**CS6301 MACHINE LEARNING LAB WEEK – 8 KNN, KMEANS, ENSEMBLE LEARNING**

**SRIHARI. S – 2018103601**

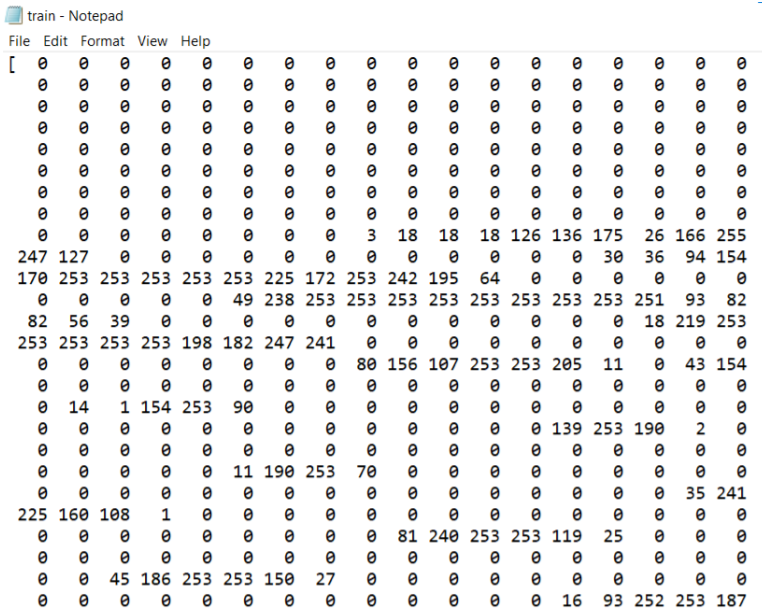
**Date**: 10-04-2021 Saturday

**Aim**: To implement ensemble learning algorithm using various classifiers – SVM, MLP, Decision Trees and KNN and determine the results.

**Dataset:** MNIST Dataset

The MNIST database is a large database of handwritten digits that is commonly used for training various image processing systems which contains 60,000 training images and 10,000 testing images. Half of the training set and half of the test set were taken from NIST's training dataset, while the other half of the training set and the other half of the test set were taken from NIST's testing dataset.

**Input:**



**K-MEANS ALGORITHM**

import sys

import sklearn

import matplotlib

import numpy as np

from keras.datasets import mnist

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

print('Training Data: {}'.format(x\_train.shape))

print('Training Labels: {}'.format(y\_train.shape))

print('Testing Data: {}'.format(x\_test.shape))

print('Testing Labels: {}'.format(y\_test.shape))

import matplotlib.pyplot as plt

# python magic function

%matplotlib inline

# create figure with 3x3 subplots using matplotlib.pyplot

fig, axs = plt.subplots(3, 3, figsize = (12, 12))

plt.gray()

# loop through subplots and add mnist images

for i, ax in enumerate(axs.flat):

ax.matshow(x\_train[i])

ax.axis('off')

ax.set\_title('Number {}'.format(y\_train[i]))

# display the figure

fig.show()

# convert each image to 1 dimensional array

X = x\_train.reshape(len(x\_train),-1)

Y = y\_train

# normalize the data to 0 - 1

X = X.astype(float) / 255.

print(X.shape)

print(X[0].shape)

from sklearn.cluster import MiniBatchKMeans

n\_digits = len(np.unique(y\_test))

print(n\_digits)

# Initialize KMeans model

kmeans = MiniBatchKMeans(n\_clusters = n\_digits)

# Fit the model to the training data

kmeans.fit(X)

def infer\_cluster\_labels(kmeans, actual\_labels):

inferred\_labels = {}

for i in range(kmeans.n\_clusters):

# find index of points in cluster

labels = []

index = np.where(kmeans.labels\_ == i)

# append actual labels for each point in cluster

labels.append(actual\_labels[index])

# determine most common label

if len(labels[0]) == 1:

counts = np.bincount(labels[0])

else:

counts = np.bincount(np.squeeze(labels))

# assign the cluster to a value in the inferred\_labels dictionary

if np.argmax(counts) in inferred\_labels:

# append the new number to the existing array at this slot

inferred\_labels[np.argmax(counts)].append(i)

else:

# create a new array in this slot

inferred\_labels[np.argmax(counts)] = [i]

#print(labels)

#print('Cluster: {}, label: {}'.format(i, np.argmax(counts)))

return inferred\_labels

def infer\_data\_labels(X\_labels, cluster\_labels):

"""

Determines label for each array, depending on the cluster it has been assigned to.

returns: predicted labels for each array

"""

# empty array of len(X)

predicted\_labels = np.zeros(len(X\_labels)).astype(np.uint8)

for i, cluster in enumerate(X\_labels):

for key, value in cluster\_labels.items():

if cluster in value:

predicted\_labels[i] = key

return predicted\_labels

# test the infer\_cluster\_labels() and infer\_data\_labels() functions

cluster\_labels = infer\_cluster\_labels(kmeans, Y)

X\_clusters = kmeans.predict(X)

predicted\_labels = infer\_data\_labels(X\_clusters, cluster\_labels)

print(predicted\_labels[:20])

print(Y[:20])

from sklearn import metrics

def calculate\_metrics(estimator, data, labels):

# Calculate and print metrics

print('Number of Clusters: {}'.format(estimator.n\_clusters))

print('Inertia: {}'.format(estimator.inertia\_))

print('Homogeneity: {}'.format(metrics.homogeneity\_score(labels, estimator.labels\_)))

clusters = [10, 16, 36, 64, 144, 256]

# test different numbers of clusters

for n\_clusters in clusters:

estimator = MiniBatchKMeans(n\_clusters = n\_clusters)

estimator.fit(X)

# print cluster metrics

calculate\_metrics(estimator, X, Y)

# determine predicted labels

cluster\_labels = infer\_cluster\_labels(estimator, Y)

predicted\_Y = infer\_data\_labels(estimator.labels\_, cluster\_labels)

# calculate and print accuracy

print('Accuracy: {}\n'.format(metrics.accuracy\_score(Y, predicted\_Y)))

# test kmeans algorithm on testing dataset

# convert each image to 1 dimensional array

X\_test = x\_test.reshape(len(x\_test),-1)

# normalize the data to 0 - 1

X\_test = X\_test.astype(float) / 255.

# initialize and fit KMeans algorithm on training data

kmeans = MiniBatchKMeans(n\_clusters = 256)

kmeans.fit(X)

cluster\_labels = infer\_cluster\_labels(kmeans, Y)

# predict labels for testing data

test\_clusters = kmeans.predict(X\_test)

predicted\_labels = infer\_data\_labels(kmeans.predict(X\_test), cluster\_labels)

# calculate and print accuracy

print('Accuracy: {}\n'.format(metrics.accuracy\_score(y\_test, predicted\_labels)))

kmeans = MiniBatchKMeans(n\_clusters = 36)

kmeans.fit(X)

# record centroid values

centroids = kmeans.cluster\_centers\_

# reshape centroids into images

images = centroids.reshape(36, 28, 28)

images \*= 255

images = images.astype(np.uint8)

# determine cluster labels

cluster\_labels = infer\_cluster\_labels(kmeans, Y)

# create figure with subplots using matplotlib.pyplot

fig, axs = plt.subplots(6, 6, figsize = (20, 20))

plt.gray()

# loop through subplots and add centroid images

for i, ax in enumerate(axs.flat):

# determine inferred label using cluster\_labels dictionary

for key, value in cluster\_labels.items():

if i in value:

ax.set\_title('Inferred Label: {}'.format(key))

# add image to subplot

ax.matshow(images[i])

ax.axis('off')

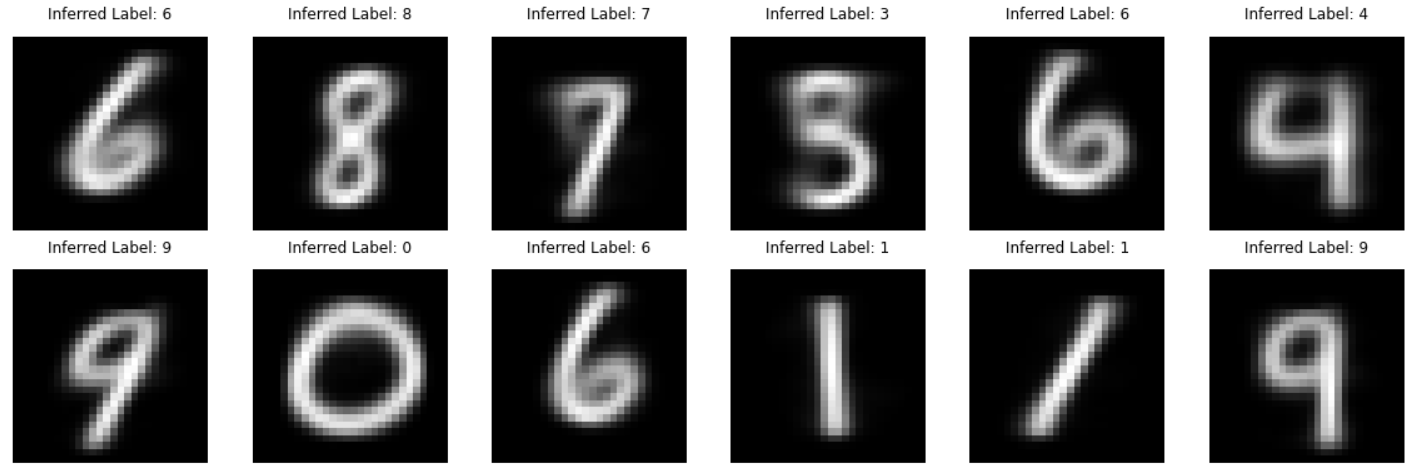
# display the figure

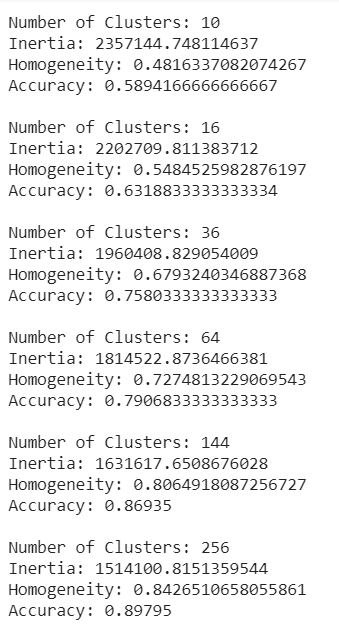
fig.show()

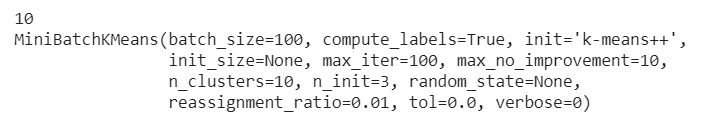
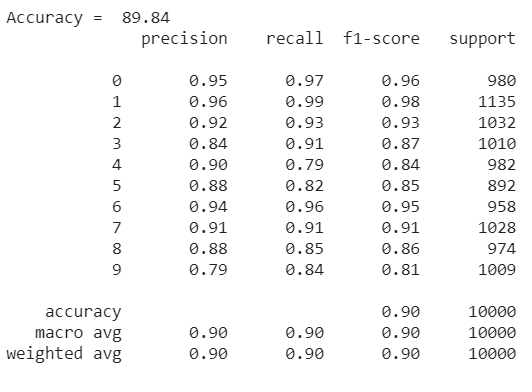
print("Accuracy = ",metrics.accuracy\_score(y\_true=y\_test, y\_pred=predicted\_labels)\*100)

class\_wise = metrics.classification\_report(y\_true=y\_test, y\_pred=predicted\_labels)

print(class\_wise)





**KNN**

import csv

import math

import random

def loadDataset(filename, split, trainingset = [], testset = []):

with open(filename,'r') as csvfile:

lines = csv.reader(csvfile)

dataset = list(lines)

for x in range(1,math.ceil((len(dataset)+1)/10)):

for y in range(1,len(dataset[x])):

dataset[x][y] = float(dataset[x][y])

if random.random() < split:

trainingset.append(dataset[x])

else:

testset.append(dataset[x])

trainingset = []

testset = []

loadDataset('/content/sample\_data/mnist\_train\_small.csv',0.66,trainingset,testset)

print("train "+repr(len(trainingset)))

print("test "+repr(len(testset)))

import math

def euclideandistance(instance1,instance2,length):

distance = 0

for x in range(1,length):

distance += pow((instance1[x]-instance2[x]),2)

return math.sqrt(distance)

import operator

def getNeighbours(trainingset,testinstance,k):

distances = []

length = len(testinstance)-1

for x in range(1,len(trainingset)):

dist = euclideandistance(testinstance, trainingset[x],length)

distances.append((trainingset[x],dist))

distances.sort(key=operator.itemgetter(1))

neighbours = []

for x in range(k):

neighbours.append(distances[x][0])

return neighbours

import operator

def getResponse(neighbours):

classvotes = {}

for x in range(len(neighbours)):

response = neighbours[x][0]

if response in classvotes:

classvotes[response] += 1

else:

classvotes[response] = 1

sortedvotes = sorted(classvotes.items(),key = operator.itemgetter(1),reverse = True)

return sortedvotes[0][0]

def getAccuracy(testset, predictions):

correct = 0

for x in range(len(testset)):

if testset[x][0] == predictions[x]:

correct= correct + 1

#print("Srihari",correct)

return (correct/float(len(testset)))\*100.0

from sklearn import metrics

trainingset = []

testset = []

split = 0.66

loadDataset('/content/sample\_data/mnist\_train\_small.csv',split,trainingset,testset)

print("train "+repr(len(trainingset)))

print("test "+repr(len(testset)))

predictions = []

y\_test = []

k = 1

i=1

for x in range(len(testset)):

neighbours = getNeighbours(trainingset,testset[x],k)

result = getResponse(neighbours)

predictions.append(result)

y\_test.append(testset[x][0])

print(i)

i = i + 1

#print("Predicted = "+repr(result)+' actual = '+repr(testset[x][-1]))

#accuracy = getAccuracy(testset, predictions)

#print("accuracy = ",repr(accuracy),"%")

confusion = metrics.confusion\_matrix(y\_true = y\_test, y\_pred = predictions)

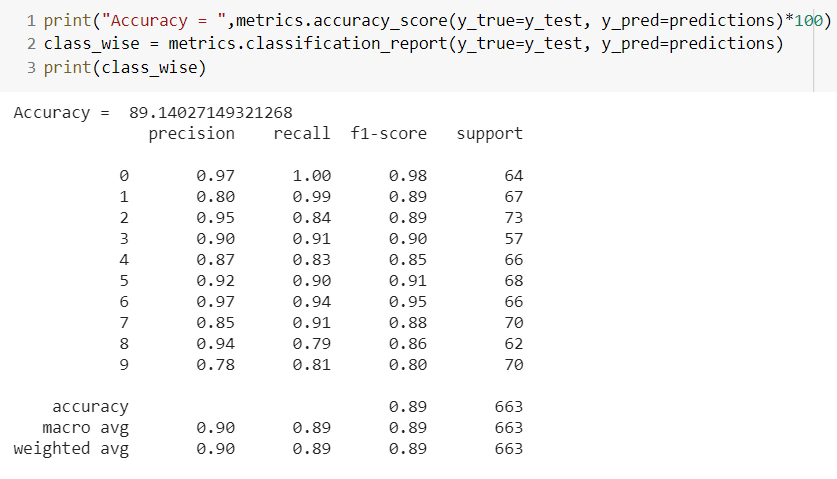
confusion

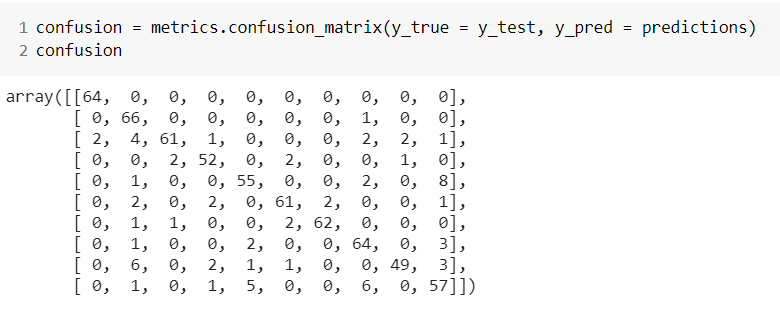
print("Accuracy = ",metrics.accuracy\_score(y\_true=y\_test, y\_pred=predictions)\*100)

class\_wise = metrics.classification\_report(y\_true=y\_test, y\_pred=predictions)

print(class\_wise)







**TABULAR INFERENCE**

|  |  |  |
| --- | --- | --- |
| **ALGORITHM** | **K-MEANS** | **K-NN** |
| **PRECISION** | 0.90 | 0.90 |
| **RECALL** | 0.90 | 0.89 |
| **F1-SCORE** | 0.90 | 0.89 |
| **ACCURACY** | 89.84 % | 90.24 % |

**ALGORITHM 1 – K-MEANS**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **VALUE OF K** | **10** | **16** | **36** | **64** | **144** | **256** |
| **ACCURACY** | 58.9% | 63.1% | 75.8% | 79.06% | 86.9% | 89.84% |

**ALGORITHM 2 – K-NN**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **VALUE OF K** | **1** | **8** | **10** | **12** | **20** |
| **ACCURACY** | 90.24% | 88.19% | 89.6% | 87.15% | 84.2% |

**Inference: Thus, the K-Means Clustering and K-Nearest Neighbours algorithms were implemented. It was applied on the MNIST dataset as well. The accuracies obtained by varying the value of k is tabulated.**

**ENSEMBLE METHODS**

**CLASSIFIER 1: Multi-Layer Perceptron**

import numpy as np

from sklearn.model\_selection import GridSearchCV

from sklearn.neural\_network import MLPClassifier

mlp = MLPClassifier()

params\_mlp = {'max\_iter':[500,1000], 'activation': ['tanh','relu','identity','logistic']}

mlp\_gs = GridSearchCV(mlp, params\_mlp, cv=5)

mlp\_gs.fit(X\_train, y\_train)

#save best model

mlp\_best = mlp\_gs.best\_estimator\_

#check best n\_neigbors value

print(mlp\_gs.best\_params\_)

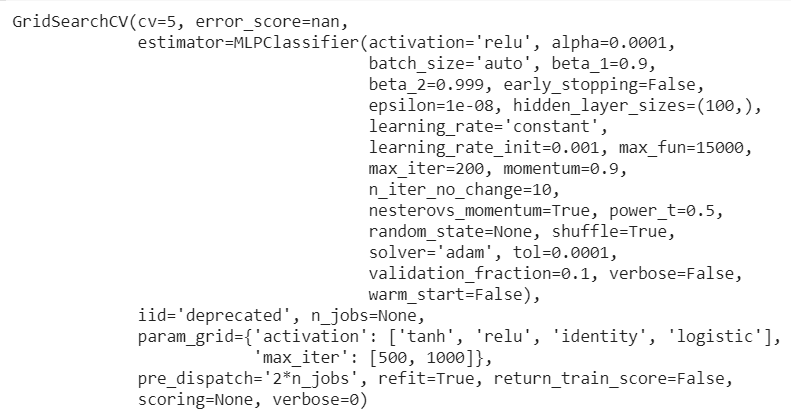
pred\_mlp\_gs = mlp\_gs.predict(X\_test)

print("Accuracy = ",metrics.accuracy\_score(y\_true=y\_test, y\_pred=pred\_mlp\_gs))

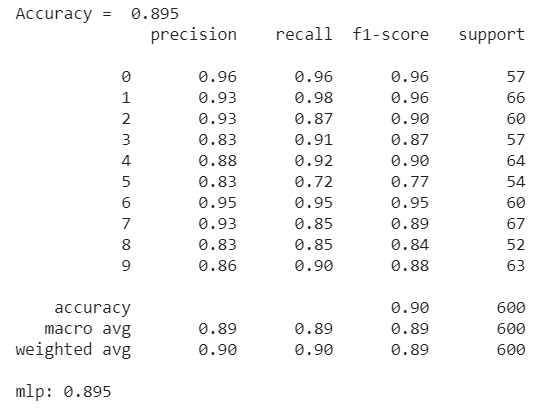
class\_wise = metrics.classification\_report(y\_true=y\_test, y\_pred=pred\_mlp\_gs)

print(class\_wise)

print('mlp: {}'.format(mlp\_gs.score(X\_test, y\_test)))







**CLASSIFIER 2: Support Vector Machine**

import numpy as np

from sklearn import metrics

from sklearn.model\_selection import GridSearchCV

from sklearn.svm import SVC

svm\_rbf = SVC(kernel='rbf')

params\_svm = {'C':[1, 10, 100], 'gamma': ['scale',1e-2, 1e-3, 1e-4]}

svm\_gs = GridSearchCV(svm\_rbf, params\_svm, cv=5)

svm\_gs.fit(X\_train, y\_train)

#save best model

svm\_best = svm\_gs.best\_estimator\_

#check best n\_neigbors value

print(svm\_gs.best\_params\_)

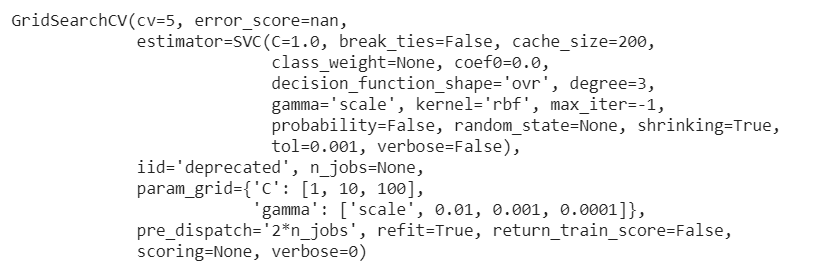
pred\_svm\_gs = svm\_gs.predict(X\_test)

print("Accuracy = ",metrics.accuracy\_score(y\_true=y\_test, y\_pred=pred\_svm\_gs))

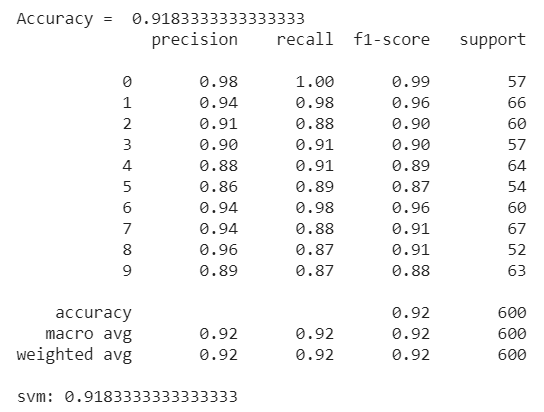
class\_wise = metrics.classification\_report(y\_true=y\_test, y\_pred=pred\_svm\_gs)

print(class\_wise)

print('svm: {}'.format(svm\_gs.score(X\_test, y\_test)))







**CLASSIFIER 3: Decision Tree**

import numpy as np

from sklearn import metrics

from sklearn.model\_selection import GridSearchCV

from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier()

params\_dt = {'max\_depth': np.arange(1, 10),'criterion':['gini','entropy'],'splitter':['best','random']}

dt\_gs = GridSearchCV(dt, params\_dt, cv=10)

#fit model to training data

dt\_gs.fit(X\_train, y\_train)

dt\_best = dt\_gs.best\_estimator\_

print(dt\_gs.best\_params\_)

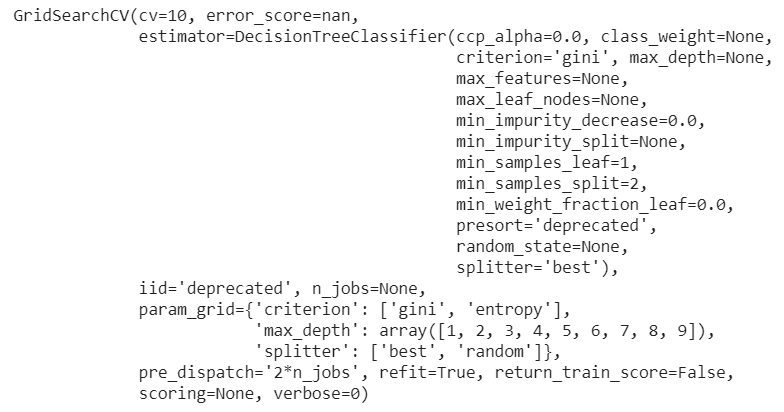
pred\_dt\_gs = dt\_gs.predict(X\_test)

print("Accuracy = ",metrics.accuracy\_score(y\_true=y\_test, y\_pred=pred\_dt\_gs))

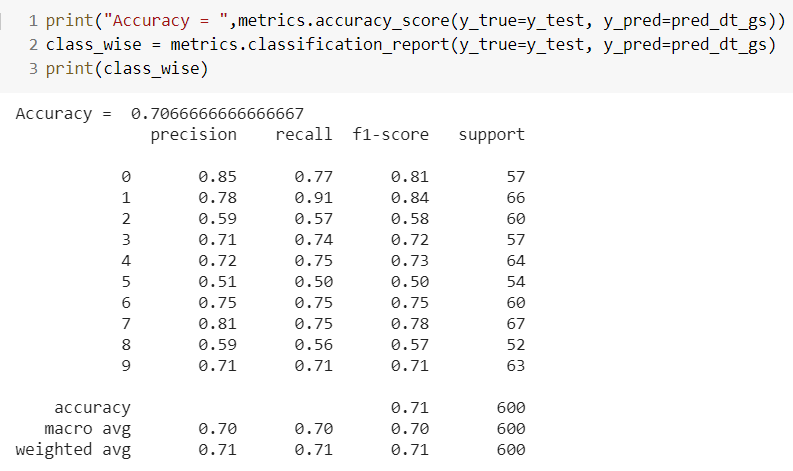
class\_wise = metrics.classification\_report(y\_true=y\_test, y\_pred=pred\_dt\_gs)

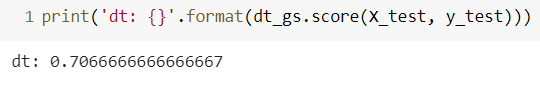
print(class\_wise)

print('dt: {}'.format(dt\_gs.score(X\_test, y\_test)))









**CLASSIFIER 4: K-Nearest Neighbours**

import numpy as np

from sklearn import metrics

from sklearn.model\_selection import GridSearchCV

import numpy as np

from sklearn.model\_selection import GridSearchCV

from sklearn.neighbors import KNeighborsClassifier

#create new a knn model

knn = KNeighborsClassifier()

#create a dictionary of all values we want to test for n\_neighbors

params\_knn = {'n\_neighbors': np.arange(1, 25)}

#use gridsearch to test all values for n\_neighbors

knn\_gs = GridSearchCV(knn, params\_knn, cv=5)

#fit model to training data

knn\_gs.fit(X\_train, y\_train)

#save best model

knn\_best = knn\_gs.best\_estimator\_

#check best n\_neigbors value

print(knn\_gs.best\_params\_)

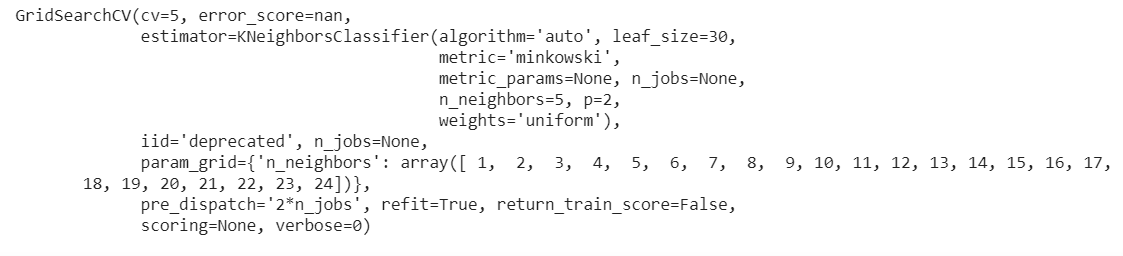
pred\_knn\_gs = knn\_gs.predict(X\_test)

print("Accuracy = ",metrics.accuracy\_score(y\_true=y\_test, y\_pred=pred\_knn\_gs))

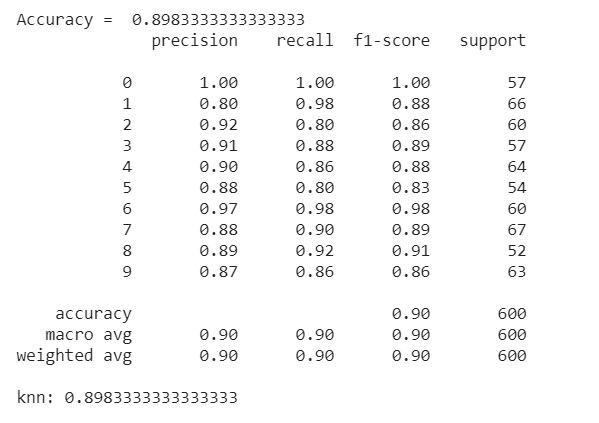
class\_wise = metrics.classification\_report(y\_true=y\_test, y\_pred=pred\_knn\_gs)

print(class\_wise)

print('knn: {}'.format(knn\_gs.score(X\_test, y\_test)))







**VOTING CLASSIFIER**



**Hard Voting:**

from sklearn.ensemble import VotingClassifier

#create a dictionary of our models

estimators=[('knn', knn\_best), ('mlp', mlp\_best), ('svm', svm\_best), ('dt', dt\_best)]

#create our voting classifier, inputting our models

ensemble = VotingClassifier(estimators, voting='hard')

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

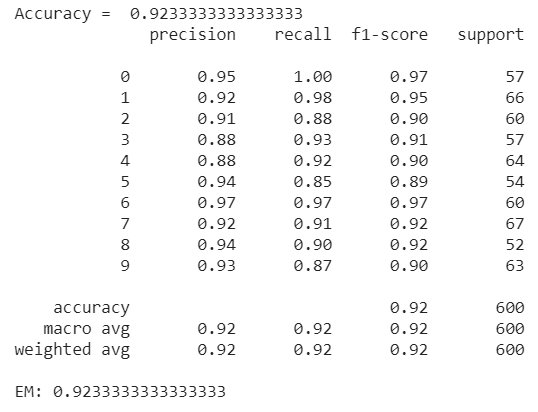
pred\_em = ensemble.predict(X\_test)

print("Accuracy = ",metrics.accuracy\_score(y\_true=y\_test, y\_pred=pred\_em))

class\_wise = metrics.classification\_report(y\_true=y\_test, y\_pred=pred\_em)

print(class\_wise)

print('EM: {}'.format(ensemble.score(X\_test, y\_test)))



**Soft Voting:**

from sklearn.ensemble import VotingClassifier

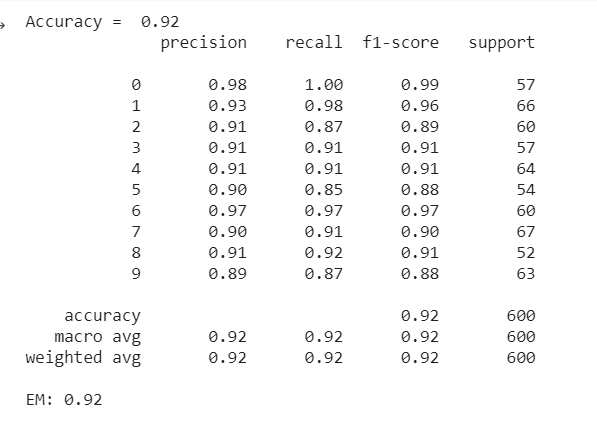
estimators=[('knn', knn\_best), ('mlp', mlp\_best), ('svm', svm\_best), ('dt', dt\_best)]

ensemble = VotingClassifier(estimators, voting='soft')

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

#test our model on the test data

ensemble.score(X\_test, y\_test)



**TABULAR INFERENCE**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CLASSIFIER** | **PRECISION** | **RECALL** | **F1-SCORE** | **ACCURACY** |
| **MLP** | 0.89 | 0.89 | 0.89 | 89.5% |
| **SVM** | 0.92 | 0.92 | 0.92 | 91.8% |
| **Decision Tree** | 0.70 | 0.70 | 0.70 | 70.6% |
| **KNN** | 0.90 | 0.90 | 0.90 | 89.8% |
| **Hard Voting Classifier** | 0.92 | 0.92 | 0.92 | **92.3%** |
| **Soft Voting Classifier** | 0.92 | 0.92 | 0.92 | **92%** |

**Inference: Thus, using four different classifiers – MLP, SVM, DT and KNN ensemble methods were implemented. It was applied on the MNIST dataset as well. The accuracies obtained are found to be higher than the individual accuracies of the classifiers using both hard voting and soft voting.**