Model Evaluation and Refinement

January 23, 2024

1 Model Evaluation and Refinement

Estimated time needed: **30** minutes

Lab Component by Dhesika

1.1 Objectives

- Evaluate and refine prediction models

Table of Contents

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

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

[1]:
```
"import piplite\nawait piplite.install(['pandas'])\nawait
piplite.install(['matplotlib'])\nawait piplite.install(['scipy'])\nawait
piplite.install(['scikit-learn'])\nawait piplite.install(['seaborn'])"
```

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

[2]:
```
#If you run the lab locally using Anaconda, you can load the correct library␣
 ↪and versions by uncommenting the following:
#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:

[3]:
```
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

[4]:
```
#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())'''
```

[4]:
```
'from pyodide.http import pyfetch\n\nasync def download(url, filename):\n
response = await pyfetch(url)\n    if response.status == 200:\n        with
open(filename, "wb") as f:\n            f.write(await response.bytes())'
```

[5]:
```
#download the dataset;
#await download('https://cf-courses-data.s3.us.cloud-object-storage.appdomain.
↪cloud/IBMDeveloperSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/
↪module_5_auto.csv','module_5_auto.csv')
```

Load the data and store it in dataframe df:

[6]:
```
df = pd.read_csv("usedcars.csv", header=0)
```

[7]:
```
df.head()
```

[7]:
```
   Unnamed: 0  symboling  normalized-losses         make num-of-doors  \
0           0          3                122  alfa-romero          two
1           1          3                122  alfa-romero          two
2           2          1                122  alfa-romero          two
3           3          2                164         audi         four
4           4          2                164         audi         four

    body-style drive-wheels engine-location  wheel-base    length  …  \
0  convertible          rwd           front        88.6  0.811148  …
1  convertible          rwd           front        88.6  0.811148  …
```

```
2   hatchback          rwd          front         94.5  0.822681  …
3       sedan          fwd          front         99.8  0.848630  …
4       sedan          4wd          front         99.4  0.848630  …

   peak-rpm  city-mpg  highway-mpg    price  city-L/100km  horsepower-binned  \
0    5000.0        21     8.703704  13495.0     11.190476                Low
1    5000.0        21     8.703704  16500.0     11.190476                Low
2    5000.0        19     9.038462  16500.0     12.368421             Medium
3    5500.0        24     7.833333  13950.0      9.791667                Low
4    5500.0        18    10.681818  17450.0     13.055556                Low

  fuel-type-diesel  fuel-type-gas  aspiration-std  aspiration-turbo
0            False           True            True             False
1            False           True            True             False
2            False           True            True             False
3            False           True            True             False
4            False           True            True             False

[5 rows x 31 columns]
```

First, let's only use numeric data:

[8]:
```
df=df._get_numeric_data()
df.head(20)
```

[8]:
```
    Unnamed: 0  symboling  normalized-losses  wheel-base    length     width  \
0            0          3                122        88.6  0.811148  0.890278
1            1          3                122        88.6  0.811148  0.890278
2            2          1                122        94.5  0.822681  0.909722
3            3          2                164        99.8  0.848630  0.919444
4            4          2                164        99.4  0.848630  0.922222
5            5          2                122        99.8  0.851994  0.920833
6            6          1                158       105.8  0.925997  0.991667
7            7          1                122       105.8  0.925997  0.991667
8            8          1                158       105.8  0.925997  0.991667
9            9          2                192       101.2  0.849592  0.900000
10          10          0                192       101.2  0.849592  0.900000
11          11          0                188       101.2  0.849592  0.900000
12          12          0                188       101.2  0.849592  0.900000
13          13          1                122       103.5  0.908217  0.929167
14          14          0                122       103.5  0.908217  0.929167
15          15          0                122       103.5  0.931283  0.943056
16          16          0                122       110.0  0.946660  0.984722
17          17          2                121        88.4  0.678039  0.837500
18          18          1                 98        94.5  0.749159  0.883333
19          19          0                 81        94.5  0.763095  0.883333

    height  curb-weight  engine-size  bore  …  horsepower  peak-rpm  \
```

0	0.816054	2548	130	3.47	…	111	5000.0
1	0.816054	2548	130	3.47	…	111	5000.0
2	0.876254	2823	152	2.68	…	154	5000.0
3	0.908027	2337	109	3.19	…	102	5500.0
4	0.908027	2824	136	3.19	…	115	5500.0
5	0.887960	2507	136	3.19	…	110	5500.0
6	0.931438	2844	136	3.19	…	110	5500.0
7	0.931438	2954	136	3.19	…	110	5500.0
8	0.934783	3086	131	3.13	…	140	5500.0
9	0.908027	2395	108	3.50	…	101	5800.0
10	0.908027	2395	108	3.50	…	101	5800.0
11	0.908027	2710	164	3.31	…	121	4250.0
12	0.908027	2765	164	3.31	…	121	4250.0
13	0.931438	3055	164	3.31	…	121	4250.0
14	0.931438	3230	209	3.62	…	182	5400.0
15	0.897993	3380	209	3.62	…	182	5400.0
16	0.941472	3505	209	3.62	…	182	5400.0
17	0.889632	1488	61	2.91	…	48	5100.0
18	0.869565	1874	90	3.03	…	70	5400.0
19	0.869565	1909	90	3.03	…	70	5400.0

	city-mpg	highway-mpg	price	city-L/100km	fuel-type-diesel	\
0	21	8.703704	13495.0	11.190476	False	
1	21	8.703704	16500.0	11.190476	False	
2	19	9.038462	16500.0	12.368421	False	
3	24	7.833333	13950.0	9.791667	False	
4	18	10.681818	17450.0	13.055556	False	
5	19	9.400000	15250.0	12.368421	False	
6	19	9.400000	17710.0	12.368421	False	
7	19	9.400000	18920.0	12.368421	False	
8	17	11.750000	23875.0	13.823529	False	
9	23	8.103448	16430.0	10.217391	False	
10	23	8.103448	16925.0	10.217391	False	
11	21	8.392857	20970.0	11.190476	False	
12	21	8.392857	21105.0	11.190476	False	
13	20	9.400000	24565.0	11.750000	False	
14	16	10.681818	30760.0	14.687500	False	
15	16	10.681818	41315.0	14.687500	False	
16	15	11.750000	36880.0	15.666667	False	
17	47	4.433962	5151.0	5.000000	False	
18	38	5.465116	6295.0	6.184211	False	
19	38	5.465116	6575.0	6.184211	False	

	fuel-type-gas	aspiration-std	aspiration-turbo
0	True	True	False
1	True	True	False
2	True	True	False

```
3            True          True          False
4            True          True          False
5            True          True          False
6            True          True          False
7            True          True          False
8            True         False           True
9            True          True          False
10           True          True          False
11           True          True          False
12           True          True          False
13           True          True          False
14           True          True          False
15           True          True          False
16           True          True          False
17           True          True          False
18           True          True          False
19           True          True          False

[20 rows x 22 columns]
```

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

[9]:
```
#df.drop(['Unnamed: 0.1', 'Unnamed: 0'], axis=1, inplace=True)

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

Libraries for plotting:

[10]:
```
from ipywidgets import interact, interactive, fixed, interact_manual
```

Functions for Plotting

[11]:
```
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()
```

[12]:
```
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))),␣
 ↪label='Predicted Function')
    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:

[13]:
```
y_data = df['price']
```

Drop price data in dataframe **x_data**:

[14]:
```
x_data=df.drop('price',axis=1)
```

Now, we randomly split our data into training and testing data using the function train_test_split.

[15]:
```
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.
 ↪10, random_state=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.

[16]:
```
x_train1, x_test1, y_train1, y_test1 = train_test_split(x_data, y_data, test_size=0.4, random_state=0)
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
```

Let's import LinearRegression from the module linear_model.

[17]:
```
from sklearn.linear_model import LinearRegression
```

We create a Linear Regression object:

[18]:
```
lre=LinearRegression()
```

We fit the model using the feature "horsepower":

[19]:
```
lre.fit(x_train[['horsepower']], y_train)
```

[19]:
```
LinearRegression()
```

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

[20]:
```
lre.score(x_test[['horsepower']], y_test)
```

[20]:
```
0.3635480624962414
```

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

[21]:
```
lre.score(x_train[['horsepower']], y_train)
```

[21]:
```
0.662028747521533
```

[22]:
```
x_train1, x_test1, y_train1, y_test1 = train_test_split(x_data, y_data, test_size=0.4, random_state=0)
lre.fit(x_train1[['horsepower']],y_train1)
lre.score(x_test1[['horsepower']],y_test1)
```

[22]:
```
0.7139737368233015
```

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.

```
[23]: 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.

```
[24]: 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:

```
[25]: Rcross
```

```
[25]: array([0.77465419, 0.51718424, 0.74814454, 0.04825398])
```

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

```
[26]: print("The mean of the folds are", Rcross.mean(), "and the standard deviation␣
      ↪is" , Rcross.std())
```

```
The mean of the folds are 0.5220592359225413 and the standard deviation is
0.29130480666118463
```

We can use negative squared error as a score by setting the parameter 'scoring' metric to 'neg_mean_squared_error'.

```
[27]: -1 * cross_val_score(lre,x_data[['horsepower']],␣
      ↪y_data,cv=4,scoring='neg_mean_squared_error')
```

```
[27]: array([20251357.7835463 , 43743920.05390439, 12525158.34507633,
             17564549.69976654])
```

```
[28]: Rc=cross_val_score(lre,x_data[['horsepower']], y_data,cv=2)
      Rc.mean()
```

```
[28]: 0.516835099979672
```

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:

```
[29]: 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:

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

```
[30]: array([14142.23793549, 14142.23793549, 20815.3029844 , 12745.549902  ,
             14762.9881726 ])
```

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.

[31]:
```
lr = LinearRegression()
lr.fit(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],␣
 ↪y_train)
```

[31]:
```
LinearRegression()
```

Prediction using training data:

[32]:
```
yhat_train = lr.predict(x_train[['horsepower', 'curb-weight', 'engine-size',␣
 ↪'highway-mpg']])
yhat_train[0:5]
```

[32]:
```
array([ 7625.80349764, 28447.913572  , 14843.22185221,  3855.72028472,
       34567.84349196])
```

Prediction using test data:

[33]:
```
yhat_test = lr.predict(x_test[['horsepower', 'curb-weight', 'engine-size',␣
 ↪'highway-mpg']])
yhat_test[0:5]
```

[33]:
```
array([11043.92953392,  5844.12954446, 11258.50532848,  6886.86402714,
       15325.73021747])
```

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

[34]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

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

[35]:
```
Title = 'Distribution  Plot of  Predicted Value Using Training Data vs Training␣
 ↪Data Distribution'
DistributionPlot(y_train, yhat_train, "Actual Values (Train)", "Predicted␣
 ↪Values (Train)", Title)
```

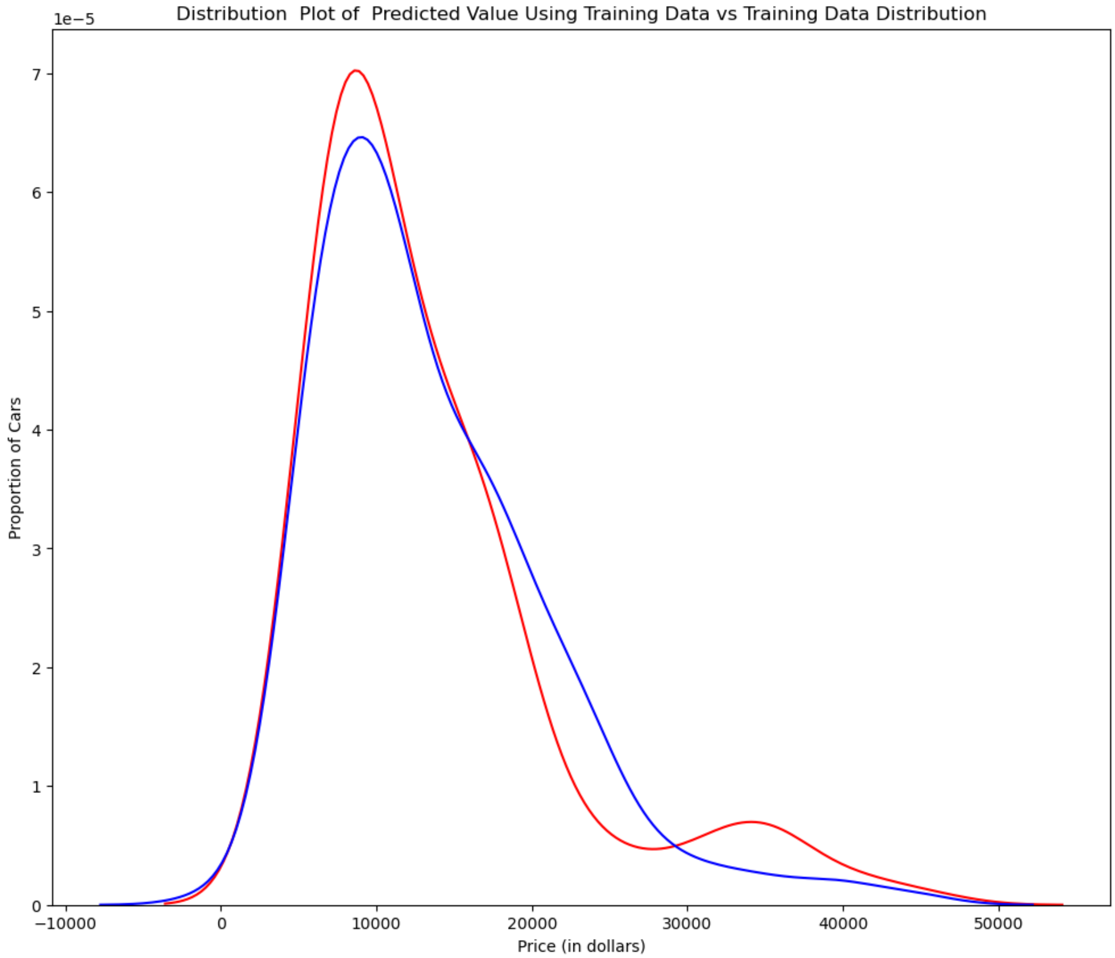


Figure 1: Plot of predicted values using the training data compared to the actual values of the training data.

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.

[36]:
```
Title='Distribution  Plot of  Predicted Value Using Test Data vs Data␣
  ↪Distribution of Test Data'
DistributionPlot(y_test,yhat_test,"Actual Values (Test)","Predicted Values␣
  ↪(Test)",Title)
```

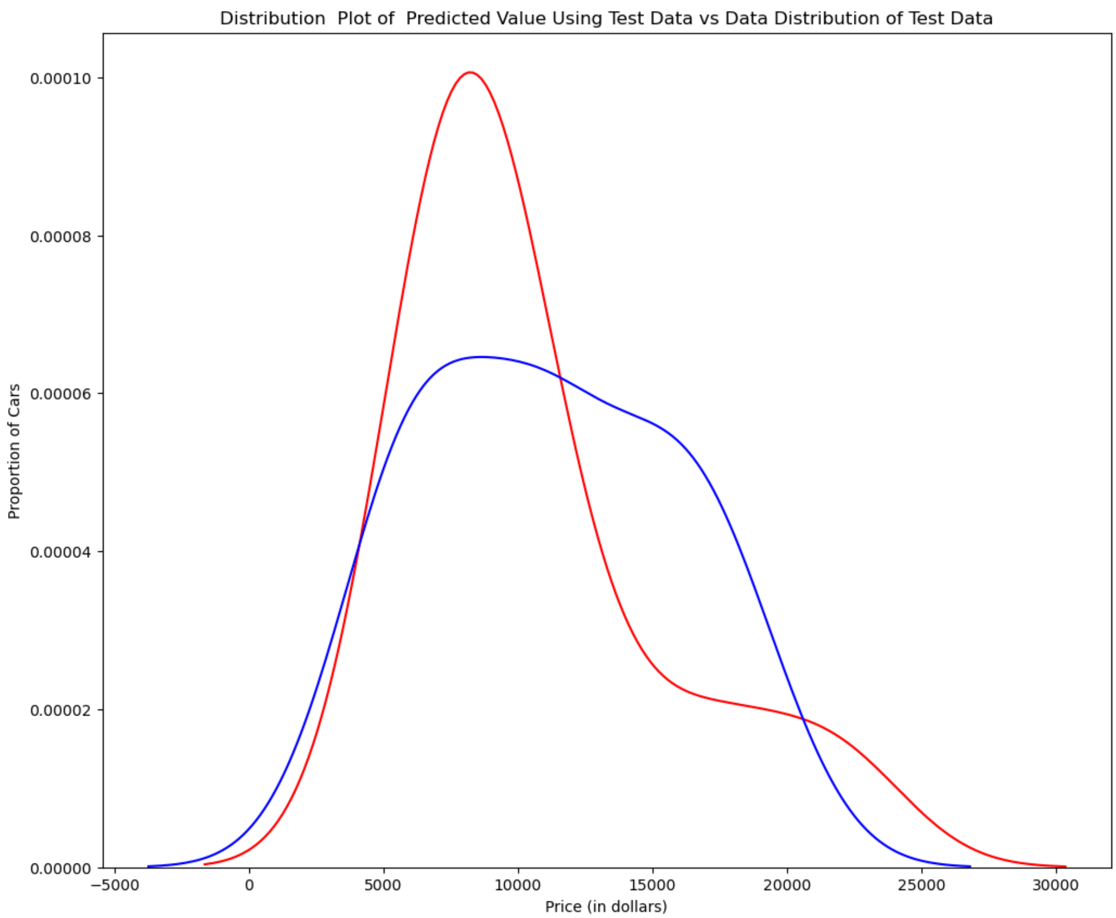


Figure 2: Plot of predicted value using the test data compared to the actual values of the test data.

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 5000 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.

[37]:
```
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:

[38]:
```
x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.
↪45, random_state=0)
```

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

[39]:
```
pr = PolynomialFeatures(degree=5)
x_train_pr = pr.fit_transform(x_train[['horsepower']])
x_test_pr = pr.fit_transform(x_test[['horsepower']])
pr
```

[39]:
```
PolynomialFeatures(degree=5)
```

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

[40]:
```
poly = LinearRegression()
poly.fit(x_train_pr, y_train)
```

[40]:
```
LinearRegression()
```

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

[41]:
```
yhat = poly.predict(x_test_pr)
yhat[0:5]
```

[41]:
```
array([ 6727.50134739,  7306.62980155, 12213.64942948, 18895.18190498,
       19997.01014697])
```

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

[42]:
```
print("Predicted values:", yhat[0:4])
print("True values:", y_test[0:4].values)
```

```
Predicted values: [ 6727.50134739  7306.62980155 12213.64942948 18895.18190498]
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.

[43]:
```
PollyPlot(x_train['horsepower'], x_test['horsepower'], y_train, y_test, poly,pr)
```

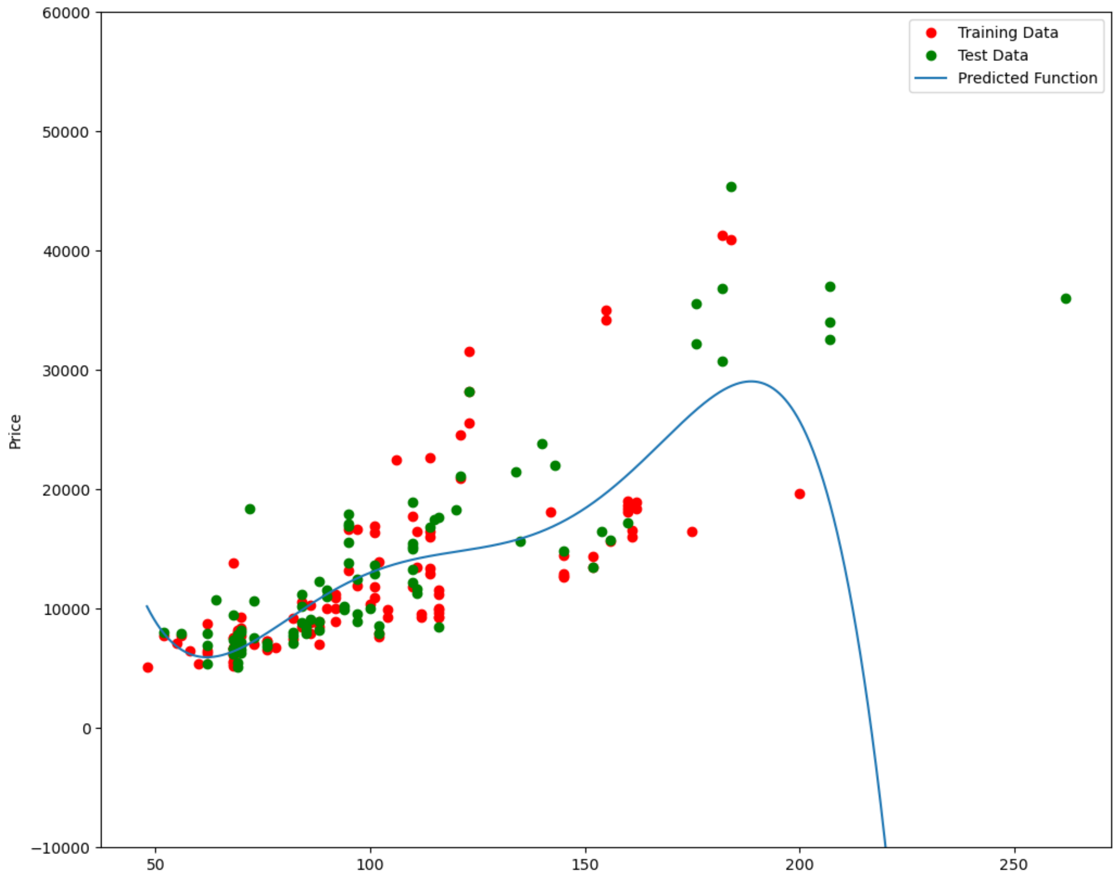


Figure 3: A polynomial regression model where red dots represent training data, green dots represent test data, and the blue line represents the model prediction.

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:

[44]:
```
poly.score(x_train_pr, y_train)
```

[44]: 0.5568527852117562

R^2 of the test data:

[45]:
```
poly.score(x_test_pr, y_test)
```

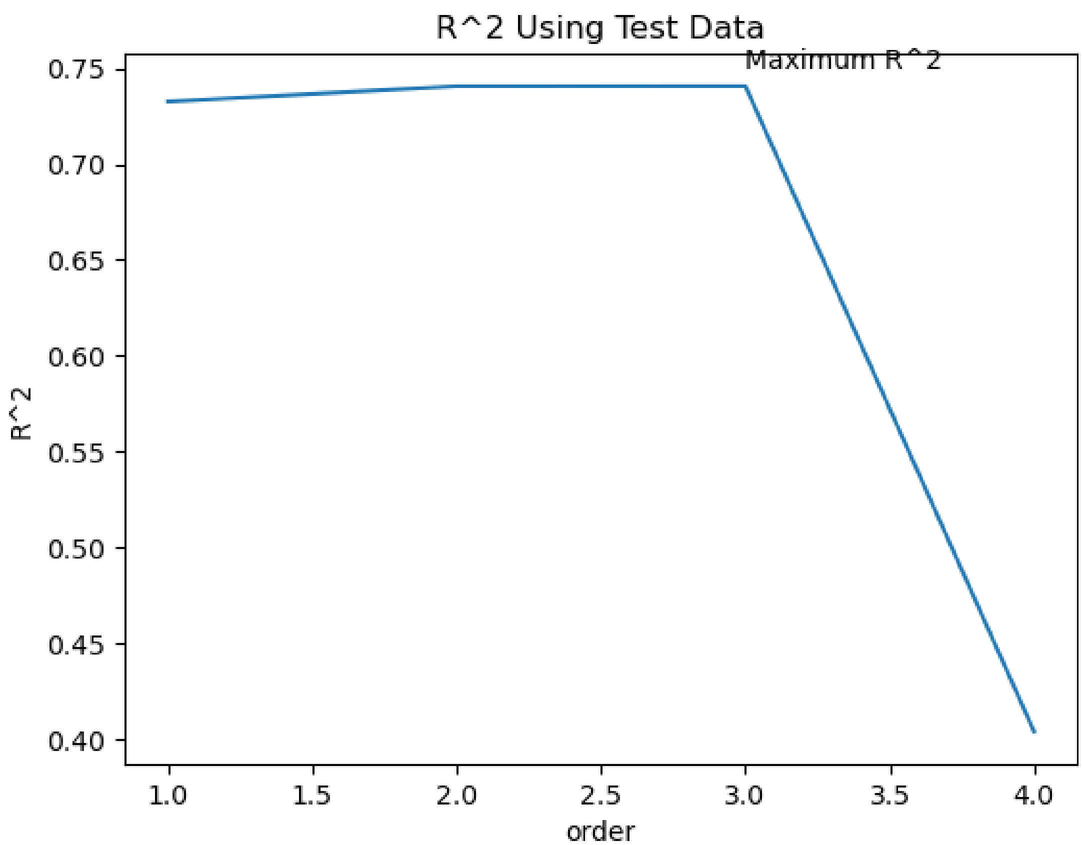
[45]: -29.815108072386607

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:

[46]:
```
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 ')
```

[46]:
```
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.

[47]:
```
def f(order, test_data):
    x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=test_data, random_state=0)
    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, poly,pr)
```

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

[48]:
```
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…
```

[48]:
```
<function __main__.f(order, test_data)>
```

[49]:
```
pr1=PolynomialFeatures(degree=2)
```

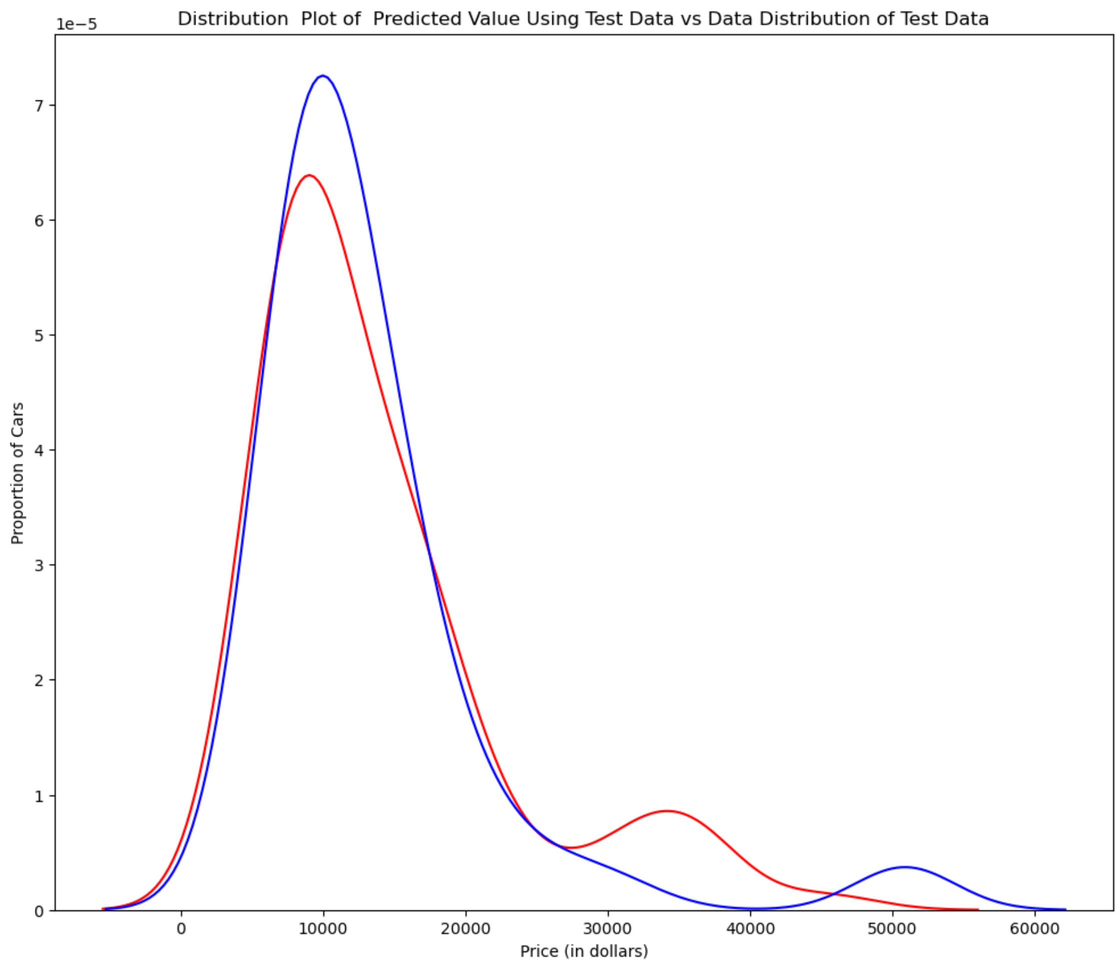
[50]:
```
x_train_pr1=pr1.fit_transform(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])

x_test_pr1=pr1.fit_transform(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])
```

[51]:
```
x_train_pr1.shape #there are now 15 features
```

[51]:
```
(110, 15)
```

[52]:
```
poly1=LinearRegression().fit(x_train_pr1,y_train)
```

[53]:
```
yhat_test1=poly1.predict(x_test_pr1)

Title='Distribution  Plot of  Predicted Value Using Test Data vs Data Distribution of Test Data'

DistributionPlot(y_test, yhat_test1, "Actual Values (Test)", "Predicted Values (Test)", Title)
```



[54]:
```
#The predicted value is higher than actual value for cars where the price␣
↪$10,000 range, conversely the predicted price is lower than the price cost␣
↪in the $30,000 to $40,000 range. As such the model is not as accurate in␣
↪these ranges.
```

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.

[55]:
```
pr=PolynomialFeatures(degree=2)
x_train_pr=pr.fit_transform(x_train[['horsepower', 'curb-weight',␣
↪'engine-size', 'highway-mpg','normalized-losses','symboling']])
x_test_pr=pr.fit_transform(x_test[['horsepower', 'curb-weight', 'engine-size',␣
↪'highway-mpg','normalized-losses','symboling']])
```

Let's import Ridge from the module linear models.

[56]:
```
from sklearn.linear_model import Ridge
```

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

[57]:
```
RigeModel=Ridge(alpha=1)
```

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

[58]:
```
RigeModel.fit(x_train_pr, y_train)
```

[58]:
```
Ridge(alpha=1)
```

Similarly, you can obtain a prediction:

[59]:
```
yhat = RigeModel.predict(x_test_pr)
```

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

[60]:
```
print('predicted:', yhat[0:4])
print('test set :', y_test[0:4].values)
```

```
predicted: [ 5501.44956318 10293.7232663  21646.09716839 19329.3421769 ]
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.

[61]:
```
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.
 ↪score(x_train_pr, y_train)

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

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

```
100%|                      | 1000/1000 [00:10<00:00,
94.76it/s, Test Score=0.673, Train Score=0.86]
```

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

[62]:
```
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()
```

[62]: `<matplotlib.legend.Legend at 0x17512f53410>`

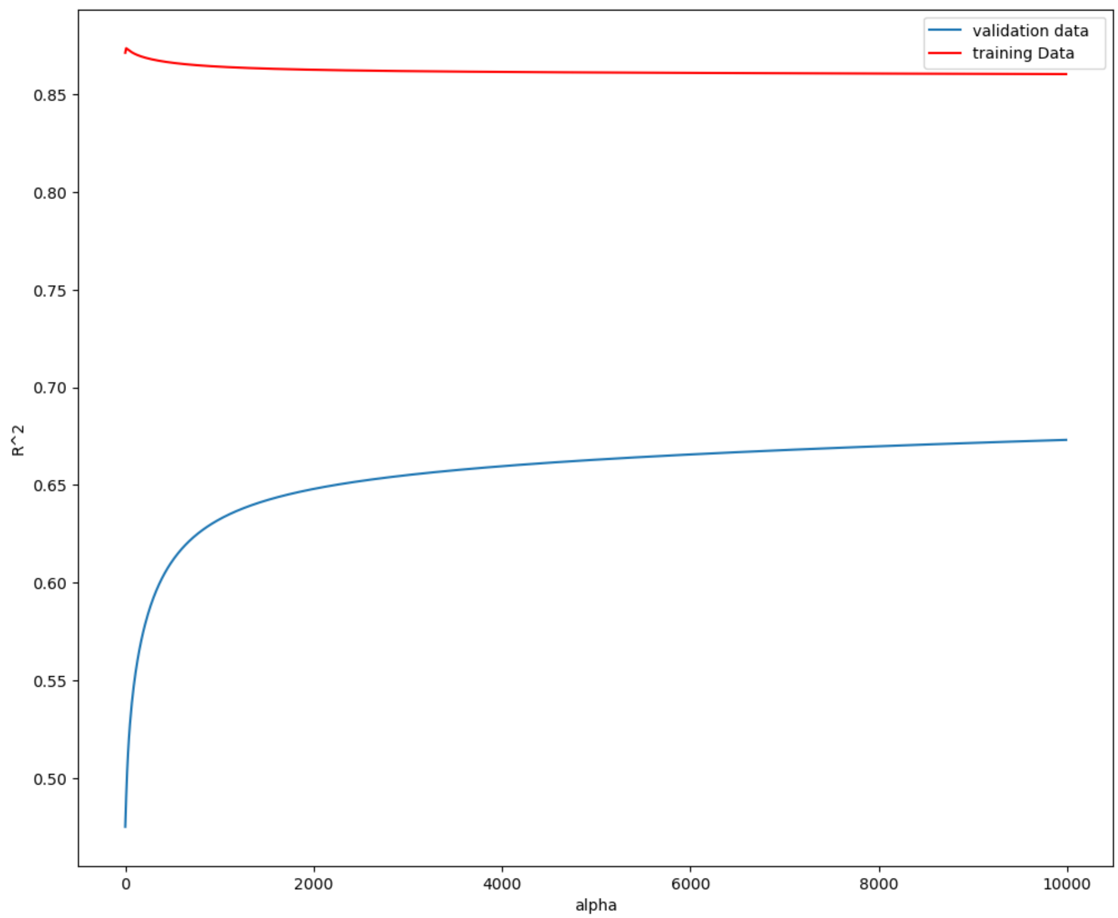


Figure 4: The blue line represents the R^2 of the validation data, and the red line represents the R^2 of the training data. The x-axis represents the different values of Alpha.

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.

[63]:
```
RigeModel = Ridge(alpha=10)
RigeModel.fit(x_train_pr, y_train)
RigeModel.score(x_test_pr, y_test)
```

[63]:
```
0.4905348461839951
```

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.

[64]:
```
from sklearn.model_selection import GridSearchCV
```

We create a dictionary of parameter values:

[65]:
```
parameters1= [{'alpha': [0.001,0.1,1, 10, 100, 1000, 10000, 100000, 100000]}]
parameters1
```

[65]:
```
[{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 100000]}]
```

Create a Ridge regression object:

[66]:
```
RR=Ridge()
RR
```

[66]:
```
Ridge()
```

Create a ridge grid search object:

[67]:
```
Grid1 = GridSearchCV(RR, parameters1,cv=4)
```

Fit the model:

[68]:
```
Grid1.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],␣
↪y_data)
```

[68]:
```
GridSearchCV(cv=4, estimator=Ridge(),
             param_grid=[{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000,
                                    100000]}])
```

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:

[69]:
```
BestRR=Grid1.best_estimator_
BestRR
```

[69]:
```
Ridge(alpha=10000)
```

We now test our model on the test data:

[70]:
```
BestRR.score(x_test[['horsepower', 'curb-weight', 'engine-size',␣
↪'highway-mpg']], y_test)
```

[70]: 0.8411508430484373

[71]:
```
parameters2 = [{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 100000]}]

Grid2 = GridSearchCV(Ridge(), parameters2, cv=4)
Grid2.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],␣
↪y_data)
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']], y_data)
```

[71]: Ridge(alpha=10000)

[ ]:
```

```