INDIAN INSTITUTE OF TECHNOLOGY MADRAS

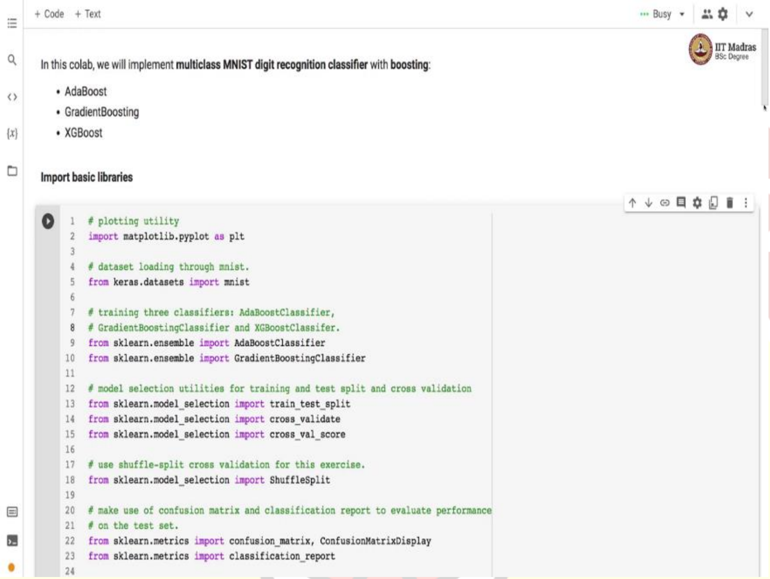
सिद्धिर्भवति कर्मजा

IIT Madras

ONLINE DEGREE

Machine Learning Practice
Professor. Ashish Tendulkar
Indian Institute of Technology, Madras
AdaBoost and GradientBoost Classifier on MNIST

(Refer Slide Time: 00:10)

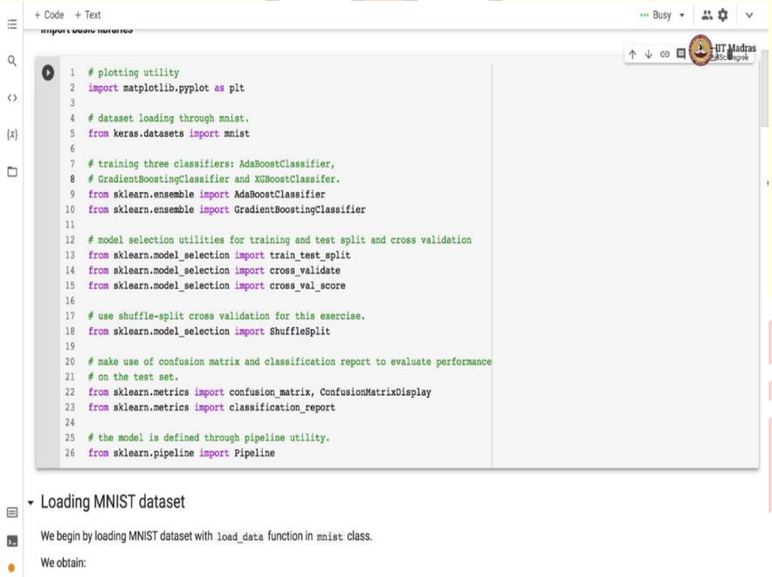


In this colab, we will implement **multiclass MNIST digit recognition classifier** with **boosting**:

- AdaBoost
- GradientBoosting
- XGBoost

Import basic libraries

```
# plotting utility
import matplotlib.pyplot as plt

# dataset loading through mnist.
from keras.datasets import mnist

# training three classifiers: AdaBoostClassifier,
# GradientBoostingClassifier and XGBoostClassifer.
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier

# model selection utilities for training and test split and cross validation
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_validate
from sklearn.model_selection import cross_val_score

# use shuffle-split cross validation for this exercise.
from sklearn.model_selection import ShuffleSplit

# make use of confusion matrix and classification report to evaluate performance
# on the test set.
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics import classification_report

```



```
# plotting utility
import matplotlib.pyplot as plt

# dataset loading through mnist.
from keras.datasets import mnist

# training three classifiers: AdaBoostClassifier,
# GradientBoostingClassifier and XGBoostClassifer.
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier

# model selection utilities for training and test split and cross validation
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_validate
from sklearn.model_selection import cross_val_score

# use shuffle-split cross validation for this exercise.
from sklearn.model_selection import ShuffleSplit

# make use of confusion matrix and classification report to evaluate performance
# on the test set.
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics import classification_report

# the model is defined through pipeline utility.
from sklearn.pipeline import Pipeline
```

▾ Loading MNIST dataset

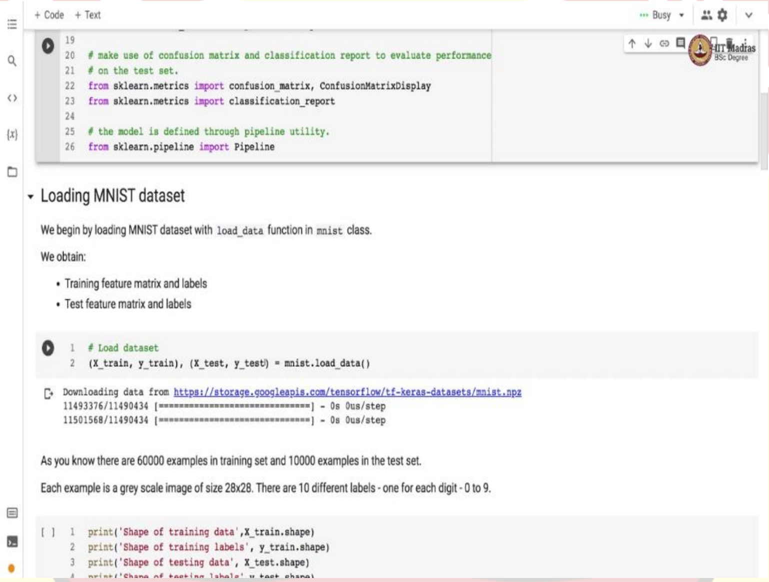
We begin by loading MNIST dataset with `load_data` function in `mnist` class.

We obtain:

Namaste! Welcome to the next video of Machine Learning Practice Course. In this video, we will implement multiclass MNIST digit recognition classifier with boosting, we will be using 3 boosting classifiers, AdaBoostClassifier, GradientBoostingClassifier and XGBoostclassifier. We begin by importing our usual Python libraries. We will be making use of matplotlib.pyplot for plotting.

We will be loading dataset through MNIST, we will be training 3 classifiers, AdaBoostClassifier and GradientBoostingClassifier that are implemented as part of sklearn.ensemble module. Then we have a bunch of model selection utilities like train _test _split, and cross-validation utilities. We will be using ShuffleSplit cross-validation for this exercise. We will make use of confusion _matrix and classification _report to evaluate the performance on the test set. And the model is defined through the pipeline utility.

(Refer Slide Time: 01:14)



```python

# make use of confusion matrix and classification report to evaluate performance
# on the test set.
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics import classification_report

# the model is defined through pipeline utility.
from sklearn.pipeline import Pipeline
```

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

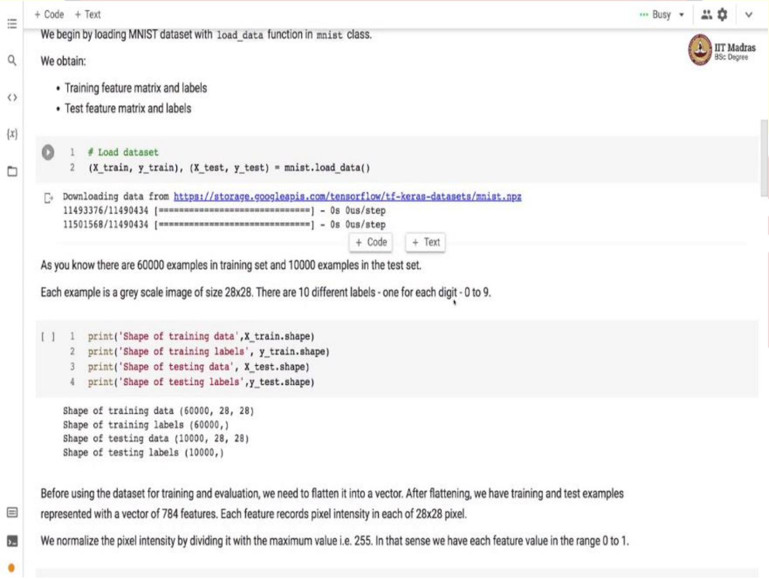
```python
# Load dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
```

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.

```python
print('Shape of training data',X_train.shape)
print('Shape of training labels', y_train.shape)
print('Shape of testing data', X_test.shape)
```



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

```python
# Load dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
```

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.

```python
print('Shape of training data',X_train.shape)
print('Shape of training labels', y_train.shape)
print('Shape of testing data', X_test.shape)
print('Shape of testing labels',y_test.shape)
```

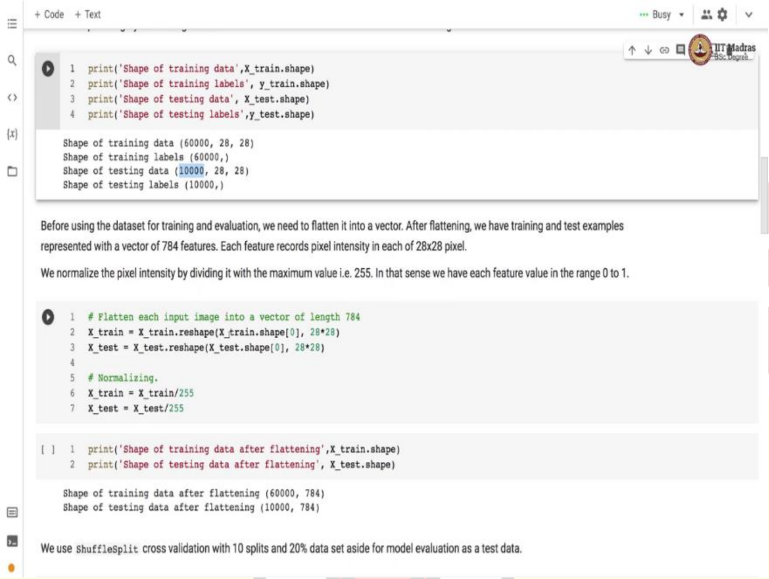
```
Shape of training data (60000, 28, 28)
Shape of training labels (60000,)
Shape of testing data (10000, 28, 28)
Shape of testing labels (10000,)
```

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.

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

(Refer Slide Time: 01:44)



```
print('Shape of training data',X_train.shape)
print('Shape of training labels', y_train.shape)
print('Shape of testing data', X_test.shape)
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.

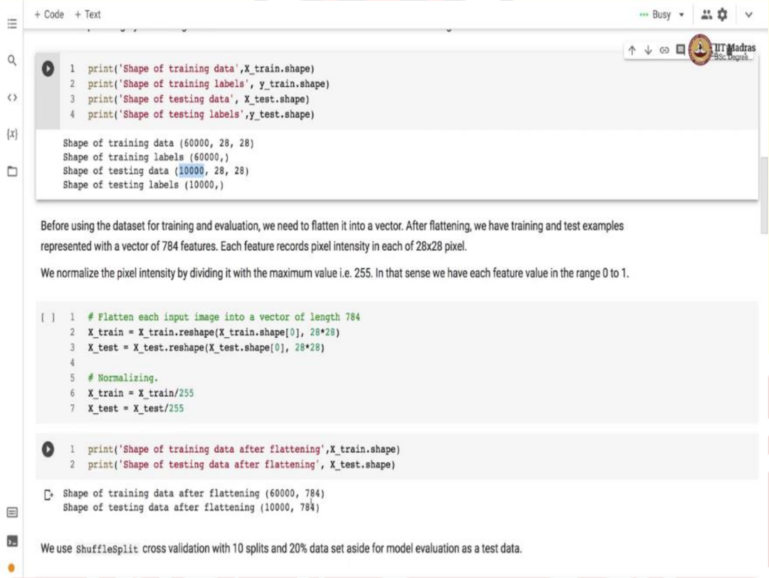
We normalize the pixel intensity by dividing it with the maximum value i.e. 255. In that sense we have each feature value in the range 0 to 1.

```
# Flatten each input image into a vector of length 784
X_train = X_train.reshape(X_train.shape[0], 28*28)
X_test = X_test.reshape(X_test.shape[0], 28*28)

# Normalizing.
X_train = X_train/255
X_test = X_test/255
```

```
print('Shape of training data after flattening',X_train.shape)
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.



```
print('Shape of training data',X_train.shape)
print('Shape of training labels', y_train.shape)
print('Shape of testing data', X_test.shape)
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.

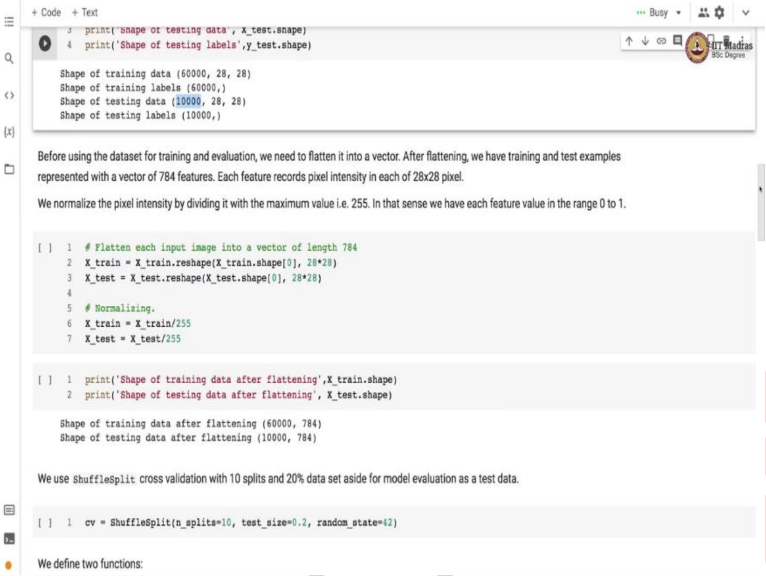
We normalize the pixel intensity by dividing it with the maximum value i.e. 255. In that sense we have each feature value in the range 0 to 1.

```
# Flatten each input image into a vector of length 784
X_train = X_train.reshape(X_train.shape[0], 28*28)
X_test = X_test.reshape(X_test.shape[0], 28*28)

# Normalizing.
X_train = X_train/255
X_test = X_test/255
```

```
print('Shape of training data after flattening',X_train.shape)
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.



```
print('Shape of testing data', X_test.shape)
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.

```
# Flatten each input image into a vector of length 784
X_train = X_train.reshape(X_train.shape[0], 28*28)
X_test = X_test.reshape(X_test.shape[0], 28*28)

# Normalizing.
X_train = X_train/255
X_test = X_test/255
```

```
print('Shape of training data after flattening',X_train.shape)
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)
```

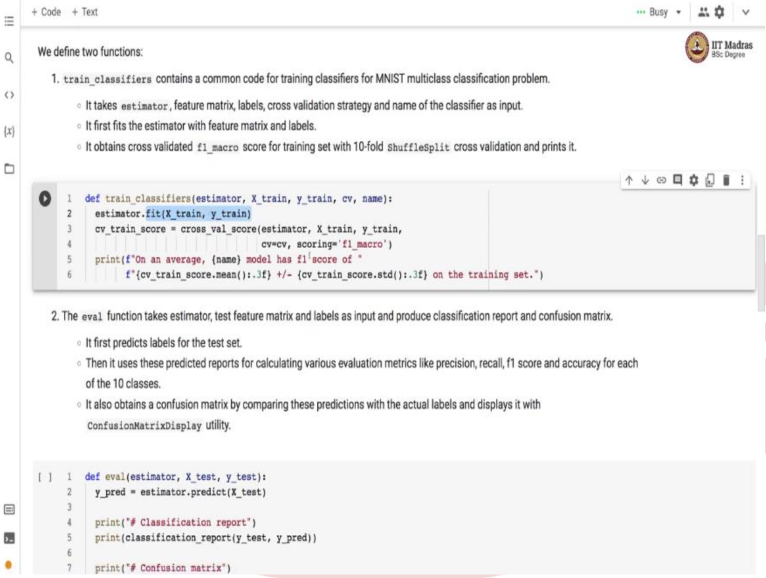
We use `ShuffleSplit` cross validation with 10 splits and 20% data set aside for model evaluation as a test data.

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

We define two functions:

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

(Refer Slide Time: 02:16)



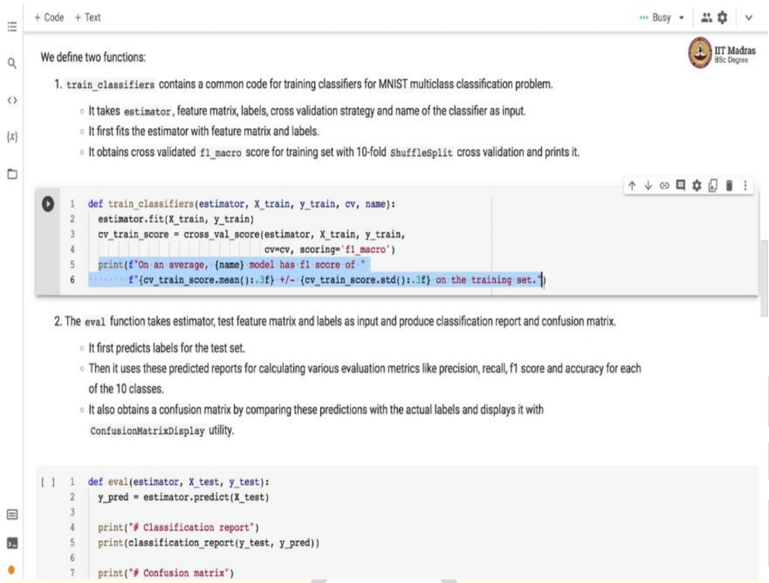
We define two functions:

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.

```
def train_classifiers(estimator, X_train, y_train, cv, name):
  estimator.fit(X_train, y_train)
  cv_train_score = cross_val_score(estimator, X_train, y_train,
                                   cv=cv, scoring='f1_macro')
  print(f"On an average, {name} model has f1 score of "
        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.

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

  print("# Classification report")
  print(classification_report(y_test, y_pred))

  print("# Confusion matrix")
```



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.

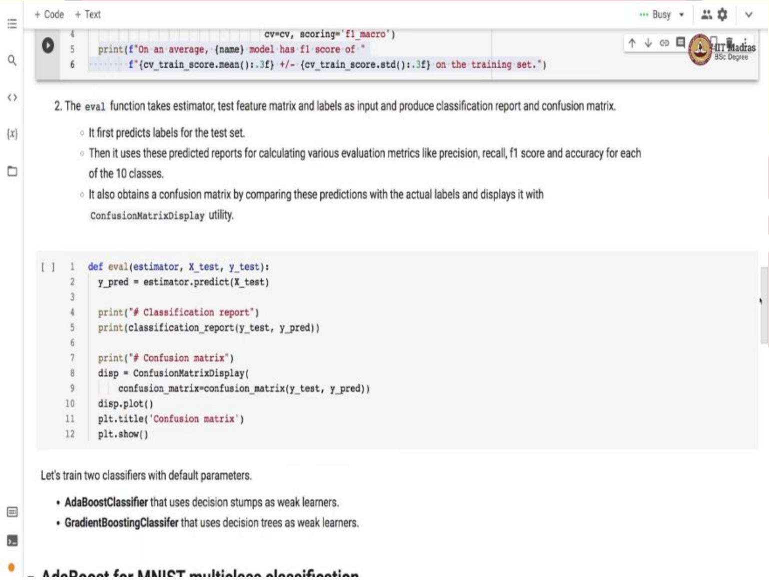
```
def train_classifiers(estimator, X_train, y_train, cv, name):
  estimator.fit(X_train, y_train)
  cv_train_score = cross_val_score(estimator, X_train, y_train,
                                   cv=cv, scoring='f1_macro')
  print(f"On an average, {name} model has f1 score of "
        f"{cv_train_score.mean():.3f} +/- {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.

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

  print("# Classification report")
  print(classification_report(y_test, y_pred))

  print("# Confusion matrix")
```

We use ShufflesSplit cross-validation strategy with 10-folds. And we set aside 20 %examples as test data for model evaluation. We define 2 functions. 1 is train _classifier that contains common code for training classifier 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 it prints it with these 2 statements.

(Refer Slide Time: 03:03)



```
                                   cv=cv, scoring='f1_macro')
  print(f"On an average, {name} model has f1 score of "
        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.

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

  print("# Classification report")
  print(classification_report(y_test, y_pred))

  print("# Confusion matrix")
  disp = ConfusionMatrixDisplay(
      confusion_matrix=confusion_matrix(y_test, y_pred))
  disp.plot()
  plt.title('Confusion matrix')
  plt.show()
```

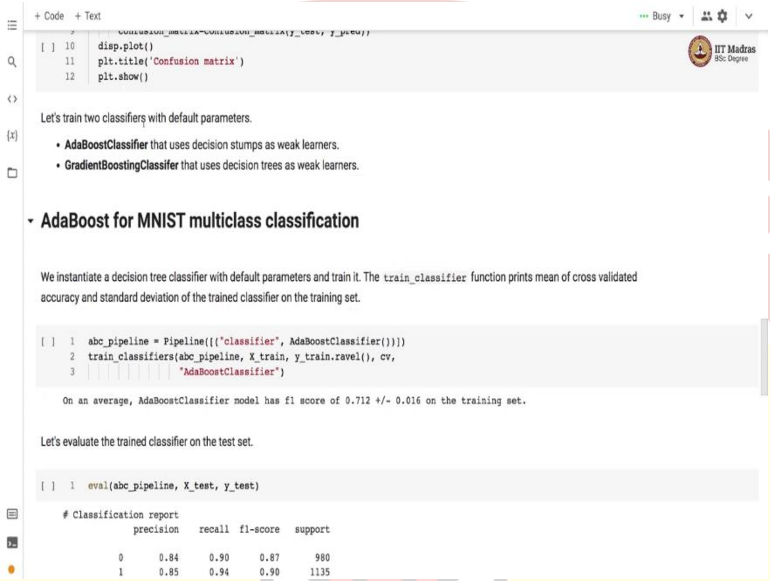
Let's train two classifiers with default parameters.

- **AdaBoostClassifier** that uses decision stumps as weak learners.
- **GradientBoostingClassifer** that uses decision trees as weak learners.

AdaBoost for MNIST multiclass classification

The eval function on the other hand takes estimator test feature matrix and labels as input and produce classification _report and confusion _matrix. It first predicts label for the test set. Then it uses the predicted labels for obtaining the classification _report. And in classification _report, we have precision recall, and f1 _score for each of the 10 classes. It also obtains confusion _matrix by comparing these predictions and displayed with ConfusionMatrixDisplay utility.

(Refer Slide Time: 03:38)



```
        confusion_matrix=confusion_matrix(y_test, y_pred))
[ ]  disp.plot()
     plt.title('Confusion matrix')
     plt.show()
```

Let's train two classifiers with default parameters.

- **AdaBoostClassifier** that uses decision stumps as weak learners.
- **GradientBoostingClassifer** that uses decision trees as weak learners.

AdaBoost 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.

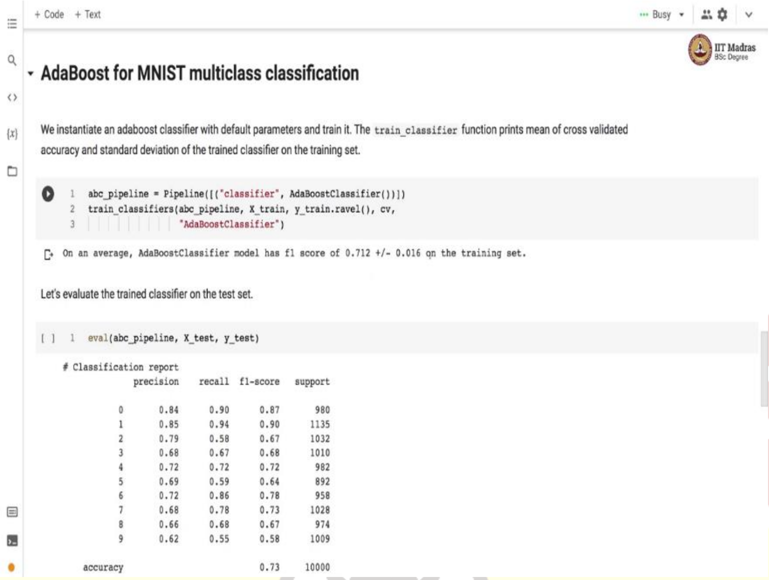
```
[ ]  abc_pipeline = Pipeline([("classifier", AdaBoostClassifier())])
     train_classifiers(abc_pipeline, X_train, y_train.ravel(), cv,
                       "AdaBoostClassifier")

  On an average, AdaBoostClassifier model has f1 score of 0.712 +/- 0.016 on the training set.
```

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

```
[ ]  eval(abc_pipeline, X_test, y_test)

  # Classification report
                precision    recall  f1-score   support

             0       0.84      0.90      0.87       980
             1       0.85      0.94      0.90      1135
```

We will train 2 classifiers with default parameters 1 is AdaBoostClassifier, and 2 is GradientBoostingClassifier. AdaBoostClassifier uses decision stumps as weak learner whereas GradientBoostingClassifier uses decision trees as weak learners.

(Refer Slide Time: 03:56)



AdaBoost for MNIST multiclass classification

We instantiate an adaboost 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.

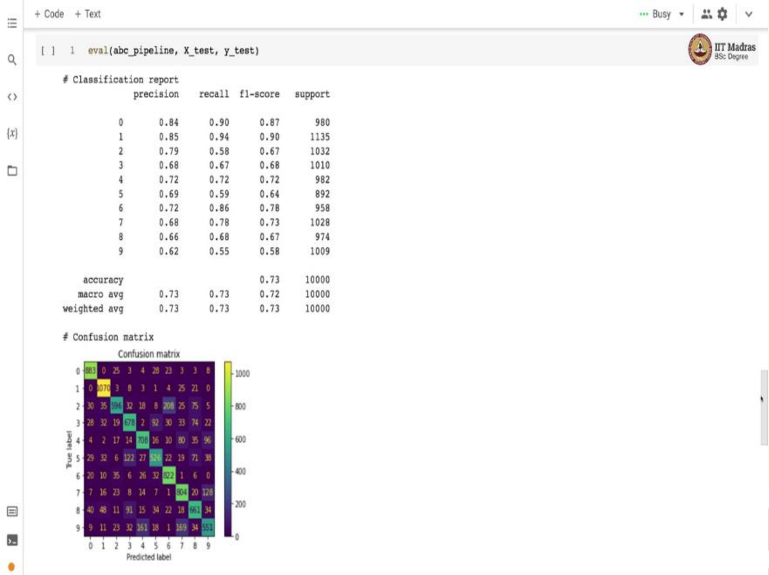
```
abc_pipeline = Pipeline([("classifier", AdaBoostClassifier())])
train_classifiers(abc_pipeline, X_train, y_train.ravel(), cv,
                  "AdaBoostClassifier")
```

```
On an average, AdaBoostClassifier model has f1 score of 0.712 +/- 0.016 qn the training set.
```

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

```
[ ]  eval(abc_pipeline, X_test, y_test)
```

```
# Classification report
              precision    recall  f1-score   support

           0       0.84      0.90      0.87       980
           1       0.85      0.94      0.90      1135
           2       0.79      0.58      0.67      1032
           3       0.68      0.67      0.68      1010
           4       0.72      0.72      0.72       982
           5       0.69      0.59      0.64       892
           6       0.72      0.86      0.78       958
           7       0.68      0.78      0.73      1028
           8       0.66      0.68      0.67       974
           9       0.62      0.55      0.58      1009

    accuracy                           0.73     10000
```



```
[ ]  eval(abc_pipeline, X_test, y_test)
```

```
# Classification report
              precision    recall  f1-score   support

           0       0.84      0.90      0.87       980
           1       0.85      0.94      0.90      1135
           2       0.79      0.58      0.67      1032
           3       0.68      0.67      0.68      1010
           4       0.72      0.72      0.72       982
           5       0.69      0.59      0.64       892
           6       0.72      0.86      0.78       958
           7       0.68      0.78      0.73      1028
           8       0.66      0.68      0.67       974
           9       0.62      0.55      0.58      1009

    accuracy                           0.73     10000
```

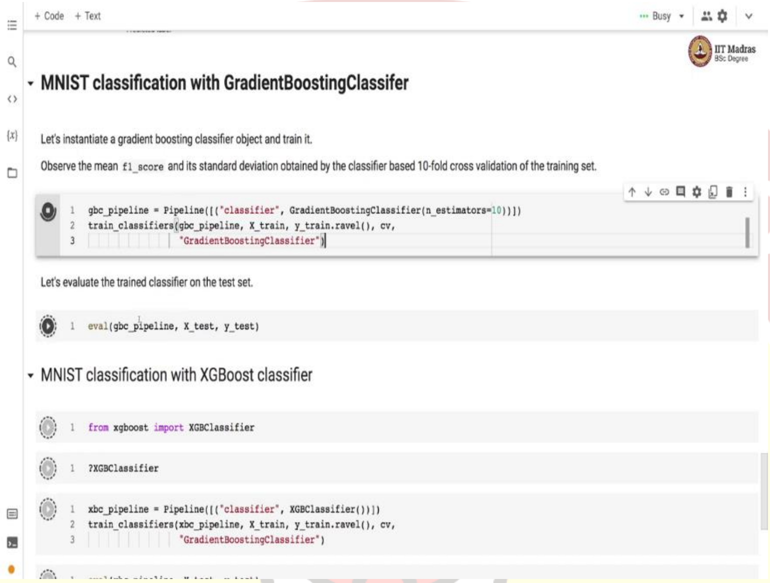
```
   macro avg       0.73      0.73      0.72     10000
weighted avg       0.73      0.73      0.73     10000

# Confusion matrix
```

	0	1	2	3	4	5	6	7	8	9
0	883	0	25	3	4	28	23	3	3	8
1	0	1070	3	8	3	1	4	25	21	0
2	30	35	596	32	18	8	208	25	75	5
3	28	32	19	678	2	92	30	33	74	22
4	4	2	17	14	708	16	10	80	35	96
5	29	32	6	122	27	526	22	19	71	38
6	20	10	35	6	26	32	822	1	6	0
7	7	16	23	8	14	7	1	804	20	128
8	40	48	11	91	15	34	22	18	661	34
9	9	11	23	32	161	18	1	169	34	551

We instantiate AdaBoostClassifier with default parameters and train it with train _classifier function. The train _classifier function prints mean of cross validated accuracy and standard deviation of the train classifier on the training set. Here, we see that AdaBoostClassifier model obtains f1 _score of 0.712 with a standard deviation of 0.016 on the training set. And you can see that AdaBoostClassifier does not really get us that great classifier if you compare this with random forest classifier or bagging classifier that we saw in the previous collab.

So, there are a lot of confusions that are happening between different classes, for example, class 9 and 4 or class 7 and class 9, or class 5 and class 3 or class 6 and class 2, there are a lot of confusions and because of which the accuracy is merely 0.73 or 73%.

(Refer Slide Time: 04:59)



MNIST classification with GradientBoostingClassifer

Let's instantiate a gradient boosting classifier object and train it.

Observe the mean `f1_score` and its standard deviation obtained by the classifier based 10-fold cross validation of the training set.

```
gbc_pipeline = Pipeline([("classifier", GradientBoostingClassifier(n_estimators=10))])
train_classifiers(gbc_pipeline, X_train, y_train.ravel(), cv,
                  "GradientBoostingClassifier")
```

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

```
eval(gbc_pipeline, X_test, y_test)
```

MNIST classification with XGBoost classifier

```
from xgboost import XGBClassifier
```

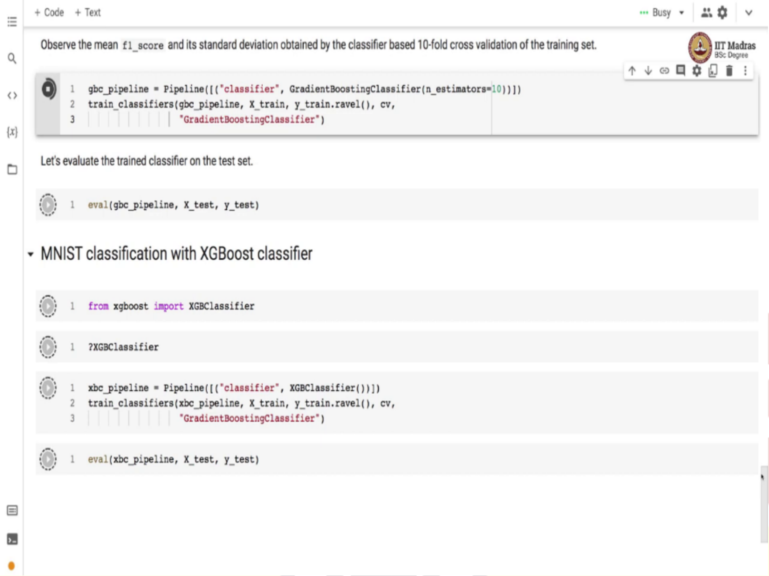
```
?XGBClassifier
```

```
xbc_pipeline = Pipeline([("classifier", XGBClassifier())])
train_classifiers(xbc_pipeline, X_train, y_train.ravel(), cv,
                  "GradientBoostingClassifier")
```

Now here we have given the code for training the model with GradientBoostingClassifier and XGBoost classifier, and as an exercise, you should run this code and obtain the accuracy with the GradientBoostingClassifier as well as XGBoost classifier and compare it with the accuracy that you obtained through AdaBoostClassifier.

So, here what we have done is we have created a pipeline object with GradientBoostingClassifier as 1 of the stages and the GradientBoostingClassifier is instantiated with number of estimators = 10. And then we train the GradientBoostingClassifier pipeline with train _classifier function. So, what you have to do is you have to run this and find out what is the performance that we get and also run the evaluation pipeline and find out the performance on the test set from the GradientBoostingClassifier.

(Refer Slide Time: 06:01)



Observe the mean `f1_score` and its standard deviation obtained by the classifier based 10-fold cross validation of the training set.

```
gbc_pipeline = Pipeline([("classifier", GradientBoostingClassifier(n_estimators=10))])
train_classifiers(gbc_pipeline, X_train, y_train.ravel(), cv,
                  "GradientBoostingClassifier")
```

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

```
eval(gbc_pipeline, X_test, y_test)
```

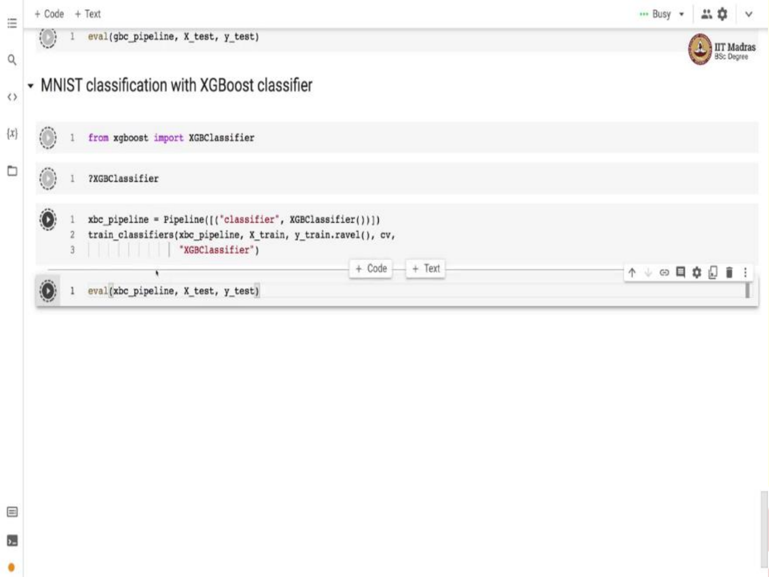
MNIST classification with XGBoost classifier

```
from xgboost import XGBClassifier
```

```
?XGBClassifier
```

```
xbc_pipeline = Pipeline([("classifier", XGBClassifier())])
train_classifiers(xbc_pipeline, X_train, y_train.ravel(), cv,
                  "GradientBoostingClassifier")
```

```
eval(xbc_pipeline, X_test, y_test)
```



```
eval(gbc_pipeline, X_test, y_test)
```

MNIST classification with XGBoost classifier

```
from xgboost import XGBClassifier
```

```
?XGBClassifier
```

```
xbc_pipeline = Pipeline([("classifier", XGBClassifier())])
train_classifiers(xbc_pipeline, X_train, y_train.ravel(), cv,
                  "XGBClassifier")
```

```
eval(xbc_pipeline, X_test, y_test)
```

So, GradientBoostingClassifier generally takes longer to run than AdaBoostClassifier. XGBoost classifier on the other hand, is expected to be efficient version of GradientBoostingClassifier. It is very similar to GradientBoostingClassifier except that it performs regularization to obtain better generalization performance.

We can import the XGBClassifier from XGBoost module. So, XGBClassifier implements classification with XGBoost. If you are interested in reading about documentation now, XGBClassifier and you can access the documentation by putting the ? followed by XGBClassifier.

And if you run this cell, you will get to see the documentation of this classifier. So, here we instantiate a pipeline object with XGBClassifier as a classification stage. And we train the XGBClassifier with train _classifier function by supplying the estimator object, the feature matrix label vector, the cross-validation strategy and name of the classifier.

So, just like gradient boosting, you also have to run this particular code and check out what is the accuracy that or what is the f1-score that we get on the training set, as well as find out the accuracy on the test set. And as an exercise compare the accuracies of AdaBoost gradient boosting and XGBoost classifier.

So, with this, we have demonstrated how to perform classification with boosting techniques. Now you have three more classifiers in your tool set, which is AdaBoostClassifier, GradientBoostingClassifier and XGBoost classifier, GradientBoostingClassifier and XGBoost classifiers work well on structured data. In many of the Kaggle competitions, we have seen that XGBoost and GradientBoostingClassifier give the state of the art performance whenever they are applied on structured dataset.