

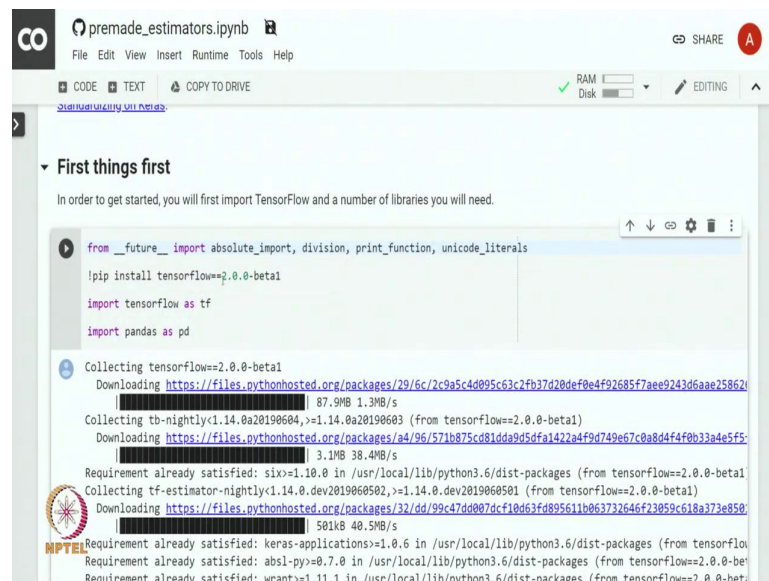
Practical Machine Learning
Dr. Ashish Tendulkar
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture - 27
Estimator API

So far in this course you are using tf.keras as an API to build our machine learning models. tf.keras is believed to be a simpler API for building models in TensorFlow 2.0. There is another way to build building TensorFlow - through Estimator API.

Estimator is TensorFlow's high level representation of a complete model and it has been designed for easy scaling and a synchronous training. In this module, we will build machine learning models with TensorFlow estimator API. In this exercise we will use Iris classification problem for demonstrating how to build machine learning models with estimator API.

(Refer Slide Time: 01:16)



```
from __future__ import absolute_import, division, print_function, unicode_literals

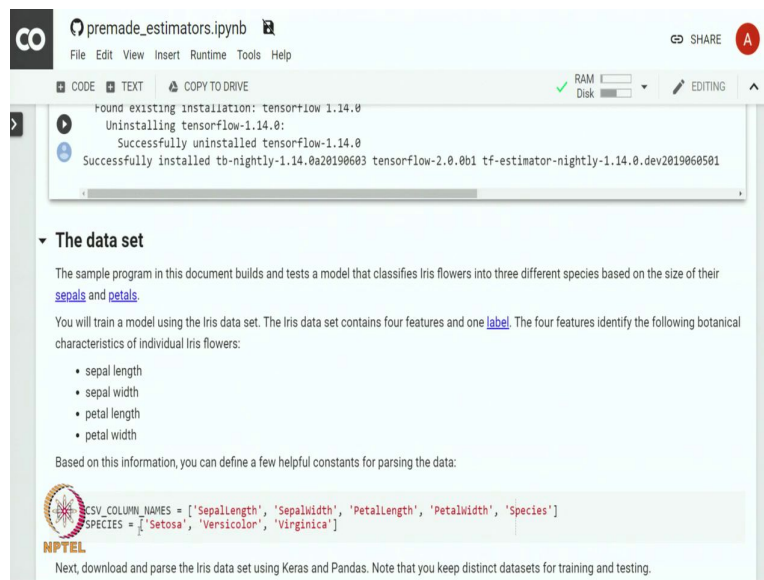
!pip install tensorflow==2.0.0-beta1

import tensorflow as tf
import pandas as pd
```

Collecting tensorflow==2.0.0-beta1
Downloading <https://files.pythonhosted.org/packages/29/6c/2c9a5c4d095c63c2fb37d20def0e4f92685f7aee9243d6aae25862/>
87.9MB 1.3MB/s
Collecting tb-nightly<1.14.0a20190604,>=1.14.0a20190603 (from tensorflow==2.0.0-beta1)
Downloading <https://files.pythonhosted.org/packages/a4/96/571b875cd81dd9d5dfa1422a4f9d749e67c0a8d4f4f0b33a4e5f5/>
3.1MB 38.4MB/s
Requirement already satisfied: six>=1.10.0 in /usr/local/lib/python3.6/dist-packages (from tensorflow==2.0.0-beta1)
Collecting tf-estimator-nightly<1.14.0.dev2019060502,>=1.14.0.dev2019060501 (from tensorflow==2.0.0-beta1)
Downloading <https://files.pythonhosted.org/packages/32/dd/99c47dd007dcf10d63fd895611b063732646f23059c618a373e850/>
501kB 40.5MB/s
Requirement already satisfied: keras-applications>=1.0.6 in /usr/local/lib/python3.6/dist-packages (from tensorflow==2.0.0-beta1)
Requirement already satisfied: absl-py>=0.7.0 in /usr/local/lib/python3.6/dist-packages (from tensorflow==2.0.0-beta1)
Requirement already satisfied: wrapt>=1.11.1 in /usr/local/lib/python3.6/dist-packages (from tensorflow==2.0.0-beta1)

Let us begin by first importing TensorFlow and number of libraries that we need. We are mainly going to use Pandas as a library for manipulating structured data.

(Refer Slide Time: 01:44)



The screenshot shows a Jupyter Notebook titled 'premade_estimators.ipynb'. The top toolbar includes 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Below the toolbar, there are tabs for 'CODE' and 'TEXT', and a 'COPY TO DRIVE' button. The main area displays the output of a code cell, which shows the successful installation of TensorFlow 2.0.0b1 and tf-estimator-nightly-1.14.0.dev2019060501. Below the output, there is a section titled 'The data set' which describes the Iris dataset. It mentions that the dataset contains four features (sepal length, sepal width, petal length, and petal width) and one label (Species). It also provides a list of constants for parsing the data, including 'SEPAL_COLUMN_NAMES' and 'SPECIES'.

```
Found existing installation: tensorflow 1.14.0
Uninstalling tensorflow-1.14.0:
  Successfully uninstalled tensorflow-1.14.0
Successfully installed tb-nightly-1.14.0a20190603 tensorflow-2.0.0b1 tf-estimator-nightly-1.14.0.dev2019060501
```

The data set

The sample program in this document builds and tests a model that classifies Iris flowers into three different species based on the size of their [sepals](#) and [petals](#).

You will train a model using the Iris data set. The Iris data set contains four features and one [label](#). The four features identify the following botanical characteristics of individual Iris flowers:

- sepal length
- sepal width
- petal length
- petal width

Based on this information, you can define a few helpful constants for parsing the data:

```
SEPAL_COLUMN_NAMES = ['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth', 'Species']
SPECIES = ['Setosa', 'Versicolor', 'Virginica']
```

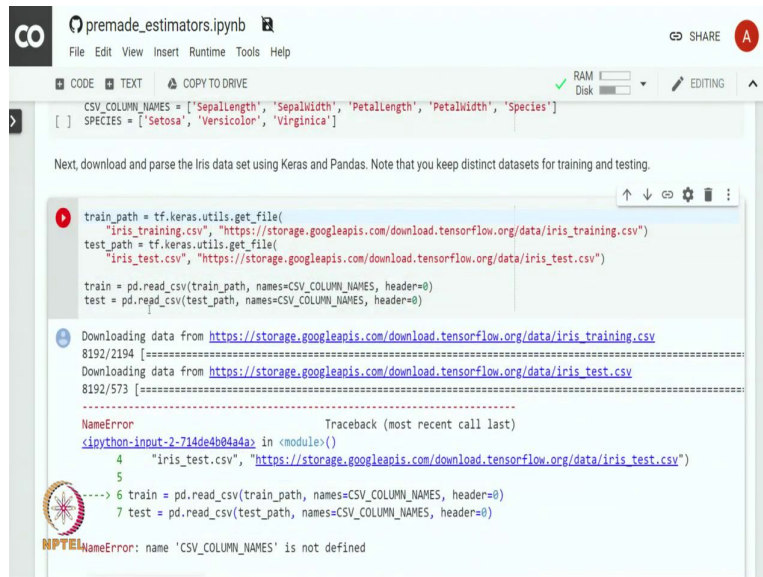
Next, download and parse the Iris data set using Keras and Pandas. Note that you keep distinct datasets for training and testing.

Now, that we have installed the required libraries or required packages. Let us get into building the model with TensorFlow estimator API. In this exercise we will build and test a model for classifying Iris flowers into three different species based on the size of the sepals and petals.

Iris dataset has four features and one label. The four features identify the following botanical characteristics of individual Iris flowers. There is a sepal length and width, and petal length and width. So, there are four columns and there are three different species in our dataset there is Setosa, Versicolor, and Virginica as species and we have to classify the incoming flower into one of these three categories.

And a flower is represented by four features - sepal length and width, and petal length and width. So, the file the input file has five columns sepal length and width petal length and width and the name of the species.

(Refer Slide Time: 03:00)



```
premade_estimators.ipynb
File Edit View Insert Runtime Tools Help

CODE TEXT COPY TO DRIVE RAM Disk EDITING

[ ] CSV_COLUMN_NAMES = ['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth', 'Species']
    SPECIES = ['Setosa', 'Versicolor', 'Virginica']

Next, download and parse the Iris data set using Keras and Pandas. Note that you keep distinct datasets for training and testing.

train_path = tf.keras.utils.get_file(
    "iris_training.csv", "https://storage.googleapis.com/download.tensorflow.org/data/iris_training.csv")
test_path = tf.keras.utils.get_file(
    "iris_test.csv", "https://storage.googleapis.com/download.tensorflow.org/data/iris_test.csv")

train = pd.read_csv(train_path, names=CSV_COLUMN_NAMES, header=0)
test = pd.read_csv(test_path, names=CSV_COLUMN_NAMES, header=0)

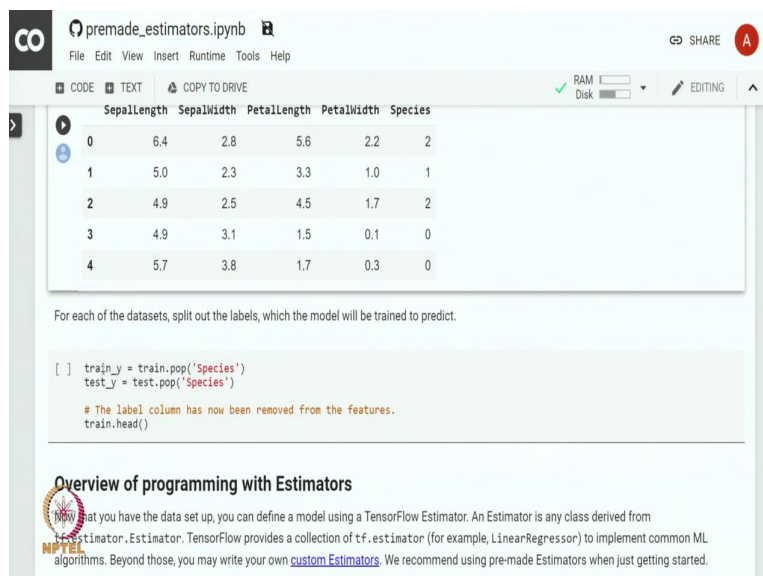
Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/iris_training.csv
8192/2194 [=====]
Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/iris_test.csv
8192/573 [=====]

NameError                                Traceback (most recent call last)
<ipython-input-2-714de4b04a4a> in <module>()
      4     "iris_test.csv", "https://storage.googleapis.com/download.tensorflow.org/data/iris_test.csv")
      5
----> 6 train = pd.read_csv(train_path, names=CSV_COLUMN_NAMES, header=0)
      7 test = pd.read_csv(test_path, names=CSV_COLUMN_NAMES, header=0)

NameError: name 'CSV_COLUMN_NAMES' is not defined
```

Let us download and parse Iris data using Keras and Pandas. Let us get the training and test file and read the CSV to get a Pandas data frame. So, the training data is saved in train dataframe and test data is saved in test dataframe. Let us examine the training data.

(Refer Slide Time: 03:30)



```
premade_estimators.ipynb
File Edit View Insert Runtime Tools Help

CODE TEXT COPY TO DRIVE RAM Disk EDITING

SepalLength SepalWidth PetalLength PetalWidth Species
0          6.4         2.8         5.6         2.2     2
1          5.0         2.3         3.3         1.0     1
2          4.9         2.5         4.5         1.7     2
3          4.9         3.1         1.5         0.1     0
4          5.7         3.8         1.7         0.3     0

For each of the datasets, split out the labels, which the model will be trained to predict.

[ ] train_y = train.pop('Species')
    test_y = test.pop('Species')

# The label column has now been removed from the features.
train.head()

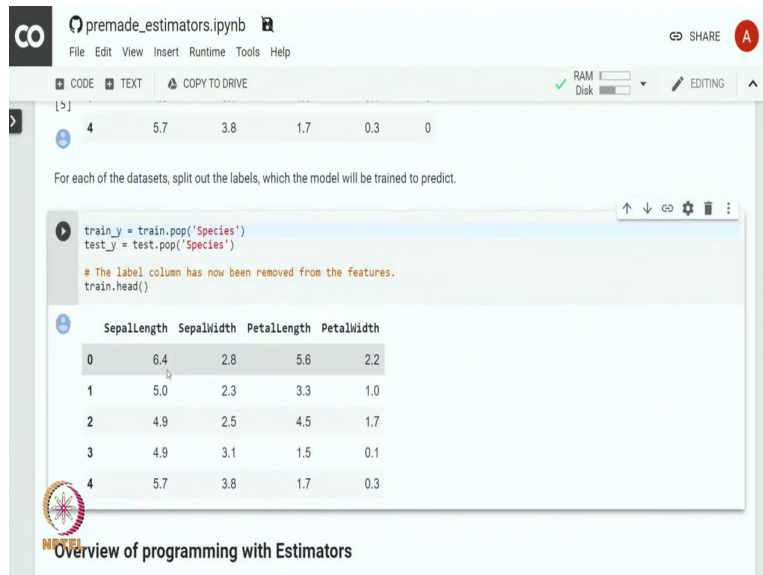
Overview of programming with Estimators

Now that you have the data set up, you can define a model using a TensorFlow Estimator. An Estimator is any class derived from
tf.estimator.Estimator. TensorFlow provides a collection of tf.estimator (for example, LinearRegressor) to implement common ML
algorithms. Beyond those, you may write your own custom Estimators. We recommend using pre-made Estimators when just getting started.
```

Let us look at the first five examples in the training; you can see that we have four columns and a last column is the desired label that we want to learn. So, here we want to learn the mapping between these four features to the species. Let us get the species column which is

the label from the data frame out and store that into; and store that into train_y list and for test we do the same thing and store that in test_y list.

(Refer Slide Time: 04:08)



```
[>]
4      5.7      3.8      1.7      0.3      0

For each of the datasets, split out the labels, which the model will be trained to predict.

train_y = train.pop('Species')
test_y = test.pop('Species')

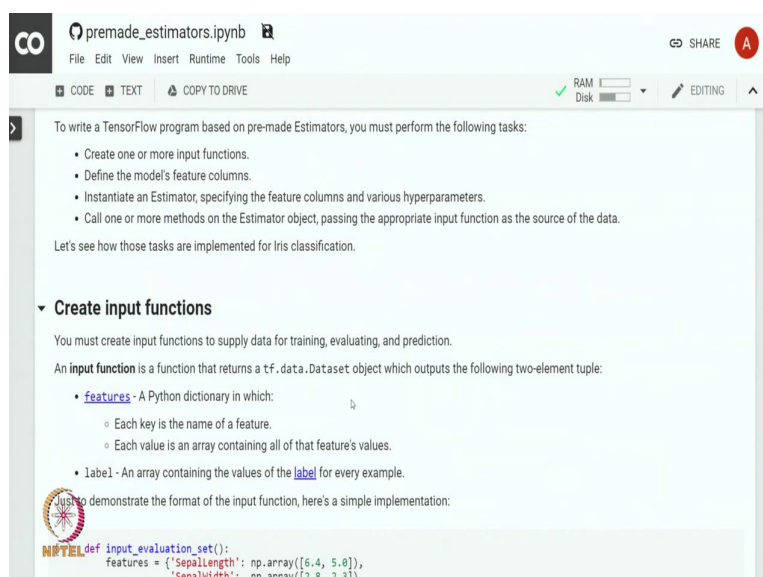
# The label column has now been removed from the features.
train.head()
```

	SepalLength	SepalWidth	PetalLength	PetalWidth
0	6.4	2.8	5.6	2.2
1	5.0	2.3	3.3	1.0
2	4.9	2.5	4.5	1.7
3	4.9	3.1	1.5	0.1
4	5.7	3.8	1.7	0.3

Overview of programming with Estimators

Let us look at the first five columns of the training now. You can see that the species column has been removed from the training because of using the pop command. Everything else remains the same.

(Refer Slide Time: 04:23)



To write a TensorFlow program based on pre-made Estimators, you must perform the following tasks:

- Create one or more input functions.
- Define the model's feature columns.
- Instantiate an Estimator, specifying the feature columns and various hyperparameters.
- Call one or more methods on the Estimator object, passing the appropriate input function as the source of the data.

Let's see how those tasks are implemented for Iris classification.

▼ **Create input functions**

You must create input functions to supply data for training, evaluating, and prediction.

An **input function** is a function that returns a `tf.data.Dataset` object which outputs the following two-element tuple:

- **features** - A Python dictionary in which:
 - Each key is the name of a feature.
 - Each value is an array containing all of that feature's values.
- **label** - An array containing the values of the **label** for every example.

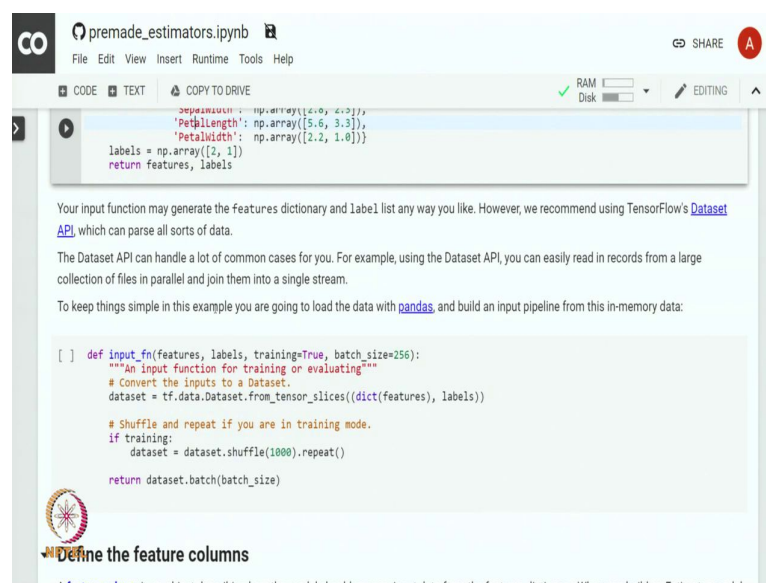
Just to demonstrate the format of the input function, here's a simple implementation:

```
def input_evaluation_set():
    features = {'SepalLength': np.array([6.4, 5.8]),
               'SepalWidth': np.array([2.8, 2.3]),
```

If you want to use estimator, there are three steps. You have to create one or more input function that defines how the data will be input to the estimator, you have to define feature columns and then instantiate an estimator specifying the feature columns and various hyper parameters and call appropriate method on the estimated object.

Let us understand how this task can be implemented for Iris classification. First let us create input function. Input function is a function that returns a `tf.data.Dataset` object which outputs the following two elements as tuple, we have features and labels.

(Refer Slide Time: 05:11)



```
def input_fn(features, labels, training=True, batch_size=256):
    """An input function for training or evaluating"""
    # Convert the inputs to a Dataset.
    dataset = tf.data.Dataset.from_tensor_slices((dict(features), labels))

    # Shuffle and repeat if you are in training mode.
    if training:
        dataset = dataset.shuffle(1000).repeat()

    return dataset.batch(batch_size)
```

Your input function may generate the features dictionary and label list any way you like. However, we recommend using TensorFlow's [Dataset API](#), which can parse all sorts of data.

The Dataset API can handle a lot of common cases for you. For example, using the Dataset API, you can easily read in records from a large collection of files in parallel and join them into a single stream.

To keep things simple in this example you are going to load the data with [pandas](#), and build an input pipeline from this in-memory data:

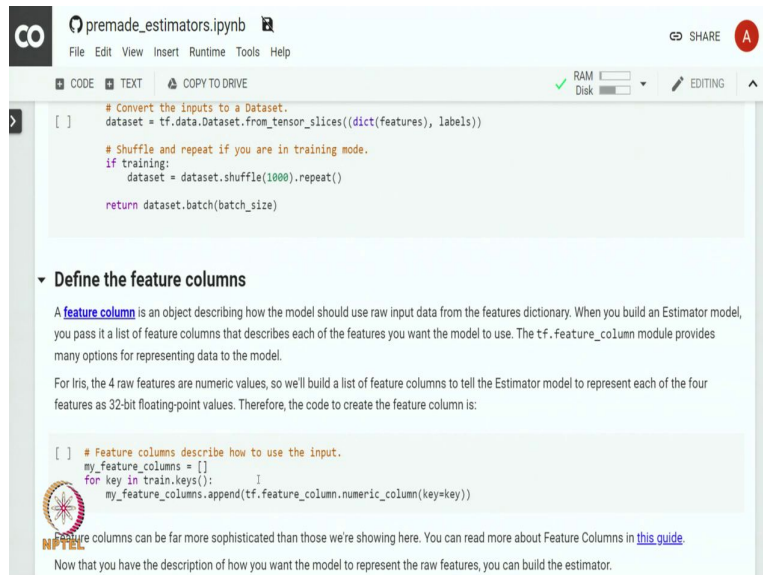
Let us look at how the dataset object looks. We have a dictionary of features which contains the feature name and list of values. You can see that the input function returns the feature dictionary and the label array.

In order to keep things simple we will be loading the data with pandas and we will build an input pipeline from this in memory data. The Dataset API is very powerful as it can easily records from a large collections of file in parallel and join them into single stream. However, for the Iris dataset this particular functionality will not be required.

So, we will use here Pandas dataframe and create the data set using `from_tensor_slices` function. We take the dictionary of features and array of labels to create a dataset object. In

case of training we shuffle the data set object and return the dataset object in a specified batch size.

(Refer Slide Time: 07:11)



```
# Convert the inputs to a Dataset.
dataset = tf.data.Dataset.from_tensor_slices((dict(features), labels))

# Shuffle and repeat if you are in training mode.
if training:
    dataset = dataset.shuffle(1000).repeat()

return dataset.batch(batch_size)
```

Define the feature columns

A [feature column](#) is an object describing how the model should use raw input data from the features dictionary. When you build an Estimator model, you pass it a list of feature columns that describes each of the features you want the model to use. The `tf.feature_column` module provides many options for representing data to the model.

For Iris, the 4 raw features are numeric values, so we'll build a list of feature columns to tell the Estimator model to represent each of the four features as 32-bit floating-point values. Therefore, the code to create the feature column is:

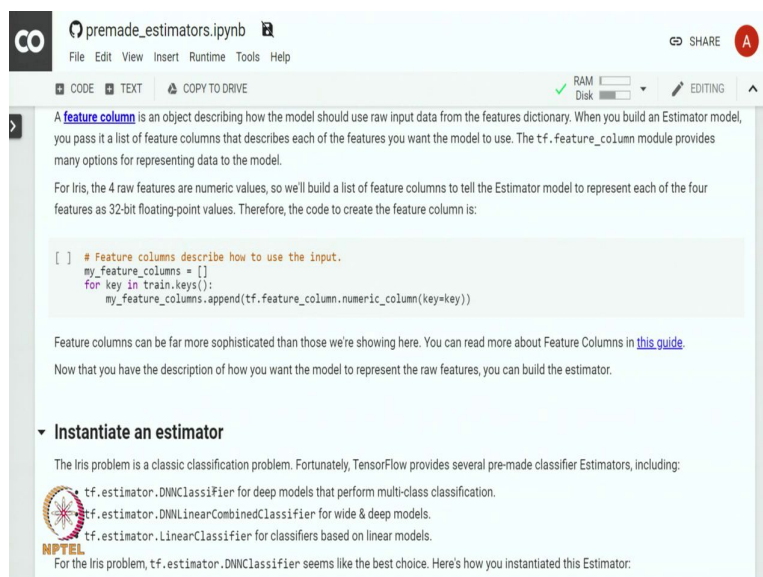
```
# Feature columns describe how to use the input.
my_feature_columns = []
for key in train.keys():
    my_feature_columns.append(tf.feature_column.numeric_column(key=key))
```

Feature columns can be far more sophisticated than those we're showing here. You can read more about Feature Columns in [this guide](#).

Now that you have the description of how you want the model to represent the raw features, you can build the estimator.

Next we define feature columns corresponding to the features. Since all the features are numeric, we will use numeric feature column.

(Refer Slide Time: 07:21)



```
# Feature columns describe how to use the input.
my_feature_columns = []
for key in train.keys():
    my_feature_columns.append(tf.feature_column.numeric_column(key=key))
```

Feature columns can be far more sophisticated than those we're showing here. You can read more about Feature Columns in [this guide](#).

Now that you have the description of how you want the model to represent the raw features, you can build the estimator.

Instantiate an estimator

The Iris problem is a classic classification problem. Fortunately, TensorFlow provides several pre-made classifier Estimators, including:

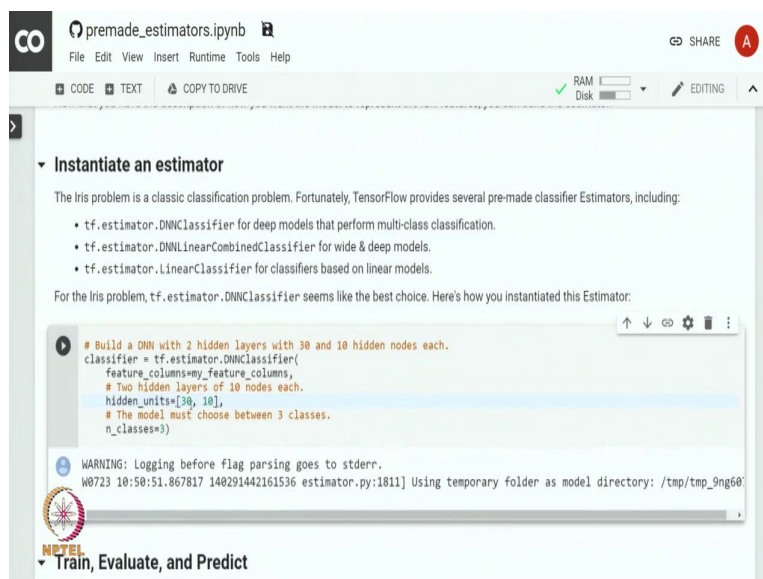
- `tf.estimator.DNNClassifier` for deep models that perform multi-class classification.
- `tf.estimator.DNNLinearCombinedClassifier` for wide & deep models.
- `tf.estimator.LinearClassifier` for classifiers based on linear models.

For the Iris problem, `tf.estimator.DNNClassifier` seems like the best choice. Here's how you instantiated this Estimator:

Now, that we have defined our input function and created feature columns, the next task is to instantiate an estimator. There are several premade classifier estimators defined in TensorFlow. There is a DNN classifier that is used for deep models on multiclass classification. A DNN linear combined classifier is used for wide and deep model.

Wide model works on a large number of features like a very large one hot encoding, and deep model works with the features which come from embeddings. So, DNN linear combined classifier is used for wide and deep models. Linear classifier is based on linear model. For Iris problem we will use a DNN classifier that helps us perform multiclass classification.

(Refer Slide Time: 08:20)



The screenshot shows a Jupyter Notebook interface with the title 'premade_estimators.ipynb'. The notebook is in 'CODE' view. The current cell is titled 'Instantiate an estimator' and contains the following text and code:

The Iris problem is a classic classification problem. Fortunately, TensorFlow provides several pre-made classifier Estimators, including:

- `tf.estimator.DNNClassifier` for deep models that perform multi-class classification.
- `tf.estimator.DNNLinearCombinedClassifier` for wide & deep models.
- `tf.estimator.LinearClassifier` for classifiers based on linear models.

For the Iris problem, `tf.estimator.DNNClassifier` seems like the best choice. Here's how you instantiated this Estimator:

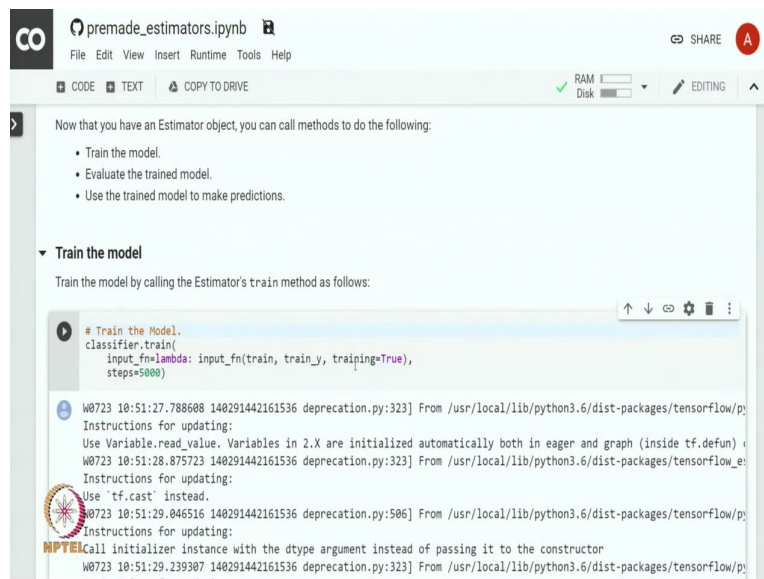
```
# Build a DNN with 2 hidden layers with 30 and 10 hidden nodes each.
classifier = tf.estimator.DNNClassifier(
    feature_columns=my_feature_columns,
    # Two hidden layers of 10 nodes each.
    hidden_units=[30, 10],
    # The model must choose between 3 classes.
    n_classes=3)
```

Below the code, there is a warning message: 'WARNING: Logging before flag parsing goes to stderr. W0723 10:50:51.867817 140291442161536 estimator.py:1811] Using temporary folder as model directory: /tmp/tmp_9ng60'.

At the bottom of the notebook, there is a section titled 'Train, Evaluate, and Predict'.

Let us see how to instantiate this estimator. Here we define a DNN classifier with two hidden layers each with 30 and 10 nodes respectively. We also specify that there are three classes, so that the corresponding output layer can be constructed and we specify our feature columns as input to the DNN classifier. Let us run, let us instantiate the DNN classifier.

(Refer Slide Time: 08:56)



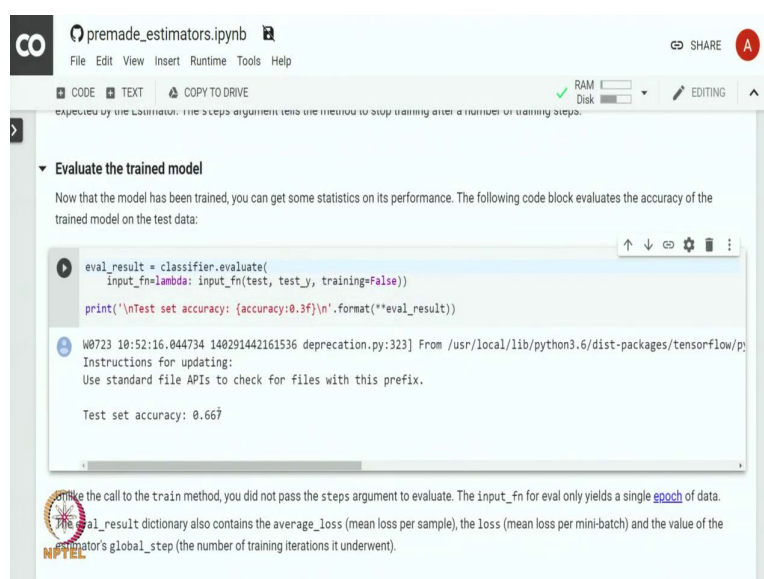
```
# Train the Model.
classifier.train(
    input_fn=lambda: input_fn(train, train_y, training=True),
    steps=5000)

W0723 10:51:27.788608 140291442161536 deprecation.py:323] From /usr/local/lib/python3.6/dist-packages/tensorflow/p
Instructions for updating:
Use Variable.read_value. Variables in 2.X are initialized automatically both in eager and graph (inside tf.defun)
W0723 10:51:28.875723 140291442161536 deprecation.py:323] From /usr/local/lib/python3.6/dist-packages/tensorflow_e
Instructions for updating:
Use 'tf.cast' instead.
W0723 10:51:29.046516 140291442161536 deprecation.py:506] From /usr/local/lib/python3.6/dist-packages/tensorflow/p
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
W0723 10:51:29.239307 140291442161536 deprecation.py:323] From /usr/local/lib/python3.6/dist-packages/tensorflow/p
```

So, we will train the model by calling the estimator's train method where we specify the input function and also specify the number of steps for which the training loop should be run.

Once the model is trained, evaluate the model with the test data. So, we use the same input function but instead of train we are going to pass the arguments corresponding to the test data. We set training to be false as against the training to be true at the time of training.

(Refer Slide Time: 09:36)



```
eval_result = classifier.evaluate(
    input_fn=lambda: input_fn(test, test_y, training=False))

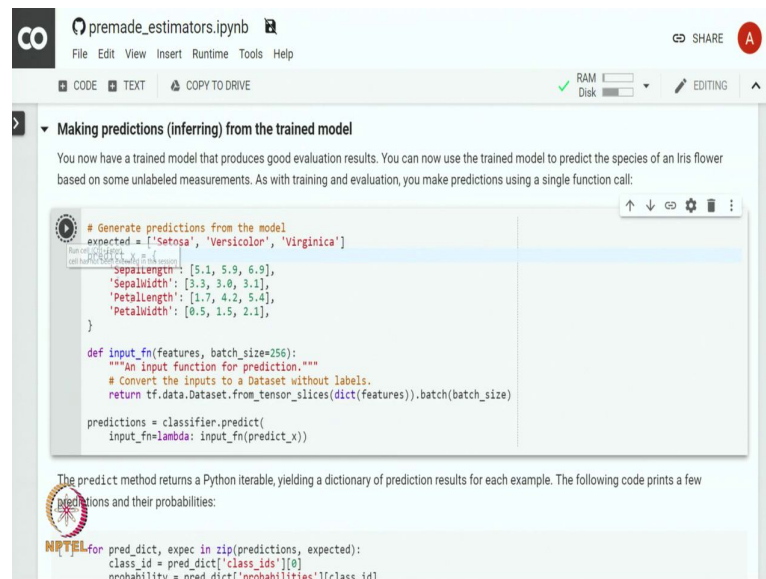
print('\nTest set accuracy: {accuracy:0.3f}\n'.format(**eval_result))

W0723 10:52:16.044734 140291442161536 deprecation.py:323] From /usr/local/lib/python3.6/dist-packages/tensorflow/p
Instructions for updating:
Use standard file APIs to check for files with this prefix.

Test set accuracy: 0.667
```


And you can see that we achieve accuracy of 66 percent, we achieved test accuracy of 66 percent on the Iris classification.

(Refer Slide Time: 09:48)



```
# Generate predictions from the model
expected = ['Setosa', 'Versicolor', 'Virginica']

features = {
    'SepalLength': [5.1, 5.9, 6.9],
    'SepalWidth': [3.3, 3.0, 3.1],
    'PetalLength': [1.7, 4.2, 5.4],
    'PetalWidth': [0.5, 1.5, 2.1],
}

def input_fn(features, batch_size=256):
    """An input function for prediction."""
    # Convert the inputs to a Dataset without labels.
    return tf.data.Dataset.from_tensor_slices(dict(features)).batch(batch_size)

predictions = classifier.predict(
    input_fn=lambda: input_fn(predict_x))

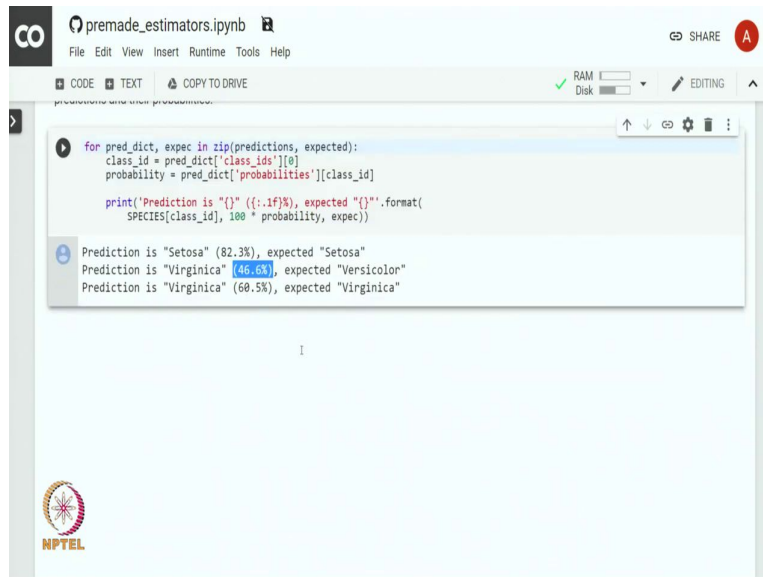
# The predict method returns a Python iterable, yielding a dictionary of prediction results for each example. The following code prints a few
# predictions and their probabilities:
for pred_dict, expec in zip(predictions, expected):
    class_id = pred_dict['class_ids'][0]
    probability = pred_dict['probabilities'][class_id]
```

Let us use this particular model for making predictions on unseen data or for inferencing. So, here we will have to first specify what is the expected output and then we specified the features. So, here the feature vector is specified as a dictionary where the key is the name of the column or name of the feature and followed by a list of values. So, here we are specifying three examples with their value specified for each of the feature in a list.

For example we have a sepal length of 5.1 5.9 and 6.9 corresponding to the three examples and they have sepal width of 3.3, 3.0 and 3.1. So, 5.1 SepalLength the flower with SepalLength of 5.1 has SepalWidth of 3.3, PetalLength of 1.7 and PetalWidth of 0.5.

So, this is how you have to interpret an example, but it is specified in a slightly different format or in a transposed form. We specify the input function to get a dataset object from tensor slices and we give the predict_x as a dictionary to the input function which returns a dataset object on which we apply the prediction. So, let us run this and see what kind of predictions are coming out.

(Refer Slide Time: 11:33)

A screenshot of a Jupyter Notebook interface. The title bar shows 'premade_estimators.ipynb'. The code cell contains a loop that iterates over predictions and expected values for the Iris dataset. It prints the predicted class, the expected class, and the probability. The output shows three predictions: Setosa (82.3% probability, expected Setosa), Virginica (46.6% probability, expected Versicolor), and Virginica (60.5% probability, expected Virginica). The NPTEL logo is visible in the bottom left corner.

```
for pred_dict, expec in zip(predictions, expected):
    class_id = pred_dict['class_ids'][0]
    probability = pred_dict['probabilities'][class_id]
    print('Prediction is "{}" ({:.1f}%), expected "{}"'.format(
        SPECIES[class_id], 100 * probability, expec))
```

Prediction is "Setosa" (82.3%), expected "Setosa"
Prediction is "Virginica" (46.6%), expected "Versicolor"
Prediction is "Virginica" (60.5%), expected "Virginica"

And let us look at the predictions and the expected result and we also we will also print the probabilities. We can see that the first prediction is Setosa where the actual label was also Setosa and we can see that this prediction is with quite good probability or quite good confidence of 82 percent. The second prediction is Virginica where the expected or where the actual prediction was Versicolor, but you can see that the probability of the prediction is less than 50 percent.

In the third case, the prediction is Virginica which matches the actual label of Virginica and has got 60.5 percent probability of the label. So, in this module we learnt how to use tf estimator API and we applied that for Iris classification. We understood that in order to define a tf estimator API we have three main steps we have to specify one or more input functions. We have to specify the feature columns and we have to instantiate tf estimator API with appropriate configuration.

In the next session we will use this tf estimator API for building a linear model. Hope to see you in the next session.