

# Model Evaluation and Refinement

Estimated time needed: **30** minutes

## Objectives

After completing this lab you will be able to:

- Evaluate and refine prediction models

## Table of Contents

- [Model Evaluation](#)
- [Over-fitting, Under-fitting and Model Selection](#)
- [Ridge Regression](#)
- [Grid Search](#)

If you are running the lab in your browser in Skills Network lab, so need to install the libraries using piplite.

```
In [1]: #you are running the lab in your browser, so we will install the libraries u  
import piplite  
await piplite.install(['pandas'])  
await piplite.install(['matplotlib'])  
await piplite.install(['scipy'])  
await piplite.install(['scikit-learn'])  
await piplite.install(['seaborn'])
```

If you run the lab locally using Anaconda, you can load the correct library and versions by uncommenting the following:

```
In [ ]: #If you run the lab locally using Anaconda, you can load the correct library  
#install specific version of libraries used in lab  
#! mamba install pandas==1.3.3-y  
#! mamba install numpy=1.21.2-y  
#! mamba install sklearn=0.20.1-y
```

Import libraries:

```
In [2]: import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
import warnings  
warnings.filterwarnings('ignore')
```

This function will download the dataset into your browser

```
In [3]: #This function will download the dataset into your browser

from pyodide.http import pyfetch

async def download(url, filename):
    response = await pyfetch(url)
    if response.status == 200:
        with open(filename, "wb") as f:
            f.write(await response.bytes())
```

This dataset was hosted on IBM Cloud object. Click [HERE](#) for free storage.

you will need to download the dataset; using the 'download()' function.

```
In [4]: #you will need to download the dataset;
await download('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.
```

Load the data and store it in dataframe df:

```
In [5]: df = pd.read_csv("module_5_auto.csv", header=0)
```

```
In [6]: df.head()
```

```
Out[6]:
```

|   | Unnamed: 0.1 | Unnamed: 0 | symboling | normalized-losses | make        | aspiration | num-of-doors | body-style  |
|---|--------------|------------|-----------|-------------------|-------------|------------|--------------|-------------|
| 0 | 0            | 0          | 3         | 122               | alfa-romero | std        | two          | convertible |
| 1 | 1            | 1          | 3         | 122               | alfa-romero | std        | two          | convertible |
| 2 | 2            | 2          | 1         | 122               | alfa-romero | std        | two          | hatchback   |
| 3 | 3            | 3          | 2         | 164               | audi        | std        | four         | sedan       |
| 4 | 4            | 4          | 2         | 164               | audi        | std        | four         | sedan       |

5 rows x 31 columns

First, let's only use numeric data:

```
In [7]: df=df._get_numeric_data()
df.head()
```

```
Out[7]:
```

|   | Unnamed: 0.1 | Unnamed: 0 | symboling | normalized-losses | wheel-base | length   | width    | height |
|---|--------------|------------|-----------|-------------------|------------|----------|----------|--------|
| 0 | 0            | 0          | 3         | 122               | 88.6       | 0.811148 | 0.890278 | 48.8   |
| 1 | 1            | 1          | 3         | 122               | 88.6       | 0.811148 | 0.890278 | 48.8   |
| 2 | 2            | 2          | 1         | 122               | 94.5       | 0.822681 | 0.909722 | 52.4   |
| 3 | 3            | 3          | 2         | 164               | 99.8       | 0.848630 | 0.919444 | 54.3   |
| 4 | 4            | 4          | 2         | 164               | 99.4       | 0.848630 | 0.922222 | 54.3   |

5 rows x 21 columns

Let's remove the columns 'Unnamed:0.1' and 'Unnamed:0' since they do not provide any value to the models.

```
In [8]: df.drop(['Unnamed: 0.1', 'Unnamed: 0'], axis=1, inplace=True)

# Let's take a look at the updated DataFrame
df.head()
```

```
Out[8]:
```

|   | symboling | normalized-losses | wheel-base | length   | width    | height | curb-weight | engine-size | bore |
|---|-----------|-------------------|------------|----------|----------|--------|-------------|-------------|------|
| 0 | 3         | 122               | 88.6       | 0.811148 | 0.890278 | 48.8   | 2548        | 130         | 3.47 |
| 1 | 3         | 122               | 88.6       | 0.811148 | 0.890278 | 48.8   | 2548        | 130         | 3.47 |
| 2 | 1         | 122               | 94.5       | 0.822681 | 0.909722 | 52.4   | 2823        | 152         | 2.68 |
| 3 | 2         | 164               | 99.8       | 0.848630 | 0.919444 | 54.3   | 2337        | 109         | 3.19 |
| 4 | 2         | 164               | 99.4       | 0.848630 | 0.922222 | 54.3   | 2824        | 136         | 3.19 |

Libraries for plotting:

```
In [13]: await piplite.install(['ipywidgets'])
```

```
In [14]: from ipywidgets import interact, interactive, fixed, interact_manual
```

## Functions for Plotting

```
In [15]: def DistributionPlot(RedFunction, BlueFunction, RedName, BlueName, Title):
width = 12
height = 10
plt.figure(figsize=(width, height))

ax1 = sns.kdeplot(RedFunction, color="r", label=RedName)
ax2 = sns.kdeplot(BlueFunction, color="b", label=BlueName, ax=ax1)

plt.title(Title)
plt.xlabel('Price (in dollars)')
plt.ylabel('Proportion of Cars')
plt.show()
plt.close()
```

```
In [16]: def PollyPlot(xtrain, xtest, y_train, y_test, lr, poly_transform):
width = 12
```

```

height = 10
plt.figure(figsize=(width, height))

#training data
#testing data
# lr: linear regression object
#poly_transform: polynomial transformation object

xmax=max([xtrain.values.max(), xtest.values.max()])

xmin=min([xtrain.values.min(), xtest.values.min()])

x=np.arange(xmin, xmax, 0.1)

plt.plot(xtrain, y_train, 'ro', label='Training Data')
plt.plot(xtest, y_test, 'go', label='Test Data')
plt.plot(x, lr.predict(poly_transform.fit_transform(x.reshape(-1, 1))), 1
plt.ylim([-10000, 60000])
plt.ylabel('Price')
plt.legend()

```

## Part 1: Training and Testing

An important step in testing your model is to split your data into training and testing data. We will place the target data **price** in a separate dataframe **y\_data**:

```
In [17]: y_data = df['price']
```

Drop price data in dataframe **x\_data**:

```
In [18]: x_data=df.drop('price',axis=1)
```

Now, we randomly split our data into training and testing data using the function **train\_test\_split**.

```
In [19]: from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.1)

print("number of test samples :", x_test.shape[0])
print("number of training samples:",x_train.shape[0])
```

```

number of test samples : 21
number of training samples: 180

```

The **test\_size** parameter sets the proportion of data that is split into the testing set. In the above, the testing set is 10% of the total dataset.

**Question #1):**

Use the function "train\_test\_split" to split up the dataset such that 40% of the data samples will be utilized for testing. Set the parameter "random\_state" equal to zero. The output of the function should be the following: "x\_train1", "x\_test1", "y\_train1" and "y\_test1".

```
In [20]: # Write your code below and press Shift+Enter to execute
x_train1, x_test1, y_train1, y_test1 = train_test_split(x_data, y_data, test_
print("number of test samples :", x_test1.shape[0])
print("number of training samples:", x_train1.shape[0])
```

number of test samples : 81  
number of training samples: 120

► Click here for the solution

Let's import **LinearRegression** from the module **linear\_model**.

```
In [21]: from sklearn.linear_model import LinearRegression
```

We create a Linear Regression object:

```
In [22]: lre=LinearRegression()
```

We fit the model using the feature "horsepower":

```
In [23]: lre.fit(x_train[['horsepower']], y_train)
```

```
Out[23]: ▼ LinearRegression
LinearRegression()
```

Let's calculate the  $R^2$  on the test data:

```
In [24]: lre.score(x_test[['horsepower']], y_test)
```

```
Out[24]: 0.3635875575078824
```

We can see the  $R^2$  is much smaller using the test data compared to the training data.

```
In [25]: lre.score(x_train[['horsepower']], y_train)
```

```
Out[25]: 0.6619724197515103
```

## Question #2):

Find the  $R^2$  on the test data using 40% of the dataset for testing.

```
In [26]: # Write your code below and press Shift+Enter to execute
x_train1, x_test1, y_train1, y_test1 = train_test_split(x_data, y_data, test_
lre.fit(x_train1[['horsepower']], y_train1)
lre.score(x_test1[['horsepower']], y_test1)
```

Out [26]: 0.7139364665406973

► [Click here for the solution](#)

Sometimes you do not have sufficient testing data; as a result, you may want to perform cross-validation. Let's go over several methods that you can use for cross-validation.

## Cross-Validation Score

Let's import `cross_val_score` from the module `model_selection`.

```
In [27]: from sklearn.model_selection import cross_val_score
```

We input the object, the feature ("horsepower"), and the target data (y\_data). The parameter 'cv' determines the number of folds. In this case, it is 4.

```
In [28]: Rcross = cross_val_score(lre, x_data[['horsepower']], y_data, cv=4)
```

The default scoring is  $R^2$ . Each element in the array has the average  $R^2$  value for the fold:

```
In [29]: Rcross
```

```
Out [29]: array([0.7746232 , 0.51716687, 0.74785353, 0.04839605])
```

We can calculate the average and standard deviation of our estimate:

```
In [30]: print("The mean of the folds are", Rcross.mean(), "and the standard deviation
```

The mean of the folds are 0.5220099150421197 and the standard deviation is 0.29118394447560203

We can use negative squared error as a score by setting the parameter 'scoring' metric to 'neg\_mean\_squared\_error'.

```
In [31]: -1 * cross_val_score(lre, x_data[['horsepower']], y_data, cv=4, scoring='neg_mea
```

```
Out [31]: array([20254142.84026702, 43745493.26505171, 12539630.34014929,
17561927.72247586])
```

### Question #3):

Calculate the average  $R^2$  using two folds, then find the average  $R^2$  for the second fold utilizing the "horsepower" feature:

```
In [32]: # Write your code below and press Shift+Enter to execute
Rcross2 = cross_val_score(lre, x_data[['horsepower']], y_data, cv=2)
Rcross2.mean()
```

```
Out [32]: 0.5166761697127429
```

► [Click here for the solution](#)

You can also use the function 'cross\_val\_predict' to predict the output. The function splits up the data into the specified number of folds, with one fold for testing and the other folds are used for training. First, import the function:

```
In [33]: from sklearn.model_selection import cross_val_predict
```

We input the object, the feature **"horsepower"**, and the target data **y\_data**. The parameter 'cv' determines the number of folds. In this case, it is 4. We can produce an output:

```
In [34]: yhat = cross_val_predict(lre, x_data[['horsepower']], y_data, cv=4)
yhat[0:5]
```

```
Out[34]: array([14141.63807508, 14141.63807508, 20814.29423473, 12745.03562306,
14762.35027598])
```

## Part 2: Overfitting, Underfitting and Model Selection

It turns out that the test data, sometimes referred to as the "out of sample data", is a much better measure of how well your model performs in the real world. One reason for this is overfitting.

Let's go over some examples. It turns out these differences are more apparent in Multiple Linear Regression and Polynomial Regression so we will explore overfitting in that context.

Let's create Multiple Linear Regression objects and train the model using **'horsepower'**, **'curb-weight'**, **'engine-size'** and **'highway-mpg'** as features.

```
In [35]: lr = LinearRegression()
lr.fit(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],
```

```
Out[35]: ▼ LinearRegression
LinearRegression()
```

Prediction using training data:

```
In [36]: yhat_train = lr.predict(x_train[['horsepower', 'curb-weight', 'engine-size',
yhat_train[0:5]
```

```
Out[36]: array([ 7426.6731551 , 28323.75090803, 14213.38819709,  4052.34146983,
34500.19124244])
```

Prediction using test data:

```
In [37]: yhat_test = lr.predict(x_test[['horsepower', 'curb-weight', 'engine-size', 'r
yhat_test[0:5]
```

```
Out[37]: array([11349.35089149,  5884.11059106, 11208.6928275 ,  6641.07786278,
15565.79920282])
```

Let's perform some model evaluation using our training and testing data separately. First, we import the seaborn and matplotlib library for plotting.

```
In [38]: import matplotlib.pyplot as plt
import matplotlib inline
import seaborn as sns
```

Let's examine the distribution of the predicted values of the training data.

```
In [39]: Title = 'Distribution Plot of Predicted Value Using Training Data vs Training Data Distribution
DistributionPlot(y_train, yhat_train, "Actual Values (Train)", "Predicted Values (Train)")
```

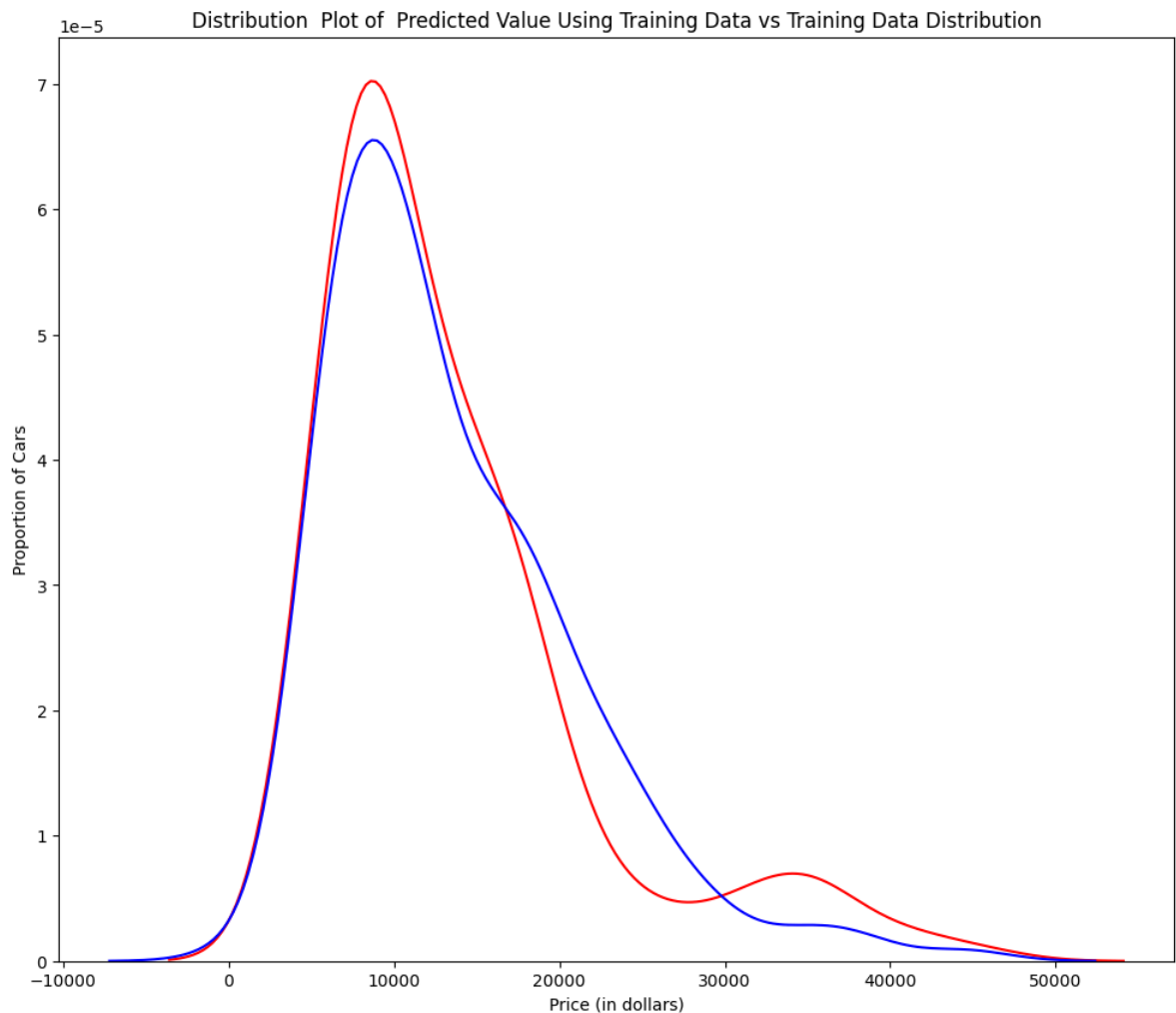


Figure 1: Plot of predicted values using the training data compared to the actual values of the training data.

So far, the model seems to be doing well in learning from the training dataset. But what happens when the model encounters new data from the testing dataset? When the model generates new values from the test data, we see the distribution of the predicted values is much different from the actual target values.

```
In [40]: Title='Distribution Plot of Predicted Value Using Test Data vs Data Distribution
DistributionPlot(y_test,yhat_test,"Actual Values (Test)","Predicted Values (Test)")
```



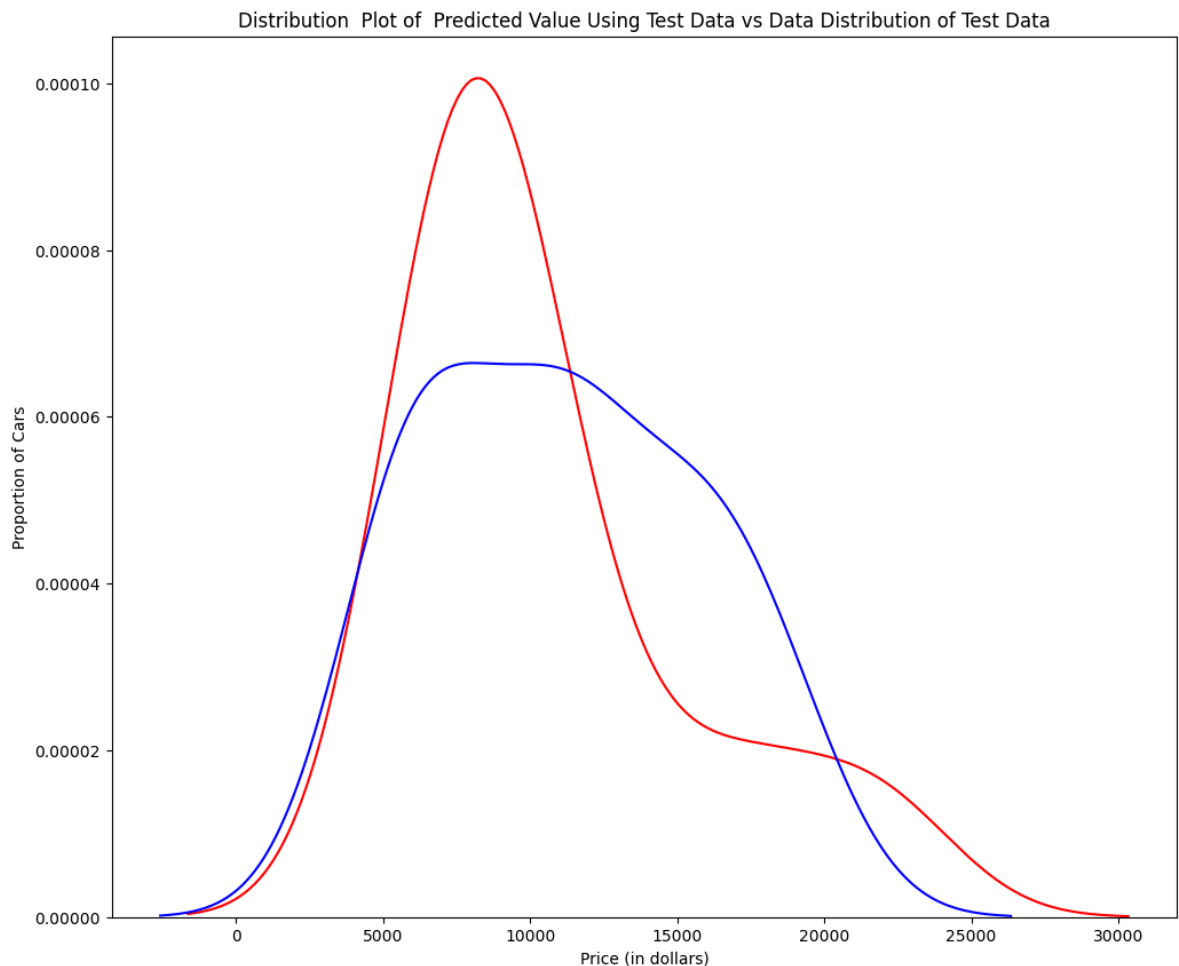


Figure 2: Plot of predicted value using the test data compared to the actual values of the test data.

Comparing Figure 1 and Figure 2, it is evident that the distribution of the test data in Figure 1 is much better at fitting the data. This difference in Figure 2 is apparent in the range of 5,000 to 15,000. This is where the shape of the distribution is extremely different. Let's see if polynomial regression also exhibits a drop in the prediction accuracy when analysing the test dataset.

```
In [41]: from sklearn.preprocessing import PolynomialFeatures
```

## Overfitting

Overfitting occurs when the model fits the noise, but not the underlying process. Therefore, when testing your model using the test set, your model does not perform as well since it is modelling noise, not the underlying process that generated the relationship. Let's create a degree 5 polynomial model.

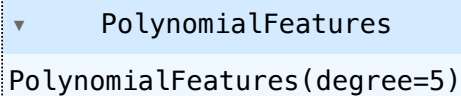
Let's use 55 percent of the data for training and the rest for testing:

```
In [42]: x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.45)
```

We will perform a degree 5 polynomial transformation on the feature 'horsepower'.

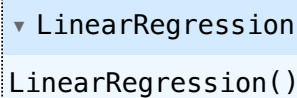
```
In [43]: pr = PolynomialFeatures(degree=5)
x_train_pr = pr.fit_transform(x_train[['horsepower']])
```

```
x_test_pr = pr.fit_transform(x_test[['horsepower']])
pr
```

Out [43]:  PolynomialFeatures  
PolynomialFeatures(degree=5)

Now, let's create a Linear Regression model "poly" and train it.

```
In [44]: poly = LinearRegression()
poly.fit(x_train_pr, y_train)
```

Out [44]:  LinearRegression  
LinearRegression()

We can see the output of our model using the method "predict." We assign the values to "yhat".

```
In [45]: yhat = poly.predict(x_test_pr)
yhat[0:5]
```

Out [45]: array([ 6728.58641321, 7307.91998787, 12213.73753589, 18893.37919224,  
 19996.10612156])

Let's take the first five predicted values and compare it to the actual targets.

```
In [46]: print("Predicted values:", yhat[0:4])
print("True values:", y_test[0:4].values)
```

Predicted values: [ 6728.58641321 7307.91998787 12213.73753589 18893.37919224]  
True values: [ 6295. 10698. 13860. 13499.]

We will use the function "PollyPlot" that we defined at the beginning of the lab to display the training data, testing data, and the predicted function.

```
In [47]: PollyPlot(x_train['horsepower'], x_test['horsepower'], y_train, y_test, poly,
```

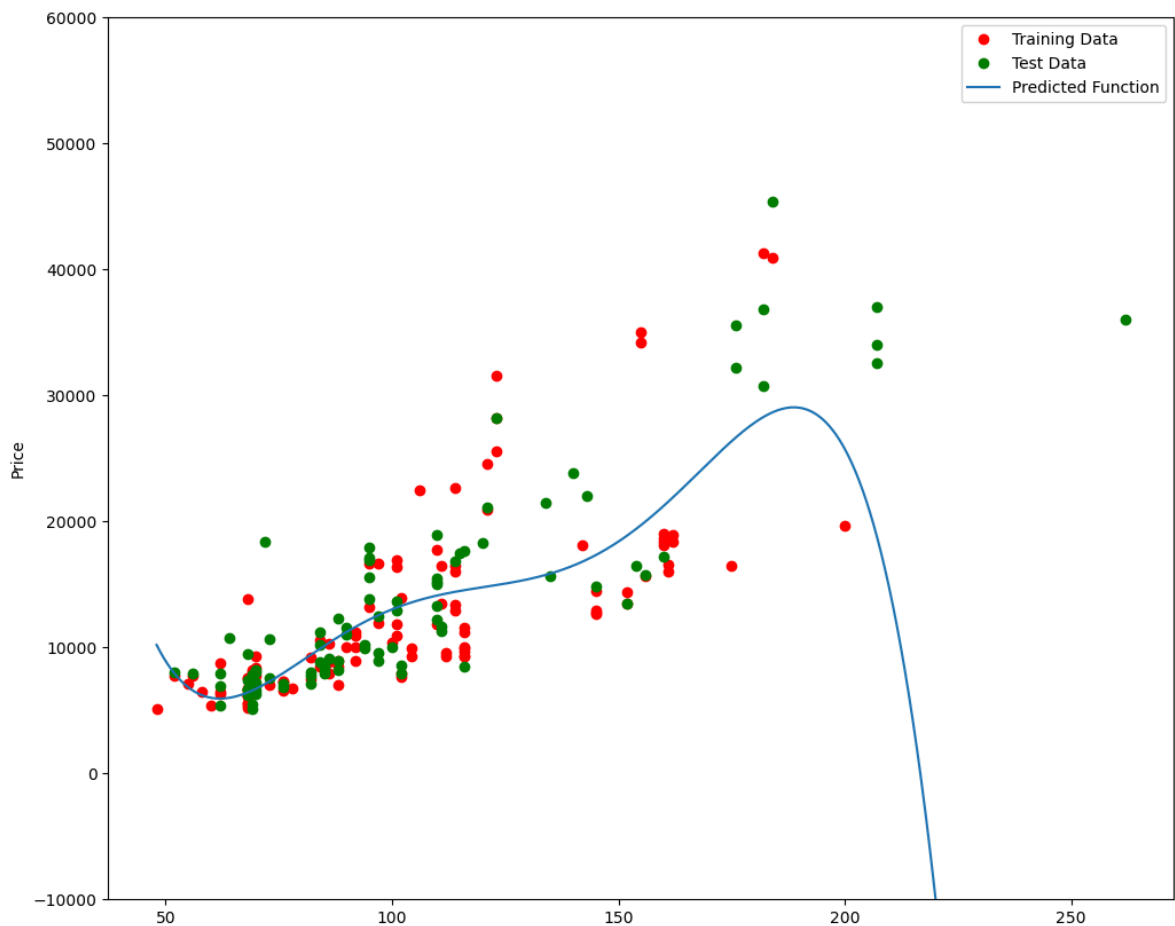


Figure 3: A polynomial regression model where red dots represent training data, green dots represent test data, and the blue line represents the model prediction.

We see that the estimated function appears to track the data but around 200 horsepower, the function begins to diverge from the data points.

$R^2$  of the training data:

```
In [48]: poly.score(x_train_pr, y_train)
```

```
Out[48]: 0.5567716897754004
```

$R^2$  of the test data:

```
In [49]: poly.score(x_test_pr, y_test)
```

```
Out[49]: -29.87099623387278
```

We see the  $R^2$  for the training data is 0.5567 while the  $R^2$  on the test data was -29.87. The lower the  $R^2$ , the worse the model. A negative  $R^2$  is a sign of overfitting.

Let's see how the  $R^2$  changes on the test data for different order polynomials and then plot the results:

```
In [50]: Rsqu_test = []
order = [1, 2, 3, 4]
for n in order:
    pr = PolynomialFeatures(degree=n)
```

```

x_train_pr = pr.fit_transform(x_train[['horsepower']])
x_test_pr = pr.fit_transform(x_test[['horsepower']])

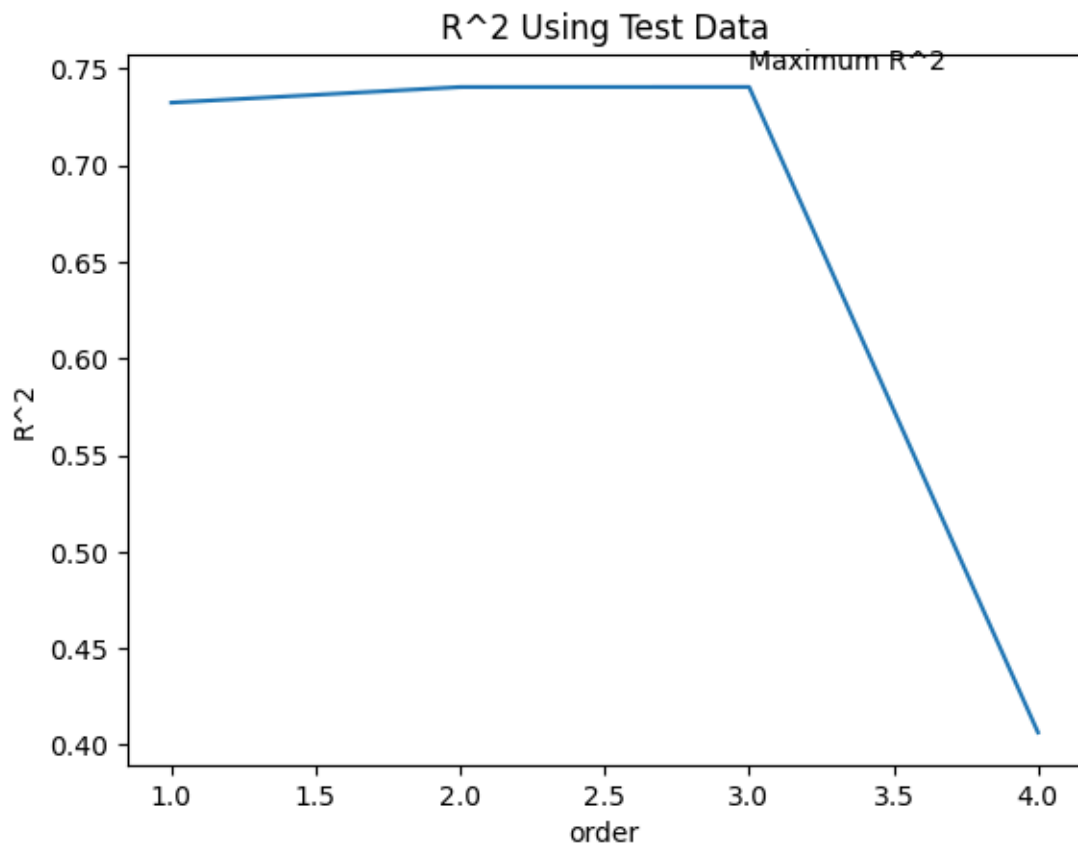
lr.fit(x_train_pr, y_train)

Rsqu_test.append(lr.score(x_test_pr, y_test))

plt.plot(order, Rsqu_test)
plt.xlabel('order')
plt.ylabel('R^2')
plt.title('R^2 Using Test Data')
plt.text(3, 0.75, 'Maximum R^2 ')

```

Out[50]: Text(3, 0.75, 'Maximum R^2 ')



We see the  $R^2$  gradually increases until an order three polynomial is used. Then, the  $R^2$  dramatically decreases at an order four polynomial.

The following function will be used in the next section. Please run the cell below.

```

In [51]: def f(order, test_data):
    x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_
    pr = PolynomialFeatures(degree=order)
    x_train_pr = pr.fit_transform(x_train[['horsepower']])
    x_test_pr = pr.fit_transform(x_test[['horsepower']])
    poly = LinearRegression()
    poly.fit(x_train_pr, y_train)
    PollyPlot(x_train['horsepower'], x_test['horsepower'], y_train, y_test, p

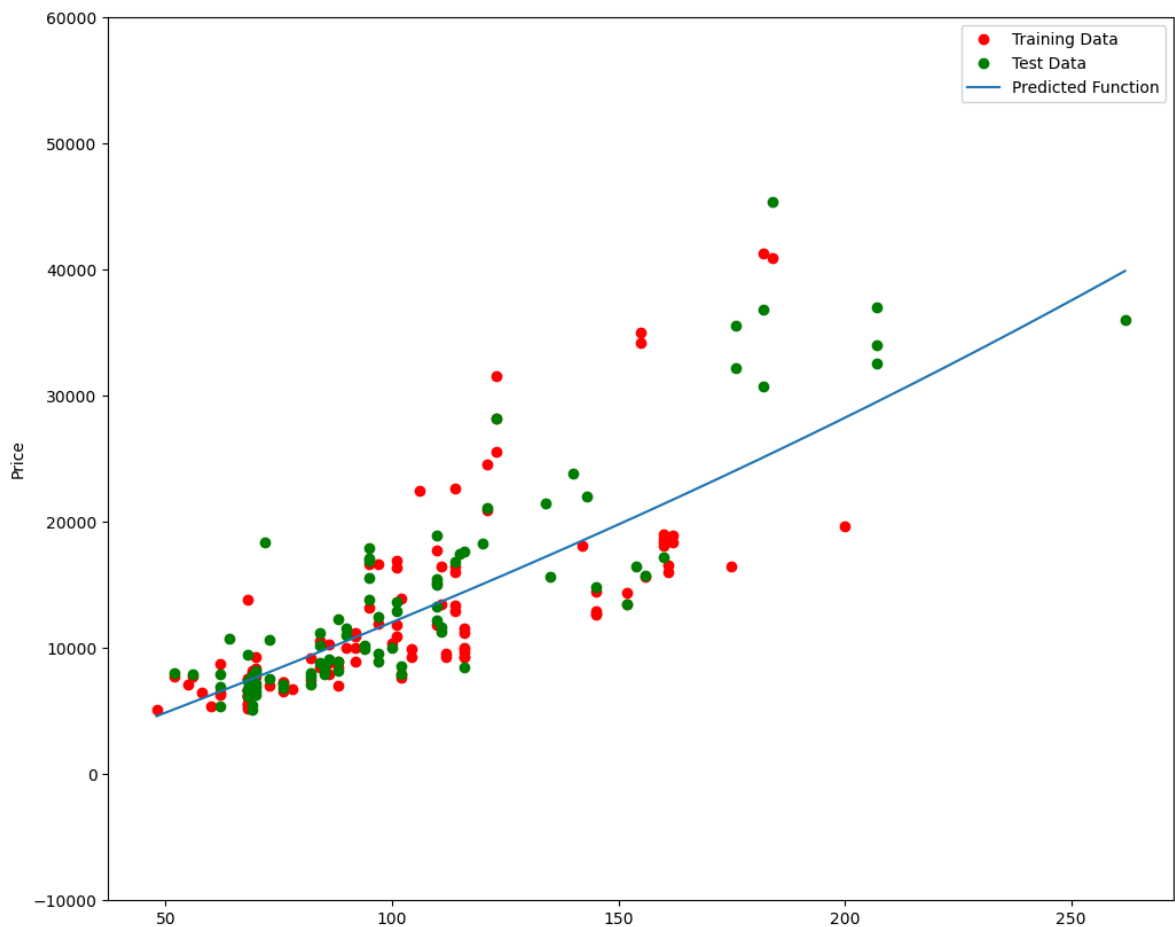
```

The following interface allows you to experiment with different polynomial orders and different amounts of data.

```
In [52]: interact(f, order=(0, 6, 1), test_data=(0.05, 0.95, 0.05))
```

```
interactive(children=(IntSlider(value=3, description='order', max=6), FloatSlider(value=0.45, description='tes...
```

```
Out[52]: <function __main__.f(order, test_data)>
```



## Question #4a):

We can perform polynomial transformations with more than one feature. Create a "PolynomialFeatures" object "pr1" of degree two.

```
In [53]: # Write your code below and press Shift+Enter to execute
pr1=PolynomialFeatures(degree=2)
```

► [Click here for the solution](#)

## Question #4b):

Transform the training and testing samples for the features 'horsepower', 'curb-weight', 'engine-size' and 'highway-mpg'. Hint: use the method "fit\_transform".

```
In [54]: # Write your code below and press Shift+Enter to execute
x_train_pr1=pr1.fit_transform(x_train[['horsepower', 'curb-weight', 'engine-s
x_test_pr1=pr1.fit_transform(x_test[['horsepower', 'curb-weight', 'engine-siz
```

► [Click here for the solution](#)

## Question #4c):

**How many dimensions does the new feature have? Hint: use the attribute "shape".**

```
In [55]: # Write your code below and press Shift+Enter to execute
x_train_pr1.shape
```

Out[55]: (110, 15)

► [Click here for the solution](#)

## Question #4d):

**Create a linear regression model "poly1". Train the object using the method "fit" using the polynomial features.**

```
In [56]: # Write your code below and press Shift+Enter to execute
poly1=LinearRegression().fit(x_train_pr1,y_train)
```

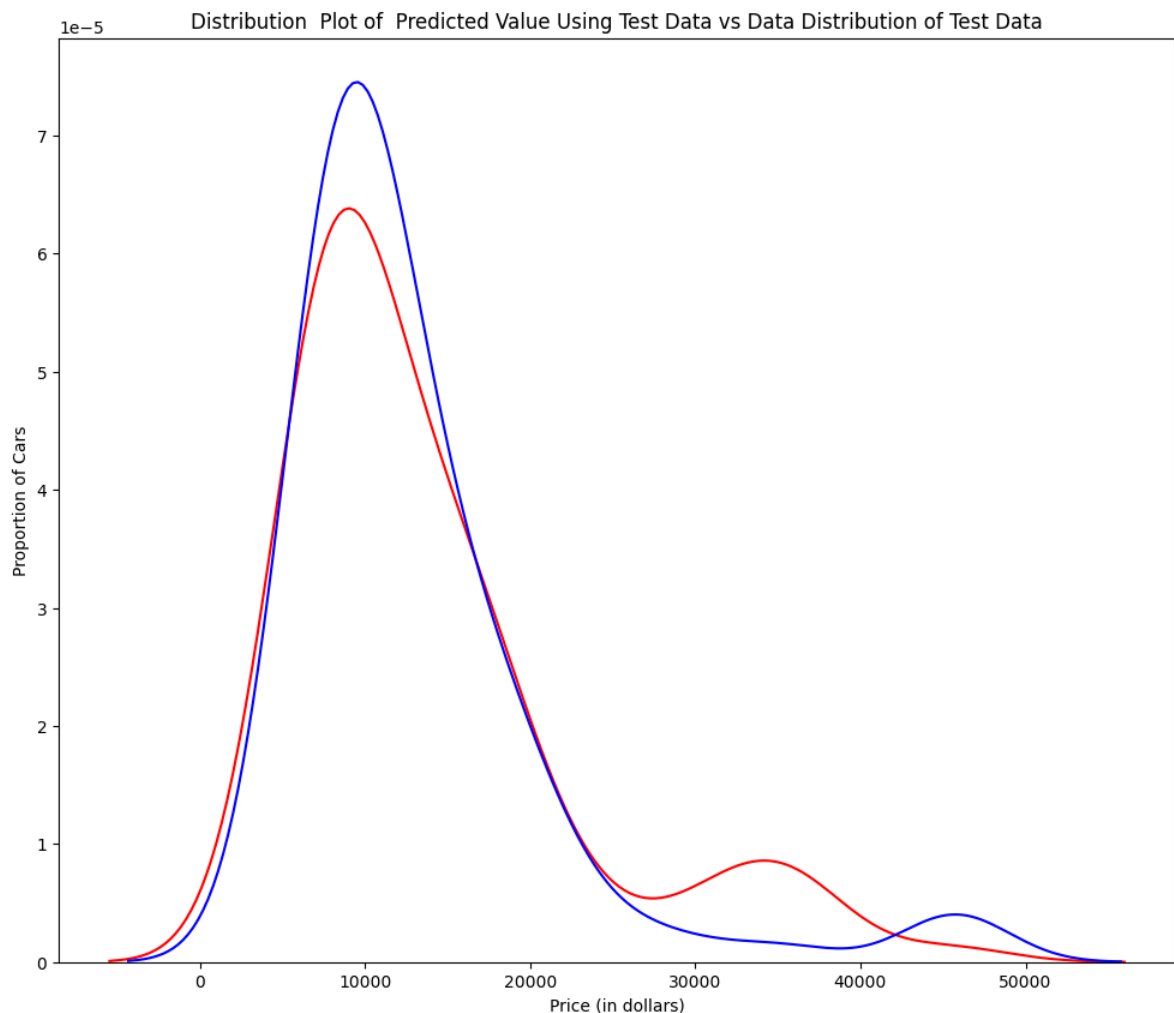
► [Click here for the solution](#)

## Question #4e):

**Use the method "predict" to predict an output on the polynomial features, then use the function "DistributionPlot" to display the distribution of the predicted test output vs. the actual test data.**

```
In [58]: # Write your code below and press Shift+Enter to execute
yhat_test1=poly1.predict(x_test_pr1)

Title='Distribution Plot of Predicted Value Using Test Data vs Data Distrib
DistributionPlot(y_test, yhat_test1, "Actual Values (Test)", "Predicted Value
```



► [Click here for the solution](#)

## Question #4f):

Using the distribution plot above, describe (in words) the two regions where the predicted prices are less accurate than the actual prices.

In [ ]: *# Write your code below and press Shift+Enter to execute*

► [Click here for the solution](#)

## Part 3: Ridge Regression

In this section, we will review Ridge Regression and see how the parameter alpha changes the model. Just a note, here our test data will be used as validation data.

Let's perform a degree two polynomial transformation on our data.

```
In [59]: pr=PolynomialFeatures(degree=2)
x_train_pr=pr.fit_transform(x_train[['horsepower', 'curb-weight', 'engine-size']
x_test_pr=pr.fit_transform(x_test[['horsepower', 'curb-weight', 'engine-size']
```

Let's import **Ridge** from the module **linear models**.

```
In [60]: from sklearn.linear_model import Ridge
```

Let's create a Ridge regression object, setting the regularization parameter (alpha) to 0.1

```
In [61]: RigeModel=Ridge(alpha=1)
```

Like regular regression, you can fit the model using the method **fit**.

```
In [62]: RigeModel.fit(x_train_pr, y_train)
```

```
Out[62]: 

▼ Ridge
  Ridge(alpha=1)


```

Similarly, you can obtain a prediction:

```
In [63]: yhat = RigeModel.predict(x_test_pr)
```

Let's compare the first five predicted samples to our test set:

```
In [64]: print('predicted:', yhat[0:4])
print('test set :', y_test[0:4].values)
```

```
predicted: [ 6570.82441941  9636.24891471 20949.92322738 19403.60313255]
test set : [ 6295. 10698. 13860. 13499.]
```

We select the value of alpha that minimizes the test error. To do so, we can use a for loop. We have also created a progress bar to see how many iterations we have completed so far.

```
In [65]: from tqdm import tqdm

Rsqu_test = []
Rsqu_train = []
dummy1 = []
Alpha = 10 * np.array(range(0,1000))
pbar = tqdm(Alpha)

for alpha in pbar:
    RigeModel = Ridge(alpha=alpha)
    RigeModel.fit(x_train_pr, y_train)
    test_score, train_score = RigeModel.score(x_test_pr, y_test), RigeModel.s

    pbar.set_postfix({"Test Score": test_score, "Train Score": train_score})

    Rsqu_test.append(test_score)
    Rsqu_train.append(train_score)
```

```
100%|██████████| 1000/1000 [00:01<00:00, 570.45it/s, Test Score=0.564, Train
Score=0.859]
```

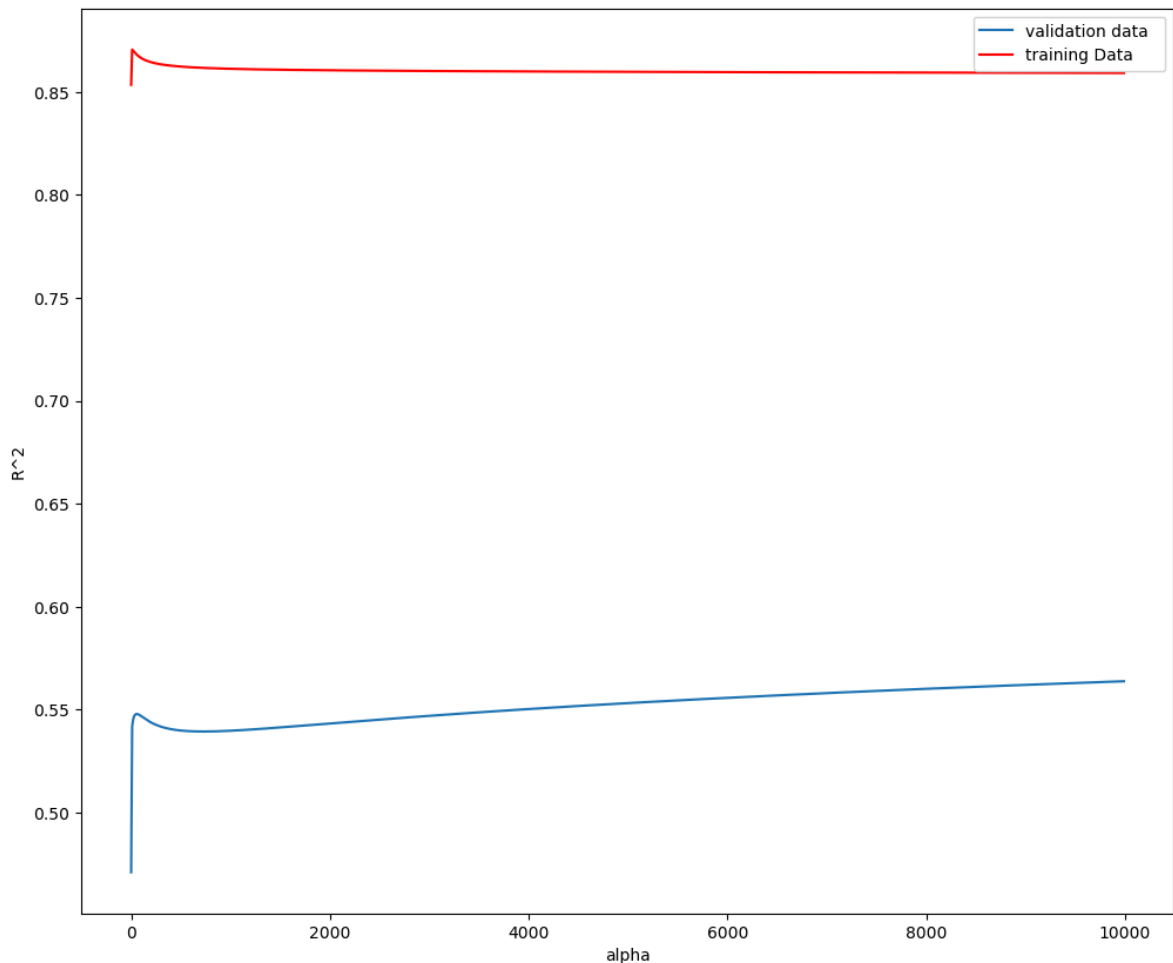
We can plot out the value of  $R^2$  for different alphas:

```
In [66]: width = 12
height = 10
plt.figure(figsize=(width, height))
```



```
plt.plot(Alpha,Rsqu_test, label='validation data ')
plt.plot(Alpha,Rsqu_train, 'r', label='training Data ')
plt.xlabel('alpha')
plt.ylabel('R^2')
plt.legend()
```

Out[66]: <matplotlib.legend.Legend at 0xcb6c660>



**Figure 4:** The blue line represents the  $R^2$  of the validation data, and the red line represents the  $R^2$  of the training data. The x-axis represents the different values of Alpha.

Here the model is built and tested on the same data, so the training and test data are the same.

The red line in Figure 4 represents the  $R^2$  of the training data. As alpha increases the  $R^2$  decreases. Therefore, as alpha increases, the model performs worse on the training data

The blue line represents the  $R^2$  on the validation data. As the value for alpha increases, the  $R^2$  increases and converges at a point.

**Question #5):**

Perform Ridge regression. Calculate the  $R^2$  using the polynomial features, use the training data to train the model and use the test data to test the model. The parameter alpha should be set to 10.

```
In [67]: # Write your code below and press Shift+Enter to execute
RidgeModel = Ridge(alpha=10)
RidgeModel.fit(x_train_pr, y_train)
RidgeModel.score(x_test_pr, y_test)
```

Out[67]: 0.5418576440208995

► [Click here for the solution](#)

## Part 4: Grid Search

The term alpha is a hyperparameter. Sklearn has the class **GridSearchCV** to make the process of finding the best hyperparameter simpler.

Let's import **GridSearchCV** from the module **model\_selection**.

```
In [68]: from sklearn.model_selection import GridSearchCV
```

We create a dictionary of parameter values:

```
In [69]: parameters1= [{'alpha': [0.001,0.1,1, 10, 100, 1000, 10000, 100000, 100000]}]
parameters1
```

Out[69]: [{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 100000]}]

Create a Ridge regression object:

```
In [70]: RR=Ridge()
RR
```

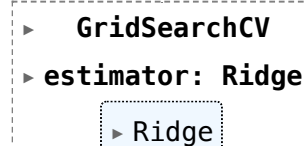
Out[70]:  Ridge()  
A small diagram showing a blue box with a downward arrow and the text 'Ridge' above a larger box containing 'Ridge()'.

Create a ridge grid search object:

```
In [71]: Grid1 = GridSearchCV(RR, parameters1,cv=4)
```

Fit the model:

```
In [72]: Grid1.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])
```

Out[72]:  GridSearchCV  
estimator: Ridge  
Ridge  
A diagram showing a dashed box containing 'GridSearchCV', 'estimator: Ridge', and a nested box with 'Ridge'.

The object finds the best parameter values on the validation data. We can obtain the estimator with the best parameters and assign it to the variable BestRR as follows:

```
In [73]: BestRR=Grid1.best_estimator_  
BestRR
```

```
Out[73]: ▼ Ridge  
Ridge(alpha=10000)
```

We now test our model on the test data:

```
In [74]: BestRR.score(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])
```

```
Out[74]: 0.8411649831036152
```

## Question #6):

Perform a grid search for the alpha parameter and the normalization parameter, then find the best values of the parameters:

```
In [75]: # Write your code below and press Shift+Enter to execute  
parameters2 = [{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 1000000]}]  
  
Grid2 = GridSearchCV(Ridge(), parameters2, cv=4)  
Grid2.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])  
best_alpha = Grid2.best_params_['alpha']  
best_ridge_model = Ridge(alpha=best_alpha)  
best_ridge_model.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])
```

```
Out[75]: ▼ Ridge  
Ridge(alpha=10000)
```

```
In [2]: pip install pandadoc-python-client
```

Collecting pandadoc-python-client

Downloading pandadoc\_python\_client-6.2.0-py3-none-any.whl.metadata (2.9 kB)

Requirement already satisfied: urllib3>=1.25.3 in /opt/anaconda3/lib/python3.12/site-packages (from pandadoc-python-client) (2.2.2)

Requirement already satisfied: python-dateutil in /opt/anaconda3/lib/python3.12/site-packages (from pandadoc-python-client) (2.9.0.post0)

Requirement already satisfied: six>=1.5 in /opt/anaconda3/lib/python3.12/site-packages (from python-dateutil->pandadoc-python-client) (1.16.0)

Downloading pandadoc\_python\_client-6.2.0-py3-none-any.whl (485 kB)

485.8/485.8 kB 1.3 MB/s eta 0:00:00a 0:00:01m

Installing collected packages: pandadoc-python-client

Successfully installed pandadoc-python-client-6.2.0

Note: you may need to restart the kernel to use updated packages.

```
In [1]: pip install pandoc
```

```
Collecting pandoc
  Downloading pandoc-2.3.tar.gz (33 kB)
  Preparing metadata (setup.py) ... done
Collecting plumbum (from pandoc)
  Downloading plumbum-1.8.3-py3-none-any.whl.metadata (10 kB)
Requirement already satisfied: ply in /opt/anaconda3/lib/python3.12/site-pack
ages (from pandoc) (3.11)
Downloading plumbum-1.8.3-py3-none-any.whl (127 kB)
_____ 127.6/127.6 kB 1.7 MB/s eta 0:00:
00 0:00:01
Building wheels for collected packages: pandoc
  Building wheel for pandoc (setup.py) ... done
  Created wheel for pandoc: filename=pandoc-2.3-py3-none-any.whl size=33263 s
ha256=131a0071c5f251e60b4654fb8dd63be9729f3fdf3cbfe3b8e924ede6eb42cbc4
  Stored in directory: /Users/nezam/Library/Caches/pip/wheels/9c/55/53/12fa87
f25207e4c1e358388add76ff41f5313a12ca20db830a
Successfully built pandoc
Installing collected packages: plumbum, pandoc
Successfully installed pandoc-2.3 plumbum-1.8.3
Note: you may need to restart the kernel to use updated packages.
```

► [Click here for the solution](#)

Thank you for completing this lab!

## Author

[Joseph Santarcangelo](#)

## Other Contributors

[Mahdi Noorian PhD](#)

Bahare Talayian

Eric Xiao

Steven Dong

Parizad

Hima Vasudevan

[Fiorella Wenver](#)

[Yi Yao.](#)

## Change Log

| Date (YYYY-MM-DD) | Version | Changed By | Change Description             |
|-------------------|---------|------------|--------------------------------|
| 2022-07-05        | 2.4     | Pooja      | Changed and added a new code   |
| 2020-10-30        | 2.3     | Lakshmi    | Changed URL of csv             |
| 2020-10-05        | 2.2     | Lakshmi    | Removed unused library imports |

| Date (YYYY-MM-DD) | Version | Changed By | Change Description                  |
|-------------------|---------|------------|-------------------------------------|
| 2020-09-14        | 2.1     | Lakshmi    | Made changes in OverFitting section |
| 2020-08-27        | 2.0     | Lavanya    | Moved lab to course repo in GitLab  |

---

© IBM Corporation 2020. All rights reserved.

In [ ]: