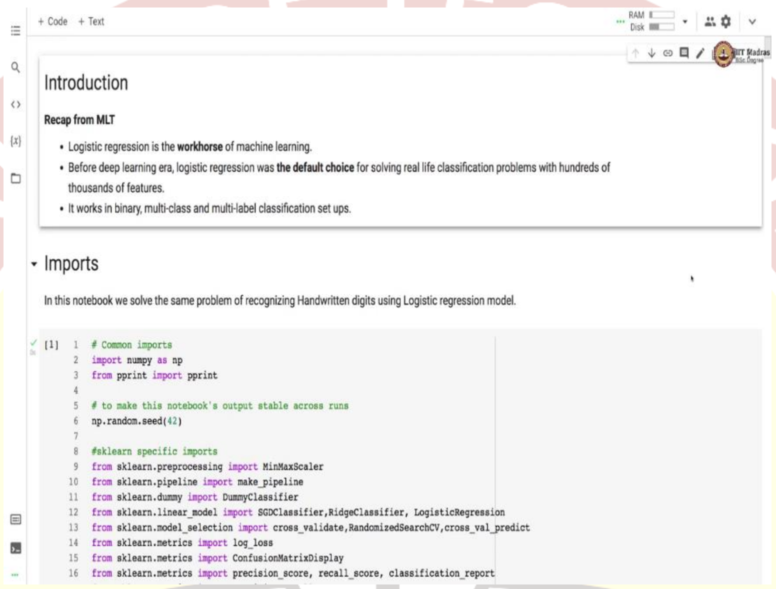
INDIAN INSTITUTE OF TECHNOLOGY MADRAS · सिद्धिर्भवति कर्मजा ·

IIT Madras
ONLINE DEGREE

Machine Learning Techniques
Professor Doctor Ashish Tendulkar
Indian Institute of Technology Madras
Demonstration: MNIST Digits Classification using SGDRegressor

(Refer Slide Time: 00:09)



Introduction

Recap from MLT

- Logistic regression is the **workhorse** of machine learning.
- Before deep learning era, logistic regression was **the default choice** for solving real life classification problems with hundreds of thousands of features.
- It works in binary, multi-class and multi-label classification set ups.

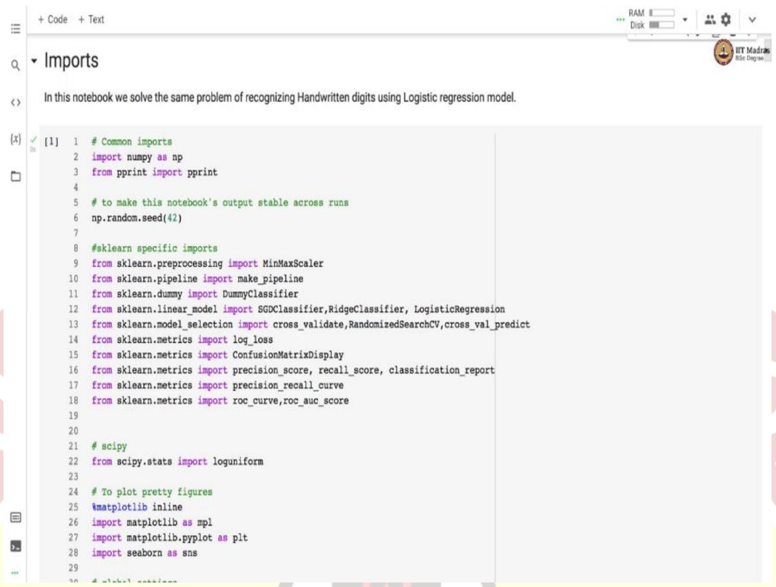
Imports

In this notebook we solve the same problem of recognizing Handwritten digits using Logistic regression model.

```
[1]
# Common imports
import numpy as np
from pprint import pprint

# to make this notebook's output stable across runs
np.random.seed(42)

#sklearn specific imports
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import make_pipeline
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import SGDClassifier,RidgeClassifier, LogisticRegression
from sklearn.model_selection import cross_validate,RandomizedSearchCV,cross_val_predict
from sklearn.metrics import log_loss
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import precision_score, recall_score, classification_report
```

Namaste! Welcome to the next video of Machine Learning Practice course. In this video, we will implement MNIST digit classification with logistic regression. Logistic regression is the workhorse of machine learning. Before deep learning era, logistic regression was the default choice for solving real life classification problems with hundreds of thousands of features. It works in binary, multi-class and multi-label classification setup.

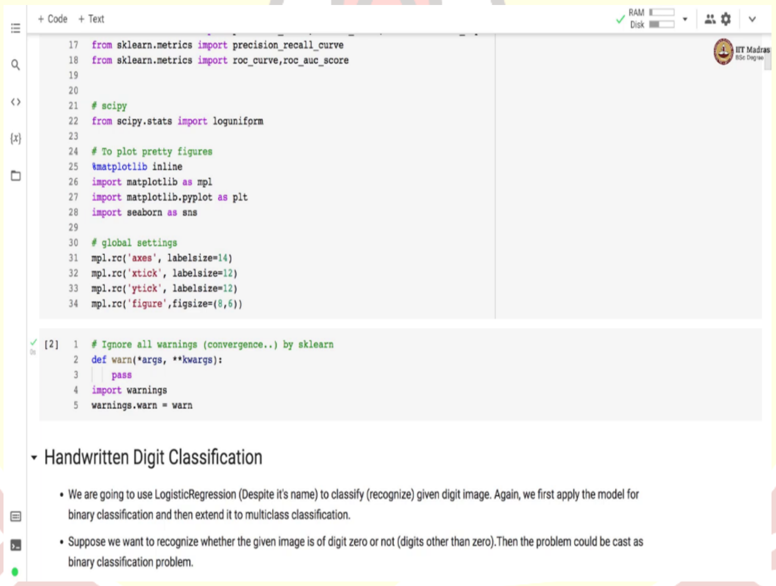
(Refer Slide Time: 00:48)



Imports

In this notebook we solve the same problem of recognizing Handwritten digits using Logistic regression model.

```
[1]
# Common imports
import numpy as np
from pprint import pprint

# to make this notebook's output stable across runs
np.random.seed(42)

#sklearn specific imports
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import make_pipeline
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import SGDClassifier,RidgeClassifier, LogisticRegression
from sklearn.model_selection import cross_validate,RandomizedSearchCV,cross_val_predict
from sklearn.metrics import log_loss
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import precision_score, recall_score, classification_report
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import roc_curve,roc_auc_score


# scipy
from scipy.stats import loguniform

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns

```



```
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import roc_curve,roc_auc_score


# scipy
from scipy.stats import loguniform

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns

# global settings
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
mpl.rc('figure',figsize=(8,6))
```

```
[2]
# Ignore all warnings (convergence..) by sklearn
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
```

Handwritten Digit Classification

- We are going to use LogisticRegression (Despite it's name) to classify (recognize) given digit image. Again, we first apply the model for binary classification and then extend it to multiclass classification.
- Suppose we want to recognize whether the given image is of digit zero or not (digits other than zero).Then the problem could be cast as binary classification problem.

So, in the last week, we solved the problem of recognizing handwritten digits using perceptron classifier. In this notebook, we will solve the same problem using logistic regression and least square classification. So, we will begin by importing commonly used imports like numpy and pprint. Then we set the random seed in order to make the notebook's output stable across run.

Then there are sklearn specific inputs, there are, there is a preprocessing API, which is MinMaxScaler, then pipeline API, make_ pipeline, then dummy classifier for building a baseline classifier, and then there are classifiers, like SGDClassifier, RidgeClassifier and logistic regression classifier. So, we will be implementing logistic regression with SGDClassifier and logistic regression classifier.

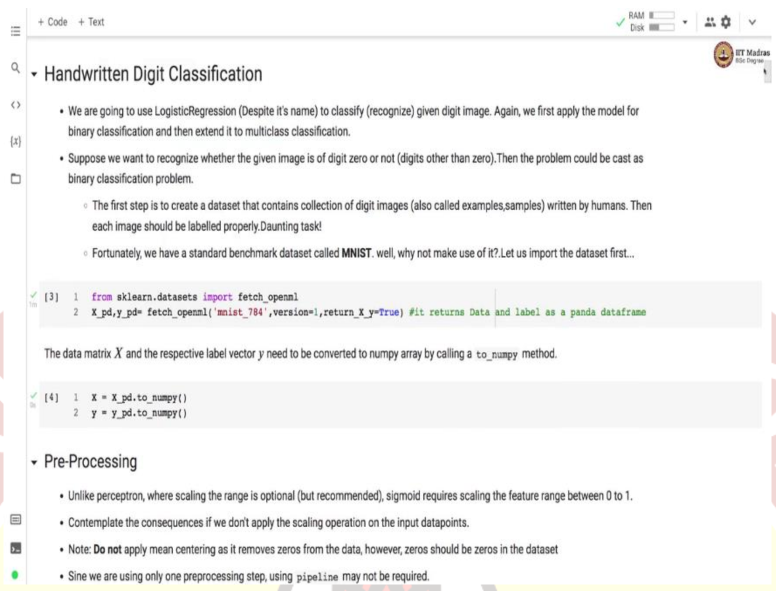
There are two ways to implement logistic regression in sklearn. And then we will implement the least square classifier with ridge classifier. Then we have bunch of APIs for model selection which we will be using, one is cross validate, then RandomizedSearchCV and cross _val _predict. So, we normally trained our model with cross validate in order to make robust model training across different training sets, and we evaluate it across different test sets.

So, when we train model with cross validate, we are generally making our training robust. Then RandomizedSearchCV is used for hyper-parameter tuning and searching for the best hyper-parameters in the given amount of budget. And cross _val _predict is used for predicting the labels, for examples, in different test sets in the cross-validation manner. Then there are a bunch of metrics that we use, log loss, which is the loss that we calculate for logistic regression.

So, log loss is also known as cross-entropy loss. Then there is a confusion matrix display that displays the confusion matrix. Then there is a precision, recall and classification report, which help us to get vital statistics about the evaluation, about the performance of the classifier. Then we have precision recall curve, ROC curve and ROC AUC curves in order to get some more metrics about the performance of the trained classifiers.

So, in RandomizedSearchCV we need to specify distributions and for specifying the log uniform distribution, we import it from scipy.stats. Then there are packages like matplotlib and seaborn that we import to plot pretty figures. Then there are some global settings about axes, ticks on the axes and the figure.

(Refer Slide Time: 04:33)



Handwritten Digit Classification

- We are going to use LogisticRegression (Despite it's name) to classify (recognize) given digit image. Again, we first apply the model for binary classification and then extend it to multiclass classification.
- Suppose we want to recognize whether the given image is of digit zero or not (digits other than zero).Then the problem could be cast as binary classification problem.
  - The first step is to create a dataset that contains collection of digit images (also called examples,samples) written by humans. Then each image should be labelled properly.Daunting task!
  - Fortunately, we have a standard benchmark dataset called **MNIST**. well, why not make use of it?.Let us import the dataset first...

```
[3]
from sklearn.datasets import fetch_openml
X_pd,y_pd= fetch_openml('mnist_784',version=1,return_X_y=True) #it returns Data and label as a panda dataframe
```

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

```
[4]
X = X_pd.to_numpy()
y = y_pd.to_numpy()
```

Pre-Processing

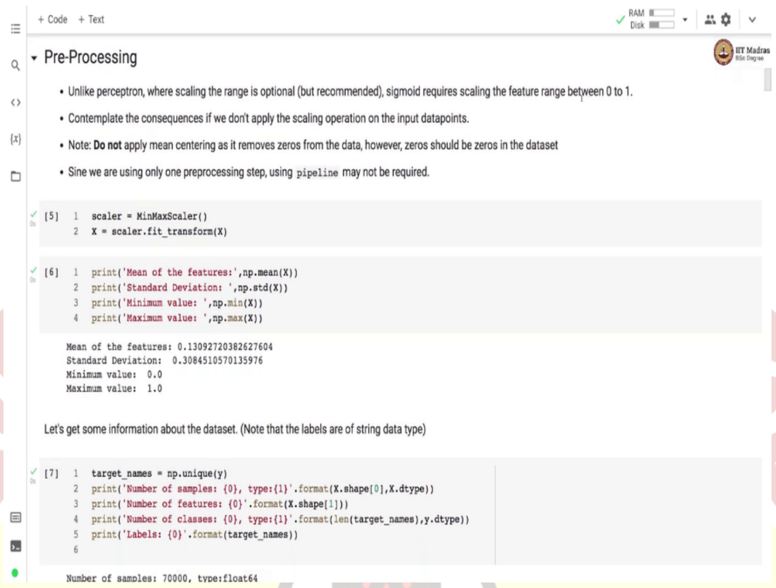
- Unlike perceptron, where scaling the range is optional (but recommended), sigmoid requires scaling the feature range between 0 to 1.
- Contemplate the consequences if we don't apply the scaling operation on the input datapoints.
- Note: **Do not** apply mean centering as it removes zeros from the data, however, zeros should be zeros in the dataset
- Sine we are using only one preprocessing step, using `pipeline` may not be required.

So, you already know about MNIST dataset. MNIST dataset has got 10 different classes. And what we have in MNIST dataset is an image of a handwritten digit and we are supposed to predict or recognize the handwritten digit. So, there are 10 different classes 0 to 9. And we will be given an image and we need to learn a classifier to classify the image into the given digit. So, MNIST handwritten digit recognition is a, is an instance of multi-class classification.

So, what we will do is, to begin with, we will convert this problem into a binary classification problem. So, where what we will do is we will try to recognize, let us say, whether an image is of digit 0 or not. So, image will either get a label of 1, if it is the image of digit 0. And it will get a label of 0 if it is not the digit 0. So, that is how we make a binary classification problem. And we will first demonstrate the usage of logistic regression in the binary classification.

So, as the first step we will download the MNIST dataset. And we download that data with fetch _openml API. We specify the name of the dataset, which is MNIST _784. And we are setting return _x _y equal to true, which returns the feature matrix and label as panda's dataframe. Now, we are converting the feature matrix and label to numpy.

(Refer Slide Time: 06:53)



+ Code + Text

Pre-Processing

- Unlike perceptron, where scaling the range is optional (but recommended), sigmoid requires scaling the feature range between 0 to 1.
- Contemplate the consequences if we don't apply the scaling operation on the input datapoints.
- Note: **Do not** apply mean centering as it removes zeros from the data, however, zeros should be zeros in the dataset
- Sine we are using only one preprocessing step, using `pipeline` may not be required.

```
[5]
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
```

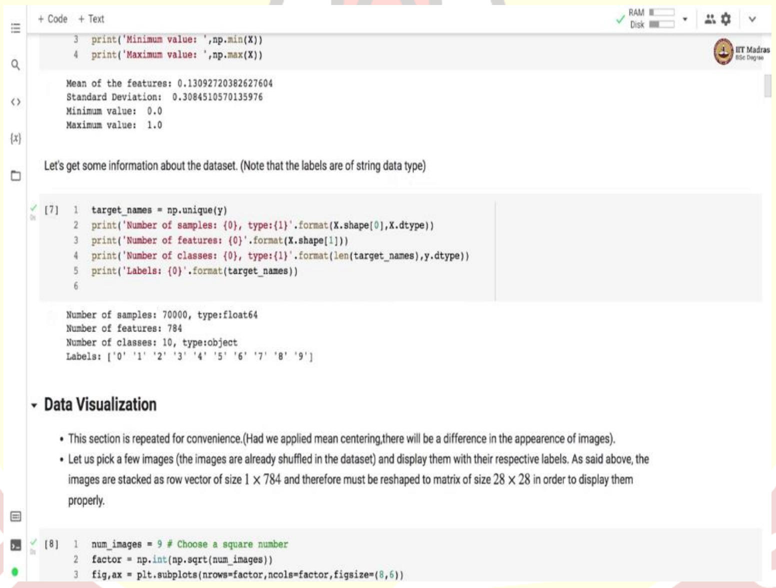
```
[6]
print('Mean of the features:',np.mean(X))
print('Standard Deviation: ',np.std(X))
print('Minimum value: ',np.min(X))
print('Maximum value: ',np.max(X))
```

```
Mean of the features: 0.13092720382627604
Standard Deviation:  0.3084510570135976
Minimum value:  0.0
Maximum value:  1.0
```

Let's get some information about the dataset. (Note that the labels are of string data type)

```
[7]
target_names = np.unique(y)
print('Number of samples: {0}, type:{1}'.format(X.shape[0],X.dtype))
print('Number of features: {0}'.format(X.shape[1]))
print('Number of classes: {0}, type:{1}'.format(len(target_names),y.dtype))
print('Labels: {0}'.format(target_names))

```

```
Number of samples: 70000, type:float64
```



+ Code + Text

```
print('Minimum value: ',np.min(X))
print('Maximum value: ',np.max(X))
```

```
Mean of the features: 0.13092720382627604
Standard Deviation:  0.3084510570135976
Minimum value:  0.0
Maximum value:  1.0
```

Let's get some information about the dataset. (Note that the labels are of string data type)

```
[7]
target_names = np.unique(y)
print('Number of samples: {0}, type:{1}'.format(X.shape[0],X.dtype))
print('Number of features: {0}'.format(X.shape[1]))
print('Number of classes: {0}, type:{1}'.format(len(target_names),y.dtype))
print('Labels: {0}'.format(target_names))

```

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

Data Visualization

- This section is repeated for convenience.(Had we applied mean centering,there will be a difference in the appearence of images).
- 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 row vector of size 1×784 and therefore must be reshaped to matrix of size 28×28 in order to display them properly.

```
[8]
num_images = 9 # Choose a square number
factor = np.int(np.sqrt(num_images))
fig,ax = plt.subplots(nrows=factor,ncols=factor,figsize=(8,6))
```

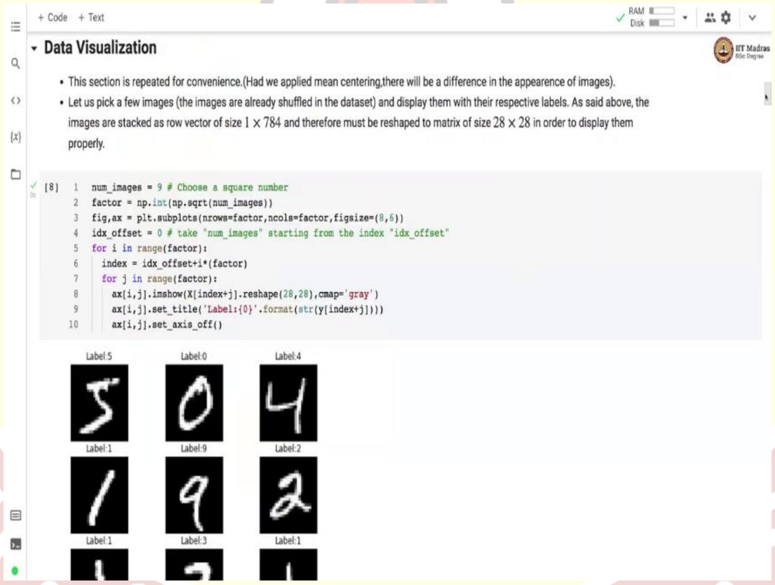
So, we have obtained the data which is the first step in any machine learning pipeline, and the next step is to preprocess the data. So, in, so what we, the input for us is an image and there are the, so each image is 28 by 28. So, there are 784 numbers corresponding to each image and each number denotes the pixel intensity at that particular location in the image.

So, the value in each feature is between 0 to 255. So, as a preprocessing step, we are going to scale the range of the feature between 0 and 1. And the scaling is important especially when we are using SGDclassifier which uses stochastic gradient descent in order to learn the parameters of the model.

So, there is a note of caution that do not apply mean centering in images as it removes 0s. So, here in this particular dataset we use scaling, feature scaling, which is the MinMaxScaler. So, we instantiate a MinMaxScaler object and we call fit _transform method on the feature matrix. So, now, you can see that the minimum value of the feature is 0 and maximum value is 1.

So, we have effectively converted each feature in the range between 0 to 1. So, let us get some information about the dataset. Note that the labels are basically strings. So, we have 70,000 samples in the dataset. Each sample has 784 features. And there are 10 different classes in all. And the labels are 0, 1, 2 all the way up to 9. So, these are 10 digits that are used as labels in this dataset.

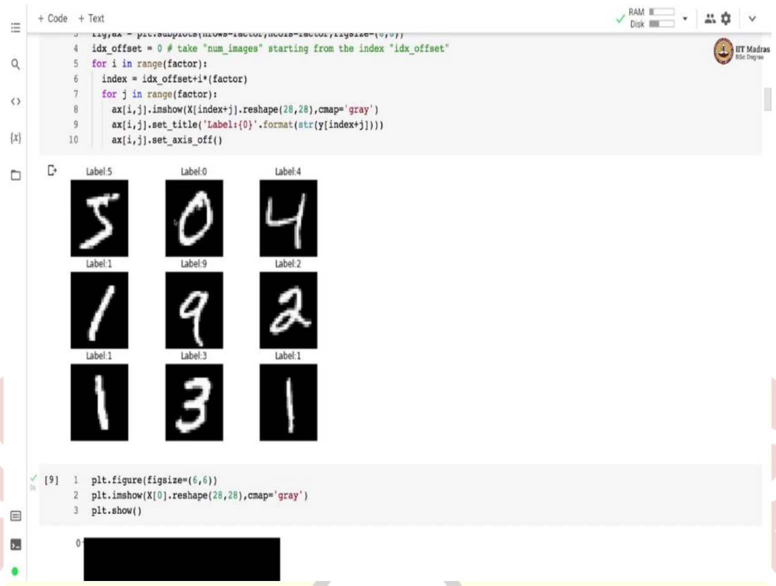
(Refer Slide Time: 09:27)



Data Visualization

- This section is repeated for convenience.(Had we applied mean centering,there will be a difference in the appearence of images).
- 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 row vector of size 1×784 and therefore must be reshaped to matrix of size 28×28 in order to display them properly.

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

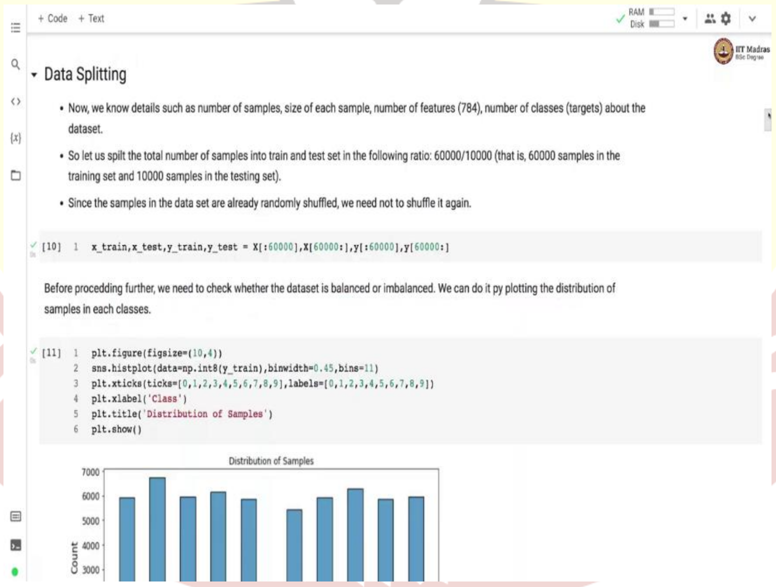


```
idx_offset = 0 # take "num_images" starting from the index "idx_offset"
for i in range(factor):
  index = idx_offset+i*(factor)
  for j in range(factor):
    ax[i,j].imshow(X[index+j].reshape(28,28),cmap='gray')
    ax[i,j].set_title('Label:{0}'.format(str(y[index+j])))
    ax[i,j].set_axis_off()
```

```
[9]
plt.figure(figsize=(6,6))
plt.imshow(X[0].reshape(28,28),cmap='gray')
plt.show()
```

Let us visualize few of the input examples. So, these are the images that are part of the input data. Each image is 28 by 28. So, there are total 784 pixels. And this particular image is of digit 5, this is image of digit 0, and so on.

(Refer Slide Time: 09:59)



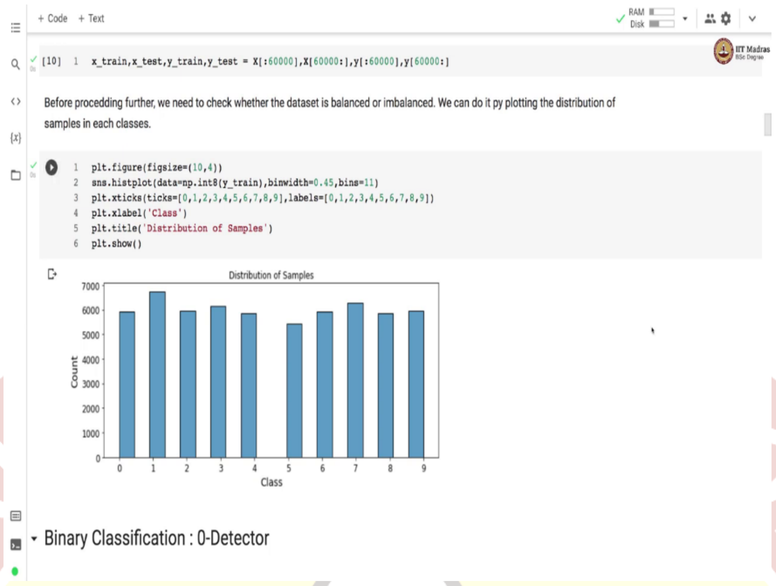
Data Splitting

- Now, we know details such as number of samples, size of each sample, number of features (784), number of classes (targets) about the dataset.
- So let us spilt the total number of samples into train and test set in the following ratio: 60000/10000 (that is, 60000 samples in the training set and 10000 samples in the testing set).
- Since the samples in the data set are already randomly shuffled, we need not to shuffle it again.

```
[10]
x_train,x_test,y_train,y_test = X[:60000],X[60000:],y[:60000],y[60000:]
```

Before procedding further, we need to check whether the dataset is balanced or imbalanced. We can do it py plotting the distribution of samples in each classes.

```
[11]
plt.figure(figsize=(10,4))
sns.histplot(data=np.int8(y_train),binwidth=0.45,bins=11)
plt.xticks(ticks=[0,1,2,3,4,5,6,7,8,9],labels=[0,1,2,3,4,5,6,7,8,9])
plt.xlabel('Class')
plt.title('Distribution of Samples')
plt.show()
```



```
[10] x_train,x_test,y_train,y_test = X[:60000],X[60000:],y[:60000],y[60000:]
```

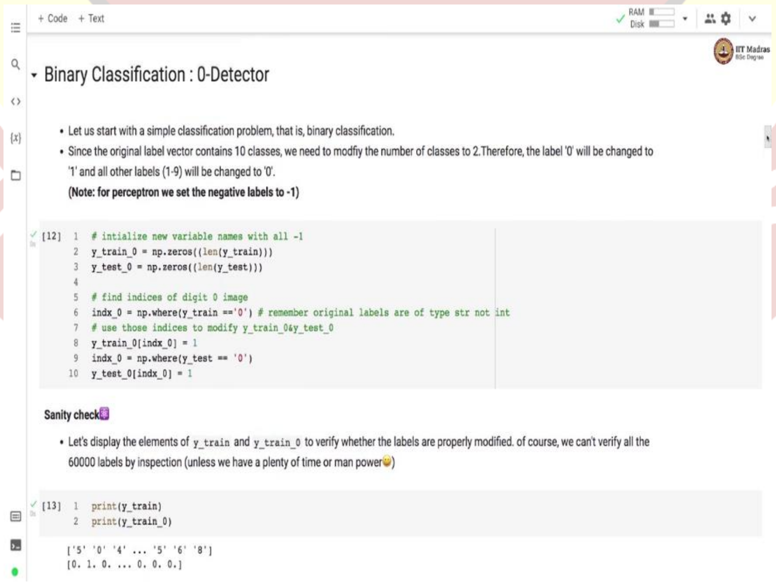
Before procedding further, we need to check whether the dataset is balanced or imbalanced. We can do it py plotting the distribution of samples in each classes.

```
plt.figure(figsize=(10,4))
sns.histplot(data=np.int8(y_train),binwidth=0.45,bins=11)
plt.xticks(ticks=[0,1,2,3,4,5,6,7,8,9],labels=[0,1,2,3,4,5,6,7,8,9])
plt.xlabel('Class')
plt.title('Distribution of Samples')
plt.show()
```

Binary Classification : 0-Detector

As a next step, we split the input examples into training and test sets. So, we use 60,000 examples in the training set and 10,000 examples in the test set. The examples or samples in the training data are already randomly shuffled. So, we need not shuffle it again. So, we get the training set, the training feature matrix and the test feature matrix and the training label and the test labels. You can see that most of the classes, so most of the classes have equal number of examples, and average number of examples per class are about 6000 in the training set.

(Refer Slide Time: 11:03)



Binary Classification : 0-Detector

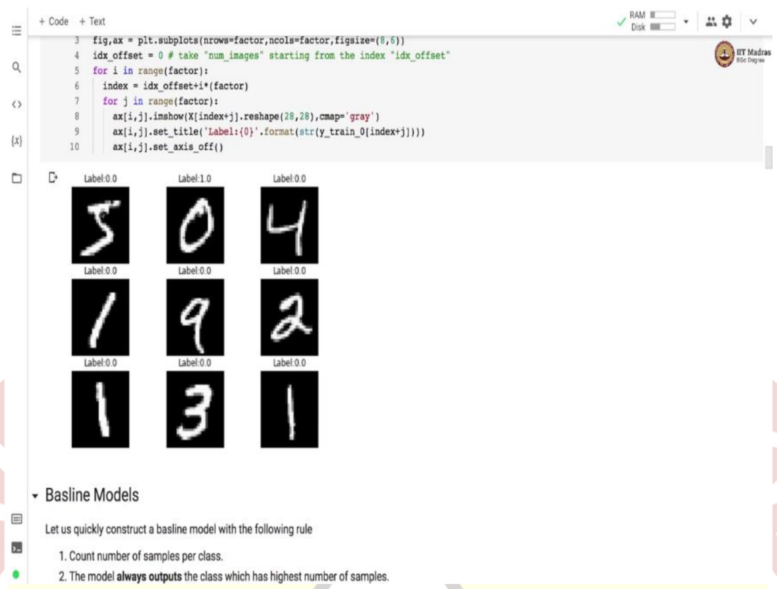
- Let us start with a simple classification problem, that is, binary classification.
- Since the original label vector contains 10 classes, we need to modfiy the number of classes to 2.Therefore, the label '0' will be changed to '1' and all other labels (1-9) will be changed to '0'.
  (Note: for perceptron we set the negative labels to -1)

```
[12] # intialize new variable names with all -1
     y_train_0 = np.zeros((len(y_train)))
     y_test_0 = np.zeros((len(y_test)))

     # find indices of digit 0 image
     indx_0 = np.where(y_train =='0') # remember original labels are of type str not int
     # use those indices to modify y_train_0&y_test_0
     y_train_0[indx_0] = 1
     indx_0 = np.where(y_test == '0')
     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😀)

```
[13] print(y_train)
     print(y_train_0)

['5' '0' '4' ... '5' '6' '8']
[0. 1. 0. ... 0. 0. 0.]
```



```
fig,ax = plt.subplots(nrows=factor,ncols=factor,figsize=(8,6))
idx_offset = 0 # take "num_images" starting from the index "idx_offset"
for i in range(factor):
  index = idx_offset+i*(factor)
  for j in range(factor):
    ax[i,j].imshow(X[index+j].reshape(28,28),cmap='gray')
    ax[i,j].set_title('Label:{0}'.format(str(y_train_0[index+j])))
    ax[i,j].set_axis_off()
```

Basline Models

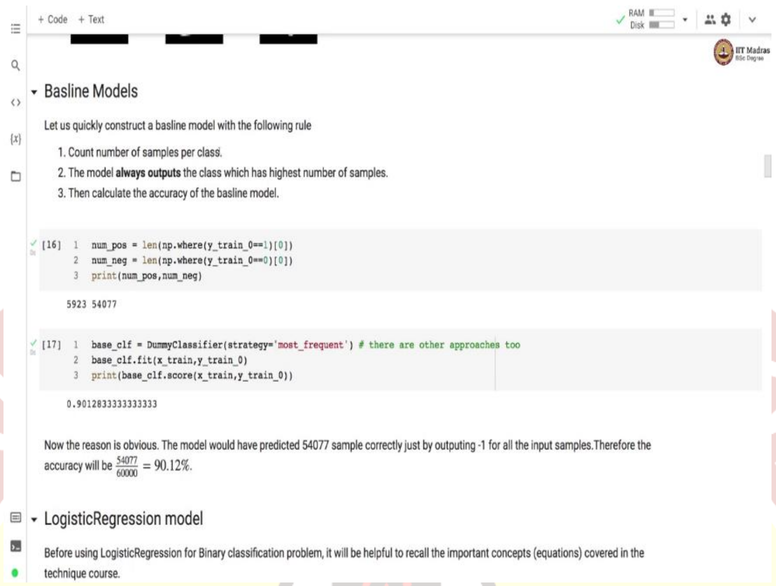
Let us quickly construct a basline model with the following rule

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

So, let us first solve the binary classification problem which deals with detecting the image of digit 0. So, in order to solve this binary classification problem, we need to change the labels of the original examples. So, we will assign label 0 to all the digits between 1 to 9. They will be assigned labeled 0. And for the digit 0 will assign a label of 1, because we are interested in learning the 0 detector.

So, remember that in perceptron we had set the negative label to -1 and here we are setting it to 0. So, now, we will verify whether the labels are assigned correctly. So, you can see that this is label, this is the digit 5 which has got a label of 0. This is digit 0 which has got label of 1. So, you can see that any digit which is non-zero has got a label of 0. Whereas whenever the digit is 0, we have label of 1.

(Refer Slide Time: 12:32)



Basline Models

Let us quickly construct a basline model with the following rule

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 basline model.

```
[16]
num_pos = len(np.where(y_train_0==1)[0])
num_neg = len(np.where(y_train_0==0)[0])
print(num_pos,num_neg)
```

```
5923 54077
```

```
[17]
base_clf = DummyClassifier(strategy='most_frequent') # there are other approaches too
base_clf.fit(x_train,y_train_0)
print(base_clf.score(x_train,y_train_0))
```

```
0.9012833333333333
```

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

LogisticRegression model

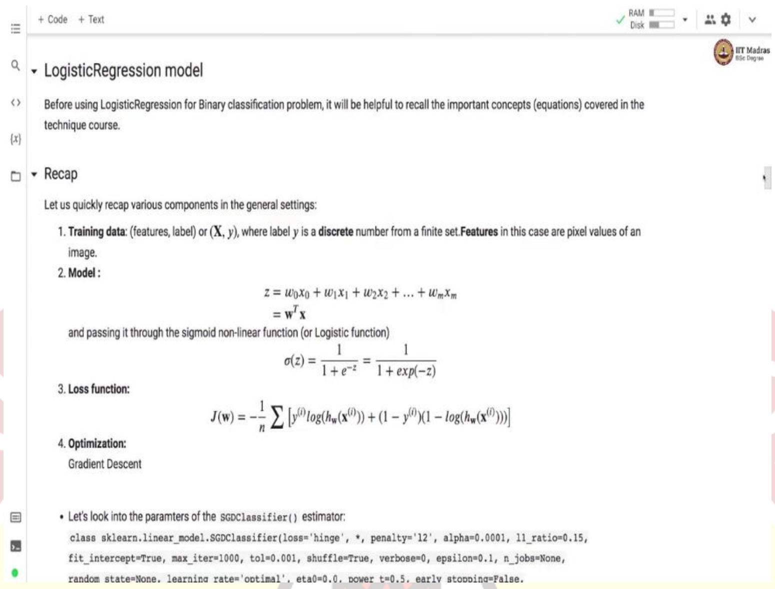
Before using LogisticRegression for Binary classification problem, it will be helpful to recall the important concepts (equations) covered in the technique course.

Let us quickly construct a baseline model with the following rule. We count the number of samples per class. The model always outputs the class which has highest number of samples. And then we calculate the accuracy of the baseline model. So, you can see that there are 5,923 positive examples and 54,077 negative examples. So, obviously, we will assign the negative class to each example in this baseline scheme.

So, we can do this through a dummy classifier by setting the strategy equal to most _frequent. Then we call the fit function or dummy classifier and then we score the accuracy of the dummy classifier and we can see that dummy classifier has accuracy of 0.90 or 90% on the training set. And we can verify that this 90% accuracy is what we obtained by assigning the negative label to each example in the training set.

So, there are 54,077 examples with level 0 and there are total examples 60,000, so the ratio of the number of negative examples to the total examples is 90%, which is what you see over here.

(Refer Slide Time: 14:16)



LogisticRegression model

Before using LogisticRegression for Binary classification problem, it will be helpful to recall the important concepts (equations) covered in the technique course.

Recap

Let us quickly recap various components in the general settings:

1. **Training data**: (features, label) or $(\mathbf{X}, y)$, where label y is a **discrete** number from a finite set. **Features** in this case are pixel values of an image.
2. **Model :**

$$z = w_0x_0 + w_1x_1 + w_2x_2 + \ldots + w_mx_m$$
$$= \mathbf{w}^T\mathbf{x}$$

and passing it through the sigmoid non-linear function (or Logistic function)

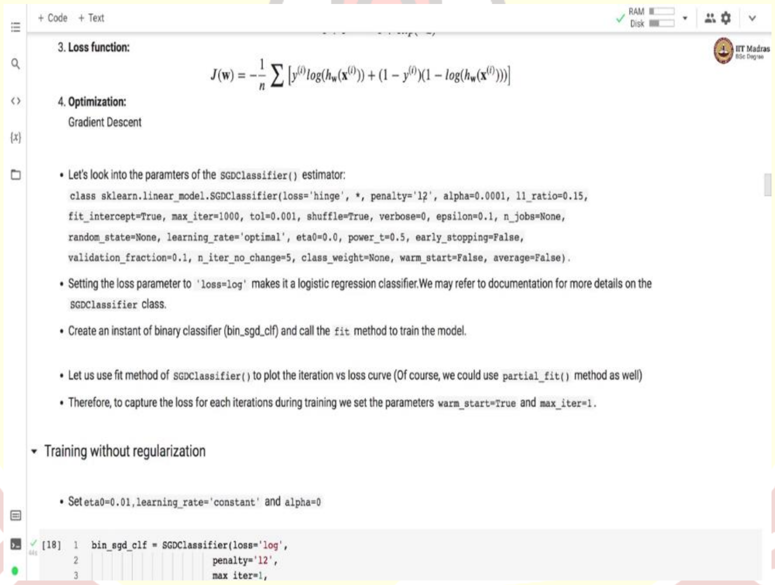
$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+exp(-z)}$$

3. **Loss function:**

$$J(\mathbf{w}) = -\frac{1}{n}\sum\left[y^{(i)}log(h_{\mathbf{w}}(\mathbf{x}^{(i)})) + (1-y^{(i)})(1-log(h_{\mathbf{w}}(\mathbf{x}^{(i)})))\right]$$

4. **Optimization:**
   Gradient Descent

- Let's look into the paramters of the `SGDClassifier()` estimator:
  `class sklearn.linear_model.SGDClassifier(loss='hinge', *, penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=1000, tol=0.001, shuffle=True, verbose=0, epsilon=0.1, n_jobs=None, random_state=None, learning_rate='optimal', eta0=0.0, power_t=0.5, early_stopping=False,`



3. **Loss function:**

$$J(\mathbf{w}) = -\frac{1}{n}\sum\left[y^{(i)}log(h_{\mathbf{w}}(\mathbf{x}^{(i)})) + (1-y^{(i)})(1-log(h_{\mathbf{w}}(\mathbf{x}^{(i)})))\right]$$

4. **Optimization:**
   Gradient Descent

- Let's look into the paramters of the `SGDClassifier()` estimator:
  `class sklearn.linear_model.SGDClassifier(loss='hinge', *, penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=1000, tol=0.001, shuffle=True, verbose=0, epsilon=0.1, n_jobs=None, random_state=None, learning_rate='optimal', eta0=0.0, power_t=0.5, early_stopping=False,`
  `validation_fraction=0.1, n_iter_no_change=5, class_weight=None, warm_start=False, average=False)`.
- Setting the loss parameter to `'loss=log'` makes it a logistic regression classifier.We may refer to documentation for more details on the `SGDClassifier` class.
- Create an instant of binary classifier (bin_sgd_clf) and call the `fit` method to train the model.
- Let us use fit method of `SGDClassifier()` to plot the iteration vs loss curve (Of course, we could use `partial_fit()` method as well)
- Therefore, to capture the loss for each iterations during training we set the parameters `warm_start=True` and `max_iter=1`.

Training without regularization

- Set `eta0=0.01,learning_rate='constant'` and `alpha=0`

```
[18]  bin_sgd_clf = SGDClassifier(loss='log',
                                penalty='l2',
                                max_iter=1,
```

As the next step we will apply the logistic regression model for this binary classification problem. So, before applying logistic regression, it is helpful to recall some of the important concepts cover in machine learning techniques course. So, in case of logistic regression, the training data consists of a pair of feature matrix and label vector.

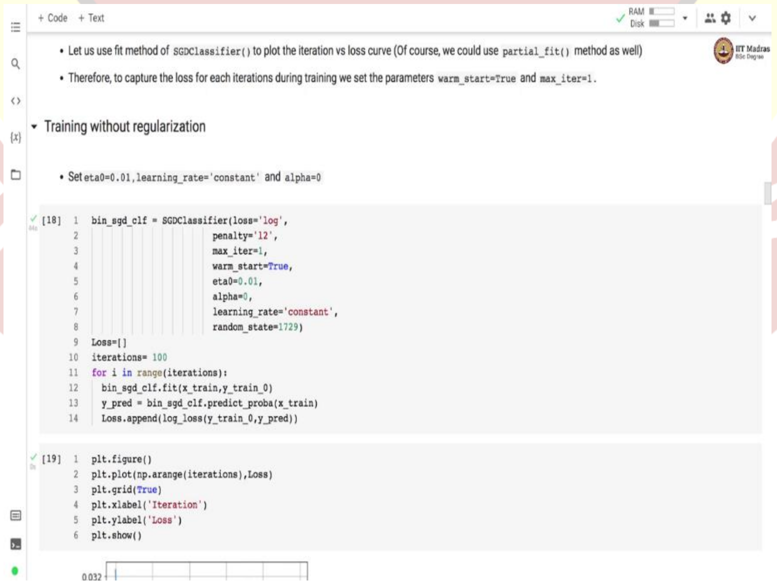
So, logistic regression model first computes the linear combination of features and then pass it through the sigmoid non-linear function and the sigmoid function has the form $\sigma(z) = \frac{1}{1+e^{-z}}$ and z is the linear combination of features. Then we use cross-entropy loss function as a function that we optimize for learning the parameters of logistic regression. And we use gradient descent optimization procedure to optimize the cross-entropy loss.

So, we will first implement the logistic regression with SGDclassifier. So, SGDclassifier has a bunch of default parameters. So, loss is equal to hinge-loss, penalty equal to l2, the value of regularization is equal to 0.0001, l1 ratio is 0.15, fit _intercept equal to true, maximum number of iteration equal to 1000, the tolerance is 0.001, and so on. So, these are the default settings for the SGDclassifier.

So, in order to use this SGDclassifier as a logistic regression classifier, we set the loss parameter to log. And when we set the log's parameter to log, SGDclassifier acts as a logistic regression classifier. So, here, what we will do is we will create an instance of a binary classification and call the fit method to train the model. In order to get the learning curves as we used to discuss in machine learning techniques course, we will use partial fit method and show how to obtain learning curves for SGDclassifier.

And while using partial fit, we set the parameters like warm_starts to true and maximum number of iterations to 1. We will see it in detail in a few minutes.

(Refer Slide Time: 17:22)



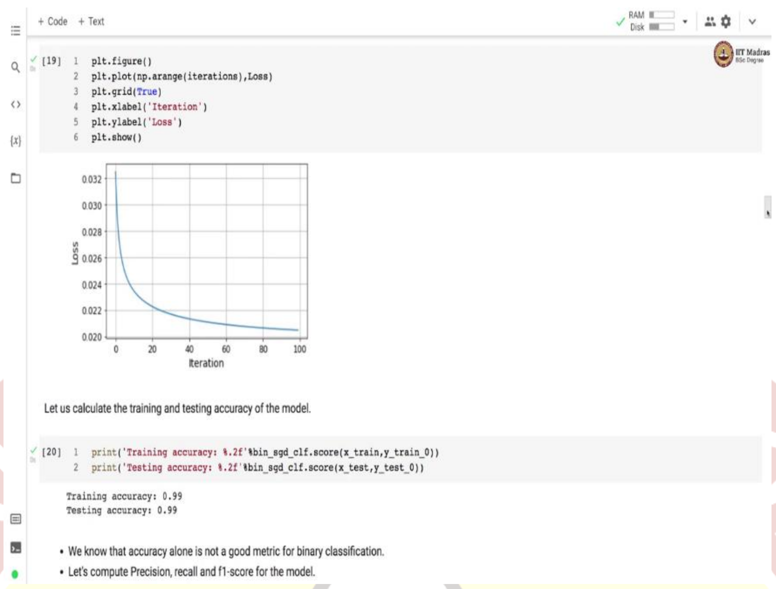
- Let us use fit method of `SGDClassifier()` to plot the iteration vs loss curve (Of course, we could use `partial_fit()` method as well)
- Therefore, to capture the loss for each iterations during training we set the parameters `warm_start=True` and `max_iter=1`.

Training without regularization

- Set `eta0=0.01,learning_rate='constant'` and `alpha=0`

```
[18]
bin_sgd_clf = SGDClassifier(loss='log',
                            penalty='l2',
                            max_iter=1,
                            warm_start=True,
                            eta0=0.01,
                            alpha=0,
                            learning_rate='constant',
                            random_state=1729)
Loss=[]
iterations= 100
for i in range(iterations):
  bin_sgd_clf.fit(x_train,y_train_0)
  y_pred = bin_sgd_clf.predict_proba(x_train)
  Loss.append(log_loss(y_train_0,y_pred))
```

```
[19]
plt.figure()
plt.plot(np.arange(iterations),Loss)
plt.grid(True)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.show()
```



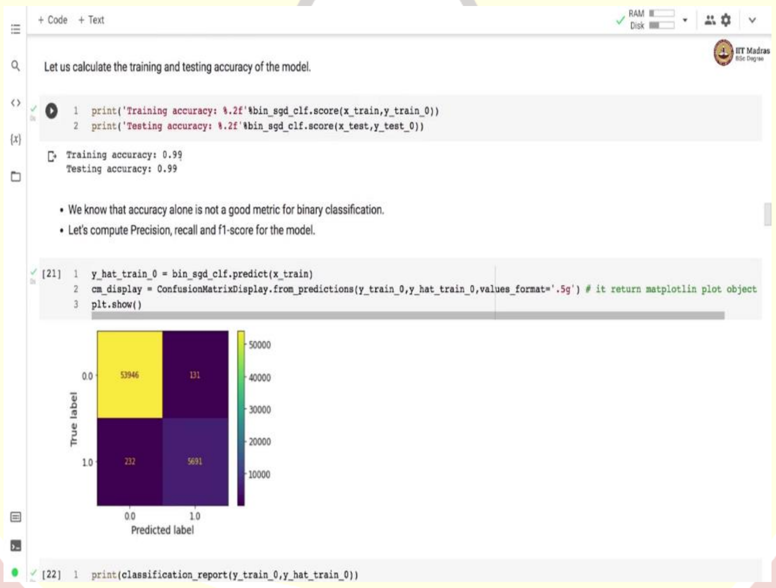
```
[19]
plt.figure()
plt.plot(np.arange(iterations),Loss)
plt.grid(True)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.show()
```

Let us calculate the training and testing accuracy of the model.

```
[20]
print('Training accuracy: %.2f'%bin_sgd_clf.score(x_train,y_train_0))
print('Testing accuracy: %.2f'%bin_sgd_clf.score(x_test,y_test_0))
```

```
Training accuracy: 0.99
Testing accuracy: 0.99
```

- We know that accuracy alone is not a good metric for binary classification.
- Let's compute Precision, recall and f1-score for the model.



Let us calculate the training and testing accuracy of the model.

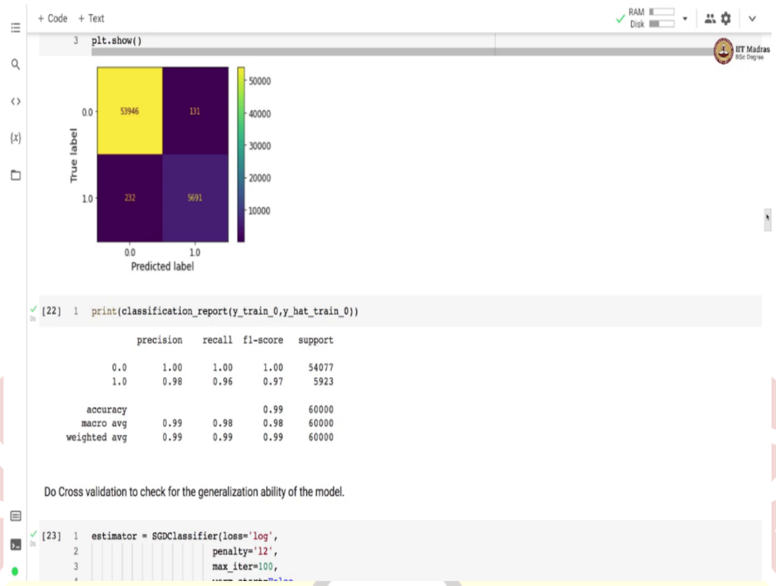
```
print('Training accuracy: %.2f'%bin_sgd_clf.score(x_train,y_train_0))
print('Testing accuracy: %.2f'%bin_sgd_clf.score(x_test,y_test_0))
```

```
Training accuracy: 0.99
Testing accuracy: 0.99
```

- We know that accuracy alone is not a good metric for binary classification.
- Let's compute Precision, recall and f1-score for the model.

```
[21]
y_hat_train_0 = bin_sgd_clf.predict(x_train)
cm_display = ConfusionMatrixDisplay.from_predictions(y_train_0,y_hat_train_0,values_format='.5g') # it return matplotlin plot object
plt.show()
```

```
[22]
print(classification_report(y_train_0,y_hat_train_0))
```



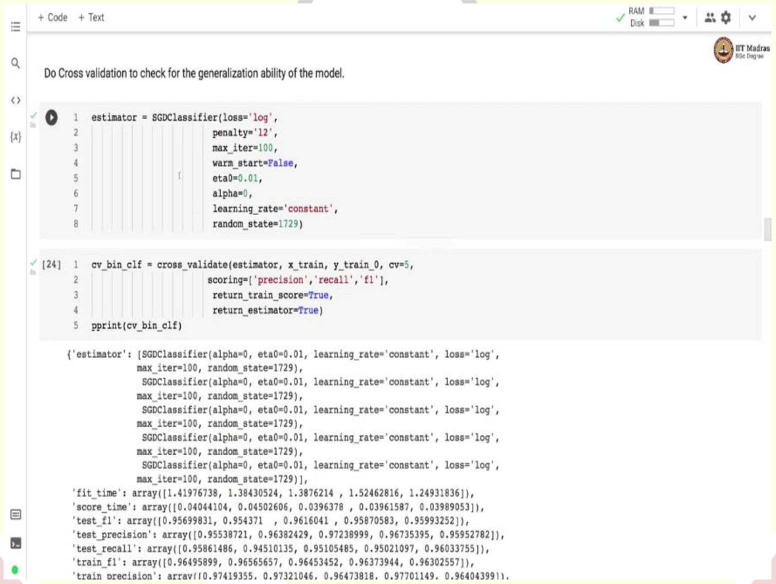
```
plt.show()
```

```
[22] print(classification_report(y_train_0,y_hat_train_0))
```

```
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00     54077
         1.0       0.98      0.96      0.97      5923

    accuracy                           0.99     60000
   macro avg       0.99      0.98      0.98     60000
weighted avg       0.99      0.99      0.99     60000
```

Do Cross validation to check for the generalization ability of the model.

```
[23] estimator = SGDClassifier(loss='log',
                          penalty='l2',
                          max_iter=100,
```

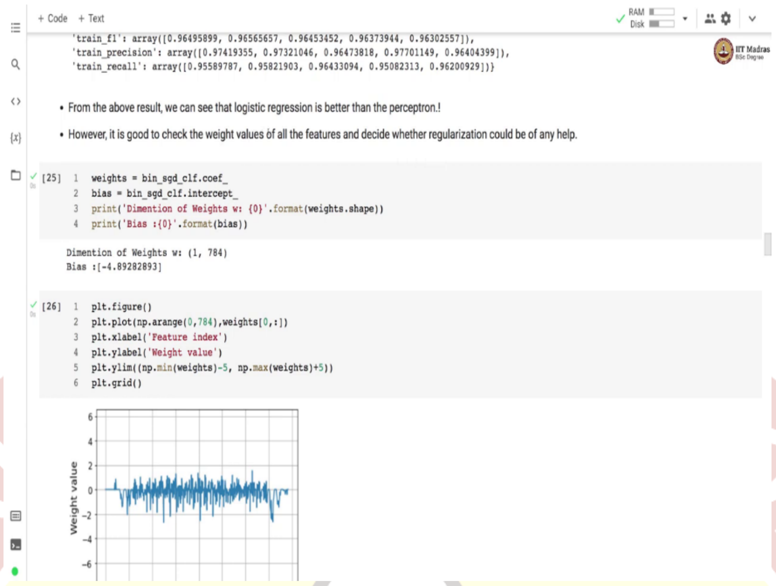


Do Cross validation to check for the generalization ability of the model.

```
estimator = SGDClassifier(loss='log',
                          penalty='l2',
                          max_iter=100,
                          warm_start=False,
                          eta0=0.01,
                          alpha=0,
                          learning_rate='constant',
                          random_state=1729)
```

```
[24] cv_bin_clf = cross_validate(estimator, x_train, y_train_0, cv=5,
                             scoring=['precision','recall','f1'],
                              return_train_score=True,
                              return_estimator=True)
pprint(cv_bin_clf)
```

```
{'estimator': [SGDClassifier(alpha=0, eta0=0.01, learning_rate='constant', loss='log',
              max_iter=100, random_state=1729),
               SGDClassifier(alpha=0, eta0=0.01, learning_rate='constant', loss='log',
              max_iter=100, random_state=1729),
               SGDClassifier(alpha=0, eta0=0.01, learning_rate='constant', loss='log',
              max_iter=100, random_state=1729),
               SGDClassifier(alpha=0, eta0=0.01, learning_rate='constant', loss='log',
              max_iter=100, random_state=1729),
               SGDClassifier(alpha=0, eta0=0.01, learning_rate='constant', loss='log',
              max_iter=100, random_state=1729)],
 'fit_time': array([1.41976738, 1.38430524, 1.3876214 , 1.52462816, 1.24931836]),
 'score_time': array([0.04044104, 0.04502606, 0.0396378 , 0.03961587, 0.03989053]),
 'test_f1': array([0.95699831, 0.954371  , 0.9616041 , 0.95870583, 0.95993252]),
 'test_precision': array([0.95538721, 0.96382429, 0.97238999, 0.96735395, 0.95952782]),
 'test_recall': array([0.95861486, 0.94510135, 0.95105485, 0.95021097, 0.96033755]),
 'train_f1': array([0.96495899, 0.96565657, 0.96453452, 0.96373944, 0.96302557]),
 'train_precision': array([0.97419355, 0.97321046, 0.96473818, 0.97701149, 0.96404399]),
```



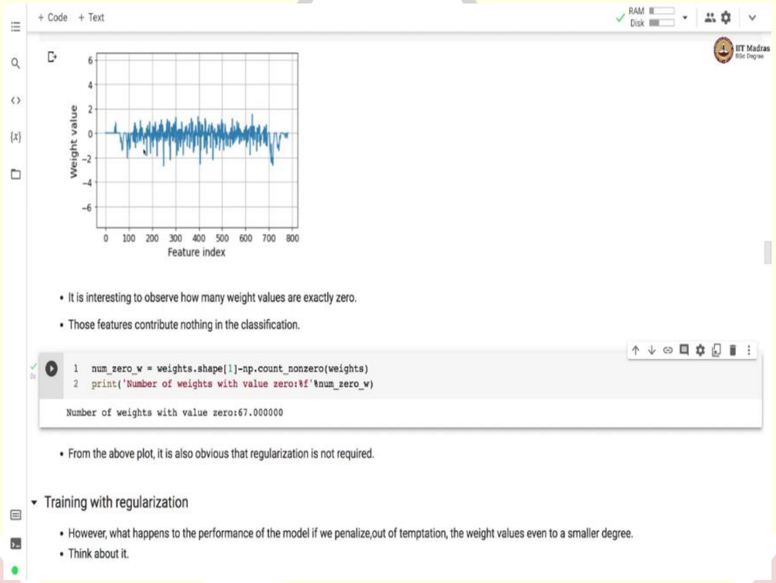
```
'train_f1': array([0.96495899, 0.96565657, 0.96453452, 0.96373944, 0.96302557]),
'train_precision': array([0.97419355, 0.97321046, 0.96473818, 0.97701149, 0.96404399]),
'train_recall': array([0.95589787, 0.95821903, 0.96433094, 0.95082313, 0.96200929])}
```

- From the above result, we can see that logistic regression is better than the perceptron.!
- However, it is good to check the weight values of all the features and decide whether regularization could be of any help.

```
[25]
weights = bin_sgd_clf.coef_
bias = bin_sgd_clf.intercept_
print('Dimention of Weights w: {0}'.format(weights.shape))
print('Bias :{0}'.format(bias))
```

```
Dimention of Weights w: (1, 784)
Bias :[-4.89282893]
```

```
[26]
plt.figure()
plt.plot(np.arange(0,784),weights[0,:])
plt.xlabel('Feature index')
plt.ylabel('Weight value')
plt.ylim((np.min(weights)-5, np.max(weights)+5))
plt.grid()
```



- It is interesting to observe how many weight values are exactly zero.
- Those features contribute nothing in the classification.

```
num_zero_w = weights.shape[1]-np.count_nonzero(weights)
print('Number of weights with value zero:%f'%num_zero_w)
```

```
Number of weights with value zero:67.000000
```

- From the above plot, it is also obvious that regularization is not required.

Training with regularization

- However, what happens to the performance of the model if we penalize,out of temptation, the weight values even to a smaller degree.
- Think about it.

So, to begin with, we train the SGDclassifier without regularization. So, we set alpha equal to 0, which is the regularization rate, and we set the learning rate which is eta0 to 0.01, eta0 is actually the initial learning rate. But since we are setting the learning rate equal to constant, this eta0 will be used as a learning rate throughout the training.

Then we instantiate SGDclassifier with loss equal to log. So, penalty is equal to l2, but we are setting alpha equal to 0. So, this is effectively we are training this model without regularization. We are setting warm start equal to true and maximum iteration equal to 1. Then eta0 which is learning rate is set to 0.01 and learning rate is set to constant.

Now, what we are going to do is we are going to train this SGDclassifier in an iterative manner, where we will run this SGDclassifier for 100 iterations. In each iteration we call the fit method. And the fit method helps us to learn the parameter of the logistic regression model. And then we call the predictor _proba method on the training set in order to get the prediction.

And then we use the log loss in order to calculate the loss due to the model parameters that we have learned and because of which we get the prediction y _pred and we compare that prediction against the actual labels. So, when we run this particular code, we get the learning curves, learning curve as follows. So, we have iteration on x-axis and loss on y-axis. And you can see that with every iteration the loss is coming down.

And as you may recall from our discussion in the machine learning techniques course, this is an ideal learning curve that we get if the model is getting trained properly. Let us calculate the training and test accuracy. And training and test accuracy in our case is 0.99. We know that accuracy alone is not a good metric for binary classification.

Hence, we compute precision, recall and F1 score for the model. So, we first calculate this metric on the training set. So, we first calculate the confusion matrix. This is the confusion matrix that we get on the training set. So, these are the examples that are correctly classified and of diagonal entries are the examples that are incorrectly classified.

We calculate the classification report. And we see that for label 1 we have decision of 0.98, recall of 0.96 and F1 score of 0.97. For label 0 we have precision, recall and F1 score equal to 1. And you might wonder why this score is 1 even if there is some misclassification. This score is 1, because we are rounding with two digits and because of that rounding, we are getting the score that is equal to 1.

As a next step, what we will do is we will train SGDclassifier with cross-validation. So, for the cross-validation based training, we set the number of iterations to 100 and we set the warm start equal to false so that is the only change from our previous training setup. And then we train this particular model with cross-validation. We are using the precision, recall, and F1 scoring for evaluating each model that we learn with cross-validation setup, where we want the training score as well as estimators from the cross-validation, hence, those parameters are set to true.

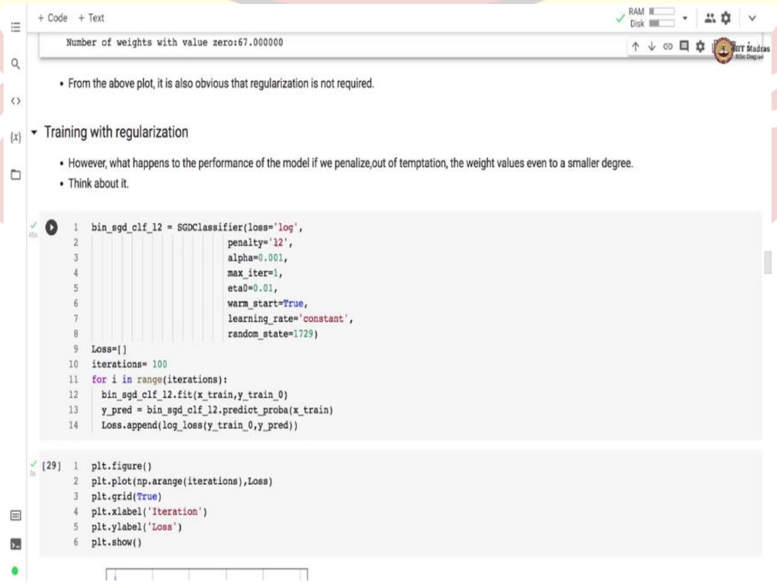
Now, you can see that we have trained this model with five fold cross-validation. Hence, we get five different classifiers. And we get the scores, the time it took to learn each of the classifier on different test sets, the scores like f1 score precision and recall on training as well as on the test set. So, from these results you can see that the logistic regression is performing better than the perceptron classifier.

And here, you can also check that different classifiers have gotten different values for F1 precision and recall and we can select the one that has got the highest values of these metrics. So, for example, the highest F1 score is obtained for this particular model, which is the third model. Now, as the next step, we will look at the weights that we learn in the training for every feature and we will decide whether regularization could be of any help.

So, the intercept term that we got is -4.89. And here what we have done is we have plotted the weights of the remaining 784 features. So, we have feature index on the x-axis and weight value on the y-axis. So, it is interesting to observe how many wait values are exactly 0. And those features contribute nothing in the classification.

So, we wrote a small code snippet to find out number of weights with 0 values. So, there are exactly 67 weights that have a value of 0. So, as such the regularization is not required in this case, because there are hardly any features with very high weight values.

(Refer Slide Time: 24:49)



```
Number of weights with value zero:67.000000
```

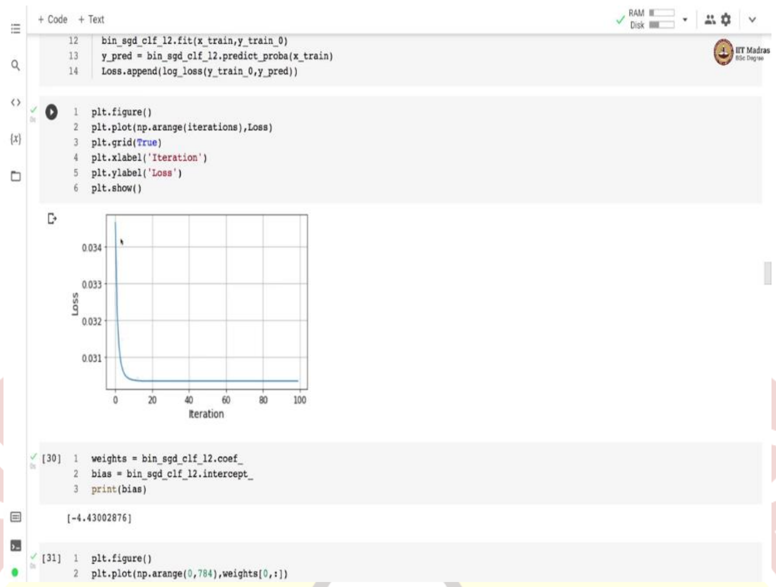
- From the above plot, it is also obvious that regularization is not required.

Training with regularization

- However, what happens to the performance of the model if we penalize,out of temptation, the weight values even to a smaller degree.
- Think about it.

```
bin_sgd_clf_l2 = SGDClassifier(loss='log',
                               penalty='l2',
                               alpha=0.001,
                               max_iter=1,
                               eta0=0.01,
                               warm_start=True,
                               learning_rate='constant',
                               random_state=1729)
Loss=[]
iterations= 100
for i in range(iterations):
  bin_sgd_clf_l2.fit(x_train,y_train_0)
  y_pred = bin_sgd_clf_l2.predict_proba(x_train)
  Loss.append(log_loss(y_train_0,y_pred))
```

```
[29]
plt.figure()
plt.plot(np.arange(iterations),Loss)
plt.grid(True)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.show()
```



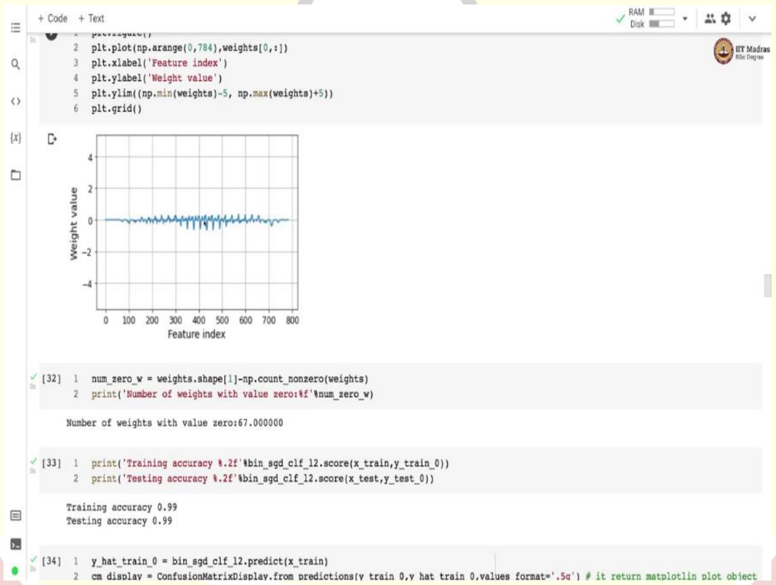
```
bin_sgd_clf_l2.fit(x_train,y_train_0)
y_pred = bin_sgd_clf_l2.predict_proba(x_train)
Loss.append(log_loss(y_train_0,y_pred))
```

```
plt.figure()
plt.plot(np.arange(iterations),Loss)
plt.grid(True)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.show()
```

```
[30]
weights = bin_sgd_clf_l2.coef_
bias = bin_sgd_clf_l2.intercept_
print(bias)
```

```
[-4.43002876]
```

```
[31]
plt.figure()
plt.plot(np.arange(0,784),weights[0,:])
```



```
plt.plot(np.arange(0,784),weights[0,:])
plt.xlabel('Feature index')
plt.ylabel('Weight value')
plt.ylim((np.min(weights)-5, np.max(weights)+5))
plt.grid()
```

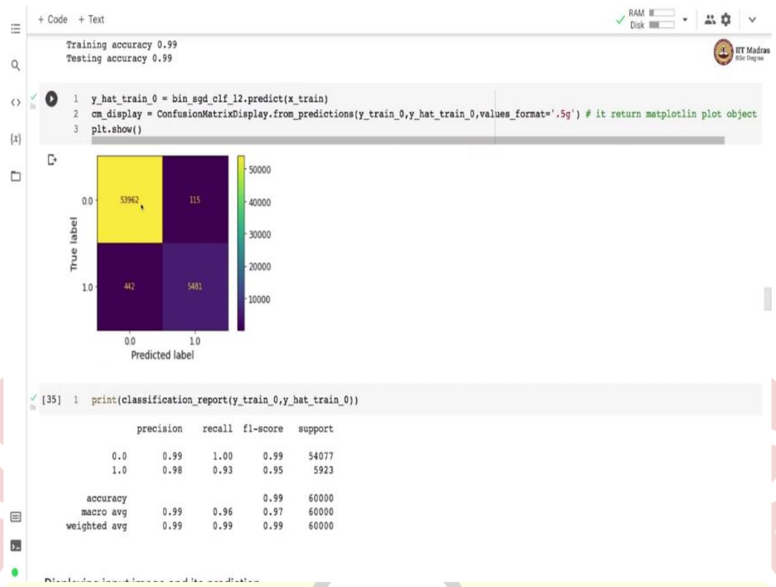
```
[32]
num_zero_w = weights.shape[1]-np.count_nonzero(weights)
print('Number of weights with value zero:%f'%num_zero_w)
```

```
Number of weights with value zero:67.000000
```

```
[33]
print('Training accuracy %.2f'%bin_sgd_clf_l2.score(x_train,y_train_0))
print('Testing accuracy %.2f'%bin_sgd_clf_l2.score(x_test,y_test_0))
```

```
Training accuracy 0.99
Testing accuracy 0.99
```

```
[34]
y_hat_train_0 = bin_sgd_clf_l2.predict(x_train)
cm_display = ConfusionMatrixDisplay.from_predictions(y_train_0,y_hat_train_0,values_format='.5g') # it return matplotlin plot object
```



```
Training accuracy 0.99
Testing accuracy 0.99
```

```
y_hat_train_0 = bin_sgd_clf_l2.predict(x_train)
cm_display = ConfusionMatrixDisplay.from_predictions(y_train_0,y_hat_train_0,values_format='.5g') # it return matplotlin plot object
plt.show()
```

```
[35] print(classification_report(y_train_0,y_hat_train_0))
```

```
              precision    recall  f1-score   support

         0.0       0.99      1.00      0.99     54077
         1.0       0.98      0.93      0.95      5923

    accuracy                           0.99     60000
   macro avg       0.99      0.96      0.97     60000
weighted avg       0.99      0.99      0.99     60000
```

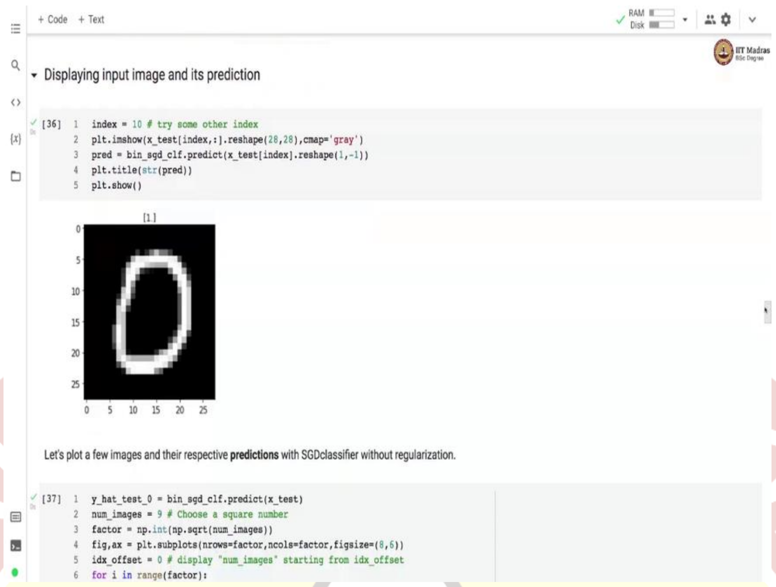
However, in order to demonstrate the regularization, we will perform the regularization. And for regularization, we are using l2 regularization with regularization rate alpha set to 0.001. And we perform the training with warm_start equal to true and maximum iteration equal to 1. And we do this in order to get the, in order to plot the learning curve.

And you can see here that the learning curve is again an ideal learning curve, where the loss is going down after every iteration. But you may have noticed that with regularization, we are getting the loss which is slightly higher than the non regularized model, which is the expected behavior. The bias value is -4.43. And here we have plotted the weights for the 784 different features.

Here you can see that most of the weights are around 0 and there are exactly 67 weights that are 0 just like the earlier classification model. So, again we get the training and test accuracy of 0.99 with the regularized model. Here is the confusion matrix for the regularized SGDclassifier. And you can see that the precision and recall numbers for class 0 and class 1 in this classification report.

For class label 0 we get precision of 0.99 and recall of 1, whereas for class label 1 we get precision of 0.98 and recall of 0.93. The F1 score is 0.95. So, it is interesting to compare this classification report with the earlier classification report. And you can see that the recall has gone down for class 1 from 0.96 to 0.93 and precision for class 0 has got down slightly which you can see over here.

(Refer Slide Time: 27:38)

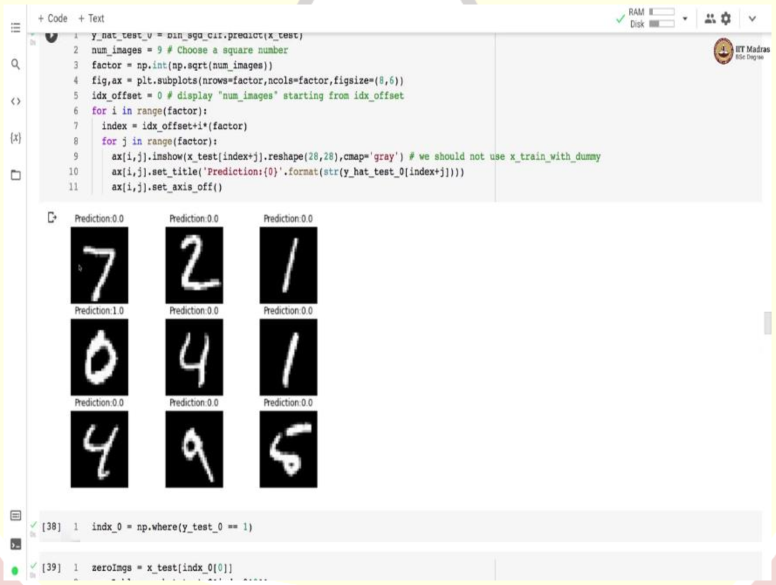


Displaying input image and its prediction

```
[36]
index = 10 # try some other index
plt.imshow(x_test[index,:].reshape(28,28),cmap='gray')
pred = bin_sgd_clf.predict(x_test[index].reshape(1,-1))
plt.title(str(pred))
plt.show()
```

Let's plot a few images and their respective **predictions** with SGDclassifier without regularization.

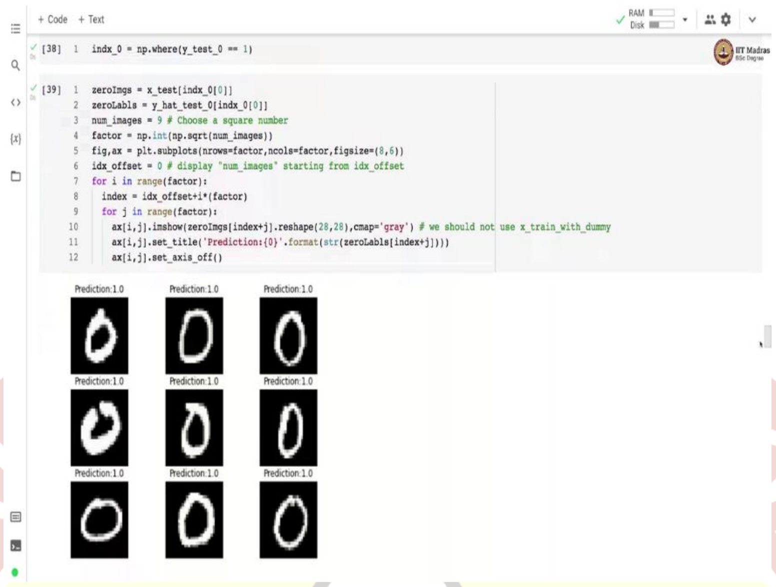
```
[37]
y_hat_test_0 = bin_sgd_clf.predict(x_test)
num_images = 9 # Choose a square number
factor = np.int(np.sqrt(num_images))
fig,ax = plt.subplots(nrows=factor,ncols=factor,figsize=(8,6))
idx_offset = 0 # display "num_images" starting from idx_offset
for i in range(factor):
```



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

```
[38]
indx_0 = np.where(y_test_0 == 1)
```

```
[39]
zeroImgs = x_test[indx_0[0]]
```



```
[38]
indx_0 = np.where(y_test_0 == 1)
```

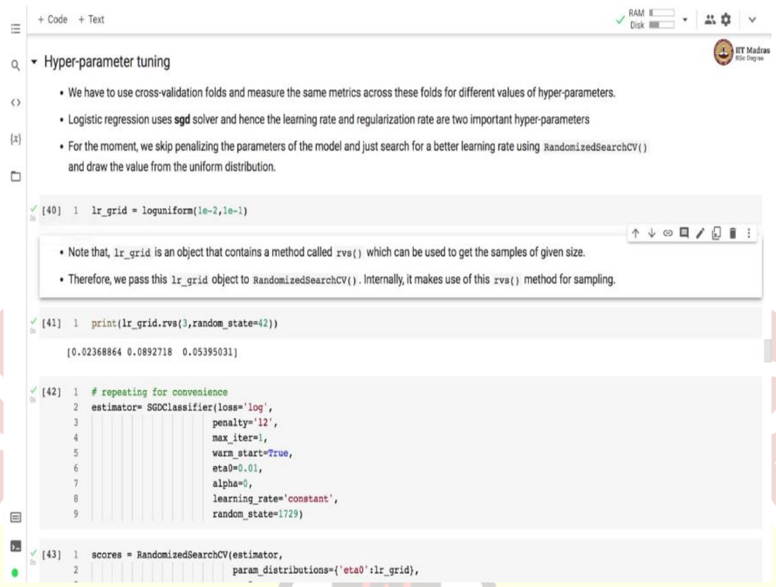
```
[39]
zeroImgs = x_test[indx_0[0]]
zeroLabls = y_hat_test_0[indx_0[0]]
num_images = 9 # Choose a square number
factor = np.int(np.sqrt(num_images))
fig,ax = plt.subplots(nrows=factor,ncols=factor,figsize=(8,6))
idx_offset = 0 # display "num_images" starting from idx_offset
for i in range(factor):
  index = idx_offset+i*(factor)
  for j in range(factor):
    ax[i,j].imshow(zeroImgs[index+j].reshape(28,28),cmap='gray') # we should not use x_train_with_dummy
    ax[i,j].set_title('Prediction:{0}'.format(str(zeroLabls[index+j])))
    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
```

Next what we do is we display the input images and their predictions. So, this is the input image which is 0. And since we have a 0 detector model, it assigns label 1 to this particular image. We will plot few more images along with their predicted labels. And you can see that in each of these cases, the predicted labels make sense.

So, this is image 7 which is non-zero image that is why we have prediction of 0. For digit 2 we have prediction of 0. For digit 1, we have prediction of 0. Similarly, here for digit 4, digit 9, digit 5 all of them get a prediction of 0, whereas for digit 0 we get prediction of 1. So, these are all different images of digit 0 that are correctly predicted to 1.

(Refer Slide Time: 28:50)



Hyper-parameter tuning

- We have to use cross-validation folds and measure the same metrics across these folds for different values of hyper-parameters.
- Logistic regression uses **sgd** solver and hence the learning rate and regularization rate are two important hyper-parameters
- For the moment, we skip penalizing the parameters of the model and just search for a better learning rate using `RandomizedSearchCV()` and draw the value from the uniform distribution.

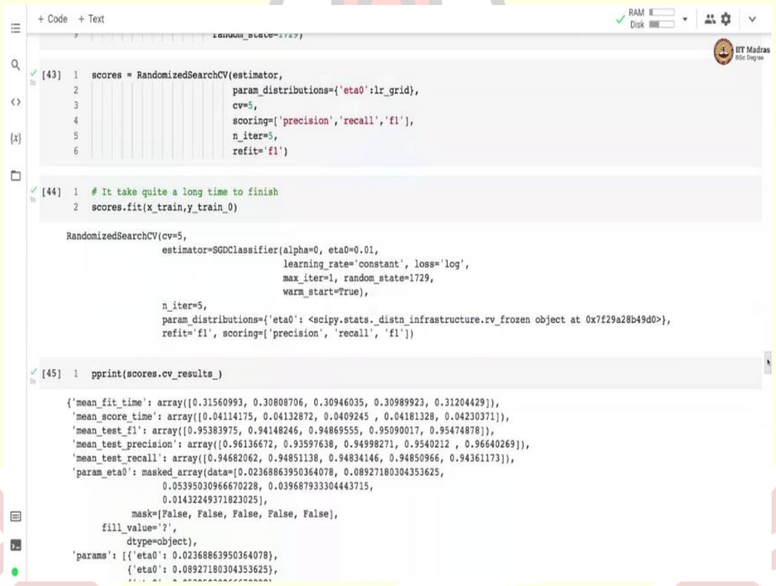
```
[40]
lr_grid = loguniform(1e-2,1e-1)
```

- Note that, `lr_grid` is an object that contains a method called `rvs()` which can be used to get the samples of given size.
- Therefore, we pass this `lr_grid` object to `RandomizedSearchCV()`. Internally, it makes use of this `rvs()` method for sampling.

```
[41]
print(lr_grid.rvs(3,random_state=42))
```

```
[0.02368864 0.0892718  0.05395031]
```

```
[42]
# repeating for convenience
estimator= SGDClassifier(loss='log',
                         penalty='l2',
                         max_iter=1,
                         warm_start=True,
                         eta0=0.01,
                         alpha=0,
                         learning_rate='constant',
                         random_state=1729)
```



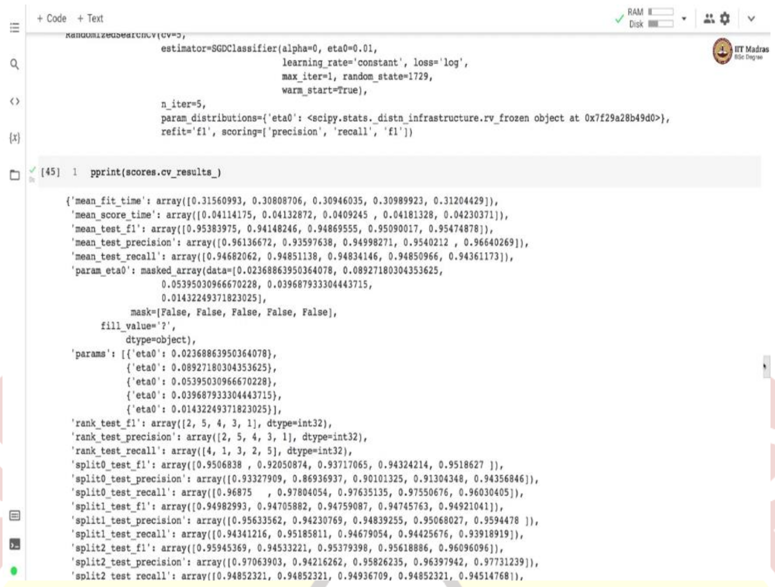
```
[43]
scores = RandomizedSearchCV(estimator,
                            param_distributions={'eta0':lr_grid},
                            cv=5,
                            scoring=['precision','recall','f1'],
                            n_iter=5,
                            refit='f1')
```

```
[44]
# It take quite a long time to finish
scores.fit(x_train,y_train_0)
```

```
RandomizedSearchCV(cv=5,
                   estimator=SGDClassifier(alpha=0, eta0=0.01,
                                           learning_rate='constant', loss='log',
                                           max_iter=1, random_state=1729,
                                           warm_start=True),
                   n_iter=5,
                   param_distributions={'eta0': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7f29a28b49d0>},
                   refit='f1', scoring=['precision', 'recall', 'f1'])
```

```
[45]
pprint(scores.cv_results_)
```

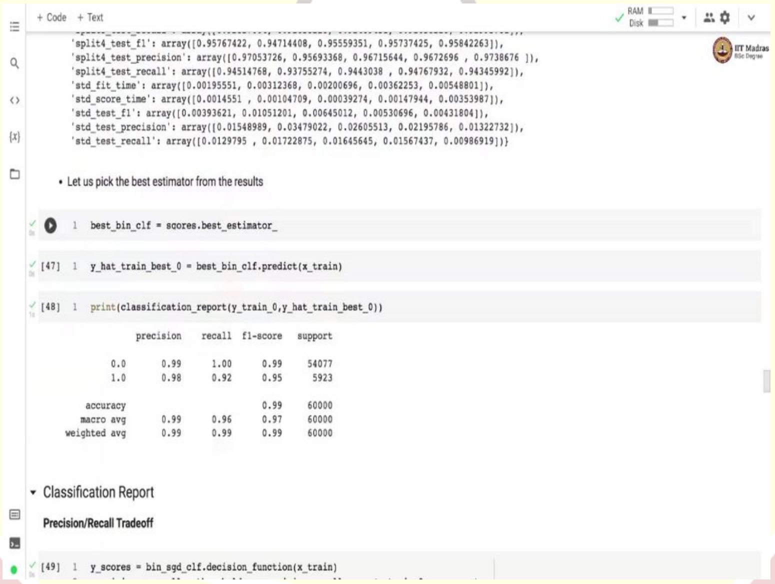
```
{'mean_fit_time': array([0.31560993, 0.30808706, 0.30946035, 0.30989923, 0.31204429]),
 'mean_score_time': array([0.04114175, 0.04132872, 0.0409245 , 0.04181328, 0.04230371]),
 'mean_test_f1': array([0.95383975, 0.94148246, 0.94869555, 0.95090017, 0.95474878]),
 'mean_test_precision': array([0.96136672, 0.93597638, 0.94998271, 0.9540212 , 0.96640269]),
 'mean_test_recall': array([0.94682062, 0.94851138, 0.94834146, 0.94850966, 0.94361173]),
 'param_eta0': masked_array(data=[0.02368863950364078, 0.08927180304353625,
                   0.05395030966670228, 0.039687933304443715,
                   0.01432249371823025],
             mask=[False, False, False, False, False],
       fill_value='?',
            dtype=object),
 'params': [{'eta0': 0.02368863950364078},
            {'eta0': 0.08927180304353625},
```



```
RandomizedSearchCV(cv=5,
                   estimator=SGDClassifier(alpha=0, eta0=0.01,
                                           learning_rate='constant', loss='log',
                                           max_iter=1, random_state=1729,
                                           warm_start=True),
                   n_iter=5,
                   param_distributions={'eta0': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7f29a28b49d0>},
                   refit='f1', scoring=['precision', 'recall', 'f1'])
```

```
[45] pprint(scores.cv_results_)
```

```
{'mean_fit_time': array([0.31560993, 0.30808706, 0.30946035, 0.30989923, 0.31204429]),
 'mean_score_time': array([0.04114175, 0.04132872, 0.0409245 , 0.04181328, 0.04230371]),
 'mean_test_f1': array([0.95383975, 0.94148246, 0.94869555, 0.95090017, 0.95474878]),
 'mean_test_precision': array([0.96136672, 0.93597638, 0.94998271, 0.9540212 , 0.96640269]),
 'mean_test_recall': array([0.94682062, 0.94851138, 0.94834146, 0.94850966, 0.94361173]),
 'param_eta0': masked_array(data=[0.02368863950364078, 0.08927180304353625,
                   0.05395030966670228, 0.039687933304443715,
                   0.01432249371823025],
             mask=[False, False, False, False, False],
       fill_value='?',
            dtype=object),
 'params': [{'eta0': 0.02368863950364078},
            {'eta0': 0.08927180304353625},
            {'eta0': 0.05395030966670228},
            {'eta0': 0.039687933304443715},
            {'eta0': 0.01432249371823025}],
 'rank_test_f1': array([2, 5, 4, 3, 1], dtype=int32),
 'rank_test_precision': array([2, 5, 4, 3, 1], dtype=int32),
 'rank_test_recall': array([4, 1, 3, 2, 5], dtype=int32),
 'split0_test_f1': array([0.9506838 , 0.92050874, 0.93717065, 0.94324214, 0.9518627 ]),
 'split0_test_precision': array([0.93327909, 0.86936937, 0.90101325, 0.91304348, 0.94356846]),
 'split0_test_recall': array([0.96875   , 0.97804054, 0.97635135, 0.97550676, 0.96030405]),
 'split1_test_f1': array([0.94982993, 0.94705882, 0.94759087, 0.94745763, 0.94921041]),
 'split1_test_precision': array([0.95633562, 0.94230769, 0.94839255, 0.95068027, 0.9594478 ]),
 'split1_test_recall': array([0.94341216, 0.95185811, 0.94679054, 0.94425676, 0.93918919]),
 'split2_test_f1': array([0.95945369, 0.94533221, 0.95379398, 0.95618886, 0.96096096]),
 'split2_test_precision': array([0.97063903, 0.94216262, 0.95826235, 0.96397942, 0.97731239]),
 'split2_test_recall': array([0.94852321, 0.94852321, 0.94936709, 0.94852321, 0.94514768]),
```



```
 'split4_test_f1': array([0.95767422, 0.94714408, 0.95559351, 0.95737425, 0.95842263]),
 'split4_test_precision': array([0.97053726, 0.95693368, 0.96715644, 0.9672696 , 0.9738676 ]),
 'split4_test_recall': array([0.94514768, 0.93755274, 0.9443038 , 0.94767932, 0.94345992]),
 'std_fit_time': array([0.00195551, 0.00312368, 0.00200696, 0.00362253, 0.00548801]),
 'std_score_time': array([0.0014551 , 0.00104709, 0.00039274, 0.00147944, 0.00353987]),
 'std_test_f1': array([0.00393621, 0.01051201, 0.00645012, 0.00530696, 0.00431804]),
 'std_test_precision': array([0.01548989, 0.03479022, 0.02605513, 0.02195786, 0.01322732]),
 'std_test_recall': array([0.0129795 , 0.01722875, 0.01645645, 0.01567437, 0.00986919])}
```

- Let us pick the best estimator from the results

```
best_bin_clf = scores.best_estimator_
```

```
[47] y_hat_train_best_0 = best_bin_clf.predict(x_train)
```

```
[48] print(classification_report(y_train_0,y_hat_train_best_0))
```

```
              precision    recall  f1-score   support

         0.0       0.99      1.00      0.99     54077
         1.0       0.98      0.92      0.95      5923

    accuracy                           0.99     60000
   macro avg       0.99      0.96      0.97     60000
weighted avg       0.99      0.99      0.99     60000
```

Classification Report

Precision/Recall Tradeoff

```
[49] y_scores = bin_sgd_clf.decision_function(x_train)
```

As a next step, we perform hyper-parameter tuning. So, there are two hyper-parameters in our case, one is the learning rate and second is regularization rate. For the moment we skip the regularization rate and we train the SGDclassifier without any regularization. And here we will just search for the learning rate using RandomizedSearchCV.

So, we define a grid for learning rate. And in case of RandomizedSearchCV we defined a distribution from where the hyper-parameters will be drawn. So, we are going to use log uniform distribution between 10^{-2} and 10^{-1}. So, lr _grid is an object that contains a method called rvs, which can be used to get samples of a given size.

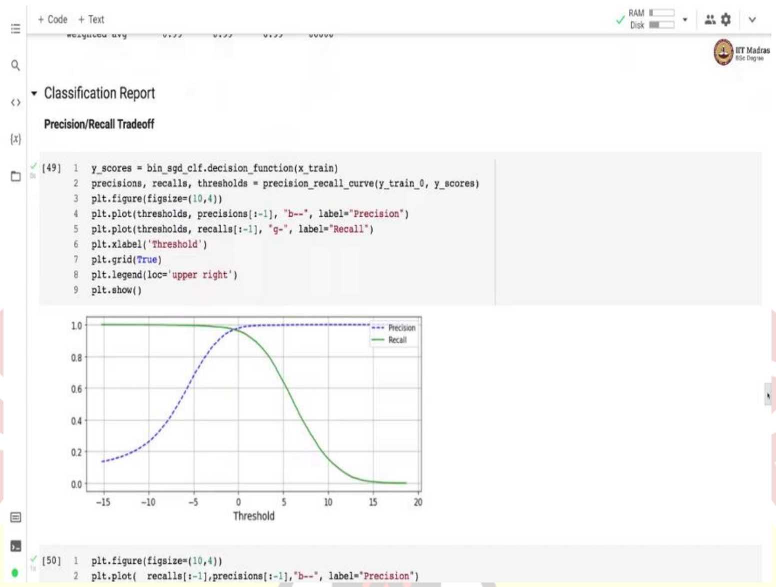
Therefore, we pass this lr _grid object to RandomizedSearchCV, which internally makes use of rvs method for sampling. So, if we call rvs method on lr _grid object with the number of samples to 3 and random state to 42, we get these three samples for the learning rates. So, we are defining the SGDclassifier with loss equal to log, penalty equal to l2, but we are setting alpha equal to 0.

So, this is effectively a non-regularized model. We are setting maximum iteration to 1 and warm _start equal to true. We are setting the initial learning rate to 0.01 with learning rate equal to constant. Now, we perform the, we define the RandomizedSearchCV object with this estimator class as an argument and then setting the parameter distribution and here we are going to search for learning rate and the parameter distribution for learning rate is set to lr _grid.

We will perform, we perform five fold cross-validation and we will use precision, recall and F1 scoring for hyper-parameter tuning. We want to perform five iterations of RandomizedSearchCV and we want to refit the model on the F1 score. So, now, we call the fit method on the RandomizedSearchCV by providing the feature matrix and the labels.

Now, we are printing the result of the cross-validation. And since you are performing five fold cross-validation, we have results from five different folds. So, we select the best estimator using scores**.**best _estimator _ is our best estimator. And for that best estimator, we see for label 1, the precision of 0.98 and recall of 0.92, whereas for the label 0 we have precision of 0.99 and recall of 1.

(Refer Slide Time: 32:20)

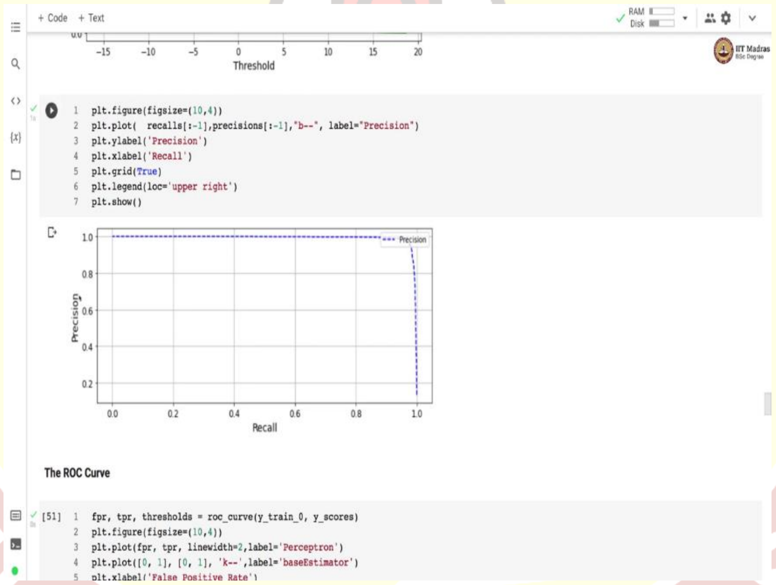


Classification Report

Precision/Recall Tradeoff

```
[49] y_scores = bin_sgd_clf.decision_function(x_train)
     precisions, recalls, thresholds = precision_recall_curve(y_train_0, y_scores)
     plt.figure(figsize=(10,4))
     plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
     plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
     plt.xlabel('Threshold')
     plt.grid(True)
     plt.legend(loc='upper right')
     plt.show()
```

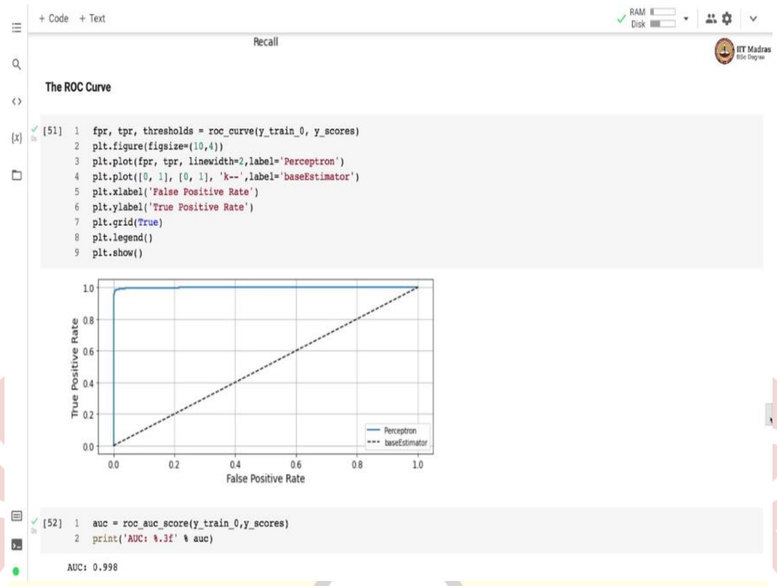
```
[50] plt.figure(figsize=(10,4))
     plt.plot(  recalls[:-1],precisions[:-1],"b--", label="Precision")
```



```
plt.figure(figsize=(10,4))
plt.plot(  recalls[:-1],precisions[:-1],"b--", label="Precision")
plt.ylabel('Precision')
plt.xlabel('Recall')
plt.grid(True)
plt.legend(loc='upper right')
plt.show()
```

The ROC Curve

```
[51] fpr, tpr, thresholds = roc_curve(y_train_0, y_scores)
     plt.figure(figsize=(10,4))
     plt.plot(fpr, tpr, linewidth=2,label='Perceptron')
     plt.plot([0, 1], [0, 1], 'k--',label='baseEstimator')
     plt.xlabel('False Positive Rate')
```



The ROC Curve

```
[51]
fpr, tpr, thresholds = roc_curve(y_train_0, y_scores)
plt.figure(figsize=(10,4))
plt.plot(fpr, tpr, linewidth=2,label='Perceptron')
plt.plot([0, 1], [0, 1], 'k--',label='baseEstimator')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.grid(True)
plt.legend()
plt.show()
```

```
[52]
auc = roc_auc_score(y_train_0,y_scores)
print('AUC: %.3f' % auc)
```

```
AUC: 0.998
```

We also get the precision recall trade off by plotting the PR curve where we have recall on x-axis and precision on y-axis and this is kind of an ideal PR curve which covers pretty much the entire grid from 1 to 1. And we also plot an ROC curve which is again a very ideal ROC curve that we have gotten.

We get the AUC under ROC curve and that AUC is 0.99 which is quite a high value and we get similar AUC, if we get AUC for the PR curve we will also get a very high area under the curve for the PR curve. In this video we used SGDclassifier to train logistic regression model to detect digit 0 from images.