INDIAN INSTITUTE OF TECHNOLOGY MADRAS • सिद्धिर्भवति कर्मजा •

IIT Madras

ONLINE DEGREE

**Machine Learning Techniques**
**Professor Doctor. Ashish Tendulkar**
**Indian Institute of Technology Madras**
**Linear Regression on California housing Dataset**

(Refer Slide Time: 0:10)

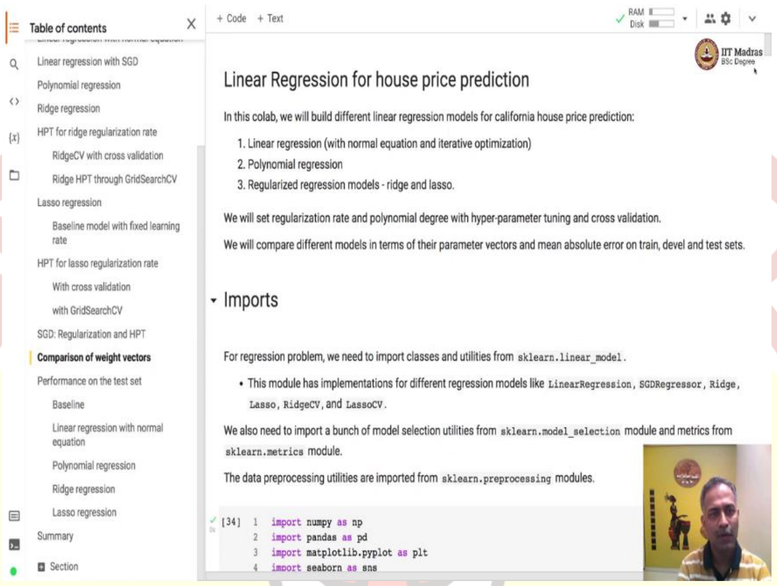


Table of contents

Linear Regression for house price prediction

In this colab, we will build different linear regression models for california house price prediction:

1. Linear regression (with normal equation and iterative optimization)
2. Polynomial regression
3. Regularized regression models - ridge and lasso.

We will set regularization rate and polynomial degree with hyper-parameter tuning and cross validation.

We will compare different models in terms of their parameter vectors and mean absolute error on train, devel and test sets.

Imports

For regression problem, we need to import classes and utilities from `sklearn.linear_model`.

- This module has implementations for different regression models like `LinearRegression`, `SGDRegressor`, `Ridge`, `Lasso`, `RidgeCV`, and `LassoCV`.

We also need to import a bunch of model selection utilities from `sklearn.model_selection` module and metrics from `sklearn.metrics` module.

The data preprocessing utilities are imported from `sklearn.preprocessing` modules.

```
[34]
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Namaste welcome to the next video of machine learning practice course. In this video, we will implement linear regression models for housing price prediction. So far, we built different kinds of linear regression models in bits and pieces for housing price prediction. In this colab, we will combine all those models and give you a holistic picture of how different regression models perform for this task.

Specifically, we will build linear regression model with normal equation and equity optimization. Then we will go on implementing the polynomial regression, followed by regularized regression models using ridge and lasso regression. We will set regularization rate and polynomial degree with hyper parameter tuning and cross validation. Finally, we will compare different models in terms of their parameter vectors and mean absolute errors on train, development set and test sets.

(Refer Slide Time: 1:18)

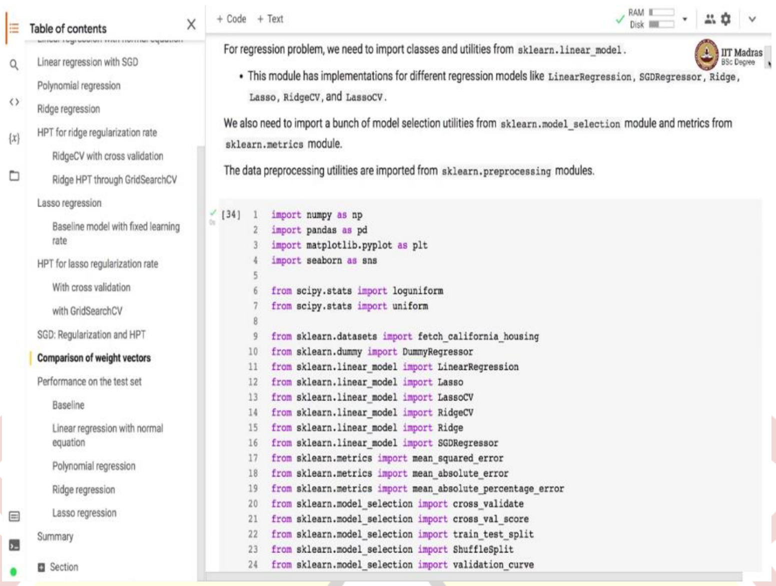


Table of contents
Linear regression with SGD
Polynomial regression
Ridge regression
HPT for ridge regularization rate
RidgeCV with cross validation
Ridge HPT through GridSearchCV
Lasso regression
Baseline model with fixed learning rate
HPT for lasso regularization rate
With cross validation
with GridSearchCV
SGD: Regularization and HPT
Comparison of weight vectors
Performance on the test set
Baseline
Linear regression with normal equation
Polynomial regression
Ridge regression
Lasso regression
Summary
Section
+ Code
+ Text
RAM
Disk
IIT Madras
BSc Degree

For regression problem, we need to import classes and utilities from `sklearn.linear_model`.

- This module has implementations for different regression models like `LinearRegression`, `SGDRegressor`, `Ridge`, `Lasso`, `RidgeCV`, and `LassoCV`.

We also need to import a bunch of model selection utilities from `sklearn.model_selection` module and metrics from `sklearn.metrics` module.

The data preprocessing utilities are imported from `sklearn.preprocessing` modules.

```
[34]
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from scipy.stats import loguniform
from scipy.stats import uniform

from sklearn.datasets import fetch_california_housing
from sklearn.dummy import DummyRegressor
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Lasso
from sklearn.linear_model import LassoCV
from sklearn.linear_model import RidgeCV
from sklearn.linear_model import Ridge
from sklearn.linear_model import SGDRegressor
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_absolute_percentage_error
from sklearn.model_selection import cross_validate
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import validation_curve
```

We begin by importing the classes that we need and the utilities from sklearn, different sklearn models. For example, we are going to use different regression models hence, we have imported multiple regression classes from sklearn.linear_module. For example, we have imported linear regression, lasso, lasso CV, ridge CV, ridge, and SGD regressor. For creating a dummy, for creating a baseline, we have also imported a dummy regressor. And in one of the last colabs, we have seen how to build baseline models for linear regression task.

(Refer Slide Time: 1:59)

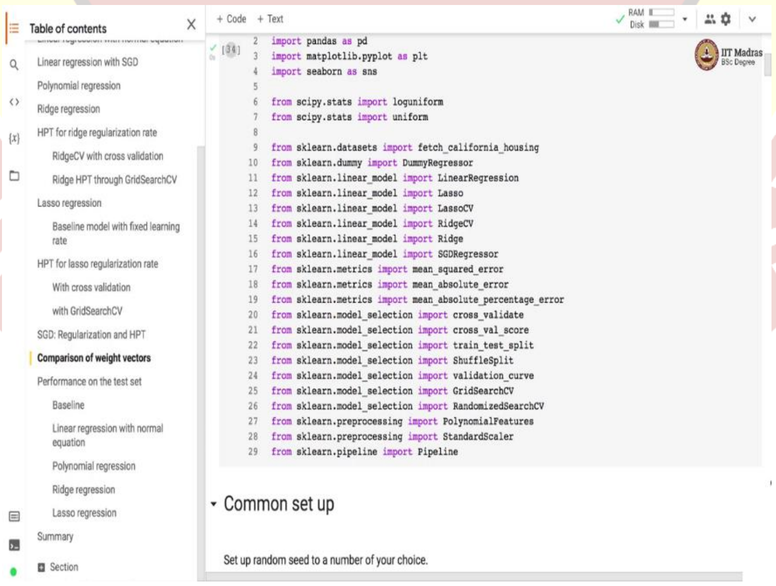


Table of contents
Linear regression with SGD
Polynomial regression
Ridge regression
HPT for ridge regularization rate
RidgeCV with cross validation
Ridge HPT through GridSearchCV
Lasso regression
Baseline model with fixed learning rate
HPT for lasso regularization rate
With cross validation
with GridSearchCV
SGD: Regularization and HPT
Comparison of weight vectors
Performance on the test set
Baseline
Linear regression with normal equation
Polynomial regression
Ridge regression
Lasso regression
Summary
Section
+ Code
+ Text
RAM
Disk
IIT Madras
BSc Degree

```
[34]
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from scipy.stats import loguniform
from scipy.stats import uniform

from sklearn.datasets import fetch_california_housing
from sklearn.dummy import DummyRegressor
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Lasso
from sklearn.linear_model import LassoCV
from sklearn.linear_model import RidgeCV
from sklearn.linear_model import Ridge
from sklearn.linear_model import SGDRegressor
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_absolute_percentage_error
from sklearn.model_selection import cross_validate
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import validation_curve
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
```

Common set up

Set up random seed to a number of your choice.

Then we have also used polynomial features from a sklearn.preprocessing modules that will be used for building polynomial regression models. Then we have various metrics and utilities from model selection that we have imported. And we will be using them to build different linear regression models and select appropriate models for the housing price prediction task. We will be doing all this processing through pipeline.

(Refer Slide Time: 2:25)

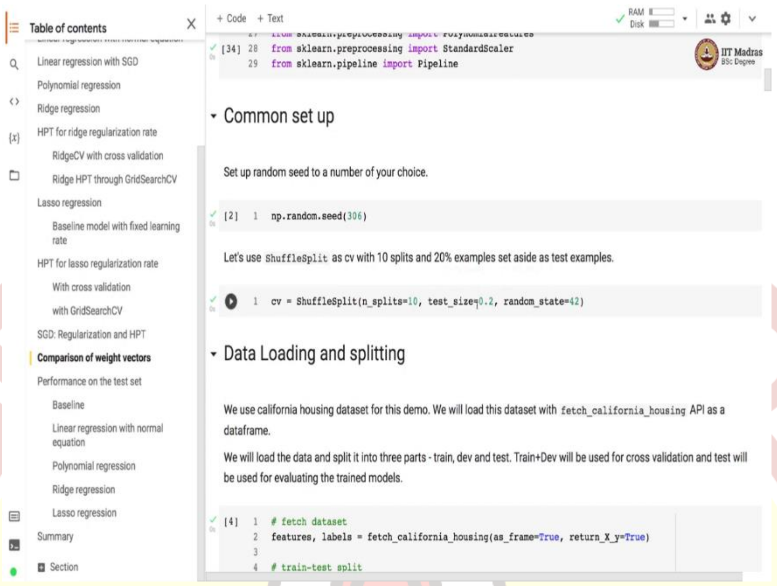


Table of contents
Linear regression with SGD
Polynomial regression
Ridge regression
HPT for ridge regularization rate
RidgeCV with cross validation
Ridge HPT through GridSearchCV
Lasso regression
Baseline model with fixed learning rate
HPT for lasso regularization rate
With cross validation
with GridSearchCV
SGD: Regularization and HPT
Comparison of weight vectors
Performance on the test set
Baseline
Linear regression with normal equation
Polynomial regression
Ridge regression
Lasso regression
Summary
Section
+ Code
+ Text
RAM
Disk
IIT Madras
BSc Degree

```
[34] from sklearn.preprocessing import StandardScaler
     from sklearn.pipeline import Pipeline
```

Common set up

Set up random seed to a number of your choice.

```
[2] np.random.seed(306)
```

Let's use `ShuffleSplit` as cv with 10 splits and 20% examples set aside as test examples.

```
cv = ShuffleSplit(n_splits=10, test_size=0.2, random_state=42)
```

Data Loading and splitting

We use california housing dataset for this demo. We will load this dataset with `fetch_california_housing` API as a dataframe.

We will load the data and split it into three parts - train, dev and test. Train+Dev will be used for cross validation and test will be used for evaluating the trained models.

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

    # train-test split
```

So we will begin by setting a random seed to a number of your choice. So here we have used 306 as a random seed. We will use shuffle script as cross validation with 10 splits. And we will set aside 20% examples as test in each fold.

(Refer Slide Time: 2:49)

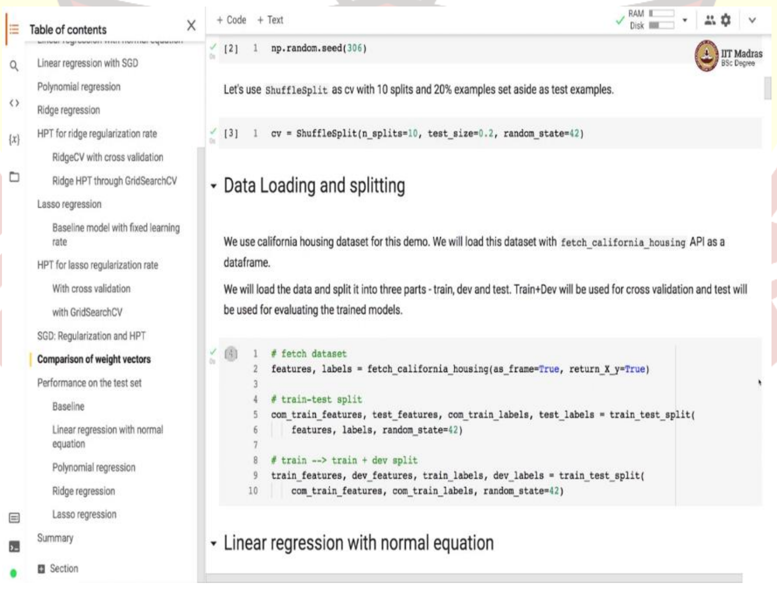


Table of contents
Linear regression with SGD
Polynomial regression
Ridge regression
HPT for ridge regularization rate
RidgeCV with cross validation
Ridge HPT through GridSearchCV
Lasso regression
Baseline model with fixed learning rate
HPT for lasso regularization rate
With cross validation
with GridSearchCV
SGD: Regularization and HPT
Comparison of weight vectors
Performance on the test set
Baseline
Linear regression with normal equation
Polynomial regression
Ridge regression
Lasso regression
Summary
Section
+ Code
+ Text
RAM
Disk
IIT Madras
BSc Degree

```
[2] np.random.seed(306)
```

Let's use `ShuffleSplit` as cv with 10 splits and 20% examples set aside as test examples.

```
[3] cv = ShuffleSplit(n_splits=10, test_size=0.2, random_state=42)
```

Data Loading and splitting

We use california housing dataset for this demo. We will load this dataset with `fetch_california_housing` API as a dataframe.

We will load the data and split it into three parts - train, dev and test. Train+Dev will be used for cross validation and test will be used for evaluating the trained models.

```
# fetch dataset
features, labels = fetch_california_housing(as_frame=True, return_X_y=True)

# train-test split
com_train_features, test_features, com_train_labels, test_labels = train_test_split(
    features, labels, random_state=42)

# train --> train + dev split
train_features, dev_features, train_labels, dev_labels = train_test_split(
    com_train_features, com_train_labels, random_state=42)
```

Linear regression with normal equation

We will use California housing data set for this demo. And we will load this data set using fetch_California_housing module in sklearn. We will be loading this data set as data frame. And we request this particular API to return xy pair. So this API returns features and labels in form of data frames. It will first perform the training and split. Followed by we will, what we will do is we will take this training data and we will split it further into training and development set.

So, here we are going to hold out the test set and we will not show it to the training in any form. This test data will be used for evaluating the performance of the model.

(Refer Slide Time: 3:42)

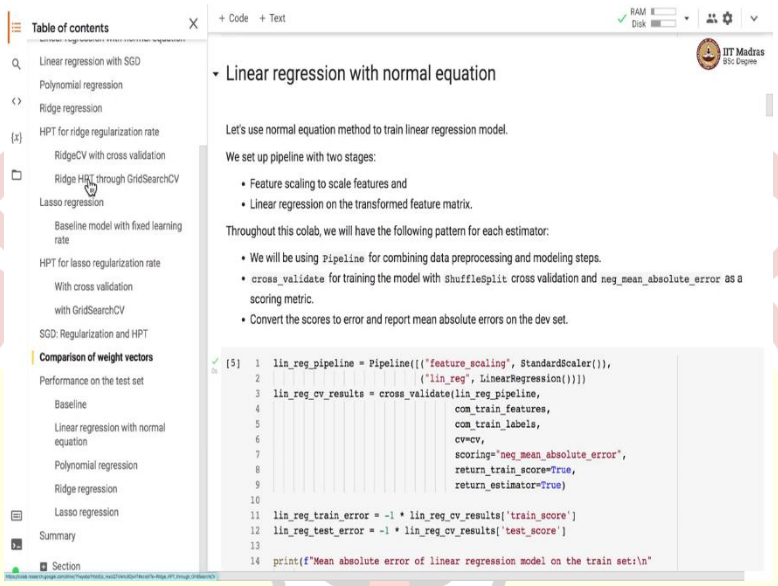


Table of contents
Linear regression with SGD
Polynomial regression
Ridge regression
HPT for ridge regularization rate
RidgeCV with cross validation
Ridge HPT through GridSearchCV
Lasso regression
Baseline model with fixed learning rate
HPT for lasso regularization rate
With cross validation
with GridSearchCV
SGD: Regularization and HPT
Comparison of weight vectors
Performance on the test set
Baseline
Linear regression with normal equation
Polynomial regression
Ridge regression
Lasso regression
Summary
Section
+ Code
+ Text
RAM
Disk
IIT Madras
BSc Degree

## Linear regression with normal equation

Let's use normal equation method to train linear regression model.

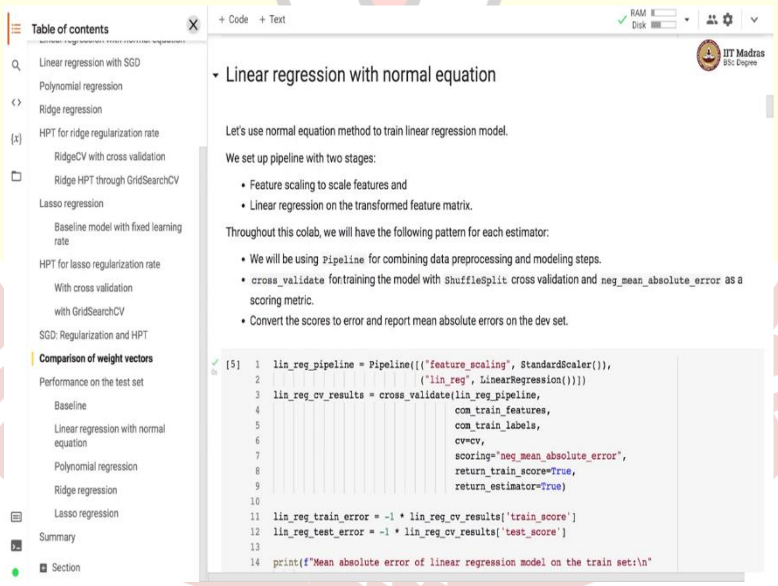
We set up pipeline with two stages:

- Feature scaling to scale features and
- Linear regression on the transformed feature matrix.

Throughout this colab, we will have the following pattern for each estimator:

- We will be using `Pipeline` for combining data preprocessing and modeling steps.
- `cross_validate` for training the model with `ShuffleSplit` cross validation and `neg_mean_absolute_error` as a scoring metric.
- Convert the scores to error and report mean absolute errors on the dev set.

```
[5]
lin_reg_pipeline = Pipeline([("feature_scaling", StandardScaler()),
                             ("lin_reg", LinearRegression())])
lin_reg_cv_results = cross_validate(lin_reg_pipeline,
                                    com_train_features,
                                    com_train_labels,
                                    cv=cv,
                                    scoring="neg_mean_absolute_error",
                                    return_train_score=True,
                                    return_estimator=True)

lin_reg_train_error = -1 * lin_reg_cv_results['train_score']
lin_reg_test_error = -1 * lin_reg_cv_results['test_score']

print(f"Mean absolute error of linear regression model on the train set:\n"
```


## Linear regression with normal equation

Let's use normal equation method to train linear regression model.

We set up pipeline with two stages:

- Feature scaling to scale features and
- Linear regression on the transformed feature matrix.

Throughout this colab, we will have the following pattern for each estimator:

- We will be using `Pipeline` for combining data preprocessing and modeling steps.
- `cross_validate` for training the model with `ShuffleSplit` cross validation and `neg_mean_absolute_error` as a scoring metric.
- Convert the scores to error and report mean absolute errors on the dev set.

```
[5]
lin_reg_pipeline = Pipeline([("feature_scaling", StandardScaler()),
                             ("lin_reg", LinearRegression())])
lin_reg_cv_results = cross_validate(lin_reg_pipeline,
                                    com_train_features,
                                    com_train_labels,
                                    cv=cv,
                                    scoring="neg_mean_absolute_error",
                                    return_train_score=True,
                                    return_estimator=True)

lin_reg_train_error = -1 * lin_reg_cv_results['train_score']
lin_reg_test_error = -1 * lin_reg_cv_results['test_score']

print(f"Mean absolute error of linear regression model on the train set:\n"
```

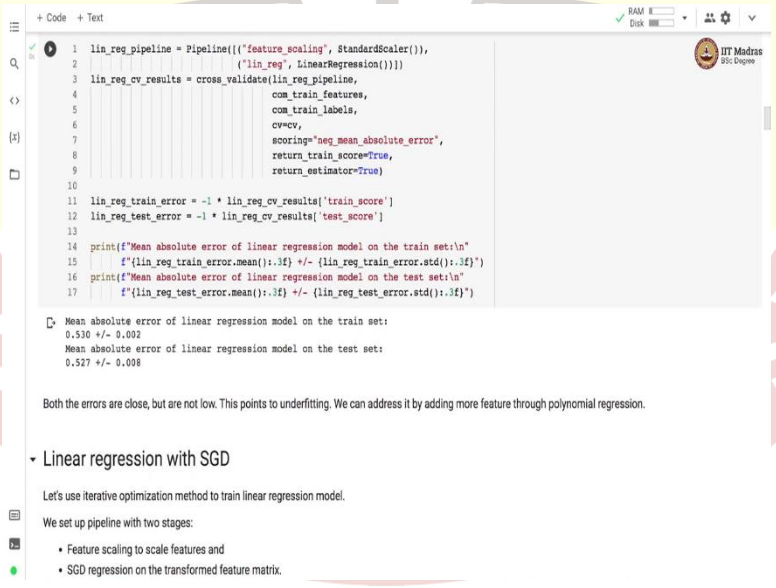
So, now what we will do is we will build a linear regression model, followed by a linear regression model with SGD then polynomial regression model, ridge regression model, and lasso regression model. Then we will also perform hyper parameter tuning for ridge and lasso. So, this hyper parameter tuning also includes hyper parameter tuning for the degree of the polynomial regression.

Now we are going to follow a specific pattern in each of these in each of these linear regression models. We will be using pipelining. And first we will be doing a feature scaling  in each and every pipeline we will first implement in feature scaling that will help us to get all the features on the same scale. Here we use standard scaler as a feature scaling mechanism. In the first pipeline, we are using linear regression class.

And this linear regression essentially implements the normal equation in order to learn the parameters of the regression task. Then we will perform the training of this pipeline through cross validation So the cross validation based training, it ensures that we have a robust training with different folds that we get in cross validation. So here we specify the linear regression pipeline as the estimator for cross validation.

We have combined training features and labels, we are able to use shuffle split CV as a cross validation mechanism, we will be using negative mean absolute error as a scoring mechanism for this cross validation. In addition to that, we are setting the optional parameters of returning the training score and estimator to true so this will give us the training score in each fold of cross validation. And we will also get the estimators that we have learned on different folds.

(Refer Slide Time: 5:56)



```
lin_reg_pipeline = Pipeline([("feature_scaling", StandardScaler()),
                             ("lin_reg", LinearRegression())])
lin_reg_cv_results = cross_validate(lin_reg_pipeline,
                                    com_train_features,
                                    com_train_labels,
                                    cv=cv,
                                    scoring="neg_mean_absolute_error",
                                    return_train_score=True,
                                    return_estimator=True)

lin_reg_train_error = -1 * lin_reg_cv_results['train_score']
lin_reg_test_error = -1 * lin_reg_cv_results['test_score']

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

```
Mean absolute error of linear regression model on the train set:
0.530 +/- 0.002
Mean absolute error of linear regression model on the test set:
0.527 +/- 0.008
```

Both the errors are close, but are not low. This points to underfitting. We can address it by adding more feature through polynomial regression.

## Linear regression with SGD

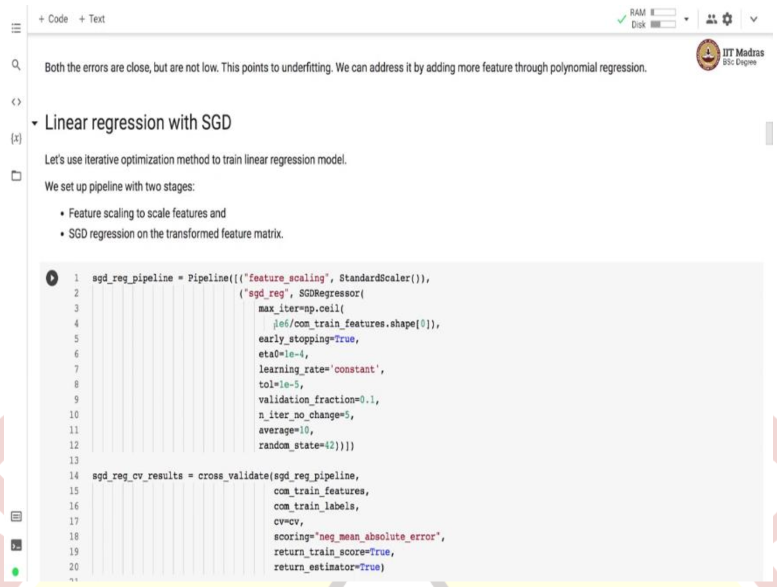
Let's use iterative optimization method to train linear regression model.

We set up pipeline with two stages:

- Feature scaling to scale features and
- SGD regression on the transformed feature matrix.

We will convert discoursed into error and we will print the errors. So, we can see that when we run this, we get the mean absolute error for linear regression model on training set, it is 0.53 and on test set it is slightly lower which is 0.527 which is very close to the training error. Here, we can see that both training and test error are close but they are not low. This points to underfitting, we can address underfitting problem by using polynomials of higher degrees.

(Refer Slide Time: 6:36)


Both the errors are close, but are not low. This points to underfitting. We can address it by adding more feature through polynomial regression.

## Linear regression with SGD

Let's use iterative optimization method to train linear regression model.

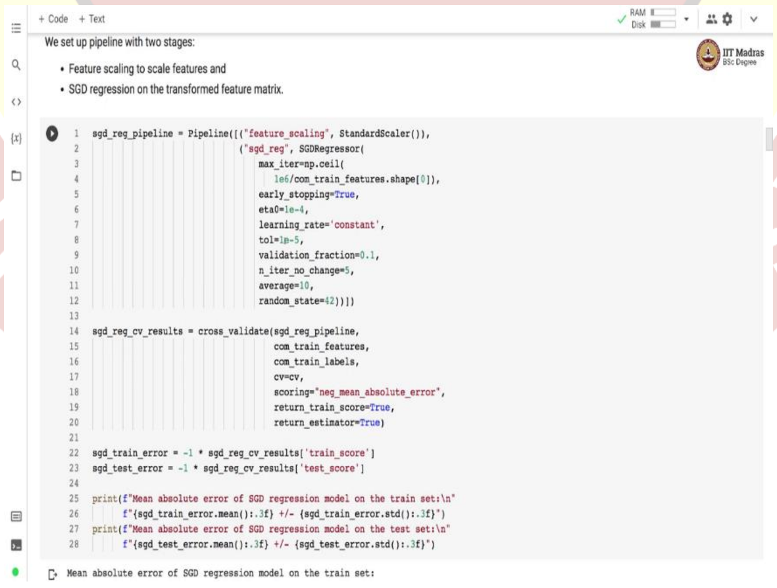
We set up pipeline with two stages:

- Feature scaling to scale features and
- SGD regression on the transformed feature matrix.

```
sgd_reg_pipeline = Pipeline([("feature_scaling", StandardScaler()),
                             ("sgd_reg", SGDRegressor(
                                 max_iter=np.ceil(
                                     1e6/com_train_features.shape[0]),
                                 early_stopping=True,
                                 eta0=1e-4,
                                 learning_rate='constant',
                                 tol=1e-5,
                                 validation_fraction=0.1,
                                 n_iter_no_change=5,
                                 average=10,
                                 random_state=42))])

sgd_reg_cv_results = cross_validate(sgd_reg_pipeline,
                                    com_train_features,
                                    com_train_labels,
                                    cv=cv,
                                    scoring="neg_mean_absolute_error",
                                    return_train_score=True,
                                    return_estimator=True)
```

Before building a polynomial regression model, we will also use a iterative optimization method to train the linear regression model. Here instead of linear regression, we use SGD regressor as an estimator. We set the max_iter based on the best practice that we saw in the last colab. So, you basically set maximum iteration such that SGD is able to see at least 10 million examples. And around 1 million examples, empirical it has been found that SGD generally converges.

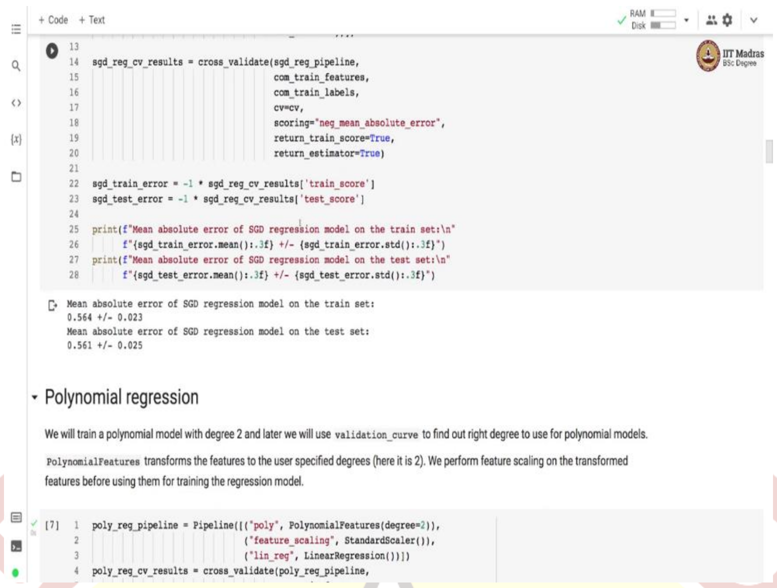
(Refer Slide Time: 7:18)


We set up pipeline with two stages:

- Feature scaling to scale features and
- SGD regression on the transformed feature matrix.

```
sgd_reg_pipeline = Pipeline([("feature_scaling", StandardScaler()),
                             ("sgd_reg", SGDRegressor(
                                 max_iter=np.ceil(
                                     1e6/com_train_features.shape[0]),
                                 early_stopping=True,
                                 eta0=1e-4,
                                 learning_rate='constant',
                                 tol=1e-5,
                                 validation_fraction=0.1,
                                 n_iter_no_change=5,
                                 average=10,
                                 random_state=42))])

sgd_reg_cv_results = cross_validate(sgd_reg_pipeline,
                                    com_train_features,
                                    com_train_labels,
                                    cv=cv,
                                    scoring="neg_mean_absolute_error",
                                    return_train_score=True,
                                    return_estimator=True)

sgd_train_error = -1 * sgd_reg_cv_results['train_score']
sgd_test_error = -1 * sgd_reg_cv_results['test_score']

print(f"Mean absolute error of SGD regression model on the train set:\n"
      f"{sgd_train_error.mean():.3f} +/- {sgd_train_error.std():.3f}")
print(f"Mean absolute error of SGD regression model on the test set:\n"
      f"{sgd_test_error.mean():.3f} +/- {sgd_test_error.std():.3f}")
```

```
Mean absolute error of SGD regression model on the train set:
```



```
sgd_reg_cv_results = cross_validate(sgd_reg_pipeline,
                                    com_train_features,
                                    com_train_labels,
                                    cv=cv,
                                    scoring="neg_mean_absolute_error",
                                    return_train_score=True,
                                    return_estimator=True)

sgd_train_error = -1 * sgd_reg_cv_results['train_score']
sgd_test_error = -1 * sgd_reg_cv_results['test_score']

print(f"Mean absolute error of SGD regression model on the train set:\n"
      f"{sgd_train_error.mean():.3f} +/- {sgd_train_error.std():.3f}")
print(f"Mean absolute error of SGD regression model on the test set:\n"
      f"{sgd_test_error.mean():.3f} +/- {sgd_test_error.std():.3f}")
```

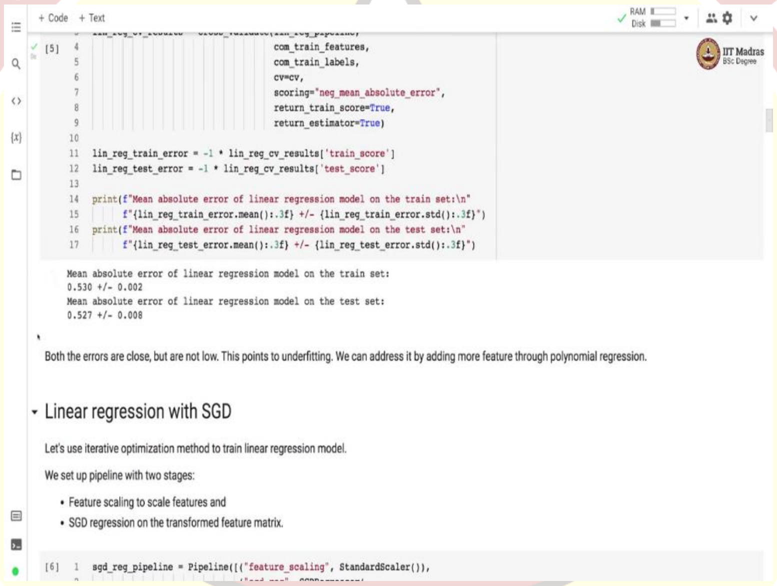
```
Mean absolute error of SGD regression model on the train set:
0.564 +/- 0.023
Mean absolute error of SGD regression model on the test set:
0.561 +/- 0.025
```

### Polynomial regression

We will train a polynomial model with degree 2 and later we will use `validation_curve` to find out right degree to use for polynomial models.

`PolynomialFeatures` transforms the features to the user specified degrees (here it is 2). We perform feature scaling on the transformed features before using them for training the regression model.

```
[7]
poly_reg_pipeline = Pipeline([("poly", PolynomialFeatures(degree=2)),
                              ("feature_scaling", StandardScaler()),
                              ("lin_reg", LinearRegression())])
poly_reg_cv_results = cross_validate(poly_reg_pipeline,
```



```
[5]
                                    com_train_features,
                                    com_train_labels,
                                    cv=cv,
                                    scoring="neg_mean_absolute_error",
                                    return_train_score=True,
                                    return_estimator=True)

lin_reg_train_error = -1 * lin_reg_cv_results['train_score']
lin_reg_test_error = -1 * lin_reg_cv_results['test_score']

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

```
Mean absolute error of linear regression model on the train set:
0.530 +/- 0.002
Mean absolute error of linear regression model on the test set:
0.527 +/- 0.008
```

Both the errors are close, but are not low. This points to underfitting. We can address it by adding more feature through polynomial regression.

### Linear regression with SGD

Let's use iterative optimization method to train linear regression model.

We set up pipeline with two stages:

- Feature scaling to scale features and
- SGD regression on the transformed feature matrix.

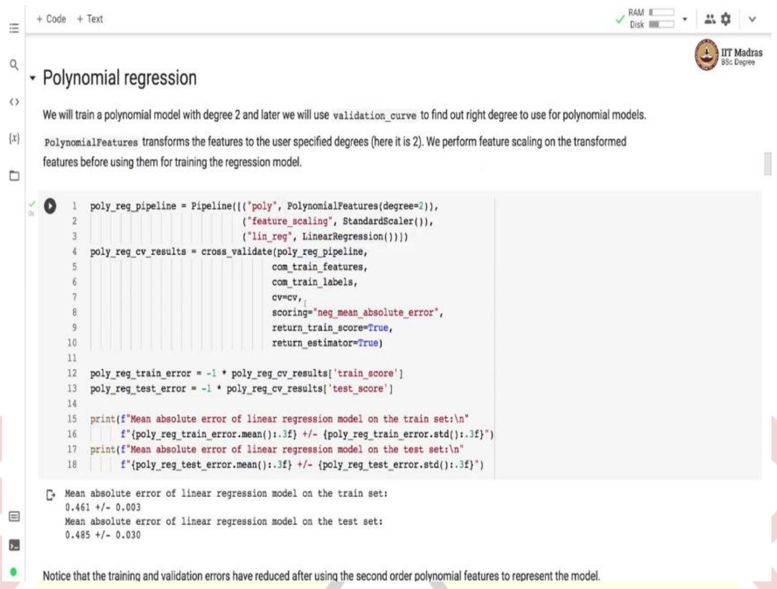
```
[6]
sgd_reg_pipeline = Pipeline([("feature_scaling", StandardScaler()),
```

You are also using early stopping and for early stopping we are setting the tolerance to $10^{-5}$ with validation fraction of 0.1 and we call the early stopping, if there is no change in the,error for 5 consecutive iterations. And a change is measured up to some tolerance value which is specified over here. Then we are going to use average SGD, so each time we set the weight updates to the average of all the weights since SGD has seen 10 samples.

We again train this model through cross validation. Now, you can see that here also we get comparable results on the training and test set. So the error which is mean absolute error is kind of comparable to what we got in the linear regression model. So, here with normal equation the error was 0.53 and with SGD it is slightly higher which is 0.56.

So, we know that SGD has a bunch of hyper parameters that needs to be tuned and we can probably get  this error down by tuning those parameters.

(Refer Slide Time: 8:42)



## Polynomial regression

We will train a polynomial model with degree 2 and later we will use `validation_curve` to find out right degree to use for polynomial models.

`PolynomialFeatures` transforms the features to the user specified degrees (here it is 2). We perform feature scaling on the transformed features before using them for training the regression model.

```python
poly_reg_pipeline = Pipeline([("poly", PolynomialFeatures(degree=2)),
                              ("feature_scaling", StandardScaler()),
                              ("lin_reg", LinearRegression())])
poly_reg_cv_results = cross_validate(poly_reg_pipeline,
                                     com_train_features,
                                     com_train_labels,
                                     cv=cv,
                                     scoring="neg_mean_absolute_error",
                                     return_train_score=True,
                                     return_estimator=True)

poly_reg_train_error = -1 * poly_reg_cv_results['train_score']
poly_reg_test_error = -1 * poly_reg_cv_results['test_score']

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

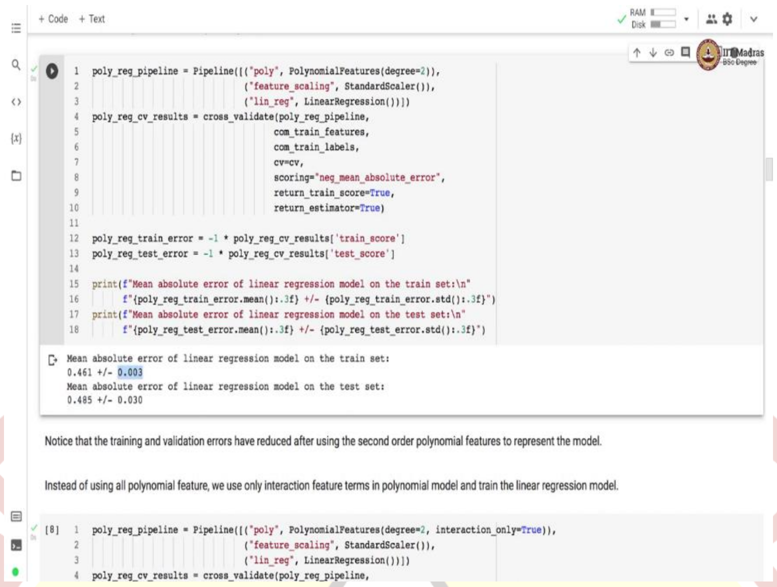
```
Mean absolute error of linear regression model on the train set:
0.461 +/- 0.003
Mean absolute error of linear regression model on the test set:
0.485 +/- 0.030
```

Notice that the training and validation errors have reduced after using the second order polynomial features to represent the model.

Next, what we do is we train a polynomial regression model with degree 2 and we will use a concept of validation curve to find out the right degree for polynomial model. So, here we have three stages now, we have first stage which is polynomial transformation. So, here we transform the input set with polynomial model, with polynomial features of degree 2. We perform standard scaling on the transform matrix through polynomial.

So, now the standard scaling is applied to the feature values that are obtained through the polynomial transformation. And after feature scaling, we basically are using the linear regression which is, which uses normal equation to train the parameters of the model. We again train this pipeline through cross validation as before and we record the training and test errors.

(Refer Slide Time: 9:42)



```
poly_reg_pipeline = Pipeline([("poly", PolynomialFeatures(degree=2)),
                              ("feature_scaling", StandardScaler()),
                              ("lin_reg", LinearRegression())])
poly_reg_cv_results = cross_validate(poly_reg_pipeline,
                                     com_train_features,
                                     com_train_labels,
                                     cv=cv,
                                     scoring="neg_mean_absolute_error",
                                     return_train_score=True,
                                     return_estimator=True)

poly_reg_train_error = -1 * poly_reg_cv_results['train_score']
poly_reg_test_error = -1 * poly_reg_cv_results['test_score']

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

```
Mean absolute error of linear regression model on the train set:
0.461 +/- 0.003
Mean absolute error of linear regression model on the test set:
0.485 +/- 0.030
```

Notice that the training and validation errors have reduced after using the second order polynomial features to represent the model.

Instead of using all polynomial feature, we use only interaction feature terms in polynomial model and train the linear regression model.

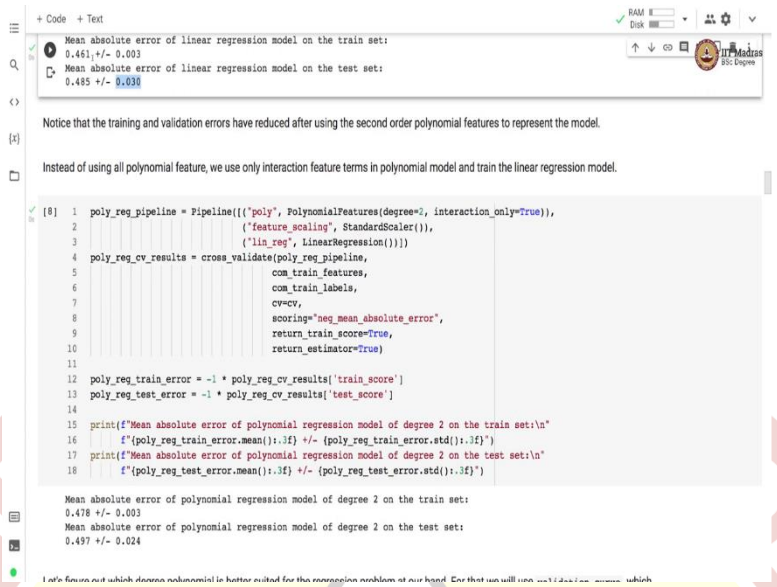
```
[8]
poly_reg_pipeline = Pipeline([("poly", PolynomialFeatures(degree=2, interaction_only=True)),
                              ("feature_scaling", StandardScaler()),
                              ("lin_reg", LinearRegression())])
poly_reg_cv_results = cross_validate(poly_reg_pipeline,
```

So here you can see that as soon as we use a polynomial model of degree 2 the error has gone down to 0.46, so the training error is 0.46 and the test error is 0.48. And since we are performing cross validation we also get the standard deviation on the error. So, you can see that standard deviation is also pretty low. So, what happens is that because we are doing cross validation, we are going to train this polynomial regression pipeline on multiple folds and on each fold, we get the estimates of test error and under training error.

And what we have done here is, we have essentially printed the mean value of the training error that was calculated on each fold along with the standard deviation and the same thing was done on the test set, on the test error. Remember here the test error is calculated on the development set, it is not calculated on the true test set and the true test set is actually hold back.

So, here because we are giving the combined training features which has got training + development set and cross validation, what happens is that one of these folds is set aside as test and this test error is calculated on the holdout set in the cross validation. And hence, since we are using a shuffled split, shuffled split cross validation with 10 folds, we basically get 10 errors, 10 training errors and 10 test errors. And what you see on the screen is the mean of training and test error and the standard deviation which is calculated from those 10 numbers that we get from the cross validation.

(Refer Slide Time: 11:35)



```
Mean absolute error of linear regression model on the train set:
0.461 +/- 0.003
Mean absolute error of linear regression model on the test set:
0.485 +/- 0.030
```

Notice that the training and validation errors have reduced after using the second order polynomial features to represent the model.

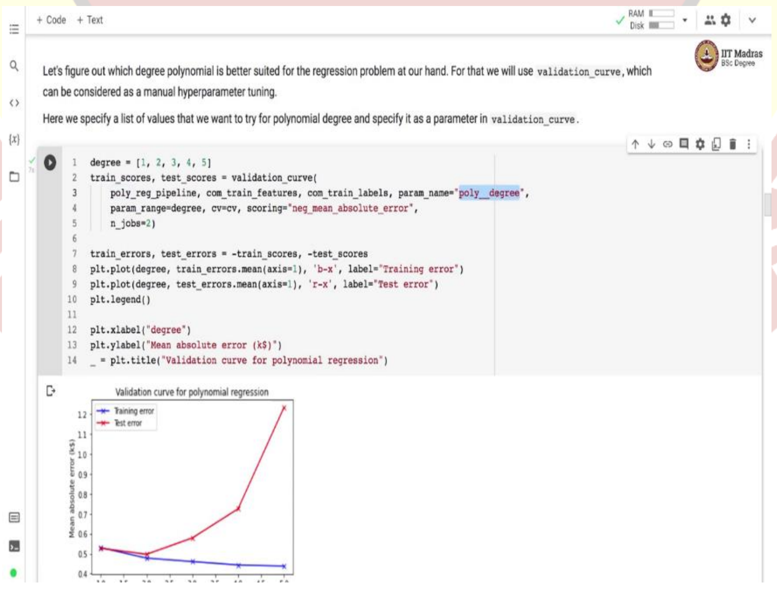
Instead of using all polynomial feature, we use only interaction feature terms in polynomial model and train the linear regression model.

```
poly_reg_pipeline = Pipeline([("poly", PolynomialFeatures(degree=2, interaction_only=True)),
                              ("feature_scaling", StandardScaler()),
                              ("lin_reg", LinearRegression())])
poly_reg_cv_results = cross_validate(poly_reg_pipeline,
                                     com_train_features,
                                     com_train_labels,
                                     cv=cv,
                                     scoring="neg_mean_absolute_error",
                                     return_train_score=True,
                                     return_estimator=True)

poly_reg_train_error = -1 * poly_reg_cv_results['train_score']
poly_reg_test_error = -1 * poly_reg_cv_results['test_score']

print(f"Mean absolute error of polynomial regression model of degree 2 on the train set:\n"
      f"{poly_reg_train_error.mean():.3f} +/- {poly_reg_train_error.std():.3f}")
print(f"Mean absolute error of polynomial regression model of degree 2 on the test set:\n"
      f"{poly_reg_test_error.mean():.3f} +/- {poly_reg_test_error.std():.3f}")
```

```
Mean absolute error of polynomial regression model of degree 2 on the train set:
0.478 +/- 0.003
Mean absolute error of polynomial regression model of degree 2 on the test set:
0.497 +/- 0.024
```

Alternatively, we can also use only interaction features in polynomial. So, if we are using only interaction features, what will happen is, it will exclude all the higher order degree features for individual features, and we will only include interaction features. So,when we do use interaction features and we perform the training through cross validation what we see is we get mean absolute error of 0.47 which is slightly higher than what we obtained when we are using all polynomial features of degree 2.

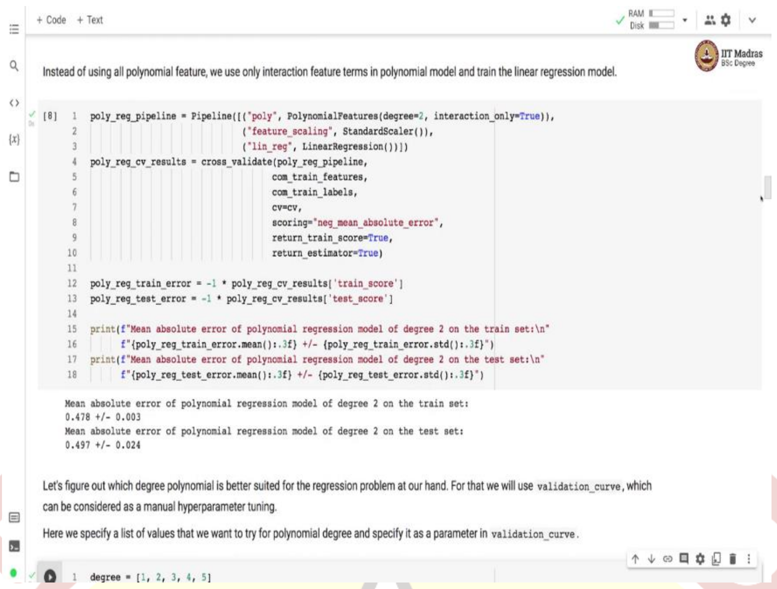
(Refer Slide Time: 12:11)



Let's figure out which degree polynomial is better suited for the regression problem at our hand. For that we will use `validation_curve`, which can be considered as a manual hyperparameter tuning.

Here we specify a list of values that we want to try for polynomial degree and specify it as a parameter in `validation_curve`.

```
degree = [1, 2, 3, 4, 5]
train_scores, test_scores = validation_curve(
    poly_reg_pipeline, com_train_features, com_train_labels, param_name="poly__degree",
    param_range=degree, cv=cv, scoring="neg_mean_absolute_error",
    n_jobs=2)

train_errors, test_errors = -train_scores, -test_scores
plt.plot(degree, train_errors.mean(axis=1), 'b-x', label="Training error")
plt.plot(degree, test_errors.mean(axis=1), 'r-x', label="Test error")
plt.legend()

plt.xlabel("degree")
plt.ylabel("Mean absolute error (k$)")
_ = plt.title("Validation curve for polynomial regression")
```


Instead of using all polynomial feature, we use only interaction feature terms in polynomial model and train the linear regression model.

```
[8]
poly_reg_pipeline = Pipeline([("poly", PolynomialFeatures(degree=2, interaction_only=True)),
                              ("feature_scaling", StandardScaler()),
                              ("lin_reg", LinearRegression())])
poly_reg_cv_results = cross_validate(poly_reg_pipeline,
                                     com_train_features,
                                     com_train_labels,
                                     cv=cv,
                                     scoring="neg_mean_absolute_error",
                                     return_train_score=True,
                                     return_estimator=True)

poly_reg_train_error = -1 * poly_reg_cv_results['train_score']
poly_reg_test_error = -1 * poly_reg_cv_results['test_score']

print(f"Mean absolute error of polynomial regression model of degree 2 on the train set:\n"
      f"{poly_reg_train_error.mean():.3f} +/- {poly_reg_train_error.std():.3f}")
print(f"Mean absolute error of polynomial regression model of degree 2 on the test set:\n"
      f"{poly_reg_test_error.mean():.3f} +/- {poly_reg_test_error.std():.3f}")
```

```
Mean absolute error of polynomial regression model of degree 2 on the train set:
0.478 +/- 0.003
Mean absolute error of polynomial regression model of degree 2 on the test set:
0.497 +/- 0.024
```

Let's figure out which degree polynomial is better suited for the regression problem at our hand. For that we will use `validation_curve`, which can be considered as a manual hyperparameter tuning.

Here we specify a list of values that we want to try for polynomial degree and specify it as a parameter in `validation_curve`.

```
degree = [1, 2, 3, 4, 5]
```

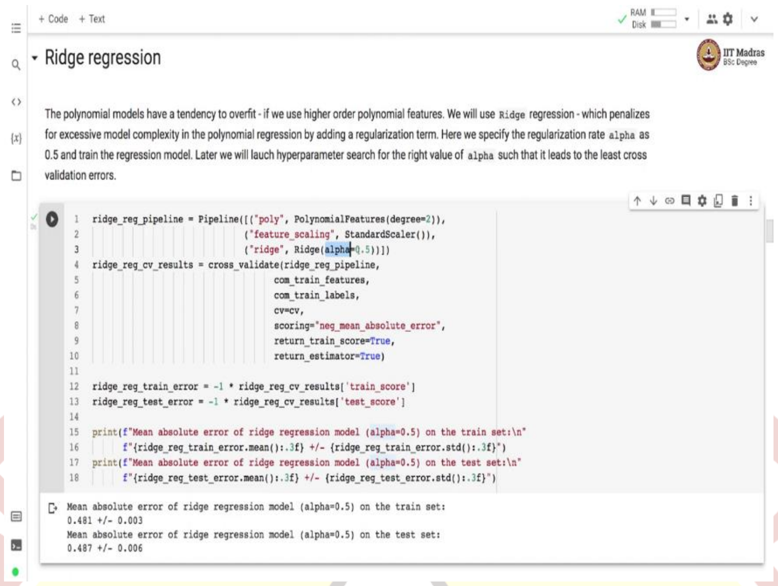
We need to figure out the degree that gives us the best performance on the regression task. And for that we will use validation curve API from sklearn. We give the polynomial regression pipeline as an estimator as a parameter to validation curve along with the feature metrics and the label metrics and the name of the parameter on which you want to perform the validation task. So, here the name of the parameter is poly__degree.

And this is the way we can specify the parameters in the pipeline. So, since we are using pipeline and the name of the stage is poly. So, that is why there is a  poly over here and this is the parameter of this particular stage which is the degree of the polynomial and the ring is specified to this degree list. We are going to use shuffle split as a cross validation and we will be using negative mean absolute error as a scoring mechanism.

So, we convert the scores that are obtained from validation curve into errors and we plot the error for each degree. So, we have degrees on x axis and we have error on y axis. So, you can see that there are 5 values for degree 1, 2, 3, 4, and 5 and we have two lines here one corresponding to the training error, the blue one corresponds to the training error and the red one corresponds to the test error.

And you can see that at degree equal to 2 we have the least value of the test error and beyond this point test error starts going up. So, this particular region shows , some kind of overfitting and probably the degree 2 is probably the appropriate value for the degree because at that value both training and test error are close and they are the lowest among the 5 different values that we have tried.

(Refer Slide Time: 14:35)



## Ridge regression

The polynomial models have a tendency to overfit - if we use higher order polynomial features. We will use `Ridge` regression - which penalizes for excessive model complexity in the polynomial regression by adding a regularization term. Here we specify the regularization rate `alpha` as 0.5 and train the regression model. Later we will lauch hyperparameter search for the right value of `alpha` such that it leads to the least cross validation errors.

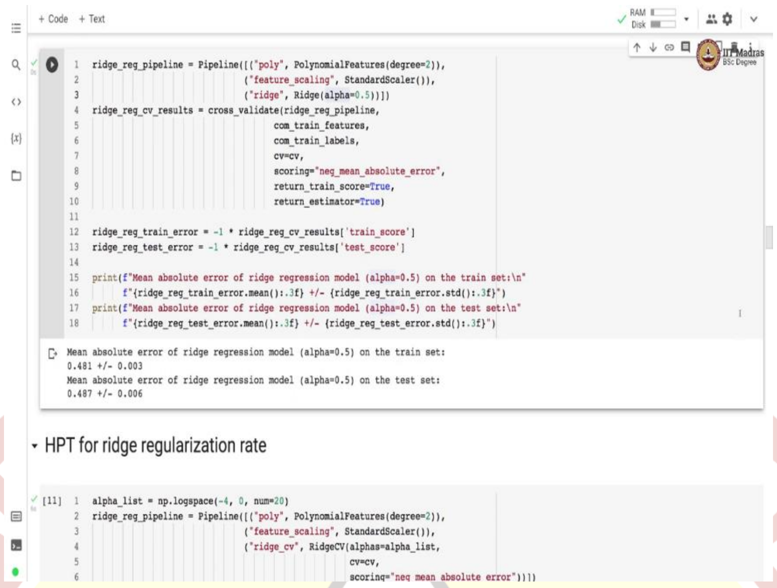
```
ridge_reg_pipeline = Pipeline([("poly", PolynomialFeatures(degree=2)),
                               ("feature_scaling", StandardScaler()),
                               ("ridge", Ridge(alpha=0.5))])
ridge_reg_cv_results = cross_validate(ridge_reg_pipeline,
                                      com_train_features,
                                      com_train_labels,
                                      cv=cv,
                                      scoring="neg_mean_absolute_error",
                                      return_train_score=True,
                                      return_estimator=True)

ridge_reg_train_error = -1 * ridge_reg_cv_results['train_score']
ridge_reg_test_error = -1 * ridge_reg_cv_results['test_score']

print(f"Mean absolute error of ridge regression model (alpha=0.5) on the train set:\n"
      f"{ridge_reg_train_error.mean():.3f} +/- {ridge_reg_train_error.std():.3f}")
print(f"Mean absolute error of ridge regression model (alpha=0.5) on the test set:\n"
      f"{ridge_reg_test_error.mean():.3f} +/- {ridge_reg_test_error.std():.3f}")
```

```
Mean absolute error of ridge regression model (alpha=0.5) on the train set:
0.481 +/- 0.003
Mean absolute error of ridge regression model (alpha=0.5) on the test set:
0.487 +/- 0.006
```

The polynomial models have a tendency to overfit. We have studied this multiple times in this course so far. So, one of the remedy for fixing the overfitting problem is to specify or is to use some kind of regularization. Here we are going to use ridge regularization. So we have built a pipeline so instead of linear regression, we are going to use ridge regression. So, ridge regression is nothing but linear regression with regularization terms added to it.

So, we are setting the regularization rate α. And again reminding you that this α is different from what we were discussing in machine learning techniques course. In machine learning techniques course we use λ to denote the regularization rate and we use α to denote the learning rate but in sklearn α is used to denote the regularization rate instead of λ. So, whenever you see α you have to draw correspondence to λ in regularization. So, here we are going to use the regularization rate of 0.5.

(Refer Slide Time: 15:46)



```python
ridge_reg_pipeline = Pipeline([("poly", PolynomialFeatures(degree=2)),
                               ("feature_scaling", StandardScaler()),
                               ("ridge", Ridge(alpha=0.5))])
ridge_reg_cv_results = cross_validate(ridge_reg_pipeline,
                                      com_train_features,
                                      com_train_labels,
                                      cv=cv,
                                      scoring="neg_mean_absolute_error",
                                      return_train_score=True,
                                      return_estimator=True)

ridge_reg_train_error = -1 * ridge_reg_cv_results['train_score']
ridge_reg_test_error = -1 * ridge_reg_cv_results['test_score']

print(f"Mean absolute error of ridge regression model (alpha=0.5) on the train set:\n"
      f"{ridge_reg_train_error.mean():.3f} +/- {ridge_reg_train_error.std():.3f}")
print(f"Mean absolute error of ridge regression model (alpha=0.5) on the test set:\n"
      f"{ridge_reg_test_error.mean():.3f} +/- {ridge_reg_test_error.std():.3f}")
```

```
Mean absolute error of ridge regression model (alpha=0.5) on the train set:
0.481 +/- 0.003
Mean absolute error of ridge regression model (alpha=0.5) on the test set:
0.487 +/- 0.006
```

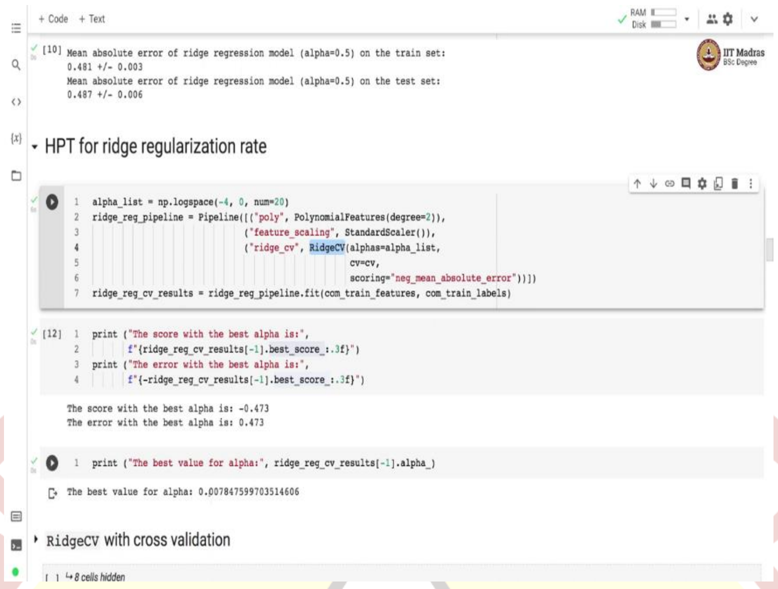
HPT for ridge regularization rate

```python
alpha_list = np.logspace(-4, 0, num=20)
ridge_reg_pipeline = Pipeline([("poly", PolynomialFeatures(degree=2)),
                               ("feature_scaling", StandardScaler()),
                               ("ridge_cv", RidgeCV(alphas=alpha_list,
                                                    cv=cv,
                                                    scoring="neg_mean_absolute_error"))])
```

We train the ridge regression pipeline in a cross validation manner. And we record the training and test scores, we convert the training and test scores to errors and we calculate the mean and standard deviation of the error across different folds in the cross validation. So, we get the training error of 0.48 compared, so this error is higher slightly higher than what we obtained in polynomial regression.

And which is going to happen because we have now added an additional regularization term. So, now this regularization term we have this λ or the regularization rate which is specified as α in case of sklearn, and we need to tune the value of alpha and find out ideal value of regularization rate that gives us the optimal performance of this particular ridge regression estimator.

(Refer Slide Time: 16:48)



```
[10] Mean absolute error of ridge regression model (alpha=0.5) on the train set:
     0.481 +/- 0.003
     Mean absolute error of ridge regression model (alpha=0.5) on the test set:
     0.487 +/- 0.006
```

HPT for ridge regularization rate

```
alpha_list = np.logspace(-4, 0, num=20)
ridge_reg_pipeline = Pipeline([("poly", PolynomialFeatures(degree=2)),
                               ("feature_scaling", StandardScaler()),
                               ("ridge_cv", RidgeCV(alphas=alpha_list,
                                                    cv=cv,
                                                    scoring="neg_mean_absolute_error"))])
ridge_reg_cv_results = ridge_reg_pipeline.fit(com_train_features, com_train_labels)
```

```
[12] print ("The score with the best alpha is:",
            f"{ridge_reg_cv_results[-1].best_score_:.3f}")
     print ("The error with the best alpha is:",
            f"{-ridge_reg_cv_results[-1].best_score_:.3f}")

The score with the best alpha is: -0.473
The error with the best alpha is: 0.473
```

```
print ("The best value for alpha:", ridge_reg_cv_results[-1].alpha_)

The best value for alpha: 0.007847599703514606
```

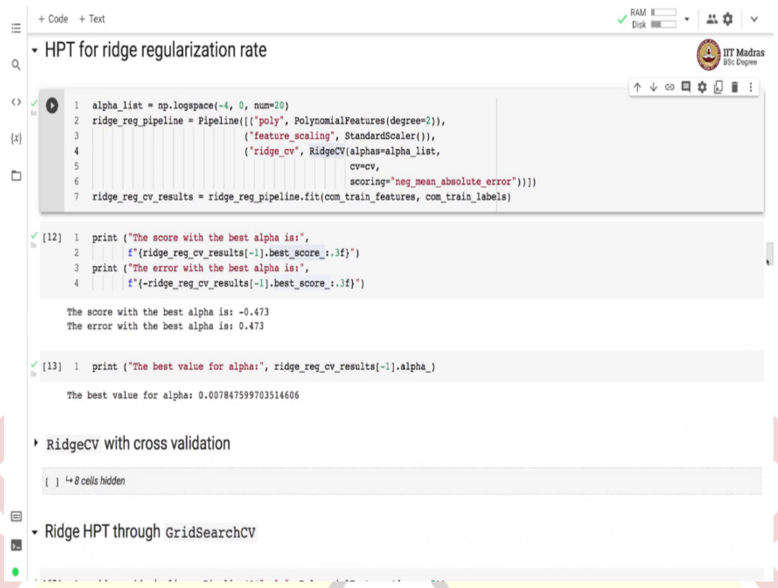
RidgeCV with cross validation

So, we can tune the value of α or we can find the value of α through multiple ways, we can either use ridge CV or other general hyper parameter tuning like grid search CV or randomize search CV. So, we will use here the ridge CV method to begin with, and in order to use ridge CV method, we need to specify the value of regularization rates or α. So here, we specify a list of regularization values to be tried.

And we train the model with the combine training features and labels. So, we can find out what is the best code that we obtain through cross validation that can be, so we can call the best_score_member variable of the ridge CV estimator in order to find out what is the best score that we obtained with the best value of alpha and that is 0.47.

And this score is now, this error is now lower than what we got from the baseline ridge regression model where we had set the value of regularization rate to 0.5. And the best value of regularization rate that we found true with CV is 0.0078.

(Refer Slide Time: 18:27)



## HPT for ridge regularization rate

```
alpha_list = np.logspace(-4, 0, num=20)
ridge_reg_pipeline = Pipeline([("poly", PolynomialFeatures(degree=2)),
                               ("feature_scaling", StandardScaler()),
                               ("ridge_cv", RidgeCV(alphas=alpha_list,
                                                    cv=cv,
                                                    scoring="neg_mean_absolute_error"))])
ridge_reg_cv_results = ridge_reg_pipeline.fit(com_train_features, com_train_labels)
```

```
[12] print ("The score with the best alpha is:",
         f"{ridge_reg_cv_results[-1].best_score_:.3f}")
     print ("The error with the best alpha is:",
         f"{-ridge_reg_cv_results[-1].best_score_:.3f}")
```

```
The score with the best alpha is: -0.473
The error with the best alpha is: 0.473
```

```
[13] print ("The best value for alpha:", ridge_reg_cv_results[-1].alpha_)
```

```
The best value for alpha: 0.007847599703514606
```

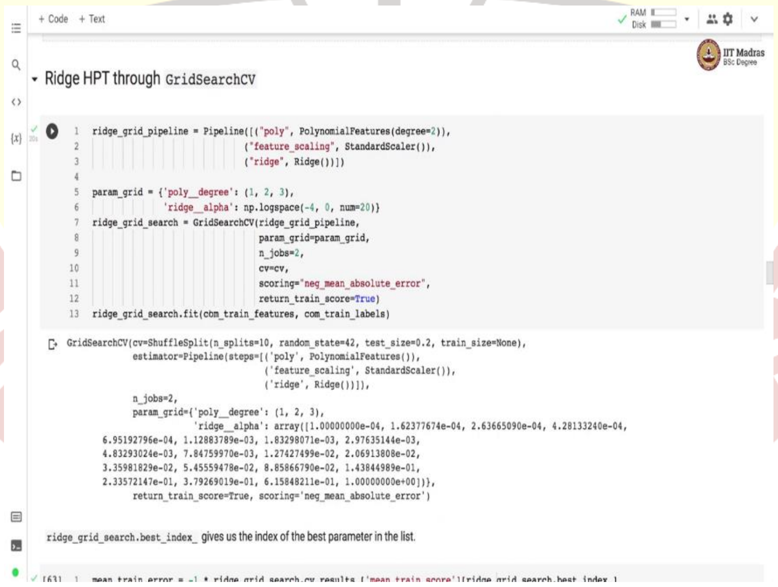
## RidgeCV with cross validation

[ ] ↳ 8 cells hidden

## Ridge HPT through GridSearchCV

We can also perform this procedure through cross validation in order to get a more robust estimate for α, and we can also understand what is the variability in α, when we train it in a cross validation manner. So, we have written code over here but we leave this as an exercise for you to experiment with this particular portion of this colab.

(Refer Slide Time: 18:57)



## Ridge HPT through GridSearchCV

```
ridge_grid_pipeline = Pipeline([("poly", PolynomialFeatures(degree=2)),
                                ("feature_scaling", StandardScaler()),
                                ("ridge", Ridge())])

param_grid = {'poly__degree': (1, 2, 3),
              'ridge__alpha': np.logspace(-4, 0, num=20)}
ridge_grid_search = GridSearchCV(ridge_grid_pipeline,
                                 param_grid=param_grid,
                                 n_jobs=2,
                                 cv=cv,
                                 scoring="neg_mean_absolute_error",
                                 return_train_score=True)
ridge_grid_search.fit(com_train_features, com_train_labels)
```

```
GridSearchCV(cv=ShuffleSplit(n_splits=10, random_state=42, test_size=0.2, train_size=None),
             estimator=Pipeline(steps=[('poly', PolynomialFeatures()),
                                       ('feature_scaling', StandardScaler()),
                                       ('ridge', Ridge())]),
             n_jobs=2,
             param_grid={'poly__degree': (1, 2, 3),
                         'ridge__alpha': array([1.00000000e-04, 1.62377674e-04, 2.63665090e-04, 4.28133240e-04,
       6.95192796e-04, 1.12883789e-03, 1.83298071e-03, 2.97635144e-03,
       4.83293024e-03, 7.84759970e-03, 1.27427499e-02, 2.06913808e-02,
       3.35981829e-02, 5.45559478e-02, 8.85866790e-02, 1.43844989e-01,
       2.33572147e-01, 3.79269019e-01, 6.15848211e-01, 1.00000000e+00])},
             return_train_score=True, scoring='neg_mean_absolute_error')
```

ridge_grid_search.best_index_ gives us the index of the best parameter in the list.

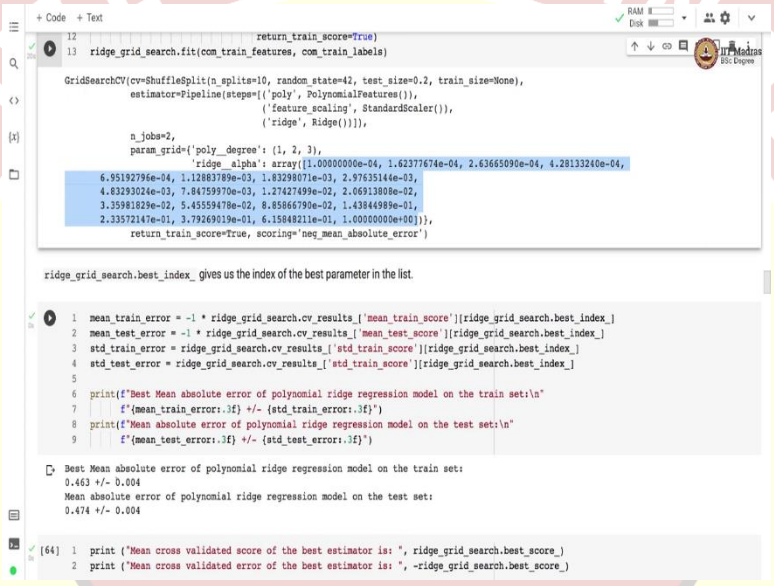
```
[63] mean_train_error = -1 * ridge_grid_search.cv_results_['mean_train_score'][ridge_grid_search.best_index_]
```

We also show how to perform the same hyper parameter tuning through more general way of hyper parameter tuning method which is grid search CV. So, here instead of grid CV we use ridge and the we actually specify the parameter grid, here there are two sets of parameters that we want to try one is we want to tune the degree of the polynomial features and we also want to tune the value of α or the regularization rate.

So we specify the parameter grid where we want to try the degrees which are 1, 2, and 3 and the value of alphas from $10^{-4}$ to 0 and that we want 20 different values in this range as values of α to be tried. Then we define grid search CV with ridge grid pipeline as an estimator, the parameter grid which is specified over here, shuffle split cross validation using negative mean absolute error as a scoring mechanism.

Then we train this particular grid search CV with the combined training features and labels. And remember this combined training features has training and development set.

(Refer Slide Time: 20:18)



```
                                  return_train_score=True)
ridge_grid_search.fit(com_train_features, com_train_labels)
```

```
GridSearchCV(cv=ShuffleSplit(n_splits=10, random_state=42, test_size=0.2, train_size=None),
             estimator=Pipeline(steps=[('poly', PolynomialFeatures()),
                                       ('feature_scaling', StandardScaler()),
                                       ('ridge', Ridge())]),
             n_jobs=2,
             param_grid={'poly__degree': (1, 2, 3),
                         'ridge__alpha': array([1.00000000e-04, 1.62377674e-04, 2.63665090e-04, 4.28133240e-04,
       6.95192796e-04, 1.12883789e-03, 1.83298071e-03, 2.97635144e-03,
       4.83293024e-03, 7.84759970e-03, 1.27427499e-02, 2.06913808e-02,
       3.35981829e-02, 5.45559478e-02, 8.85866790e-02, 1.43844989e-01,
       2.33572147e-01, 3.79269019e-01, 6.15848211e-01, 1.00000000e+00])},
             return_train_score=True, scoring='neg_mean_absolute_error')
```

`ridge_grid_search.best_index_` gives us the index of the best parameter in the list.

```
mean_train_error = -1 * ridge_grid_search.cv_results_['mean_train_score'][ridge_grid_search.best_index_]
mean_test_error = -1 * ridge_grid_search.cv_results_['mean_test_score'][ridge_grid_search.best_index_]
std_train_error = ridge_grid_search.cv_results_['std_train_score'][ridge_grid_search.best_index_]
std_test_error = ridge_grid_search.cv_results_['std_train_score'][ridge_grid_search.best_index_]

print(f"Best Mean absolute error of polynomial ridge regression model on the train set:\n"
      f"{mean_train_error:.3f} +/- {std_train_error:.3f}")
print(f"Mean absolute error of polynomial ridge regression model on the test set:\n"
      f"{mean_test_error:.3f} +/- {std_test_error:.3f}")
```

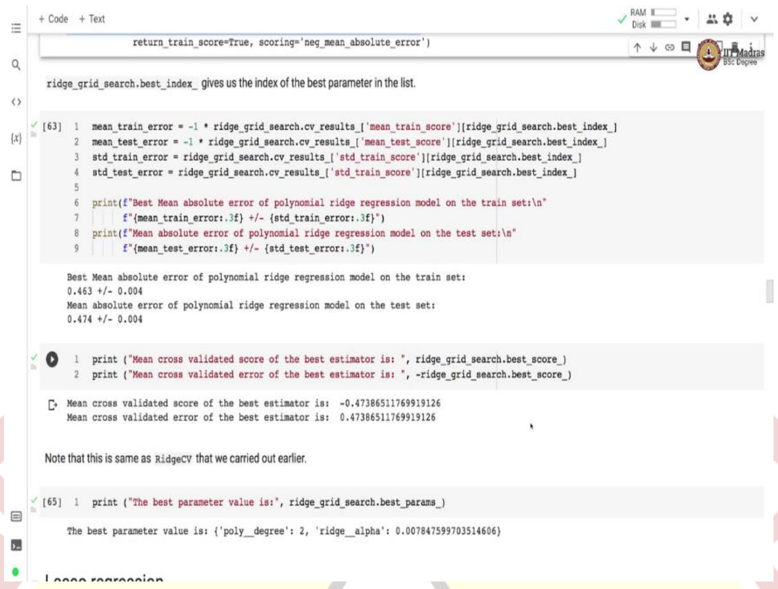
```
Best Mean absolute error of polynomial ridge regression model on the train set:
0.463 +/- 0.004
Mean absolute error of polynomial ridge regression model on the test set:
0.474 +/- 0.004
```

```
[64]
print ("Mean cross validated score of the best estimator is: ", ridge_grid_search.best_score_)
print ("Mean cross validated error of the best estimator is: ", -ridge_grid_search.best_score_)
```

So, we see here the specification of the parameter grid, the value of the degrees that we are trying is 1, 2, 3 and the values of α or the regularization rates that these are the 20 different values that will be tried out. Now, what we will do is we will find out the best parameter from the cross validation. So, we can get this best parameter through to the best_index_member variable.

So, this is the index of the best parameter in the list and  we can find out what is the mean training error and the standard deviation on training and test errors. So, we can see that for the best value of the hyper parameters we obtain the training error of 0.46 and the test error of 0.47 but you can see that these errors are lower than the base ridge regression model that we tried out.

(Refer Slide Time: 21:23)



```
                return_train_score=True, scoring='neg_mean_absolute_error')
```

`ridge_grid_search.best_index_` gives us the index of the best parameter in the list.

```
[63]
mean_train_error = -1 * ridge_grid_search.cv_results_['mean_train_score'][ridge_grid_search.best_index_]
mean_test_error = -1 * ridge_grid_search.cv_results_['mean_test_score'][ridge_grid_search.best_index_]
std_train_error = ridge_grid_search.cv_results_['std_train_score'][ridge_grid_search.best_index_]
std_test_error = ridge_grid_search.cv_results_['std_train_score'][ridge_grid_search.best_index_]

print(f"Best Mean absolute error of polynomial ridge regression model on the train set:\n"
      f"{mean_train_error:.3f} +/- {std_train_error:.3f}")
print(f"Mean absolute error of polynomial ridge regression model on the test set:\n"
      f"{mean_test_error:.3f} +/- {std_test_error:.3f}")
```

```
Best Mean absolute error of polynomial ridge regression model on the train set:
0.463 +/- 0.004
Mean absolute error of polynomial ridge regression model on the test set:
0.474 +/- 0.004
```

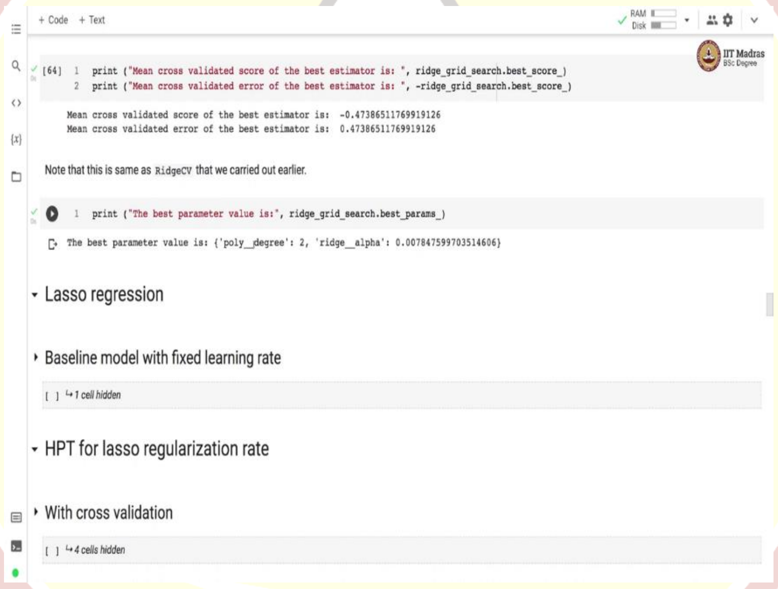
```
print ("Mean cross validated score of the best estimator is: ", ridge_grid_search.best_score_)
print ("Mean cross validated error of the best estimator is: ", -ridge_grid_search.best_score_)
```

```
Mean cross validated score of the best estimator is:  -0.47386511769919126
Mean cross validated error of the best estimator is:  0.47386511769919126
```

Note that this is same as `RidgeCV` that we carried out earlier.

```
[65]
print ("The best parameter value is:", ridge_grid_search.best_params_)
```

```
The best parameter value is: {'poly__degree': 2, 'ridge__alpha': 0.007847599703514606}
```



```
[64]
print ("Mean cross validated score of the best estimator is: ", ridge_grid_search.best_score_)
print ("Mean cross validated error of the best estimator is: ", -ridge_grid_search.best_score_)
```

```
Mean cross validated score of the best estimator is:  -0.47386511769919126
Mean cross validated error of the best estimator is:  0.47386511769919126
```

Note that this is same as `RidgeCV` that we carried out earlier.

```
print ("The best parameter value is:", ridge_grid_search.best_params_)
```

```
The best parameter value is: {'poly__degree': 2, 'ridge__alpha': 0.007847599703514606}
```

Lasso regression

Baseline model with fixed learning rate

[ ] ↳ 1 cell hidden

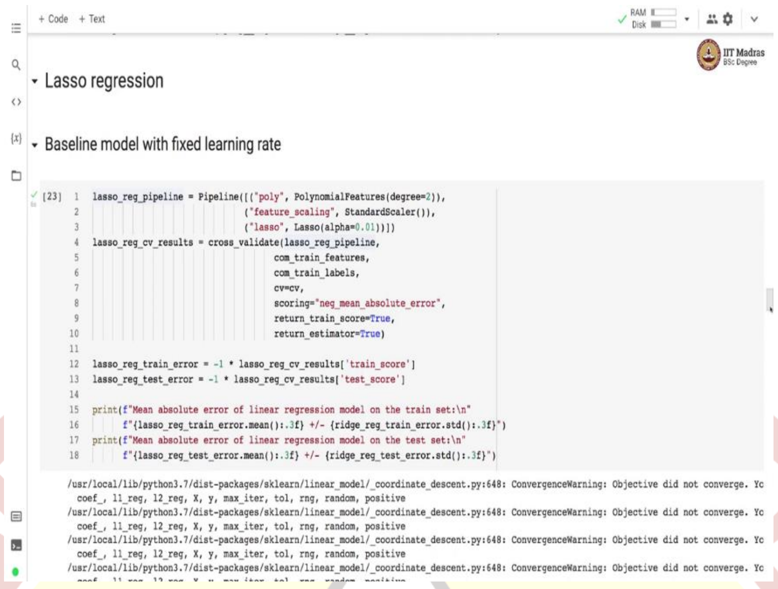
HPT for lasso regularization rate

With cross validation

[ ] ↳ 4 cells hidden

So, the best error that we obtain is 0.473. And we can also see what are the best parameters. So, we found out the polynomial degree of 2 and the value of regularization rate of 0.0078 we obtained the optimal performance on the cross validation set.

(Refer Slide Time: 21:59)

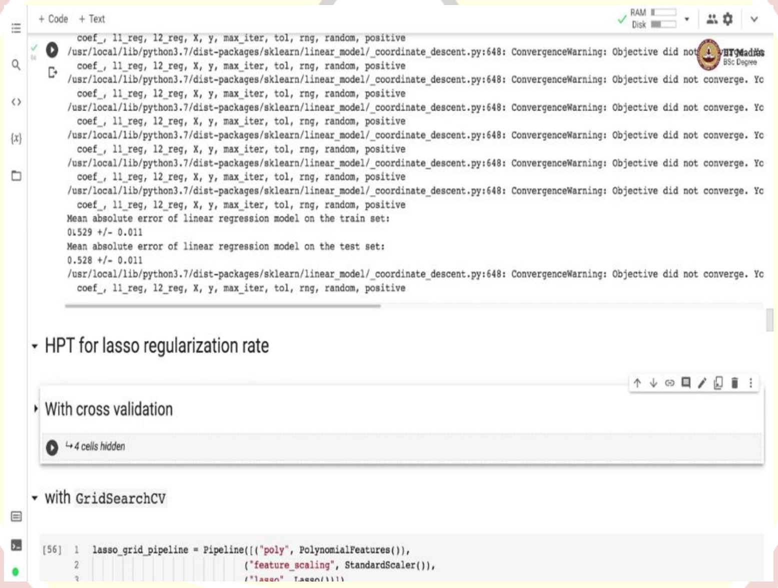


## Lasso regression

### Baseline model with fixed learning rate

```
[23]
lasso_reg_pipeline = Pipeline([("poly", PolynomialFeatures(degree=2)),
                               ("feature_scaling", StandardScaler()),
                               ("lasso", Lasso(alpha=0.01))])
lasso_reg_cv_results = cross_validate(lasso_reg_pipeline,
                                      com_train_features,
                                      com_train_labels,
                                      cv=cv,
                                      scoring="neg_mean_absolute_error",
                                      return_train_score=True,
                                      return_estimator=True)

lasso_reg_train_error = -1 * lasso_reg_cv_results['train_score']
lasso_reg_test_error = -1 * lasso_reg_cv_results['test_score']

print(f"Mean absolute error of linear regression model on the train set:\n"
      f"{lasso_reg_train_error.mean():.3f} +/- {ridge_reg_train_error.std():.3f}")
print(f"Mean absolute error of linear regression model on the test set:\n"
      f"{lasso_reg_test_error.mean():.3f} +/- {ridge_reg_test_error.std():.3f}")
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_coordinate_descent.py:648: ConvergenceWarning: Objective did not converge. Yo
  coef_, l1_reg, l2_reg, X, y, max_iter, tol, rng, random, positive
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_coordinate_descent.py:648: ConvergenceWarning: Objective did not converge. Yo
  coef_, l1_reg, l2_reg, X, y, max_iter, tol, rng, random, positive
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_coordinate_descent.py:648: ConvergenceWarning: Objective did not converge. Yo
  coef_, l1_reg, l2_reg, X, y, max_iter, tol, rng, random, positive
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_coordinate_descent.py:648: ConvergenceWarning: Objective did not converge. Yo
```



```
  coef_, l1_reg, l2_reg, X, y, max_iter, tol, rng, random, positive
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_coordinate_descent.py:648: ConvergenceWarning: Objective did not
  coef_, l1_reg, l2_reg, X, y, max_iter, tol, rng, random, positive
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_coordinate_descent.py:648: ConvergenceWarning: Objective did not converge. Yo
  coef_, l1_reg, l2_reg, X, y, max_iter, tol, rng, random, positive
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_coordinate_descent.py:648: ConvergenceWarning: Objective did not converge. Yo
  coef_, l1_reg, l2_reg, X, y, max_iter, tol, rng, random, positive
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_coordinate_descent.py:648: ConvergenceWarning: Objective did not converge. Yo
  coef_, l1_reg, l2_reg, X, y, max_iter, tol, rng, random, positive
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_coordinate_descent.py:648: ConvergenceWarning: Objective did not converge. Yo
  coef_, l1_reg, l2_reg, X, y, max_iter, tol, rng, random, positive
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_coordinate_descent.py:648: ConvergenceWarning: Objective did not converge. Yo
  coef_, l1_reg, l2_reg, X, y, max_iter, tol, rng, random, positive
Mean absolute error of linear regression model on the train set:
0.529 +/- 0.011
Mean absolute error of linear regression model on the test set:
0.528 +/- 0.011
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_coordinate_descent.py:648: ConvergenceWarning: Objective did not converge. Yo
  coef_, l1_reg, l2_reg, X, y, max_iter, tol, rng, random, positive
```

### HPT for lasso regularization rate

#### With cross validation

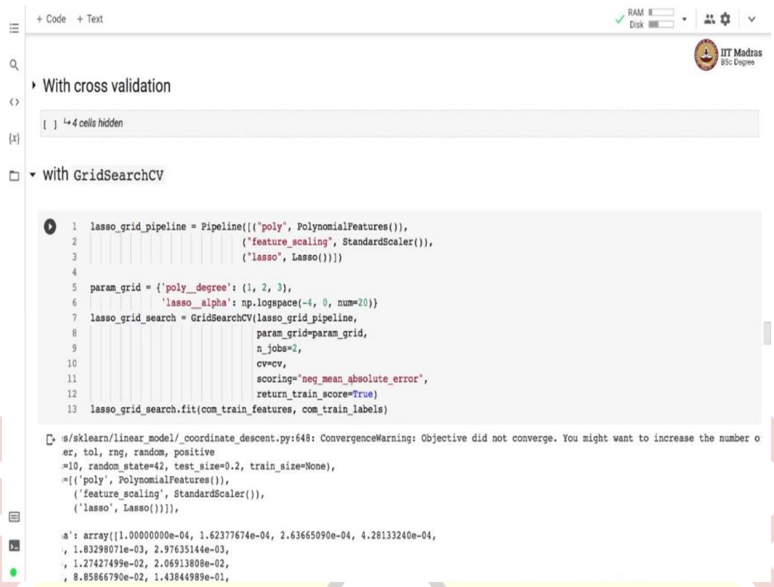
↳ 4 cells hidden

#### with GridSearchCV

```
[56]
lasso_grid_pipeline = Pipeline([("poly", PolynomialFeatures()),
                                ("feature_scaling", StandardScaler()),
                                ("lasso", Lasso())])
```

Next, we try more aggressive regularization, which is lasso regularization. Here we set up the pipeline with lasso as a regularization and we have set the regularization rate by hand to 0.01. So, we train this lasso with cross validation and we can see that the error is pretty high which is 0.52 compared to what we obtained through ridge regression and through polynomial regression. Remember, here we are also using the polynomial features.

(Refer Slide Time: 22:36)



With cross validation

[ ] ↳ 4 cells hidden

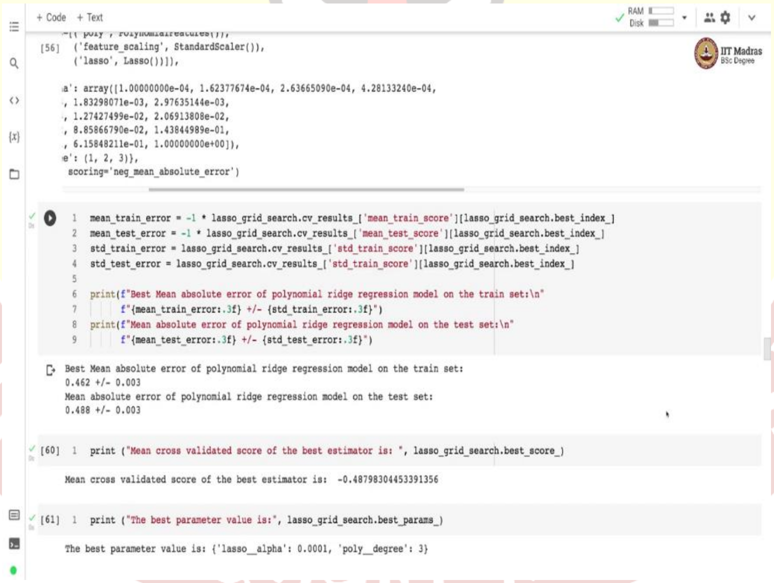
with GridSearchCV

```
lasso_grid_pipeline = Pipeline([("poly", PolynomialFeatures()),
                                ("feature_scaling", StandardScaler()),
                                ("lasso", Lasso())])

param_grid = {'poly__degree': (1, 2, 3),
              'lasso__alpha': np.logspace(-4, 0, num=20)}
lasso_grid_search = GridSearchCV(lasso_grid_pipeline,
                                 param_grid=param_grid,
                                 n_jobs=2,
                                 cv=cv,
                                 scoring="neg_mean_absolute_error",
                                 return_train_score=True)
lasso_grid_search.fit(com_train_features, com_train_labels)
```

```
s/sklearn/linear_model/_coordinate_descent.py:648: ConvergenceWarning: Objective did not converge. You might want to increase the number o
er, tol, rng, random, positive
=10, random_state=42, test_size=0.2, train_size=None),
=[('poly', PolynomialFeatures()),
  ('feature_scaling', StandardScaler()),
  ('lasso', Lasso())]),

a': array([1.00000000e-04, 1.62377674e-04, 2.63665090e-04, 4.28133240e-04,
, 1.83298071e-03, 2.97635144e-03,
, 1.27427499e-02, 2.06913808e-02,
, 8.85866790e-02, 1.43844989e-01,
```

Now, what we do is, we will perform hyper parameter tuning for the degree of the polynomial and regularization rate in lasso we use pretty much the same values as with the ridge regression in the parameter grid.

(Refer Slide Time: 23:04)



```
[56]  ('feature_scaling', StandardScaler()),
      ('lasso', Lasso())]),

a': array([1.00000000e-04, 1.62377674e-04, 2.63665090e-04, 4.28133240e-04,
, 1.83298071e-03, 2.97635144e-03,
, 1.27427499e-02, 2.06913808e-02,
, 8.85866790e-02, 1.43844989e-01,
, 6.15848211e-01, 1.00000000e+00]),
e': (1, 2, 3)},
 scoring='neg_mean_absolute_error')
```

```
mean_train_error = -1 * lasso_grid_search.cv_results_['mean_train_score'][lasso_grid_search.best_index_]
mean_test_error = -1 * lasso_grid_search.cv_results_['mean_test_score'][lasso_grid_search.best_index_]
std_train_error = lasso_grid_search.cv_results_['std_train_score'][lasso_grid_search.best_index_]
std_test_error = lasso_grid_search.cv_results_['std_train_score'][lasso_grid_search.best_index_]

print(f"Best Mean absolute error of polynomial ridge regression model on the train set:\n"
      f"{mean_train_error:.3f} +/- {std_train_error:.3f}")
print(f"Mean absolute error of polynomial ridge regression model on the test set:\n"
      f"{mean_test_error:.3f} +/- {std_test_error:.3f}")
```

```
Best Mean absolute error of polynomial ridge regression model on the train set:
0.462 +/- 0.003
Mean absolute error of polynomial ridge regression model on the test set:
0.488 +/- 0.003
```

```
[60] print ("Mean cross validated score of the best estimator is: ", lasso_grid_search.best_score_)
```

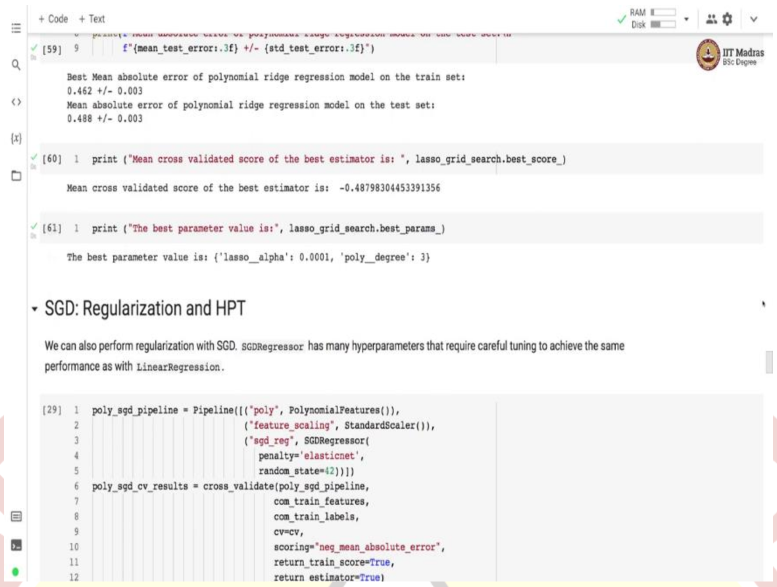
```
Mean cross validated score of the best estimator is:  -0.48798304453391356
```

```
[61] print ("The best parameter value is:", lasso_grid_search.best_params_)
```

```
The best parameter value is: {'lasso__alpha': 0.0001, 'poly__degree': 3}
```

We perform the grid search and found that the best mean absolute error on the training set is 0.46 and on test set it is 0.48. Now, you can see that these errors are much smaller than what we obtained in the base lasso estimator and these errors are comparable to what we obtained to the errors that we obtained in ridge regression case.

(Refer Slide Time: 23:35)



```
f"{mean_test_error:.3f} +/- {std_test_error:.3f}")
```

```
Best Mean absolute error of polynomial ridge regression model on the train set:
0.462 +/- 0.003
Mean absolute error of polynomial ridge regression model on the test set:
0.488 +/- 0.003
```

```
print ("Mean cross validated score of the best estimator is: ", lasso_grid_search.best_score_)
```

```
Mean cross validated score of the best estimator is:  -0.48798304453391356
```

```
print ("The best parameter value is:", lasso_grid_search.best_params_)
```

```
The best parameter value is: {'lasso__alpha': 0.0001, 'poly__degree': 3}
```

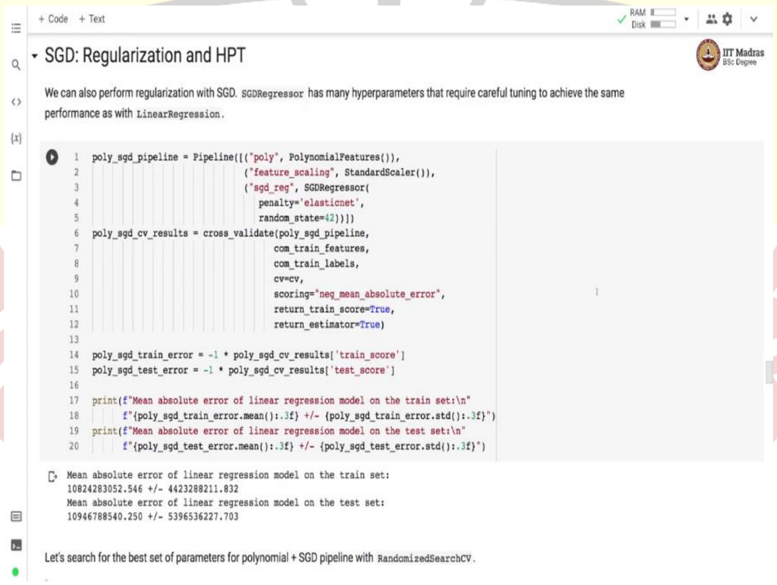
## SGD: Regularization and HPT

We can also perform regularization with SGD. `SGDRegressor` has many hyperparameters that require careful tuning to achieve the same performance as with `LinearRegression`.

```
poly_sgd_pipeline = Pipeline([("poly", PolynomialFeatures()),
                              ("feature_scaling", StandardScaler()),
                              ("sgd_reg", SGDRegressor(
                                penalty='elasticnet',
                                random_state=42))])
poly_sgd_cv_results = cross_validate(poly_sgd_pipeline,
                                     com_train_features,
                                     com_train_labels,
                                     cv=cv,
                                     scoring="neg_mean_absolute_error",
                                     return_train_score=True,
                                     return_estimator=True)
```

And you can see that the best values of the parameter, the α value was found to be 0.0001 and , the best value of degree was 3 for this combination of the parameter set degree of 3 and value of α which is regularization rate which is 0.0001 we found that the lasso regression has the optimal performance on the cross validation set.

(Refer Slide Time: 24:14)



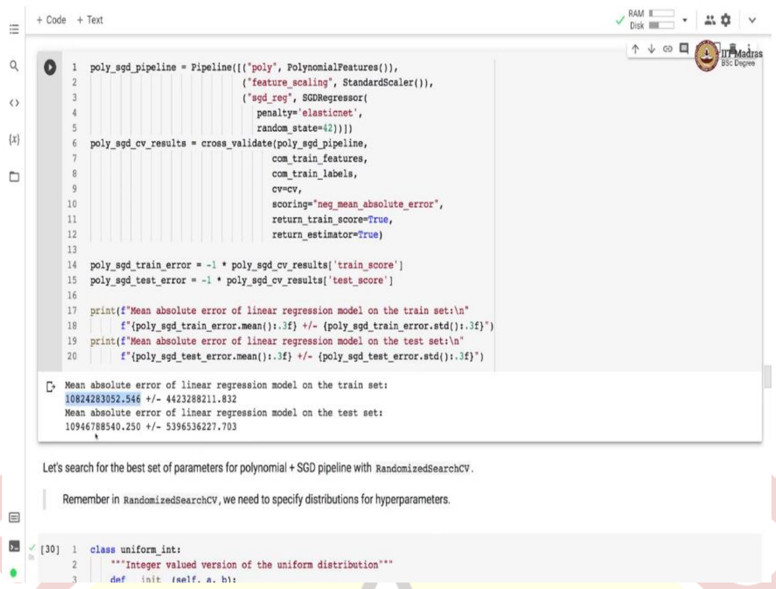
## SGD: Regularization and HPT

We can also perform regularization with SGD. `SGDRegressor` has many hyperparameters that require careful tuning to achieve the same performance as with `LinearRegression`.

```
poly_sgd_pipeline = Pipeline([("poly", PolynomialFeatures()),
                              ("feature_scaling", StandardScaler()),
                              ("sgd_reg", SGDRegressor(
                                penalty='elasticnet',
                                random_state=42))])
poly_sgd_cv_results = cross_validate(poly_sgd_pipeline,
                                     com_train_features,
                                     com_train_labels,
                                     cv=cv,
                                     scoring="neg_mean_absolute_error",
                                     return_train_score=True,
                                     return_estimator=True)

poly_sgd_train_error = -1 * poly_sgd_cv_results['train_score']
poly_sgd_test_error = -1 * poly_sgd_cv_results['test_score']

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

```
Mean absolute error of linear regression model on the train set:
10824283052.546 +/- 4423288211.832
Mean absolute error of linear regression model on the test set:
10946788540.250 +/- 5396536227.703
```

Let's search for the best set of parameters for polynomial + SGD pipeline with `RandomizedSearchCV`.



```python
poly_sgd_pipeline = Pipeline([("poly", PolynomialFeatures()),
                              ("feature_scaling", StandardScaler()),
                              ("sgd_reg", SGDRegressor(
                                penalty='elasticnet',
                                random_state=42))])
poly_sgd_cv_results = cross_validate(poly_sgd_pipeline,
                                     com_train_features,
                                     com_train_labels,
                                     cv=cv,
                                     scoring="neg_mean_absolute_error",
                                     return_train_score=True,
                                     return_estimator=True)

poly_sgd_train_error = -1 * poly_sgd_cv_results['train_score']
poly_sgd_test_error = -1 * poly_sgd_cv_results['test_score']

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

```
Mean absolute error of linear regression model on the train set:
10824283052.546 +/- 4423288211.832
Mean absolute error of linear regression model on the test set:
10946788540.250 +/- 5396536227.703
```

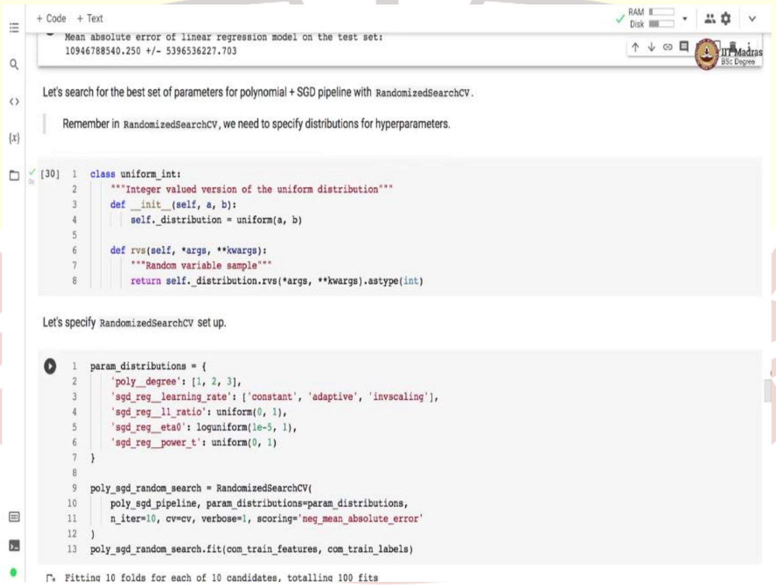
Let's search for the best set of parameters for polynomial + SGD pipeline with `RandomizedSearchCV`.

> Remember in `RandomizedSearchCV`, we need to specify distributions for hyperparameters.

```python
[30] class uniform_int:
         """Integer valued version of the uniform distribution"""
         def __init__(self, a, b):
```

Finally, we will perform the hyper parameter tuning of various parameters in SGD regressor. So, you can see that if we use SGD regressor with elastic net penalty, so elastic net penalty uses both L1 and L2 regularization. So, the base case of elastic net penalty with all other hyper parameters set to the default values we obtained not so great error. So these are very high errors compared to what we have seen so far.

(Refer Slide Time: 24:56)



```
Mean absolute error of linear regression model on the test set:
10946788540.250 +/- 5396536227.703
```

Let's search for the best set of parameters for polynomial + SGD pipeline with `RandomizedSearchCV`.

> Remember in `RandomizedSearchCV`, we need to specify distributions for hyperparameters.

```python
[30] class uniform_int:
         """Integer valued version of the uniform distribution"""
         def __init__(self, a, b):
             self._distribution = uniform(a, b)

         def rvs(self, *args, **kwargs):
             """Random variable sample"""
             return self._distribution.rvs(*args, **kwargs).astype(int)
```

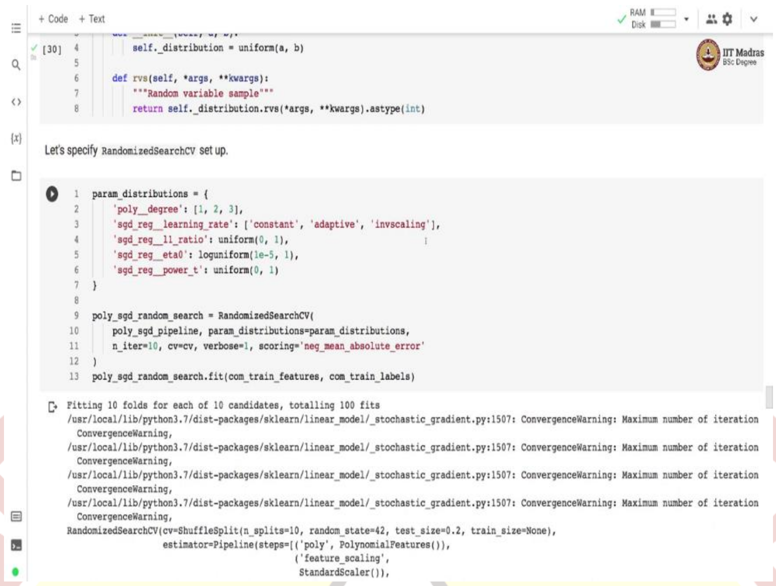
Let's specify `RandomizedSearchCV` set up.

```python
param_distributions = {
    'poly__degree': [1, 2, 3],
    'sgd_reg__learning_rate': ['constant', 'adaptive', 'invscaling'],
    'sgd_reg__l1_ratio': uniform(0, 1),
    'sgd_reg__eta0': loguniform(1e-5, 1),
    'sgd_reg__power_t': uniform(0, 1)
}

poly_sgd_random_search = RandomizedSearchCV(
    poly_sgd_pipeline, param_distributions=param_distributions,
    n_iter=10, cv=cv, verbose=1, scoring='neg_mean_absolute_error'
)
poly_sgd_random_search.fit(com_train_features, com_train_labels)
```

```
Fitting 10 folds for each of 10 candidates, totalling 100 fits
```

So what we do is we perform a hyper parameter tuning on SGD but instead of using grid search CV we are going to use randomize search CV here for different hyper parameters. So, in randomize search CV we need to specify distributions for each hyper parameter.

(Refer Slide Time: 25:11)

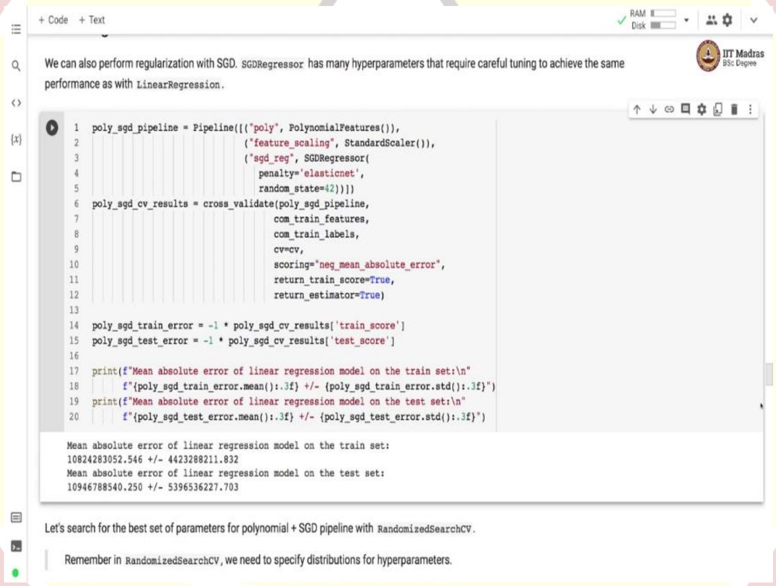


```
        self._distribution = uniform(a, b)

    def rvs(self, *args, **kwargs):
        """Random variable sample"""
        return self._distribution.rvs(*args, **kwargs).astype(int)
```

Let's specify `RandomizedSearchCV` set up.

```
param_distributions = {
    'poly__degree': [1, 2, 3],
    'sgd_reg__learning_rate': ['constant', 'adaptive', 'invscaling'],
    'sgd_reg__l1_ratio': uniform(0, 1),
    'sgd_reg__eta0': loguniform(1e-5, 1),
    'sgd_reg__power_t': uniform(0, 1)
}

poly_sgd_random_search = RandomizedSearchCV(
    poly_sgd_pipeline, param_distributions=param_distributions,
    n_iter=10, cv=cv, verbose=1, scoring='neg_mean_absolute_error'
)
poly_sgd_random_search.fit(com_train_features, com_train_labels)
```

```
Fitting 10 folds for each of 10 candidates, totalling 100 fits
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_stochastic_gradient.py:1507: ConvergenceWarning: Maximum number of iteration
  ConvergenceWarning,
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_stochastic_gradient.py:1507: ConvergenceWarning: Maximum number of iteration
  ConvergenceWarning,
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_stochastic_gradient.py:1507: ConvergenceWarning: Maximum number of iteration
  ConvergenceWarning,
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_stochastic_gradient.py:1507: ConvergenceWarning: Maximum number of iteration
  ConvergenceWarning,
RandomizedSearchCV(cv=ShuffleSplit(n_splits=10, random_state=42, test_size=0.2, train_size=None),
                   estimator=Pipeline(steps=[('poly', PolynomialFeatures()),
                                             ('feature_scaling',
                                              StandardScaler()),
```



We can also perform regularization with SGD. `SGDRegressor` has many hyperparameters that require careful tuning to achieve the same performance as with `LinearRegression`.

```
poly_sgd_pipeline = Pipeline([("poly", PolynomialFeatures()),
                              ("feature_scaling", StandardScaler()),
                              ("sgd_reg", SGDRegressor(
                                  penalty='elasticnet',
                                  random_state=42))])
poly_sgd_cv_results = cross_validate(poly_sgd_pipeline,
                                     com_train_features,
                                     com_train_labels,
                                     cv=cv,
                                     scoring="neg_mean_absolute_error",
                                     return_train_score=True,
                                     return_estimator=True)

poly_sgd_train_error = -1 * poly_sgd_cv_results['train_score']
poly_sgd_test_error = -1 * poly_sgd_cv_results['test_score']

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

```
Mean absolute error of linear regression model on the train set:
10824283052.546 +/- 4423288211.832
Mean absolute error of linear regression model on the test set:
10946788540.250 +/- 5396536227.703
```

Let's search for the best set of parameters for polynomial + SGD pipeline with `RandomizedSearchCV`.

> Remember in `RandomizedSearchCV`, we need to specify distributions for hyperparameters.

So, here we are specifying distributions for different hyper parameters that we want to tune. So, we want to tune the degree  of the polynomial features and we want to try three degrees 1, 2, and 3. We also want to try different learning rate strategies in SGD regressor. And those are basically strings, which are constant adaptive and inverse scaling. For L1 ratio, so L1 ratio is what we specify in elastic net regularization.

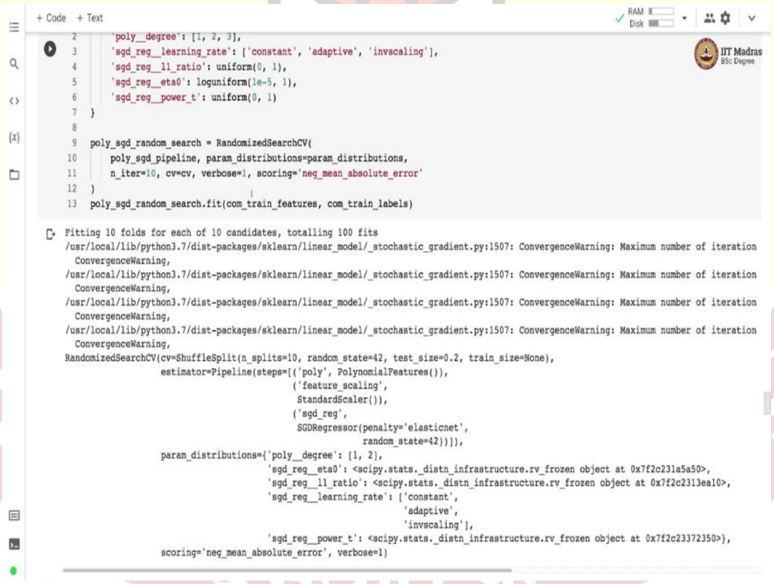
Remember, the elastic net regularization is a convex combination of ridge and lasso regularization and we specify what is the weight that is assigned to L1 or lasso regularizer and the weights that will be assigned to L2 or ridge regularizer will be 1-the weight assigned to lasso regularizer.

So here, we want to try L1 ratio uniformly between 0 to 1, the value of η is  0 which is initial learning rate in SGD regressor we want to try, we want to draw that from log uniform distribution with the range $10^{-5}$ to 1 and the power from the uniform distribution from 0 to 1. So, this power controls, how the learning rate will be reduced from one iteration to the other in case of inverse scaling.

So, we specify the random search CV object with polynomial SGD pipeline as an estimator and that pipeline is what we have set over here, it has got polynomial features followed by feature scaling and then SGD regressor with elastic net penalty. So, we specify the random search CV with poly SGD pipeline as an estimator the parameter distributions for various hyper parameters we want to run randomize search CV for 10 iterations using shuffle split cross validation and we are going to use negative mean absolute error for scoring.

So, number of iteration is a way of controlling, how many times we want to perform the randomize search. So, this is one way of controlling the budget or the time that we spent in performing the parameter search.

(Refer Slide Time: 28:02)

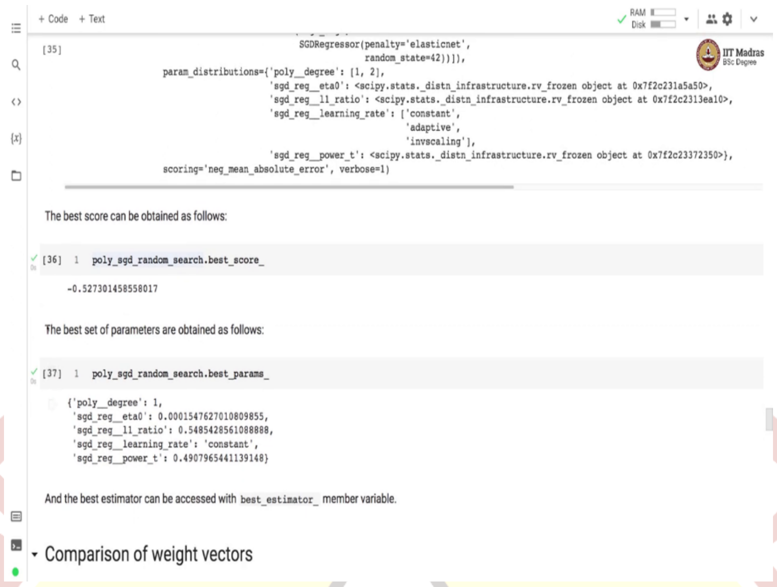


```
    'poly__degree': [1, 2, 3],
    'sgd_reg__learning_rate': ['constant', 'adaptive', 'invscaling'],
    'sgd_reg__l1_ratio': uniform(0, 1),
    'sgd_reg__eta0': loguniform(1e-5, 1),
    'sgd_reg__power_t': uniform(0, 1)
}

poly_sgd_random_search = RandomizedSearchCV(
    poly_sgd_pipeline, param_distributions=param_distributions,
    n_iter=10, cv=cv, verbose=1, scoring='neg_mean_absolute_error'
)
poly_sgd_random_search.fit(com_train_features, com_train_labels)
```

```
Fitting 10 folds for each of 10 candidates, totalling 100 fits
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_stochastic_gradient.py:1507: ConvergenceWarning: Maximum number of iteration
  ConvergenceWarning,
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_stochastic_gradient.py:1507: ConvergenceWarning: Maximum number of iteration
  ConvergenceWarning,
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_stochastic_gradient.py:1507: ConvergenceWarning: Maximum number of iteration
  ConvergenceWarning,
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_stochastic_gradient.py:1507: ConvergenceWarning: Maximum number of iteration
  ConvergenceWarning,
RandomizedSearchCV(cv=ShuffleSplit(n_splits=10, random_state=42, test_size=0.2, train_size=None),
                   estimator=Pipeline(steps=[('poly', PolynomialFeatures()),
                                             ('feature_scaling',
                                              StandardScaler()),
                                             ('sgd_reg',
                                              SGDRegressor(penalty='elasticnet',
                                                           random_state=42))]),
                   param_distributions={'poly__degree': [1, 2],
                                        'sgd_reg__eta0': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7f2c231a5a50>,
                                        'sgd_reg__l1_ratio': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7f2c2313ea10>,
                                        'sgd_reg__learning_rate': ['constant',
                                                                   'adaptive',
                                                                   'invscaling'],
                                        'sgd_reg__power_t': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7f2c23372350>},
                   scoring='neg_mean_absolute_error', verbose=1)
```

So, we basically use the combined feature matrix and labels for training this randomize search CV hyper parameter tuning strategy.

(Refer Slide Time: 28:19)



```
[35]                                  SGDRegressor(penalty='elasticnet',
                                                   random_state=42))]),
                   param_distributions={'poly__degree': [1, 2],
                                        'sgd_reg__eta0': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7f2c231a5a50>,
                                        'sgd_reg__l1_ratio': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7f2c2313ea10>,
                                        'sgd_reg__learning_rate': ['constant',
                                                                   'adaptive',
                                                                   'invscaling'],
                                        'sgd_reg__power_t': <scipy.stats._distn_infrastructure.rv_frozen object at 0x7f2c23372350>},
                   scoring='neg_mean_absolute_error', verbose=1)
```

The best score can be obtained as follows:

```
[36] poly_sgd_random_search.best_score_

-0.527301458558017
```

The best set of parameters are obtained as follows:

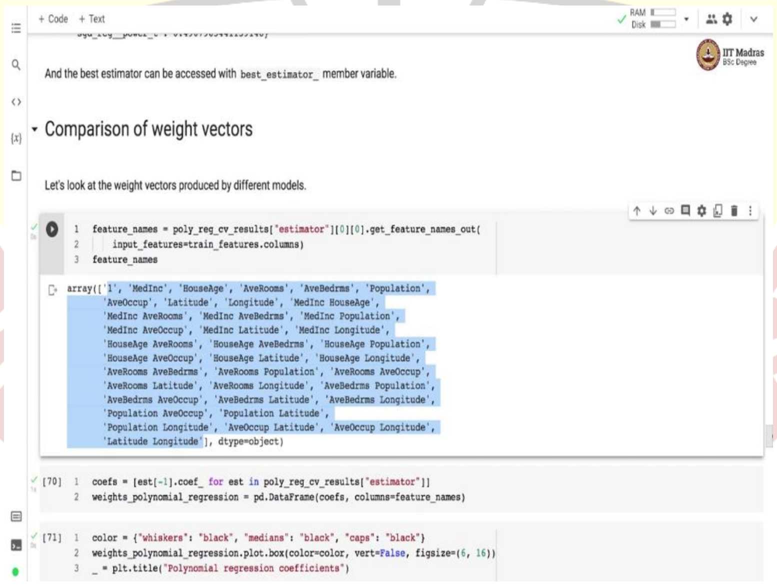
```
[37] poly_sgd_random_search.best_params_

{'poly__degree': 1,
 'sgd_reg__eta0': 0.0001547627010809855,
 'sgd_reg__l1_ratio': 0.5485428561088888,
 'sgd_reg__learning_rate': 'constant',
 'sgd_reg__power_t': 0.4907965441139148}
```

And the best estimator can be accessed with `best_estimator_` member variable.

Comparison of weight vectors

So, we got the best score of -0.52. And then we have gotten the best parameters where polynomial degree is 1, the initial learning rate is 0.00015, the L1 ratio is 0.54, learning rate style is constant and the power_t to be 0.49. So the best estimator can be accessed with best_estimator_member variable of the randomized search CV object.

(Refer Slide Time: 28:57)



And the best estimator can be accessed with `best_estimator_` member variable.

Comparison of weight vectors

Let's look at the weight vectors produced by different models.

```
feature_names = poly_reg_cv_results["estimator"][0][0].get_feature_names_out(
    input_features=train_features.columns)
feature_names

array(['1', 'MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Population',
       'AveOccup', 'Latitude', 'Longitude', 'MedInc HouseAge',
       'MedInc AveRooms', 'MedInc AveBedrms', 'MedInc Population',
       'MedInc AveOccup', 'MedInc Latitude', 'MedInc Longitude',
       'HouseAge AveRooms', 'HouseAge AveBedrms', 'HouseAge Population',
       'HouseAge AveOccup', 'HouseAge Latitude', 'HouseAge Longitude',
       'AveRooms AveBedrms', 'AveRooms Population', 'AveRooms AveOccup',
       'AveRooms Latitude', 'AveRooms Longitude', 'AveBedrms Population',
       'AveBedrms AveOccup', 'AveBedrms Latitude', 'AveBedrms Longitude',
       'Population AveOccup', 'Population Latitude',
       'Population Longitude', 'AveOccup Latitude', 'AveOccup Longitude',
       'Latitude Longitude'], dtype=object)
```

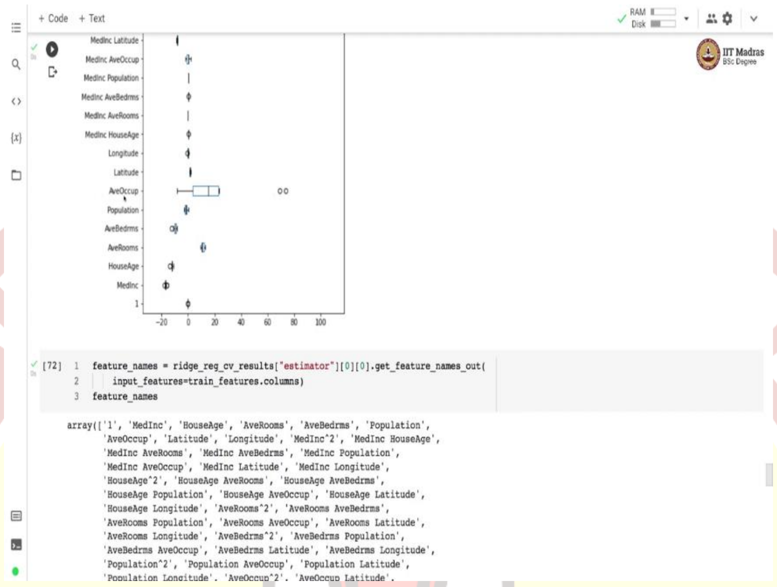
```
[70] coefs = [est[-1].coef_ for est in poly_reg_cv_results["estimator"]]
     weights_polynomial_regression = pd.DataFrame(coefs, columns=feature_names)
```

```
[71] color = {"whiskers": "black", "medians": "black", "caps": "black"}
     weights_polynomial_regression.plot.box(color=color, vert=False, figsize=(6, 16))
     _ = plt.title("Polynomial regression coefficients")
```

So, we can, now that we have trained multiple linear regression models, we can look at the weight vectors produced by different models. So, since we are using polynomial regression, we will first list down the features that we obtained through polynomial regression. So, you can see that these are original features. This is one that we add in polynomial regression. And these are the interaction features.

So here we are combining medium income and house age. So, this is how different features are combined. And here we are mostly using only interaction features, we are not using higher order degrees of each of the features.

(Refer Slide Time: 29:54)

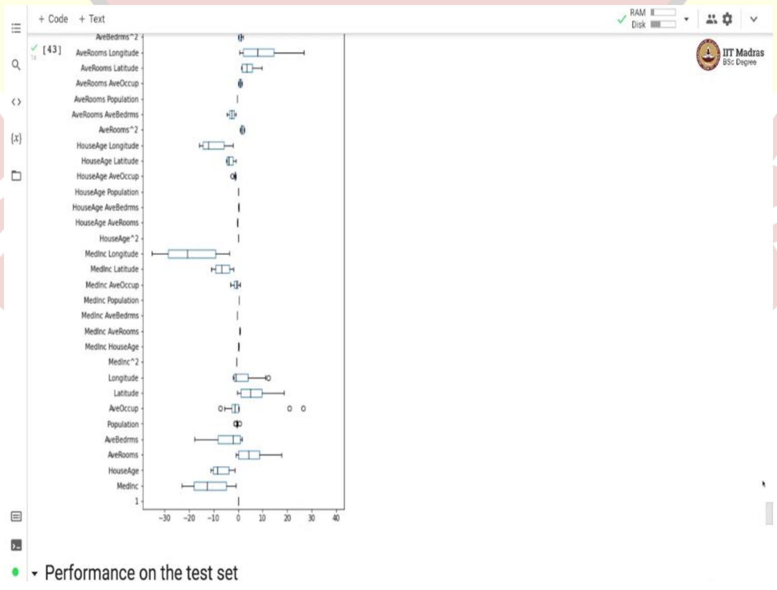


```
[72] feature_names = ridge_reg_cv_results["estimator"][0][0].get_feature_names_out(
         input_features=train_features.columns)
     feature_names
```

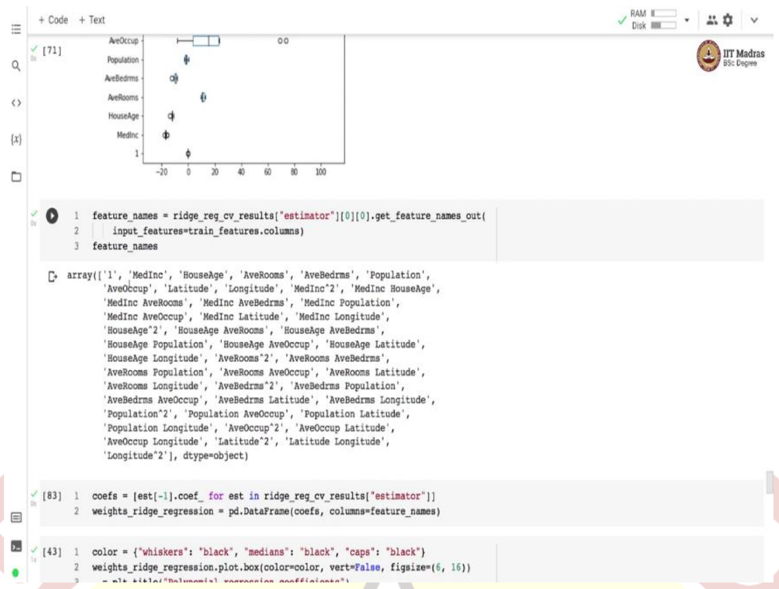
```
array(['1', 'MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Population',
       'AveOccup', 'Latitude', 'Longitude', 'MedInc^2', 'MedInc HouseAge',
       'MedInc AveRooms', 'MedInc AveBedrms', 'MedInc Population',
       'MedInc AveOccup', 'MedInc Latitude', 'MedInc Longitude',
       'HouseAge^2', 'HouseAge AveRooms', 'HouseAge AveBedrms',
       'HouseAge Population', 'HouseAge AveOccup', 'HouseAge Latitude',
       'HouseAge Longitude', 'AveRooms^2', 'AveRooms AveBedrms',
       'AveRooms Population', 'AveRooms AveOccup', 'AveRooms Latitude',
       'AveRooms Longitude', 'AveBedrms^2', 'AveBedrms Population',
       'AveBedrms AveOccup', 'AveBedrms Latitude', 'AveBedrms Longitude',
       'Population^2', 'Population AveOccup', 'Population Latitude',
       'Population Longitude', 'AveOccup^2', 'AveOccup Latitude',
```

And you can see what is the, variation in the in the weights of different polynomials. So, you can see that the point node is the range and  range is quite high. So, something like average occupancy in some folds the weight values given to average occupancies are quite high.

(Refer Slide Time: 30:28)



Performance on the test set



```
feature_names = ridge_reg_cv_results["estimator"][0][0].get_feature_names_out(
    input_features=train_features.columns)
feature_names
```

```
array(['1', 'MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', 'Population',
       'AveOccup', 'Latitude', 'Longitude', 'MedInc^2', 'MedInc HouseAge',
       'MedInc AveRooms', 'MedInc AveBedrms', 'MedInc Population',
       'MedInc AveOccup', 'MedInc Latitude', 'MedInc Longitude',
       'HouseAge^2', 'HouseAge AveRooms', 'HouseAge AveBedrms',
       'HouseAge Population', 'HouseAge AveOccup', 'HouseAge Latitude',
       'HouseAge Longitude', 'AveRooms^2', 'AveRooms AveBedrms',
       'AveRooms Population', 'AveRooms AveOccup', 'AveRooms Latitude',
       'AveRooms Longitude', 'AveBedrms^2', 'AveBedrms Population',
       'AveBedrms AveOccup', 'AveBedrms Latitude', 'AveBedrms Longitude',
       'Population^2', 'Population AveOccup', 'Population Latitude',
       'Population Longitude', 'AveOccup^2', 'AveOccup Latitude',
       'AveOccup Longitude', 'Latitude^2', 'Latitude Longitude',
       'Longitude^2'], dtype=object)
```

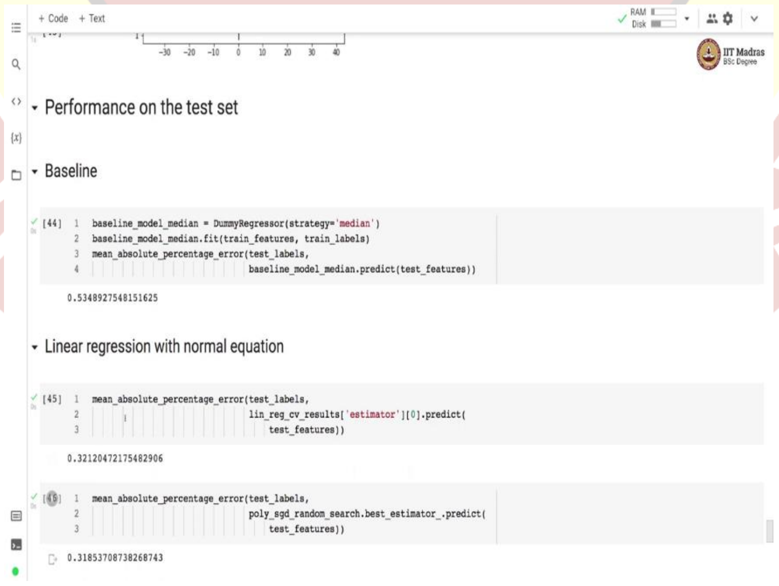
```
[83] coefs = [est[-1].coef_ for est in ridge_reg_cv_results["estimator"]]
     weights_ridge_regression = pd.DataFrame(coefs, columns=feature_names)
```

```
[43] color = {"whiskers": "black", "medians": "black", "caps": "black"}
     weights_ridge_regression.plot.box(color=color, vert=False, figsize=(6, 16))
```

We can fix it, through ridge regression where you can see that the range is now more uniform around 0 it is between -30 to +30 rather than being skewed in earlier case. And in this case, we are using, in case of ridge regression CV we are using the polynomial features which also include a higher degrees of individual features.

So, you can see that this is (medium income)$^2$, this is (medium income* house age), this is (house age)$^2$. So, we are using polynomial features in case of this estimator, which also includes higher order degree features of individual features.

(Refer Slide Time: 31:20)



Performance on the test set

Baseline

```
[44] baseline_model_median = DummyRegressor(strategy='median')
     baseline_model_median.fit(train_features, train_labels)
     mean_absolute_percentage_error(test_labels,
                                    baseline_model_median.predict(test_features))
```

```
0.5348927548151625
```

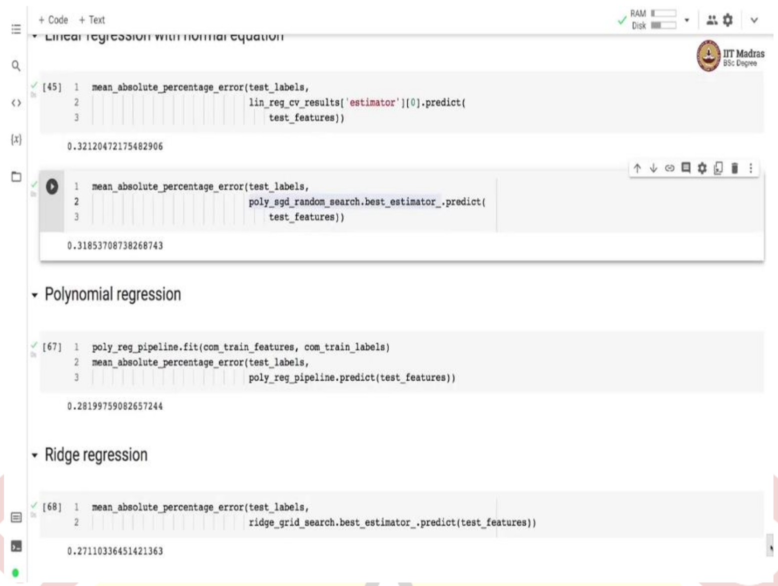
Linear regression with normal equation

```
[45] mean_absolute_percentage_error(test_labels,
                                    lin_reg_cv_results['estimator'][0].predict(
                                      test_features))
```

```
0.32120472175482906
```

```
[46] mean_absolute_percentage_error(test_labels,
                                    poly_sgd_random_search.best_estimator_.predict(
                                      test_features))
```

```
0.31853708738268743
```



Linear regression with normal equation

```
[45] mean_absolute_percentage_error(test_labels,
                                    lin_reg_cv_results['estimator'][0].predict(
                                      test_features))

0.32120472175482906
```

```
mean_absolute_percentage_error(test_labels,
                                    poly_sgd_random_search.best_estimator_.predict(
                                      test_features))

0.31853708738268743
```

Polynomial regression

```
[67] poly_reg_pipeline.fit(com_train_features, com_train_labels)
     mean_absolute_percentage_error(test_labels,
                                    poly_reg_pipeline.predict(test_features))

0.28199759082657244
```

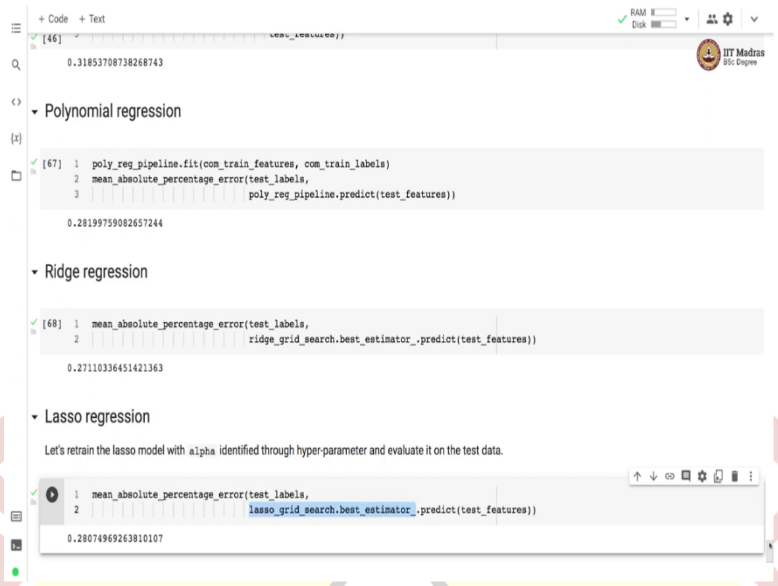
Ridge regression

```
[68] mean_absolute_percentage_error(test_labels,
                                    ridge_grid_search.best_estimator_.predict(test_features))

0.27110336451421363
```

Finally, we can compare the performance of different regression model on the test set. And performance in the test set, this is the true test set that was holdout that was not at all exposed during the training. And this will give us a reasonable estimate of how our models will perform on the feature data. So, we start with the baseline model which is a dummy regressor that gives us the mean absolute percentage error of 0.53; note that we are using mean absolute percentage error as the error metric. And that error metric turns out to be 0.53 for baseline model.

When we use linear regression with normal equation, we get the mean absolute error down to 0.32 and when we use SGD we get it down to 0.31 slightly lower than what we obtained through the normal equation route. And again remember this is the best estimator that we obtained through randomized search CV.

(Refer Slide Time: 32:36)



```
[46]
0.31853708738268743
```

Polynomial regression

```
[67]
poly_reg_pipeline.fit(com_train_features, com_train_labels)
mean_absolute_percentage_error(test_labels,
                               poly_reg_pipeline.predict(test_features))

0.28199759082657244
```

Ridge regression

```
[68]
mean_absolute_percentage_error(test_labels,
                               ridge_grid_search.best_estimator_.predict(test_features))

0.27110336451421363
```

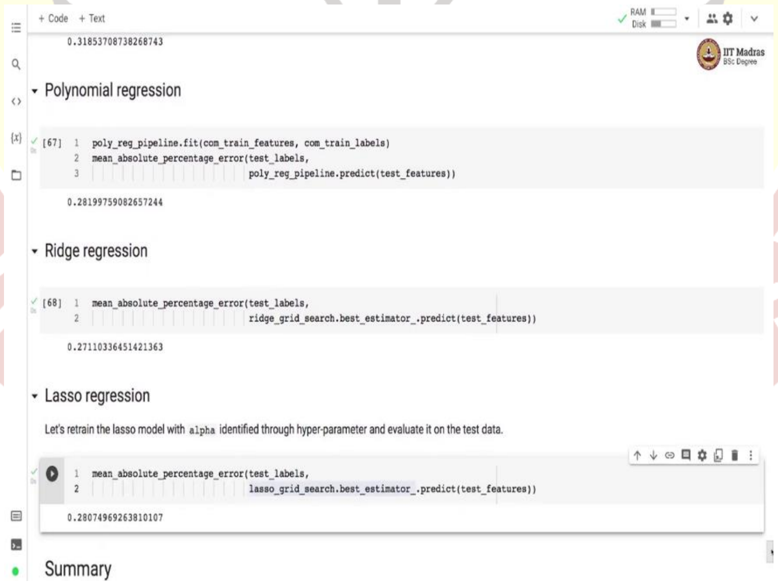
Lasso regression

Let's retrain the lasso model with `alpha` identified through hyper-parameter and evaluate it on the test data.

```
mean_absolute_percentage_error(test_labels,
                               lasso_grid_search.best_estimator_.predict(test_features))

0.28074969263810107
```

For polynomial regression if we use polynomial regression we get it down to 0.28 and we use if we use ridge regression we are getting it down to 0.27 and with lasso it is 0.28. Again it is slightly lower than what we obtain in polynomial regression but again remember here we use, here we use polynomial features of degree 3. Because that is what we found through the grid search CV for the lasso pipeline.

(Refer Slide Time: 33:11)



```
0.31853708738268743
```

Polynomial regression

```
[67]
poly_reg_pipeline.fit(com_train_features, com_train_labels)
mean_absolute_percentage_error(test_labels,
                               poly_reg_pipeline.predict(test_features))

0.28199759082657244
```

Ridge regression

```
[68]
mean_absolute_percentage_error(test_labels,
                               ridge_grid_search.best_estimator_.predict(test_features))

0.27110336451421363
```

Lasso regression

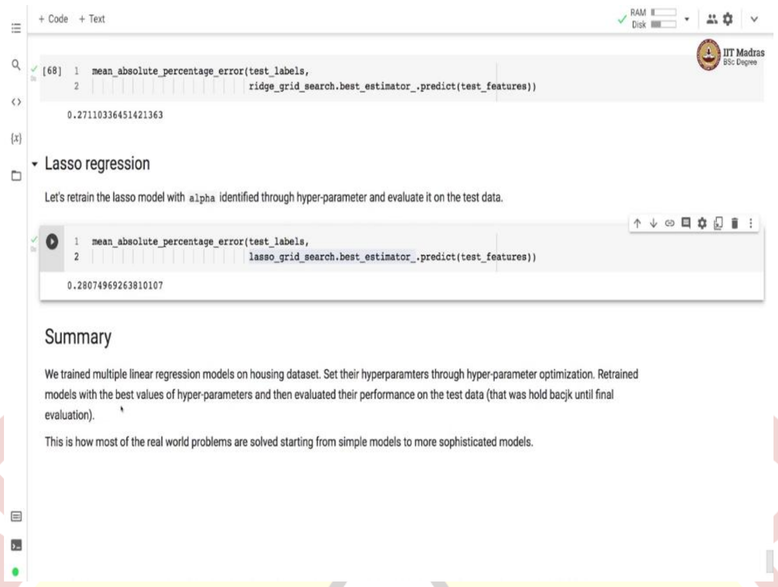
Let's retrain the lasso model with `alpha` identified through hyper-parameter and evaluate it on the test data.

```
mean_absolute_percentage_error(test_labels,
                               lasso_grid_search.best_estimator_.predict(test_features))

0.28074969263810107
```

Summary

So, you can see that among these different classifiers ridge regression seems to be doing pretty well and we can use ridge regression for future prediction because it has given by far the best performance than the other regression methods.

(Refer Slide Time: 33:33)



```
[68] mean_absolute_percentage_error(test_labels,
                                    ridge_grid_search.best_estimator_.predict(test_features))

0.27110336451421363
```

Lasso regression

Let's retrain the lasso model with `alpha` identified through hyper-parameter and evaluate it on the test data.

```
mean_absolute_percentage_error(test_labels,
                               lasso_grid_search.best_estimator_.predict(test_features))

0.28074969263810107
```

Summary

We trained multiple linear regression models on housing dataset. Set their hyperparamters through hyper-parameter optimization. Retrained models with the best values of hyper-parameters and then evaluated their performance on the test data (that was hold bacjk until final evaluation).

This is how most of the real world problems are solved starting from simple models to more sophisticated models.

So, here in this colab we train multiple linear regression models on housing data set. We set their hyper parameters, through hyper parameter optimization methods like grid search CV, randomized search CV and some specialized cross validations like ridge CV and lasso CV. So, we retrain the models with best values of hyper parameters and evaluated their performance on the test set. We did not do this explicitly but this happens as part of randomized search CV and grid search CV procedures.

And you know this particular colab is an example of how most of the real world problems are solved. So we started with simple baseline model like dummy regressor and then we went on building more and more sophisticated models. So, this particular colab serves as a template of how you can build regressions models for the task at hand.