

IIT Madras

ONLINE DEGREE

Machine Learning Program

Indian Institute of Technology, Madras

Demonstration: KNN with MNIST

(Refer Slide Time: 0:10)

```
+ Code + Text
Imports
• Let's load all usual functionalities and the classifier API XNeighborsClassifier from sklearn.neighbors

[ ] 1 # Common imports
2 import numpy as np
3 from pprint import pprint
4
5 # to make this notebook's output stable across runs
6 np.random.seed(42)
7
8 #sklearn specific imports
9 # Dataset fetching
10 from sklearn.datasets import fetch_openml
11
12 # Feature scaling
13 from sklearn.preprocessing import MinMaxScaler
14
15 # pca
16 from sklearn.decomposition import PCA
17
18 # Pipeline utility
19 from sklearn.pipeline import make_pipeline
20
21 # Classifiers: dummy, KNN
22 from sklearn.dummy import DummyClassifier
23 from sklearn.neighbors import KNeighborsClassifier
24
25 # Model selection
26 from sklearn.model_selection import cross_validate, RandomizedSearchCV, GridSearchCV, cross_val_predict
27 from sklearn.model_selection import learning_curve
28
```

Namaste! Welcome to the next video of Machine Learning Practice Course. In this video, we will use K-nearest neighbor classifier for handwritten digit recognition. Let us first import all usual functionalities, and the classifier API, K-neighbors classifier from sklearn.neighbors module.

(Refer Slide Time: 0:30)

```
+ Code + Text
Handwritten Digit classification
Dataset:
• Once again, we are going to use MNIST
• Each datapoint is contained in  $x_i \in \mathbb{R}^{784}$  and the label  $y_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ 

[ ] 1 x, y = fetch_openml('mnist_784', version=1, return_X_y=True)

2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

Binary Classification
Change Labels to Binary
• Let us do binary classification with KNN classifier and eventually extend it to Multiclass set-up

[ ] 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')
```

So, for handwritten digit recognition, we are once again going to use MNIST dataset. And you are aware that each point in MNIST dataset is represented with 784 numbers. And there are 10 labels, 0 to 9. We first fetch the MNIST dataset, divide that into training and test set.

(Refer Slide Time: 0:53)



```
[ ] 2 y = y.to_numpy()

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

Binary Classification

Change Labels to Binary

• Let us do binary classification with KNN classifier and eventually extend it to Multiclass set-up

[ ] 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')
7 y_train_0[indx_0] = 1
8 indx_0 = np.where(y_test == '0')
9 y_test_0[indx_0] = 1

Data Visualization in Lower dimension

• Let us apply PCA on the datapoints and reduce the dimensions to 2D and then to 3D.
• This will give us some rough idea about the points in  $R^{784}$ .
• One interesting thing to look at is the change in the magnitude of the data points before and after applying PCA.
• we use the variables pipe_pca_2d for pre-processing the samples alone and pipe_clf_pca_2d for classification.

[ ] 1 pipe_pca_2d = make_pipeline(MinMaxScaler(), PCA(n_components=2))
```

First, we will solve a binary classification problem for detecting 0 digit from all other digits. So, for binary classification problem, we first create a training data with labels 0 and 1, the label 1 is assigned to the digit 0 and rest of the digits get labeled 0.

(Refer Slide Time: 01:12)



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

Binary Classification

Change Labels to Binary

• Let us do binary classification with KNN classifier and eventually extend it to Multiclass set-up

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')
7 y_train_0[indx_0] = 1
8 indx_0 = np.where(y_test == '0')
9 y_test_0[indx_0] = 1

Data Visualization in Lower dimension

• Let us apply PCA on the datapoints and reduce the dimensions to 2D and then to 3D.
• This will give us some rough idea about the points in  $R^{784}$ .
• One interesting thing to look at is the change in the magnitude of the data points before and after applying PCA.
• we use the variables pipe_pca_2d for pre-processing the samples alone and pipe_clf_pca_2d for classification.

[ ] 1 pipe_pca_2d = make_pipeline(MinMaxScaler(), PCA(n_components=2))
2 x_train_pca_2d = pipe_pca_2d.fit_transform(x_train)
```



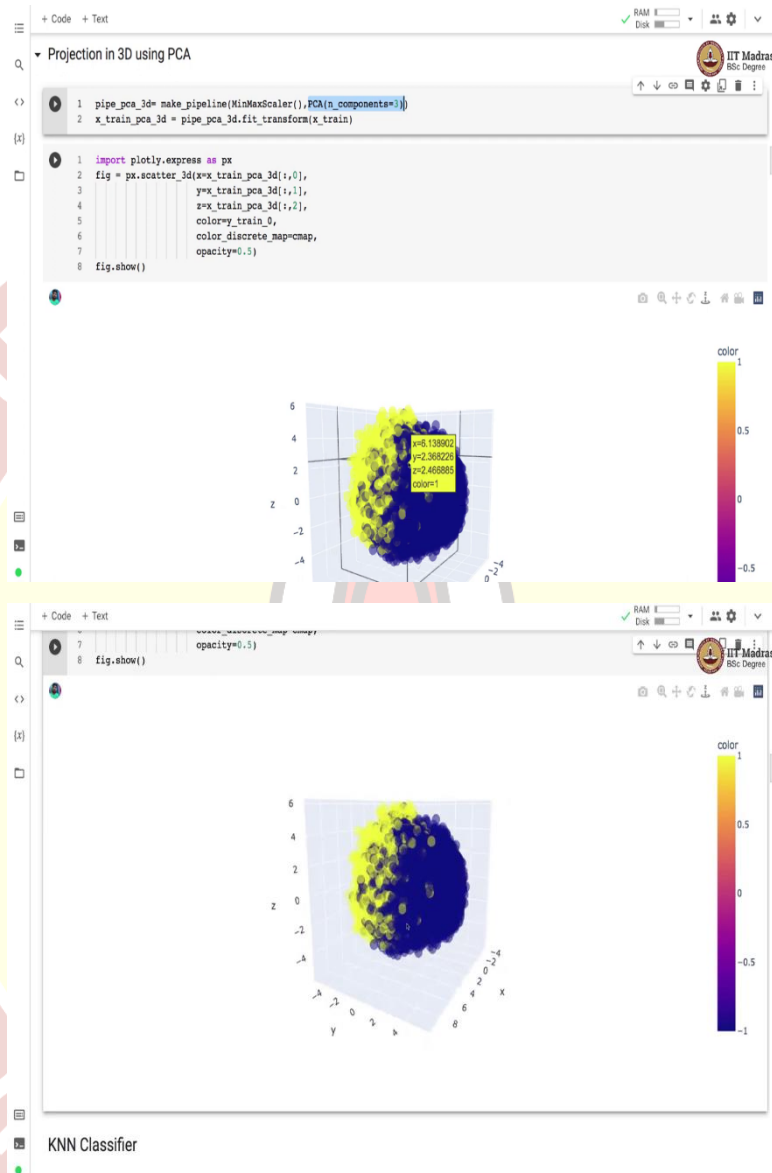
We will first apply Principal Component Analysis on the data points and reduce the dimensions from 784 to 2, and then to 3. This will give us a rough idea about how power points are located in the original feature space by looking at the visualization in the projective space. One interesting thing to look at is the change in the magnitude of the data points before and after applying PCA.

So, here, we make a pipeline, which first scales the features, and then we apply PCA with number of components equal to 2, then we call the fit transform method by passing the feature matrix and we get the representation in the principal component space. Here we have plotted the points in the principal component space.

The blue points corresponds to the positive class and the red points corresponds to the negative class. In our case, the positive class is the digit 0 and negative class is rest of the digits. So, you

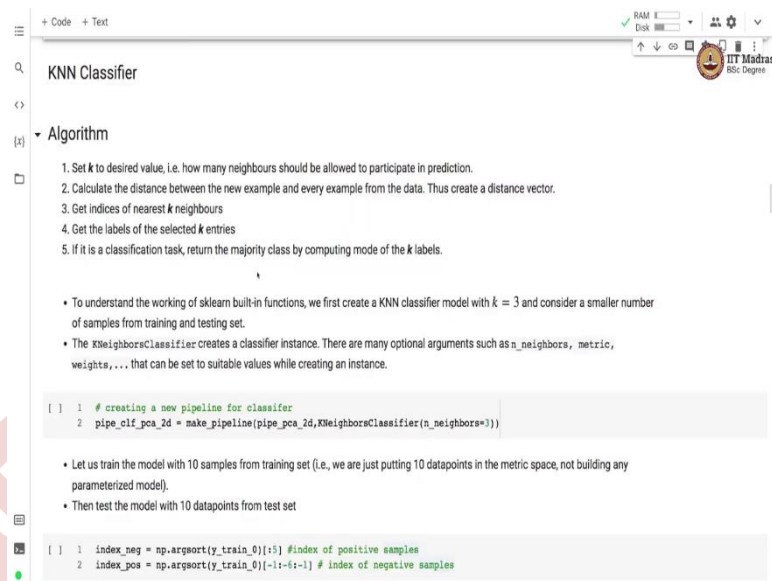
can see that there is some separation between the classes, but there are some images of 0 that are very close to the non-zero images.

(Refer Slide Time: 02:38)



We repeat the same analysis with three principal components. We modify our pipeline to perform PCA with 3 components. After performing PCA, we visualize the training data in the projected 3D space. Here is an interactive visualization. So, you can play with this visualization and try to view the points from different perspective.

(Refer Slide Time: 03:12)



The screenshot shows a Jupyter Notebook interface with a title bar 'KNN Classifier' and a toolbar. The left sidebar has icons for file operations and search. The main area is titled 'Algorithm' and contains a list of five steps for the K-Nearest Neighbors algorithm. Below the steps, there are two code cells. The first code cell creates a pipeline for a KNeighborsClassifier with n_neighbors=3. The second code cell trains the model with 10 samples from the training set and tests it with 10 datapoints from the test set.

```
[ ] 1 # creating a new pipeline for classifier
2 pipe_clf_pca_2d = make_pipeline(pipe_pca_2d, KNeighborsClassifier(n_neighbors=3))

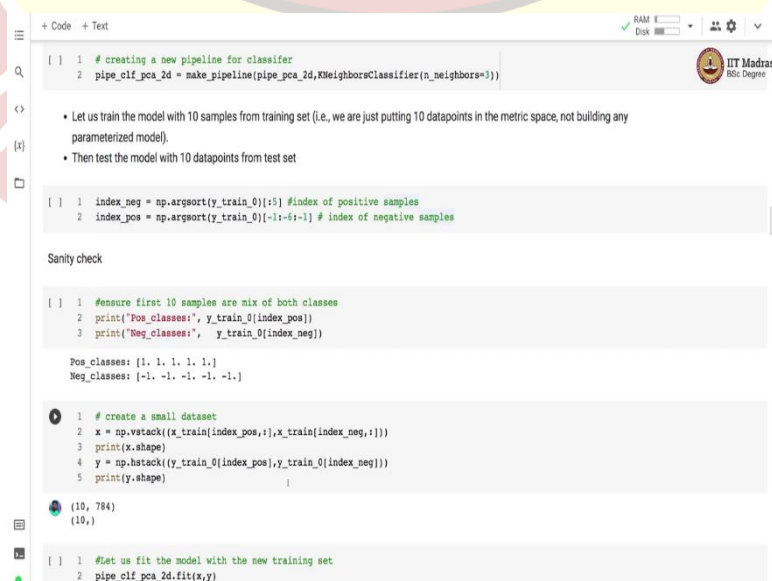
[ ] 1 Let us train the model with 10 samples from training set (i.e., we are just putting 10 datapoints in the metric space, not building any
2 parameterized model).
3 Then test the model with 10 datapoints from test set

[ ] 1 index_neg = np.argsort(y_train_0)[:5] # index of positive samples
2 index_pos = np.argsort(y_train_0)[-1:-5:-1] # index of negative samples
```

As the next step, let us train a K-NN classifier and you know the algorithm K-NN, we first set k to the desired value, which is how many neighbors should be allowed to participate in prediction, we calculate the distance between the new example and every example in the training data. And thus, we create a distance vector, we get the indices of nearest k -neighbors, get the labels of selected k -neighbors.

And if it is a classification task, we return the majority class by computing the mode of the k labels. To understand the working of sklearn built-in function, we first create a K-NN classifier model with k equal to 3 and with a small amount of training and test data.

(Refer Slide Time: 04:01)



The screenshot shows a Jupyter Notebook interface with a title bar 'KNN Classifier' and a toolbar. The left sidebar has icons for file operations and search. The main area contains two code cells. The first code cell trains the model with 10 samples from the training set and tests it with 10 datapoints from the test set. The second code cell performs a sanity check by printing the first 10 samples and their labels. Below the code cells, there is a 'Sanity check' section with a code cell that prints the first 10 samples and their labels. The output shows the first 10 samples and their labels, indicating a mix of both classes.

```
[ ] 1 # creating a new pipeline for classifier
2 pipe_clf_pca_2d = make_pipeline(pipe_pca_2d, KNeighborsClassifier(n_neighbors=3))

[ ] 1 Let us train the model with 10 samples from training set (i.e., we are just putting 10 datapoints in the metric space, not building any
2 parameterized model).
3 Then test the model with 10 datapoints from test set

[ ] 1 index_neg = np.argsort(y_train_0)[:5] # index of positive samples
2 index_pos = np.argsort(y_train_0)[-1:-5:-1] # index of negative samples

Sanity check

[ ] 1 # ensure first 10 samples are mix of both classes
2 print("Pos_classes:", y_train_0[index_pos])
3 print("Neg_classes:", y_train_0[index_neg])

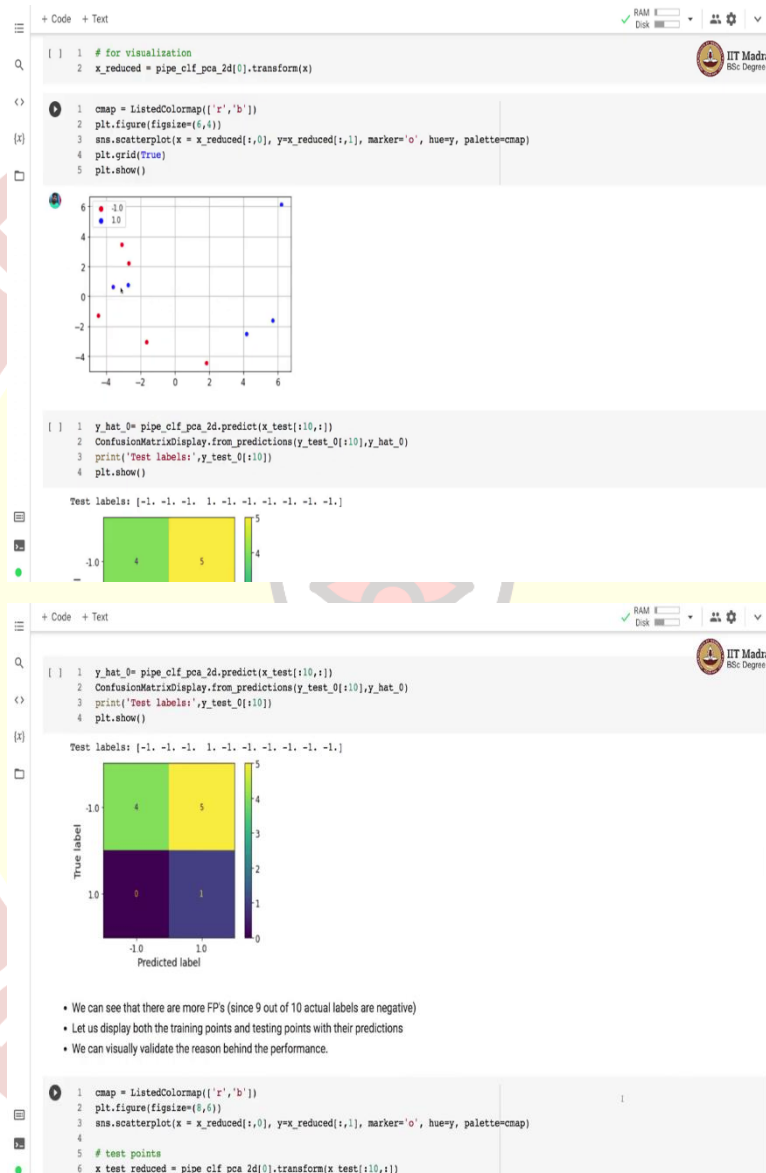
Pos_classes: [1. 1. 1. 1. 1.]
Neg_classes: [-1. -1. -1. -1. -1.]

[ ] 1 # create a small dataset
2 x = np.vstack((x_train[index_pos,:], x_train[index_neg,:]))
3 print(x.shape)
4 y = np.hstack((y_train_0[index_pos], y_train_0[index_neg]))
5 print(y.shape)

(10, 784)
(10,)
```

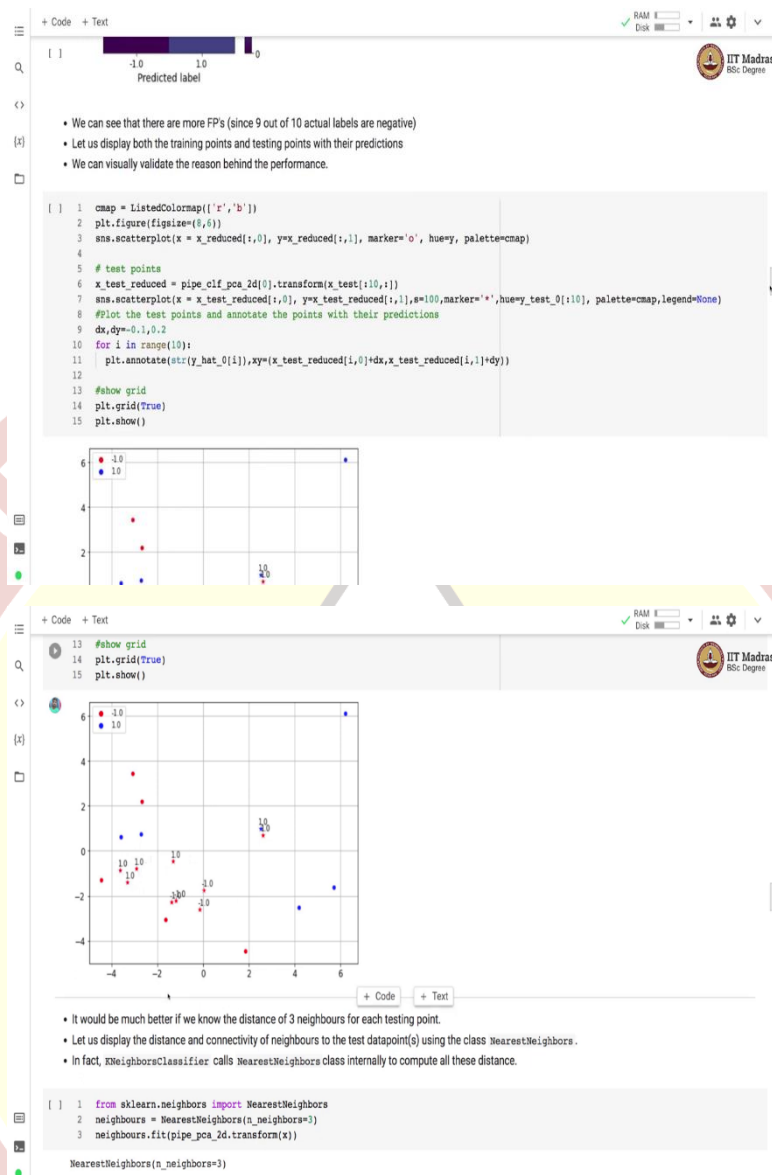

Let us train the model with 10 samples from the training set. And we will test it with another 10 samples from the test set. So, here we create data containing positive and negative examples, there are exactly 10 samples in the training set and each sample is represented with 784 numbers.

(Refer Slide Time: 4:21)



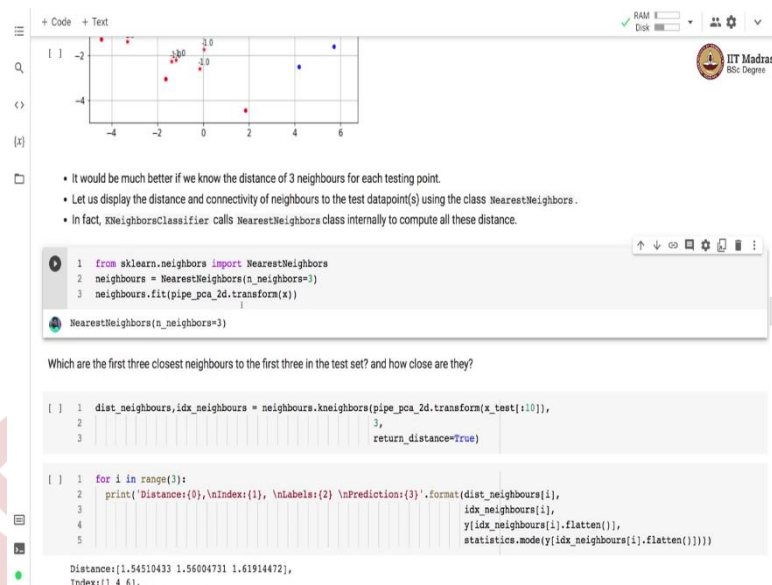
There are 10 points that we have displayed here which contains 5 points from the positive class and 5 points from the negative class. And now, what we do is we perform prediction on the test set and display the confusion matrix. So, you can see that there are 9 points out of 10 that are labeled as negative. So, the points that are actually belonging to the negative class are labeled as positive and these 5 points are misclassified. So, let us display both training point and test points along with the predictions.

(Refer Slide Time: 05:04)



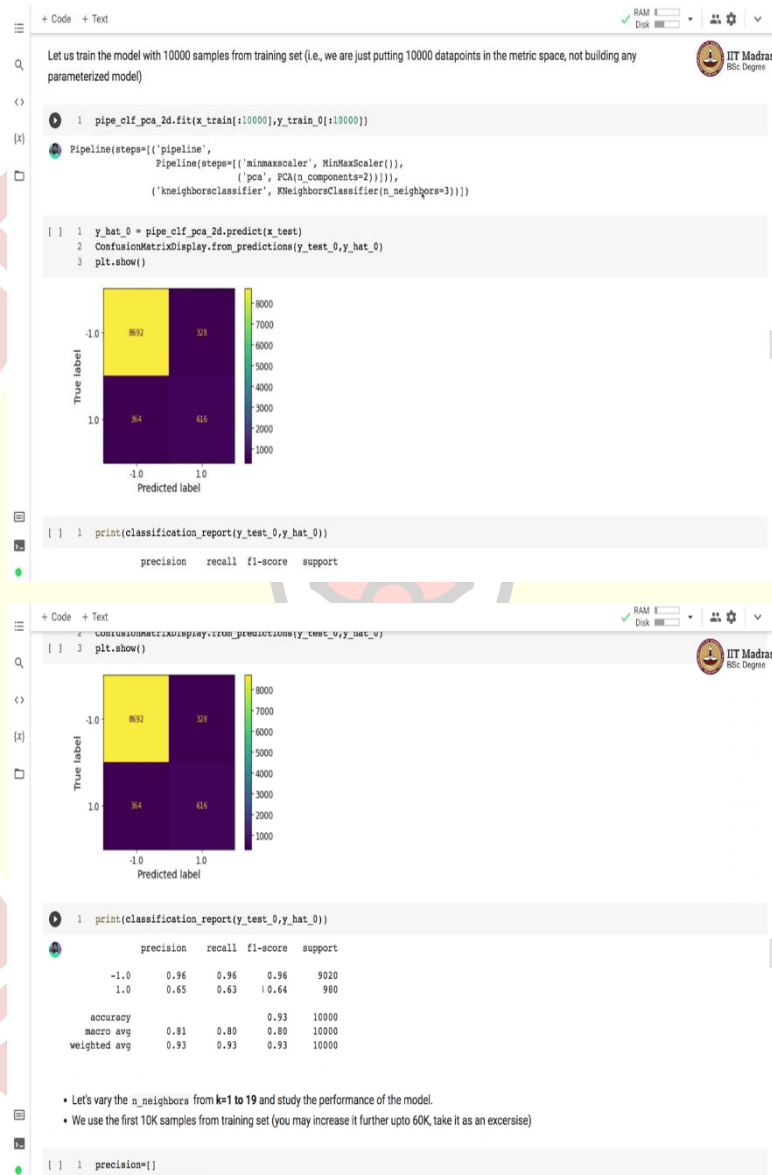
So, that will help us to visualize the reason for this performance. So, here you can see that the test points are marked with star along with their labels.

(Refer Slide Time: 05:20)



We print the distances to the 3 nearest neighbors along with the indices and labels. We also print the prediction that we obtained from the labels of the nearest neighbors. So, you can see that in the first example, the class 1 is the dominant class, hence we assign the label 1 to the test example.

(Refer Slide Time: 06:32)

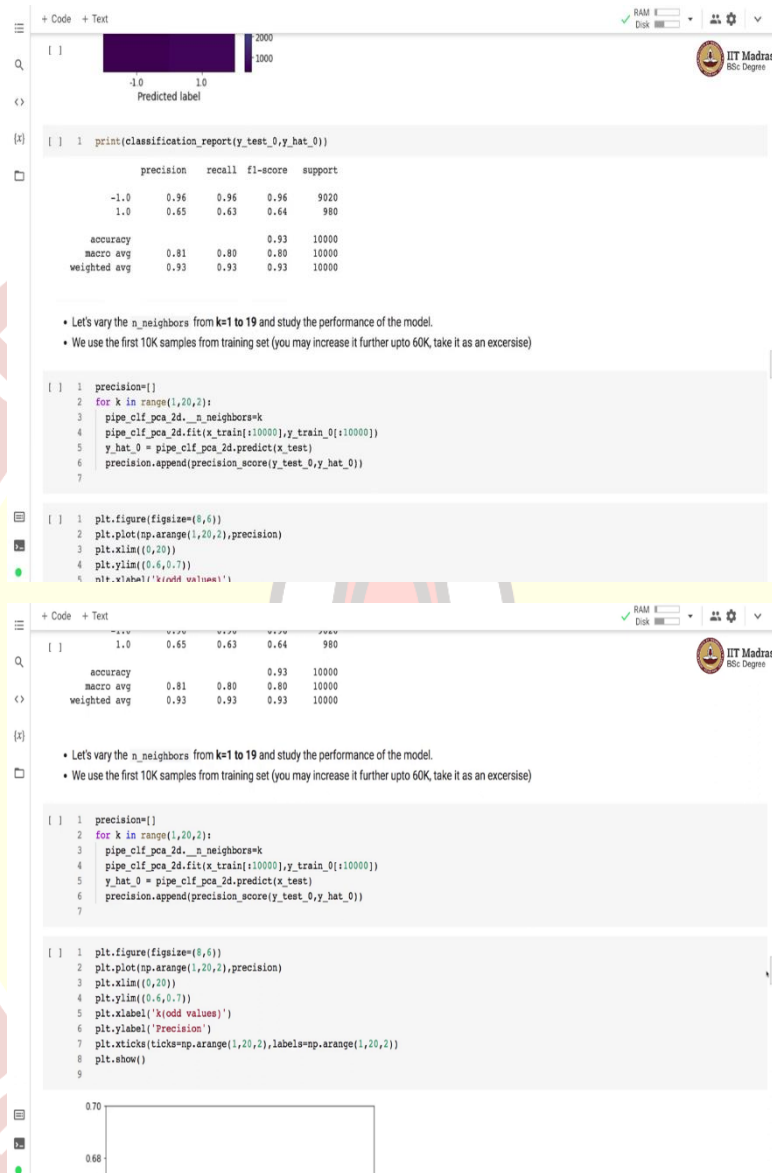


Let us train the model with 10,000 samples from the training set. So, here we use a pipeline that contains the scaling operation followed by Principal Component Analysis, and then the K-neighbors classifier. So, here, we transform each example in the principal component space with 2 components.

So, we make the prediction on the test set, and we have displayed a confusion matrix on the screen. We also get the classification report. And in the classification report, we can observe

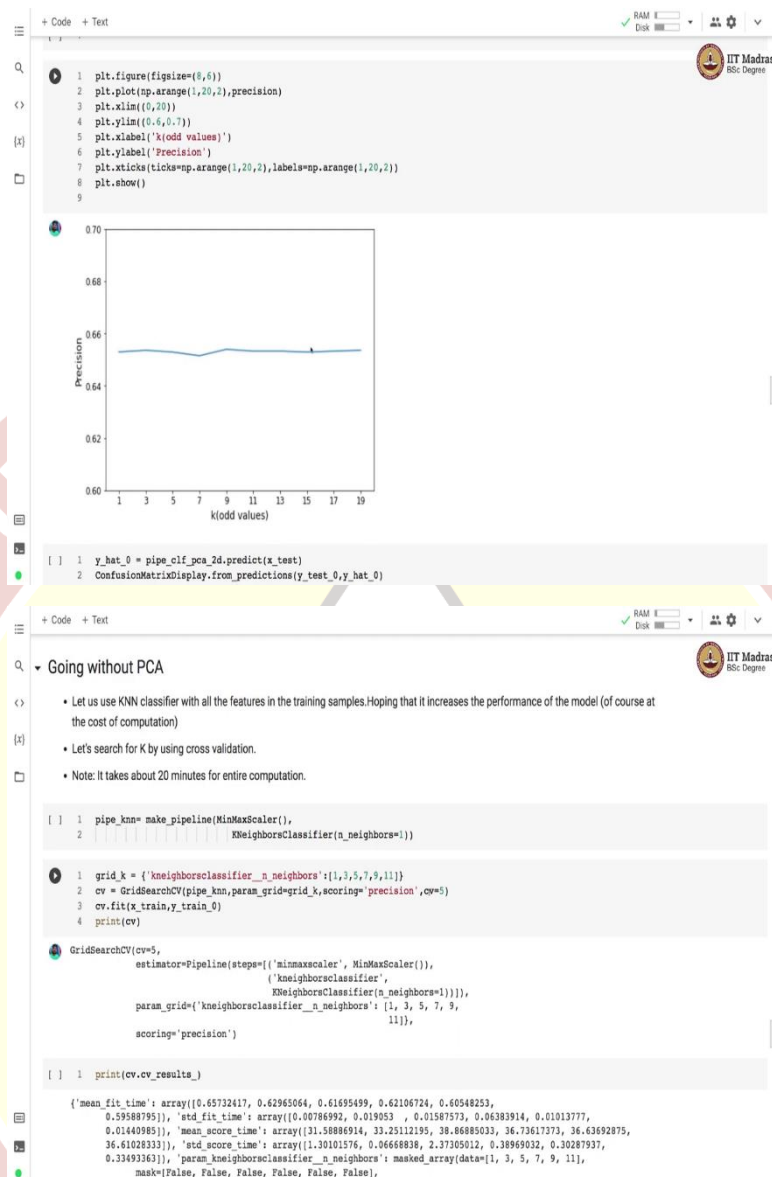
that for class 1, the precision is 0.65. And recall is 0.63 and f1-score is 0.64, which is far lesser than what we have obtained earlier.

(Refer Slide Time: 07:27)



So, now in order to tune the number of neighbors, we vary the number of neighbors from 1 to 19 and study the performance of the model. For this study, we are selecting first 10,000 examples in the training set, we can also increase it up to 60k. And for each iteration, we store the precision of the model.

(Refer Slide Time: 07:57)



So, here we have plotted the precision for different values of k. And you can see that the precision is around 0.65. Let us use K-NN classifier with all the features in the training samples. We hope that it will increase the performance of the model, of course, it will increase the amount of computation for calculating the distance to the nearest neighbors.

So, you search k using cross-validation. So, here we define a pipeline that contains a feature scaling followed by the nearest neighbor classifier. We perform GridSearchCV for finding the optimal value of k and here we specify the number of neighbors which is k to 1, 3, 5, 7, 9 and 11. We are going to score the search using precision and we are using 5 fold cross-validation.

(Refer Slide Time: 09:03)

```
+ Code + Text
[ ] 4 print(cv)

GridSearchCV(cv=5,
              estimator=Pipeline(steps=[('minmaxscaler', MinMaxScaler()),
                                         ('kneighborsclassifier',
                                          KNeighborsClassifier(n_neighbors=1))]),
              param_grid=[('kneighborsclassifier_n_neighbors': [1, 3, 5, 7, 9,
                                                                11]),
                          ],
              scoring='precision')

[ ] 1 print(cv.cv_results_)

{'mean_fit_time': array([0.65732417, 0.62965064, 0.61695499, 0.62106724, 0.60548253,
                          0.59588795]), 'std_fit_time': array([0.00786992, 0.019053 , 0.01587573, 0.06383914, 0.01013777,
                          0.01440985]), 'mean_score_time': array([31.58886914, 33.25112195, 38.86885033, 36.73617373, 36.63692875,
                          36.61028333]), 'std_score_time': array([1.30101576, 0.06668838, 2.37805012, 0.38969032, 0.30287937,
                          0.33493363]), 'param_kneighborsclassifier_n_neighbors': masked_array(data=[1, 3, 5, 7, 9, 11],
                                         mask=[False, False, False, False, False, False],
                                         fill_value='?'),
 'dtype=object', 'params': [{'kneighborsclassifier_n_neighbors': 1}, {'kneighborsclassifier_n_neighbors': 3}, {'kneighborsclas
0.97921862}], 'split1_test_score': array([0.96954733, 0.97761194, 0.97676349, 0.97838736, 0.97918401,
0.97831526]), 'split2_test_score': array([0.9775 , 0.98076923, 0.98238255, 0.98484848, 0.98313659,
0.98316498]), 'split3_test_score': array([0.98073702, 0.98155909, 0.98569024, 0.98403361, 0.98154362,
0.98402019]), 'split4_test_score': array([0.98329156, 0.98657718, 0.98411371, 0.98414023, 0.9825 ,
0.97920133]), 'mean_test_score': array([0.97805545, 0.98229094, 0.98212333, 0.98245498, 0.98111657,
0.98078408]), 'std_test_score': array([0.00466085, 0.00316404, 0.00302285, 0.00245495, 0.001644 ,
0.00233203]), 'rank_test_score': array([6, 2, 3, 1, 4, 5], dtype=int32))

• The best value for k is 7

[ ] 1 pipe_knn= make_pipeline(MinMaxScaler(),
2                               KNeighborsClassifier(n_neighbors=7))
3
```

```
+ Code + Text
Performance on test set
1 y_hat_0 = pipe_knn.predict(x_test)
2 ConfusionMatrixDisplay.from_predictions(y_test_0,y_hat_0)
3 plt.show()

True label
-1.0 3592 28
1.0 7 971
Predicted label
-1.0 1.0

Multi-Class Classification
• Extending KNN classifier to multi-class problem is quite straightforward.

[ ] 1 print(pipe_knn)

Pipeline(steps=[('minmaxscaler', MinMaxScaler()),
                 ('kneighborsclassifier', KNeighborsClassifier(n_neighbors=7))])
```

After performing the grid search, we found that the best value of k is 7. Let us evaluate the performance of the K-NN pipeline on the test set. Here we have displayed a confusion matrix and we found that there are 35 examples that are misclassified from the test set.

(Refer Slide Time: 09:24)

```
+ Code + Text
Multi-Class Classification
• Extending KNN classifier to multi-class problem is quite straightforward.

[ ] 1 print(pipe_knn)

Pipeline(steps=[('minmaxscaler', MinMaxScaler()),
                 ('kneighborsclassifier', KNeighborsClassifier(n_neighbors=7))])

1 pipe_knn.fit(x_train,y_train)
2 y_hat = pipe_knn.predict(x_test)
3 ConfusionMatrixDisplay.from_predictions(y_test,y_hat)
4 plt.show()

True label
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
Predicted label

[ ] 1 pipe_knn.classes_

array(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'], dtype=object)
```

Extending K-NN to multiclass problem is quite straightforward. So, here we define a K-NN pipeline with the first stage as the MinMaxScaler. And the second stage, we have KNeighborsClassifier with number of neighbors equal to 7. We perform the fit by supplying the training feature metrics and labels. And then, we perform the prediction on the test set.

And here is the confusion matrix that we obtain from predictions of the test samples. And you can see that the confusion matrix has much lesser number of confusions. So, there are usual confusions between a label 8 and label 5 or label 3 and label 5 or label 4 and label 9.

(Refer Slide Time: 10:24)

```
+ Code + Text
[ ] 9 5 4 3 6 4 1 0 7 0
Predicted label

[ ] 1 pipe_knn.classes_

array(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'], dtype=object)

1 print(classification_report(y_test,y_hat))

precision recall f1-score support

0 0.97 0.99 0.98 980
1 0.95 1.00 0.97 1135
2 0.98 0.96 0.97 1032
3 0.97 0.97 0.97 1010
4 0.98 0.96 0.97 982
5 0.97 0.97 0.97 892
6 0.98 0.99 0.98 958
7 0.96 0.96 0.96 1028
8 0.99 0.94 0.96 974
9 0.96 0.95 0.96 1009

accuracy 0.97 10000
macro avg 0.97 0.97 0.97 10000
weighted avg 0.97 0.97 0.97 10000
```

Let us print the classification report and we have very, very interesting f1-scores for our problem. So, we have all the f1-scores greater than 0.95 which is what we have not experienced

in our earlier methods. So, here we have the highest f1-score of 0.98 for digit 0, digit 6, then we have bunch of f1-scores in the range of 0.96 and 0.97. So, the lowest f1-score is 0.96, so the overall accuracy is 97% and macro average, precision, recall and f1-scores are 0.97. So, in this video, we demonstrated how to use K-NN and for a real live classification problem.

