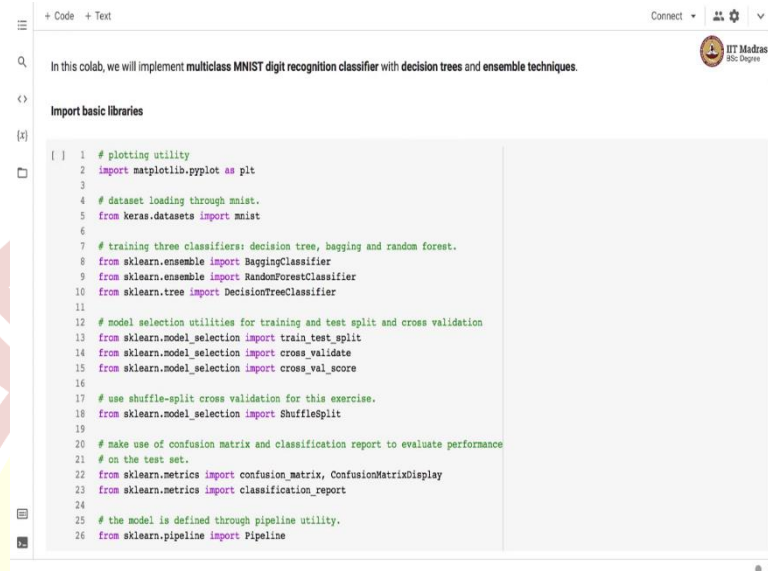


IIT Madras

ONLINE DEGREE

Machine Learning Practice
Professor. Doctor Ashish Tendulkar
Indian Institute of Technology, Madras
Bagging and RandomForestClassifier on MNIST

(Refer Slide Time: 00:10)



```
[ ] 1 # plotting utility
2 import matplotlib.pyplot as plt
3
4 # dataset loading through mnist.
5 from keras.datasets import mnist
6
7 # training three classifiers: decision tree, bagging and random forest.
8 from sklearn.ensemble import BaggingClassifier
9 from sklearn.ensemble import RandomForestClassifier
10 from sklearn.tree import DecisionTreeClassifier
11
12 # model selection utilities for training and test split and cross validation
13 from sklearn.model_selection import train_test_split
14 from sklearn.model_selection import cross_validate
15 from sklearn.model_selection import cross_val_score
16
17 # use shuffle-split cross validation for this exercise.
18 from sklearn.model_selection import ShuffleSplit
19
20 # make use of confusion matrix and classification report to evaluate performance
21 # on the test set.
22 from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
23 from sklearn.metrics import classification_report
24
25 # the model is defined through pipeline utility.
26 from sklearn.pipeline import Pipeline
```

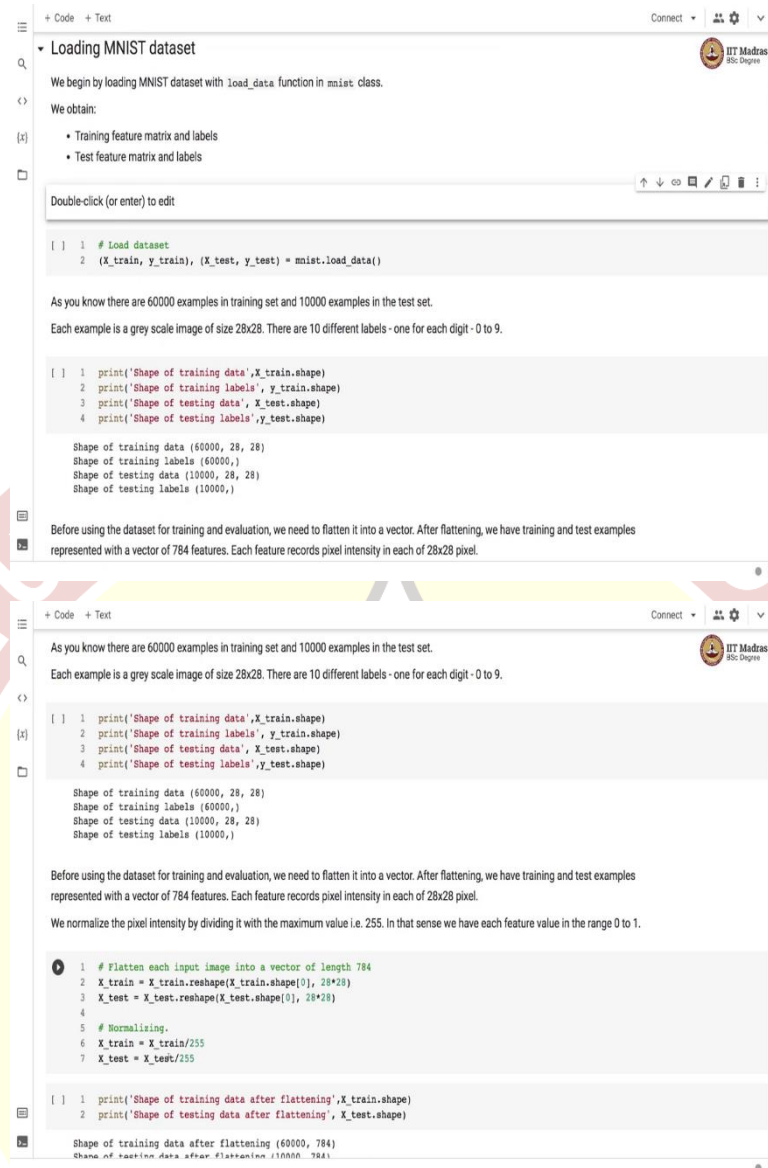
Namaste! Welcome to the next video of Machine Learning Practice Course. In this video, we will implement multiclass MNIST digit recognition classifier with decision trees and ensemble techniques. We will begin by importing basic libraries like matplotlib.pyplot for plotting, then the data will be loaded through MNIST library from keras.datasets module.

Then there are a bunch of classifiers, 3 classifiers to be specific, the BaggingClassifier, RandomForestClassifier and DecisionTreeClassifier that are imported from the modules. BaggingClassifier and RandomForestClassifiers are implemented as part of sklearn.ensemble module, whereas DecisionTreeClassifier is implemented as part of sklearn.tree module.

Then we import model selection utilities for training and test split which is train_test_split. And for cross-validation, there are a couple of cross-validation utilities that we are importing. We use shuffle split cross-validation for this exercise, and we also import ShuffleSplit from sklearn.model_selection module.

We make use of confusion_matrix and classification_report to evaluate performance on the test set. The confusion_matrix is displayed with ConfusionMatrixDisplay API from sklearn.metrics module. And finally, the model is defined through pipeline utility as we have been doing in all other collapse in this course.

(Refer Slide Time: 01:48)



```
+ Code + Text
Loading MNIST dataset

We begin by loading MNIST dataset with load_data function in mnist class.

We obtain:
• Training feature matrix and labels
• Test feature matrix and labels

Double-click (or enter) to edit

1 # Load dataset
2 (X_train, y_train), (X_test, y_test) = mnist.load_data()

As you know there are 60000 examples in training set and 10000 examples in the test set.
Each example is a grey scale image of size 28x28. There are 10 different labels - one for each digit - 0 to 9.

1 print('Shape of training data', X_train.shape)
2 print('Shape of training labels', y_train.shape)
3 print('Shape of testing data', X_test.shape)
4 print('Shape of testing labels', y_test.shape)

Shape of training data (60000, 28, 28)
Shape of training labels (60000,)
Shape of testing data (10000, 28, 28)
Shape of testing labels (10000,)

Before using the dataset for training and evaluation, we need to flatten it into a vector. After flattening, we have training and test examples represented with a vector of 784 features. Each feature records pixel intensity in each of 28x28 pixel.

+ Code + Text
As you know there are 60000 examples in training set and 10000 examples in the test set.
Each example is a grey scale image of size 28x28. There are 10 different labels - one for each digit - 0 to 9.

1 print('Shape of training data', X_train.shape)
2 print('Shape of training labels', y_train.shape)
3 print('Shape of testing data', X_test.shape)
4 print('Shape of testing labels', y_test.shape)

Shape of training data (60000, 28, 28)
Shape of training labels (60000,)
Shape of testing data (10000, 28, 28)
Shape of testing labels (10000,)

Before using the dataset for training and evaluation, we need to flatten it into a vector. After flattening, we have training and test examples represented with a vector of 784 features. Each feature records pixel intensity in each of 28x28 pixel.

We normalize the pixel intensity by dividing it with the maximum value i.e. 255. In that sense we have each feature value in the range 0 to 1.

1 # Flatten each input image into a vector of length 784
2 X_train = X_train.reshape(X_train.shape[0], 28*28)
3 X_test = X_test.reshape(X_test.shape[0], 28*28)
4
5 # Normalising.
6 X_train = X_train/255
7 X_test = X_test/255

1 print('Shape of training data after flattening', X_train.shape)
2 print('Shape of testing data after flattening', X_test.shape)

Shape of training data after flattening (60000, 784)
Shape of testing data after flattening (10000, 784)
```

We begin by loading MNIST dataset with load data function in MNIST class. We obtained training feature matrix and labels as well as test feature matrix and labels. As you know, there are 60,000 examples in training set and 10,000 examples in the test set. Every example is a gray scale image of size 28×28 . And there are 10 different labels 1 for each digit from 0 to 9.

Before using the dataset for training and evaluation, we need to flatten it into a vector. After flattening we have training and test examples represented with 784 features. Each feature records pixel intensity in each of 28×28 pixel image. We normalize the pixel intensity by dividing it with maximum value that is 255. In that sense, we have each feature value now in the range between 0 to 1.

(Refer Slide Time: 02:52)

```
+ Code + Text
2 X_train = X_train.reshape(X_train.shape[0], 28*28)
3 X_test = X_test.reshape(X_test.shape[0], 28*28)
4
5 # Normalizing.
6 X_train = X_train/255
7 X_test = X_test/255

1 print('Shape of training data after flattening', X_train.shape)
2 print('Shape of testing data after flattening', X_test.shape)

Shape of training data after flattening (60000, 784)
Shape of testing data after flattening (10000, 784)

We use ShuffleSplit cross validation with 10 splits and 20% data set aside for model evaluation as a test data.

1 cv = ShuffleSplit(n_splits=10, test_size=0.2, random_state=42)

We define two functions:

1. train_classifiers contains a common code for training classifiers for MNIST multiclass classification problem.
  • It takes estimator, feature matrix, labels, cross validation strategy and name of the classifier as input.
  • It first fits the estimator with feature matrix and labels.
  • It obtains cross validated f1_macro score for training set with 10-fold ShuffleSplit cross validation and prints it.

1 def train_classifiers(estimator, X_train, y_train, cv, name):
2     estimator.fit(X_train, y_train)
3     cv_train_score = cross_val_score(estimator, X_train, y_train,
4                                     cv=cv, scoring='f1_macro')
5     print(f'On an average, {name} model has f1 score of ')
6     f'(cv_train_score.mean():.3f) +/- (cv_train_score.std():.3f) on the training set.)

2. The eval function takes estimator, test feature matrix and labels as input and produce classification report and confusion matrix.
  • It first predicts labels for the test set.
  • Then it uses these predicted reports for calculating various evaluation metrics like precision, recall, f1 score and accuracy for each of the 10 classes.
  • It also obtains a confusion matrix by comparing these predictions with the actual labels and displays it with ConfusionMatrixDisplay utility.

1 def eval(estimator, X_test, y_test):
2     y_pred = estimator.predict(X_test)
```

So, you can see that the shape of training data after flattening is 60,000 by 784, whereas the shape of testing data after flattening is 10,000 by 784. Here, we use ShuffleSplit validation with 10 folds and 20% dataset aside for model evaluation as a test data.

(Refer Slide Time: 03:18)

```
+ Code + Text
1 cv = ShuffleSplit(n_splits=10, test_size=0.2, random_state=42)

We define two functions:

1. train_classifiers contains a common code for training classifiers for MNIST multiclass classification problem.
  • It takes estimator, feature matrix, labels, cross validation strategy and name of the classifier as input.
  • It first fits the estimator with feature matrix and labels.
  • It obtains cross validated f1_macro score for training set with 10-fold ShuffleSplit cross validation and prints it.

1 def train_classifiers(estimator, X_train, y_train, cv, name):
2     estimator.fit(X_train, y_train)
3     cv_train_score = cross_val_score(estimator, X_train, y_train,
4                                     cv=cv, scoring='f1_macro')
5     print(f'On an average, {name} model has f1 score of ')
6     f'(cv_train_score.mean():.3f) +/- (cv_train_score.std():.3f) on the training set.)

2. The eval function takes estimator, test feature matrix and labels as input and produce classification report and confusion matrix.
  • It first predicts labels for the test set.
  • Then it uses these predicted reports for calculating various evaluation metrics like precision, recall, f1 score and accuracy for each of the 10 classes.
  • It also obtains a confusion matrix by comparing these predictions with the actual labels and displays it with ConfusionMatrixDisplay utility.

1 def eval(estimator, X_test, y_test):
2     y_pred = estimator.predict(X_test)
```

We define 2 functions 1 for training the classifiers and 2 for evaluation. The train classifier function contains common code for training classifiers for MNIST multiclass classification problem. It takes estimator, feature matrix, labels and cross-validation strategy along with the name of the classifier as input. It first fits the estimator with feature matrix and labels as input, and then it obtains cross validated f1 _macro score for training set with 10 for ShuffleSplit cross-validation, and finally it prints the value of f1 _macro.

(Refer Slide Time: 03:58)

```

+ Code + Text
Connect
2. The eval function takes estimator, test feature matrix and labels as input and produce classification report and confusion matrix.
    • It first predicts labels for the test set.
    • Then it uses these predicted reports for calculating various evaluation metrics like precision, recall, f1 score and accuracy for each of the 10 classes.
    • It also obtains a confusion matrix by comparing these predictions with the actual labels and displays it with ConfusionMatrixDisplay utility.

1 def eval(estimator, X_test, y_test):
2     y_pred = estimator.predict(X_test)
3
4     print("# Classification report")
5     print(classification_report(y_test, y_pred))
6
7     print("# Confusion matrix")
8     disp = ConfusionMatrixDisplay(
9         confusion_matrix=confusion_matrix(y_test, y_pred))
10    disp.plot()
11    plt.title('Confusion matrix')
12    plt.show()

Let's train three classifiers with default parameters.

• Decision tree
• Bagging classifier - which uses decision tree as a default classifier and trains multiple decision tree classifiers on different bags obtained through bootstrap sampling of training set.
• Random forest classifier - which is also a bagging technique, which trains different decision tree classifiers by randomly selecting attributes for splitting on bags of bootstrap sample of training set.

```

The eval function on the other hand, takes estimator test feature matrix and labels as input and produce classification _report and confusion _matrix as output. It first predicts the labels for the test set then it uses these predicted labels for calculating classification _report, which outputs various evaluation metrics like precision, recall, f1 _score accuracy for each of the 10 classes. It also obtains confusion _matrix by comparing these predictions with the actual label and displays it with ConfusionMatrixDisplay utility.

(Refer Slide Time: 04:42)

```

+ Code + Text
Connect
Let's train three classifiers with default parameters.

• Decision tree
• Bagging classifier - which uses decision tree as a default classifier and trains multiple decision tree classifiers on different bags obtained through bootstrap sampling of training set.
• Random forest classifier - which is also a bagging technique, which trains different decision tree classifiers by randomly selecting attributes for splitting on bags of bootstrap sample of training set.

• Decision trees for MNIST multiclass classification

We instantiate a decision tree classifier with default parameters and train it. The train_classifier function prints mean of cross validated accuracy and standard deviation of the trained classifier on the training set.

1 decision_tree_pipeline = Pipeline([("classifier", DecisionTreeClassifier())])
2 train_classifiers(decision_tree_pipeline, X_train, y_train.ravel(), cv,
3                  "decision tree")

On an average, decision tree model has f1 score of 0.867 +/- 0.005 on the training set.

Let's evaluate the trained classifier on the test set.

1 eval(decision_tree_pipeline, X_test, y_test)

# Classification report
precision    recall  f1-score   support

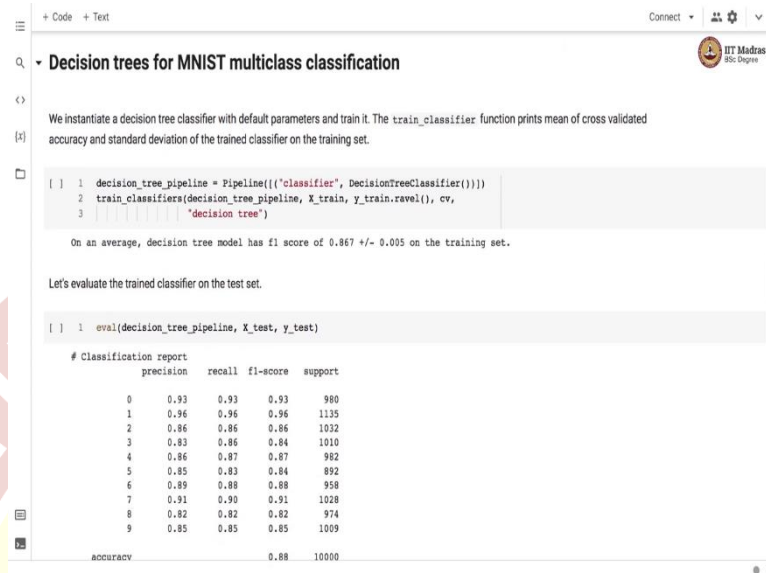
0         0.93    0.93    0.93     980

```

Let us train these 3 classifiers with default parameters. The first classifier is a decision tree, 2 is BaggingClassifier, which also uses decision tree as a default classifier and it trains in fact multiple DecisionTreeClassifiers on different bags obtained through bootstrap sampling or training set. And we also train a RandomForestClassifier, which is also a bagging technique,

and it trains different DecisionTreeClassifiers by randomly selecting attributes for splitting on bags or bootstrap sample or training set.

(Refer Slide Time: 05:22)



```
+ Code + Text
Connect
Decision trees for MNIST multiclass classification

We instantiate a decision tree classifier with default parameters and train it. The train_classifier function prints mean of cross validated accuracy and standard deviation of the trained classifier on the training set.

[ ] 1 decision_tree_pipeline = Pipeline([('classifier', DecisionTreeClassifier())])
    2 train_classifiers(decision_tree_pipeline, X_train, y_train.ravel()), cv,
    3 | | | | | "decision tree"

On an average, decision tree model has f1 score of 0.867 +/- 0.005 on the training set.

Let's evaluate the trained classifier on the test set.

[ ] 1 eval(decision_tree_pipeline, X_test, y_test)

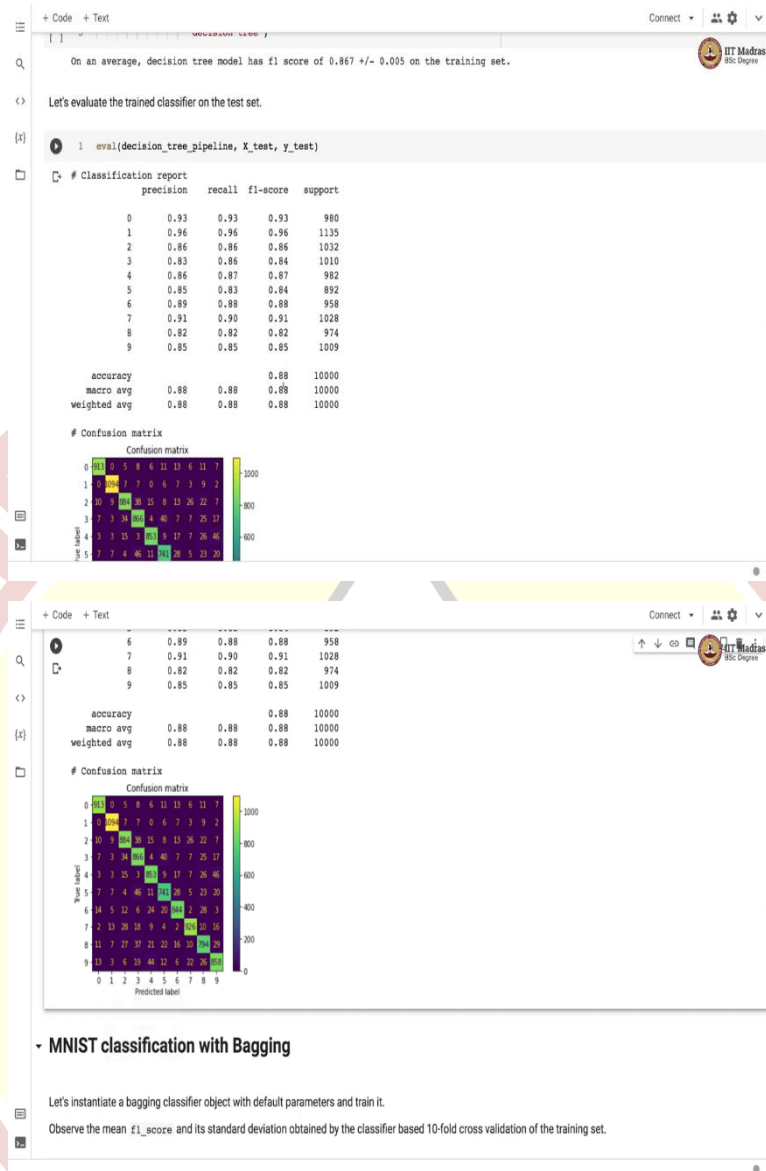
# Classification report
precision    recall  f1-score   support

0         0.93      0.93      0.93        980
1         0.96      0.96      0.96       1135
2         0.86      0.86      0.86       1032
3         0.83      0.85      0.84       1010
4         0.86      0.87      0.87        982
5         0.85      0.83      0.84        892
6         0.89      0.88      0.88        958
7         0.91      0.90      0.91       1028
8         0.82      0.82      0.82        974
9         0.85      0.85      0.85       1009

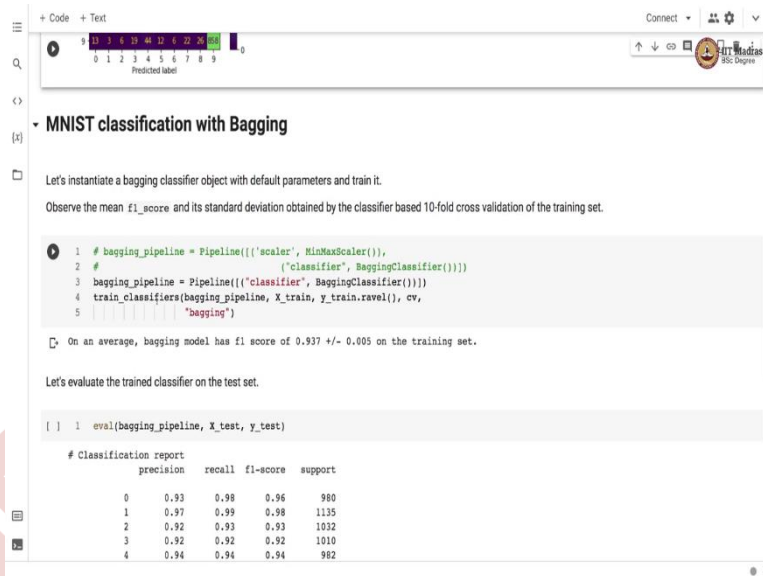
accuracy          0.88      10000
```

Let us train the MNIST multiclass classifier with decision trees. We instantiate our DecisionTreeClassifier in the pipeline with default parameters and train it with train_classifier function. The train_classifier function brings mean of cross validated accuracy and standard deviation of the trained classifier on the training set. So, here we print f1_macro score and f1_macro score from decision tree is 0.86 and the standard deviation is 0.005, which is a very small value. Let us evaluate the DecisionTreeClassifier on the test set.

And you can see that on the test set, it has got accuracy of 0.88. And this is a confusion matrix for your reference where we have true label on y-axis and predicted labels on the x-axis. There are 10 labels, both on x and y-axis. And on the diagonal, you see the number of images that are correctly classified to their corresponding digits.



(Refer Slide Time: 06:32)



```
1 # bagging_pipeline = Pipeline([('scaler', MinMaxScaler()),
2                               ('classifier', BaggingClassifier())])
3 bagging_pipeline = Pipeline([('classifier', BaggingClassifier())])
4 train_classifiers(bagging_pipeline, X_train, y_train.ravel(), cv,
5                  'bagging')
```

On an average, bagging model has f1 score of 0.937 +/- 0.005 on the training set.

Let's evaluate the trained classifier on the test set.

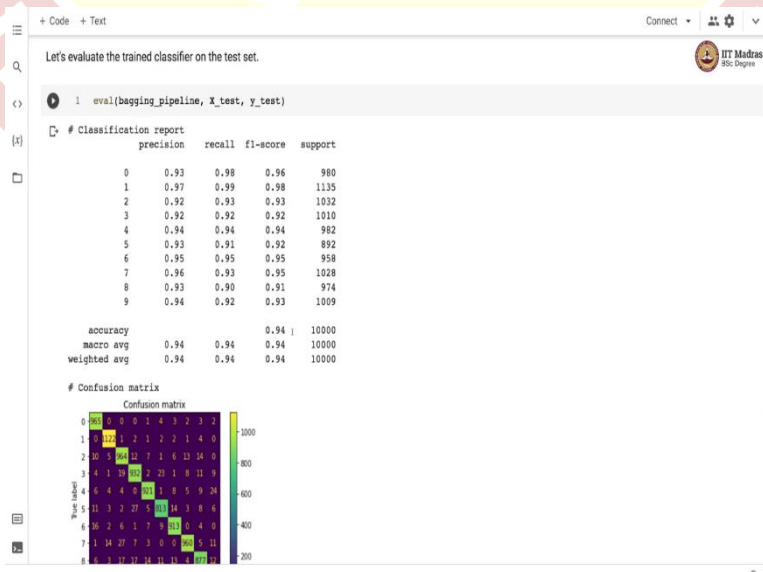
```
[ ] 1 eval(bagging_pipeline, X_test, y_test)
```

| # Classification report | | | | |
|-------------------------|-----------|--------|----------|---------|
| | precision | recall | f1-score | support |
| 0 | 0.93 | 0.98 | 0.96 | 980 |
| 1 | 0.97 | 0.99 | 0.98 | 1135 |
| 2 | 0.92 | 0.93 | 0.93 | 1032 |
| 3 | 0.92 | 0.92 | 0.92 | 1010 |
| 4 | 0.94 | 0.94 | 0.94 | 962 |

Next, we train the MNIST classifier with bagging technique. So here, we instantiate a BaggingClassifier with default parameters, and we train it with train _classifiers function. Then we observe f1 _score and standard deviation as obtained by the classifier on the training set based on 10-fold cross-validation.

So here, we obtained f1 _score of 0.937 with a small standard deviation on the training set. So, you can see that we are able to get better f1 _score. So, earlier reference code with decision trees was 0.86. And now we have gotten the f1 _score of 0.993, which is about 7% point increase in the f1 _score. And we got this increase just by using bagging which trains multiple DecisionTreeClassifiers.

(Refer Slide Time: 07:25)



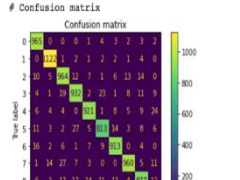
```
1 eval(bagging_pipeline, X_test, y_test)
```

| # Classification report | | | | |
|-------------------------|-----------|--------|----------|---------|
| | precision | recall | f1-score | support |
| 0 | 0.93 | 0.98 | 0.96 | 980 |
| 1 | 0.97 | 0.99 | 0.98 | 1135 |
| 2 | 0.92 | 0.93 | 0.93 | 1032 |
| 3 | 0.92 | 0.92 | 0.92 | 1010 |
| 4 | 0.94 | 0.94 | 0.94 | 962 |
| 5 | 0.93 | 0.91 | 0.92 | 892 |
| 6 | 0.95 | 0.95 | 0.95 | 958 |
| 7 | 0.96 | 0.93 | 0.95 | 1028 |
| 8 | 0.93 | 0.90 | 0.91 | 974 |
| 9 | 0.94 | 0.92 | 0.93 | 1009 |

| | accuracy | macro avg | weighted avg | |
|--|----------|-----------|--------------|-------|
| | 0.94 | 0.94 | 0.94 | 10000 |
| | 0.94 | 0.94 | 0.94 | 10000 |
| | 0.94 | 0.94 | 0.94 | 10000 |

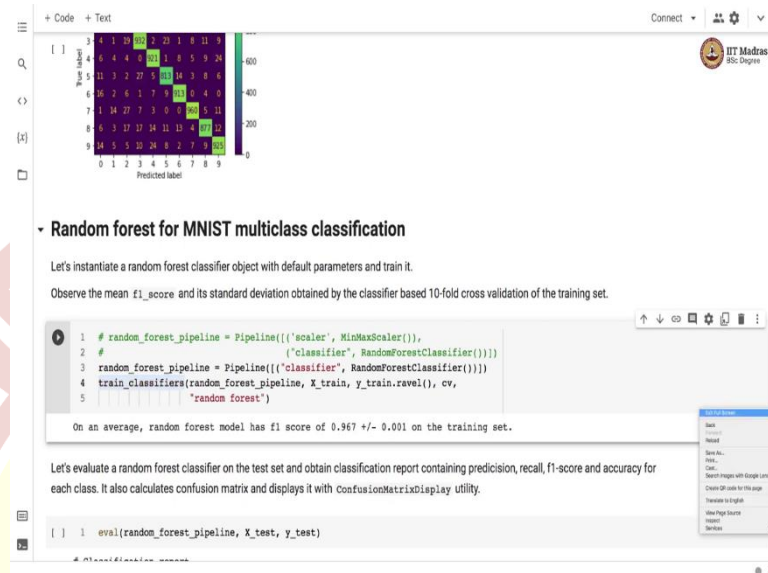
Confusion matrix

Confusion matrix



Let us evaluate the trained classifier on the test set. So, on the bagging we obtain accuracy of 0.94 which is again 6% point jump over what we obtained with decision tree, in case of decision tree this was 0.88.

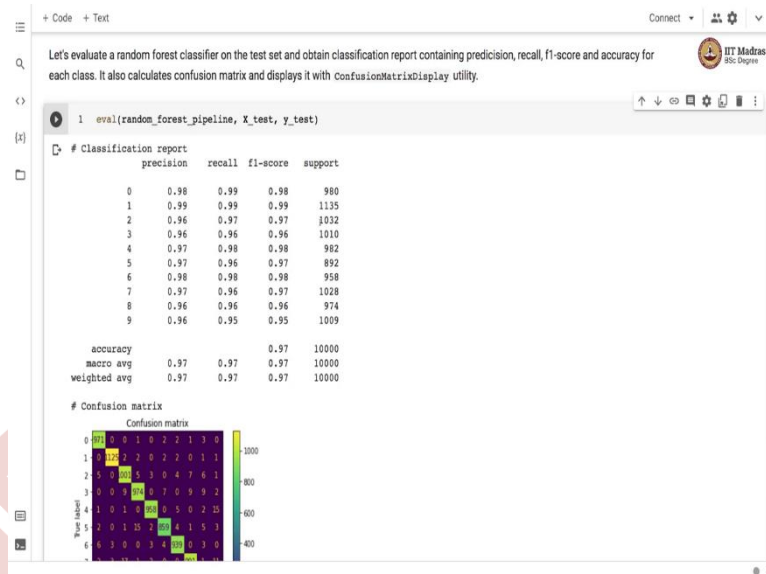
(Refer Slide Time: 07:43)



And finally, we train a `RandomForestClassifier` for the multiclass classification for recognizing handwritten digits. So here, we instantiate `RandomForestClassifier` object with default parameters and train it with `train_classifiers` function, we observe the `f1_score` and its standard deviation as obtained by the classifier on the training set based on 10-fold cross-validation.

So, we see that random forest model achieves `f1_score` of 0.967 close to 0.997. So random forest, with random forest we are able to get even higher accuracy than the bagging so there is almost 3- 4 % point improvement by using random forest over `BaggingClassifier`.

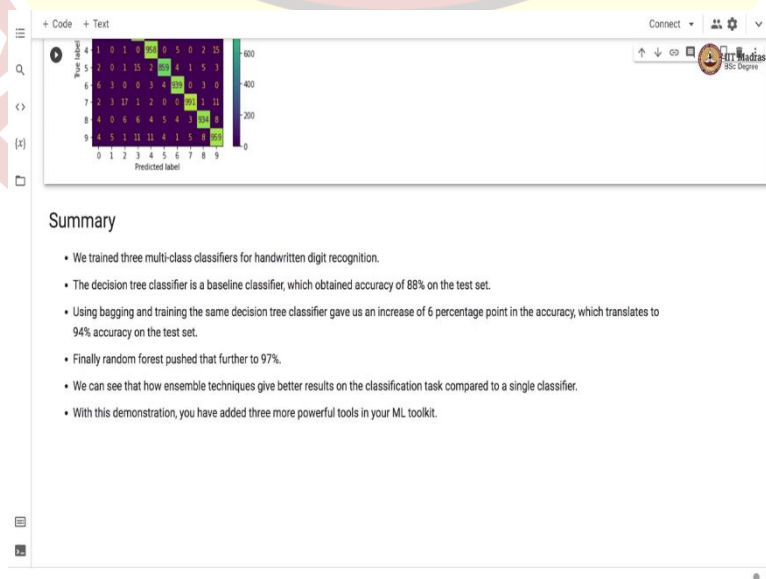
(Refer Slide Time: 08:31)



So let us evaluate a RandomForestClassifier on the test set and obtain classification _report which contains precision recall f1 _score and accuracy for each class. It also calculates confusion _matrix and displays it with ConfusionMatrixDisplay utility. So, you can see that we are often accuracy of 0.97 with RandomForestClassifier and most of the digits are recognized fairly accurately, except for digit 9 which has got a full score of 0.95.

So, this particular accuracy is at par with K-nearest neighbour classifier. K-nearest neighbour classifier also achieves accuracy in this particular range. If you recall, when we train, MNIST digit recognition with K-nearest neighbour classifier.

(Refer Slide Time: 09:21)



So, we trained 3 multiclass classifiers for handwritten digit recognition. The DecisionTreeClassifier is a baseline classifier which obtained accuracy of 88% on the test set. Using bagging and training the same DecisionTreeClassifier gave us an increase of 6% point in accuracy, which translates to 94% accuracy on the test set. Finally, there are no forests pushed the accuracy further to 97%.

So, we can see that how ensemble techniques give better result on classification tasks compared to a single classifier. With this demonstration, you have added 3 more powerful tools in your ML tool kit.

