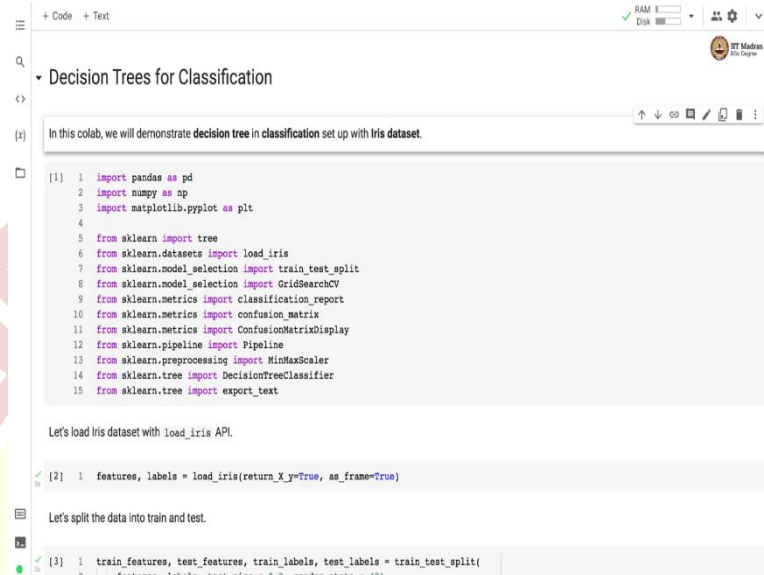


IIT Madras

ONLINE DEGREE

Machine Learning Practice
Professor Dr. Ashish Tendulkar
Indian Institute of Technology, Madras
Decision Trees for Classification – Iris

(Refer Slide Time: 00:10)



```
[1] 1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 from sklearn import tree
6 from sklearn.datasets import load_iris
7 from sklearn.model_selection import train_test_split
8 from sklearn.model_selection import GridSearchCV
9 from sklearn.metrics import classification_report
10 from sklearn.metrics import confusion_matrix
11 from sklearn.metrics import ConfusionMatrixDisplay
12 from sklearn.pipeline import Pipeline
13 from sklearn.preprocessing import MinMaxScaler
14 from sklearn.tree import DecisionTreeClassifier
15 from sklearn.tree import export_text

Let's load Iris dataset with load_iris API.

[2] 1 features, labels = load_iris(return_X_y=True, as_frame=True)

Let's split the data into train and test.

[3] 1 train_features, test_features, train_labels, test_labels = train_test_split(
2     features, labels, test_size = 0.7, random_state = 42)
```

Namaste! Welcome to the next video of Machine Learning Practice Course. In this collab, we will demonstrate decision tree in classification setup with Iris dataset. Let us first import some of the useful libraries like pandas and NumPy and matplotlib.pyplot for plotting certain figures.

Then since we are going to use decision trees, we import trees from sklearn and DecisionTreeClassifier from sklearn.tree module as well as export_text. Then there are some other usual libraries like train_test_split, GridSearchCV, classification_report, confusion_matrix, ConfusionMatrixDisplay, pipeline, MinMaxScaler are also loaded. Since we are going to use Iris dataset, we have load_iris API imported from sklearn.datasets module.

(Refer Slide Time: 01:20)

```
+ Code + Text
Let's load Iris dataset with load_iris API.
1 features, labels = load_iris(return_X_y=True, as_frame=True)

Let's split the data into train and test.
(3) 1 train_features, test_features, train_labels, test_labels = train_test_split(
    2     features, labels, test_size = 0.2, random_state = 42)

Define the decision tree classifier as part of pipeline.
(4) 1 dt_pipeline = Pipeline(steps=[('feature_scaling', MinMaxScaler()),
    2     ('dt_classifier',
    3         DecisionTreeClassifier(max_depth=3,
    4             random_state = 42))])

Train the classifier.
(5) 1 dt_pipeline.fit(train_features, train_labels)

Now that the classifier is trained, let's evaluate it on the test set with
• Confusion matrix
• Classification report
```

So, the first step let us load Iris dataset with load_iris API. So, load_iris takes two argument, one is return_x_y, we have set that flag to true and as frame flag also to true. So, what happens is because of this flag, we get the data in form of a feature matrix and label vector.

And since we have set as_frame = true, we get this data in form of a data frame. As the next step, we split this data into train and test. So, for that, we set aside 20% example for test. And we use train_test_split API for this particular splitting. So, we get training and test feature matrix as well as training and test label vectors.

(Refer Slide Time: 02:23)

```
+ Code + Text
Let's split the data into train and test.
(3) 1 train_features, test_features, train_labels, test_labels = train_test_split(
    2     features, labels, test_size = 0.2, random_state = 42)

Define the decision tree classifier as part of pipeline.
(4) 1 dt_pipeline = Pipeline(steps=[('feature_scaling', MinMaxScaler()),
    2     ('dt_classifier',
    3         DecisionTreeClassifier(max_depth=3,
    4             random_state = 42))])

Train the classifier.
(5) 1 dt_pipeline.fit(train_features, train_labels)

Now that the classifier is trained, let's evaluate it on the test set with
• Confusion matrix
• Classification report

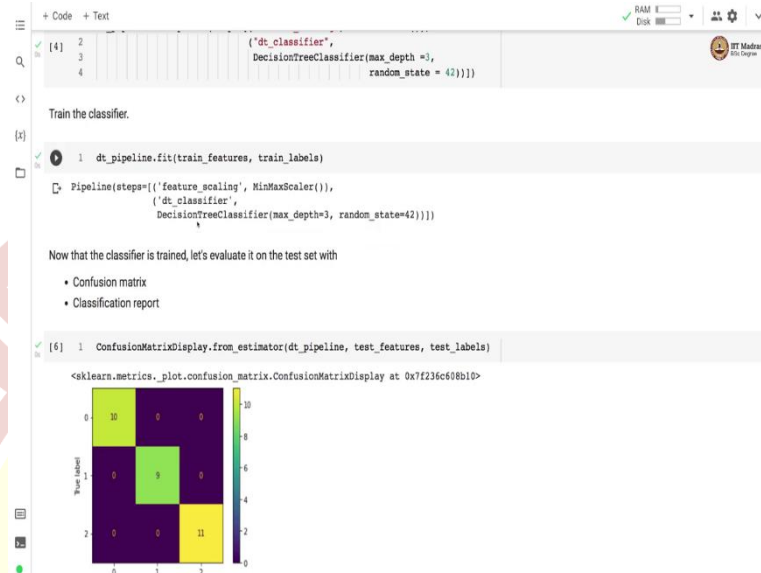
(6) 1 ConfusionMatrixDisplay.from_estimator(dt_pipeline, test_features, test_labels)

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f236c608b10>
```

As a next step, we define the DecisionTreeClassifier as part of the pipeline. And in pipeline, there are two stages. The first stage is feature scaling. For that we use MinMaxScaler. And

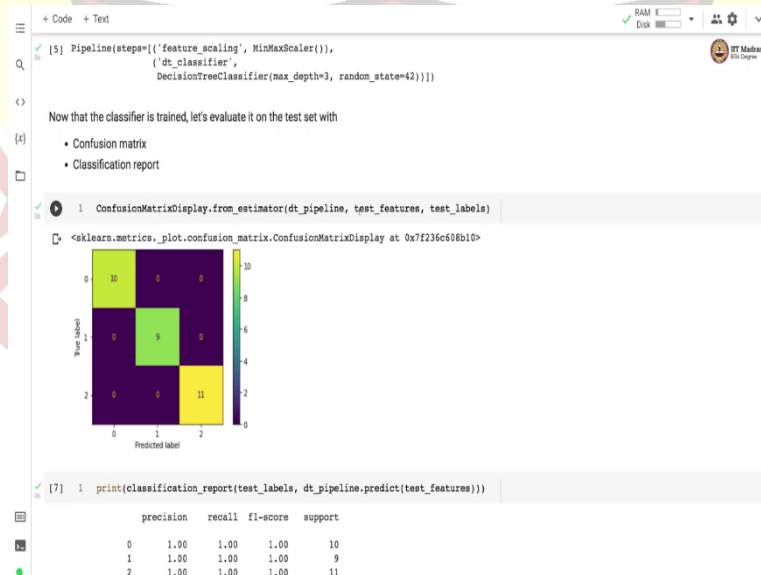
then the second stage is DecisionTreeClassifier. So, we instantiate a DecisionTreeClassifier with `max_depth = 3`. And then we train the classifier by calling the Fit function on the pipeline object by supplying the training feature matrix and label vector.

(Refer Slide Time: 03:00)



So, after the step, the classifier is trained, and we will evaluate it on the test set with confusion matrix and classification report.

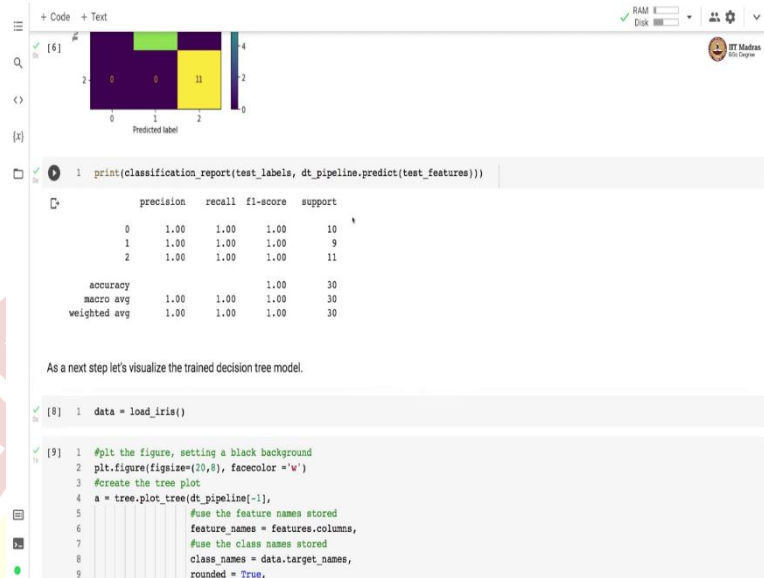
(Refer Slide Time: 03:12)



So, we use ConfusionMatrixDisplay API, and we call from `_estimator` method on this API. We supply the name of the estimator, the test feature matrix and label vectors. And it prints the confusion matrix or other displays the confusion matrix in a colourful format. So, we have two labels on the y-axis and predicted labels on the x-axis. And you can see that it has

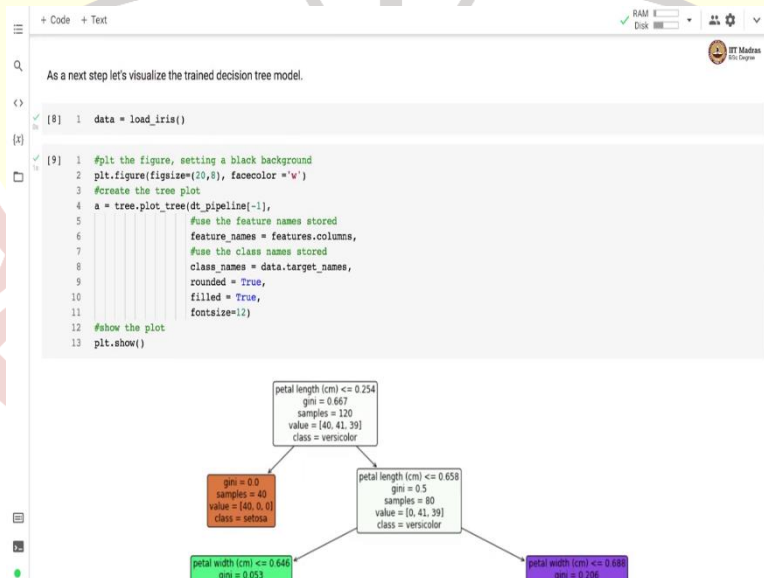
managed to predict all the labels correctly. There are 0's in all of diagonal entries, which means there is no miss classification whatsoever there is happening for any of the classes.

(Refer Slide Time: 04:04)



Let us look at the classification _report. And in classification _report you can see that the precision and recall is 1 for each of the class on the test set.

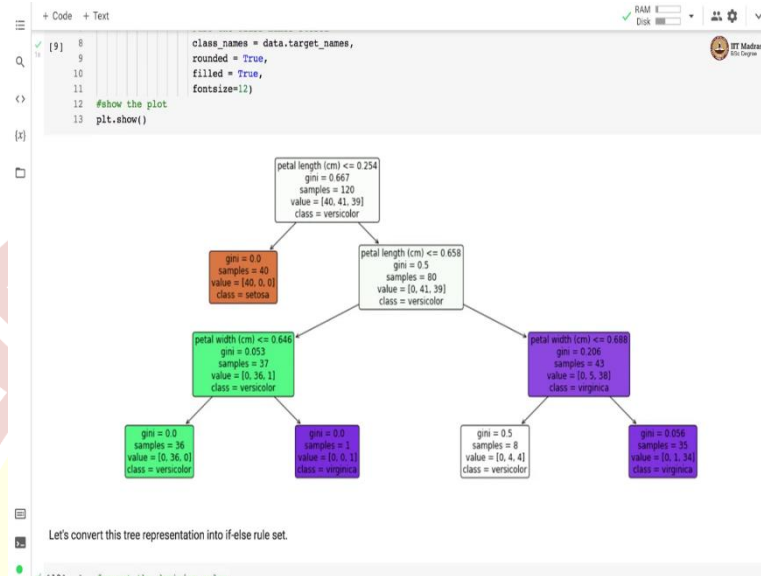
(Refer Slide Time: 04:22)



So, since we are using decision trees, we can visualize the decision tree and understand how the classifier has worked. So, because the tree visualization needs, the names of the classes, which we do not get because we loaded our data in from data frame. Here, we again call `load_iris` and store the data in the form of a bunch object.

Now what we do here is for plotting the tree we call `plot_tree` method on the tree object. And here we specify the name of the decision tree estimator or rather the object of the decision tree estimator, we specify the `feature_names` as well as `class_names`.

(Refer Slide Time: 05:16)



And here you can see the decision tree. The top-level node is based on the feature called petal length, which is in centimetres and the petal length is less than 0.254. Then it classifies it as a class Setosa. And then it basically on the right-hand side, we again check on the petal length, then here in on petal width and here again on petal width. So, on the third level, we are checking things on petal width to decide the split in the decision tree.

And you can see that the leaf nodes, we are specifying we are predicting class Setosa on this leaf node is a class versicolor here, there is class virginica here and there is versicolor and virginica over here. So, we have also printed gini index as well as number of samples in the training set, and we also get to see the split by the classes.

So, here you can see that gini index is 0, and here all examples belong to Setosa. Here, versicolor is in majority, which is printed over here. And then we get pretty much most of the examples from versicolor and here all of the examples are from versicolor. So, what decision tree does is it keeps on refining the regions such that there is no impurity in any of the regions.

(Refer Slide Time: 07:09)

```
+ Code + Text
Let's convert this tree representation into if-else rule set.

1 #export the decision rules
2 tree_rules = export_text(dt_pipeline[-1],
3                           feature_names = list(features.columns))
4 #print the result
5 print(tree_rules)

[10] --- petal length (cm) <= 0.25
      --- class: 0
      --- petal length (cm) > 0.25
            --- petal length (cm) <= 0.66
                  --- petal width (cm) <= 0.65
                        --- class: 1
                  --- petal width (cm) > 0.65
                        --- class: 2
            --- petal length (cm) > 0.66
                  --- petal width (cm) <= 0.69
                        --- class: 1
                  --- petal width (cm) > 0.69
                        --- class: 2

Let's get the feature importance from the trained decision tree model.

[11] 1 #extract importance
2 importance = pd.DataFrame({'feature': features.columns,
3                           'importance': np.round(
4                               dt_pipeline[-1].feature_importances_, 3)})
5 importance.sort_values('importance', ascending=False, inplace = True)
6 print(importance)

feature importance
2 petal length (cm) 0.935
```

We can convert this tree representation into if-else rule set for that, we use `export_text` method by supplying the tree estimator and the list of feature _names. And here, we see if-else rules. So, we have based on petal length, we are specifying the class 0, we are deciding the class 0. If this is not true, then we go for another check on petal length and so on. So, this is basically the rule set that we can use to classify the iris flowers. And is this rule set is learned from the DecisionTreeClassifier or decision tree estimator.

(Refer Slide Time: 08:01)

```
+ Code + Text
[10] --- petal width (cm) <= 0.69
      --- class: 1
      --- petal width (cm) > 0.69
            --- class: 2

Let's get the feature importance from the trained decision tree model.

[11] 1 #extract importance
2 importance = pd.DataFrame({'feature': features.columns,
3                           'importance': np.round(
4                               dt_pipeline[-1].feature_importances_, 3)})
5 importance.sort_values('importance', ascending=False, inplace = True)
6 print(importance)

feature importance
2 petal length (cm) 0.935
3 petal width (cm) 0.065
0 sepal length (cm) 0.000
1 sepal width (cm) 0.000

There are two configurable parameters in the tree classifier:

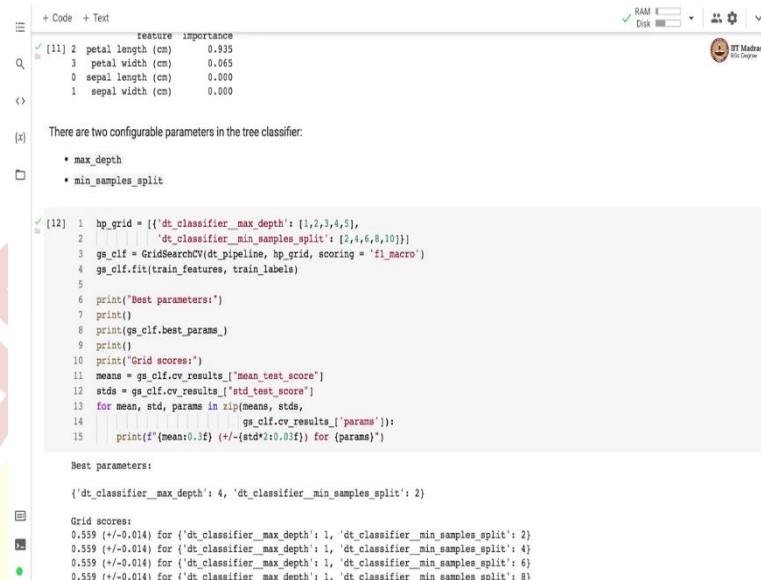
• max_depth
• min_samples_split

[12] 1 hp_grid = [{'dt_classifier_max_depth': [1,2,3,4,5],
2             'dt_classifier_min_samples_split': [2,4,6,8,10]}]
3 gs_clf = GridSearchCV(dt_pipeline, hp_grid, scoring = 'f1_macro')
4 gs_clf.fit(train_features, train_labels)
5
6 print("Best parameters:")
7 print()
8 print(gs_clf.best_params_)
```

Let us get the feature importance from the train model. So, we can get the feature importance by calling `feature_importance_` member variable of the decision tree estimator. And here we have printed the importance in its descending order. So, petal length is the most important

feature, it has got score of 0.935 followed by petal width. Sepal length and sepal width have important score of 0. So, they are not that important in this particular DecisionTreeClassifier that we have trained.

(Refer Slide Time: 08:41)



The screenshot shows a Jupyter Notebook interface. At the top, a table displays feature importance:

	feature	importance
[11]	2 petal length (cm)	0.935
	3 petal width (cm)	0.065
	0 sepal length (cm)	0.000
	1 sepal width (cm)	0.000

Below the table, a message states: "There are two configurable parameters in the tree classifier:"

- max_depth
- min_samples_split

The code cell [12] contains the following Python code:

```
1 hp_grid = [{'dt_classifier_max_depth': [1,2,3,4,5],
2             'dt_classifier_min_samples_split': [2,4,6,8,10]}]
3 gs_clf = GridSearchCV(dt_pipeline, hp_grid, scoring = 'f1_macro')
4 gs_clf.fit(train_features, train_labels)
5
6 print("Best parameters:")
7 print()
8 print(gs_clf.best_params_)
9 print()
10 print("Grid scores:")
11 means = gs_clf.cv_results_['mean_test_score']
12 stds = gs_clf.cv_results_['std_test_score']
13 for mean, std, params in zip(means, stds,
14                             gs_clf.cv_results_['params']):
15     print(f'{mean:0.3f} (+/-{std*2:0.01f}) for {params}')

Best parameters:
{'dt_classifier_max_depth': 4, 'dt_classifier_min_samples_split': 2}

Grid scores:
0.559 (+/-0.014) for {'dt_classifier_max_depth': 1, 'dt_classifier_min_samples_split': 2}
0.559 (+/-0.014) for {'dt_classifier_max_depth': 1, 'dt_classifier_min_samples_split': 4}
0.559 (+/-0.014) for {'dt_classifier_max_depth': 1, 'dt_classifier_min_samples_split': 6}
0.559 (+/-0.014) for {'dt_classifier_max_depth': 1, 'dt_classifier_min_samples_split': 8}
```

Now, there are two configurable parameters in the tree classifier one is max _depth and second is min _samples _split. So, max _depth define the maximum depth of the tree and minimum sample split decides how much sample should be there when we are trying to split the node. So, here we define the hyper-parameter grid on max _depth and min _samples _split.

So, max _depth we want to try values like 1, 2, 3, 4 and 5. For min _samples _split, we want to try values like 2, 4, 6, 8 and 10. We use GridSearchCV for hyper-parameter tuning, we use the pipeline object, we specify the hyper-parameter grid and we use f1 _macro as a scoring function. We call the Fit method on GridSearchCV object to launch this hyper-parameter search. And after hyper-parameter search is complete, we get best parameters and grid values.

(Refer Slide Time: 09:48)

```
+ Code + Text
4 gs_clf.fit(train_features, train_labels)
5
6 print("Best parameters:")
7 print()
8 print(gs_clf.best_params_)
9 print()
10 print("Grid scores:")
11 means = gs_clf.cv_results_['mean_test_score']
12 stds = gs_clf.cv_results_['std_test_score']
13 for mean, std, params in zip(means, stds,
14                             gs_clf.cv_results_['params']):
15     print(f'{mean:0.3f} (+/-{std*2:0.3f}) for {params}')

Best parameters:
{'dt_classifier_max_depth': 4, 'dt_classifier_min_samples_split': 2}

Grid scores:
0.559 (+/-0.014) for {'dt_classifier_max_depth': 1, 'dt_classifier_min_samples_split': 2}
0.559 (+/-0.014) for {'dt_classifier_max_depth': 1, 'dt_classifier_min_samples_split': 4}
0.559 (+/-0.014) for {'dt_classifier_max_depth': 1, 'dt_classifier_min_samples_split': 6}
0.559 (+/-0.014) for {'dt_classifier_max_depth': 1, 'dt_classifier_min_samples_split': 8}
0.559 (+/-0.014) for {'dt_classifier_max_depth': 1, 'dt_classifier_min_samples_split': 10}
0.916 (+/-0.091) for {'dt_classifier_max_depth': 2, 'dt_classifier_min_samples_split': 2}
0.916 (+/-0.091) for {'dt_classifier_max_depth': 2, 'dt_classifier_min_samples_split': 4}
0.916 (+/-0.091) for {'dt_classifier_max_depth': 2, 'dt_classifier_min_samples_split': 6}
0.916 (+/-0.091) for {'dt_classifier_max_depth': 2, 'dt_classifier_min_samples_split': 8}
0.916 (+/-0.091) for {'dt_classifier_max_depth': 2, 'dt_classifier_min_samples_split': 10}
0.932 (+/-0.115) for {'dt_classifier_max_depth': 3, 'dt_classifier_min_samples_split': 2}
0.932 (+/-0.115) for {'dt_classifier_max_depth': 3, 'dt_classifier_min_samples_split': 4}
0.932 (+/-0.115) for {'dt_classifier_max_depth': 3, 'dt_classifier_min_samples_split': 6}
0.932 (+/-0.115) for {'dt_classifier_max_depth': 3, 'dt_classifier_min_samples_split': 8}
0.932 (+/-0.115) for {'dt_classifier_max_depth': 3, 'dt_classifier_min_samples_split': 10}
0.941 (+/-0.115) for {'dt_classifier_max_depth': 4, 'dt_classifier_min_samples_split': 2}
0.932 (+/-0.115) for {'dt_classifier_max_depth': 4, 'dt_classifier_min_samples_split': 4}
0.941 (+/-0.115) for {'dt_classifier_max_depth': 4, 'dt_classifier_min_samples_split': 6}
0.932 (+/-0.115) for {'dt_classifier_max_depth': 4, 'dt_classifier_min_samples_split': 8}
0.932 (+/-0.115) for {'dt_classifier_max_depth': 4, 'dt_classifier_min_samples_split': 10}
```

So, we found the best parameters with max_depth = 4 and min_samples_split = 2.

(Refer Slide Time: 09:59)



So, after performing the GridSearchCV we get the best estimator and for that particular best estimator, we evaluated by calculating the confusion_matrix on the test set. And you can see that we have obtained a perfect confusion_matrix with 0 entries in each of diagonal cells. So, in this video, we demonstrate the utility of decision trees in multi class classification. We used Iris dataset, which has got three classes and we showed that decision tree can produce almost perfect Iris classification.