Model Evaluation and Refinement

Estimated time needed: **30** minutes

Objectives

After completing this lab you will be able to:

- Evaluate and refine prediction models

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

```
#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'])
await piplite.install(['ipywidgets'])
```

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

```
#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:

```
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

```
#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.

```
#you will need to 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:

```
df = pd.read_csv("module_5_auto.csv", header=0)

df.head()

   Unnamed: 0.1  Unnamed: 0  symboling  normalized-losses         make
\
0             0           0          3                122  alfa-romero

1             1           1          3                122  alfa-romero

2             2           2          1                122  alfa-romero

3             3           3          2                164         audi

4             4           4          2                164         audi


  aspiration num-of-doors   body-style drive-wheels engine-
location  ...  \
0        std          two  convertible          rwd
front  ...
1        std          two  convertible          rwd
front  ...
2        std          two    hatchback          rwd
front  ...
3        std         four        sedan          fwd
front  ...
4        std         four        sedan          4wd
front  ...

   compression-ratio  horsepower  peak-rpm  city-mpg  highway-mpg
```

```
price  \
0                9.0       111.0    5000.0        21            27
13495.0
1                9.0       111.0    5000.0        21            27
16500.0
2                9.0       154.0    5000.0        19            26
16500.0
3               10.0       102.0    5500.0        24            30
13950.0
4                8.0       115.0    5500.0        18            22
17450.0

  city-L/100km  horsepower-binned diesel  gas
0    11.190476             Medium      0    1
1    11.190476             Medium      0    1
2    12.368421             Medium      0    1
3     9.791667             Medium      0    1
4    13.055556             Medium      0    1

[5 rows x 31 columns]
```

First, let's only use numeric data:

```
df=df._get_numeric_data()
df.head()

   Unnamed: 0.1  Unnamed: 0  symboling  normalized-losses  wheel-base
\
0             0           0          3                122        88.6

1             1           1          3                122        88.6

2             2           2          1                122        94.5

3             3           3          2                164        99.8

4             4           4          2                164        99.4


     length     width  height  curb-weight  engine-size  ...
stroke  \
0  0.811148  0.890278    48.8         2548          130  ...    2.68

1  0.811148  0.890278    48.8         2548          130  ...    2.68

2  0.822681  0.909722    52.4         2823          152  ...    3.47

3  0.848630  0.919444    54.3         2337          109  ...    3.40

4  0.848630  0.922222    54.3         2824          136  ...    3.40
```

```
   compression-ratio  horsepower  peak-rpm  city-mpg  highway-mpg
price  \
0                9.0       111.0    5000.0        21           27
13495.0
1                9.0       111.0    5000.0        21           27
16500.0
2                9.0       154.0    5000.0        19           26
16500.0
3               10.0       102.0    5500.0        24           30
13950.0
4                8.0       115.0    5500.0        18           22
17450.0

   city-L/100km  diesel  gas
0     11.190476       0    1
1     11.190476       0    1
2     12.368421       0    1
3      9.791667       0    1
4     13.055556       0    1

[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.

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

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

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

   curb-weight  engine-size  bore  stroke  compression-ratio
horsepower  \
0         2548          130  3.47    2.68                9.0
111.0
1         2548          130  3.47    2.68                9.0
```

```
111.0
2          2823          152  2.68     3.47                   9.0
154.0
3          2337          109  3.19     3.40                  10.0
102.0
4          2824          136  3.19     3.40                   8.0
115.0

   peak-rpm  city-mpg  highway-mpg    price  city-L/100km  diesel  gas

0    5000.0        21           27  13495.0     11.190476       0    1

1    5000.0        21           27  16500.0     11.190476       0    1

2    5000.0        19           26  16500.0     12.368421       0    1

3    5500.0        24           30  13950.0      9.791667       0    1

4    5500.0        18           22  17450.0     13.055556       0    1
```

Libraries for plotting:

```
from ipywidgets import interact, interactive, fixed, interact_manual

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()

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()

y_data = df['price']
```

Drop price data in dataframe **x_data**:

```
x_data=df.drop('price',axis=1)
```

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

```
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.

```
# 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_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.

```
from sklearn.linear_model import LinearRegression
```

We create a Linear Regression object:

```
lre=LinearRegression()
```

We fit the model using the feature "horsepower":

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

```
LinearRegression()
```

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

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

```
0.3635875575078824
```

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

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

```
0.6619724197515103
```

```
# 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_size=0.4, random_state=0)
lre.fit(x_train1[['horsepower']],y_train1)
lre.score(x_test1[['horsepower']],y_test1)
```

```
0.7139364665406973
```

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.

Let's import cross_val_score from the module model_selection.

```
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.

```
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:

```
Rcross
```

```
array([0.7746232 , 0.51716687, 0.74785353, 0.04839605])
```

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

```
print("The mean of the folds are", Rcross.mean(), "and the standard deviation is" , Rcross.std())
```

```
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'.

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

```
array([20254142.84026702, 43745493.26505171, 12539630.34014929,
       17561927.72247586])
```

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

```
0.5166761697127429
```

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:

```
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:

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

```
array([14141.63807508, 14141.63807508, 20814.29423473, 12745.03562306,
       14762.35027598])
```

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

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

```
LinearRegression()
```

Prediction using training data:

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

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

Prediction using test data:

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

```
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.

```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

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

```
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)
```

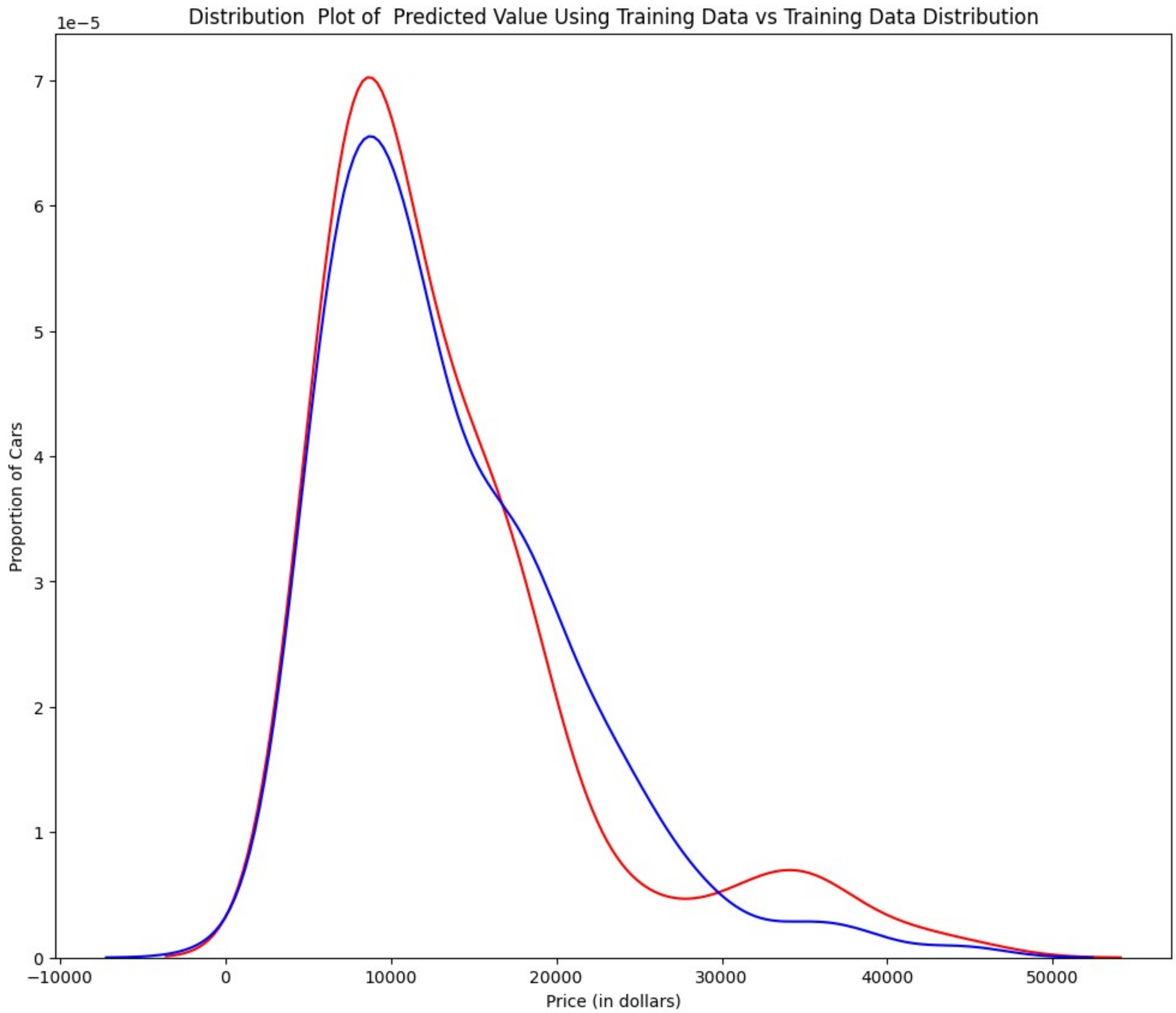


Figure 1: Plot of predicted values using the training data compared to the actual values of the training data.

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.

```
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)
```

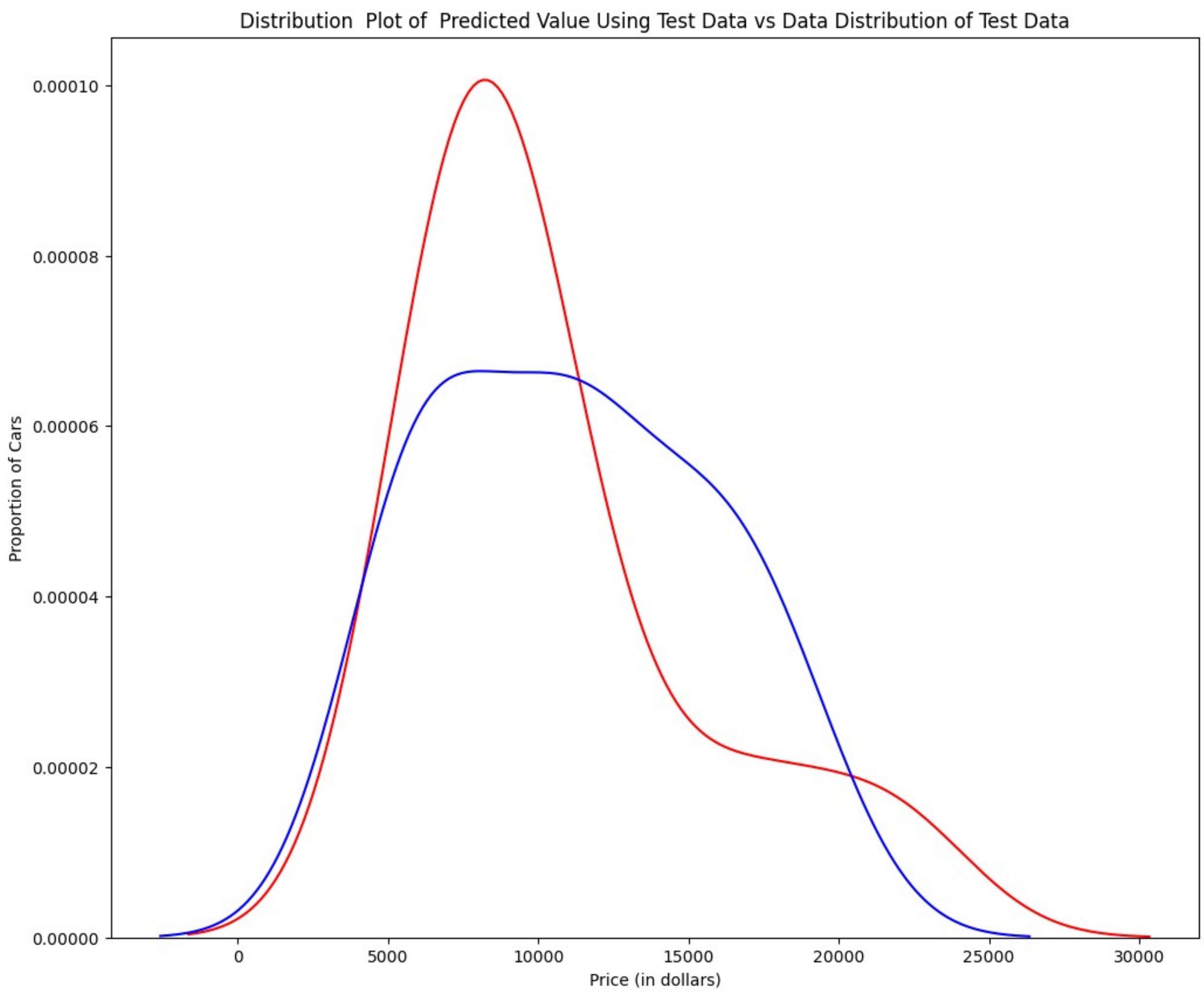


Figure 2: Plot of predicted value using the test data compared to the actual values of the test data.

```
from sklearn.preprocessing import PolynomialFeatures
```

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

```
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'.

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

PolynomialFeatures(degree=5)
```

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

```
poly = LinearRegression()
poly.fit(x_train_pr, y_train)
```

```
LinearRegression()
```

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

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

```
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.

```
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.

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

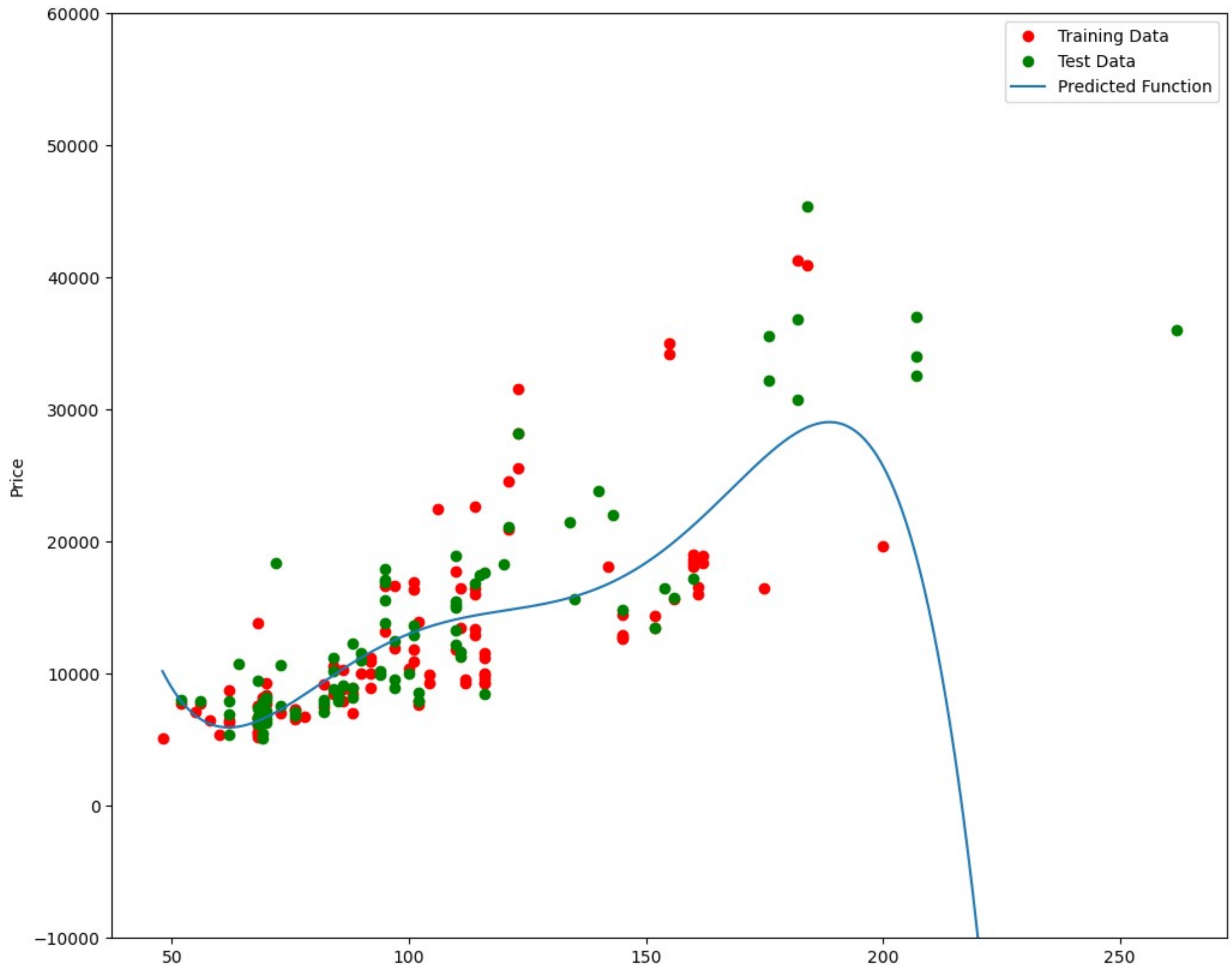


Figure 3: A polynomial regression model where red dots represent training data, green dots represent test data, and the blue line represents the model prediction.

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:

```
poly.score(x_train_pr, y_train)
```

```
0.5567716897754004
```

R^2 of the test data:

```
poly.score(x_test_pr, y_test)
```

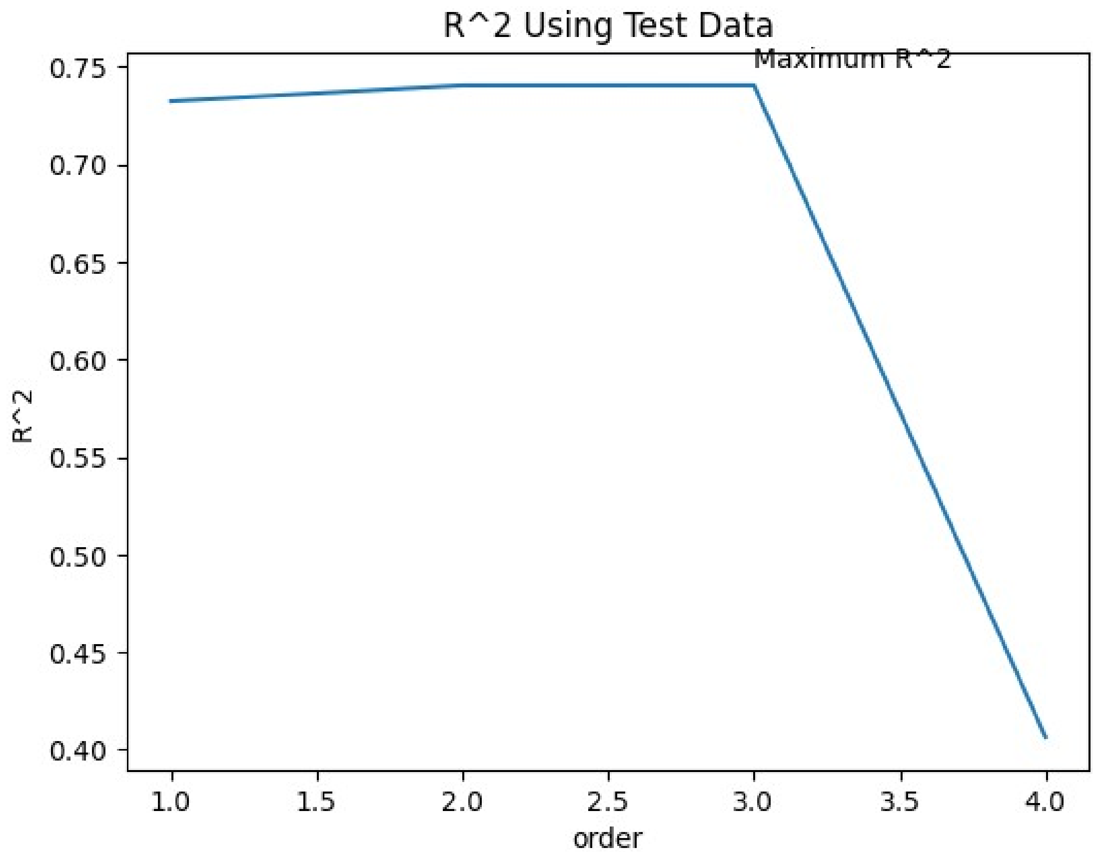
```
-29.87099623387278
```

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:

```
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 ')
```

```
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.

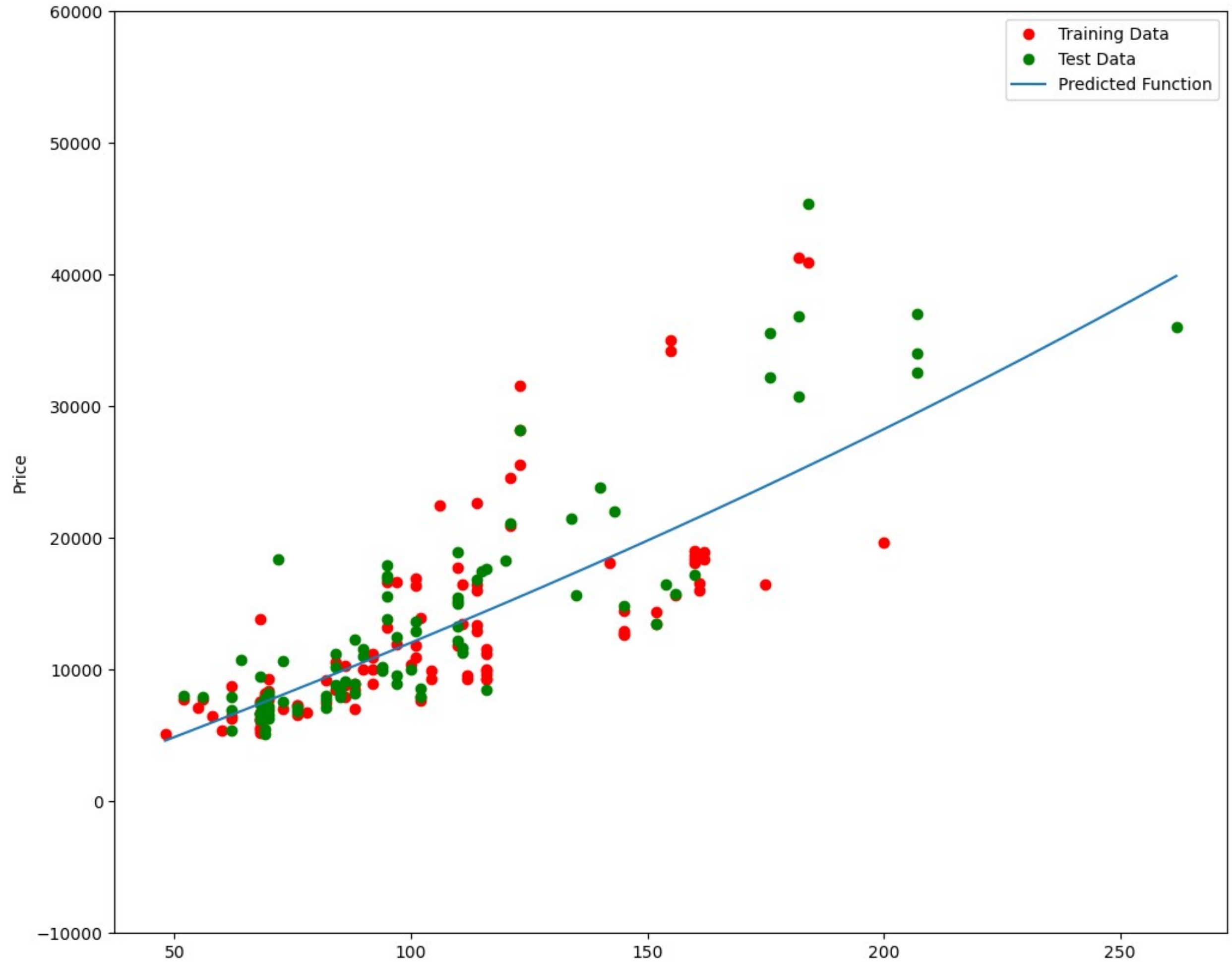
```
def f(order, test_data):
    x_train, x_test, y_train, y_test = train_test_split(x_data, 
y_data, test_size=test_data, random_state=0)
    pr = PolynomialFeatures(degree=order)
    x_train_pr = pr.fit_transform(x_train[['horsepower']])
    x_test_pr = pr.fit_transform(x_test[['horsepower']])
    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.

```
interact(f, order=(0, 6, 1), test_data=(0.05, 0.95, 0.05))
```

{"model_id":"a32a3f83a0f1446cbb24daa1c274aca3","version_major":2,"version_minor":0}

```
<function __main__.f(order, test_data)>
```

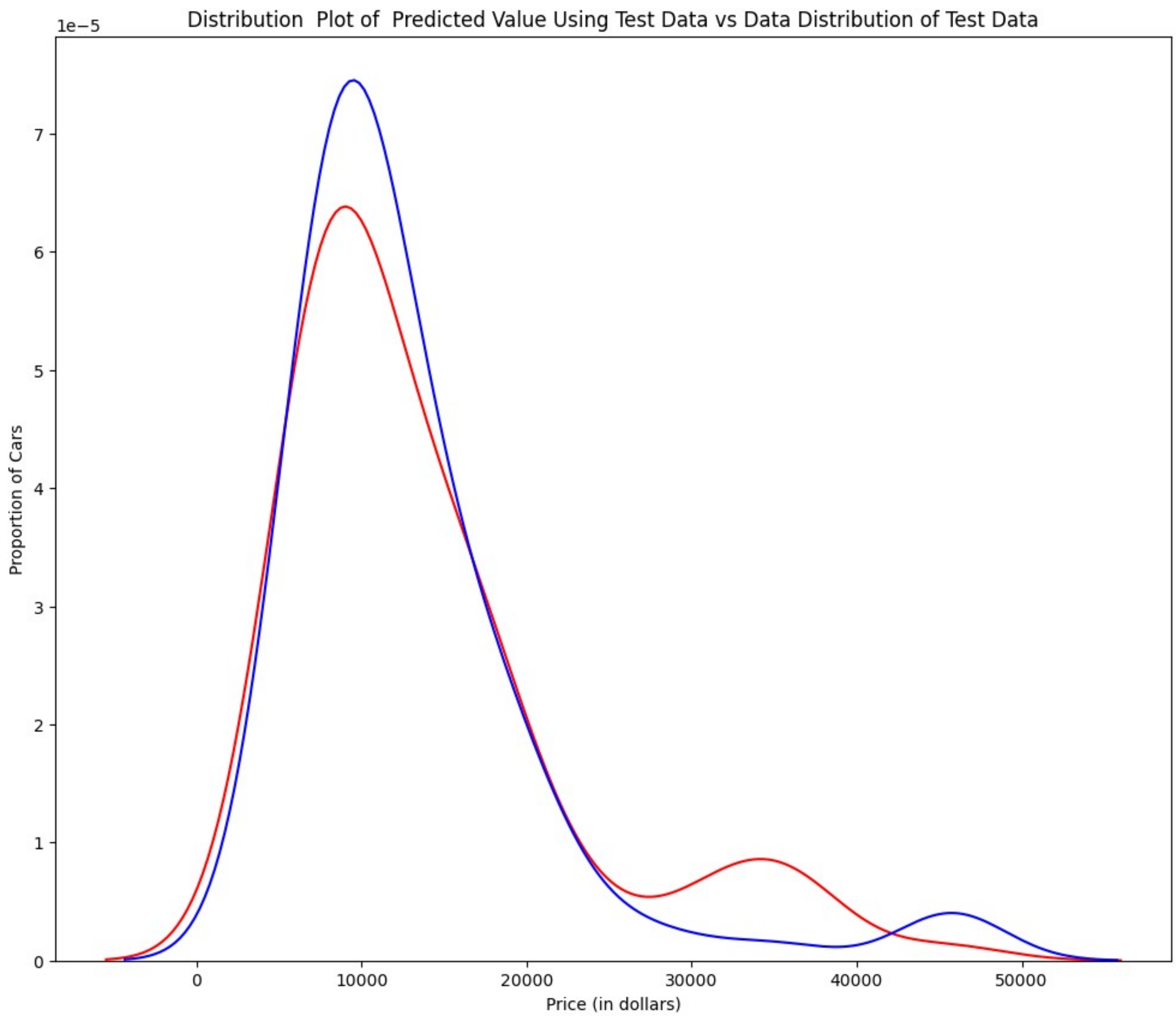



```
# Write your code below and press Shift+Enter to execute
pr1 = PolynomialFeatures(degree=2)
pr1

PolynomialFeatures()

# Write your code below and press Shift+Enter to execute
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']])

# Write your code below and press Shift+Enter to execute
x_train_pr1.shape #there are now 15 features

(110, 15)
```

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

LinearRegression()
```

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.

```
# 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 Distribution of Test Data'
DistributionPlot(y_test, yhat_test1, "Actual Values (Test)", "Predicted Values (Test)", Title)
```



```
# Write your code below and press Shift+Enter to execute
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.
```

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.

```
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.

```
from sklearn.linear_model import Ridge
```

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

```
RigeModel=Ridge(alpha=1)
```

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

```
RigeModel.fit(x_train_pr, y_train)

Ridge(alpha=1)
```

Similarly, you can obtain a prediction:

```
yhat = RigeModel.predict(x_test_pr)
```

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

```
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.

```python
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 [00:26<00:00, 37.26it/s, Test Score=0.564,
Train Score=0.859]
```

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

```python
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()
```

```
<matplotlib.legend.Legend at 0xc9c4fe8>
```

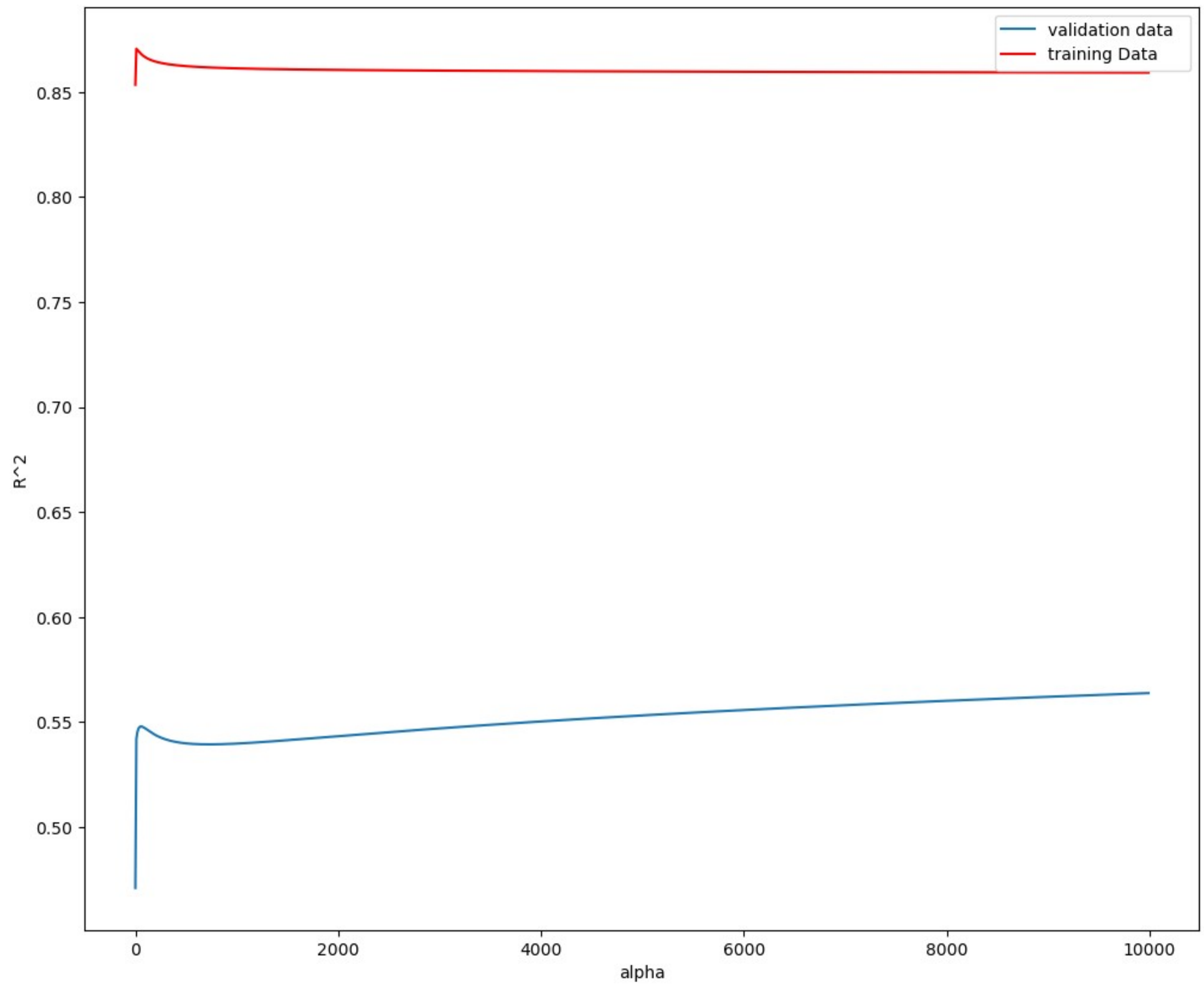


Figure 4: The blue line represents the R^2 of the validation data, and the red line represents the R^2 of the training data. The x-axis represents the different values of Alpha.

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.

```
# Write your code below and press Shift+Enter to execute
RigeModel = Ridge(alpha=10)
RigeModel.fit(x_train_pr, y_train)
RigeModel.score(x_test_pr, y_test)
```

```
0.5418576440208995
```

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.

```
from sklearn.model_selection import GridSearchCV
```

We create a dictionary of parameter values:

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

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

Create a Ridge regression object:

```
RR=Ridge()
RR

Ridge()
```

Create a ridge grid search object:

```
Grid1 = GridSearchCV(RR, parameters1,cv=4)
```

Fit the model:

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

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:

```
BestRR=Grid1.best_estimator_
BestRR

Ridge(alpha=10000)
```

We now test our model on the test data:

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

0.8411649831036152
```

```python
# Write your code below and press Shift+Enter to execute
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)
```

```
Ridge(alpha=10000)
```

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
2020-09-14	2.1	Lakshmi	Made changes in OverFitting section
2020-08-27	2.0	Lavanya	Moved lab to course repo in GitLab