

IIT Madras
ONLINE DEGREE

Machine Learning Practice
Professor Doctor Ashish Tendulkar
Indian Institute of Technology, Madras
Model Evaluation

(Refer Slide Time: 0:12)



Model evaluation



Namaste, welcome to the next video of Machine Learning practice course, in this video we will discuss how to evaluate linear regression models in sklearn.

(Refer Slide Time: 0:25)



General steps in model evaluation

STEP 1: Split data into train and test

```
1 from sklearn.model_selection import train_test_split  
2 X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

STEP 2: Fit linear regression estimator on training set.

STEP 3: Calculate training error (a.k.a. empirical error)

STEP 4: Calculate test error (a.k.a. generalization error)

Compare training and test errors



Let us look at general steps in model evaluation. We first split the data into training and test. For that, we use the `train_test_split` class from `sklearn.model selection` module, `train_test_split` takes

the feature metric x and label vector y or it could also be label metric in case of multi-output regression and a random seed. And it returns the train feature metric test, feature metric, train labels, and test labels.

We fit the linear regression estimator on the training set. We calculate training error which is also called empirical error. And finally, we calculate test error which is called as generalization error. Then we compare training and test error to understand whether our model is the right fit or whether it is underfitting or overfitting.

(Refer Slide Time: 1:40)

How to evaluate trained Linear Regression model?

Using `score` method on linear regression object:

```
1 # Evaluation on the eval set with
2 # 1. feature matrix
3 # 2. label vector or matrix (single/multi-output)
4 linear_regressor.score(X_test, y_test)
```

The score returns R^2 or coefficient of determination

$$R^2 = \left(1 - \frac{u}{v}\right)$$

residual sum of squares:
 $u = (Xw - y)^T (Xw - y)$
Sum of squared error (actual and predicted label)

total sum of square
Sum of squared error (actual and mean predicted label)
 $v = (y - \hat{y}_{\text{mean}})^T (y - \hat{y}_{\text{mean}})$

So, let us study how to evaluate a trained linear regression model in sklearn. Sklearn provides score method on estimator objects, we are going to use this score method for evaluating the trained linear regression model. So, what we do is we call the score method on the regression estimator, we supply the feature matrix and a label vector or matrix of the test set.

And the score method returns a quantity which is known as R square or coefficient of determination. Let us see what this R square is, let us see how this R square is computed. So, the R square is computed as $(1 - u/v)$, where u is residual sum of squares and this is a familiar equation to us x w gives us the predicted labels, y is an actual label and this is nothing but the difference between the predicted label and actual label.

We take the transpose of this different set and multiply it by itself. So, this is like finding the square of the difference between the predicted labels and actual levels, which is what is the loss function

for the linear regression. On the other hand, v is total sum of square and this v is computed as a difference between the actual label-the mean of the predicted label, we take transpose and multiply this difference vector with itself.

So, this is square of difference between the actual label and the mean of the predicted label. So, here it was a predicted label but here we take mean of the predicted label and that is how we calculate what is called as sum of squared errors. So, now this is also sum of squared error but this is sum of squared error between actual and mean predicted label, whereas u is sum of squared label, sum of squared error between actual and a predicted label.

(Refer Slide Time: 4:18)

The score returns R^2 or coefficient of determination

$$R^2 = \left(1 - \frac{u}{v}\right)$$

When?

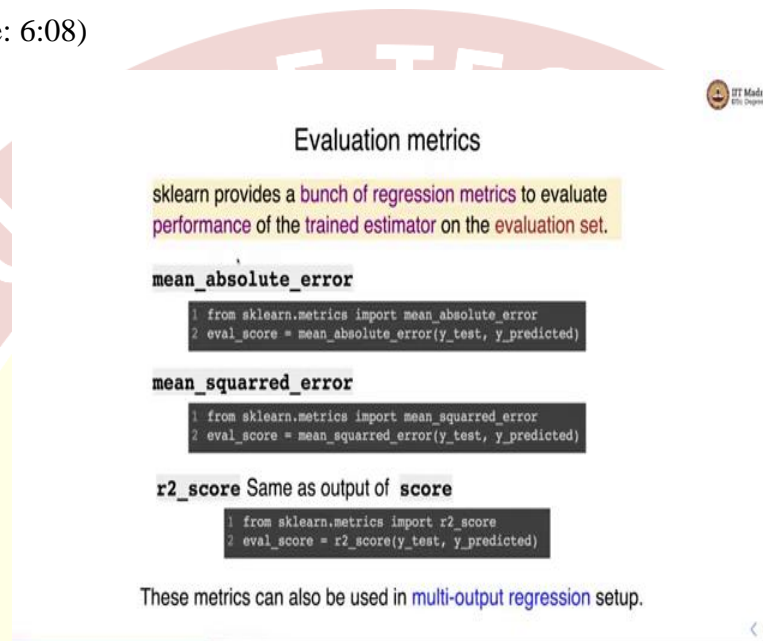
- The best possible score is 1.0. u , sum of squared error = 0
- A constant model that always predicts the expected value of y , would get a score of 0.0. $u = v$
- The score can be negative (because the model can be arbitrarily worse).

So, the R square has the best possible score of y . A constant model that always predicts the expected value of y would get a score of 0. And the score can be negative because the model can be arbitrarily worse. Let us see when these scores occur. So, we get the score of 1 when u that is the sum of squared error is 0. So, when u is 0 obviously this quantity is 0 so $1-0$ is 1, so we get R square of 1 when u is equal to 0.

A constant model that predicts the expected value of y would get score of 0 and that would happen when u is exactly like v , this is the total sum of squared error and this is a residual sum of squared error for this we calculate, this particular quantity is calculated as the difference between the actual label-mean of the predicted labels and this quantity is calculated as the difference between the predicted label and actual level and then we square it up.

So, when the predicted label is exactly equal to the mean of all predicted labels we will get into a situation where u will be exactly equal to v and when that happens this quantity becomes 1 and $1-1$ becomes 0 so R square returns the value of 0. And if model is arbitrary worse u is expected to be greater than v and then the score can become negative.

(Refer Slide Time: 6:08)



Evaluation metrics

sklearn provides a bunch of regression metrics to evaluate performance of the trained estimator on the evaluation set.

mean_absolute_error

```
1 from sklearn.metrics import mean_absolute_error
2 eval_score = mean_absolute_error(y_test, y_predicted)
```

mean_squared_error

```
1 from sklearn.metrics import mean_squared_error
2 eval_score = mean_squared_error(y_test, y_predicted)
```

r2_score Same as output of **score**

```
1 from sklearn.metrics import r2_score
2 eval_score = r2_score(y_test, y_predicted)
```

These metrics can also be used in multi-output regression setup.

So, apart from score so score method returns R square but there could be another regression metric that could be of our interest. Fortunately, sklearn provides a bunch of such regression metrics for us to evaluate the performance of the trend estimator on the evaluation set, let us look at those evaluation metrics.

The first one is mean absolute error. So, all these metrics are implemented as part of sklearn.metric module, we can import these metrics one by one. So, from sklearn. metric import mean absolute error and we calculate mean absolute error by supplying the actual labels and the predicted labels.

So, this particular function would return the score that is mean absolute error between the predicted values, between the predicted values and the actual values. In the similar manner we have mean squared error which is again implemented as part of sklearn.metric package. And we calculate this mean squared error exactly like mean absolute error where we supply the actual labels and the predicted labels.

So, you can see that all these metrics need two argument one is the actual label and second is a predicted label and then that particular metric is applied on this, on the supplied arguments and we

get the evaluation score. Then there is r2 score which gives same output as the score method and this r2 is exactly what we saw in the previous slide which is the coefficient of determination. So, we import r2_score from sklearn.metrics module and we supply the actual values and the predicted values to obtain the r2 score. So, all these metrics can also be used in multi output regression setup.

(Refer Slide Time: 8:35)



mean_squared_log_error

```
1 from sklearn.metrics import mean_squared_log_error
2 eval_score = mean_squared_log_error(y_test, y_predicted)
```

- Useful for **targets with exponential growths** like population, sales growth etc,
- **Penalizes under-estimation heavier than the over-estimation.**

mean_absolute_percentage_error

```
1 from sklearn.metrics import mean_absolute_percentage_error
2 eval_score = mean_absolute_percentage_error(y_test, y_predicted)
```

- **Sensitive to relative error.**

median_absolute_error

```
1 from sklearn.metrics import median_absolute_error
2 eval_score = median_absolute_error(y_test, y_predicted)
```

- **Robust to outliers**

Then you have an additional matrix that is not amenable to multi-output regression setup so one is mean squared log error and this particular metric is useful for targets with exponential growth so when our labels denote exponential growths like population sales growth of a company etc. then we use mean squared log error. So, this particular metric penalizes underestimation heavier than the overestimation.

Then we have mean absolute percentage error which is sensitive to relative error so when we want to evaluate our predictions that are sensitive to relative errors we should be using mean absolute percentage error, we also have a mean absolute error which is robust to outliers, it does not give a lot of importance to the outliers and our evaluation becomes robust to the outliers.

(Refer Slide Time: 9:37)

How to evaluate regression model on worst case error?



Use metrics `max_error`

Worst case error on train set can be calculated as follows:

```
1 from sklearn.metrics import max_error
2 train_error = max_error(y_train, y_predicted)
```

Worst case error on test set can be calculated as follows:

```
1 from sklearn.metrics import max_error
2 test_error = max_error(y_test, y_predicted)
```

This metrics can, however, be used only for **single output regression**. It **does not support multi-output regression**.



So, let us see how to evaluate the regression model on worst-case error. So, we can use metric `max_error` which gets us the worst-case error. Let us first calculate the worst-case error on the training set. So, we import `max_error` from `sklearn.metrics` and we supply the actual values and the predicted values, in this case, we supply the training labels and the predicted training labels and we calculate the worst-case training error.

The worst-case error on test data can also be calculated in a similar manner when we supply instead of training we have supplied test labels and the predicted test labels to get the worst-case error on the test set. This metric can only be used in single output regression; it does not support multi-output regression.

(Refer Slide Time: 10:40)



Scores and Errors

- Score is a metric for which higher value is better.
- Error is a metric for which lower value is better.

Convert error metric to score metric by adding `neg_` suffix.

Function	Scoring
<code>metrics.mean_absolute_error</code>	<code>neg_mean_absolute_error</code>
<code>metrics.mean_squared_error</code>	<code>neg_mean_squared_error</code>
<code>metrics.mean_squared_error</code>	<code>neg_root_mean_squared_error</code>
<code>metrics.mean_squared_log_error</code>	<code>neg_mean_squared_log_error</code>
<code>metrics.median_absolute_error</code>	<code>neg_median_absolute_error</code>

< >

So, while studying these metrics you will come across two terms, the ones that are ending in scores and the other metric that are ending in errors. So, the score is a metric for which a higher value is better and error is a metric for which a lower value is better. So, let us keep this convention in mind while evaluating regression models or for that matter any other estimator in any other estimator class.

So, we convert the error metric to square metric by adding `neg_` suffix. For example, these are some of the error functions that are there in sklearn and on the right-hand side, we see the corresponding scoring functions. For example, mean absolute error is the error function or is the error metric that is defined in a sklearn.metric module.

The corresponding scoring function is `neg_mean absolute error`. So, you can see that we have added a `neg_` suffix to this particular error metric and we got this particular scoring metric and you can see that pattern getting repeated throughout this table, so for `mean_square_error` the corresponding scoring metric is `neg_mean_square_error`. Then we have similar scoring corresponding scoring, scoring metric for mean squared error, mean squared log error, and median absolute error.

(Refer Slide Time: 12:24)



In case, we get comparable performance on train and test with this split, is this performance guaranteed on other splits too?

- Is test set sufficiently large?
 - In case it is small, the test error obtained may be unstable and would not reflect the true test error on large test set.
- What is the chance that the easiest examples were kept aside as test by chance?
 - This if happens would lead to optimistic estimation of the true test error.

We use cross validation for robust performance evaluation.



So, now in the case when we get comparable performance and training and test set with a particular split is this performance guaranteed on other scripts. So, you remember I mean we use `train_test_split` function to get a split on training and test set that was a single split that we got. So, what is the guarantee that the performance that we have gotten on this training and test split is also replicable or is also guaranteed on other splits?

So, there are a couple of questions is the test set sufficiently large? In case the test set is small the test error obtained may be unstable and it would not reflect the true test error on the larger test set. The second question is what is the chance that the easiest examples were kept aside as a test. And if this happens we would get an optimistic estimation of true test error. And both of these situations are not really good as far as evaluating the generalization error or the test error of the model.

And they will not give us the right kind of estimate about how the model will perform on unseen data when deployed. So, we need something better and fortunately, we have a cross-validation tool, we have studied cross-validation in detail in machine learning techniques class and here we will use this cross validation. So, cross validation what it does is it creates multiple training and test sets and it evaluates the performance on different test set, that way we are using different test splits and hopefully when we perform this evaluation of multiple test splits we get robust performance evaluation.

(Refer Slide Time: 14:32)

Cross-validation performs **robust evaluation** of model performance

- by **repeated splitting** and
- providing **many training and test errors**

This enables us to **estimate variability in generalization performance** of the model.

sklearn implements the following cross validation iterators

KFold

RepeatedKfold

LeaveOneOut

ShuffleSplit

So, cross-validation performs the robust evaluation of model performance by repeated splitting and providing many training and test errors. This enables us to estimate variability in the generalization performance of the model. So, sklearn implements the following cross-validation iterators, so we need to know how to perform the splits and there are various ways in which the splits can be made. And sklearn implements different cross-validation iterators.

So, there is KFold cross-validation iterator which is very similar to what we have studied in the machine learning techniques course where we divide data into K different folds or K different partitions and then we select K-1 partition for training and one partition for the test or for evaluation and we do this for different splits of these K partitions that way we get multiple training sets as well as evaluation sets.

Then we have what is, then we have another iterator called as repeated KFold this essentially repeats the KFold splitting. Then we have to leave one out which is a special case of KFold where K is equal to the number of training samples, so in leave one out we leave out exactly one example as a test and we use rest n-1 examples for training. So, we get n such different splits on which we can perform training as well as the evaluation. And finally, we have shuffle split which shuffles the training data before splitting and it does a random permutation of the data points and create training and test sets. And it does it repetitively for a specific number of times that is specified by the user.

(Refer Slide Time: 16:54)

How to obtain cross validated performance measure using KFold?



```
1 from sklearn.model_selection import cross_val_score
2 from sklearn.linear_model import linear_regression
3
4 lin_reg = linear_regression()
5 score = cross_val_score(lin_reg, X, y, cv=5)
```

- Uses KFold cross validation iterator, that divides training data into 5 folds.
- In each run, it uses 4 folds for training and 1 for evaluation.

Alternate way of writing the same thing

```
1 from sklearn.model_selection import cross_val_score
2 from sklearn.model_selection import KFold
3 from sklearn.linear_model import linear_regression
4
5 lin_reg = linear_regression()
6 kfold_cv = KFold(n_splits=5, random_state=42)
7 score = cross_val_score(lin_reg, X, y, cv=kfold_cv)
```



So, let us try to find out how to obtain cross-validated performance measure using KFold. So, here we use a `cross_val_score` api implemented in `sklearn.model_selection` module. So, we have linear regression estimator and on this linear regression estimator we want to calculate cross validation score.

So, we provide the name of the estimator, the feature matrix, the label, the label vector or label matrix depending on what kind of regression problem we are solving and we also specify number of folds in cross validation. When we specify a integer value to the `cv` parameter of `cross_val_score` it uses KFold cross validation iterator and divides the training data into the specified number of folds.

It uses four folds for training and one-fold for evaluation there is an alternate way of writing the same thing where we will be explicitly using KFold cross-validation iterator, so we import KFold from `sklearn.model_selection` module and we instantiate an object of a KFold iterator by specifying number of splits and a random seed.

Then in `cross_val_score` instead of specifying an integer value over here we specify the KFold object and when you specify the KFold object it performs the KFold, it actually does the cross validation with KFold iterator. So, these are exactly, these two code snippets exactly do the same thing.

In case we want to use different cross validation iterator we can simply choose one from the `model_selection` module, instantiate that object and specify that cross validation iterator in the `cv` parameter of cross validation score.

(Refer Slide Time: 19:10)

How to obtain cross validated performance measure using `LeaveOneOut`?

```
1 from sklearn.model_selection import cross_val_score
2 from sklearn.model_selection import LeaveOneOut
3 from sklearn.linear_model import linear_regression
4
5 lin_reg = linear_regression()
6 loocv = LeaveOneOut()
7 score = cross_val_score(lin_reg, X, y, cv=loocv)
```

which is same as

```
1 from sklearn.model_selection import cross_val_score
2 from sklearn.model_selection import KFold
3 from sklearn.linear_model import linear_regression
4
5 lin_reg = linear_regression()
6 n = X.shape[0]
7 kfold_cv = KFold(n_splits=n)
8 score = cross_val_score(lin_reg, X, y, cv=kfold_cv)
```

Let us see couple of examples of that, let us first see how to use leave one out cross validation iterator for calculating the cross-validator performance. So, here we import leave one out class from `sklearn.model_selection` module. So, you can see that all model underscore, all model selection related utilities are present in `model_selection` module of `sklearn`.

And this is because of the design principles of `sklearn` api that we studied in the first week of machine learning practice course. So, here what we do is we instantiate an object of leave one out cross validation iterator and that object is used as a parameter for the `cv` parameter, that object is used as a value of `cv` parameter in cross validation score. So, when we write this particular, when we execute this particular statement we get a score that is calculated using leave one out cross validation iterator.

Now, this is same as using `KFold` cross validation with number of splits equal to `n` which is essentially the number of samples in the training set. So, here what we do is we first calculate the number of samples in the training set and we set `n_splits` to `n` in `KFold` cross validation and then we calculate the cross-validation score with the object of `KFold` cross validation iterator as the value of `cv` parameter. Again, these two code snippets would do the same thing.

(Refer Slide Time: 21:11)

How to obtain cross validated performance measure using `ShuffleSplit`?



```
1 from sklearn.linear_model import linear_regression
2 from sklearn.model_selection import cross_val_score
3 from sklearn.model_selection import ShuffleSplit
4
5 lin_reg = linear_regression()
6 shuffle_split = ShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
7 score = cross_val_score(lin_reg, X, y, cv=shuffle_split)
```

It is also called **random permutation based cross validation strategy**.

- Generates user defined number of train/test splits.
- It is robust to class distribution.

In each iteration, it shuffles order of data samples and then splits it into train and test.



Let us look at how to obtain cross-validated performance using shuffle split. Again, the strategy is the same we import shuffle split, we instantiate object of a shuffle split by specifying number of splits the size of the test, and the random seed and we use that shuffle split object as the value of the cross-validation parameter in cross-validation score function.

Shuffle split is also called as random permutation. It generates user-defined number of train test split as specified in `n_splits` and it is robust to the class distribution. So, even if there are different class distributions if you do not have uniform cross, if we do not have uniform, if you do not have uniform class distribution even in that case shuffle splits work reasonably well. In each iteration, shuffle split shuffles the order of data samples and then splits it into training and test.

सिद्धिर्भवति कर्मजा

(Refer Slide Time: 22:22)

How to specify a performance measure in **cross_val_score**

```
1 from sklearn.linear_model import linear_regression
2 from sklearn.model_selection import cross_val_score
3 from sklearn.model_selection import ShuffleSplit
4
5 lin_reg = linear_regression()
6 shuffle_split = ShuffleSplit(n_splits=5, test_size=0.2, random_state=42)
7 score = cross_val_score(lin_reg, X, y, cv=shuffle_split,
8                          scoring='neg_mean_absolute_error')
```

scoring parameter can be set to one of the scoring schemes implemented in sklearn as follows

max_error	r2
neg_mean_absolute_error	neg_mean_squared_error
neg_mean_squared_log_error	neg_median_absolute_error
neg_root_mean_squared_error	

So far, we were using, we are using default scoring mechanism for calculating cross validation score. Now, here we will try to specify different scoring mechanisms. So, we have already seen some of the scoring metrics which correspond to error metric before few slides. So, scoring parameter can be set to one of the scoring schemes implemented in sklearn, so we can use max error, r2 or all these negative of the error metric that is negative mean absolute error, negative mean squared error, negative mean squared log error, negative median absolute error and negative root mean squared error.

So, we can substitute these values in the scoring parameter and we get that particular score by after doing the cross validation. So, since we are doing cross validation hopefully the scores are robust than done calculating this course just on a single training and test split.

(Refer Slide Time: 23:38)



How to obtain **test scores** from different folds?

```
1 from sklearn.model_selection import cross_validate
2 from sklearn.model_selection import ShuffleSplit
3
4 cv = ShuffleSplit(n_splits=40, test_size=0.3, random_state=0)
5 cv_results = cross_validate(
6     regressor, data, target, cv=cv, scoring='neg_mean_absolute_error')
```

The results are stored in python dictionary with the following keys:

fit_time
score_time
test_score
estimator (optional)
train_score (optional)

Let us see how to obtain test scores for different folds. So, here what we have done is we have used shuffle split cross validation iterator with number of splits to be 40, size of the test is 30 percent and random seed equal to 0. Then we calculate the cross, then we call a cross validate function, so instead of calling cross_val_score we are now calling cross_validate function with parameters which is the estimator which is a regressor, the data which is the feature, which is a feature matrix, target is the label, label vector or metric depending on whether we are solving single or multi-output regression problem. Then the cross-validation iterator and we also specify the scoring scheme.

Now, this cross_validate returns, it returns this result which is essentially a python dictionary with the following keys. It has got fit time, score time, test score, estimator and train score. Estimator and train score are optional and in order to obtain them we need to specify additional parameter to cross_validate, by default we do not get estimator and train_score.

So, fit_time gives the time that was required for fitting that particular training, for fitting the regression model on that training set. Score time is the time it took to get the score on the evaluation set. And test score is actually the score that was obtained on that particular evaluation set. And this score in this case will be negative in absolute error because that is what we have specified in the scoring parameter.

(Refer Slide Time: 25:42)



How to obtain **trained estimators** and **scores** on training data during cross validation?

- For trained estimator, set **return_estimator = True**
- For scores on training set, set **return_train_score = True**

```
1 from sklearn.model_selection import cross_validate
2 from sklearn.model_selection import ShuffleSplit
3
4 cv = ShuffleSplit(n_splits=40, test_size=0.3,
5                 random_state=0)
6 cv_results = cross_validate(
7     regressor, data, target,
8     cv=cv, scoring="neg_mean_absolute_error",
9     return_train_score=True,
10    return_estimator=True)
```

The estimators can be accessed through **estimator** key of the dictionary returned by **cross_validate**

< >

In the same manner we can also obtain trained estimators and scores for each fold. For obtaining trained estimator we need to set `return_estimator` parameter to true, it is false by default. For scores on the training set we need to set `return_train_score` parameter to true. Let us look at an example.

So, here we are calling the `cross_validate` function by specifying all those parameters that we saw in the previous slide in addition to that we are specifying couple of more parameters which is `return_train_score`, setting it to true and `return_estimator` and also setting that to true.

When we do this, we get in this particular dictionary which is `cv_result` we get different estimators that were trained on different splits, we also get the training scores. That helps us to compare training and test scores as well as we can also examine these estimators that were learnt on different splits. The estimators can be accessed to `estimator` key of this dictionary.

(Refer Slide Time: 26:56)

How to evaluate **multiple metrics** of regression in cross validation set up?



```
1 from sklearn.model_selection import cross_validate
2 from sklearn.model_selection import ShuffleSplit
3
4 cv = ShuffleSplit(n_splits=40, test_size=0.3,
5                 random_state=0)
6 cv_results = cross_validate(
7     regressor, data, target,
8     cv=cv,
9     scoring=["neg_mean_absolute_error", "neg_mean_squared_error"]
10    return_train_score=True,
11    return_estimator=True)
```

cross_validate allows us to specify multiple scoring metrics
unlike **cross_val_score**

So, cross validate, so cross validate function also allows us to specify multiple scoring metrics. This facility is not available in cross validation score function, so that is the main difference between cross validate and cross validation score. Cross validation score also only returns the score on, that only returns a score after cross validation, whereas cross validate returns a lot more information like it returns the score for every fold, it also returns the estimator and the training error that was, that it obtains on different folds while performing cross validation. So, in this case we have just specified couple of scoring parameters, one is negative mean absolute error and negative mean squared error.

(Refer Slide Time: 28:00)



How to study effect of #samples on training and test errors?

STEP 1: Instantiate an object of `learning_curve` class with `estimator`, `training data`, `size`, `cross validation strategy` and `scoring scheme` as arguments.

```
1 from sklearn.model_selection import learning_curve
2
3 results = learning_curve(
4     lin_reg, X_train, y_train, train_sizes=train_sizes, cv=cv,
5     scoring="neg_mean_absolute_error")
6 train_size, train_scores, test_scores = results[:3]
7 # Convert the scores into errors
8 train_errors, test_errors = -train_scores, -test_scores
```

STEP 2: Plot `training and test scores` as function of the `size` of training sets. And make assessment about model fitment: `under/overfitting` or `right fit`.

So, let us see how to study effect of number of samples on training and test errors. So, sklearn provide us, sklearn provides us with `learning_curve` class. This `learning_curve` class takes `estimator`, `training data`, `size` of the training, `cross validation strategy` and `scoring scheme` as arguments which is pretty much same as the `cross_validate` function.

So, we instantiate an object of learning curve by specifying the linear regression estimator training feature metric, label metrics, the size of the training, training data, then the cross-validation iterator and the scoring scheme. And we can get the training size, training score and test score by accessing first three values of these particular results that was returned by `learning_curve`.

We can convert since these are scoring schemes, in this scoring we are using basically the negative of error, so in order to obtain the actual errors we multiply each of these scores by -1. So, we multiply the trained score by -1, we get train error, we multiply test score by -1 we get test error. This is how you can convert scores into errors if you want to output the error due to a specific training or specific evaluation set for that estimator.

In step 2 we generally plot training and test course as function of the size of the training set and this plot helps us to make assessment about the model fitment. We can diagnose whether a model is just a right fit or whether it is under fitting or overfitting and then take appropriate measures to alleviate any of such kind of fitment concerns.

(Refer Slide Time: 30:23)

Underfitting/Overfitting diagnosis

STEP 1: Fit linear models with different number of features.

STEP 2: For each model, obtain training and test errors.

STEP 3: Plot #features vs error graph - one each for training and test errors.

STEP 4: Examine the graphs to detect under/overfitting.

We can replace #features with any other tunable hyperparameter to do this diagnosis for setting that hyperparameter to the appropriate value.



In order to perform underfitting and overfitting diagnosis we have this four-step process which we have also studied in machine learning techniques course. We will repeat, we will revise it over here and in the collab we will actually implement this and experience how it works in sklearn.

So, here the first step is to fit linear model with different number of features. For each model we obtain training and test error and then what we do is we plot number of features versus error graph. And in this error graph there will be two curves, one for training and second for test errors.

We examine these graphs to detect underfitting or overfitting issues. So, we can replace these number of features that we are using here in the step one with any other tunable hyperparameter. And when we do that we are able to find out whether the model is underfitting or overfitting or whether that is the right fit for what kind of values of that hyperparameter. And then we can set the appropriate value of the hyperparameter such that we are getting optimal evaluation error on the test set. So, we can follow this particular general process in order to tune the hyperparameters.

सिद्धिर्भवति कर्मजा