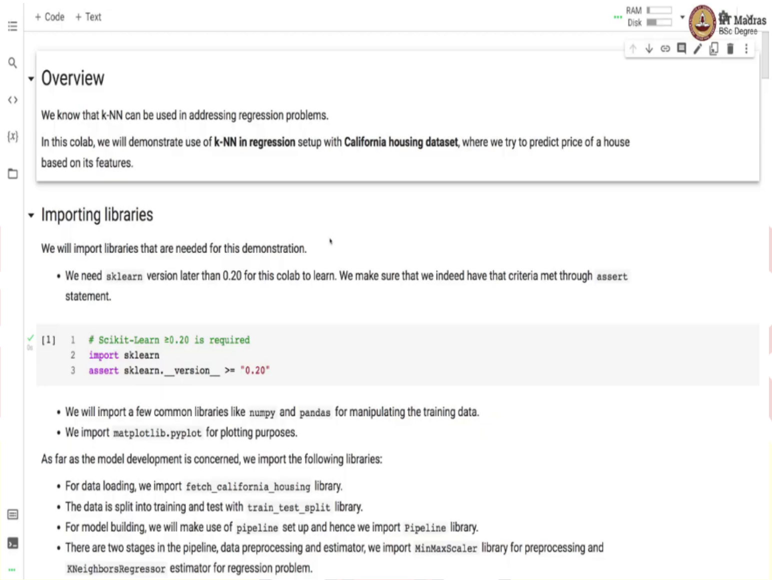
INDIAN INSTITUTE OF TECHNOLOGY MADRAS • सिद्धिर्भवति कर्मजा •

IIT Madras

ONLINE DEGREE

Machine Learning Program
Indian Institute of Technology, Madras
Demonstration: K-NN with California Housing Dataset

(Refer Slide Time: 0:10)



Overview

We know that k-NN can be used in addressing regression problems.

In this colab, we will demonstrate use of **k-NN in regression** setup with **California housing dataset**, where we try to predict price of a house based on its features.

Importing libraries

We will import libraries that are needed for this demonstration.

- We need `sklearn` version later than 0.20 for this colab to learn. We make sure that we indeed have that criteria met through `assert` statement.

```
[1]  # Scikit-Learn ≥0.20 is required
     import sklearn
     assert sklearn.__version__ >= "0.20"
```

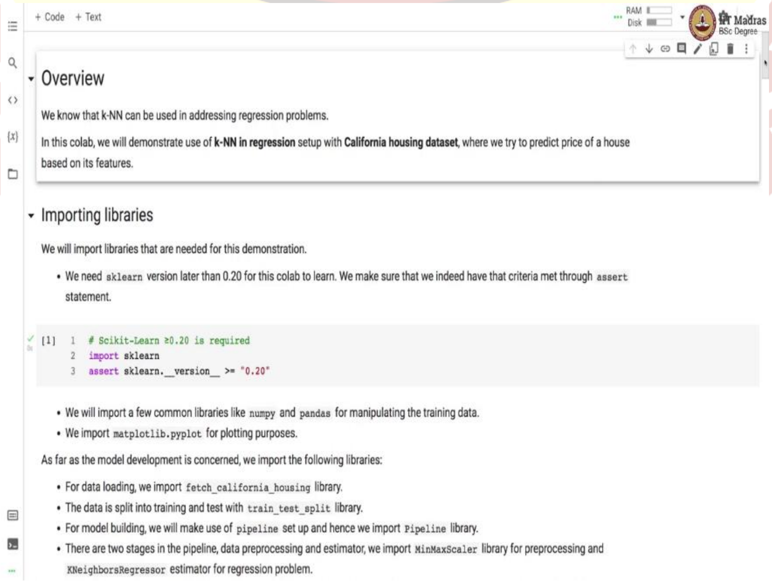
- We will import a few common libraries like `numpy` and `pandas` for manipulating the training data.
- We import `matplotlib.pyplot` for plotting purposes.

As far as the model development is concerned, we import the following libraries:

- For data loading, we import `fetch_california_housing` library.
- The data is split into training and test with `train_test_split` library.
- For model building, we will make use of `pipeline` set up and hence we import `Pipeline` library.
- There are two stages in the pipeline, data preprocessing and estimator, we import `MinMaxScaler` library for preprocessing and `KNeighborsRegressor` estimator for regression problem.

Namaste! Welcome to the next video of machine learning practice course. In this video, we will demonstrate use of K-NN in regression setup with California Housing dataset, where we will try to predict price of a house based on its feature. We know that K-NN can be used in addressing regression problems. And this notebook is an attempt to demonstrate application of gain in regression problems.

(Refer Slide Time: 0:43)



Overview

We know that k-NN can be used in addressing regression problems.

In this colab, we will demonstrate use of **k-NN in regression** setup with **California housing dataset**, where we try to predict price of a house based on its features.

Importing libraries

We will import libraries that are needed for this demonstration.

- We need `sklearn` version later than 0.20 for this colab to learn. We make sure that we indeed have that criteria met through `assert` statement.

```
[1]  # Scikit-Learn ≥0.20 is required
     import sklearn
     assert sklearn.__version__ >= "0.20"
```

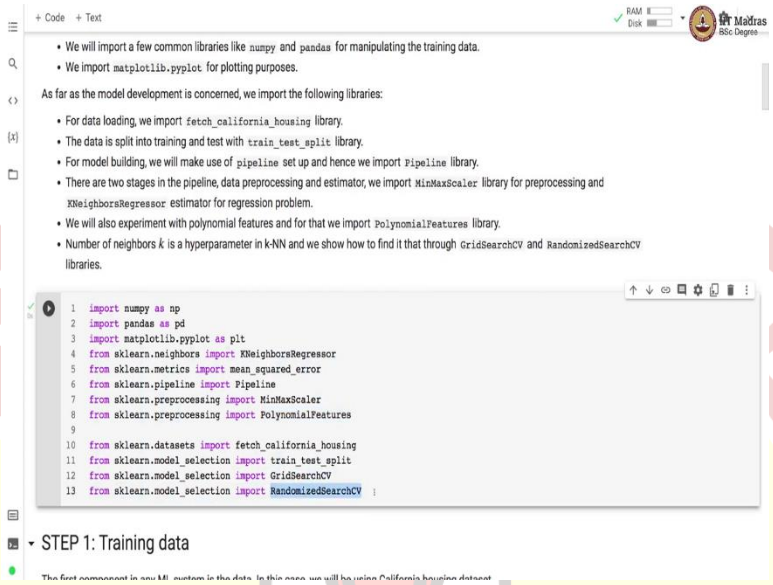
- We will import a few common libraries like `numpy` and `pandas` for manipulating the training data.
- We import `matplotlib.pyplot` for plotting purposes.

As far as the model development is concerned, we import the following libraries:

- For data loading, we import `fetch_california_housing` library.
- The data is split into training and test with `train_test_split` library.
- For model building, we will make use of `pipeline` set up and hence we import `Pipeline` library.
- There are two stages in the pipeline, data preprocessing and estimator, we import `MinMaxScaler` library for preprocessing and `KNeighborsRegressor` estimator for regression problem.

Will import libraries that are needed for this demonstration, we need scikit-learn version later than 0.20 for this colab to run, we make sure that we indeed have that criteria made to the assert statement.

(Refer Slide Time: 01:04)



- We will import a few common libraries like `numpy` and `pandas` for manipulating the training data.
- We import `matplotlib.pyplot` for plotting purposes.

As far as the model development is concerned, we import the following libraries:

- For data loading, we import `fetch_california_housing` library.
- The data is split into training and test with `train_test_split` library.
- For model building, we will make use of `pipeline` set up and hence we import `Pipeline` library.
- There are two stages in the pipeline, data preprocessing and estimator, we import `MinMaxScaler` library for preprocessing and `KNeighborsRegressor` estimator for regression problem.
- We will also experiment with polynomial features and for that we import `PolynomialFeatures` library.
- Number of neighbors k is a hyperparameter in k-NN and we show how to find it that through `GridSearchCV` and `RandomizedSearchCV` libraries.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import PolynomialFeatures

from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
```

STEP 1: Training data

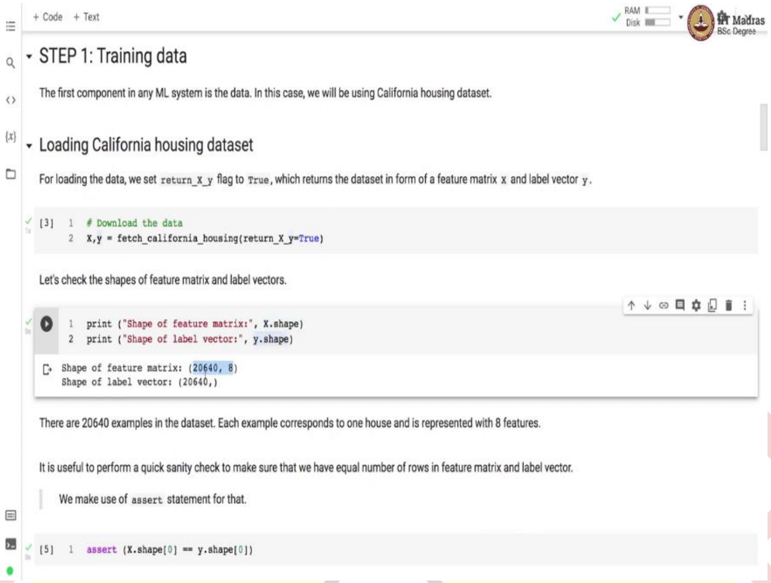
The first component in any ML system is the data. In this case, we will be using California housing dataset

Next, we will import a few common libraries like NumPy, and Pandas for manipulating the training data. We import matplotlib.pyplot for plotting various graphs and figures. As far as model development is concerned, we import the following libraries. For data loading, we import fetch _California _housing library.

The data is split into training and test with train _test _split library. For model building, we use pipeline setup and hence we import the pipeline library. There are 2 stages in the pipeline data processing and estimator. We import MinMaxScaler for preprocessing and KNeighborsRegressor for regression.

We will also explain to polynomial features. And for that we import PolynomialFeatures library. The number of neighbors, k is hyper-parameter in K-NN and we show how to find it through GridSearchCV and RandomizedSearchCV libraries.

(Refer Slide Time: 02:20)



STEP 1: Training data

The first component in any ML system is the data. In this case, we will be using California housing dataset.

Loading California housing dataset

For loading the data, we set `return_X_y` flag to `True`, which returns the dataset in form of a feature matrix `X` and label vector `y`.

```
[3]  # Download the data
     X,y = fetch_california_housing(return_X_y=True)
```

Let's check the shapes of feature matrix and label vectors.

```
print ("Shape of feature matrix:", X.shape)
print ("Shape of label vector:", y.shape)
```

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

There are 20640 examples in the dataset. Each example corresponds to one house and is represented with 8 features.

It is useful to perform a quick sanity check to make sure that we have equal number of rows in feature matrix and label vector.

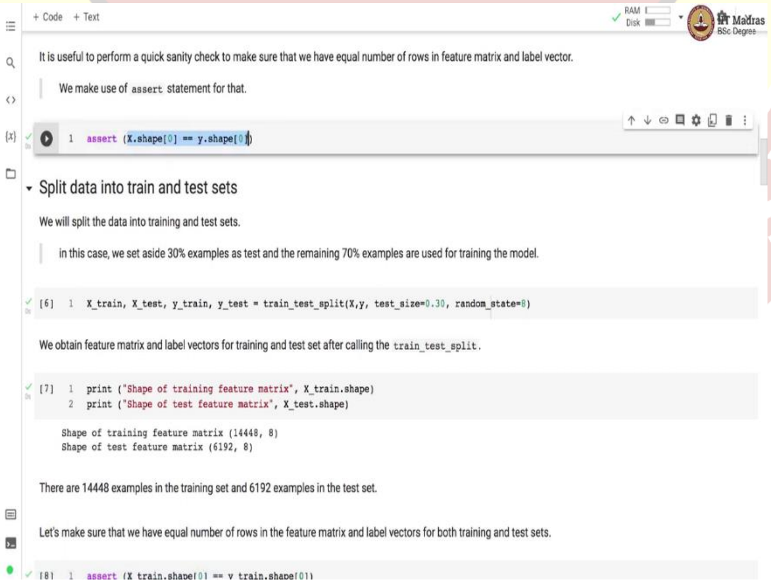
> We make use of `assert` statement for that.

```
[5]  assert (X.shape[0] == y.shape[0])
```

So, that was about importing various libraries for this demo. The most important prerequisite in any ML system is the data. In this case, we will be using California Housing dataset. For loading the data, we set return _X _y parameter or flag to true which returns the data set in form of a feature matrix X and label vector y.

Let us check the shapes of feature matrix and label vector. There are 20,640 examples in the dataset. Each example corresponds to one house and is represented with 8 features.

(Refer Slide Time: 03:09)



It is useful to perform a quick sanity check to make sure that we have equal number of rows in feature matrix and label vector.

> We make use of `assert` statement for that.

```
assert (X.shape[0] == y.shape[0])
```

Split data into train and test sets

We will split the data into training and test sets.

> in this case, we set aside 30% examples as test and the remaining 70% examples are used for training the model.

```
[6]  X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.30, random_state=8)
```

We obtain feature matrix and label vectors for training and test set after calling the `train_test_split`.

```
[7]  print ("Shape of training feature matrix", X_train.shape)
     print ("Shape of test feature matrix", X_test.shape)
```

```
Shape of training feature matrix (14448, 8)
Shape of test feature matrix (6192, 8)
```

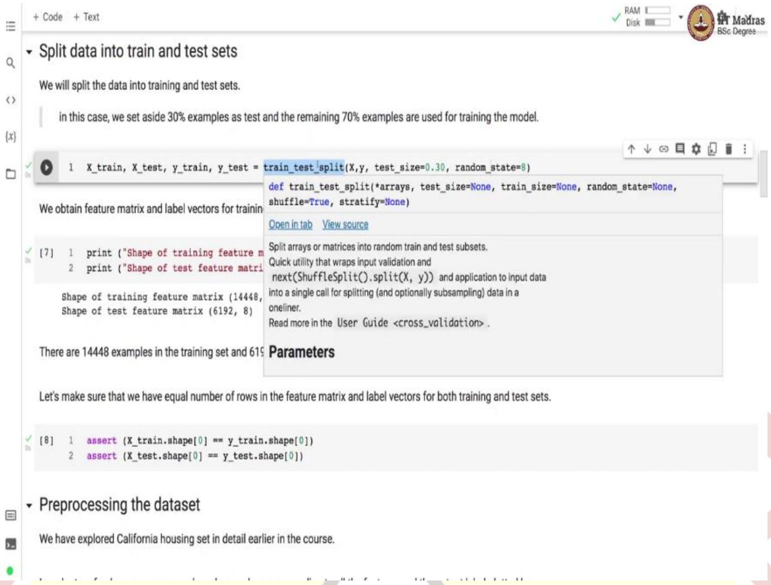
There are 14448 examples in the training set and 6192 examples in the test set.

Let's make sure that we have equal number of rows in the feature matrix and label vectors for both training and test sets.

```
[8]  assert (X_train.shape[0] == y_train.shape[0])
```

It is useful to perform a quick sanity check to make sure you have equal number of rows in feature matrix and label vector, you make use of an assert statement for that purpose and assertion is true, hence the condition is met.

(Refer Slide Time: 03:28)



Split data into train and test sets

We will split the data into training and test sets.

> in this case, we set aside 30% examples as test and the remaining 70% examples are used for training the model.

```
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.30, random_state=8)
```

```
def train_test_split(*arrays, test_size=None, train_size=None, random_state=None,
shuffle=True, stratify=None)
```

Open in tab View source

Split arrays or matrices into random train and test subsets.
Quick utility that wraps input validation and `next(ShuffleSplit().split(X, y))` and application to input data into a single call for splitting (and optionally subsampling) data in a oneliner.
Read more in the `User Guide <cross_validation>`.

Parameters

We obtain feature matrix and label vectors for trainin

```
[7]  print ("Shape of training feature m
     print ("Shape of test feature matri
```

```
Shape of training feature matrix (14448,
Shape of test feature matrix (6192, 8)
```

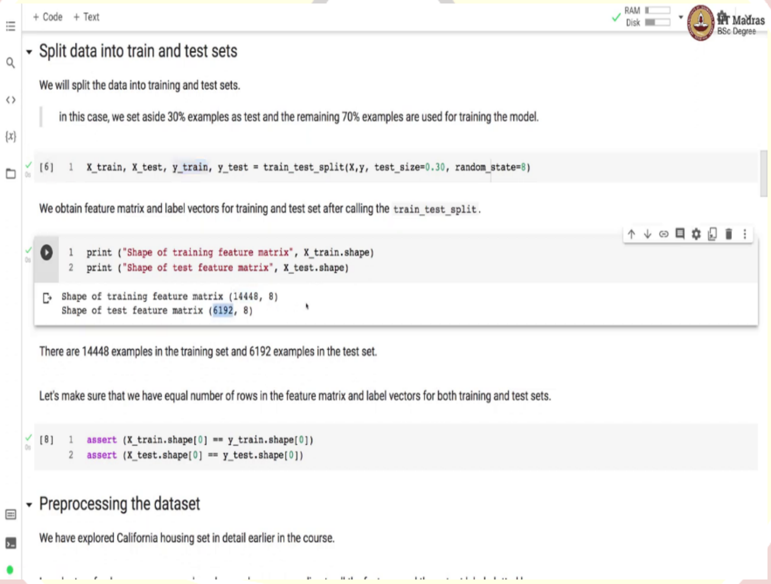
There are 14448 examples in the training set and 619

Let's make sure that we have equal number of rows in the feature matrix and label vectors for both training and test sets.

```
[8]  assert (X_train.shape[0] == y_train.shape[0])
     assert (X_test.shape[0] == y_test.shape[0])
```

Preprocessing the dataset

We have explored California housing set in detail earlier in the course.



Split data into train and test sets

We will split the data into training and test sets.

> in this case, we set aside 30% examples as test and the remaining 70% examples are used for training the model.

```
[6]  X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.30, random_state=8)
```

We obtain feature matrix and label vectors for training and test set after calling the `train_test_split`.

```
print ("Shape of training feature matrix", X_train.shape)
print ("Shape of test feature matrix", X_test.shape)
```

```
Shape of training feature matrix (14448, 8)
Shape of test feature matrix (6192, 8)
```

There are 14448 examples in the training set and 6192 examples in the test set.

Let's make sure that we have equal number of rows in the feature matrix and label vectors for both training and test sets.

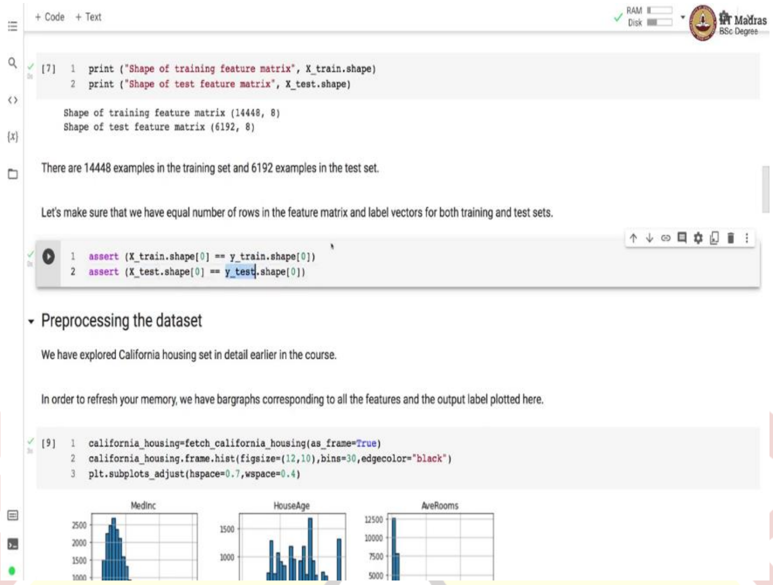
```
[8]  assert (X_train.shape[0] == y_train.shape[0])
     assert (X_test.shape[0] == y_test.shape[0])
```

Preprocessing the dataset

We have explored California housing set in detail earlier in the course.

Next, we will split the data into training and test sets. For that, we use train _test _split library. And we basically set aside 30% examples as test and remaining 70% examples are used for training the model. We obtain feature matrix and label vectors corresponding to training and test sets. We quickly check the shapes of training feature metrics and the test feature metrics. So, there are 14,448 examples in the training set and 6,192 examples in the test set.

(Refer Slide Time: 04:15)



```
[7]  print ("Shape of training feature matrix", X_train.shape)
     print ("Shape of test feature matrix", X_test.shape)

     Shape of training feature matrix (14448, 8)
     Shape of test feature matrix (6192, 8)
```

There are 14448 examples in the training set and 6192 examples in the test set.

Let's make sure that we have equal number of rows in the feature matrix and label vectors for both training and test sets.

```
assert (X_train.shape[0] == y_train.shape[0])
assert (X_test.shape[0] == y_test.shape[0])
```

Preprocessing the dataset

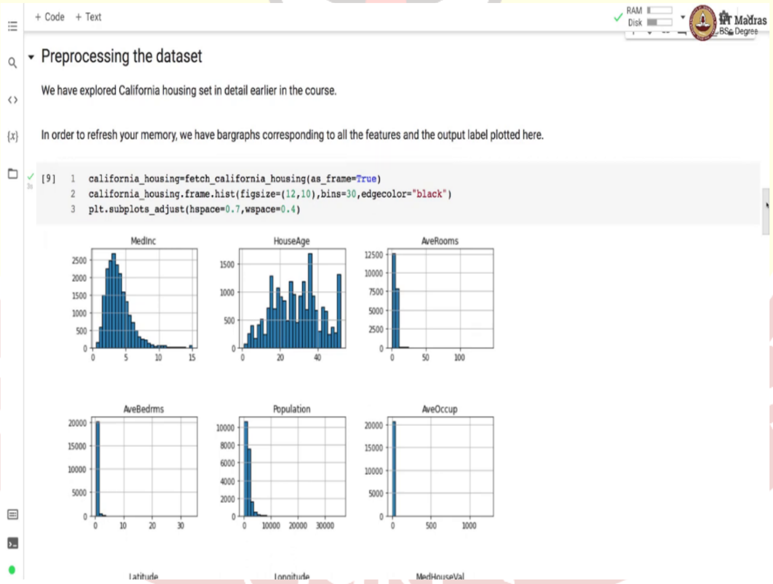
We have explored California housing set in detail earlier in the course.

In order to refresh your memory, we have bargraphs corresponding to all the features and the output label plotted here.

```
[9]  california_housing=fetch_california_housing(as_frame=True)
     california_housing.frame.hist(figsize=(12,10),bins=30,edgecolor="black")
     plt.subplots_adjust(hspace=0.7,wspace=0.4)
```

Let us make sure that we have equal number of rows in the feature matrix and label vectors of both training and test sets. And both these assertions turned out to be true. Hence, we are good to proceed further.

(Refer Slide Time: 04:33)

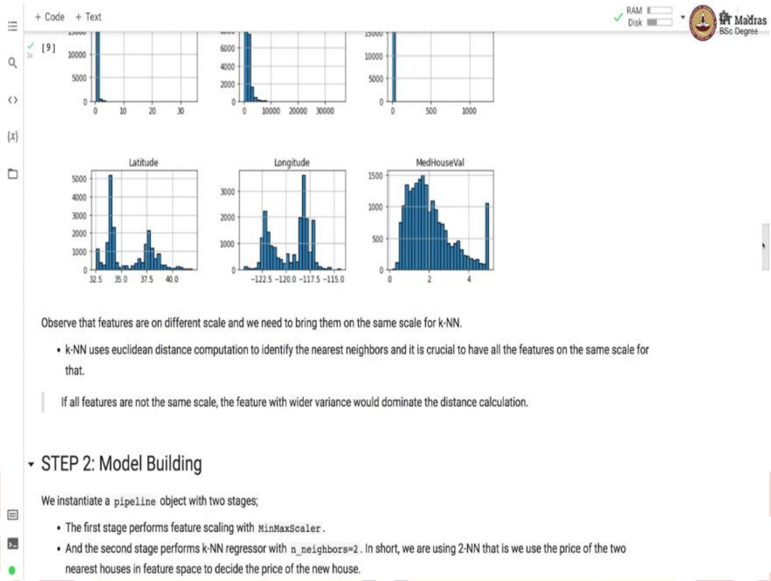


Preprocessing the dataset

We have explored California housing set in detail earlier in the course.

In order to refresh your memory, we have bargraphs corresponding to all the features and the output label plotted here.

```
[9]  california_housing=fetch_california_housing(as_frame=True)
     california_housing.frame.hist(figsize=(12,10),bins=30,edgecolor="black")
     plt.subplots_adjust(hspace=0.7,wspace=0.4)
```



Observe that features are on different scale and we need to bring them on the same scale for k-NN.

- k-NN uses euclidean distance computation to identify the nearest neighbors and it is crucial to have all the features on the same scale for that.

> If all features are not the same scale, the feature with wider variance would dominate the distance calculation.

STEP 2: Model Building

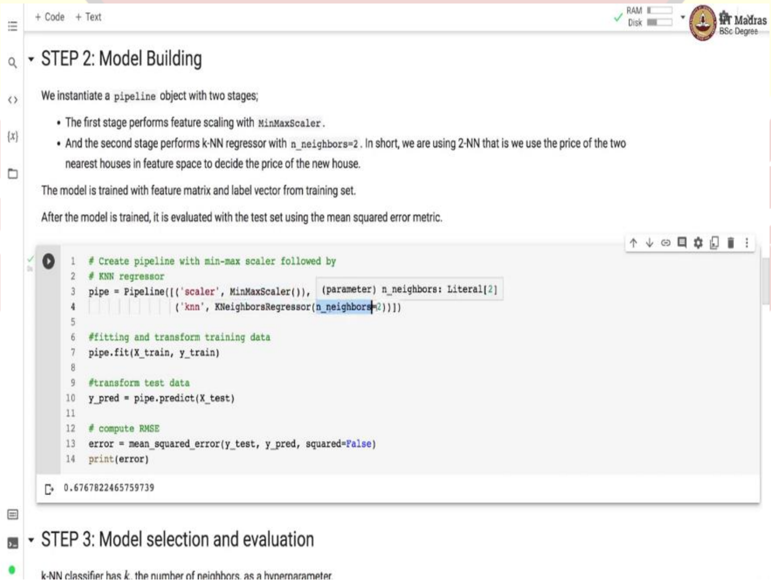
We instantiate a `pipeline` object with two stages;

- The first stage performs feature scaling with `MinMaxScaler`.
- And the second stage performs k-NN regressor with `n_neighbors=2`. In short, we are using 2-NN that is we use the price of the two nearest houses in feature space to decide the price of the new house.

Next step is preprocessing the dataset. We have explored California housing dataset in detail earlier in this course. In order to refresh your memory, we have bar graphs corresponding to all the features and output label plotted here. So, observe that features are on different scale and we need to bring them on the same scale for k-NN.

Remember that k-NN uses Euclidean distance computation to identify the nearest neighbors. And it is crucial to have all the features on the same scale. For that, if features are not on the same scale, the feature with wider variance would dominate the distance calculation.

(Refer Slide Time:05:21)



STEP 2: Model Building

We instantiate a `pipeline` object with two stages;

- The first stage performs feature scaling with `MinMaxScaler`.
- And the second stage performs k-NN regressor with `n_neighbors=2`. In short, we are using 2-NN that is we use the price of the two nearest houses in feature space to decide the price of the new house.

The model is trained with feature matrix and label vector from training set.

After the model is trained, it is evaluated with the test set using the mean squared error metric.

```
# Create pipeline with min-max scaler followed by
# KNN regressor
pipe = Pipeline([('scaler', MinMaxScaler()),  (parameter) n_neighbors: Literal[2]
                 ('knn', KNeighborsRegressor(n_neighbors=2))])

#fitting and transform training data
pipe.fit(X_train, y_train)

#transform test data
y_pred = pipe.predict(X_test)

# compute RMSE
error = mean_squared_error(y_test, y_pred, squared=False)
print(error)
```

```
0.6767822465759739
```

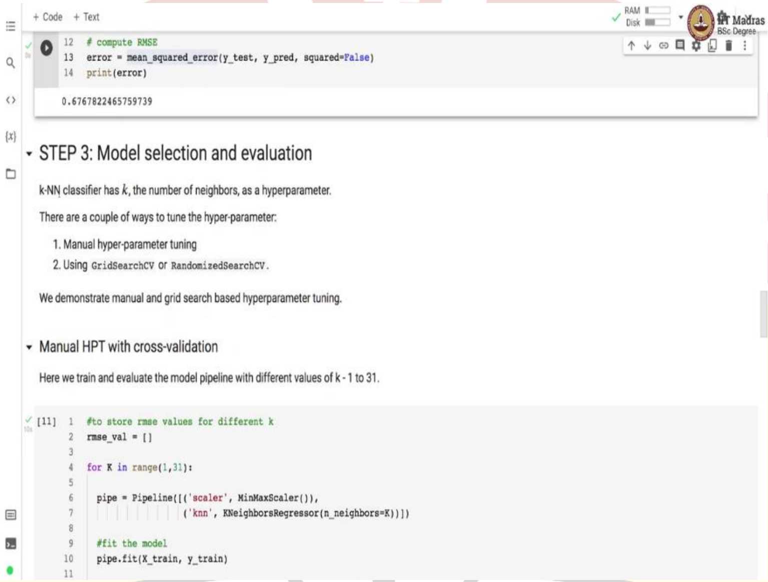
STEP 3: Model selection and evaluation

k-NN classifier has k, the number of neighbors, as a hyperparameter.

The next step is model building. For that we instantiate a pipeline object with 2 stages. The first stage performs feature scaling with MinMaxScaler. And the second stage performs gain and regressor with the number of neighbors equal to 2. In short, we are using 2-NN that is we use the price of 2 nearest houses in the feature space to decide the price of the new house.

The model is trained with feature matrix and label vector from the training set. After the model is trained, it is evaluated using the test set with mean squared error metric. So, we get the mean squared error of 0.67 on the test set.

(Refer Slide Time: 06:17)



```
# compute RMSE
error = mean_squared_error(y_test, y_pred, squared=False)
print(error)
```

```
0.6767822465759739
```

STEP 3: Model selection and evaluation

k-NN classifier has k, the number of neighbors, as a hyperparameter.

There are a couple of ways to tune the hyper-parameter:

1. Manual hyper-parameter tuning
2. Using `GridSearchCV` or `RandomizedSearchCV`.

We demonstrate manual and grid search based hyperparameter tuning.

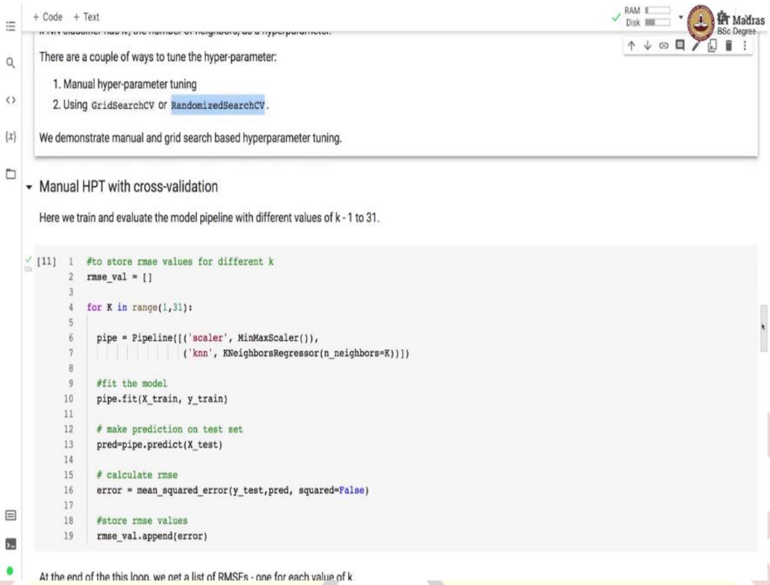
Manual HPT with cross-validation

Here we train and evaluate the model pipeline with different values of k - 1 to 31.

```
[11]
#to store rmse values for different k
rmse_val = []

for K in range(1,31):

    pipe = Pipeline([('scaler', MinMaxScaler()),
                     ('knn', KNeighborsRegressor(n_neighbors=K))])

    #fit the model
    pipe.fit(X_train, y_train)

```

So, the next step is model selection and evaluation. So, K-NN classifier has k, which is a number of neighbors as a hyper-parameter. There are a couple of ways to tune the hyper-parameter. One is manual hyper-parameter tuning, or using automated ways like GridSearchCV, or randomizedSearchCV. We demonstrate manual and grid search based hyper-parameter tuning in this demo.

(Refer Slide Time: 06:48)



There are a couple of ways to tune the hyper-parameter:

1. Manual hyper-parameter tuning
2. Using `GridSearchCV` or `RandomizedSearchCV`.

We demonstrate manual and grid search based hyperparameter tuning.

Manual HPT with cross-validation

Here we train and evaluate the model pipeline with different values of k - 1 to 31.

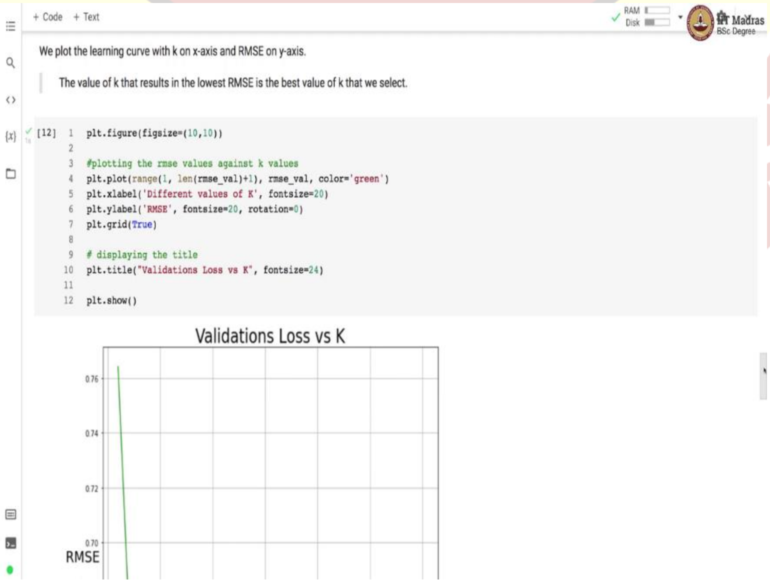
```
[11]
#to store rmse values for different k
rmse_val = []

for K in range(1,31):

  pipe = Pipeline([('scaler', MinMaxScaler()),
                   ('knn', KNeighborsRegressor(n_neighbors=K))])

  #fit the model
  pipe.fit(X_train, y_train)

  # make prediction on test set
  pred=pipe.predict(X_test)

  # calculate rmse
  error = mean_squared_error(y_test,pred, squared=False)

  #store rmse values
  rmse_val.append(error)
```

At the end of the this loop, we get a list of RMSEs - one for each value of k.

In manual hyper-parameter tuning, we use cross-validation. Here we train and evaluate the model pipeline for different values of k in the range between 1 and 31. So, we have an outer loop running here, which has different values of k. And then we have the same pipeline setup as before, except that the number of nearest neighbor are now parameterized.

And they take the value of k, which varies iteration after iteration, then we fit the model, we make the prediction on the test set and we calculate the RMSE metric on the test set. At the end of the loop, we get a list of RMSE 1 for each value of k.

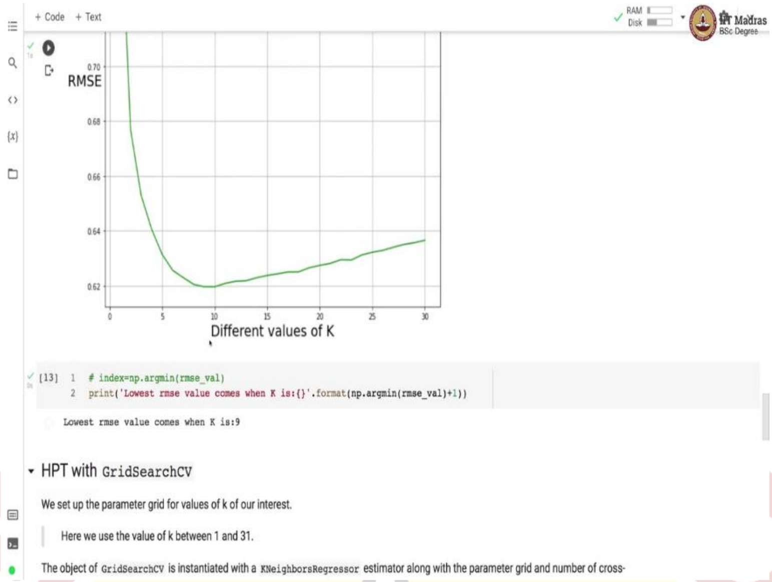
(Refer Slide Time: 07:37)



We plot the learning curve with k on x-axis and RMSE on y-axis.

> The value of k that results in the lowest RMSE is the best value of k that we select.

```
[12]
plt.figure(figsize=(10,10))

#plotting the rmse values against k values
plt.plot(range(1, len(rmse_val)+1), rmse_val, color='green')
plt.xlabel('Different values of K', fontsize=20)
plt.ylabel('RMSE', fontsize=20, rotation=0)
plt.grid(True)

# displaying the title
plt.title("Validations Loss vs K", fontsize=24)

plt.show()
```



```
[13]
# index=np.argmin(rmse_val)
print('Lowest rmse value comes when K is:{}'.format(np.argmin(rmse_val)+1))
```

```
Lowest rmse value comes when K is:9
```

HPT with `GridSearchCV`

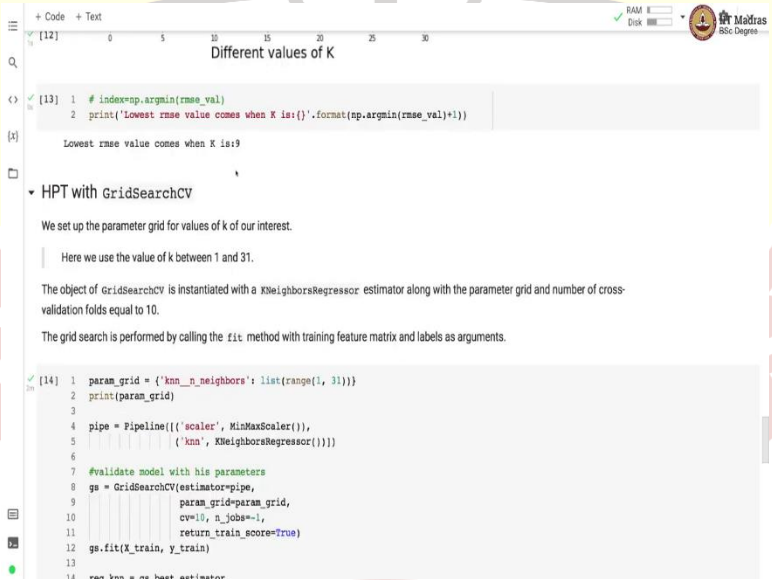
We set up the parameter grid for values of k of our interest.

> Here we use the value of k between 1 and 31.

The object of `GridSearchCV` is instantiated with a `KNeighborsRegressor` estimator along with the parameter grid and number of cross-

We plot the learning curve with k on x-axis and RMSE on y-axis, the value of k that result in the lowest RMSE is the best value k that we select. So, here we plot the graph where we have k on x-axis and RMSE on y-axis, and you can see that initially the RMSE dropped sharply up to a point after which the RMSE start going up. So, this point is possibly the best value of k that we can use in our estimator. So, the lowest RMSE value is at k equal to 9.

(Refer Slide Time: 08:22)



```
[13]
# index=np.argmin(rmse_val)
print('Lowest rmse value comes when K is:{}'.format(np.argmin(rmse_val)+1))
```

```
Lowest rmse value comes when K is:9
```

HPT with `GridSearchCV`

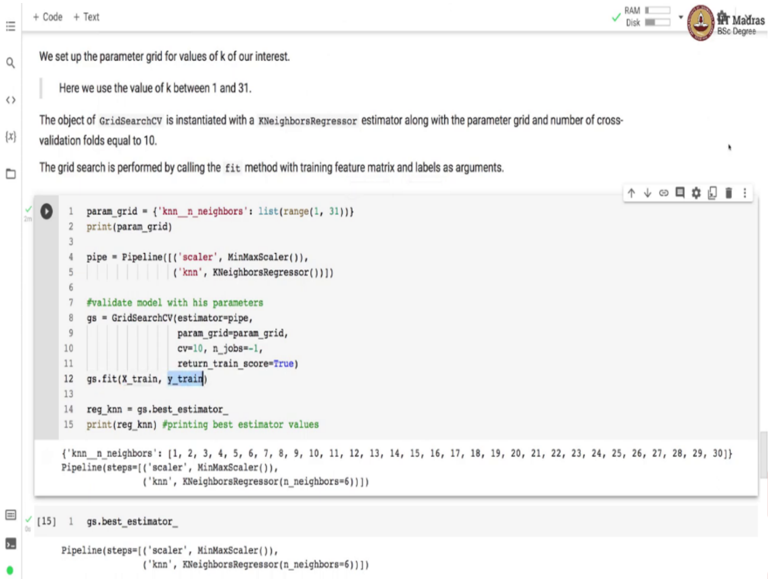
We set up the parameter grid for values of k of our interest.

> Here we use the value of k between 1 and 31.

The object of `GridSearchCV` is instantiated with a `KNeighborsRegressor` estimator along with the parameter grid and number of cross-validation folds equal to 10.

The grid search is performed by calling the `fit` method with training feature matrix and labels as arguments.

```
[14]
param_grid = {'knn__n_neighbors': list(range(1, 31))}
print(param_grid)

pipe = Pipeline([('scaler', MinMaxScaler()),
                 ('knn', KNeighborsRegressor())])

#validate model with his parameters
gs = GridSearchCV(estimator=pipe,
                  param_grid=param_grid,
                  cv=10, n_jobs=-1,
                  return_train_score=True)
gs.fit(X_train, y_train)

reg_knn = gs.best_estimator
```



We set up the parameter grid for values of k of our interest.

> Here we use the value of k between 1 and 31.

The object of `GridSearchCV` is instantiated with a `KNeighborsRegressor` estimator along with the parameter grid and number of cross-validation folds equal to 10.

The grid search is performed by calling the `fit` method with training feature matrix and labels as arguments.

```
param_grid = {'knn__n_neighbors': list(range(1, 31))}
print(param_grid)

pipe = Pipeline([('scaler', MinMaxScaler()),
                 ('knn', KNeighborsRegressor())])

#validate model with his parameters
gs = GridSearchCV(estimator=pipe,
                  param_grid=param_grid,
                  cv=10, n_jobs=-1,
                  return_train_score=True)
gs.fit(X_train, y_train)

reg_knn = gs.best_estimator_
print(reg_knn) #printing best estimator values
```

```
{'knn__n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]}
Pipeline(steps=[('scaler', MinMaxScaler()),
                ('knn', KNeighborsRegressor(n_neighbors=6))])
```

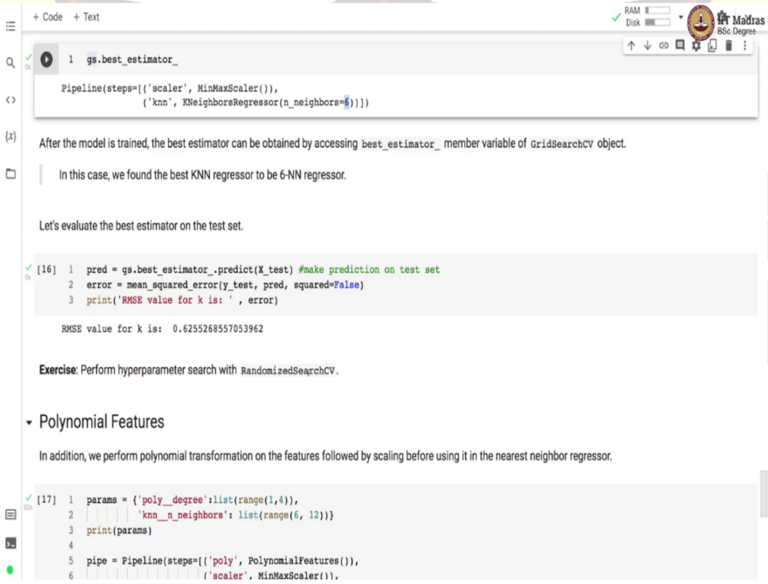
```
[15] gs.best_estimator_
```

```
Pipeline(steps=[('scaler', MinMaxScaler()),
                ('knn', KNeighborsRegressor(n_neighbors=6))])
```

We can perform the same thing in an automated values in GridSearchCV. For grid search, we need to set up a parameter grid for the values of k that are of interest. Here we use the value of k between 1 and 31. The object of grid CV is instantiated with k-nearest neighbor regressor along with the parameter grid and number of cross-validation folds equal to 10.

The grid search is performed by calling the fit method with training feature matrix and labels as argument.

(Refer Slide Time: 09:02)



```
gs.best_estimator_
```

```
Pipeline(steps=[('scaler', MinMaxScaler()),
                ('knn', KNeighborsRegressor(n_neighbors=6))])
```

After the model is trained, the best estimator can be obtained by accessing `best_estimator_` member variable of `GridSearchCV` object.

> In this case, we found the best KNN regressor to be 6-NN regressor.

Let's evaluate the best estimator on the test set.

```
[16] pred = gs.best_estimator_.predict(X_test) #make prediction on test set
     error = mean_squared_error(y_test, pred, squared=False)
     print('RMSE value for k is: ' , error)
```

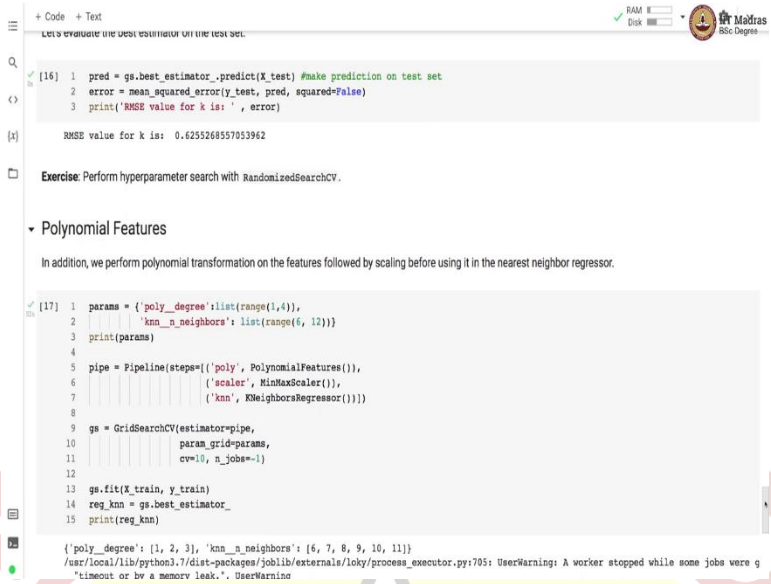
```
RMSE value for k is:  0.6255268557053962
```

Exercise: Perform hyperparameter search with `RandomizedSearchCV`.

Polynomial Features

In addition, we perform polynomial transformation on the features followed by scaling before using it in the nearest neighbor regressor.

```
[17] params = {'poly__degree':list(range(1,4)),
               'knn__n_neighbors': list(range(6, 12))}
     print(params)

     pipe = Pipeline(steps=[('poly', PolynomialFeatures()),
                            ('scaler', MinMaxScaler()),
```



Let's evaluate the best estimator on the test set.

```
[16]
pred = gs.best_estimator_.predict(X_test) #make prediction on test set
error = mean_squared_error(y_test, pred, squared=False)
print('RMSE value for k is: ' , error)
```

```
RMSE value for k is:  0.6255268557053962
```

Exercise: Perform hyperparameter search with `RandomizedSearchCV`.

Polynomial Features

In addition, we perform polynomial transformation on the features followed by scaling before using it in the nearest neighbor regressor.

```
[17]
params = {'poly__degree':list(range(1,4)),
          'knn__n_neighbors': list(range(6, 12))}
print(params)

pipe = Pipeline(steps=[('poly', PolynomialFeatures()),
                       ('scaler', MinMaxScaler()),
                       ('knn', KNeighborsRegressor())])

gs = GridSearchCV(estimator=pipe,
                  param_grid=params,
                  cv=10, n_jobs=-1)

gs.fit(X_train, y_train)
reg_knn = gs.best_estimator_
print(reg_knn)
```

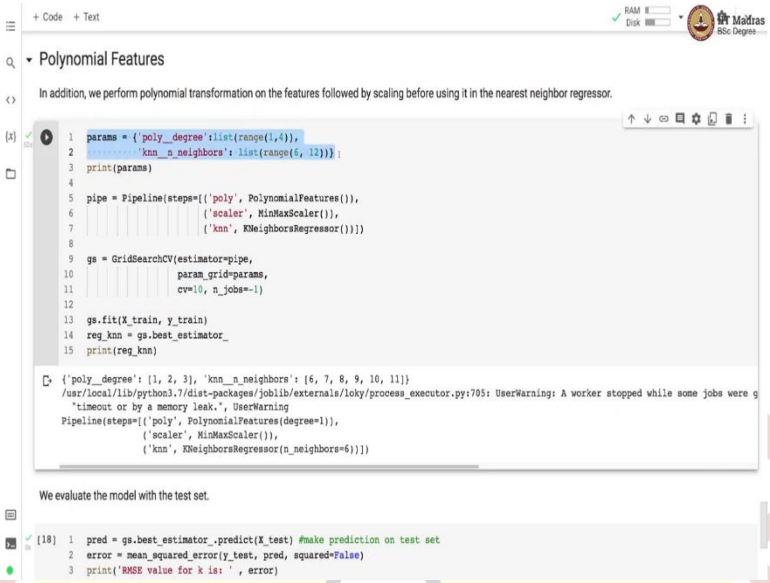
```
{'poly__degree': [1, 2, 3], 'knn__n_neighbors': [6, 7, 8, 9, 10, 11]}
/usr/local/lib/python3.7/dist-packages/joblib/externals/loky/process_executor.py:705: UserWarning: A worker stopped while some jobs were g
  "timeout or by a memory leak.", UserWarning
```

After the model is trained, the best estimator can be obtained by accessing the best estimator _ member variable of GridSearchCV object which is gs. In this case, we found the best gain and regressor to be 6-NN regressor, so number of neighbors that we found to be a CV the optimal value of that is 6.

Let us evaluate the best estimator on the test set and RMSE value here is 0.62. Now comparing this with the RMSE value earlier which was 0.67 we indeed obtain a smaller RMSE value with 6-NN regressor.

So, as an exercise, I would like you to perform hyper-parameter search with RandomizedSearchCV and it will also result into similar RMSE value. If you of course allow it to run for enough amount of time with enough resources. So, in practice, you have to use either GridSearchCV or RandomizedSearchCV to perform hyper-parameter search in K-NN..

(Refer Slide Time: 10:19)



Polynomial Features

In addition, we perform polynomial transformation on the features followed by scaling before using it in the nearest neighbor regressor.

```
params = {'poly__degree':list(range(1,4)),
          'knn__n_neighbors': list(range(6, 12))}
print(params)

pipe = Pipeline(steps=[('poly', PolynomialFeatures()),
                       ('scaler', MinMaxScaler()),
                       ('knn', KNeighborsRegressor())])

gs = GridSearchCV(estimator=pipe,
                  param_grid=params,
                  cv=10, n_jobs=-1)

gs.fit(X_train, y_train)
reg_knn = gs.best_estimator_
print(reg_knn)
```

```
{'poly__degree': [1, 2, 3], 'knn__n_neighbors': [6, 7, 8, 9, 10, 11]}
/usr/local/lib/python3.7/dist-packages/joblib/externals/loky/process_executor.py:705: UserWarning: A worker stopped while some jobs were g
  "timeout or by a memory leak.", UserWarning
Pipeline(steps=[('poly', PolynomialFeatures(degree=1)),
                ('scaler', MinMaxScaler()),
                ('knn', KNeighborsRegressor(n_neighbors=6))])
```

We evaluate the model with the test set.

```
[18]
pred = gs.best_estimator_.predict(X_test) #make prediction on test set
error = mean_squared_error(y_test, pred, squared=False)
print('RMSE value for k is: ' , error)
```

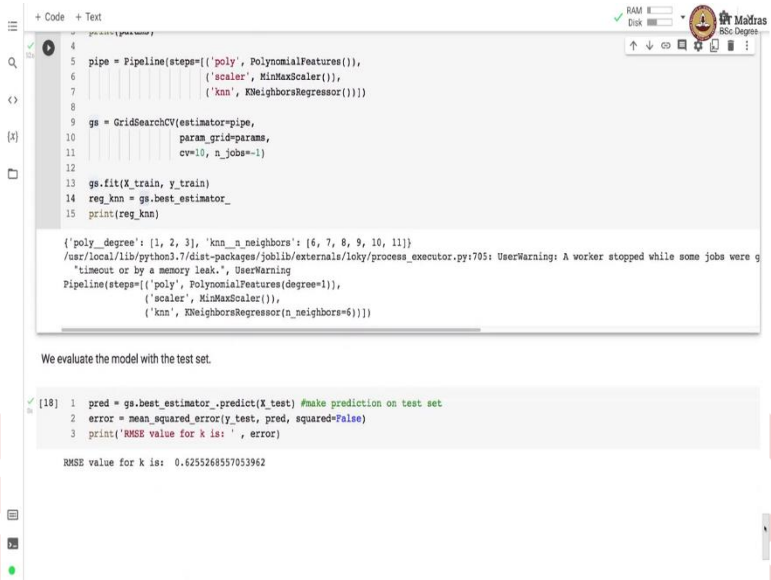
So, in addition, we perform polynomial transformation on the features followed by scaling before using it in the nearest neighbor regressor. So, here we set up a pipeline object with PolynomialFeatures followed by scaling, followed by the KNeighborsRegressor estimator. Here, we are setting, we want to basically search for the best value of the degree in PolynomialFeatures and number of neighbors in K-NN.

And we set the parameter grid for the polynomial degree in the range 1 to 4 and for nearest neighbor in the range 6 to 12. We performed the GridSearchCV on the pipeline object which contains a polynomial transformation followed by the k nearest neighbor regressor.

Here we set the parameter grid to the grid that is defined over here and number of cross-validation folds is set to 10. We perform the GridSearchCV by calling the fit method and passing the training feature matrix and the training label vector. Once the GridSearchCV is performed, we obtain the best estimator by accessing the member variable which is best _estimator _member variable of the GridSearchCV object which is gs in this case.

And you can see that the grid search found the polynomial or transformation with degree 1 to be optimal, along with the number of neighbors equal to 6.

(Refer Slide Time: 12:02)



```
pipe = Pipeline(steps=[('poly', PolynomialFeatures()),
                       ('scaler', MinMaxScaler()),
                       ('knn', KNeighborsRegressor())])

gs = GridSearchCV(estimator=pipe,
                  param_grid=params,
                  cv=10, n_jobs=-1)

gs.fit(X_train, y_train)
reg_knn = gs.best_estimator_
print(reg_knn)
```

```
{'poly__degree': [1, 2, 3], 'knn__n_neighbors': [6, 7, 8, 9, 10, 11]}
/usr/local/lib/python3.7/dist-packages/joblib/externals/loky/process_executor.py:705: UserWarning: A worker stopped while some jobs were g
  "timeout or by a memory leak.", UserWarning
Pipeline(steps=[('poly', PolynomialFeatures(degree=1)),
                ('scaler', MinMaxScaler()),
                ('knn', KNeighborsRegressor(n_neighbors=6))])
```

We evaluate the model with the test set.

```
[18]
pred = gs.best_estimator_.predict(X_test) #make prediction on test set
error = mean_squared_error(y_test, pred, squared=False)
print('RMSE value for k is: ' , error)
```

```
RMSE value for k is:  0.6255268557053962
```

We evaluate the model with the test set and we obtain comparable accuracy of 0.62 with the earlier grid search that we performed without polynomial regression. And a kind of equivalent because we found a polynomial transformation with degree equal to 1 to be optimal. In this video, we studied how to use K-NN in regression, we also discussed how to find the optimal value of k through GridSearchCV.