

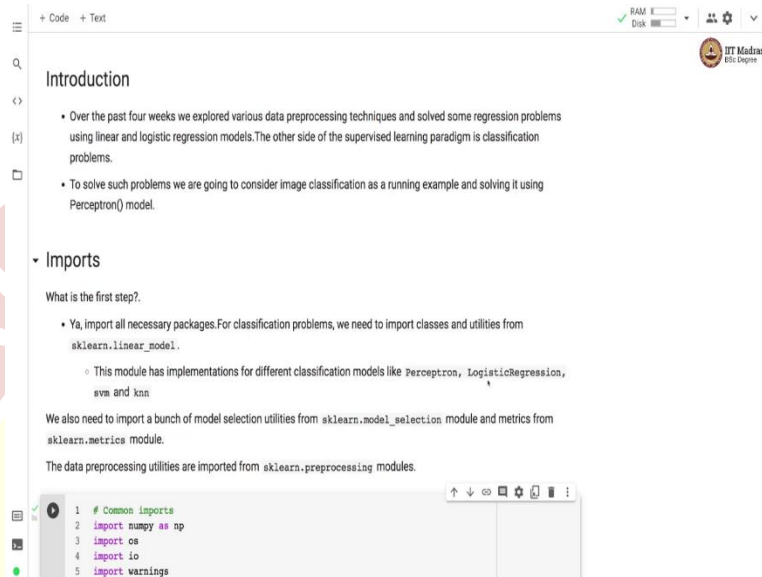
IIT Madras

ONLINE DEGREE

Machine Learning Program

Demonstration Binary class Image Classification with Perceptron

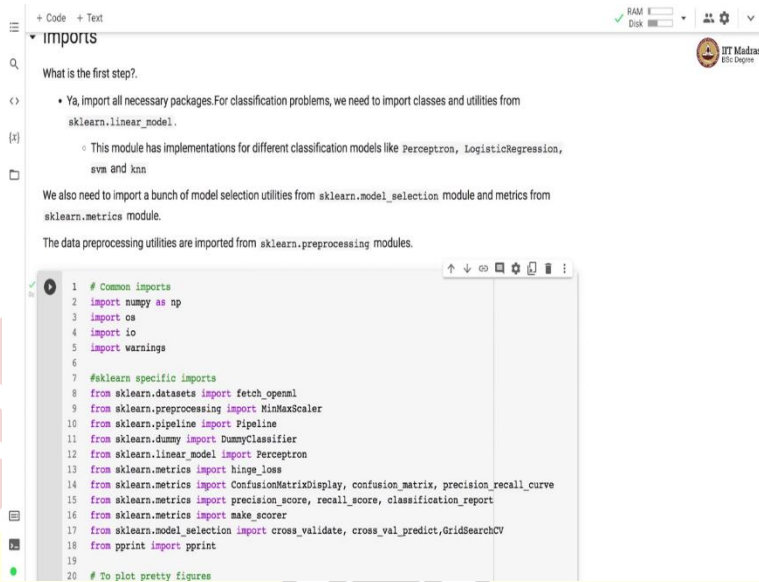
(Refer Slide Time: 0:10)



Namaste, welcome to the next video of machine learning practice course. In this video, we will be using a perceptron model for image classification. Specifically, we are given an image of a digit and we are required to predict the digit present in the image. This is an instance of a multi class classification problem. There are 10 digits in total. So, in a given image, there is a single digit present, and we have to predict which one of the 10 digits are present in that image.

Initially, we will solve this problem as a binary classification problem, and later solve a complete multi class classification problem. We will be recording this collab in two parts. In the first part, we will be recording the binary classification problem, and in the second part we will be recording the multi class classification problem. So, let us begin. So, the first step will import all necessary packages for the classification problem.

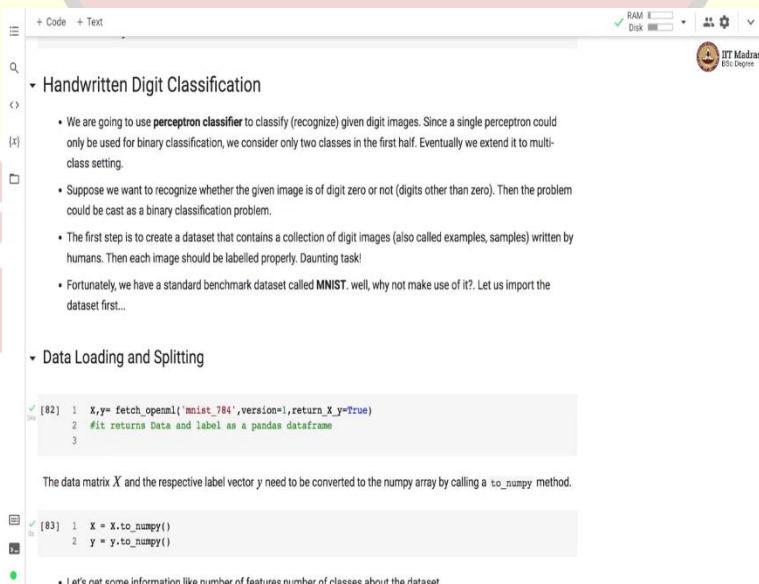
(Refer Slide Time: 1:23)



```
1 # Common imports
2 import numpy as np
3 import os
4 import io
5 import warnings
6
7 #sklearn specific imports
8 from sklearn.datasets import fetch_openml
9 from sklearn.preprocessing import MinMaxScaler
10 from sklearn.pipeline import Pipeline
11 from sklearn.dummy import DummyClassifier
12 from sklearn.linear_model import Perceptron
13 from sklearn.metrics import hinge_loss
14 from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix, precision_recall_curve
15 from sklearn.metrics import precision_score, recall_score, classification_report
16 from sklearn.metrics import make_scorer
17 from sklearn.model_selection import cross_validate, cross_val_predict, GridSearchCV
18 from pprint import pprint
19
20 # To plot pretty figures
```

We need to import classes and utilities from sklearn. So, we are going to import a perceptron classifier from sklearn. linear_model. We also import a dummy classifier as a baseline classifier. Then there are other utilities like pipeline are also imported. We are also importing MinMaxScalar() as a pre processing as a pre processing module. And data set will be loaded from fetch_openml library. We are also importing a bunch of metrics and model selection utilities.

(Refer Slide Time: 2:13)



```
[82] 1 X,y= fetch_openml('mnist_784',version=1,return_X_y=True)
      2 #it returns Data and label as a pandas dataframe
      3

The data matrix X and the respective label vector y need to be converted to the numpy array by calling a to_numpy method.

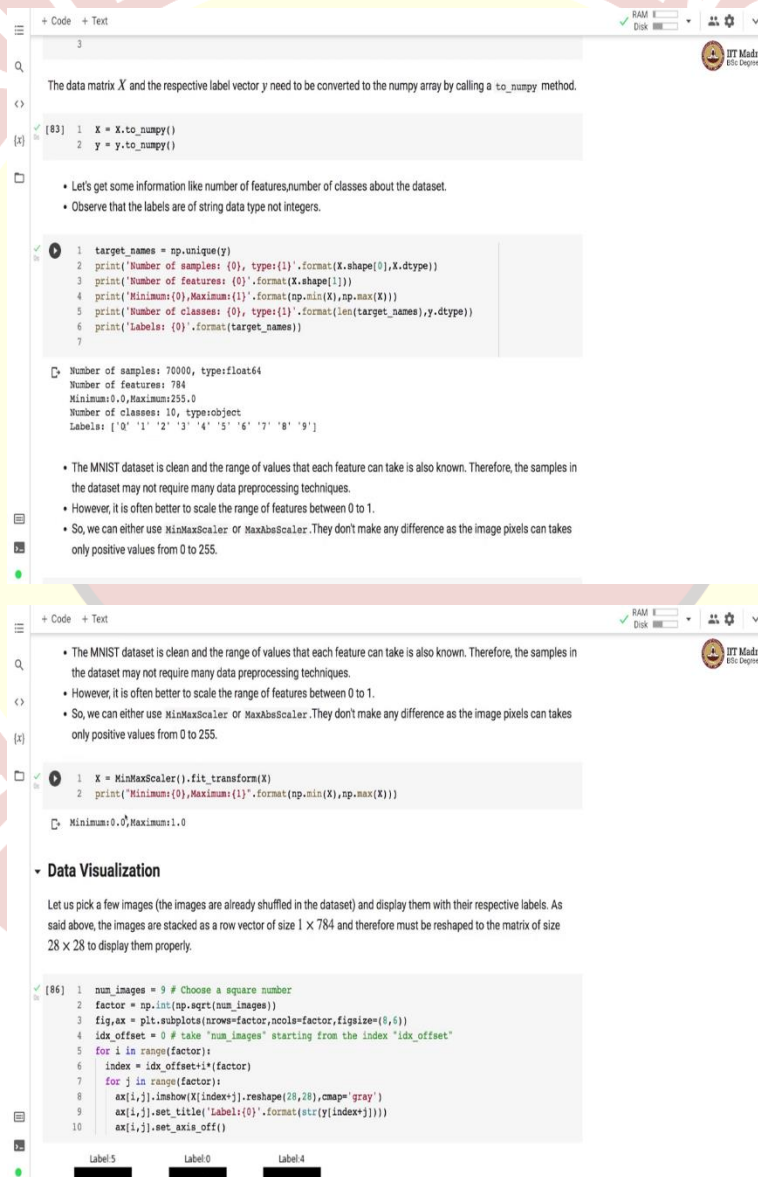
[83] 1 X = X.to_numpy()
      2 y = y.to_numpy()
```

So, we are going to use perceptron classifier to classify a given image of a digit. So, in the first part of the problem, we are going to solve this problem as a binary classification problem. So, for

that, what we will do is, we will check whether the given image is that of digit 0 or not. So, as a first step, we need to create a dataset that contains a collection of image of digit written by human.

And then each image should be also labeled properly. And you can imagine that this is a challenging task and also a time consuming one. However, we have a standard benchmark called a MNIST data set, and we will be using the MNIST data set. So, MNIST data set has an image of a digit, and along with the label of the digit present in that image.

(Refer Slide Time: 3:16)



```
3
The data matrix X and the respective label vector y need to be converted to the numpy array by calling a to_numpy method.

[83] 1 X = X.to_numpy()
      2 y = y.to_numpy()

• Let's get some information like number of features, number of classes about the dataset.
• Observe that the labels are of string data type not integers.

1 target_names = np.unique(y)
2 print('Number of samples: (0), type:{}'.format(X.shape[0], X.dtype))
3 print('Number of features: (0)', format(X.shape[1]))
4 print('Minimum: (0), Maximum: (1)', format(np.min(X), np.max(X)))
5 print('Number of classes: (0), type:{}'.format(len(target_names), y.dtype))
6 print('Labels: (0)', format(target_names))
7

Number of samples: 70000, type:float64
Number of features: 784
Minimum: 0.0, Maximum: 255.0
Number of classes: 10, type:object
Labels: ['0' '1' '2' '3' '4' '5' '6' '7' '8' '9']

• The MNIST dataset is clean and the range of values that each feature can take is also known. Therefore, the samples in the dataset may not require many data preprocessing techniques.
• However, it is often better to scale the range of features between 0 to 1.
• So, we can either use MinMaxScaler or MaxAbsScaler. They don't make any difference as the image pixels can take only positive values from 0 to 255.

+ Code + Text
The MNIST dataset is clean and the range of values that each feature can take is also known. Therefore, the samples in the dataset may not require many data preprocessing techniques.
• However, it is often better to scale the range of features between 0 to 1.
• So, we can either use MinMaxScaler or MaxAbsScaler. They don't make any difference as the image pixels can take only positive values from 0 to 255.

1 X = MinMaxScaler().fit_transform(X)
2 print('Minimum: (0), Maximum: (1)', format(np.min(X), np.max(X)))

Minimum: 0.0, Maximum: 1.0

• Data Visualization

Let us pick a few images (the images are already shuffled in the dataset) and display them with their respective labels. As said above, the images are stacked as a row vector of size 1 x 784 and therefore must be reshaped to the matrix of size 28 x 28 to display them properly.

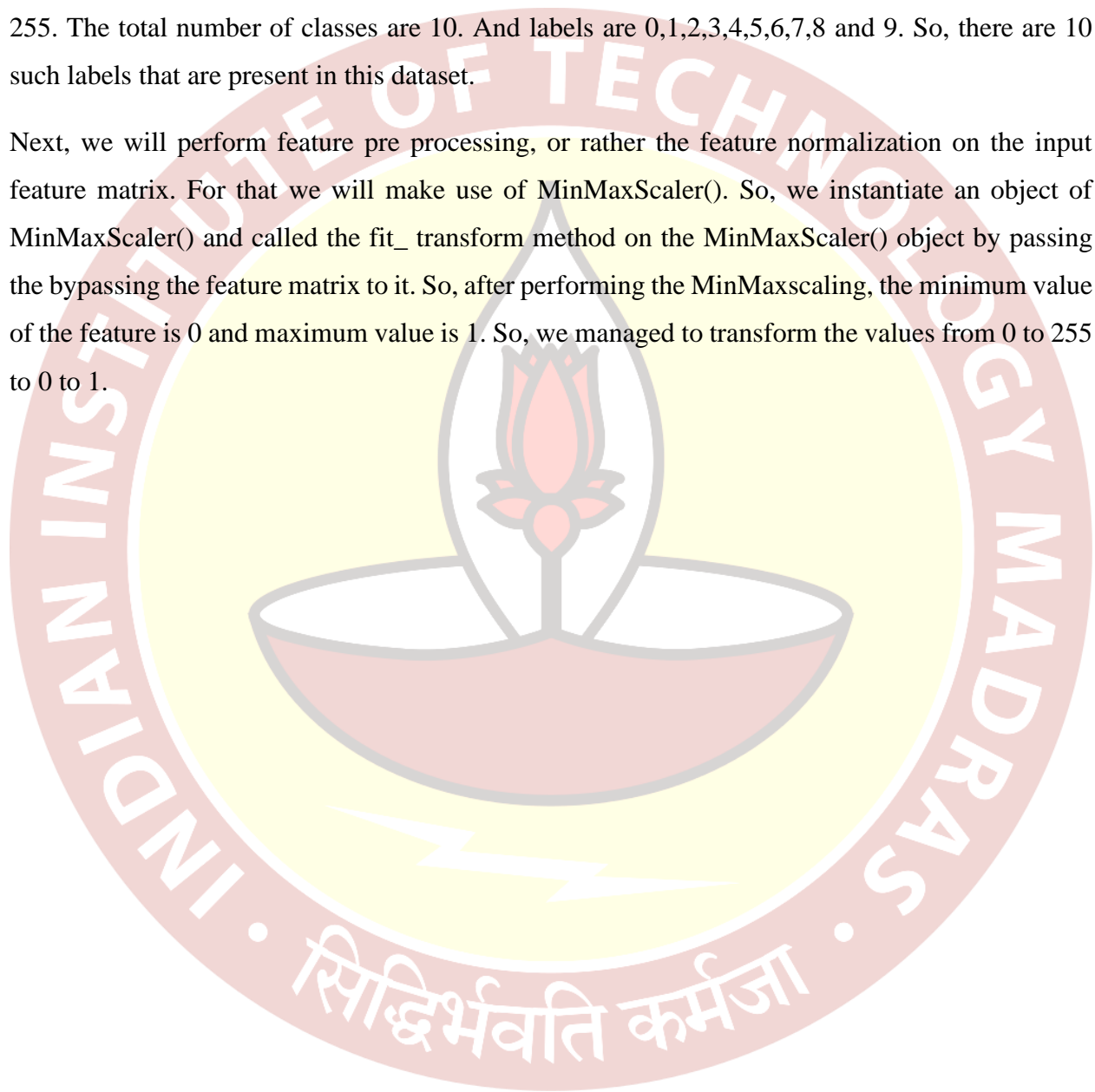
[86] 1 num_images = 9 # Choose a square number
      2 factor = np.int(np.sqrt(num_images))
      3 fig, ax = plt.subplots(nrows=factor, ncols=factor, figsize=(6,6))
      4 idx_offset = 0 # take "num_images" starting from the index "idx_offset"
      5 for i in range(factor):
      6     index = idx_offset + i * factor
      7     for j in range(factor):
      8         ax[i,j].imshow(X[index+j].reshape((28,28)), cmap='gray')
      9         ax[i,j].set_title('Label: (0)', format(str(y[index+j])))
     10         ax[i,j].set_axis_off()
```

So, we will be fetching the MNIST dataset from fetch_openml. So, there is a mnist_784 dataset and we will be returning MNIST data set in form of a feature vector and label. So, you can see that

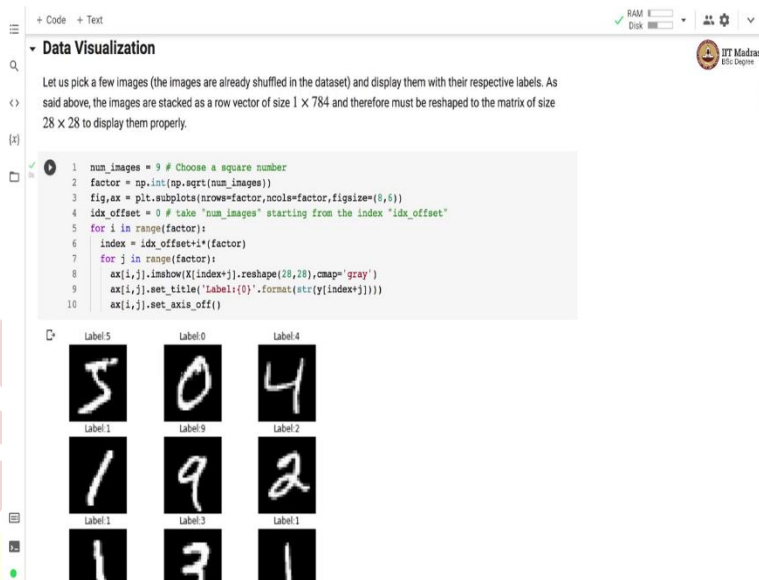
there are 70,000 total samples in this data set and each image comes with 784 feature. So, what happens is each image is present as a 28×28 grid. And if you multiply 28 with itself or the square of 28, it is 784.

So, each feature represent a value in the pixel and the minimum value is 0 and maximum value is 255. The total number of classes are 10. And labels are 0,1,2,3,4,5,6,7,8 and 9. So, there are 10 such labels that are present in this dataset.

Next, we will perform feature pre processing, or rather the feature normalization on the input feature matrix. For that we will make use of `MinMaxScaler()`. So, we instantiate an object of `MinMaxScaler()` and called the `fit_transform` method on the `MinMaxScaler()` object by passing the bypassing the feature matrix to it. So, after performing the `MinMaxscaling`, the minimum value of the feature is 0 and maximum value is 1. So, we managed to transform the values from 0 to 255 to 0 to 1.

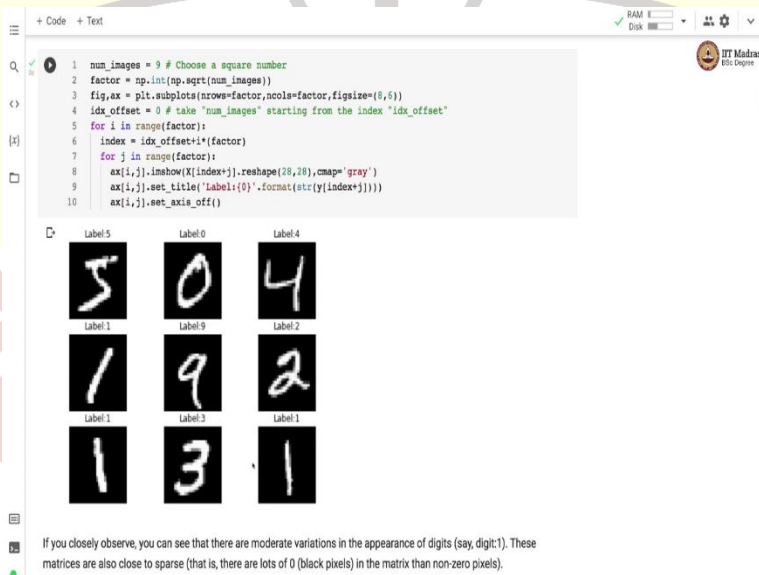


(Refer Slide Time: 5:14)



Let us pick up a few images and display them along with the labels. So, each image is stacked as a row of size 1×784 . So, what we will do is we will first reshape, reshape each image into a matrix of size 28×28 and display them.

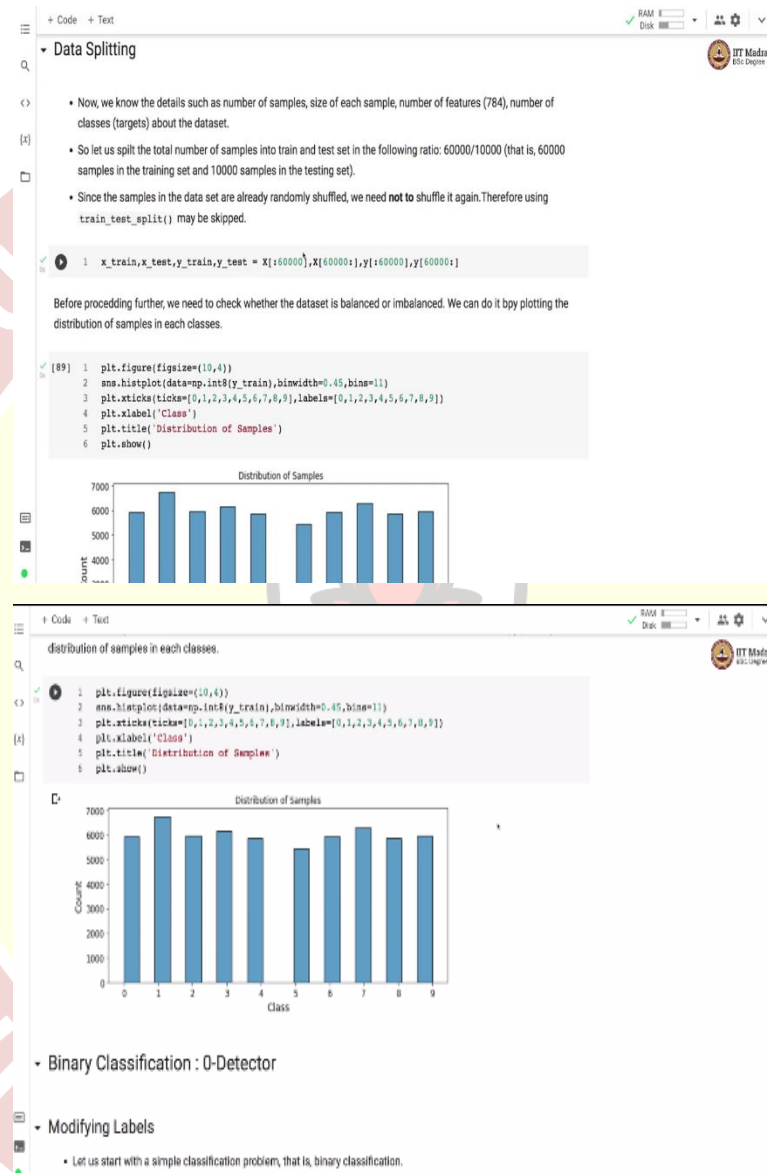
(Refer Slide Time: 5:42)



So here, there are 9 images that are selected and displayed on the screen. So, you can see that this is the image of handwritten digit 5, handwritten digit 5, this 0 is 4, and you can also see the label for this particular image. So this image is what is the input and this is a label that we are supposed to predict by processing this image through perceptron classifier. You can see that there is some

variation in the way digits are written. So here, digit 1 is repeated. So there are 3 instances of digit 1 and you can see that there is slight variation in the way one is written.

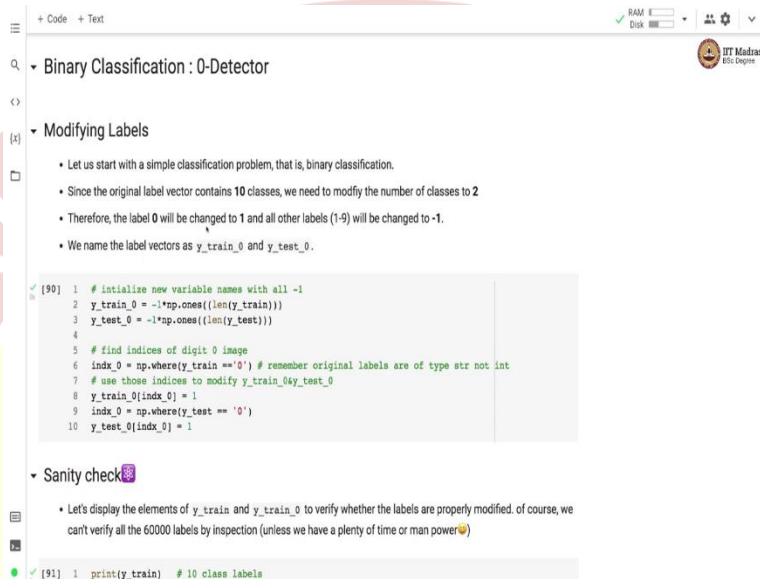
(Refer Slide Time: 6:29)



Next we will split the data into training and test set. So, what we will do is we will set aside 60,000 images for training and 10,000 images for test since the data is already randomly shuffled when not shuffle it again. And here we are using a manual method of the splitting. So, you can also use `train_test` and `_split` method on this data set to obtain training and test splits.

We will also check if our data set is balanced or imbalanced. So for that, what we do is we plot the distribution of samples in each class. So, we have 10 different classes. And you can see that roughly there are about 6000 samples in each class.

(Refer Slide Time: 7:29)



```
+ Code + Text
RAM 16GB
Disk 100GB
JIT Madras
800.000000

Binary Classification : 0-Detector

Modifying Labels

• Let us start with a simple classification problem, that is, binary classification.
• Since the original label vector contains 10 classes, we need to modify the number of classes to 2
• Therefore, the label 0 will be changed to 1 and all other labels (1-9) will be changed to -1.
• We name the label vectors as y_train_0 and y_test_0.

[90] 1 # initialize new variable names with all -1
2 y_train_0 = -1*np.ones((len(y_train)))
3 y_test_0 = -1*np.ones((len(y_test)))
4
5 # find indices of digit 0 image
6 indx_0 = np.where(y_train == '0') # remember original labels are of type str not int
7 # use those indices to modify y_train_0&y_test_0
8 y_train_0[indx_0] = 1
9 indx_0 = np.where(y_test == '0')
10 y_test_0[indx_0] = 1

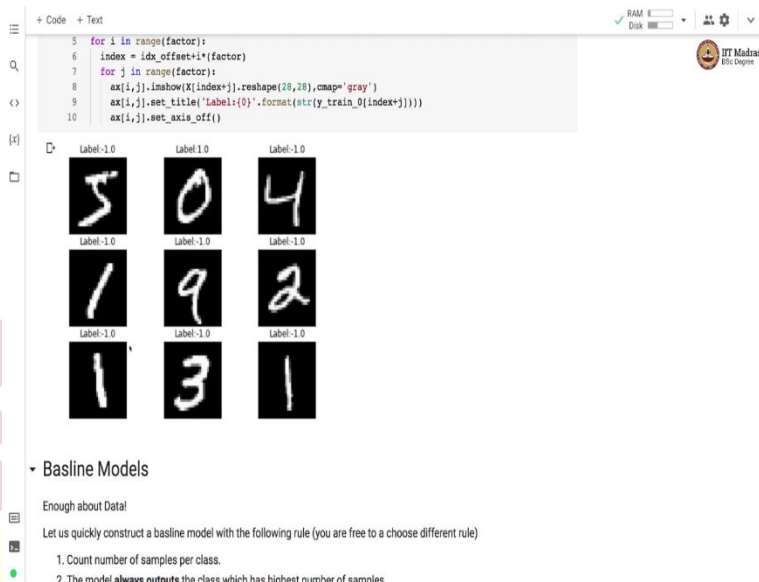
Sanity check

• Let's display the elements of y_train and y_train_0 to verify whether the labels are properly modified. of course, we
  can't verify all the 60000 labels by inspection (unless we have a plenty of time or man power🤖)

[91] 1 print(y_train) # 10 class labels
```

Let us solve a binary classification problem for detecting a 0 digit. So, we call it as a 0 detector. So, in order to train a model for 0 detector, we need to first modify our labels. Since the original label vector that contains 10 classes, we need to modify to 2 classes, 1 is the label 0, which will be changed to 1 and all other labels will be changed to -1. So, we will have two labels +1 and -1.

(Refer Slide Time: 8:09)



```
5 for i in range(factor):
6     index = idx_offset+i*(factor)
7     for j in range(factor):
8         ax[i,j].imshow(X[index+j].reshape(28,28), cmap='gray')
9         ax[i,j].set_title('Label:{}'.format(str(y_train_0[index+j])))
10        ax[i,j].set_axis_off()
```

Label: -1.0 Label: 1.0 Label: -1.0

Label: -1.0 Label: -1.0 Label: -1.0

Label: -1.0 Label: -1.0 Label: -1.0

5 0 4

1 9 2

1 3 1

Basline Models

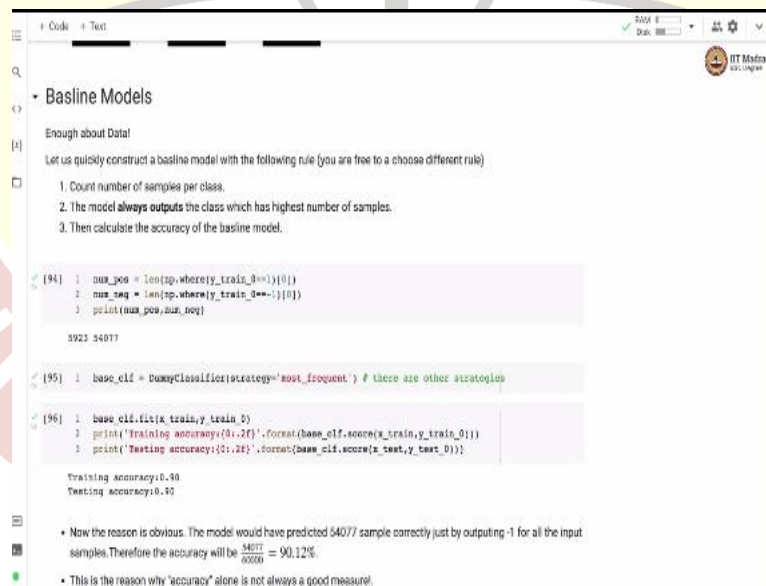
Enough about Data!

Let us quickly construct a baseline model with the following rule (you are free to choose different rule)

1. Count number of samples per class.
2. The model **always** outputs the class which has highest number of samples

So, you can see that wherever we have digit 0 the label is 1 and wherever there is a nonzero digit, the label is -1. So you can see here, this is the image of digit 5 and label is -1. For image of digit 4 label is also -1, and so on.

(Refer Slide Time: 8:35)



```
94 1 max_pos = len(sp.where(y_train_0==1)[0])
2 max_neg = len(sp.where(y_train_0==-1)[0])
3 print(max_pos,max_neg)

3923 54077

95 1 base_clf = DummyClassifier(strategy='most_frequent') # there are other strategies

96 1 base_clf.fit(x_train,y_train_0)
2 print('Training accuracy:{0.2f}'.format(base_clf.score(x_train,y_train_0)))
3 print('Testing accuracy:{0.2f}'.format(base_clf.score(x_test,y_test_0)))

Training accuracy:0.90
Testing accuracy:0.90
```

Now the reason is obvious. The model would have predicted 54077 sample correctly just by outputting -1 for all the input samples. Therefore the accuracy will be $\frac{54077}{60000} = 90.12\%$.

This is the reason why "accuracy" alone is not always a good measure!

```
+ Code + Text
RAM 8 GB
Disk 100 GB
JIT Madras
Elio Engine

1. Count number of samples per class.
2. The model always outputs the class which has highest number of samples.
3. Then calculate the accuracy of the baseline model.

[94] 1 num_pos = len(np.where(y_train_0==1)[0])
     2 num_neg = len(np.where(y_train_0==-1)[0])
     3 print(num_pos,num_neg)

5923 54077

[95] 1 base_clf = DummyClassifier(strategy='most_frequent') # there are other strategies

[96] 1 base_clf.fit(x_train,y_train_0)
     2 print('Training accuracy:{0:.2f}'.format(base_clf.score(x_train,y_train_0)))
     3 print('Testing accuracy:{0:.2f}'.format(base_clf.score(x_test,y_test_0)))

Training accuracy:0.90
Testing accuracy:0.90

• Now the reason is obvious. The model would have predicted 54077 sample correctly just by outputting -1 for all the input samples. Therefore the accuracy will be  $\frac{54077}{60000} = 90.12\%$ .
• This is the reason why "accuracy" alone is not always a good measure!

▼ Perceptron model
Before using Perceptron for Binary Classification, it will be helpful to recall the important concepts (equations) covered in technique course.
```

First, we will build a baseline model. And the simplest baseline model is to assign the label of the majority class to each example. So, we can use dummy classifier with strategy = most frequent. So, what will happen is that the label that is most frequent in the training data will be applied to each training example. So, you can see that with that, we get the training and test accuracy of 0.9. So, we are 90 percent accurate, by using the most frequent label. So here, in this case, there are 54,000 samples that are labeled with -1 and -1 is the most frequent label. So if we, assign the label -1 to each example, we get 90 percent accuracy. And you can see that this is not really a good major, because we are not able to detect 0's by utilizing by using this particular strategy. So, you can conclude that or you can again see that accuracy is not a very good major, at least in this case.

(Refer Slide Time: 10:05)

Recap (Theory)

Let us quickly recap various components in the general settings:

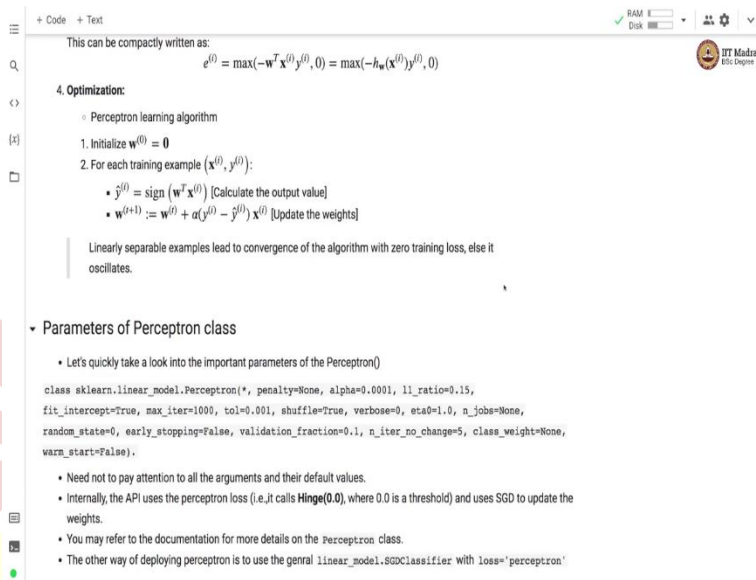
- 1. Training data:** ((features, label) or (X, y)), where label y is a **discrete** number from a finite set. **Features** in this case are pixel values of an image.
- 2. Model:**
$$h_w : y = g(w^T x)$$
$$= g(w_0 + w_1 x_1 + \dots + w_n x_n)$$
where,
 - w is a weight vector in $\mathbb{R}^{(n+1)}$ i.e. it has components: $\{w_0, w_1, \dots, w_n\}$
 - $g(\cdot)$ is a non-linear activation function given by a **signum** function:
$$g(z) = \begin{cases} +1, & \text{if } z \geq 0 \\ -1, & \text{otherwise (i.e. } z < 0) \end{cases}$$
- 3. Loss function:** Let $\hat{y}^{(i)} \in \{-1, +1\}$ be the prediction from perceptron and $y^{(i)}$ be the actual label for i -th example. The error is
$$e^{(i)} = \begin{cases} 0, & \text{if } \hat{y}^{(i)} = y^{(i)} \\ -w^T x^{(i)} y^{(i)}, & \text{otherwise (i.e. } \hat{y}^{(i)} \neq y^{(i)}) \end{cases}$$
This can be compactly written as:
$$e^{(i)} = \max(-w^T x^{(i)} y^{(i)}, 0) = \max(-h_w(x^{(i)}) y^{(i)}, 0)$$
- 4. Optimization:**
 - Perceptron learning algorithm
 - 1. Initialize $w^{(0)} = 0$
 - 2. For each training example $(x^{(i)}, y^{(i)})$:

Now what we will do is we will use perceptron for binary classification. Let us quickly recap the concepts of perceptron. So, in perceptron, we have training data consisting of feature matrix and a label vector. And the label is a discrete quantity from a finite set and here the features are pixel values. So, the perceptron model, we first perform the linear combination of weights and the features and then pass it through a nonlinear activation function.

And this nonlinear activation function in case of perceptron is based on the sign of the linear combination. If the linear combination is ≥ 0 then we assign the label of +1 otherwise we assign a label of -1. So, if the linear combination is a - number we assign the label of -1. Then we have a perceptron loss, which is $\max(-w^T x^{(i)} y^{(i)}, 0)$. So, the max of these quantities is assigned as a loss for i th example.

And we have a perceptron update tool as a perceptron learning algorithm. So we change the weight of the training example, if it is incorrectly classified in case of perceptron.

(Refer Slide Time: 11:49)



This can be compactly written as:

$$e^{(j)} = \max(-w^T x^{(j)} y^{(j)}, 0) = \max(-h_w(x^{(j)}) y^{(j)}, 0)$$

4. Optimization:

- Perceptron learning algorithm
- 1. Initialize $w^{(0)} = 0$
- 2. For each training example $(x^{(j)}, y^{(j)})$:
 - $\hat{y}^{(j)} = \text{sign}(w^T x^{(j)})$ [Calculate the output value]
 - $w^{(i+1)} := w^{(i)} + \alpha(y^{(j)} - \hat{y}^{(j)}) x^{(j)}$ [Update the weights]

Linearly separable examples lead to convergence of the algorithm with zero training loss, else it oscillates.

Parameters of Perceptron class

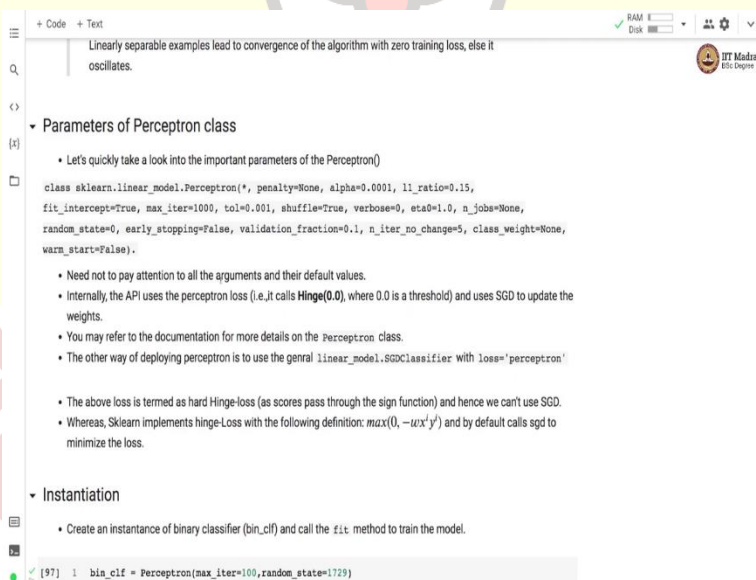
- Let's quickly take a look into the important parameters of the Perceptron()

```
class sklearn.linear_model.Perceptron(*, penalty=None, alpha=0.0001, l1_ratio=0.15,
fit_intercept=True, max_iter=1000, tol=0.001, shuffle=True, verbose=0, eta0=1.0, n_jobs=None,
random_state=0, early_stopping=False, validation_fraction=0.1, n_iter_no_change=5, class_weight=None,
warm_start=False).
```

- Need not to pay attention to all the arguments and their default values.
- Internally, the API uses the perceptron loss (i.e. it calls **Hinge(0.0)**, where 0.0 is a threshold) and uses SGD to update the weights.
- You may refer to the documentation for more details on the Perceptron class.
- The other way of deploying perceptron is to use the general `linear_model.SGDClassifier` with `loss='perceptron'`

See for examples are linearly separable, then that leads to convergence of the perceptron algorithm and 0 training loss. Otherwise, perceptron algorithm oscillates.

(Refer Slide Time: 12:02)



Linearly separable examples lead to convergence of the algorithm with zero training loss, else it oscillates.

Parameters of Perceptron class

- Let's quickly take a look into the important parameters of the Perceptron()

```
class sklearn.linear_model.Perceptron(*, penalty=None, alpha=0.0001, l1_ratio=0.15,
fit_intercept=True, max_iter=1000, tol=0.001, shuffle=True, verbose=0, eta0=1.0, n_jobs=None,
random_state=0, early_stopping=False, validation_fraction=0.1, n_iter_no_change=5, class_weight=None,
warm_start=False).
```

- Need not to pay attention to all the arguments and their default values.
- Internally, the API uses the perceptron loss (i.e. it calls **Hinge(0.0)**, where 0.0 is a threshold) and uses SGD to update the weights.
- You may refer to the documentation for more details on the Perceptron class.
- The other way of deploying perceptron is to use the general `linear_model.SGDClassifier` with `loss='perceptron'`

The above loss is termed as hard Hinge-loss (as scores pass through the sign function) and hence we can't use SGD.

Whereas, Sklearn implements hinge-Loss with the following definition: $\max(0, -wx^T y)$ and by default calls sgd to minimize the loss.

Instantiation

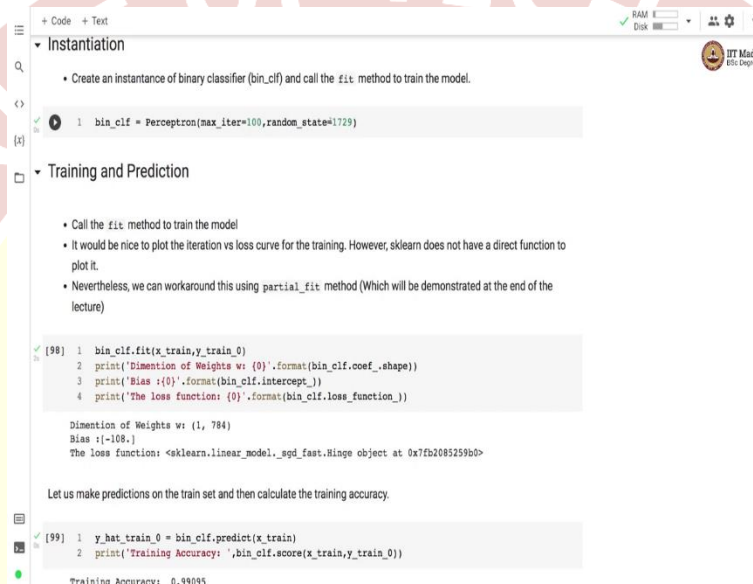
- Create an instance of binary classifier (`bin_clf`) and call the `fit` method to train the model.

```
[97] 1 bin_clf = Perceptron(max_iter=100, random_state=1729)
```

Let us look at the sklearn implementation of perceptron. So, perceptron is implemented in `sklearn.linear_model` module. And it is a perceptron API. It has various it has got various parameters like `penalty`, α , `l1_ratio`, `fit_intercept`, `max_writer`, tolerance `shuffle`, `verbose`, `uta 0`, `N jobs`, `random states`, `early stopping`, `validation traction`, `number of iteration`, `no change`, `+weight`, and `warm start`.

So, you might recall a lot of these parameters from SGD. So, tolerance, eta 0, early stopping, n_iter, no change. These are parameters that we have also seen in SGD classifier as well as SGD regressors. So, internally this particular API calls hinge 0, it uses a hinge loss with 0 as a threshold. And it uses SGD to update the weights. Alternatively, we can also use a SGD classifier with loss = perceptron to build a perceptron model.

(Refer Slide Time: 13:37)



The screenshot shows a Jupyter Notebook interface with a code editor and a console output. The code defines a Perceptron model and trains it on a dataset. The output shows the dimensions of the weights, the bias, and the loss function. The training accuracy is 0.98045.

```
+ Code + Text
Instantiation
• Create an instance of binary classifier (bin_clf) and call the fit method to train the model.

1 bin_clf = Perceptron(max_iter=100, random_state=1729)

Training and Prediction
• Call the fit method to train the model
• It would be nice to plot the iteration vs loss curve for the training. However, sklearn does not have a direct function to plot it.
• Nevertheless, we can workaround this using partial_fit method (Which will be demonstrated at the end of the lecture)

[98] 1 bin_clf.fit(x_train, y_train_0)
      2 print('Dimension of Weights w: {}'.format(bin_clf.coef_.shape))
      3 print('Bias : {}'.format(bin_clf.intercept_))
      4 print('The loss function: {}'.format(bin_clf.loss_function_))

      Dimension of Weights w: (1, 784)
      Bias : [-108.]
      The loss function: <sklearn.linear_model._sgd_fast.Hinge object at 0x7fb2085259b0>

Let us make predictions on the train set and then calculate the training accuracy.

[99] 1 y_hat_train_0 = bin_clf.predict(x_train)
      2 print('Training Accuracy: ', bin_clf.score(x_train, y_train_0))

Training Accuracy: 0.98045
```

So, let us create an instance of a perceptron model. So, we create a perceptron model with maximum iteration set 200 and random state set to 1729. In order to train the perceptron algorithm, we use the fit method and send a feature matrix and a label vector. Since we are solving a 0 detected problem, we have a modified label vector as input to the perceptron.

(Refer Slide Time: 14:13)

```
+ Code + Text
plot it.
• Nevertheless, we can work around this using partial_fit method (Which will be demonstrated at the end of the lecture)

[98] 1 bin_clf.fit(x_train,y_train_0)
2 print('Dimension of Weights w: {}'.format(bin_clf.coef_.shape))
3 print('Bias b: {}'.format(bin_clf.intercept_))
4 print('The loss function: {}'.format(bin_clf.loss_function_))

Dimension of Weights w: (1, 784)
Bias b: [-168.]
The loss function: <sklearn.linear_model._sgd_fast.Hinge object at 0x7fb2085259b0>

Let us make predictions on the train set and then calculate the training accuracy.

[99] 1 y_hat_train_0 = bin_clf.predict(x_train)
2 print('Training Accuracy: ',bin_clf.score(x_train,y_train_0))

Training Accuracy: 0.99095

Let us make the predictions on the test set and then calculate the testing accuracy.

[100] 1 print('Test accuracy: ',bin_clf.score(x_test,y_test_0))

Test accuracy: 0.989










Displaying predictions
• Take few images from the testset at random and display it with the corresponding predictions.
• Plot a few images in a single figure window along with their respective predictions
```

Once the perceptron model is trained, we can we can predict labels and on the training set, we get accuracy of 0.99. Whereas on the test set, we get accuracy of 0.98 which is slightly lower than the training accuracy, but nevertheless they are comparable.

(Refer Slide Time: 14:43)

```
+ Code + Text
Displaying predictions
• Take few images from the testset at random and display it with the corresponding predictions.
• Plot a few images in a single figure window along with their respective predictions

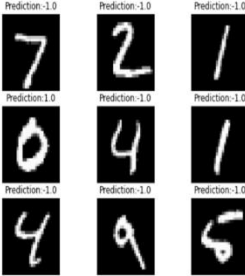
[101] 1 y_hat_test_0 = bin_clf.predict(x_test)
2 num_images = 9 # Choose a square number
3 factor = np.int(np.sqrt(num_images))
4 fig,ax = plt.subplots(nrows=factor,ncols=factor,figsize=(8,6))
5 idx_offset = 0 # display "num_images" starting from idx_offset
6 for i in range(factor):
7     index = idx_offset+i*(factor)
8     for j in range(factor):
9         ax[i,j].imshow(x_test[index+j].reshape(28,28),cmap='gray') # we should not use x_train_with
10        ax[i,j].set_title('Prediction:{}'.format(str(y_hat_test_0[index+j])))
11        ax[i,j].set_axis_off()
```

Prediction-1.0 	Prediction-1.0 	Prediction-1.0 
Prediction-1.0 	Prediction-1.0 	Prediction-1.0 
Prediction-1.0 	Prediction-1.0 	Prediction-1.0 


```

+ Code + Text
1 x_test = np.loadtxt('test_images.txt')
2 fig, ax = plt.subplots(nrows=factor, ncols=factor, figsize=(8,6))
3 idx_offset = 0 # display "num_images" starting from idx_offset
4 for i in range(factor):
5     index = idx_offset+i*(factor)
6     for j in range(factor):
7         ax[i,j].imshow(x_test[index+j].reshape(28,28), cmap='gray') # we should not use x_train_with_
8         ax[i,j].set_title('Prediction:0').format(str(y_hat_test_0[index+j]))
9         ax[i,j].set_axis_off()
10
11
12

```



• Display images of positive classes from testset along with their predictions.

```

[102] 1  indx_0 = np.where(y_test_0 == 1)

```

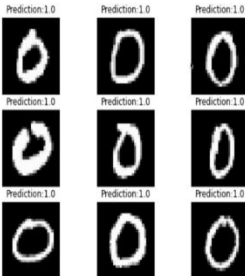
So, what we will do is we will now draw images or plot images and we will also specify what is the prediction for each of the image. So, here you can see that the image is 7 and the prediction is -1 which is the correct prediction. This image of 0 and prediction is 1 which is also a correct prediction. And you can see that most of the images in this display are correctly predicted.

(Refer Slide Time: 15:26)

```

+ Code + Text
1 fig, ax = plt.subplots(nrows=factor, ncols=factor, figsize=(8,6))
2 idx_offset = 0 # display "num_images" starting from idx_offset
3 for i in range(factor):
4     index = idx_offset+i*(factor)
5     for j in range(factor):
6         ax[i,j].imshow(zeros_imgs[index+j].reshape(28,28), cmap='gray') # we should not use x_train_with_
7         ax[i,j].set_title('Prediction:0').format(str(zeroLabels[index+j]))
8         ax[i,j].set_axis_off()
9
10
11

```



It seems that there are a significant number of images that are correctly classified. Let's see how many?

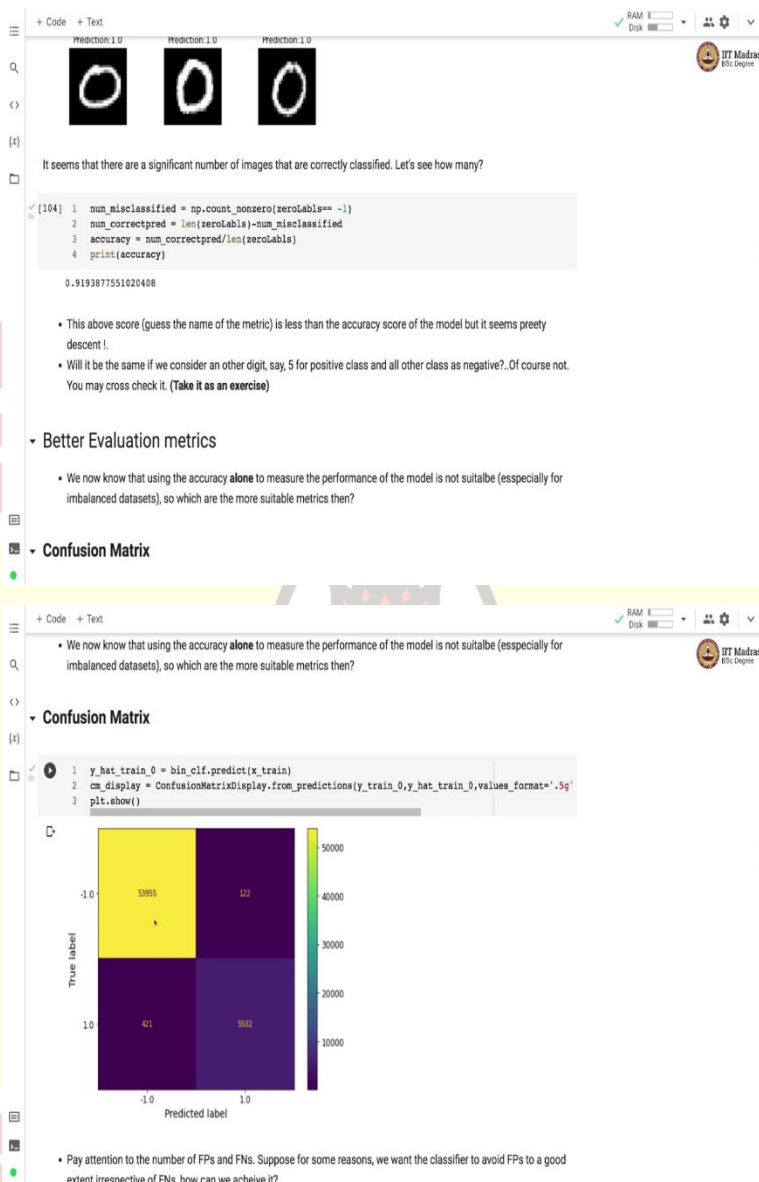
```

[104] 1  num_misclassified = np.count_nonzero(zeroLabels == -1)
2     num_correctpred = len(zeroLabels)-num_misclassified
3     accuracy = num_correctpred/len(zeroLabels)
4     print(accuracy)

```

So, you can see that various types of 0's are correctly predicted or are correctly predicted by the perceptron algorithm.

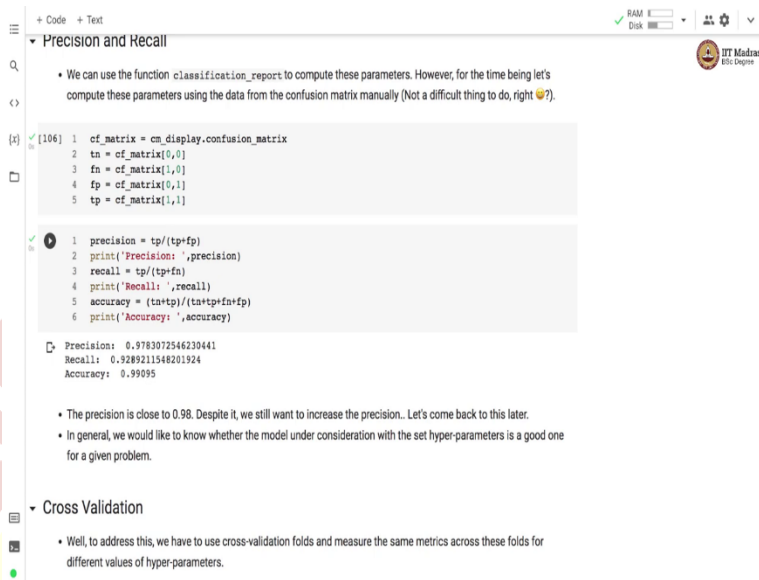
(Refer Slide Time: 15:40)



So, there are about 91 percent images are correctly classified. So, let us try to predict or let us try to plot confusion matrix of based on the prediction. So, here we are plotting a confusion matrix on the training set. And here you can see that 53,955 examples are correctly classified as -1 and 5502 examples are correctly classified to be in class 1. Whereas, the off diagonal entries signify the digits that are incorrectly classified.

These are the digits which were actually -one, but they are predicted to be +1 and 421 digits, which are actually +1 are predicted to be -1.

(Refer Slide Time: 16:54)



```
+ Code + Text
Precision and Recall

• We can use the function classification_report to compute these parameters. However, for the time being let's compute these parameters using the data from the confusion matrix manually (Not a difficult thing to do, right 😊?).

In [106]: 1 cf_matrix = cm_display.confusion_matrix
          2 tn = cf_matrix[0,0]
          3 fn = cf_matrix[1,0]
          4 fp = cf_matrix[0,1]
          5 tp = cf_matrix[1,1]

1 precision = tp/(tp+fp)
2 print('Precision: ',precision)
3 recall = tp/(tp+fn)
4 print('Recall: ',recall)
5 accuracy = (tn+tp)/(tn+tp+fn+fp)
6 print('Accuracy: ',accuracy)

Precision: 0.9783072546230441
Recall: 0.9289211548201924
Accuracy: 0.99095

• The precision is close to 0.98. Despite it, we still want to increase the precision.. Let's come back to this later.
• In general, we would like to know whether the model under consideration with the set hyper-parameters is a good one for a given problem.

Cross Validation

• Well, to address this, we have to use cross-validation folds and measure the same metrics across these folds for different values of hyper-parameters.
```

Let us calculate precision and recall from the confusion matrix. And here you can see that we have precision of 0.97 and recall of 0.92.

(Refer Slide Time: 17:14)



```
+ Code + Text

• The precision is close to 0.98. Despite it, we still want to increase the precision.. Let's come back to this later.
• In general, we would like to know whether the model under consideration with the set hyper-parameters is a good one for a given problem.

Cross Validation

• Well, to address this, we have to use cross-validation folds and measure the same metrics across these folds for different values of hyper-parameters.
• However, perceptron does not many hyperparameters other than the learning rate.
• For the moment, we set the learning rate to its default value. Later, we use GridSearchCV to find the better value for the learning rate.

In [ ]: 1 bin_clf = Perceptron(max_iter=100, random_state=1729) # repeating for readability
        2 scores= cross_validate(bin_clf, x_train, y_train, cv=5,
        3                          scoring=['precision', 'recall', 'f1'],
        4                          return_estimator=True)
        5 pprint(scores)

{'estimator': (Perceptron(max_iter=100, random_state=1729),
               Perceptron(max_iter=100, random_state=1729),
               Perceptron(max_iter=100, random_state=1729),
               Perceptron(max_iter=100, random_state=1729),
               Perceptron(max_iter=100, random_state=1729)),
 'fit_time': array([1.17687774, 2.34696865, 1.67342091, 1.53141808, 1.92304611]),
 'score_time': array([0.04768467, 0.04581189, 0.04647231, 0.04626894, 0.06043673]),
 'test_f1': array([0.95238095, 0.91666667, 0.94915254, 0.94117647, 0.95991763]),
 'test_precision': array([0.95890411, 0.98828125, 0.95319149, 0.95090439, 0.96200345]),
 'test_recall': array([0.94594595, 0.85472973, 0.94514768, 0.93164557, 0.94008439])

• Note:
  The perceptron estimator passed as an argument to the function cross_validate is internally cloned num_fold
```

In order to get more robust prediction we will train the perceptron classifier in cross validated manner. So, we make use of `cross_validate` function supply the perceptron estimator along with the feature matrix and the label vector we specify number of folds for cross validation = 5.

(Refer Slide Time: 17:45)

```
+ Code + Text
• However, perceptron does not many hyperparameters other than the learning rate.
• For the moment, we set the learning rate to its default value. Later, we use GridSearchCV to find the better value for the learning rate.

[29] bin_clf = Perceptron(max_iter=100, random_state=1729) # repeating for readability
scores = cross_validate(bin_clf, x_train, y_train, cv=5,
                        scoring= ['precision', 'recall', 'f1'],
                        return_estimator=True)
pprint(scores)

{'estimator': (Perceptron(max_iter=100, random_state=1729),
               Perceptron(max_iter=100, random_state=1729),
               Perceptron(max_iter=100, random_state=1729),
               Perceptron(max_iter=100, random_state=1729),
               Perceptron(max_iter=100, random_state=1729)),
 'fit_time': array([1.17687774, 2.34696865, 1.67342091, 1.53141809, 1.92304611]),
 'score_time': array([0.04768467, 0.04581189, 0.04647231, 0.04626894, 0.06043673]),
 'test_f1': array([0.95238095, 0.91666667, 0.94915254, 0.94117647, 0.95091763]),
 'test_precision': array([0.95890411, 0.98828125, 0.95319149, 0.95090439, 0.96200345]),
 'test_recall': array([0.94594595, 0.85472973, 0.94514768, 0.93164557, 0.94008439])

• Note:
The perceptron estimator passed as an argument to the function cross_validate is internally cloned num_fold
(cv=5) times and fitted independently on each fold. (you can check this by setting warm_start=True)
• Compute the average and standard deviation of scores for all three metrics on (k=5) folds to measure the
generalization!

[30] print('f1',      avg:(0.2f), std:(1.1f)).format(scores['test_f1'].mean(), scores['test_f1'].std())
      print('precision, avg:(0.2f), std:(1.1f)).format(scores['test_precision'].mean(), scores['test_precision'].std())
      print('recall,  avg:(0.2f), std:(1.1f)).format(scores['test_recall'].mean(), scores['test_recall'].std())
```

And once it was trained through cross validation, you can see that we got 5 different perceptron algorithms or 5 different perceptron instances and each one of them results into slightly different precision and recall on the test sets. And we can also look at the f1 score which is combination of precision and recall. So, the first perceptron model seems to be getting us the best f1 score or 0.95.

(Refer Slide Time: 18:28)

```
+ Code + Text
{'estimator': (Perceptron(max_iter=100, random_state=1729),
               Perceptron(max_iter=100, random_state=1729),
               Perceptron(max_iter=100, random_state=1729),
               Perceptron(max_iter=100, random_state=1729),
               Perceptron(max_iter=100, random_state=1729)),
 'fit_time': array([1.17687774, 2.34696865, 1.67342091, 1.53141809, 1.92304611]),
 'score_time': array([0.04768467, 0.04581189, 0.04647231, 0.04626894, 0.06043673]),
 'test_f1': array([0.95238095, 0.91666667, 0.94915254, 0.94117647, 0.95091763]),
 'test_precision': array([0.95890411, 0.98828125, 0.95319149, 0.95090439, 0.96200345]),
 'test_recall': array([0.94594595, 0.85472973, 0.94514768, 0.93164557, 0.94008439])

• Note:
The perceptron estimator passed as an argument to the function cross_validate is internally cloned num_fold
(cv=5) times and fitted independently on each fold. (you can check this by setting warm_start=True)
• Compute the average and standard deviation of scores for all three metrics on (k=5) folds to measure the
generalization!

[30] print('f1',      avg:(0.2f), std:(1.1f)).format(scores['test_f1'].mean(), scores['test_f1'].std())
      print('precision, avg:(0.2f), std:(1.1f)).format(scores['test_precision'].mean(), scores['test_precision'].std())
      print('recall,  avg:(0.2f), std:(1.1f)).format(scores['test_recall'].mean(), scores['test_recall'].std())

f1,      avg:0.94, std:0.013
precision, avg:0.96, std:0.01
recall,  avg:0.92, std:0.03

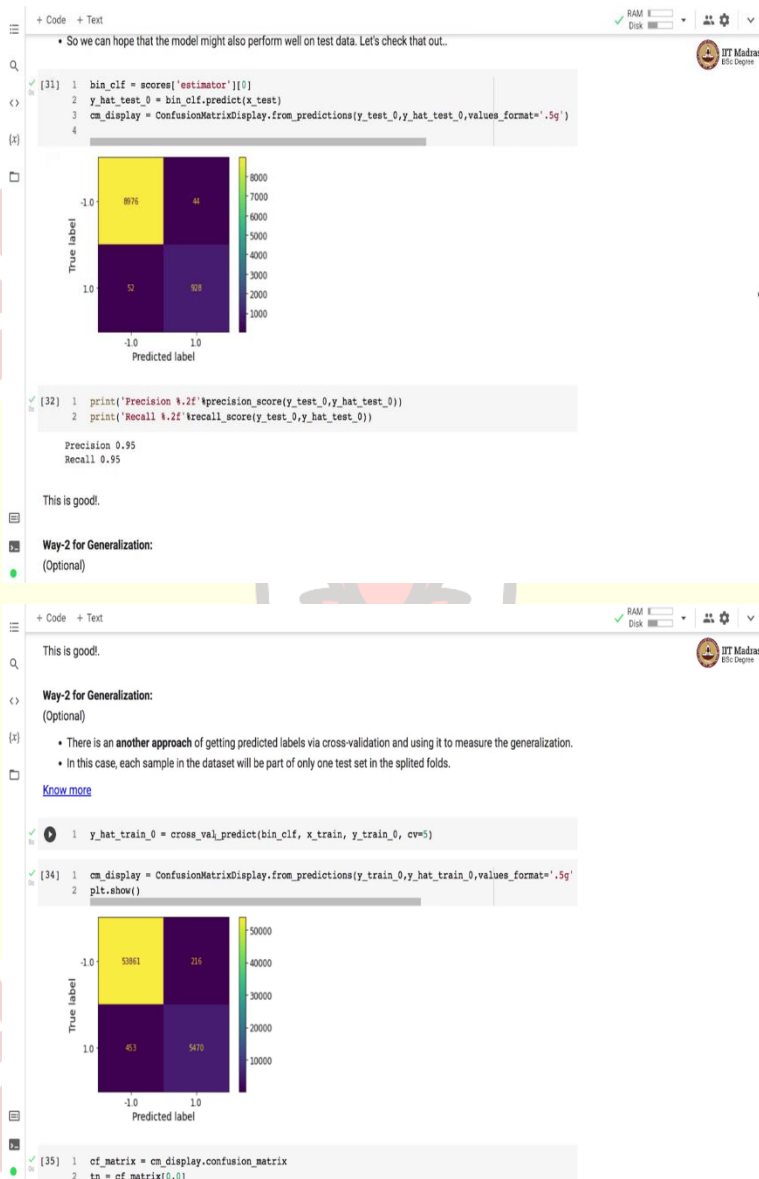
• Let us pick the first estimator returned by the cross-validate function.
• So we can hope that the model might also perform well on test data. Let's check that out.

[31] bin_clf = scores['estimator'][0]
      y_hat_test_0 = bin_clf.predict(x_test)
      cm = ConfusionMatrixDisolv.from_predictions(y_hat_test_0, y_test_0, values_format='%s')
```

So, we can also compute the average and the standard deviation for each of these metrics. So, that helps us to understand how what is the variation in the performance of different perceptron models that we trained through cross validation. So, you can see that the standard deviation is pretty small

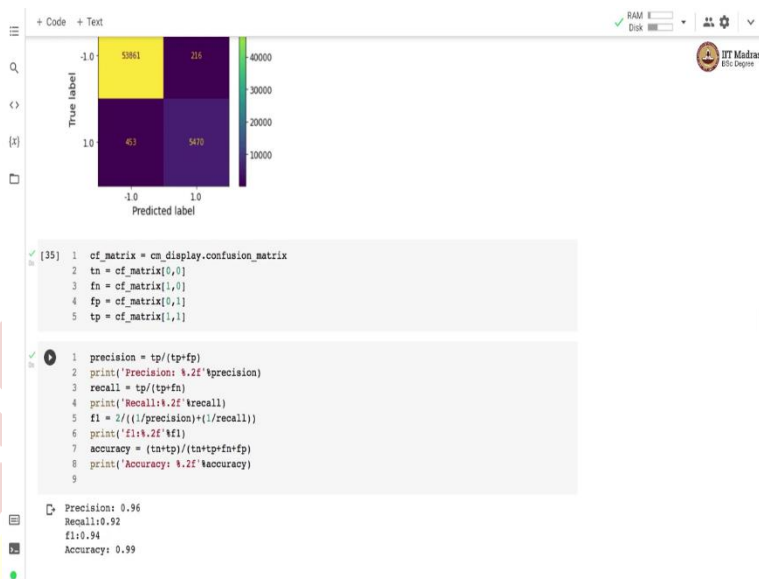
and most of the perceptron algorithm have more or less very similar performance. So, we got we get the average precision of 0.96 recall a 0.92 an average f1 score of 0.94.

(Refer Slide Time: 19:20)



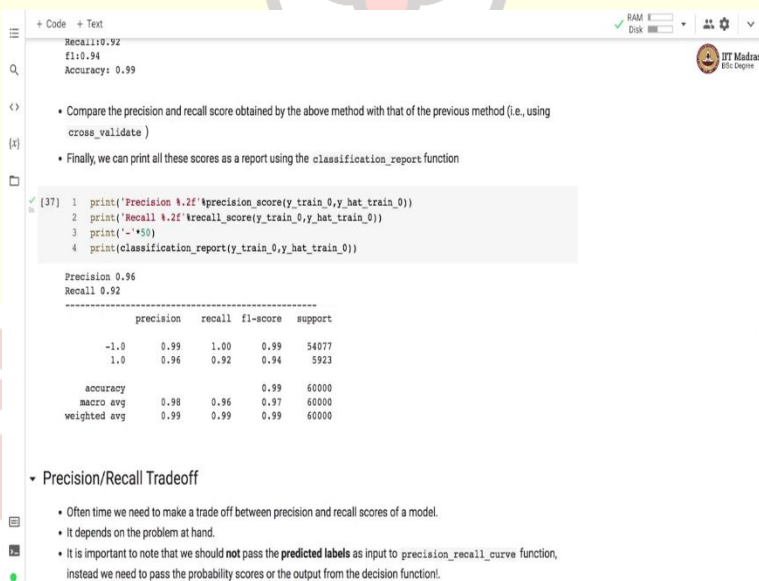
As you can see that the first estimator has got the best f1 score and we use that estimator to build the confusion matrix we can make use of `cross_val_predict` function in order to get a prediction for each example when it was part of the test set. And we with that we got a confusion matrix on the full set.

(Refer Slide Time: 20:01)



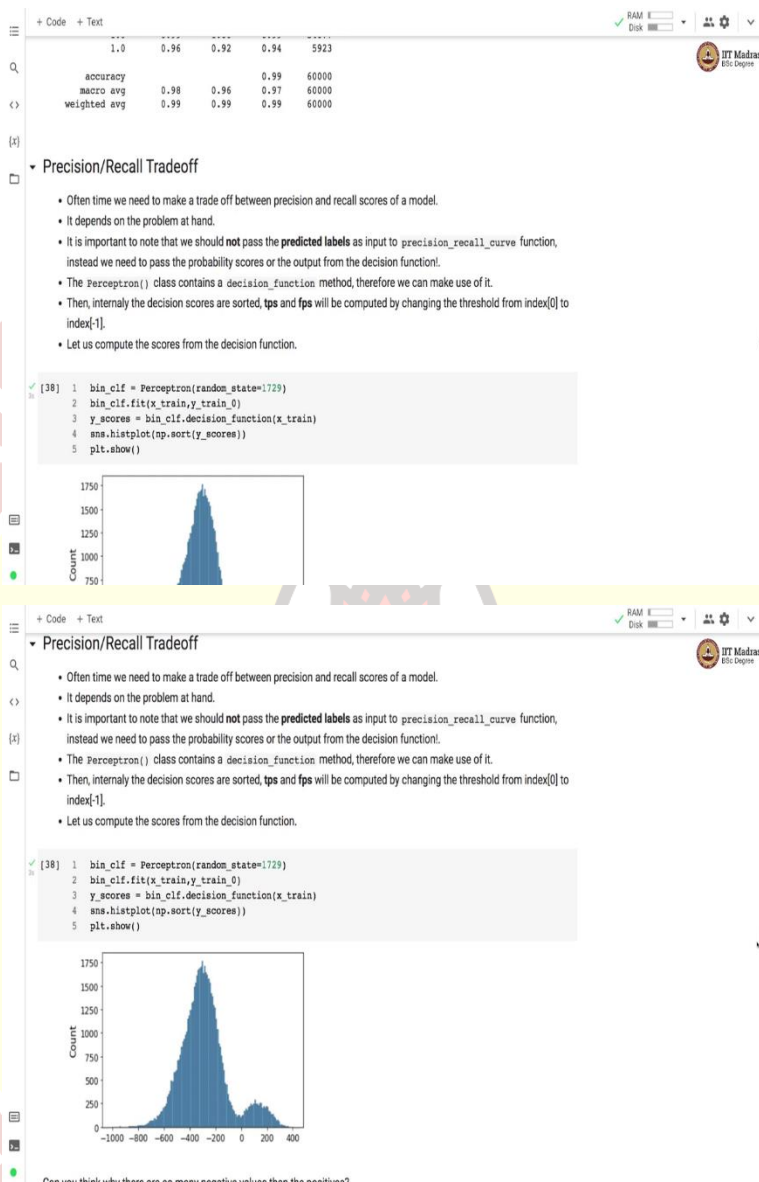
Here you can see that the precision we achieved precision of 0.96 recall or 0.92 and overall f1 score of 0.94.

(Refer Slide Time: 20:16)



So, alternatively, we can also use the `classification_report` function in order to get precision recall and f1 score for both the classes we also get other metrics like accuracy and macro average and weighted average accuracy based on the number of examples in each class. So, you can see that for the negative class precision is 0.99 and recall is 1 whereas, for the positive class the precision is 0.96 and recall is 0.92.

(Refer Slide Time: 21:01)

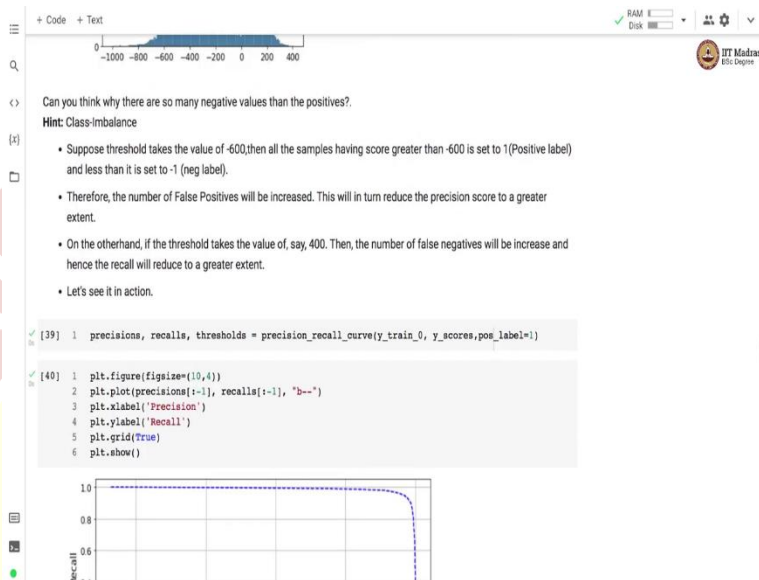


So, let us try to look at the tradeoff between precision and recall. So, if we increase precision, there is a drop in recall and vice versa. So, here in order to study the tradeoff between precision and recall, instead of getting the prediction, we are getting the values from the decision function. So, these are the values of the linear combination before applying a nonlinear decision function on it.

So, we get values which are real, which are real numbers. And you can see that there are a lot more negative examples in our training data set. Hence, there are a lot more negative values

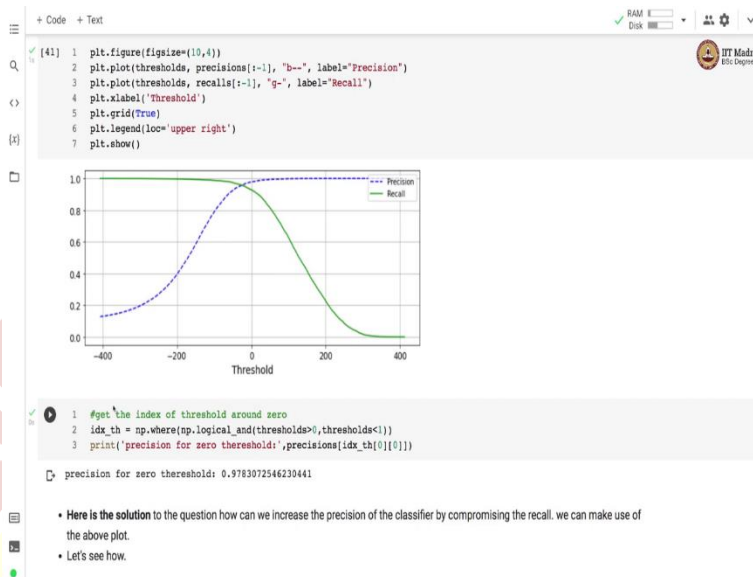
that we obtained through decision function and small number of positive values. And those mostly corresponds to the positive class.

(Refer Slide Time: 22:01)



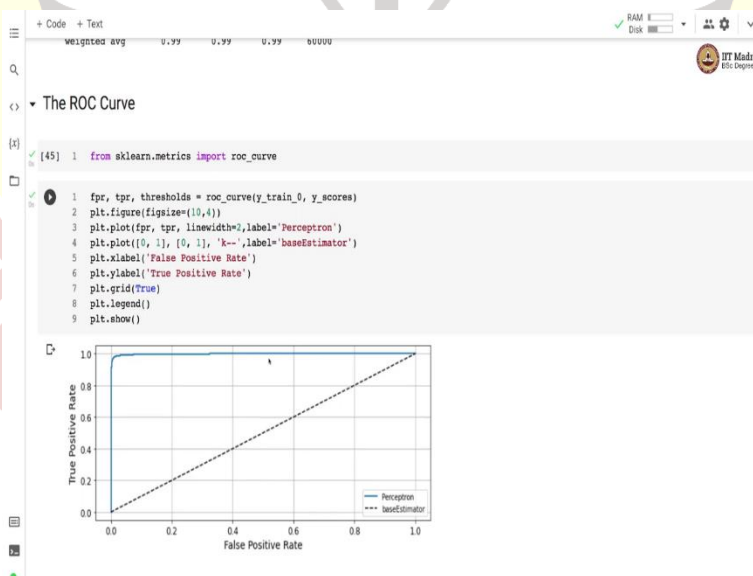
Now, what we will do is we will vary the threshold we will vary the decision function threshold and for each threshold, we will calculate precision and recall and then we then we will plot that precision recall in what is called as PR curve. So, we have precision_recall_curve function in sklearn metric that helps us to get the PR curve. But the way that is done is by changing the threshold and for every threshold we calculate precision recall values.

(Refer Slide Time: 22:38)



You can see the relationship between precision recall more clearly in this particular graph. So, you can see that as the threshold goes up precision tends to increase and recall tends to decrease. So, there is an inverse relationship or there is a tradeoff between precision and recall. So, if you use a zero threshold, we have a precision of 0.978.

(Refer Slide Time: 23:10)

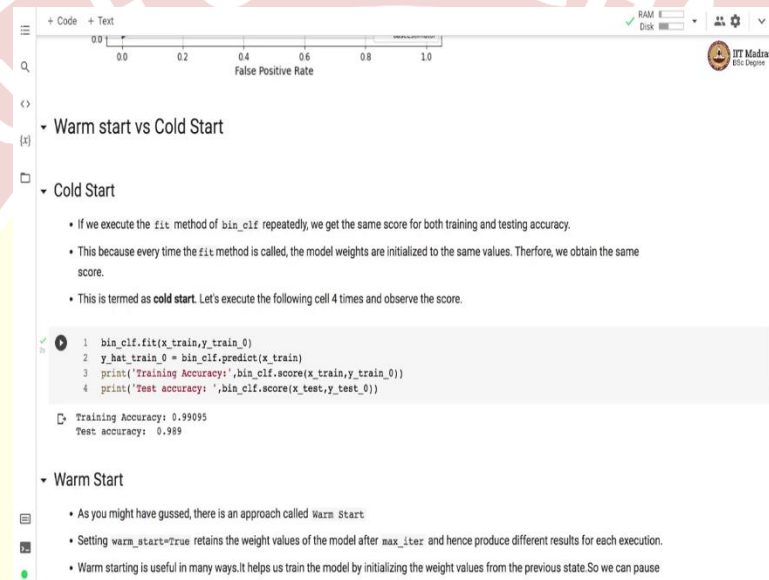


Alternatively, we can also plot what is called as our ROC curve, that has got false positive rate on x axis and true positive rate on the y axis. So, the process is exactly the same, we have different thresholds on the decision function and for each threshold we calculate false positive rate and true

positive rate and then plot them in our ROC curve. So, ideal ROC curve should be what you are seeing on your screen.

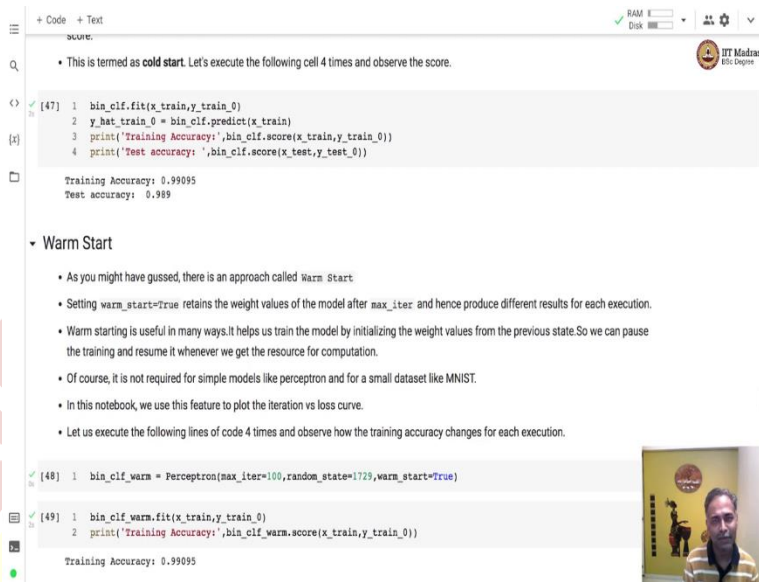
So, this is an ideal ROC curve because we are getting very good accuracy numbers on both the classes. And we calculate quantity like area under the ROC curve as major of how good is ROC curve.

(Refer Slide Time: 24:02)



Finally, let us look at warm start and cold start. So, whenever we call the fit method on the perceptron estimator, it basically start afresh it initializes the parameter to some values and start the training from scratch.

(Refer Slide Time: 24:34)



The screenshot shows a Jupyter Notebook interface. At the top, there's a tab labeled '+ Code + Text'. Below it, a search bar and a RAM/Disk usage indicator. The main content area has a text block starting with 'This is termed as cold start. Let's execute the following cell 4 times and observe the score.' followed by a code cell [47] with the following code:

```
1 bin_clf.fit(x_train,y_train_0)
2 y_hat_train_0 = bin_clf.predict(x_train)
3 print('Training Accuracy:',bin_clf.score(x_train,y_train_0))
4 print('Test accuracy: ',bin_clf.score(x_test,y_test_0))
```

The output of this cell shows 'Training Accuracy: 0.99095' and 'Test accuracy: 0.989'. Below this, a section titled 'Warm Start' contains a list of bullet points explaining the concept and its benefits. The code cell [48] shows the initialization of a perceptron with warm start:

```
[48] 1 bin_clf_warm = Perceptron(max_iter=100,random_state=1729,warm_start=True)
```

Below this, code cell [49] shows the training and prediction:

```
[49] 1 bin_clf_warm.fit(x_train,y_train_0)
2 print('Training Accuracy:',bin_clf_warm.score(x_train,y_train_0))
```

The output of this cell shows 'Training Accuracy: 0.99095'. A small video thumbnail of a man is visible on the right side of the notebook interface.

So, if we use warm start in state by setting the warm start = true in the constructor of the perceptron object, what happens is whenever we set warm start = true, the previous the weights obtained from the previous run of perceptron are used for to initialize the new run. So, every new run is in every new run the weights are initialized to the weights obtained in the previous run. So, that helps us in speeding up the training.

And the warm start is also useful if we are doing training in batches, where we do a training on a batch, we store the weights and next time when we got another batch or when we got resources available for training we can start perceptron training from where we had left it. So, warm start will be very useful utility when we are training on large scale data. So in this video, we studied how to use perceptron classifier to detect a 0 from the images of digits. In the next video, we will try to solve a multi class classification problem with perceptron.