

IIT Madras

ONLINE DEGREE

Machine Learning Practice
Professor Doctor Ashish Tendulkar
Indian Institute of Technology, Madras
Demonstration- SVC on MNIST dataset

(Refer Slide Time: 0:11)



```
[1] 1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import seaborn as sns
4 import numpy as np
5 from pandas.plotting import scatter_matrix
6 from sklearn.preprocessing import OrdinalEncoder
7 from sklearn.model_selection import train_test_split

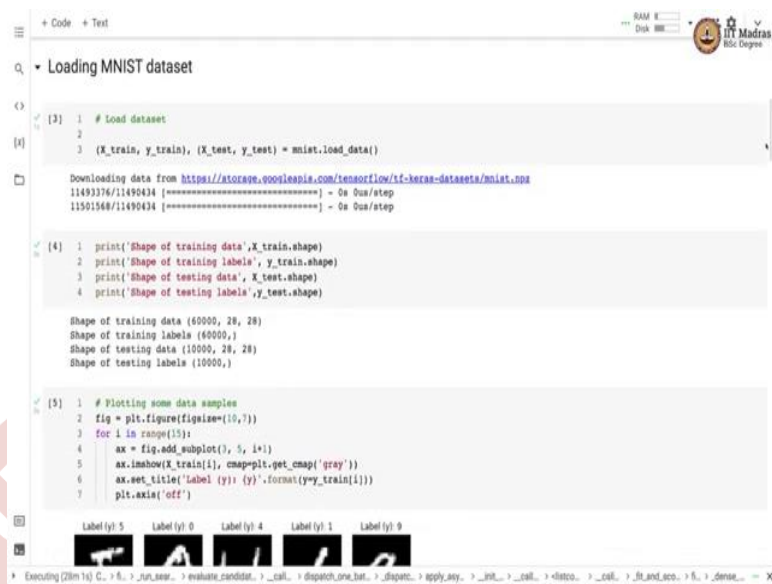
[2] 1 # Import the libraries for performing classification
2 from keras.datasets import mnist
3 from sklearn.svm import SVC
4 from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
5 from sklearn.metrics import classification_report
6 from sklearn.metrics import plot_roc_curve
7 from sklearn.metrics import roc_auc_score
8 from sklearn.preprocessing import StandardScaler, MinMaxScaler
9 from sklearn.pipeline import Pipeline
10 from sklearn import metrics
11 from sklearn.metrics import plot_confusion_matrix
12 from sklearn.model_selection import cross_val_score
13 from sklearn.model_selection import GridSearchCV
14 from sklearn.model_selection import StratifiedShuffleSplit

[3] 1 # Load dataset
2
3 (X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Namaste, welcome to the next video of Machine Learning Practice course. In this scholar we will implement multi-class MNIST digit recognition classifier with SVMs, we begin by importing basic libraries like pandas, matplotlib, cbon, numpy, then pandas.plotting, sql under pre-processing and model selection.

We also import libraries for performing classification such as sklearn.svm from that library or from that module we import SVC api. We also have bunch of imports from sklearn metrics to obtain the classification report and confusion matrix, from model selection we have imported val score, grid search CV and stratified shuffle split.

(Refer Slide Time: 1:07)



```
+ Code + Text
Loading MNIST dataset

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


Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [=====] - 0s 0us/step
11501548/11490434 [=====] - 0s 0us/step

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

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


[5]: 1 # Plotting some data samples
      2 fig = plt.figure(figsize=(10,7))
      3 for i in range(15):
      4     ax = fig.add_subplot(3, 5, i+1)
      5     ax.imshow(X_train[i], cmap=plt.get_cmap('gray'))
      6     ax.set_title('Label (y): %s'%format(y_train[i]))
      7     plt.axis('off')
```

Label (y): 5 Label (y): 0 Label (y): 4 Label (y): 1 Label (y): 9



We begin by loading MNIST data set, this time we use keras.dataset utility MNIST to load the data. So, we call `mnist.load_data` and that returns the training feature matrix and training label vector along with test feature matrix and test label vector. So, training data contains 60000 images and each image is 28×28 . In the test data we have 10000 images again with the same dimension which is 28×28 and the label vectors have 60000 examples in the training and 10000 examples in the test.

(Refer Slide Time: 1:53)




```
+ Code + Text

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

[4]: 1 # Plotting some data samples
      2 fig = plt.figure(figsize=(10,7))
      3 for i in range(15):
      4     ax = fig.add_subplot(3, 5, i+1)
      5     ax.imshow(X_train[i], cmap=plt.get_cmap('gray'))
      6     ax.set_title('Label (y): %s'%format(y_train[i]))
      7     plt.axis('off')
```


Label (y): 5 Label (y): 0 Label (y): 4 Label (y): 1 Label (y): 9

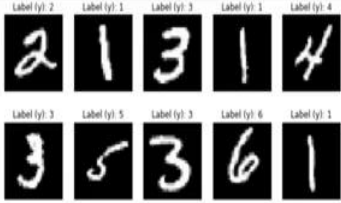


Label (y): 2 Label (y): 1 Label (y): 3 Label (y): 1 Label (y): 4



Label (y): 3 Label (y): 5 Label (y): 3 Label (y): 6 Label (y): 1





```

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

[7] 1 print('Shape of training data after flattening', X_train.shape)
    2 print('Shape of testing data after flattening', X_test.shape)

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

Let us consider the first 10,000 images in training dataset and first 2,000 images in testing dataset.

[8] 1 X_train = X_train[0:10000,:]

```

We will plot some of the data samples so you can see image corresponding to digits along with the label, so this is the image for digit 5 and the label is 5. So, the first step which flatten each input image into a vector of length 784, so we reshape the training feature matrix or training feature tensor with the shape 60000 by 784 and the test tensor is reshaped to 10000 by 784. Then we normalize the training feature matrix and a test feature matrix by dividing each value in it by 255.

(Refer Slide Time: 2:47)

```

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

[7] 1 print('Shape of training data after flattening', X_train.shape)
    2 print('Shape of testing data after flattening', X_test.shape)

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

Let us consider the first 10,000 images in training dataset and first 2,000 images in testing dataset.

[8] 1 X_train = X_train[0:10000,:]
    2 y_train = y_train[0:10000]
    3 X_test = X_test[0:2000,:]
    4 y_test = y_test[0:2000]

[9] 1 print('Shape of training data', X_train.shape)
    2 print('Shape of training labels', y_train.shape)
    3 print('Shape of testing data', X_test.shape)
    4 print('Shape of testing labels', y_test.shape)

Shape of training data (10000, 784)
Shape of training labels (10000,)
Shape of testing data (2000, 784)
Shape of testing labels (2000,)

```

```

+ Code + Text
[7] 1 print('Shape of training data after flattening', X_train.shape)
2 print('Shape of testing data after flattening', X_test.shape)

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

Let us consider the first 10,000 images in training dataset and first 2,000 images in testing dataset.

[8] 1 X_train = X_train[0:10000,:]
2 y_train = y_train[0:10000]
3 X_test = X_test[0:2000,:]
4 y_test = y_test[0:2000]

[9] 1 print('Shape of training data', X_train.shape)
2 print('Shape of training labels', y_train.shape)
3 print('Shape of testing data', X_test.shape)
4 print('Shape of testing labels', y_test.shape)

Shape of training data (10000, 784)
Shape of training labels (10000,)
Shape of testing data (2000, 784)
Shape of testing labels (2000,)

- Linear SVM for MNIST multiclass classification

- Using Pipeline

```

So, you can see that the shape of the training data after flattening is 60000 by 784 and the shape of the test data after flattening is 10000 by 784. In order to run these experiments quickly we consider first 10000 images as training set and the first 2000 images in the test set as test. You can re-run this collab by taking the full training and full test data. So, we have 10000 examples in the training and 2000 examples in the test.

(Refer Slide Time: 3:27)

```

+ Code + Text
- Linear SVM for MNIST multiclass classification

- Using Pipeline

[10] 1 pipe_1 = Pipeline([['scaler', MinMaxScaler()],
2 ('classifier', SVC(kernel = 'linear', C = 1))])
3 pipe_1.fit(X_train, y_train.ravel())
4
5 # Evaluate the model using crossvalidation
6 acc = cross_val_score(pipe_1, X_train, y_train.ravel(), cv=2)
7 print('Training Accuracy: {:.2f} %'.format(acc.mean()*100))

Training Accuracy: 91.07 %

[11] 1 # visualizing the confusion matrix
2 y_pred = pipe_1.predict(X_test)
3 cm = confusion_matrix(y_test, y_pred)
4 disp = ConfusionMatrixDisplay(confusion_matrix=cm)
5 disp.plot()
6 plt.title('Confusion matrix')
7 plt.show()

Confusion matrix

```

```

+ Code + Text
Linear SVM for MNIST multiclass classification

Using Pipeline

1 pipe_1 = Pipeline([('scaler', MinMaxScaler()),
2                 ('clf', SVC(kernel='linear', C=1))])
3 pipe_1.fit(X_train, y_train)

# Evaluate the model using cross-validation
4 acc = cross_val_score(pipe_1, X_train, y_train, cv=5)
5 print("Training Accuracy: %.2f" % acc)

Training Accuracy: 91.07 %

The transformation is given by:
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
where min, max = feature_range

This estimator scales and translates each feature individually such
that it is in the given range on the training set, e.g. between
zero and one.

# visualising the confusion matrix
6 y_pred = pipe_1.predict(X_test)
7 cm = confusion_matrix(y_test, y_pred)
8 disp = ConfusionMatrixDisplay(cm)
9 disp.plot()
10 plt.title('Confusion matrix')
11 plt.show()

Confusion matrix

```

Now, what we do is we will first use linear SVM for MNIST multi-class classification. So, here we define a pipeline which first perform feature scaling with min max scalar and in the second step we have got SVC SVM classifier, we use linear kernel with the value of C set to 1. We train the pipeline with the training feature matrix and training label vector. We evaluate the model with validation score. So, here we obtain the training accuracy of 91.07.

(Refer Slide Time: 4:10)

```

+ Code + Text

[10] 5 # Evaluate the model using cross-validation
6 acc = cross_val_score(pipe_1, X_train, y_train, cv=5)
7 print("Training Accuracy: %.2f" % acc)

Training Accuracy: 91.07 %

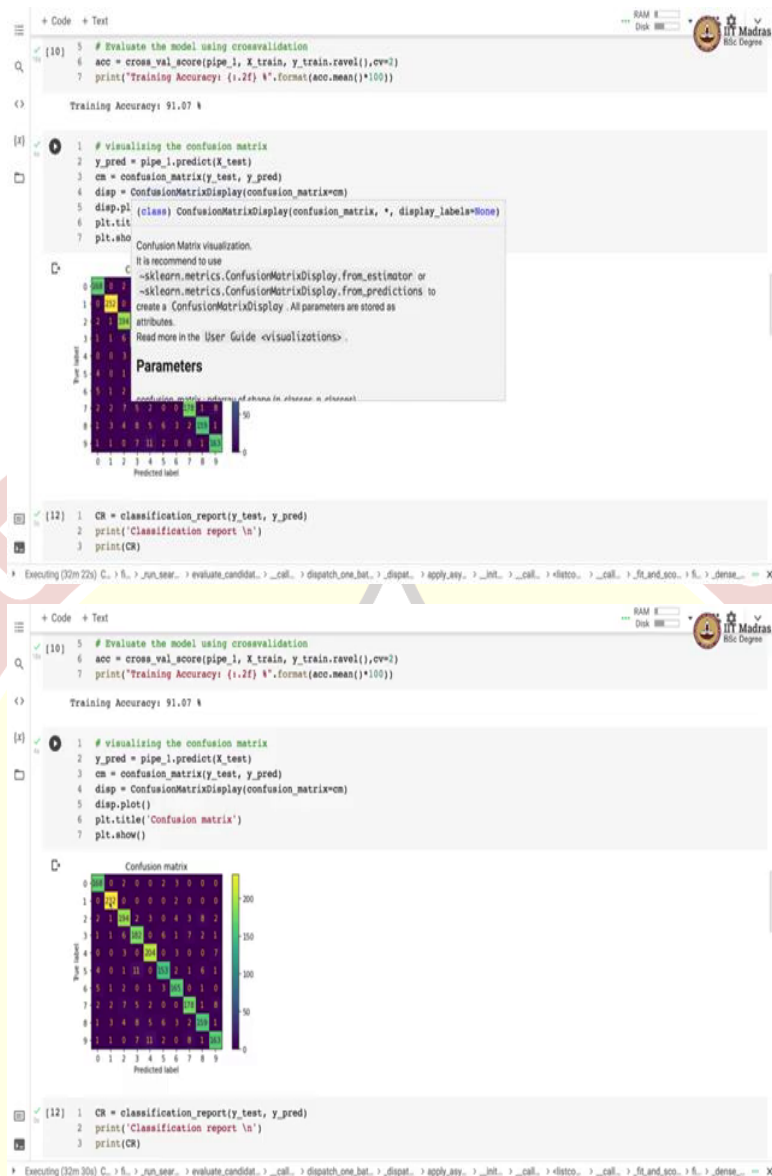
# visualising the confusion matrix
1 y_pred = pipe_1.predict(X_test)
2 cm = confusion_matrix(y_test, y_pred)
3 disp = ConfusionMatrixDisplay(cm)
4 disp.plot()
5 plt.title('Confusion matrix')
6 plt.show()

Confusion matrix

[12] 1 CR = classification_report(y_test, y_pred)
2 print('Classification report\n')
3 print(CR)

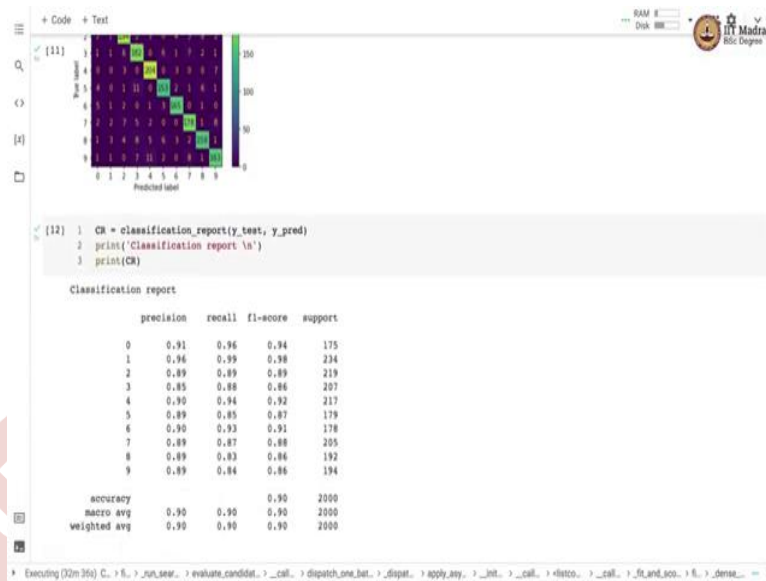
Classification report\n

```

Let us visualize the confusion matrix on the test set. So, for that we first obtain the prediction on the test set and then we get the confusion matrix by comparing the predicted label with the actual labels. Then we display the confusion matrix with confusion matrix display utility. So, this is the confusion matrix on the test set.

(Refer Slide Time: 4:37)



We also obtained classification report that will give us precision, recall, f1-score and accuracy for the test set. So, here you can see that each row has precision, recall and f1-score for each class label. So, for example for 0, the class label 0 we have precision of 0.91, recall of 0.96 and f1-score of 0.94. So, for example, for label 5 we have precision of 0.89, recall of 0.85 and f1-score of 0.87. So, the overall accuracy is 0.90.

(Refer Slide Time: 5:19)

The screenshot shows a Jupyter Notebook interface. The code cell [12] contains the following Python code:

```
accuracy 0.90 2000
macro avg 0.90 0.90 0.90 2000
weighted avg 0.90 0.90 0.90 2000
```

Below the code cell, the notebook title is "Nonlinear SVM for MNIST multiclass classification". Under the heading "Using Pipeline", the code cell [13] contains the following Python code:

```
1 pipe_2 = Pipeline([('scaler', MinMaxScaler()),
2 ('classifier', SVC(kernel = 'rbf', gamma = 0.1, C = 1))])
3 pipe_2.fit(X_train, y_train.ravel())
4
5 # Evaluate the model using crossvalidation
6 acc = cross_val_score(pipe_2, X_train, y_train.ravel(), cv=2)
7 print('Training Accuracy: {:.2f} %'.format(acc.mean()*100))
```

The output of the code is "Training Accuracy: 82.87 %".

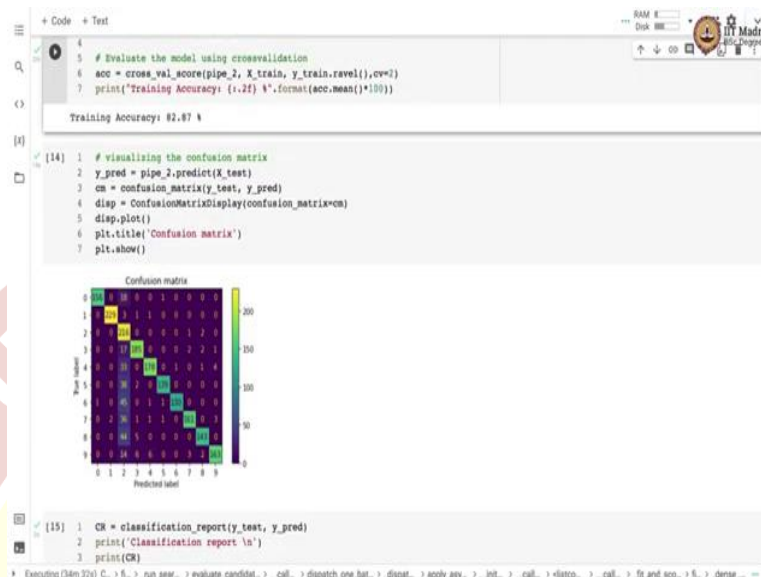
Below the code cell, the notebook title is "Confusion matrix". Under the heading "Confusion matrix", the code cell [14] contains the following Python code:

```
1 # visualizing the confusion matrix
2 y_pred = pipe_2.predict(X_test)
3 cm = confusion_matrix(y_test, y_pred)
4 disp = ConfusionMatrixDisplay(confusion_matrix=cm)
5 disp.plot()
6 plt.title('Confusion matrix')
7 plt.show()
```

Now, what we will do is we will also use non-linear SVM for this multi-class classification problem. So, here for non-linear SVM we use rbf kernel along with the value of gamma = 0.1 and value of C = 1. We again implement SVM with SVC, here the pipeline contains the feature scaling followed by SVC implementing SVM classifier. We train the pipeline with the training

feature matrix and training label vector. We evaluate the model with `val_score` and we obtain the training accuracy of 82.87.

(Refer Slide Time: 6:04)



Let us visualize a confusion matrix on the test set. So, you can see that we have obtained quite low accuracy on the training set itself, so we expect the confusion matrix to contain lot more confusion than the earlier confusion matrix. So, here you can see that the labels which are 4, 5, 6, 7 and 8 they are getting misclassified as 2 and that number is quite significant over here in this confusion matrix.

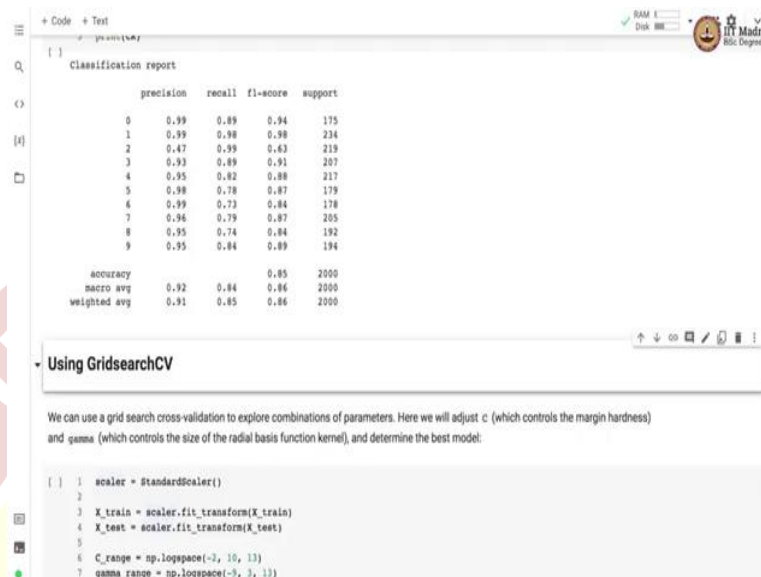
(Refer Slide Time: 6:37)



So, the poor performance of the classifier here is reflected in the classification report the overall accuracy has dropped to 0.85 from 0.90. So, some of the worst predicted classes are like label

2 which has got precision of 0.47, recall of 0.99 and f1-score of 0.63. Again 0 is classified fairly accurately with f1-score of 0.94.

(Refer Slide Time: 7:07)



The screenshot shows a Jupyter Notebook interface. The top part displays a 'Classification report' table. Below it, there is a code cell titled 'Using GridsearchCV' containing Python code for standardizing data and setting up GridSearchCV parameters.

	precision	recall	f1-score	support
0	0.99	0.89	0.94	175
1	0.99	0.98	0.98	234
2	0.47	0.99	0.63	219
3	0.93	0.89	0.91	207
4	0.95	0.82	0.88	217
5	0.99	0.78	0.87	179
6	0.99	0.73	0.84	178
7	0.96	0.79	0.87	205
8	0.95	0.74	0.84	192
9	0.95	0.84	0.89	194
accuracy			0.85	2000
macro avg	0.92	0.84	0.86	2000
weighted avg	0.91	0.85	0.86	2000

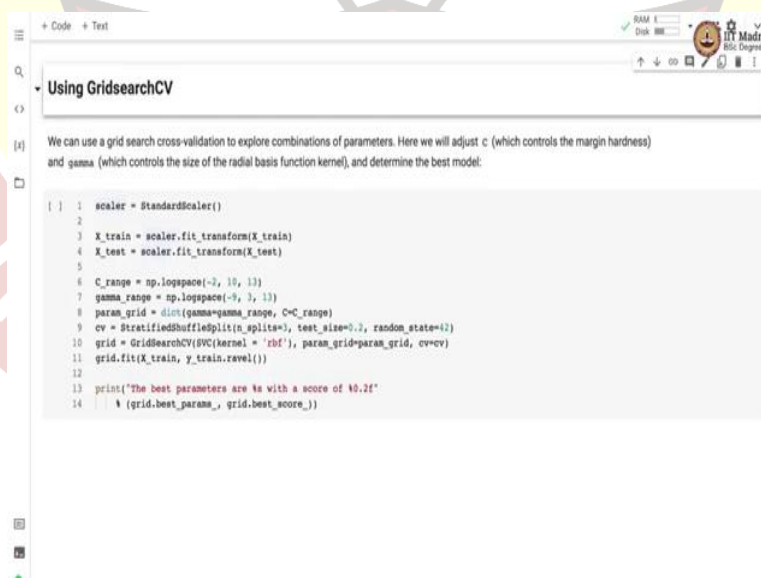
Using GridsearchCV

We can use a grid search cross-validation to explore combinations of parameters. Here we will adjust c (which controls the margin hardness) and γ (which controls the size of the radial basis function kernel), and determine the best model:

```
[ ] 1 scaler = StandardScaler()
2
3 X_train = scaler.fit_transform(X_train)
4 X_test = scaler.fit_transform(X_test)
5
6 C_range = np.logspace(-2, 10, 13)
7 gamma_range = np.logspace(-5, 3, 13)
```

This performance can be fixed by performing a grid search over the parameter of the kernel estimator that we are using. We can adjust C which controls the margin hardness and γ that controls the size of radial basis function kernel.

(Refer Slide Time: 7:26)



The screenshot shows a Jupyter Notebook interface. The top part displays a code cell titled 'Using GridsearchCV' containing Python code for standardizing data and setting up GridSearchCV parameters. The bottom part shows the execution of the code, which prints the best parameters and score.

```
[ ] 1 scaler = StandardScaler()
2
3 X_train = scaler.fit_transform(X_train)
4 X_test = scaler.fit_transform(X_test)
5
6 C_range = np.logspace(-2, 10, 13)
7 gamma_range = np.logspace(-5, 3, 13)
8 param_grid = dict(gamma=gamma_range, C=C_range)
9 cv = StratifiedShuffleSplit(n_splits=5, test_size=0.3, random_state=42)
10 grid = GridSearchCV(SVC(kernel='rbf'), param_grid=param_grid, cv=cv)
11 grid.fit(X_train, y_train.ravel())
12
13 print('The best parameters are %s with a score of %0.2f'
14       % (grid.best_params_, grid.best_score_))
```

So, here what we do is we use the standard scalar as a scaling function and then use an SVC with kernel = rbf. We also define a parameter grid containing the value of γ and the values of c . So, here we have defined a parameter grid for values of C and for values of γ .

And then we have set up the parameter grid as a dictionary for the values of gamma and value of c.

Then we use stratified shuffle split as a way of performing validation. Then we instantiate a grid CV object with the SVC estimator and the parameter grid set to the parameter grid that is instantiated over here along with stratified shuffle split validation. We call the fit function on grid search CV with train feature matrix and train label vector.

The best parameters can be obtained by accessing best underscore param underscore member variable of the grid, grid search CV object and the best score is obtained with best underscore score underscore member variable of the grid search CV object. So, this particular piece of code takes quite a long time to run.

So, as an exercise I would request you to run this code and also plot the confusion matrix and get the classification report on the test set. Then compare the results of this grid search CV estimator with the vanilla SVC estimator with rbf kernel. In this collab we implemented MNIST multi-class classifier with support vector machine.

(Refer Slide Time: 9:21)



```
1 from sklearn import svm, datasets
2
3 X_train = scaler.fit_transform(X_train)
4 X_test = scaler.fit_transform(X_test)
5
6 C_range = np.logspace(-2, 10, 10)
7 gamma_range = np.logspace(-9, 3, 10)
8 param_grid = dict(gamma=gamma_range, C=C_range)
9 cv = StratifiedShuffleSplit(n_splits=10, test_size=0.2, random_state=0)
10 grid = GridSearchCV(SVC(kernel='rbf'), param_grid=param_grid, cv=cv)
11 grid.fit(X_train, y_train.ravel())
12
13 print("The best parameters are %s with a score of %0.2f"
14       % (grid.best_params_, grid.best_score_))
```

Now, you have one more tool in your toolkit to implement classifiers which is using SVM in sklearn.