be seen that as the epochs increases, the training and testing accuracy increases during the training and testing loss decrease.
* Approximately after 100 epochs, the accuracy and loss curves are almost parallel to the X-axis, which means that accuracy and loss values are not increasing and decreasing, respectively.

**2(d).**

**Approach :**

* Load the data using the loadmat() function.
* Divide the dataset into a list of folds(as per the number of folds)
* Create the list of permissible Regularization Constant value.
* For each regularisation constant:
* For each fold :
* Update weights and bias for the specified number of epochs using the Cost function that includes a regularised cost.
* For each epoch, calculate the training and testing accuracy.
* For each epoch, calculate the training and loss accuracy.
* Store the above values for each fold.
* Store the above values for each regularisation constant.
* For each regularisation constant, calculate the average testing accuracy for all the folds and store it in the 'reg' list.
* Extract regularisation constant value that corresponds to the reg list's maximum value and name as optimal regularisation constant.
* Plot result for all the folds using the optimal regularisation constant.
* Print Table Stats for using the optimal regularisation constant.

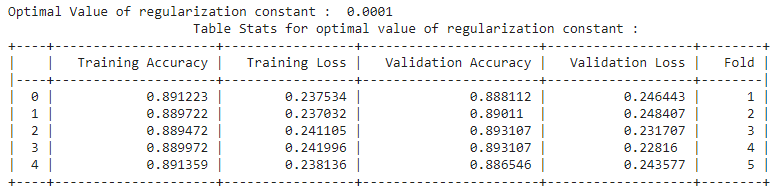
**Hyperparameters Value :**

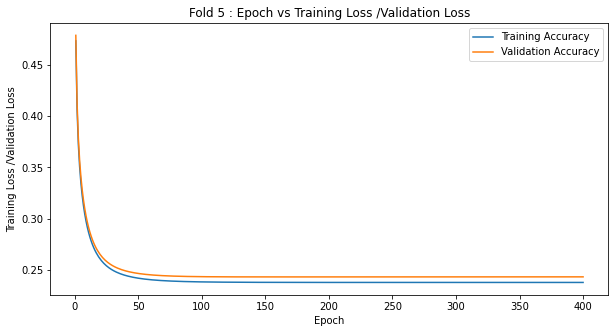
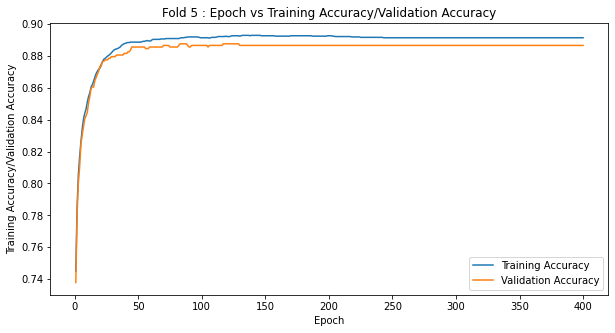
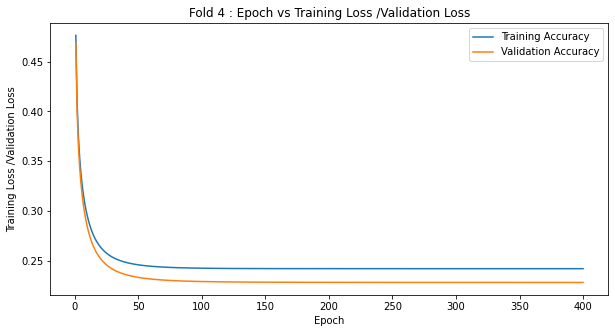
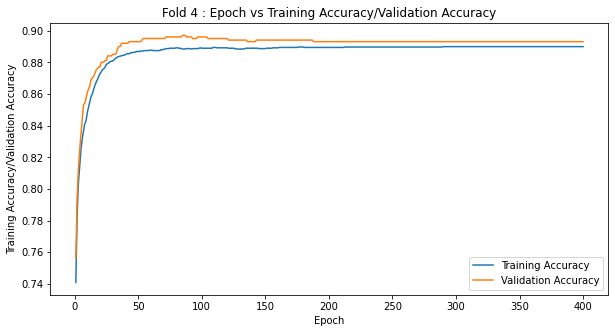
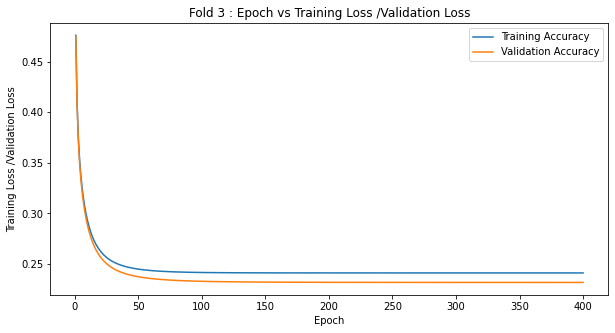
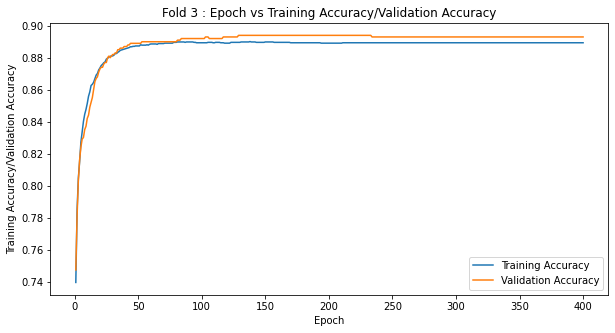
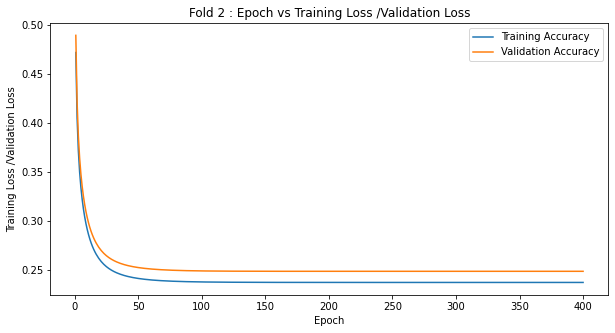
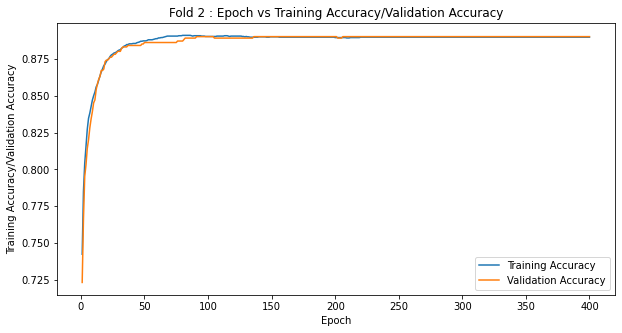
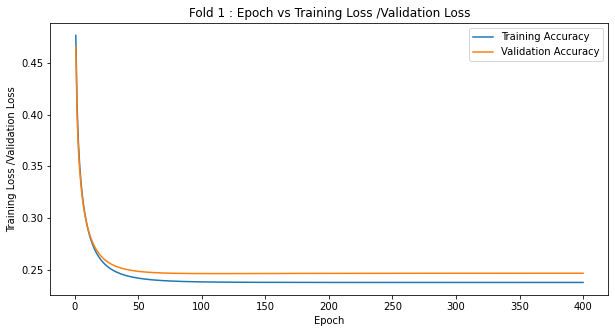
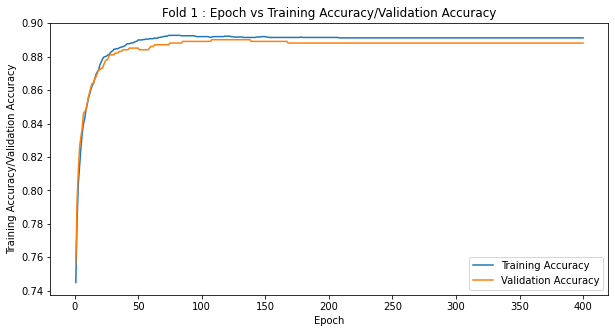
* Learning Rate = 0.0001
* Epoch = 400

**List of Regularization Constants Taken :**

* 0.1
* 0.01
* 0.001
* 0.0001

**Output :**



****

**Observation :**

* Consistency of different stats values across different folds for the optimal value of Regularization Constant.
* From the plots of accuracy and loss curves, it can be seen that as the epochs increases, the training and testing accuracy increases during the training and testing loss decrease.
* Approximately after 100 epochs, the accuracy and loss curves are almost parallel to the X-axis, which means that accuracy and loss values are not increasing and decreasing, respectively.
* The performance of the model across different folds with and without the Regularization Constant is similar.
* This can be observed from Table Stats for both the model.

**2(e).**

**Approach :**

**For Normal Method :**

* Load the data using the loadmat() function.
* Divide the dataset into a list of folds(as per the number of folds)
* For each fold :
* Fit the model using : LogisticRegression(penalty='none', max\_iter=epochs)
* Store the above values of training and testing accuracy for each fold.
* Store the above values of training and testing loss for each fold.
* Print Table Stats.

**For Grid Search of Regularization Constant Using Sklearn :**

* Load the data using the loadmat() function.
* Divide the dataset into a list of folds(as per the number of folds)
* Create the list of permissible Regularization Constant value.
* For each regularisation constant:
* For each fold :
* Fit the model using : LogisticRegression(penalty='l2', C = (1/reg[k]), max\_iter=epochs,solver='newton-cg')
* Store the above values of training and testing accuracy for each fold.
* Store the above values of training and testing loss for each fold.
* Store the above values for each regularisation constant.
* For each regularisation constant, calculate the average testing accuracy for all the folds and store it in the 'reg' list.
* Extract regularisation constant value that corresponds to the reg list's maximum value and name as optimal regularisation constant.
* Print Table Stats for using the optimal regularisation constant.

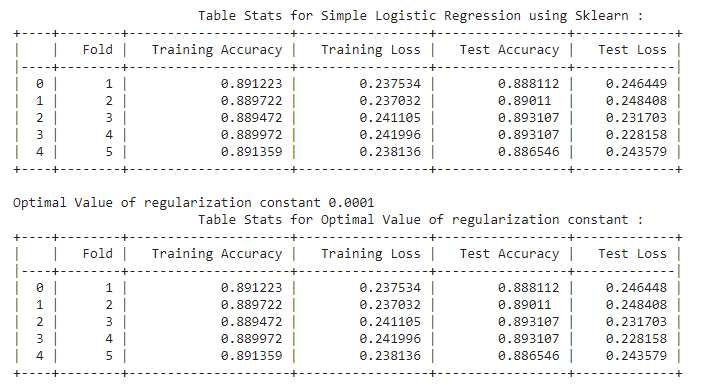
**Hyperparameters Value :**

* Learning Rate = 0.0001
* Epoch = 400

**List of Regularization Constants Taken :**

* 0.1
* 0.01
* 0.001
* 0.0001

**Output :**



**Observation :**

* The optimal value of the Regularization Constant from the Sklearn method and the user-defined method is the same.
* The performance of the model across different folds with sklearn is similar to the performance obtained from the above model trained on a user-defined model from scratch (part (b) and (c)).
* This can be observed from Table Stats for all the three methods.

**3(a).**

**Scatter Plotting :**

def Scatter\_plot(self,arg):

      Data = self.Load\_DataSet(arg)

      Data = pd.DataFrame(Data, columns =['D1','D2','Label'])

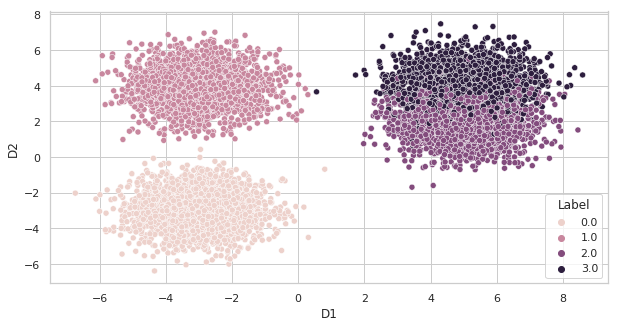
      plt.figure(figsize=(10,5))

      sn.set(style = 'whitegrid')

      sn.scatterplot(data = Data ,x='D1',y='D2',hue='Label')

      plt.show()

**Output :**



**Observation :**

* Class 0 and 1 are linearly separable from each other.
* Class 2 and 3 are not linearly separable from each other. There is an overlapping of data points for these classes.

**3(b).**

**Approach :**

* Load the data using the loadmat() function.
* Divide the dataset into a list of folds(as per the number of folds).
* For each fold :
* Create the list of a different combination of classifiers which is needed to be trained.
* For each classifier(A, B) in the above list :
* Extract the data points from the Training Data that includes the class label as either A or B and then encode the training data to 1/0 format.
* Fit the above training data using the normal logistic regression with Regularization Constant.
* Report the predicted class probability for both training and testing data.
* Store the above values for each classifier in a numpy matrices.
* For each row in the above matrices, extra the label corresponds to the maximum probability value and assign that row with that label.
* Report the accuracy of both training and testing data using the above-prediceted class labels.
* Report the classwise accuracy of both training and testing data using the above-prediceted class labels.
* Store the above accuracies for each fold**.**
* Print Table Stats

**Utility Functions :**

def predict(self,X\_test):

        w = arg[4]

        if self.multi ==True:

          y = []

          y\_predicted = np.dot(X\_test,w.T)

          y\_predicted = 1/(np.exp(-y\_predicted) + 1)

          for i in range(y\_predicted.shape[0]):

            y.append(np.argmax(y\_predicted[i,:]))

          y\_predicted = np.array(y)

        else:

          b = arg[5]

          h = X\_test\*w

          h = np.sum(h,axis=1)

          h = h + b

          h = np.exp(-h) + 1

          h = 1/h

          y\_predicted = h

        return y\_predicted

def Classwise\_Accuracy(self,Ypred,Yact):

      C = np.array(Yact)

      C = list(np.unique(C))

      C.sort()

      Main\_Acc = []

      for i in range(len(C)):

        Pred = []

        Act = []

        for j in range(len(Ypred)):

          if(Yact[j]==C[i]):

            Pred.append(Ypred[j])

            Act.append(Yact[j])

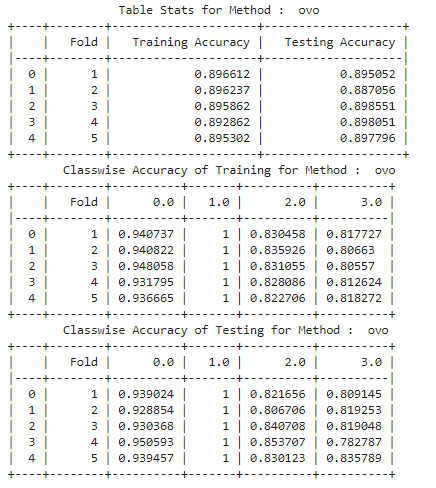
        Main\_Acc.append(self.accuracy(Act,Pred))

      return Main\_Acc

**Hyperparameters Value :**

* Learning Rate = 0.0001
* Epoch = 400
* Regularization Constant = 0.0001

**Output :**



**Observations :**

* The training and testing accuracy across each fold is similar, which shows that rightly fits the data.
* The classwise training and testing accuracy for the 0 and 1 are high compared to the 2 and 3.This is mainly because of the separation of class 1 from the other classes.

**3(c).**

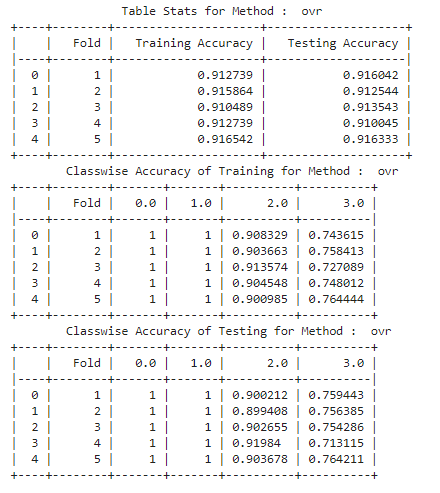
**Approach :**

* Load the data using the loadmat() function.
* Perform the one-hot encoding of the loaded data.
* Divide the dataset into a list of folds(as per the number of folds).
* For each fold :
* Fit the training data with the C(classes ) number of logistic regression classifiers using the normal logistic regression equations with Regularization Constant.
* Report the predicted class label for both training and testing data using the learned weights.
* Report the accuracy of both training and testing data using the above-predicted class probabilities.
* Report the classwise accuracy of both training and testing data using the above-predicted class labels.
* Store the above accuracies values for each fold**.**
* Print Table Stats

**Hyperparameters Value :**

* Learning Rate = 0.0001
* Epoch = 400
* Regularization Constant = 0.0001

**Output :**



**Observation :**

* The training and testing accuracy across each fold is similar, which shows that rightly fits the data.
* These accuracies are high as compared to the above 'ovo' model.
* The classwise training and testing accuracy for the 0 and 1 are high(100 %) compared to the 2 and 3.This is mainly because of the separation of classes 0 and 1 from the other classes.

**3(d).**

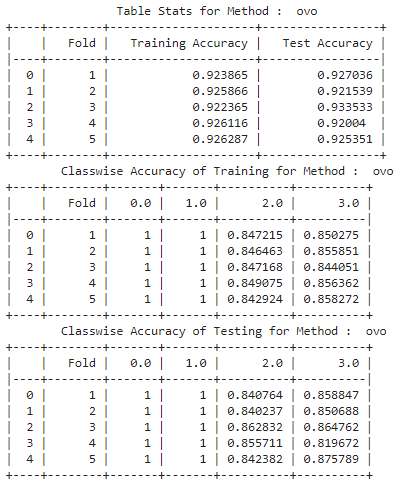
**Approach for OVO :**

* Load the data using the loadmat() function.
* Divide the dataset into a list of folds(as per the number of folds).
* For each fold :
* Fit the above training data using : OneVsOneClassifier(LogisticRegression(penalty='l2',max\_iter=epochs,C = (1.0/reg)))
* Predict the class label for both training testing data.
* Report the accuracy of both training and testing data using the above-predicted class labels.
* Report the classwise accuracy of both training and testing data using the above-predicted class labels.
* Store the above accuracies for each fold**.**
* Print Table Stats

**Hyperparameters Value :**

* Learning Rate = 0.0001
* Epoch = 400
* Regularization Constant = 0.0001

**Output :**



**Observation :**

* As compared to the user-defined ovo method, both training and testing accuracies increase by 3%(approx).
* The classwise accuracies for training and testing data across each fold also increase by some percentage.

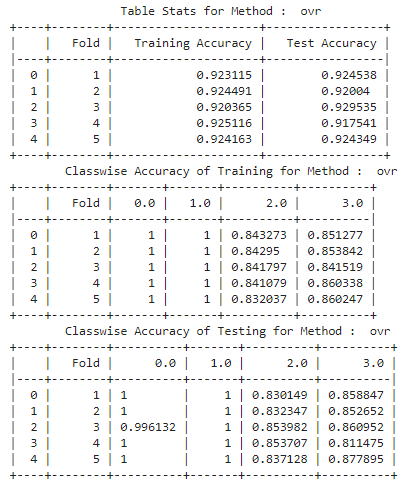
**Approach for OVR :**

* Load the data using the loadmat() function.
* Divide the dataset into a list of folds(as per the number of folds).
* For each fold :
* Fit the above training data using : LogisticRegression(penalty='l2', multi\_class = ‘ovr’ ,max\_iter=epochs,C = (1.0/reg))
* Predict the class label for both training testing data.
* Report the accuracy of both training and testing data using the above-predicted class labels.
* Report the classwise accuracy of both training and testing data using the above-predicted class labels.
* Store the above accuracies for each fold**.**
* Print Table Stats

**Hyperparameters Value :**

* Learning Rate = 0.0001
* Epoch = 400
* Regularization Constant = 0.0001

**Output :**



**Observation :**

* As compared to the user-defined ovr method, both training and testing accuracies increase by 1%(approx).
* The classwise accuracies for training and testing data across each fold also increase by some percentage.