


IIT Madras
ONLINE DEGREE

Professor Doctor Ashish Tendulkar
Indian Institute of Technology, Madras
Linear regression

Namaste. Welcome to the next video of the Machine Learning Practice course. In this video, we will start discussing how to implement linear regression models with a scaler.

(Refer Slide Time: 0:21)



How to build baseline regression model?

DummyRegressor helps in creating a **baseline** for regression.

```
1 from sklearn.dummy import DummyRegressor
2
3 dummy_regr = DummyRegressor(strategy="mean")
4 dummy_regr.fit(X_train, y_train)
5 dummy_regr.predict(X_test)
6 dummy_regr.score(X_test, y_test)
```

- It makes a prediction as specified by the **strategy**.
- Strategy is based on **some statistical property** of the training set or **user specified value**.

Strategy	mean	median	quantile	constant
			↑	↑
			quantile	constant

We start our discussion by building what is called as a baseline regression model. As you know, any baseline model helps us to generate a reasonable baseline based on some kind of a common-sense prediction. Sklearn implements a DummyRegressor class that helps us in creating a baseline for regression.

Let us look at a sample code snippet. The DummyRegressor class is implemented by sklearn.dummy module. We instantiate a dummy regressor object by specifying the strategy for making the prediction. Here we have specified the strategy as mean. That means we will be making predictions based on the mean value of the label.

So, we learn this mean value of the label through the fit function on the training set. So, we look at all the training labels and learn what is the mean value of the prediction. We will use this mean value later to predict for the test samples. And we calculate the score due to this particular prediction by using the test data. The score returns R square or coefficient of determination. We will study the R square or coefficient of determination later in this module.

So, DummyRegressor makes a prediction as specified by the strategy. And the strategy is based on some statistical property of the training set or a user-specified value. There are four

strategies that are supported by DummyRegressor. First is a mean, second is the median, third is the quantile and fourth is the constant, which quantile to be used as a prediction is specified by a quantile parameter. And a constant value is also specified by the user through the constant parameter, the DummyRegressor constructor.

Now, that we have created a DummyRegressor baseline, let us start to do something more intelligent with the training data that we have and we will start actually building some linear regression models based on what we have studied in the theory course.

(Refer Slide Time: 2:50)

How is **Linear Regression** model trained?

Step 1: Instantiate **object** of a suitable **linear regression estimator** from one of the following two options

Normal equation

```
1 from sklearn.linear_model import LinearRegression
2 linear_regressor = LinearRegression()
```

Iterative optimization

```
1 from sklearn.linear_model import SGDRegressor
2 linear_regressor = SGDRegressor()
```

Step 2: Call **fit** method on **linear regression object** with **training feature matrix** and **label vector** as arguments.

```
1 # Model training with feature matrix X_train and label vector y_train
2 # label vector or matrix y_train
3 linear_regressor.fit(X_train, y_train)
```

Works for both single and multi-output regression.

Let us understand how to train a linear regression model in a scalar. In the first step, we instantiate the object of a suitable linear regression estimator from one of the following two options. So, we know that linear regression can be solved with one of the two methods; one is normal equation and the second is iterative optimization. So, let us first understand how to solve linear regression with the normal equation.

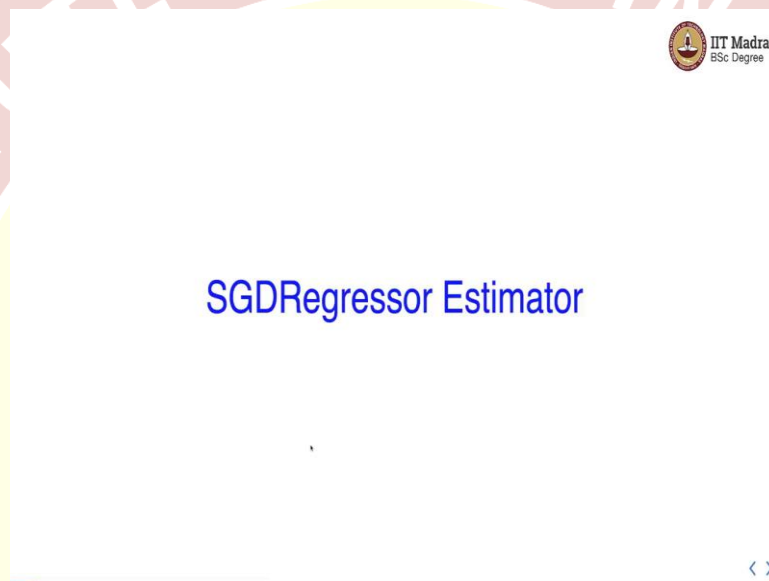
So, we first instantiate the object of a linear regression class, and this linear regression class is implemented in the `sklearn.linear_model` module. So, this is how we instantiate the linear regression object. The other option is to use iterative optimization. The iterative optimization is implemented by `SGDRegressor`. It is again implemented as part of a `sklearn.linear_model` module. So, we can either use linear regression for a normal equation or `SGDRegressor` for iterative optimization.

Second, we call `fit` method on linear regression object with training feature matrix and label vector as arguments. So, here, we use a linear regressor object which could be either from linear

regression or SGDRegressor and we call a fit method with two arguments one is `x_train`, which is a feature matrix and `y_train` is a label vector. And both the feature matrix and label vector come from the training set.

And this particular way of constructing a linear regression model works both for single and multi-output regression. In multi-output regression, the second argument in the fit which is `y_train` will be a label matrix instead of a label vector as in the single output case. And why would it be a label matrix because we have multiple outputs in the label?

(Refer Slide Time: 5:16)



Let us study SGDRegressor in a bit more detail because it provides a lot more flexibility than the linear regression class which is used to solve the normal equation method.

(Refer Slide Time: 5:33)



SGDRegressor Estimator

- Implements [stochastic gradient descent](#)
- Use for large training set up (> 10k samples)
- Provides [greater control on optimization process](#) through provision for [hyperparameter](#) settings.

• loss= 'squared error'
• loss = 'huber'

• penalty = 'l1'
• penalty = 'l2'
• penalty = 'elasticnet'

SGDRegressor

• learning_rate = 'constant'
• learning_rate = 'optimal'
• learning_rate = 'invscaling'
• learning_rate = 'adaptive'

• early_stopping = 'True'
• early_stopping = 'False'

So, the SGDRegressor estimator implements a stochastic gradient descent algorithm. It is used for large training set up with more than 10,000 samples. And SGDRegressor provides greater control on the optimization process through the provision of provision for different hyperparameter settings.

Let us look at different hyperparameters that we can specify in the linear in the SGDRegressor. So, we can configure SGDRegressor for different loss functions. There are two loss functions that we can use one is squared error, which is a standard loss function that we have also studied in the context of linear regression in the theory class. And second is the Huber loss, which we have not studied in the theory course. Huber loss is used for making the linear regression robust to outliers.

So, if you suspect that you have outliers in your data, you can use Huber loss for training to SGDRegressor. We can also specify different penalties in form of regularization. So, if you want to perform regularization inside an SGDRegressor we can set the penalty parameters. There are three penalty parameters that are supported one is l1 regularization, l2 regularization and elastic net. Elasticnet is a convex combination of l1 and l2 regularization.

Then we can also specify the learning rate. So, learning rate can be specified to a constant learning rate, optimal learning rate, inverse scaling, and adaptive learning rate. By default, SGDRegressor uses inverse scaling as a learning rate, as a way of deciding the learning rate for each iteration. But we can also specify the other three learning rates as well.

And finally, we can also provide the way to stop the SGDRegressor iterations. We have a parameter called early stopping that we can set to true or false. Early stopping helps us to stop the iterations of SGDRegressor once we reach a specified condition of convergence.

(Refer Slide Time: 8:14)



It's a good idea to use a **random seed** of your choice while instantiating SGDRegressor object. It helps us get **reproducible results**.

Set **random_state** to seed of your choice.

```
1 from sklearn.linear_model import SGDRegressor
2 linear_regressor = SGDRegressor(random_state=42)
```

Note: In the rest of the presentation, we won't set the random seed for sake of brevity. However while coding, always set the random seed in the constructor.

So, there is a small tip that it is a good idea to use a random seed of your choice while instantiating SGDRegressor object. It helps us in getting reproducible results. So, we can set the random seed with the parameter random_state in the constructor of the SGDRegressor. Here, I have set the random seed to 42, but you can set it to any value of your choice. In the rest of the presentation, we would not be setting the random seed for sake of brevity. However, while coding, it is recommended that you always set the random seed in the constructor of the SGDRegressor object.

(Refer Slide Time: 9:04)



How to perform feature scaling for SGDRegressor?

SGD is **sensitive to feature scaling**, so it is **highly recommended to scale** input feature matrix.

```
1 from sklearn.linear_model import SGDRegressor
2 from sklearn.pipeline import Pipeline
3 from sklearn.preprocessing import StandardScaler
4
5 sgd = Pipeline([
6     ('feature_scaling', StandardScaler()),
7     ('sgd_regressor', SGDRegressor())
8 ])
9 sgd.fit(X_train, y_train)
```

- Note**
- Feature scaling is **not needed** for **word frequencies** and **indicator features** as they have intrinsic scale.
 - Features extracted using PCA should be **scaled by some constant c** such that the average L2 norm of the training data equals one.

So, let us try to understand how to perform feature scaling for SGDRegressor. Remember that SGD is sensitive to feature scaling. Hence, it is highly recommended to scale the input feature matrix. We can perform feature scaling through the pipeline construct. We can use feature scaling; we can use one of the methods that we studied in data pre-processing for feature scaling.

Here we use standard scalar as a method for feature scaling. So, we have a standard scalar followed by SGDRegressor. So, what will happen is that in this pipeline, the input metrics are first scaled according to the standard scalar method and then SGDRegressor is applied to the input on the transformed feature matrix.

So, feature scaling is not needed for word frequencies or indicator features, these features are intrinsic scales, and they do not need any feature scaling. Features extracted using principal component analysis should be scaled by some constant c that helps us to make sure that the average L2 norm for the PCA transform feature matrix is equal to 1. And that way, the transform data through PCA can be used inside SGD.

(Refer Slide Time: 10:45)

How to shuffle training data after each epoch in SGDRegressor?



```
1 from sklearn.linear_model import SGDRegressor
2 linear_regressor = SGDRegressor(shuffle=True)
```

Let us try to understand how to shuffle training data after each epoch SGDRegressor. We can simply set the shuffle parameter in the SGDRegressor constructor to true that way the data will be shuffled before every epoch in the SGDRegressor.

(Refer Slide Time: 11:04)

How to use set learning rate in SGDRegressor?



- learning_rate = 'constant'
- learning_rate = 'invscaling'
- learning_rate = 'adaptive'

```
1 from sklearn.linear_model import SGDRegressor
2 linear_regressor = SGDRegressor(random_state=42)
```

What is the default setting?

- learning_rate = 'invscaling'
- eta0 = 1e-2
- power_t = 0.25

Learning rate reduces after every iteration:
$$\text{eta} = \text{eta0} / \text{pow}(t, \text{power_t})$$

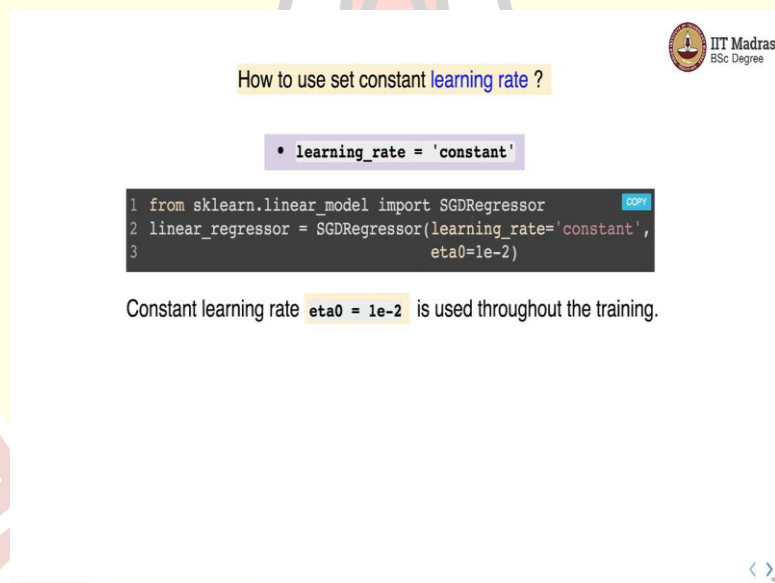
Note: You can make changes to these parameters to speed up or slow down the training process.

Let us try to understand how to set learning rates in SGDRegressor. So, we can set the learning rate to constant, inverse scaling, or adaptive. So, here the default setting for the learning rate is inverse scaling, we use eta 0 as 0.01 And the power_t hyper parameter as 0.25. So, there are these two parameters eta 0 and power_t, which are two important parameters that helps us to decide the learning rate SGDRegressor uses different learning rate in different iteration.

And that learning rate in inverse scaling is decided by the following formula. So, the eta that is learning rate for the iteration is calculated as η_0 which is the initial learning rate divided by power of t which is the current iteration. So, we raise the current iteration to power t . So, power t is another hyperparameter that we set for deciding the learning rate in every step while using inverse scaling learning rate.

So, we can make changes to these parameters that are η_0 and power t to speed up or slow down the learning process. And, and how do you know you know whether to speed up or slow down we can look at the learning curves we can look at the value of the loss function after every iteration. If the loss is changing too slowly, we need to speed up the speed of the training process and if there is some kind of oscillation in the loss, that means the learning rate is too high and we need ways to reduce it. And we can control the speeding up or slowdown of the learning rate through η_0 and power t parameters.

(Refer Slide Time: 13:20)



How to use set constant learning rate ?

- learning_rate = 'constant'

```
1 from sklearn.linear_model import SGDRegressor
2 linear_regressor = SGDRegressor(learning_rate='constant',
3                                eta0=1e-2)
```

Constant learning rate $\eta_0 = 1e-2$ is used throughout the training.

So, let us see how to set a constant learning rate. So, we can simply call SGDRegressor constructor with learning rate set to constant and this constant learning rate is specified in η_0 . So, we are specified 0.01 as a learning rate and that will be kept constant throughout the throughout the training process of SGDRegressor.

(Refer Slide Time: 13:52)



How to set adaptive learning rate?

```
1 from sklearn.linear_model import SGDRegressor
2 linear_regressor = SGDRegressor(learning_rate='adaptive',
3                                eta0=1e-2)
```

- The learning rate is kept to initial value as long as the training loss decreases.
- When the stopping criterion is reached, the learning rate is divided by 5, and the training loop continues.
- The algorithm stops when the learning rate goes below 10^{-6} .

So, let us try to understand how to set an adaptive learning rate. So, we have to specify the learning rate style to adaptive in the constructor of SGDRegressor and we have to specify what is the initial learning rate through eta0 parameter. So, in the adaptive learning rate, a learning rate is kept to the initial value as long as the training loss decreases. And when the stopping criteria are reached, that is training loss does not decrease further the learning rate is divided by 5, and the training loop continues.

So, we keep the learning rate constant till we reach the stopping criteria and at stopping criteria we change the learning rate, and the way we change it is by dividing it by 5 to the existing learning rate. And then we continue with the training loop again. Now, we need a stopping criterion and we stop the algorithm when the learning rate goes below 10^{-6} .

(Refer Slide Time: 15:00)

How to use set #epochs in SGDRegressor?



Set `max_iter` to desired #epochs. The default value is 1000.

```
1 from sklearn.linear_model import SGDRegressor
2 linear_regressor = SGDRegressor(max_iter=100)
```

Remember one epoch is one full pass over the training data.

Practical tip

SGD converges after observing approximately 10^6 training samples. Thus, a reasonable first guess for the number of iterations for n sampled training set is

```
max_iter = np.ceil(106/n)
```

Now, that we have set up the learning rate, we also need to understand how to set the number of epochs in an SGDRegressor. In this case, there is a parameter called `max_iter` that we can set to the desired number of epochs. And the default value for `max_iter` is 1000. So, we can set `max_iter` parameter in the constructor of SGDRegressor. And here, I would like to remind you that one epoch is one full pass over the training set.

So, it has been found to explain that SGD converges after observing approximately 10^6 training samples. Thus, a reasonable first guess for the number of iterations for n sample training set is we can actually divide 10^6 by n , which is the number of samples in the training set, and we take the ceiling value of this particular division and set `max_iter` to that particular value. If you do that, then possibly our SGDRegressor will converge.

(Refer Slide Time: 16:18)



How to use set **stopping criteria** in SGDRegressor?

Option #1 `tol, n_iter_no_change, max_iter.`

```
1 from sklearn.linear_model import SGDRegressor
2 linear_regressor = SGDRegressor(loss='squared_error',
3                                 max_iter=500,
4                                 tol=1e-3,
5                                 n_iter_no_change=5)
```

The SGDRegressor **stops**

- when the training loss does not improve (`loss > best_loss - tol`) for `n_iter_no_change` consecutive epochs
- else after a maximum number of iteration `max_iter`.

We can also stop SGDRegressor by specifying stopping criteria. Let us study how to use different stopping criteria in SGDRegressor. So, option one is to set stopping criteria based on three parameters. One is tolerance second is `n_iter_no_change` and `max_iter`.

So here, the SGDRegressor stops when the training loss does not improve further. For `n_iter_no_change` consecutive epoch. So, if training loss does not improve for a specified number of consecutive epochs, the training stops. And if this condition is not met, we stop the training after reaching the maximum number of iteration that is specified in `max_iter`. So, through these three parameters, we can specify the stopping criteria for HCD regressor as one of the options.

(Refer Slide Time: 17:30)



How to use set **stopping criteria** in SGDRegressor?

Option #2 **early_stopping**, **validation_fraction**

```
1 from sklearn.linear_model import SGDRegressor
2 linear_regressor = SGDRegressor(loss='squared_error',
3                                 early_stopping=True
4                                 max_iter=500,
5                                 tol=1e-3,
6                                 validation_fraction=0.2,
7                                 n_iter_no_change=5)
```

Set aside **validation_fraction** percentage records from training set as validation set. Use **score** method to obtain validation score.

The SGDRegressor **stops** when

- **validation score** does not improve by at least **tol** for **n_iter_no_change** consecutive epochs.
- else after a maximum number of iteration **max_iter**.

In the second option, we specify the early stopping criteria. For the early stopping criteria, we need to specify two more additional parameters, which is early stopping equal to true and we also have to specify the validation fraction. The validation fraction is used to set aside some data as validation data. And we use the score method to obtain the validation score.

Now, the SGDRegressor stops when the validation score does not improve by at least the tolerance value for a specified number of consecutive epochs. And that threshold is specified by the **n_iter_no_change** parameter. Else, the SGDRegressor stops after reaching the maximum number of iterations specified in the **max_iter** parameter.

(Refer Slide Time: 18:31)

How to use different **loss functions** in SGDRegressor?



Set **loss** parameter to one of the supported values

'squared_error' {studied in this course}

```
1 from sklearn.linear_model import SGDRegressor
2 linear_regressor = SGDRegressor(loss='squared_error')
```

It also supports other losses as documented in [sklearn API](#)

Let us study how to use different loss functions in SGDRegressor. So, we can set the loss parameter to one of the supported values which are squared error, which we have studied already in the course or we can use other losses such as Huber loss, and you can actually find out more about these losses in the sklearn API documentation. But the loss value the point here is loss value has to be specified in the constructor of the SGDRegressor object. So, here earlier, we have seen that there is a squared loss and Huber loss; those are two losses that we can use here.

(Refer Slide Time: 19:14)

How to use averaged SGD?



Averaged SGD updates the weight vector to **average of weights** from previous updates.

Option #1: Averaging across all updates **average=True**

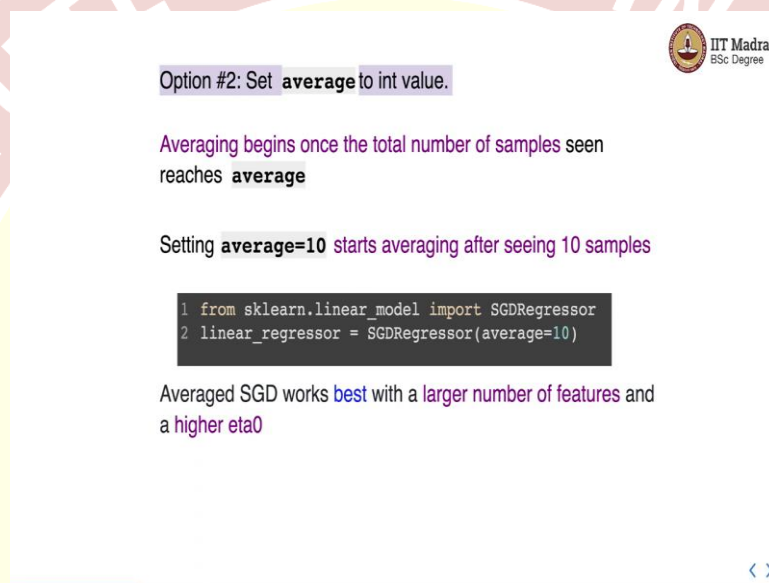
```
1 from sklearn.linear_model import SGDRegressor
2 linear_regressor = SGDRegressor(average=True)
```

So, what happens is SGD, SGD makes provides weight update after looking at every sample. So, when SGD does that by looking at one sample when it provides the update by looking at

one sample, the SGD updates might be a bit unstable. So, we can make better, we can give a better weight by updating the weight vectors based on average of some previous updates and averaged SGD specifically does that.

So, we can we can do averaging across all updates as one of the option by setting average equal to true. So, we have to set this average parameter in SGDRegressor constructor and set it to true. When we do that, we get weight, next weight updated to average of weights from the previous updates.

(Refer Slide Time: 20:18)



Option #2: Set **average** to int value.

Averaging begins once the total number of samples seen reaches **average**

Setting **average=10** starts averaging after seeing 10 samples

```
1 from sklearn.linear_model import SGDRegressor
2 linear_regressor = SGDRegressor(average=10)
```

Averaged SGD works **best** with a **larger number of features** and a **higher eta0**

In the second option, we set average to some integer value. When we set average to integer value, averaging begins once the total number of samples seen, which is the average. So, what we do is we ignore first few samples and we start averaging only after we exceed the specified number of samples in the average.

So, for example, setting average equal to 10 would start averaging after seeing 10 samples. So, this is how you can specify the integer value in the average argument. So, average SGD works best with a large number of features and higher value of eta0, and eta0 to remind you is the initial learning rate that we specify in the SGDRegressor.

(Refer Slide Time: 21:11)

How do we initialize SGD with weight vector of the previous run?



Set `warm_start = TRUE`

while instantiating object of `SGDRegressor`

```
1 from sklearn.linear_model import SGDRegressor
2 linear_regressor = SGDRegressor(warm_start=True)
```

By default `warm_start = False`

Sometimes what we do is we do multiple runs of `SGDRegressor`. And we want to actually use the value or the value of the weight vector at the end of the previous SGD run as the initialization for the next SGD run. We can do that by setting the `warm_start` parameter to true. So, hereby instantiating, the `SGDRegressor` object, we set `warm_start` to true. So, by default is values false, which means every time we run `SGDRegressor`, it is initialized to a new weight vector. And whatever work we have done in the previous run of day `SGDRegressor` is ignored.

(Refer Slide Time: 22:03)

How to monitor SGD loss iteration after iteration?



Make use of `warm_start = TRUE`

```
1 sgd_reg = SGDRegressor(max_iter=1, tol=np.infty, warm_start=True,
2                        penalty=None, learning_rate='constant', eta0=0.0005)
3
4 for epoch in range(1000):
5     sgd_reg.fit(X_train, y_train) # continues where it left off
6     y_val_predict = sgd_reg.predict(X_val)
7     val_error = mean_squared_error(y_val, y_val_predict)
```

So, let us make use of `form_start` parameter to monitor SGD loss iteration after iteration. So here, what we do is we set `max_iter` equal to 1 so we are going to make one iteration of

SGDRegressor with warm_start equal to true. We use a constant learning rate with the learning respects with learning rate specified in it as 0.

Now, what we do is we run we basically run this loop for 4000 times and in every iteration, we call SGDRegressor fit object on training set. Now, what will happen is, when we are calling the fit, we are actually using the value of the weight vector from the previous iteration, because we have set warm_start equal to true. Then we get the prediction and calculate the mean squared error for that iteration. So, this is how we can keep track of add loss iteration after iteration.

(Refer Slide Time: 23:21)

Model inspection

How to access the weights of trained **Linear Regression** model?

$$\hat{y} = w_0 + w_1x_1 + w_2x_2 + \dots + w_mx_m = \mathbf{w}^T \mathbf{x}$$

The **weights** w_1, w_2, \dots, w_m are stored in **coef_** class variable.

```
1 linear_regressor.coef_
```

The **intercept** w_0 is stored in **intercept_** class variable.

```
1 linear_regressor.intercept_
```

Note: These code snippets works for both **LinearRegression** and **SGDRegressor**, and for that matter to **all regression estimators** that we will study in this module. Why?

All of them are estimators.

Now, that we have seen how to train linear regression models using sklearn API's, let us move on to the next topic, which is how to inspect the train model. The specific question that we are asking here is how to access the weights of trained linear regression model. Remember that the

linear regression model obtains the label as a linear combination of features, specifically, y is equal to $w_0 + w_1 x_1 + w_2 x_2$ all the way up to $w_m x_m$, where m is total number of features that are used to describe a given training example.

And we can we can compactly represent is as $W^T X$ where X is the feature vector for this particular example. So, here as part of the training process, we learn the weight vector that is, these weights which is w_1, w_2, \dots all the way up to w_m . So, the weights w_1 to w_m are stored in `coef_class` variable of linear regression object. So, we can access that by simply calling `linear_regressor.coef_` this is a member variable, where w_1 to w_m weights are stored.

The intercept weight which is w_0 is stored `intercept_class` variable. So, we get access that by calling the `intercept_plus` variable on the linear regressor object. Note that this code snippet work for both linear regression as well as `SGDRegressor`. And for that matter, this works also for all regression estimators.

Why does that work? Simply because all of them are estimators and we have seen that sklearn API is kind of designed in such a way that all estimators share some kind of common methods and member variables so that it makes it easy for us to remember those methods and member variables and use them use them in our code without really worrying about what kind of estimator object is being initialized.

(Refer Slide Time: 25:51)



How to make predictions on new data in Linear Regression model?



Step 1: Arrange data for prediction in a feature matrix of shape (#samples, #features) or in sparse matrix format.

Step 2: Call `predict` method on linear regression object with `feature matrix` as an argument.

```
1 # Predict labels for feature matrix X_test
2 linear_regressor.predict(X_test)
```

Same code works for all regression estimators.



So, let us understand how to make inferences in sklearn. So, the specific question that we are asking is how to make predictions on new data in the linear regression model. So, there are two steps, we first arrange data for prediction in a feature matrix of shape number of samples by the number of features, or in a sparse matrix format.

In step two, we call the `predict` method on linear regression object with feature matrix as an argument. For example, here we are linear regressor object, and we call `predict` method on it with `x_test`, which is a feature matrix for the test set as an argument, and this will return the predicted label for the test set. And again, the same code works for all regression estimators.

So, in this video, we studied how to train linear regression model with normal equation and ad optimization. We use linear regression class for solving linear regression problems with the normal equation method. And we use `SGDRegressor` class for solving linear regression problems with the iterative optimization method. We studied `SGDRegressor` class in detail and looked at various parameters that it provides for us to tune the optimization process.

Later, we looked at how to inspect the model by accessing the weight vector through `coefficient` and `intercept` member variable of the linear regression object. Finally, we looked at how to make predictions on the new data. So, this gives you a set of tools for solving linear regression problems. In the next video, we will study how to evaluate a linear regression model trained through one of these estimators. Until then, goodbye and namaste.