INDIAN INSTITUTE OF TECHNOLOGY MADRAS

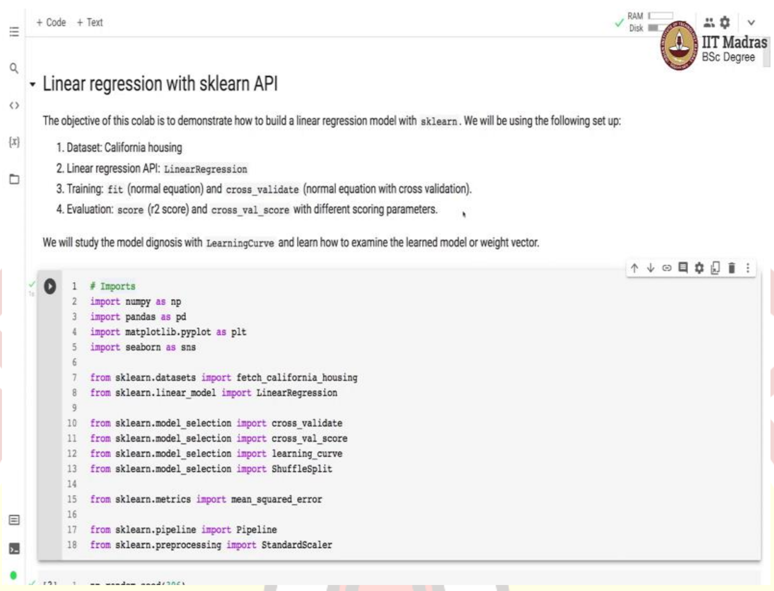
सिद्धिर्भवति कर्मजा

IIT Madras

ONLINE DEGREE

Machine Learning Program
Indian Institute of Technology, Madras
Linear Regression Demonstration

(Refer Slide Time: 0:10)



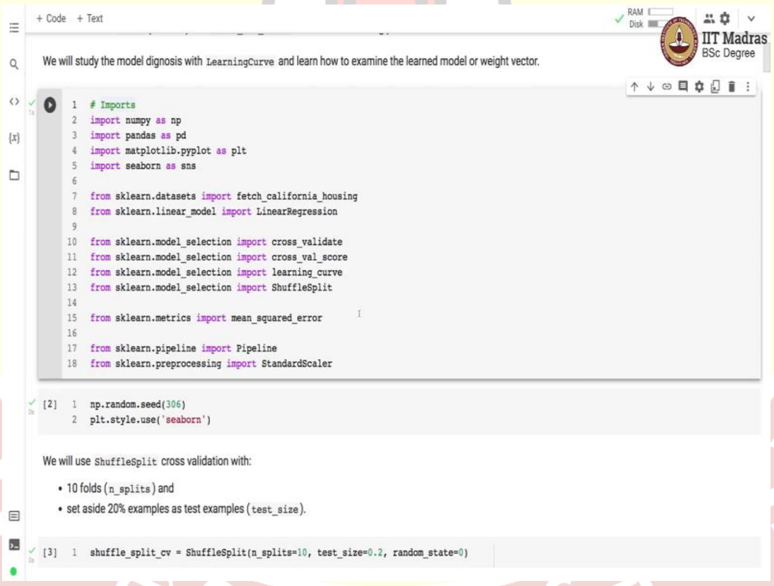
Linear regression with sklearn API

The objective of this colab is to demonstrate how to build a linear regression model with `sklearn`. We will be using the following set up:

1. Dataset: California housing
2. Linear regression API: `LinearRegression`
3. Training: `fit` (normal equation) and `cross_validate` (normal equation with cross validation).
4. Evaluation: `score` (r2 score) and `cross_val_score` with different scoring parameters.

We will study the model dignosis with `LearningCurve` and learn how to examine the learned model or weight vector.

```python
# Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.datasets import fetch_california_housing
from sklearn.linear_model import LinearRegression

from sklearn.model_selection import cross_validate
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import learning_curve
from sklearn.model_selection import ShuffleSplit

from sklearn.metrics import mean_squared_error

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
```



We will study the model dignosis with `LearningCurve` and learn how to examine the learned model or weight vector.

```python
# Imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.datasets import fetch_california_housing
from sklearn.linear_model import LinearRegression

from sklearn.model_selection import cross_validate
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import learning_curve
from sklearn.model_selection import ShuffleSplit

from sklearn.metrics import mean_squared_error

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
```

```python
[2]
np.random.seed(306)
plt.style.use('seaborn')
```

We will use `ShuffleSplit` cross validation with:

- 10 folds (`n_splits`) and
- set aside 20% examples as test examples (`test_size`).

```python
[3]
shuffle_split_cv = ShuffleSplit(n_splits=10, test_size=0.2, random_state=0)
```

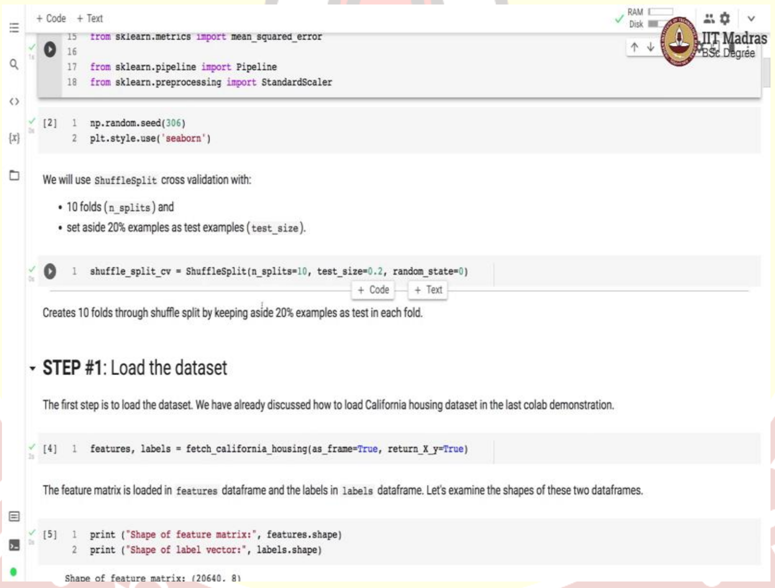
Namaste. Welcome to the next video of the machine learning practice course. In this video, we will demonstrate linear regression with sklearn API. The objective of this collab is to demonstrate how to build a linear regression model with sklearn. We will be using the California housing dataset for this exercise. We will be implementing a linear regression API that uses the normal equation for training the model. Then we will also train the model with cross-validation.

After the model is trained, we will evaluate the model with a score function that uses r2 score. We will also use cross val score with different scoring parameters in order to evaluate the training model with different scoring metrics. We will also study the model diagnosis with learning curve and how to examine the learned model or weight vector. Let us start by importing some useful packages.

Here we are importing some basic Python packages for manipulation and visualization. Then we are, then we import fetch California Housing from sklearn. dataset module that helps us to fetch the dataset. The linear regression, the linear regression API that we are importing from sklearn.linear_model helps us to build the linear regression model.

Then we have a bunch of APIs imported from model selection module and that helps us to perform cross -validation, plot the learning curve and performing the shuffle split, shuffle split in cross -validation. We are going to use mean square error as an additional metrics to r2 and mean square error is implemented in sklearn.metrics module. We will be using pipelining for combining the preprocessing and the model in a pipeline manner.

(Refer Slide Time: 2:39)



```
from sklearn.metrics import mean_squared_error

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
```

```
[2]
np.random.seed(306)
plt.style.use('seaborn')
```

We will use `ShuffleSplit` cross validation with:

- 10 folds (`n_splits`) and
- set aside 20% examples as test examples (`test_size`).

```
shuffle_split_cv = ShuffleSplit(n_splits=10, test_size=0.2, random_state=0)
```

Creates 10 folds through shuffle split by keeping aside 20% examples as test in each fold.

STEP #1: Load the dataset

The first step is to load the dataset. We have already discussed how to load California housing dataset in the last colab demonstration.

```
[4]
features, labels = fetch_california_housing(as_frame=True, return_X_y=True)
```

The feature matrix is loaded in `features` dataframe and the labels in `labels` dataframe. Let's examine the shapes of these two dataframes.

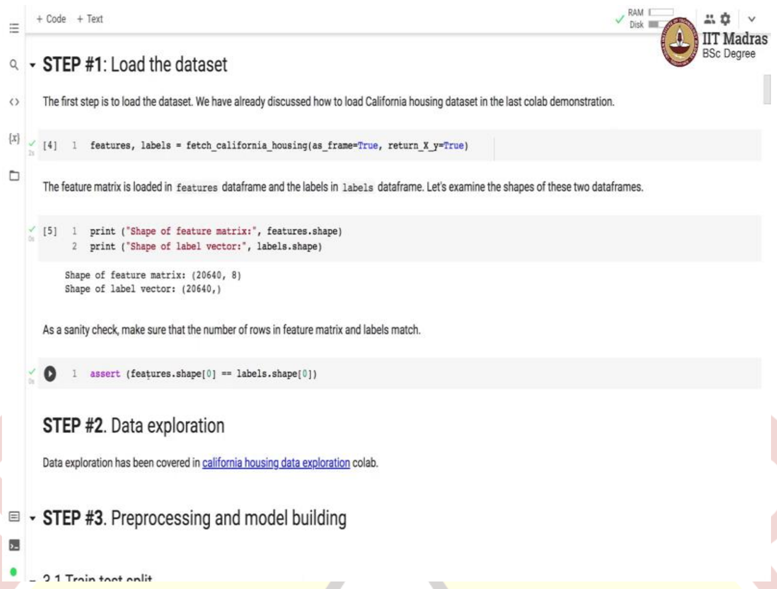
```
[5]
print ("Shape of feature matrix:", features.shape)
print ("Shape of label vector:", labels.shape)
```

```
Shape of feature matrix: (20640, 8)
```

It is always a good idea to set a random seed so that we get reproducible results across different runs. So we are going to set the seed to 306, you can set it to any number of your choice and we are going to use the plotting style that is seaborn. First, we will use shuffle split cross-validation with 10 folds and we will set aside 20 percent examples as test examples.

So, you can see that here we are instantiating a shuffle split object with n splits equal to 10, test size equal to 0.2 and we have set a random seed. This particular statement creates 10 folds through shuffles split by keeping the side 20 percent examples as test in each fold. We will be using the shuffle split strategy throughout this notebook.

(Refer Slide Time: 3:41)



STEP #1: Load the dataset

The first step is to load the dataset. We have already discussed how to load California housing dataset in the last colab demonstration.

```
[4]  features, labels = fetch_california_housing(as_frame=True, return_X_y=True)
```

The feature matrix is loaded in `features` dataframe and the labels in `labels` dataframe. Let's examine the shapes of these two dataframes.

```
[5]  print ("Shape of feature matrix:", features.shape)
     print ("Shape of label vector:", labels.shape)

     Shape of feature matrix: (20640, 8)
     Shape of label vector: (20640,)
```

As a sanity check, make sure that the number of rows in feature matrix and labels match.

```
assert (features.shape[0] == labels.shape[0])
```

STEP #2. Data exploration

Data exploration has been covered in california housing data exploration colab.

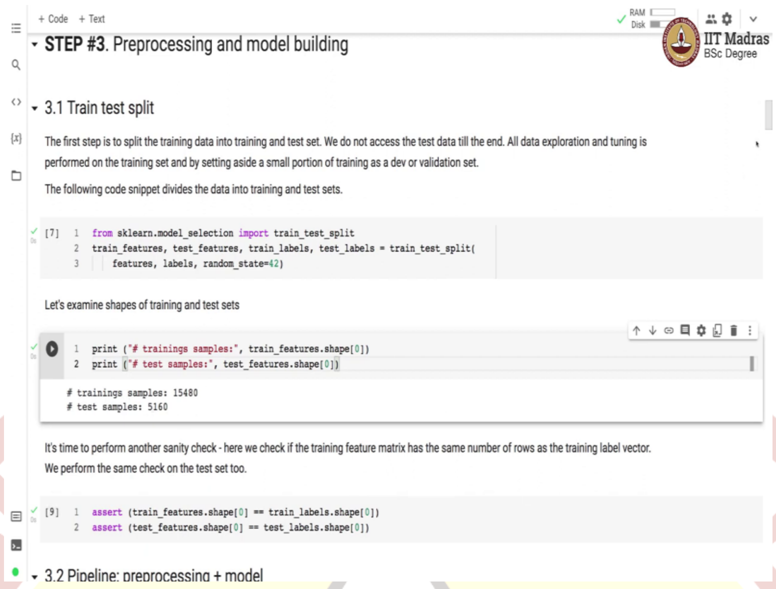
STEP #3. Preprocessing and model building

The first step is to load the data. We have already discussed how to load the California housing dataset in the last colab demonstration. So, here we use Fetch California Housing API. And we set as the frame is equal to true and return_x_y parameter to true. So, because of this parameter, we only get features and labels. And since we are setting as frame equal to true, we get both these return values as a data frame.

So we have a feature data frame and we have a label data frame. So let us examine the shapes of the feature matrix and the label vector. We can see that we have 20,640 examples each with 8 features. And in the label vector, we have 20,640 entries. As a sanity check, we need to make sure that the number of rows in the feature matrix and labels match. And we can do that with assert statement by looking at the zeroth element of our shape in the feature matrix and label vector.

The second step is to perform data exploration which has been covered in detail in the last colab demonstration where we studied the California Housing dataset.

(Refer Slide Time: 5:21)



STEP #3. Preprocessing and model building

3.1 Train test split

The first step is to split the training data into training and test set. We do not access the test data till the end. All data exploration and tuning is performed on the training set and by setting aside a small portion of training as a dev or validation set.

The following code snippet divides the data into training and test sets.

```
[7]  from sklearn.model_selection import train_test_split
     train_features, test_features, train_labels, test_labels = train_test_split(
         features, labels, random_state=42)
```

Let's examine shapes of training and test sets

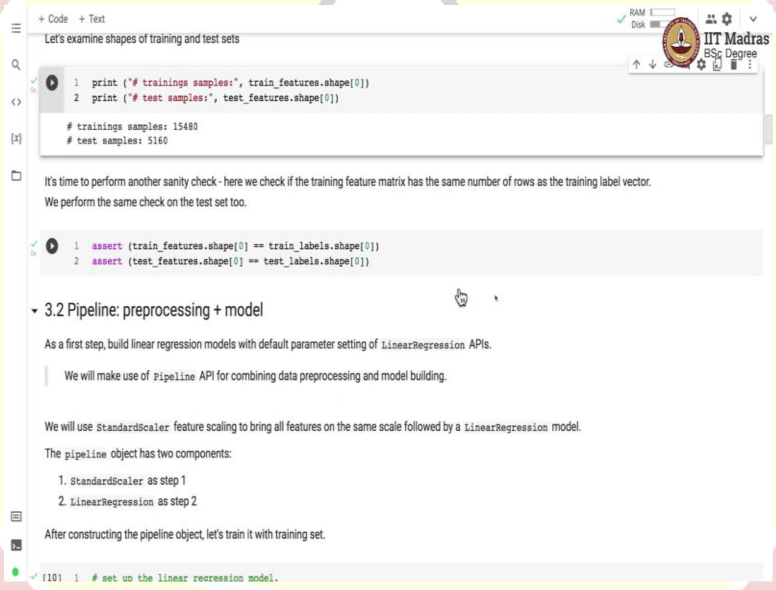
```
print ("# trainings samples:", train_features.shape[0])
print ("# test samples:", test_features.shape[0])
```

```
# trainings samples: 15480
# test samples: 5160
```

It's time to perform another sanity check - here we check if the training feature matrix has the same number of rows as the training label vector. We perform the same check on the test set too.

```
[9]  assert (train_features.shape[0] == train_labels.shape[0])
     assert (test_features.shape[0] == test_labels.shape[0])
```

3.2 Pipeline: preprocessing + model



Let's examine shapes of training and test sets

```
print ("# trainings samples:", train_features.shape[0])
print ("# test samples:", test_features.shape[0])
```

```
# trainings samples: 15480
# test samples: 5160
```

It's time to perform another sanity check - here we check if the training feature matrix has the same number of rows as the training label vector. We perform the same check on the test set too.

```
assert (train_features.shape[0] == train_labels.shape[0])
assert (test_features.shape[0] == test_labels.shape[0])
```

3.2 Pipeline: preprocessing + model

As a first step, build linear regression models with default parameter setting of `LinearRegression` APIs.

> We will make use of `Pipeline` API for combining data preprocessing and model building.

We will use `StandardScaler` feature scaling to bring all features on the same scale followed by a `LinearRegression` model.

The `pipeline` object has two components:

1. `StandardScaler` as step 1
2. `LinearRegression` as step 2

After constructing the pipeline object, let's train it with training set.

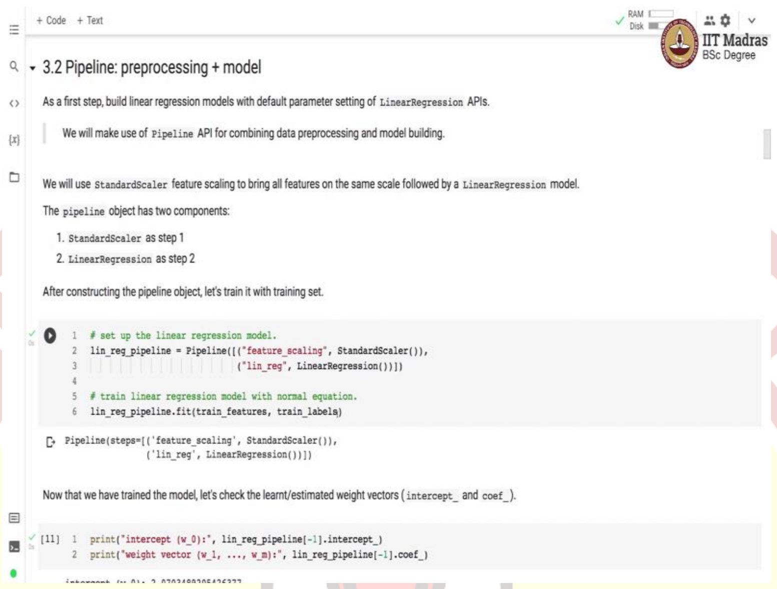
```
[10]  # set up the linear regression model.
```

In the next part of Step 3, where we are going to do preprocessing and model building, we start by splitting the training data into training and test sets. We do not access this data till the end, or data exploration and tuning are performed on the training set. And by setting aside a small portion of training set as a dev or validation set. So we perform train test split by using a training test split class from sklearn.model selection module.

The train test split splits the given feature matrix and label vector into training and test feature matrices and train and test label vectors. Again, in the similar manner as feature metrics and label vector, we also examine the shapes of training and test sets. Here we have got 15,480 examples in training and 5160 example in test.

We also perform a sanity check to make sure that the training feature matrix has the same number of rows as training the label vector. And also, we perform the same check in the test set 2. So, we use again assert statements to make sure that these conditions are met.

(Refer Slide Time: 7:00)



3.2 Pipeline: preprocessing + model

As a first step, build linear regression models with default parameter setting of `LinearRegression` APIs.

> We will make use of `Pipeline` API for combining data preprocessing and model building.

We will use `StandardScaler` feature scaling to bring all features on the same scale followed by a `LinearRegression` model.

The `pipeline` object has two components:

1. `StandardScaler` as step 1
2. `LinearRegression` as step 2

After constructing the pipeline object, let's train it with training set.

```
# set up the linear regression model.
lin_reg_pipeline = Pipeline([("feature_scaling", StandardScaler()),
                             ("lin_reg", LinearRegression())])

# train linear regression model with normal equation.
lin_reg_pipeline.fit(train_features, train_labels)
```

```
Pipeline(steps=[('feature_scaling', StandardScaler()),
                ('lin_reg', LinearRegression())])
```

Now that we have trained the model, let's check the learnt/estimated weight vectors (`intercept_` and `coef_`).

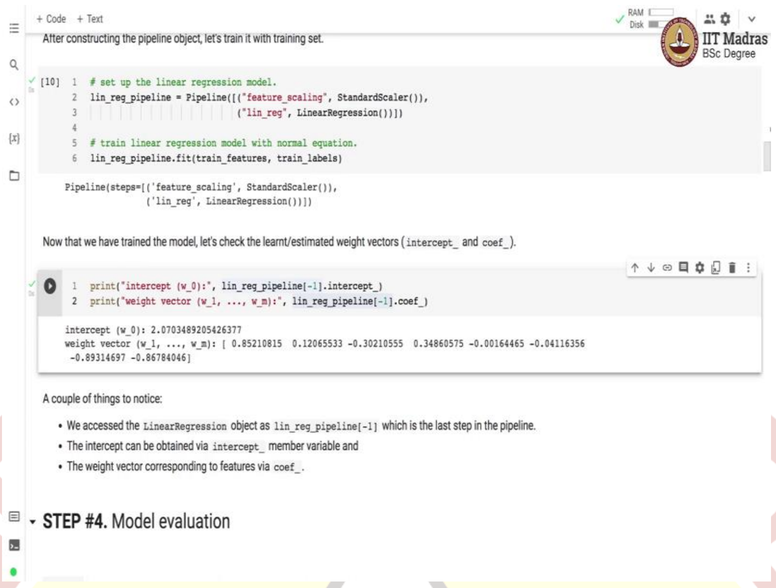
```
[11]
print("intercept (w_0):", lin_reg_pipeline[-1].intercept_)
print("weight vector (w_1, ..., w_m):", lin_reg_pipeline[-1].coef_)
```

Next, we build a linear model with default parameter settings of linear regression API. We will make use of pipeline API for combining data preprocessing and model building. We will use a standard scaler for scaling all the features to the same scale followed by the linear regression model.

So, this pipeline that we have built here has two parts. The first part is feature scaling, where we use standard scalar objects or standard scalar class. Here we instantiate the object of standard scalar plus, which helps us to bring all features on the same scale.

Then in the second step of the pipeline, we have linear regression object instantiated, which helps us to perform linear regression with normal equation. Once this pipeline is constructed, we can train the pipeline by calling the Fit function on the pipeline object. And we supply the training feature metrics and training label vector as parameters to the Fit function.

(Refer Slide Time: 8:10)


After constructing the pipeline object, let's train it with training set.

```
[10]
# set up the linear regression model.
lin_reg_pipeline = Pipeline([("feature_scaling", StandardScaler()),
                             ("lin_reg", LinearRegression())])

# train linear regression model with normal equation.
lin_reg_pipeline.fit(train_features, train_labels)
```

```
Pipeline(steps=[('feature_scaling', StandardScaler()),
                ('lin_reg', LinearRegression())])
```

Now that we have trained the model, let's check the learnt/estimated weight vectors (`intercept_` and `coef_`).

```
print("intercept (w_0):", lin_reg_pipeline[-1].intercept_)
print("weight vector (w_1, ..., w_m):", lin_reg_pipeline[-1].coef_)
```

```
intercept (w_0): 2.0703489205426377
weight vector (w_1, ..., w_m): [ 0.85210815  0.12065533 -0.30210555  0.34860575 -0.00164465 -0.04116356
 -0.89314697 -0.86784046]
```

A couple of things to notice:

- We accessed the `LinearRegression` object as `lin_reg_pipeline[-1]` which is the last step in the pipeline.
- The intercept can be obtained via `intercept_` member variable and
- The weight vector corresponding to features via `coef_`.

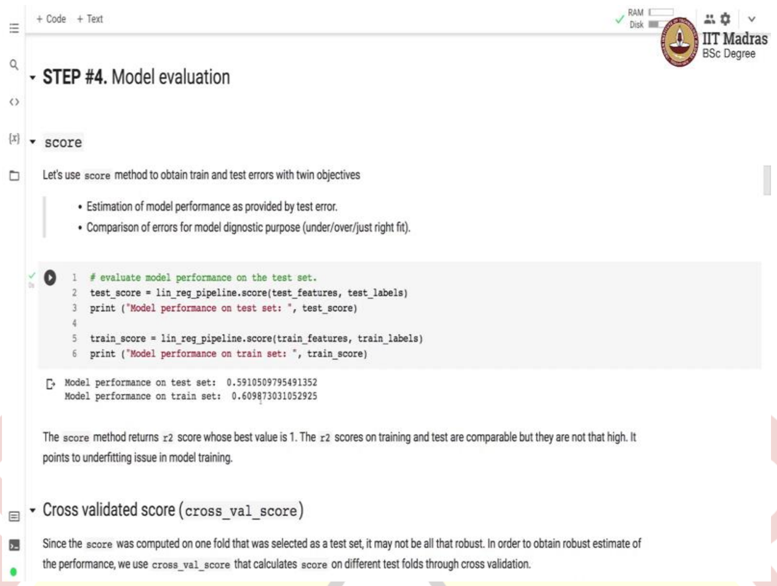
STEP #4. Model evaluation

Now that we have trained a model, let us check the estimated weight vectors. And these weight vectors can be accessed through the intercept_and coef_member variable of the linear regression object. So, here note that in the pipeline, we have two stages, first is feature scaling and the second is linear regression. So, we can access the linear regression object by using the index in the linear regression pipeline object of the pipeline class.

So, we can, we can access this particular estimator by lin_reg_pipeline of -1. And you are very well aware of how indexing works in Python. So, since we are trying to access -1 index, which is the last step, in this particular pipeline, which is linear regression. So, this particular expression gives us the linear regression object. And then on that object we call intercept, underscore, which is a member variable of this object that gives us the weight for the intercept, that is w0.

And when we access the coef_member variable of the same object, we get weights for the remaining parameters. So intercept, w0 is 2.07. And a weight vector that contains M entries w1, w2, all the way up to Wm, where Wm is a number of features. And since there are 8 features, you will see eight entries over here in the weight vector.

(Refer Slide Time: 10:04)



STEP #4. Model evaluation

score

Let's use `score` method to obtain train and test errors with twin objectives

- Estimation of model performance as provided by test error.
- Comparison of errors for model dignostic purpose (under/over/just right fit).

```
# evaluate model performance on the test set.
test_score = lin_reg_pipeline.score(test_features, test_labels)
print ("Model performance on test set: ", test_score)

train_score = lin_reg_pipeline.score(train_features, train_labels)
print ("Model performance on train set: ", train_score)
```

```
Model performance on test set:  0.5910509795491352
Model performance on train set:  0.609873031052925
```

The `score` method returns `r2` score whose best value is 1. The `r2` scores on training and test are comparable but they are not that high. It points to underfitting issue in model training.

Cross validated score (`cross_val_score`)

Since the `score` was computed on one fold that was selected as a test set, it may not be all that robust. In order to obtain robust estimate of the performance, we use `cross_val_score` that calculates `score` on different test folds through cross validation.

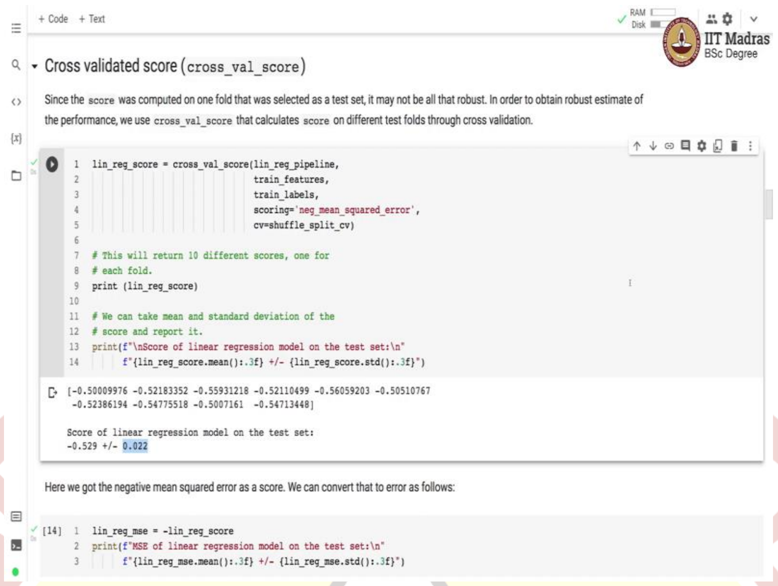
Now that model is trained, let us move on to the next step which is model evaluation. In evaluation, we will first use the scoring method to obtain train and test error with you between objectives. We will use the score obtained for estimation of model performance and at estimation is based on what we get in the test error. We also compare training and test error for model diagnostic purposes, where we want to figure out whether model is underfitting, overfitting or the model has got gotten the just tight fit.

So, what we do is, we call the score method on the pipeline object. So, what pipeline does it, it takes this feature matrix, first applies feature scaling and then called the predict and then it scores based on the predicted value and the actual value of the labels. So, we get this score and by default this score is r square or the coefficient of determination.

We perform the same thing with the training set. The score method returns r2 score. And since this is the score, the higher the better and you can see that we have slightly higher performance on the training set and the test set. And remember that the best value in r2 score is 1.

So, we are far from being the best, but we have a decent performance here. Another thing to note that the performance is comparable on test and training. But it is not that high and this could well point to the underfitting issue in the model.

(Refer Slide Time: 12:05)



Cross validated score (`cross_val_score`)

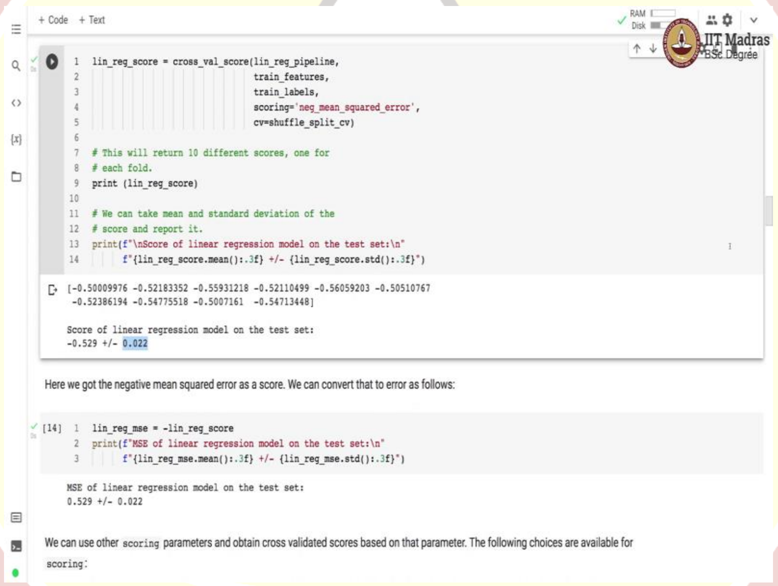
Since the `score` was computed on one fold that was selected as a test set, it may not be all that robust. In order to obtain robust estimate of the performance, we use `cross_val_score` that calculates `score` on different test folds through cross validation.

```python
lin_reg_score = cross_val_score(lin_reg_pipeline,
                                train_features,
                                train_labels,
                                scoring='neg_mean_squared_error',
                                cv=shuffle_split_cv)

# This will return 10 different scores, one for
# each fold.
print (lin_reg_score)

# We can take mean and standard deviation of the
# score and report it.
print(f"\nScore of linear regression model on the test set:\n"
      f"{lin_reg_score.mean():.3f} +/- {lin_reg_score.std():.3f}")
```

```
[-0.50009976 -0.52183352 -0.55931218 -0.52110499 -0.56059203 -0.50510767
 -0.52386194 -0.54775518 -0.5007161  -0.54713448]

Score of linear regression model on the test set:
-0.529 +/- 0.022
```

Here we got the negative mean squared error as a score. We can convert that to error as follows:

```python
[14]
lin_reg_mse = -lin_reg_score
print(f"MSE of linear regression model on the test set:\n"
      f"{lin_reg_mse.mean():.3f} +/- {lin_reg_mse.std():.3f}")
```



```python
lin_reg_score = cross_val_score(lin_reg_pipeline,
                                train_features,
                                train_labels,
                                scoring='neg_mean_squared_error',
                                cv=shuffle_split_cv)

# This will return 10 different scores, one for
# each fold.
print (lin_reg_score)

# We can take mean and standard deviation of the
# score and report it.
print(f"\nScore of linear regression model on the test set:\n"
      f"{lin_reg_score.mean():.3f} +/- {lin_reg_score.std():.3f}")
```

```
[-0.50009976 -0.52183352 -0.55931218 -0.52110499 -0.56059203 -0.50510767
 -0.52386194 -0.54775518 -0.5007161  -0.54713448]

Score of linear regression model on the test set:
-0.529 +/- 0.022
```

Here we got the negative mean squared error as a score. We can convert that to error as follows:

```python
[14]
lin_reg_mse = -lin_reg_score
print(f"MSE of linear regression model on the test set:\n"
      f"{lin_reg_mse.mean():.3f} +/- {lin_reg_mse.std():.3f}")
```

```
MSE of linear regression model on the test set:
0.529 +/- 0.022
```

We can use other `scoring` parameters and obtain cross validated scores based on that parameter. The following choices are available for `scoring`:

In the score method, we have calculated the score on one fold that was selected as a test set. So that is why it may not be all that robust. In order to obtain a robust estimate of the performance, we use cross -validated score with cross_val_score API from sklearn.model_selection module. This helps us to calculate score on different test folds through cross -validation.

So, the way we call cross_val_score is by specifying the estimator, the training features, and labels, we can also specify what kind of scoring function we want to use, here we are specifying the negative mean squared error. So, this is negative of the mean squared error and this is scoring, so in scoring; the score higher the better and error if you are calculating error, the lower the error is a better error, is the better error.

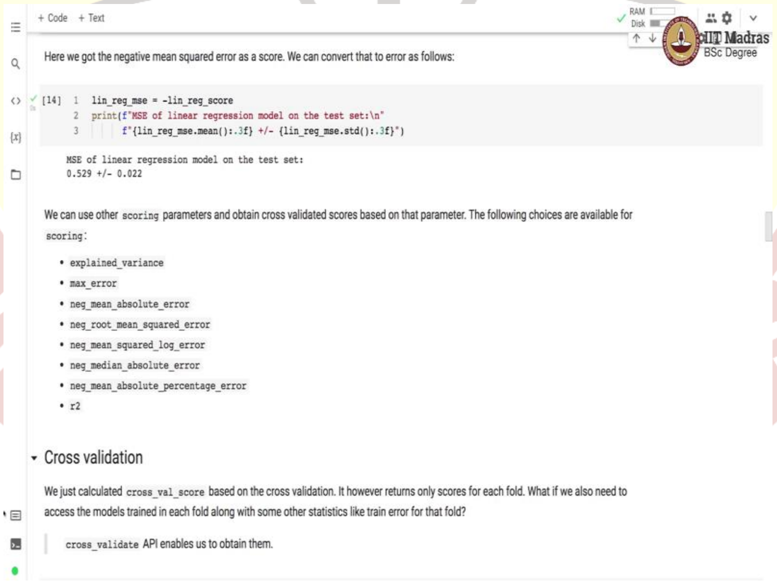
And we have seen this in our slides on this topic and we are going to use the shuffle split cv as a cross-validation strategy. Here we have initialized this cross shuffle split cv with 10 different folds. So, we will get 10 different scores one for each fold. For every fold, there are some examples that are set aside as test and we calculate this cross-validation score on that test set.

Since there are 10 different test sets, we get 10 different scores through this particular method. And these are 10 different scores that you can see here. And now since we have 10 different scores, we report the mean and standard deviation in this particular scores. So, we see that the negative mean squared error is the mean of that is -0.52. And the standard deviation is 0.022.

So, you can see that most of the scores are more or less comparable. And that is also that we can also verify through the standard deviation value which is fairly low. So, since we were calculating the score which is negative of the mean squared error, we got this particular number.

Now we want to convert let us say this score to the error we can convert it as follows. So we can take this particular score multiplied by -1 and we get the mean squared error. Now we can see that the mean square error is 0.52 and the standard deviation is 0.02. Here we use negative mean squared error as a scoring parameter.

(Refer Slide Time: 15:26)



Here we got the negative mean squared error as a score. We can convert that to error as follows:

```
[14]
lin_reg_mse = -lin_reg_score
print(f"MSE of linear regression model on the test set:\n"
      f"{lin_reg_mse.mean():.3f} +/- {lin_reg_mse.std():.3f}")
```

```
MSE of linear regression model on the test set:
0.529 +/- 0.022
```

We can use other `scoring` parameters and obtain cross validated scores based on that parameter. The following choices are available for `scoring`:

- `explained_variance`
- `max_error`
- `neg_mean_absolute_error`
- `neg_root_mean_squared_error`
- `neg_mean_squared_log_error`
- `neg_median_absolute_error`
- `neg_mean_absolute_percentage_error`
- `r2`

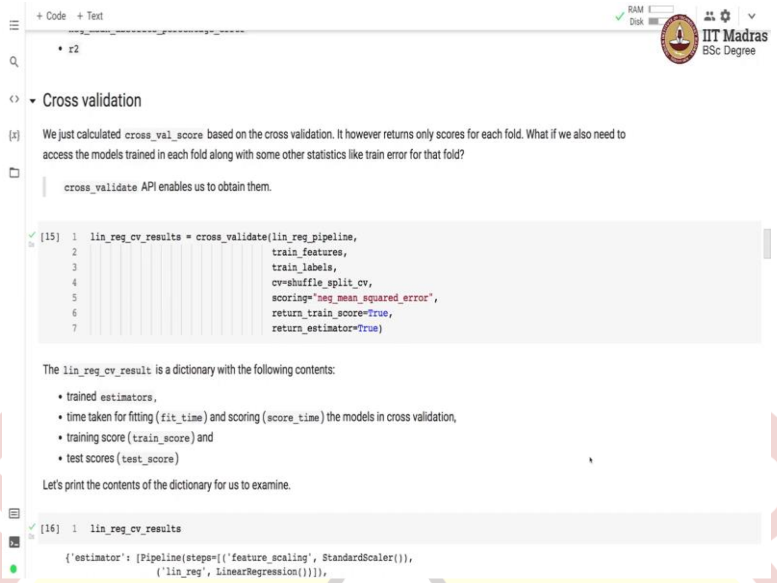
Cross validation

We just calculated `cross_val_score` based on the cross validation. It however returns only scores for each fold. What if we also need to access the models trained in each fold along with some other statistics like train error for that fold?

> `cross_validate` API enables us to obtain them.

There are some more choices that are available as scoring parameter, which is like explained variance, max error, negative mean absolute error, negative root mean squared error, negative mean squared log error, negative mean absolute error, negative mean absolute percentage error and r2. You have seen these, we have seen the choices in a bit more detail in the presentation on this topic.

(Refer Slide Time: 15:58)



- r2

Cross validation

We just calculated `cross_val_score` based on the cross validation. It however returns only scores for each fold. What if we also need to access the models trained in each fold along with some other statistics like train error for that fold?

> `cross_validate` API enables us to obtain them.

```
[15]
lin_reg_cv_results = cross_validate(lin_reg_pipeline,
                                    train_features,
                                    train_labels,
                                    cv=shuffle_split_cv,
                                    scoring="neg_mean_squared_error",
                                    return_train_score=True,
                                    return_estimator=True)
```

The `lin_reg_cv_result` is a dictionary with the following contents:

- trained `estimators`,
- time taken for fitting (`fit_time`) and scoring (`score_time`) the models in cross validation,
- training score (`train_score`) and
- test scores (`test_score`)

Let's print the contents of the dictionary for us to examine.

```
[16]
lin_reg_cv_results
```

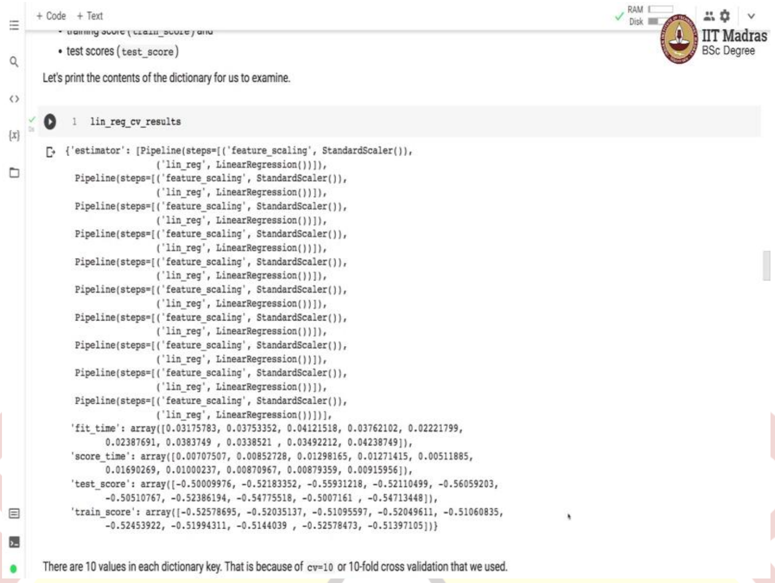
```
{'estimator': [Pipeline(steps=[('feature_scaling', StandardScaler()),
                ('lin_reg', LinearRegression())]),
```

So, the cross Val score just gave us score based on cross-validation. If we want to let us say access, the model is trained in each fold along with some of the statistics, like train error, we can use cross-validation API. And cross-validation API has pretty much the same parameters like cross val score, it takes the estimator, then the training, training feature metrics, training labels, the cross-validation strategy which is shuffled split cv, the scoring metric which is negative mean squared error.

And we also set these two parameters to true which are return train score and return estimator. So, return estimator when we set to true, we also get the estimators that were trained on the different fold. And when we said this particular parameter, we also get the score on the training folds.

And remember that these two by default are false since we need to explicitly set them to true. So as a result, we get a dictionary object that has got trained estimator time taken for fitting and scoring the models in cross-validation, we also have training score and the test score.

(Refer Slide Time: 17:21)



- test scores (`test_score`)

Let's print the contents of the dictionary for us to examine.

```
lin_reg_cv_results
```

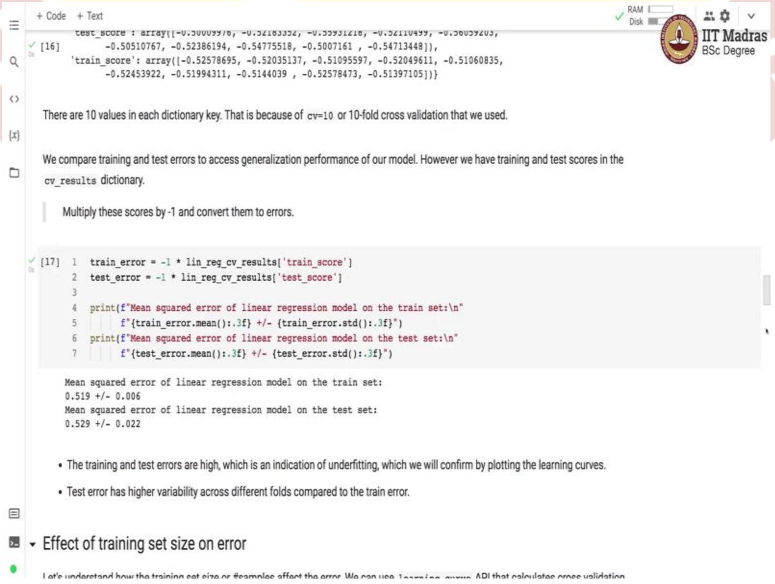
```
{'estimator': [Pipeline(steps=[('feature_scaling', StandardScaler()),
                ('lin_reg', LinearRegression())]),
  Pipeline(steps=[('feature_scaling', StandardScaler()),
                ('lin_reg', LinearRegression())]),
  Pipeline(steps=[('feature_scaling', StandardScaler()),
                ('lin_reg', LinearRegression())]),
  Pipeline(steps=[('feature_scaling', StandardScaler()),
                ('lin_reg', LinearRegression())]),
  Pipeline(steps=[('feature_scaling', StandardScaler()),
                ('lin_reg', LinearRegression())]),
  Pipeline(steps=[('feature_scaling', StandardScaler()),
                ('lin_reg', LinearRegression())]),
  Pipeline(steps=[('feature_scaling', StandardScaler()),
                ('lin_reg', LinearRegression())]),
  Pipeline(steps=[('feature_scaling', StandardScaler()),
                ('lin_reg', LinearRegression())]),
  Pipeline(steps=[('feature_scaling', StandardScaler()),
                ('lin_reg', LinearRegression())]),
  Pipeline(steps=[('feature_scaling', StandardScaler()),
                ('lin_reg', LinearRegression())])],
 'fit_time': array([0.03175783, 0.03753352, 0.04121518, 0.03762102, 0.02221799,
        0.02387691, 0.0383749 , 0.0338521 , 0.03492212, 0.04238749]),
 'score_time': array([0.00707507, 0.00852728, 0.01298165, 0.01271415, 0.00511885,
        0.01690269, 0.01000237, 0.00870967, 0.00879359, 0.00915956]),
 'test_score': array([-0.50009976, -0.52183352, -0.55931218, -0.52110499, -0.56059203,
        -0.50510767, -0.52386194, -0.54775518, -0.5007161 , -0.54713448]),
 'train_score': array([-0.52578695, -0.52035137, -0.51095597, -0.52049611, -0.51060835,
        -0.52453922, -0.51994311, -0.5144039 , -0.52578473, -0.51397105])}
```

There are 10 values in each dictionary key. That is because of `cv=10` or 10-fold cross validation that we used.

Let us print the content of the dictionary for us to examine. So, you can see that there is a key and a value. So there is this estimator, which has got 10 different estimators that were trained on 10 different folds, we have fit times which is again a list of 10 numbers. This was this is a time that was taken to fit the model in each fold, then the score time, in the same manner, the time it took to calculate the score in each fold.

Then we also get 10 different numbers here, which are test scores, as well as the train scores over here. And remember, these are the scores, and the higher the scores, the better the performance. So, you can see that our performance, in general, is better on the training set than the test set which is what you normally get.

(Refer Slide Time: 18:22)



```
        -0.50510767, -0.52386194, -0.54775518, -0.5007161 , -0.54713448]),
 'train_score': array([-0.52578695, -0.52035137, -0.51095597, -0.52049611, -0.51060835,
        -0.52453922, -0.51994311, -0.5144039 , -0.52578473, -0.51397105])}
```

There are 10 values in each dictionary key. That is because of `cv=10` or 10-fold cross validation that we used.

We compare training and test errors to access generalization performance of our model. However we have training and test scores in the `cv_results` dictionary.

> Multiply these scores by -1 and convert them to errors.

```
train_error = -1 * lin_reg_cv_results['train_score']
test_error = -1 * lin_reg_cv_results['test_score']

print(f"Mean squared error of linear regression model on the train set:\n"
      f"{train_error.mean():.3f} +/- {train_error.std():.3f}")
print(f"Mean squared error of linear regression model on the test set:\n"
      f"{test_error.mean():.3f} +/- {test_error.std():.3f}")
```

```
Mean squared error of linear regression model on the train set:
0.519 +/- 0.006
Mean squared error of linear regression model on the test set:
0.529 +/- 0.022
```

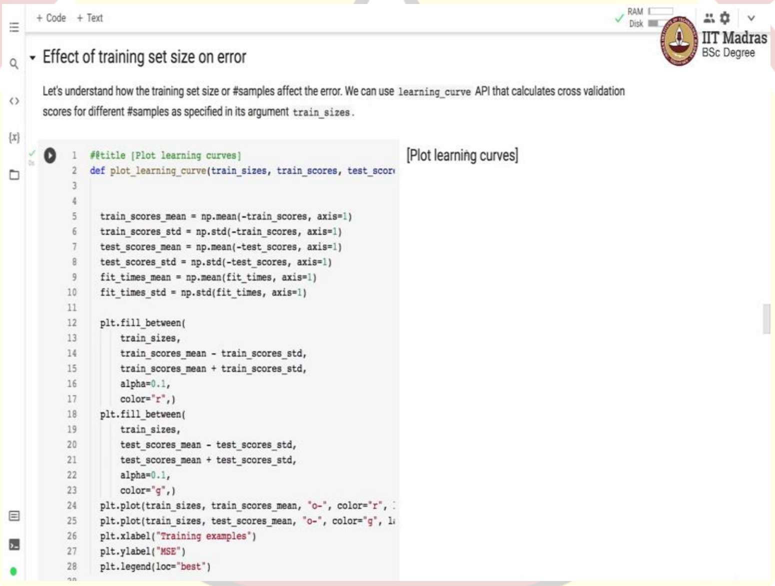
- The training and test errors are high, which is an indication of underfitting, which we will confirm by plotting the learning curves.
- Test error has higher variability across different folds compared to the train error.

Effect of training set size on error

We will compare the training and test error to assess the generalization performance of our model. However, we have training, training and test scores in cv results dictionary, we multiply these errors, we multiply the scores by -1 and convert them to errors. So, we multiply train score and test score by -1 and that way we have now the errors. So, the mean squared error for the training set is 0.51 and for the test set it is 0.52

So, in training set there is a smaller standard deviation than the test error. So, training has got a better error or the lower error, hence training performs better than the test. And these scores are comparable, but they are still high. And that actually points us to possibly the problem of underfitting. We will confirm that by plotting the learning curves.

(Refer Slide Time: 19:34)



Effect of training set size on error

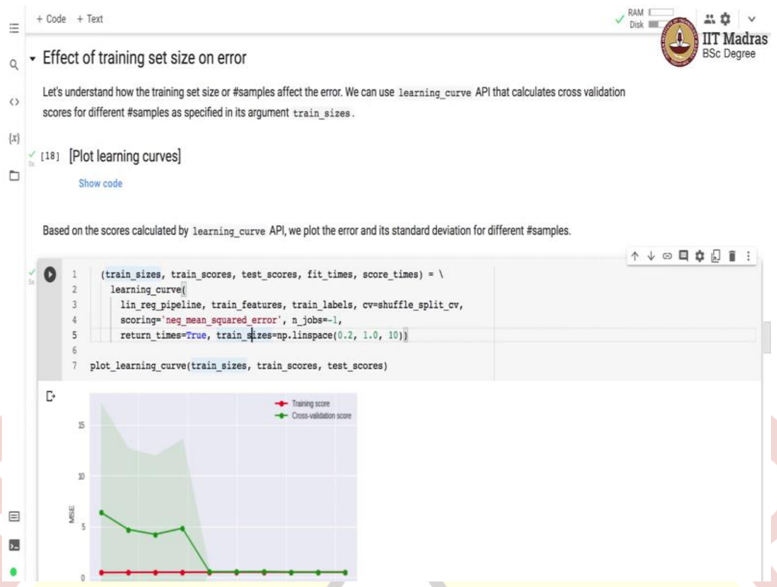
Let's understand how the training set size or #samples affect the error. We can use `learning_curve` API that calculates cross validation scores for different #samples as specified in its argument `train_sizes`.

[Plot learning curves]

```
#@title [Plot learning curves]
def plot_learning_curve(train_sizes, train_scores, test_scor


  train_scores_mean = np.mean(-train_scores, axis=1)
  train_scores_std = np.std(-train_scores, axis=1)
  test_scores_mean = np.mean(-test_scores, axis=1)
  test_scores_std = np.std(-test_scores, axis=1)
  fit_times_mean = np.mean(fit_times, axis=1)
  fit_times_std = np.std(fit_times, axis=1)

  plt.fill_between(
      train_sizes,
      train_scores_mean - train_scores_std,
      train_scores_mean + train_scores_std,
      alpha=0.1,
      color="r",)
  plt.fill_between(
      train_sizes,
      test_scores_mean - test_scores_std,
      test_scores_mean + test_scores_std,
      alpha=0.1,
      color="g",)
  plt.plot(train_sizes, train_scores_mean, "o-", color="r",
  plt.plot(train_sizes, test_scores_mean, "o-", color="g", l
  plt.xlabel("Training examples")
  plt.ylabel("MSE")
  plt.legend(loc="best")
```

The learning curves have got slightly different meanings in the context of sklearn. In the machine learning techniques course, we talked about learning curves as the iteration on the x-axis and the laws on the y axis. So, here this is a slightly different kind of learning curve where we have the training set size on x-axis and we have the error on Y-axis. So, what has really changed is the x-axis. Here, instead of iteration, we have training set size as the x-axis.

(Refer Slide Time: 20:17)



Effect of training set size on error

Let's understand how the training set size or #samples affect the error. We can use `learning_curve` API that calculates cross validation scores for different #samples as specified in its argument `train_sizes`.

[18] [Plot learning curves]

Show code

Based on the scores calculated by `learning_curve` API, we plot the error and its standard deviation for different #samples.

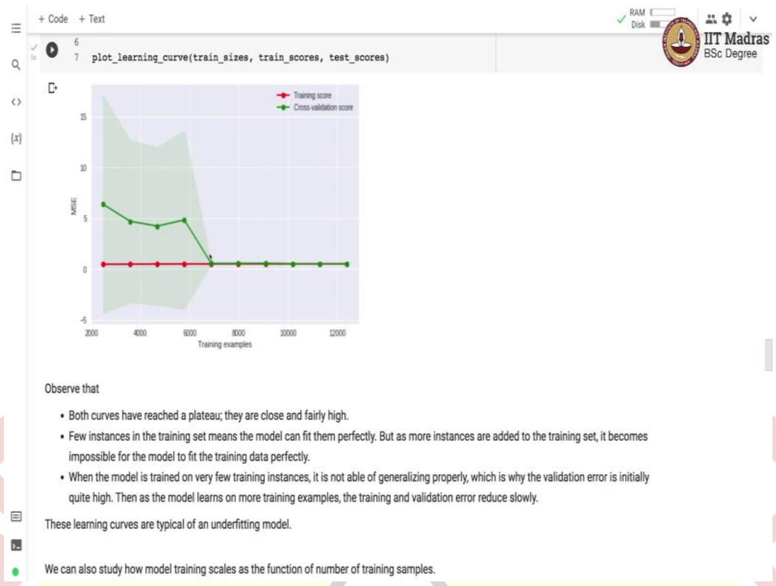
```
(train_sizes, train_scores, test_scores, fit_times, score_times) = \
  learning_curve(
    lin_reg_pipeline, train_features, train_labels, cv=shuffle_split_cv,
    scoring='neg_mean_squared_error', n_jobs=-1,
    return_times=True, train_sizes=np.linspace(0.2, 1.0, 10))

plot_learning_curve(train_sizes, train_scores, test_scores)
```

So for this, we will be using; for this, we will be using learning curve API that calculates cross-validation scores for the different number of samples as specified in the argument which is training_sizes. Let us check out the learning curve API. Meanwhile, here, there is a code snippet for plotting the learning curve and we are hiding it because this has got code for plotting the learning curves.

So, let us see how to call the learning curve API. So, the learning curve has got estimator training, and test feature metrics and labels. In the cross-validation strategy, the scoring metric is the number of jobs, which specifies whether we should be using parallelism and how much parallelism we should be using in this particular API. And we set return times to true and we specified training sizes. So, all these arguments up to this point, are exactly the same as cross val score or cross-validate.

So, we now have return times and training sizes and training sizes is the most important argument because here we have to specify what is the range of values that we should be used in training. So, we use values between 0.2 and 1 and there is 10 such kind of values that will be used.

(Refer Slide Time: 21:38)



```
plot_learning_curve(train_sizes, train_scores, test_scores)
```

Observe that

- Both curves have reached a plateau; they are close and fairly high.
- Few instances in the training set means the model can fit them perfectly. But as more instances are added to the training set, it becomes impossible for the model to fit the training data perfectly.
- When the model is trained on very few training instances, it is not able of generalizing properly, which is why the validation error is initially quite high. Then as the model learns on more training examples, the training and validation error reduce slowly.

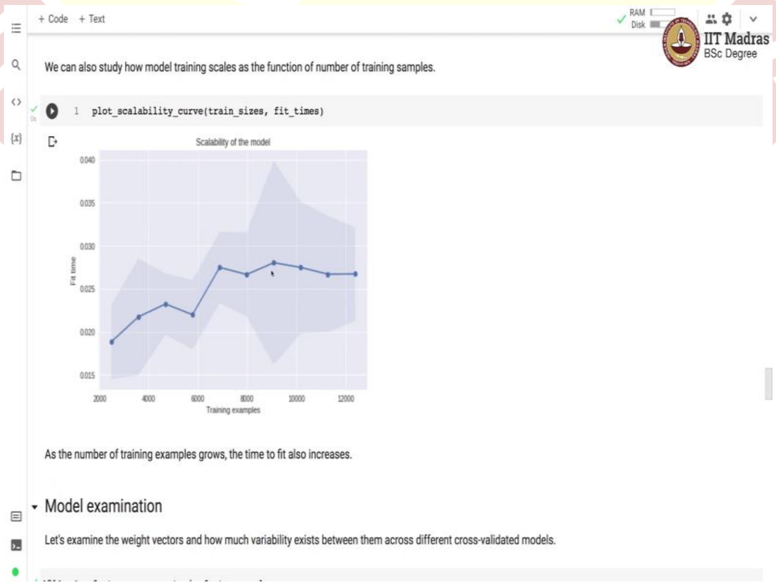
These learning curves are typical of an underfitting model.

We can also study how model training scales as the function of number of training samples.

We get some such kind of plot. After plotting the training examples versus mean squared error, we have training examples on the x-axis and mean squared error on the y-axis. So, you can see that there are 10 different points with varying numbers of training examples, and for each of these points, there is corresponding mean square error that you are plotting on the y axis. So, we can see that both curves have reached a plateau and are close and fairly high.

So, a few instances in the training means that model can fit those examples perfectly. And we can see that when the model is trained on a few instances, the error on the test set is quite high. And as we add more training examples, the test error gradually comes down. So, this learning curve is a type of underfitting model.

(Refer Slide Time: 22:47)



We can also study how model training scales as the function of number of training samples.

```
plot_scalability_curve(train_sizes, fit_times)
```

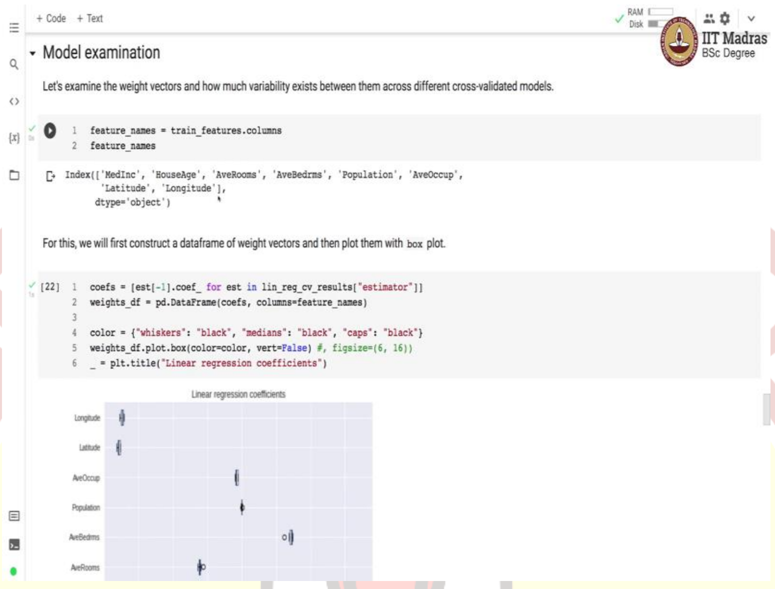
As the number of training examples grows, the time to fit also increases.

Model examination

Let's examine the weight vectors and how much variability exists between them across different cross-validated models.

We also study how the model training scales as a function of number of examples. So, we have fit times we and we plot the training examples versus fit times. And you can see that as the number of training example grows, the time that it takes to fit the model also increases.

(Refer Slide Time: 23:11)



Model examination

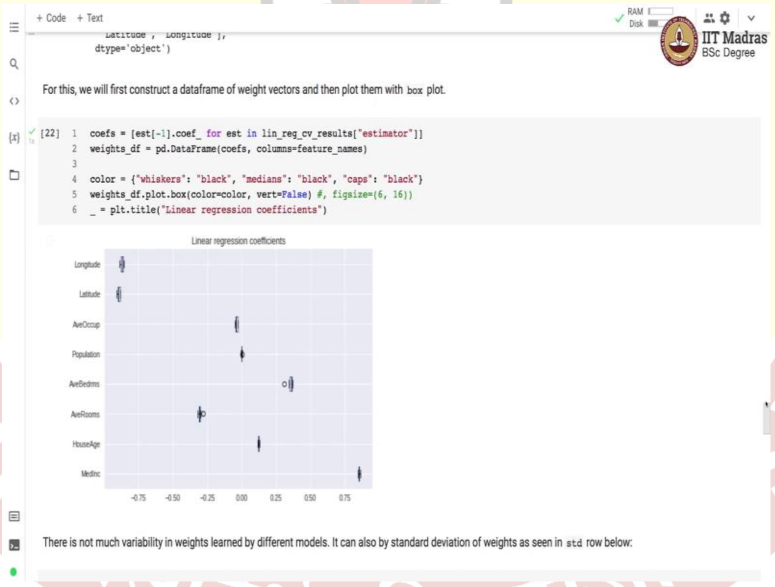
Let's examine the weight vectors and how much variability exists between them across different cross-validated models.

```
feature_names = train_features.columns
feature_names
```

```
Index(['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Population', 'AveOccup',
       'Latitude', 'Longitude'],
      dtype='object')
```

For this, we will first construct a dataframe of weight vectors and then plot them with `box` plot.

```
[22]
coefs = [est[-1].coef_ for est in lin_reg_cv_results["estimator"]]
weights_df = pd.DataFrame(coefs, columns=feature_names)

color = {"whiskers": "black", "medians": "black", "caps": "black"}
weights_df.plot.box(color=color, vert=False) #, figsize=(6, 16))
_ = plt.title("Linear regression coefficients")
```



dtype='object')

For this, we will first construct a dataframe of weight vectors and then plot them with `box` plot.

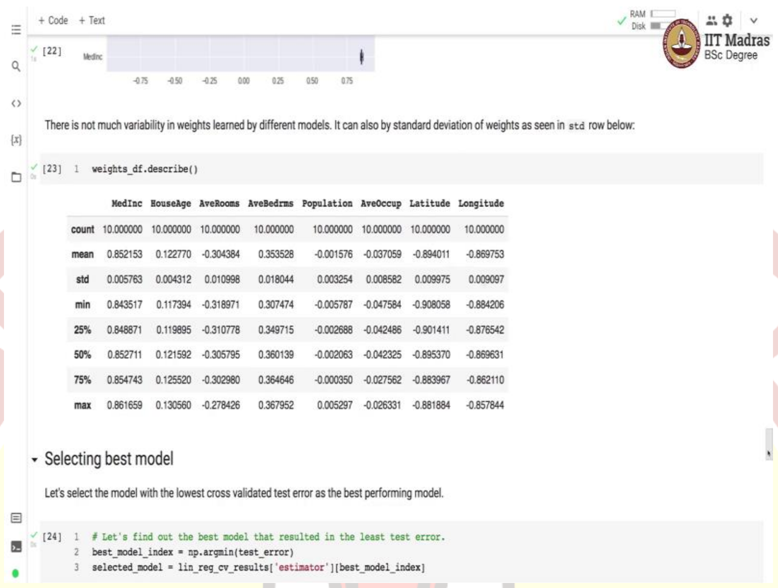
```
[22]
coefs = [est[-1].coef_ for est in lin_reg_cv_results["estimator"]]
weights_df = pd.DataFrame(coefs, columns=feature_names)

color = {"whiskers": "black", "medians": "black", "caps": "black"}
weights_df.plot.box(color=color, vert=False) #, figsize=(6, 16))
_ = plt.title("Linear regression coefficients")
```

There is not much variability in weights learned by different models. It can also by standard deviation of weights as seen in `std` row below:

Let us examine the weights and the variability in the weight vector across different folds in cross validation. So, here, we have a boxplot for linear regression coefficients that we obtain in in different folds. So, you can see that this is a boxplot for the weights that are assigned to longitude across different folds and the variability is quite small, because box plot is quite compact here.

Whereas, in average bedroom, average rooms and population there are there seems to be some outlier weights. In most of the cases, the boxplot is quite compact, that means there is a lesser variability in the weights that we are obtained, that we obtained across different folds.

(Refer Slide Time: 24:21)



There is not much variability in weights learned by different models. It can also by standard deviation of weights as seen in `std` row below:

```
weights_df.describe()
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
count	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000
mean	0.852153	0.122770	-0.304384	0.353528	-0.001576	-0.037059	-0.894011	-0.869753
std	0.005763	0.004312	0.010998	0.018044	0.003254	0.008582	0.009975	0.009097
min	0.843517	0.117394	-0.318971	0.307474	-0.005787	-0.047584	-0.908058	-0.884206
25%	0.848871	0.119895	-0.310778	0.349715	-0.002688	-0.042486	-0.901411	-0.876542
50%	0.852711	0.121592	-0.305795	0.360139	-0.002063	-0.042325	-0.895370	-0.869631
75%	0.854743	0.125520	-0.302980	0.364646	-0.000350	-0.027562	-0.883967	-0.862110
max	0.861659	0.130560	-0.278426	0.367952	0.005297	-0.026331	-0.881884	-0.857844

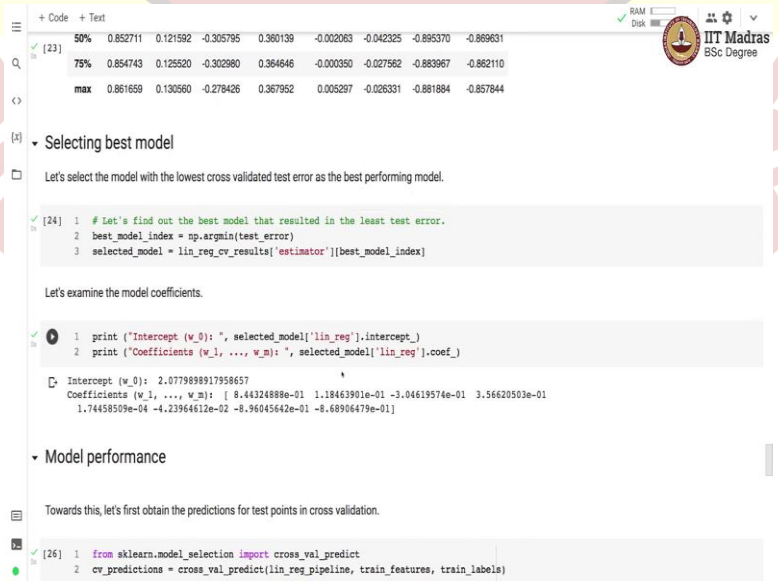
Selecting best model

Let's select the model with the lowest cross validated test error as the best performing model.

```
# Let's find out the best model that resulted in the least test error.
best_model_index = np.argmin(test_error)
selected_model = lin_reg_cv_results['estimator'][best_model_index]
```

And we can check that by also calling the description on the weights data frame. And you can see that the standard deviation in most of the cases is fairly small, except for these two cases it is higher than the 10 other cases. And you can also see the presence of outliers in the case of its population where the seventy-fifth percentile and the max have some difference right.

(Refer Slide Time: 25:09)



50%	0.852711	0.121592	-0.305795	0.360139	-0.002063	-0.042325	-0.895370	-0.869631
75%	0.854743	0.125520	-0.302980	0.364646	-0.000350	-0.027562	-0.883967	-0.862110
max	0.861659	0.130560	-0.278426	0.367952	0.005297	-0.026331	-0.881884	-0.857844

Selecting best model

Let's select the model with the lowest cross validated test error as the best performing model.

```
# Let's find out the best model that resulted in the least test error.
best_model_index = np.argmin(test_error)
selected_model = lin_reg_cv_results['estimator'][best_model_index]
```

Let's examine the model coefficients.

```
print ("Intercept (w_0): ", selected_model['lin_reg'].intercept_)
print ("Coefficients (w_1, ..., w_m): ", selected_model['lin_reg'].coef_)
```

```
Intercept (w_0):  2.0779898917958657
Coefficients (w_1, ..., w_m):  [ 8.44324888e-01  1.18463901e-01 -3.04619574e-01  3.56620503e-01
  1.74458509e-04 -4.23964612e-02 -8.96045642e-01 -8.68906479e-01]
```

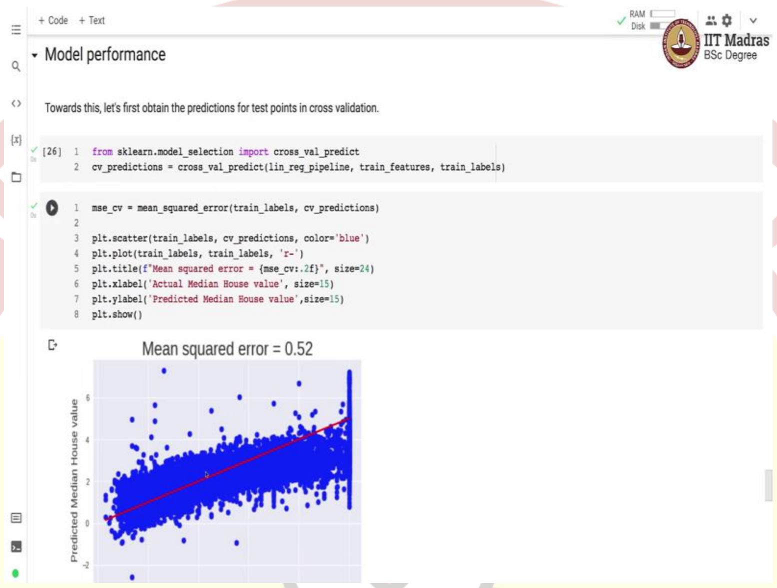
Model performance

Towards this, let's first obtain the predictions for test points in cross validation.

```
from sklearn.model_selection import cross_val_predict
cv_predictions = cross_val_predict(lin_reg_pipeline, train_features, train_labels)
```

So, we have selected, we have actually obtained multiple models in cross validation, and we need to select one of the one model which is which gives us the best performance on the validation set. So, we can we can find out the best model by calculating the one that results in the least error, we can found that by calculate, by calculating the np.argmin on the test error, we get the index of the best model and we can access the best model but from the from the, from the dictionary that was obtained in cross validate. We can examine the model coefficients of the selected model in the same way as we examined the coefficients earlier in the colab.

(Refer Slide Time: 26:06)



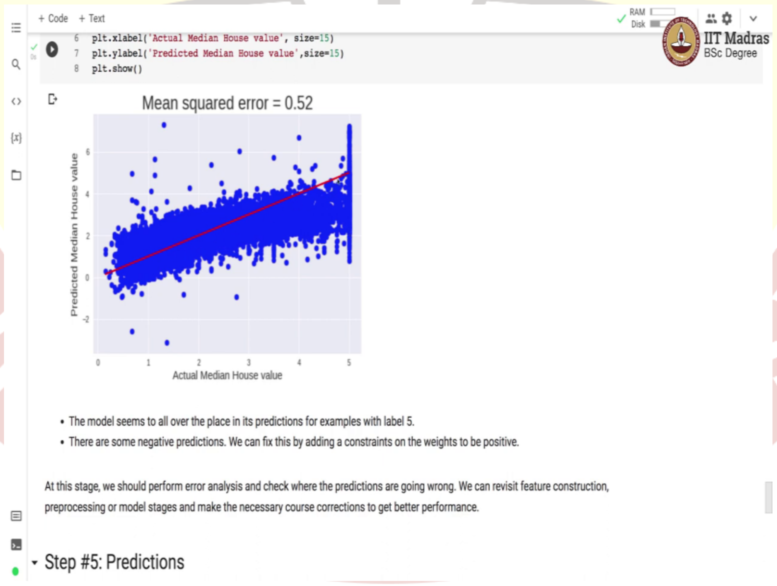
Model performance

Towards this, let's first obtain the predictions for test points in cross validation.

```
[26]
from sklearn.model_selection import cross_val_predict
cv_predictions = cross_val_predict(lin_reg_pipeline, train_features, train_labels)
```

```
mse_cv = mean_squared_error(train_labels, cv_predictions)

plt.scatter(train_labels, cv_predictions, color='blue')
plt.plot(train_labels, train_labels, 'r-')
plt.title(f"Mean squared error = {mse_cv:.2f}", size=24)
plt.xlabel('Actual Median House value', size=15)
plt.ylabel('Predicted Median House value',size=15)
plt.show()
```

Mean squared error = 0.52
Predicted Median House value



```
plt.xlabel('Actual Median House value', size=15)
plt.ylabel('Predicted Median House value',size=15)
plt.show()
```

Mean squared error = 0.52
Predicted Median House value
Actual Median House value

* The model seems to all over the place in its predictions for examples with label 5.
* There are some negative predictions. We can fix this by adding a constraints on the weights to be positive.

At this stage, we should perform error analysis and check where the predictions are going wrong. We can revisit feature construction, preprocessing or model stages and make the necessary course corrections to get better performance.

Step #5: Predictions

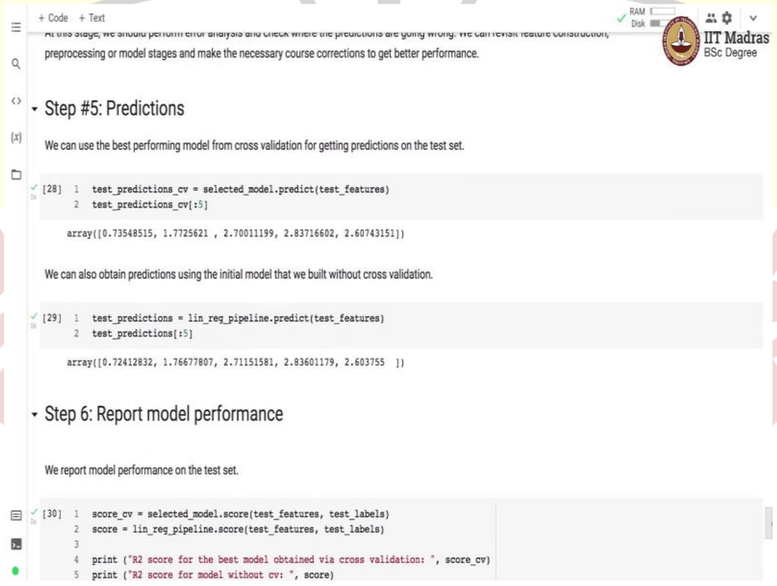
Finally, we also want to find out how the predictions on the test points look like compared to their actual values. So, we use cross val predict, we use cross val predict API in order to get the prediction on the test points. This particular API gives us the score for points when they were part of the test set in cross validation. We calculate the mean squared error based on the actual training labels and the labels that we have obtained in cross validation. And we simply plot the graph of the actual label and the predicted label.

Ideally, if we have perfect classification, all the labels should be along this 45-degree line, but we can see that most of the points are scattered around this particular line. And that shows the amount of error that we have in prediction.

So, if you carefully examine this graph, you will find that the model seems to be all over the place for this particular value which is label 5, for this value 5 we are predicting all possible values and there are some negative values that did not make much sense because house price cannot be negative. We can fix this by adding the constraint and the weights to be positive.

So, at this stage normally in the data in your machine learning project, we perform error analysis and check whether predictions are going wrong. Then we can revisit the model construction through feature by offering additional features you know, applying different preprocessing features, or changing the model. So, in that way, we make a necessary course correction to get better performance in the next run. So, you are where the iterative process of model building comes in.

(Refer Slide Time: 28:38)



preprocessing or model stages and make the necessary course corrections to get better performance.

Step #5: Predictions

We can use the best performing model from cross validation for getting predictions on the test set.

```
[28] test_predictions_cv = selected_model.predict(test_features)
     test_predictions_cv[:5]
```

```
array([0.73548515, 1.7725621 , 2.70011199, 2.83716602, 2.60743151])
```

We can also obtain predictions using the initial model that we built without cross validation.

```
[29] test_predictions = lin_reg_pipeline.predict(test_features)
     test_predictions[:5]
```

```
array([0.72412832, 1.76677807, 2.71151581, 2.83601179, 2.603755  ])
```

Step 6: Report model performance

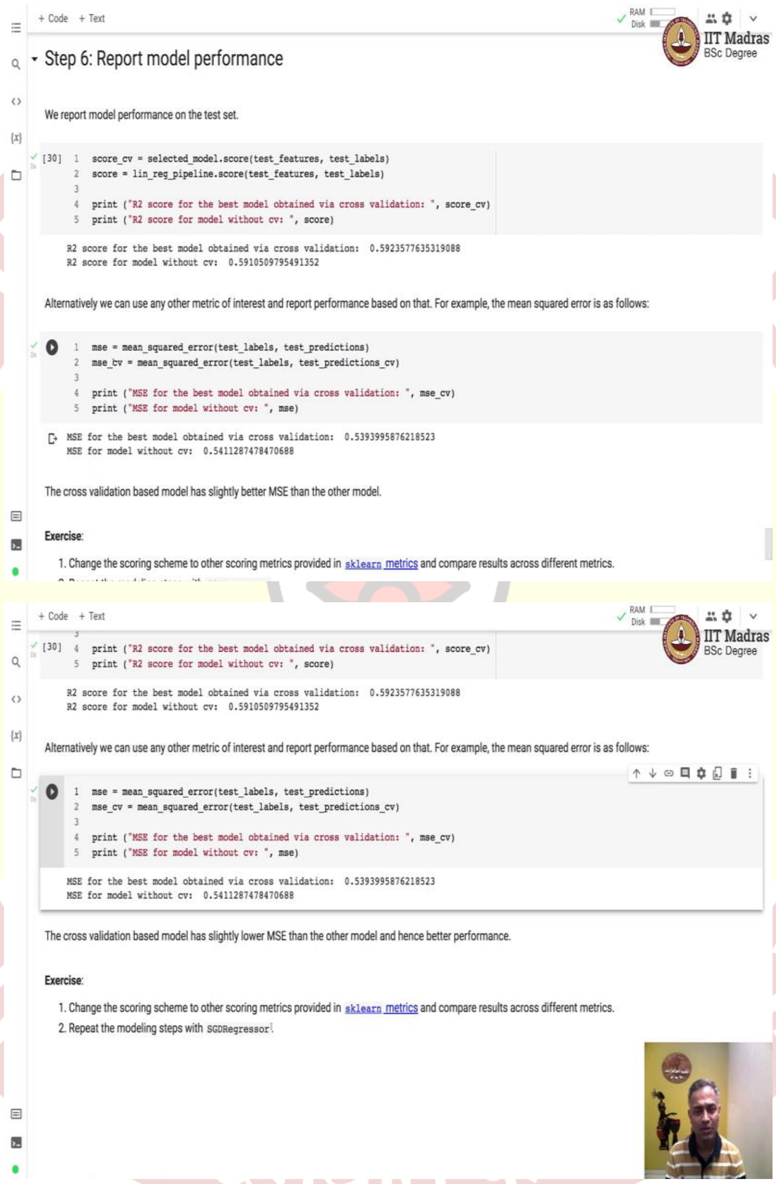
We report model performance on the test set.

```
[30] score_cv = selected_model.score(test_features, test_labels)
     score = lin_reg_pipeline.score(test_features, test_labels)

     print ("R2 score for the best model obtained via cross validation: ", score_cv)
     print ("R2 score for model without cv: ", score)
```

And once you finally build a model, we obtain the predictions on the test set. And remember, this test set was set aside we did not use it throughout the training process. And now we are using the test process, now we are using the test set, now that the training is over in order to estimate how the model will perform on the future examples.

So, we call the predict function on the selected model by supplying the test feature matrix to it we get the test predictions. If the test labels are available to you, you can also perform, we can also plot such a plot. Normally in real life, the test labels may not be available when you are using the model in production. Hence, we have not added it over here.

(Refer Slide Time: 29:31)



Step 6: Report model performance

We report model performance on the test set.

```
[30]
score_cv = selected_model.score(test_features, test_labels)
score = lin_reg_pipeline.score(test_features, test_labels)

print ("R2 score for the best model obtained via cross validation: ", score_cv)
print ("R2 score for model without cv: ", score)
```

```
R2 score for the best model obtained via cross validation:  0.5923577635319088
R2 score for model without cv:  0.5910509795491352
```

Alternatively we can use any other metric of interest and report performance based on that. For example, the mean squared error is as follows:

```
mse = mean_squared_error(test_labels, test_predictions)
mse_cv = mean_squared_error(test_labels, test_predictions_cv)

print ("MSE for the best model obtained via cross validation: ", mse_cv)
print ("MSE for model without cv: ", mse)
```

```
MSE for the best model obtained via cross validation:  0.5393995876218523
MSE for model without cv:  0.5411287478470688
```

The cross validation based model has slightly better MSE than the other model.

Exercise:

1. Change the scoring scheme to other scoring metrics provided in sklearn metrics and compare results across different metrics.

```
[30]
print ("R2 score for the best model obtained via cross validation: ", score_cv)
print ("R2 score for model without cv: ", score)
```

```
R2 score for the best model obtained via cross validation:  0.5923577635319088
R2 score for model without cv:  0.5910509795491352
```

Alternatively we can use any other metric of interest and report performance based on that. For example, the mean squared error is as follows:

```
mse = mean_squared_error(test_labels, test_predictions)
mse_cv = mean_squared_error(test_labels, test_predictions_cv)

print ("MSE for the best model obtained via cross validation: ", mse_cv)
print ("MSE for model without cv: ", mse)
```

```
MSE for the best model obtained via cross validation:  0.5393995876218523
MSE for model without cv:  0.5411287478470688
```

The cross validation based model has slightly lower MSE than the other model and hence better performance.

Exercise:

1. Change the scoring scheme to other scoring metrics provided in sklearn metrics and compare results across different metrics.
2. Repeat the modeling steps with `SGDRegressor`.

Finally, we report the model performance on the test set. And for that we use the scoring method. We can also use any other metric of our interest like mean squared error or mean absolute error. And all the scoring metrics take the actual labels and predicted labels and provide us the metric

Now here we obtained two metric one with, the one with cv and another one without cv. You can observe that the cross validation base model has slightly lower mean squared error than the model without cross validation. And hence this model performs we can conclude that the cross validator model performs better than the one without cross validation.

So, as an exercise, I would request you to change the scoring schemes and compare the results across different metrics. Another exercise is to repeat all the modeling step with SGDRegressor API that implements that that trains the linear regression model through iterative optimization. So, this was the demonstration of linear regression model with the normal equation.

We have tried to use different modeling steps in order to give you the flavor of different scale in APIs that are available at our disposal. In the next video, we will look at how to build a baseline model. Till then, Namaste and goodbye.