model-evaluation-and-refinement

April 23, 2020

Module 5: Model Evaluation and Refinement

We have built models and made predictions of vehicle prices. Now we will determine how accurate these predictions are.

Table of content

Model Evaluation

Over-fitting, Under-fitting and Model Selection

Ridge Regression

Grid Search

[2]:
```
import pandas as pd
import numpy as np

# Import clean data
path = 'https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/
 ↪CognitiveClass/DA0101EN/module_5_auto.csv'
df = pd.read_csv(path)
```

[3]:
```
df.to_csv('module_5_auto.csv')
```

First lets only use numeric data

[4]:
```
df=df._get_numeric_data()
df.head()
```

[4]:

	Unnamed: 0	Unnamed: 0.1	symboling	normalized-losses	wheel-base	…	stroke
0	0	0	3	122	88.6	…	2.68
1	1	1	3	122	88.6	…	2.68
2	2	2	1	122	94.5	…	3.47
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]
```

Libraries for plotting

[5]:
```
%%capture  #  cell
! pip install ipywidgets
```

[6]:
```
from IPython.display import display
from IPython.html import widgets
from IPython.display import display
from ipywidgets import interact, interactive, fixed, interact_manual
```

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/IPython/html.py:14: ShimWarning: The `IPython.html` package has been
deprecated since IPython 4.0. You should import from `notebook` instead.
`IPython.html.widgets` has moved to `ipywidgets`.
  "`IPython.html.widgets` has moved to `ipywidgets`.", ShimWarning)
```

Functions for plotting

[7]:
```
def DistributionPlot(RedFunction, BlueFunction, RedName, BlueName, Title):
    width = 12
    height = 10
    plt.figure(figsize=(width, height))

    ax1 = sns.distplot(RedFunction, hist=False, color="r", label=RedName)
    ax2 = sns.distplot(BlueFunction, hist=False, color="b", label=BlueName,␣
 ↪ax=ax1)

    plt.title(Title)
    plt.xlabel('Price (in dollars)')
```

```
    plt.ylabel('Proportion of Cars')

    plt.show()
    plt.close()
```

[8]:
```
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()
    plt.show()
```

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:

[9]:
```
y_data = df['price']
```

drop price data in x data

[10]:
```
x_data=df.drop('price',axis=1)
#axis{0 or 'index', 1 or 'columns'}, default 0
#Whether to drop labels from the index (0 or 'index') or columns (1 or␣
 ↪'columns').
```

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

[11]:
```
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.
 ↪15, random_state=1)

# The shape attribute for numpy arrays returns the dimensions of the array.
# If Y has n rows and m columns, then Y.shape is (n,m). So Y.shape[0] is n.

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

```
number of test samples : 31
number of training samples: 170
```

[12]:
```
x_data.shape
```

[12]:
```
(201, 20)
```

The test_size parameter sets the proportion of data that is split into the testing set. In the above, the testing set is set to 10% of the total dataset.

[13]:
```
x_train1,x_test1,y_train1,y_test1 = train_test_split(x_data,y_data,test_size=0.
 ↪4,random_state=1)
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.

[14]:
```
from sklearn.linear_model import LinearRegression
```

We create a Linear Regression object:

[15]:
```
lre=LinearRegression()
```

we fit the model using the feature horsepower

[16]:
```
lre.fit(x_train[['horsepower']], y_train)
```

[16]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

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

[17]:
```
lre.score(x_test[['horsepower']], y_test)
```

[17]: 0.707688374146705

we can see the R^2 of train data is much smaller using the test data.

[18]:
```
lre.score(x_train[['horsepower']], y_train)
```

[18]: 0.6449517437659684

[19]:
```
x_train2,x_test2,y_train2,y_test2 = train_test_split(x_data,y_data,test_size=0.
↪1,random_state=1)
lre.fit(x_train2[['horsepower']],y_train2)
lre.score(x_test2[['horsepower']],y_test2)
```

[19]: 0.36358755750788263

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

Lets import model_selection from the module cross_val_score.

[20]:
```
from sklearn.model_selection import cross_val_score
```

We input the object, the feature in this case ' horsepower', the target data (y_data). **The parameter 'cv' determines the number of folds**; in this case 4.

[21]:
```
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 in the fold:

[22]:
```
Rcross
```

[22]: array([0.7746232 , 0.51716687, 0.74785353, 0.04839605])

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

[23]:
```
print("The mean of the folds are", Rcross.mean(), "and the standard deviation␣
↪is" , Rcross.std())
```

```
The mean of the folds are 0.522009915042119 and the standard deviation is
0.291183944475603
```

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

[24]:
```
-1 * cross_val_score(lre,x_data[['horsepower']],␣
↪y_data,cv=4,scoring='neg_mean_squared_error')
```

[24]:
```
array([20254142.84026702, 43745493.2650517 , 12539630.34014931,
       17561927.72247591])
```

[25]:
```
Rc = cross_val_score(lre,x_data[['horsepower']],y_data,cv=2)
np.mean(Rc)
```

[25]:
```
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,

using one fold to get a prediction while the rest of the folds are used as test data. First import the function:

[26]:
```
from sklearn.model_selection import cross_val_predict
```

We input the object, the feature in this case 'horsepower' , the target data y_data. The parameter 'cv' determines the number of folds; in this case 4. We can produce an output:

[27]:
```
yhat = cross_val_predict(lre,x_data[['horsepower']], y_data,cv=4)
yhat[0:5]
```

[27]:
```
array([14141.63807508, 14141.63807508, 20814.29423473, 12745.03562306,
       14762.35027598])
```

[28]:
```
y_data[0:5]
```

[28]:
```
0    13495.0
1    16500.0
2    16500.0
3    13950.0
4    17450.0
Name: price, dtype: float64
```

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.

[64]:
```
lr = LinearRegression()
lr.fit(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],␣
↪y_train)
```

[64]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

Prediction using training data:

[30]:
```
yhat_train = lr.predict(x_train[['horsepower', 'curb-weight', 'engine-size',␣
 ↪'highway-mpg']])
yhat_train[0:5]
```

[30]:
```
array([11927.70699817, 11236.71672034,  6436.91775515, 21890.22064982,
       16667.18254832])
```

Prediction using test data:

[31]:
```
yhat_test = lr.predict(x_test[['horsepower', 'curb-weight', 'engine-size',␣
 ↪'highway-mpg']])
yhat_test[0:5]
```

[31]:
```
array([11349.16502418,  5914.48335385, 11243.76325987,  6662.03197043,
       15555.76936275])
```

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

[65]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

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

[66]:
```
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)
```

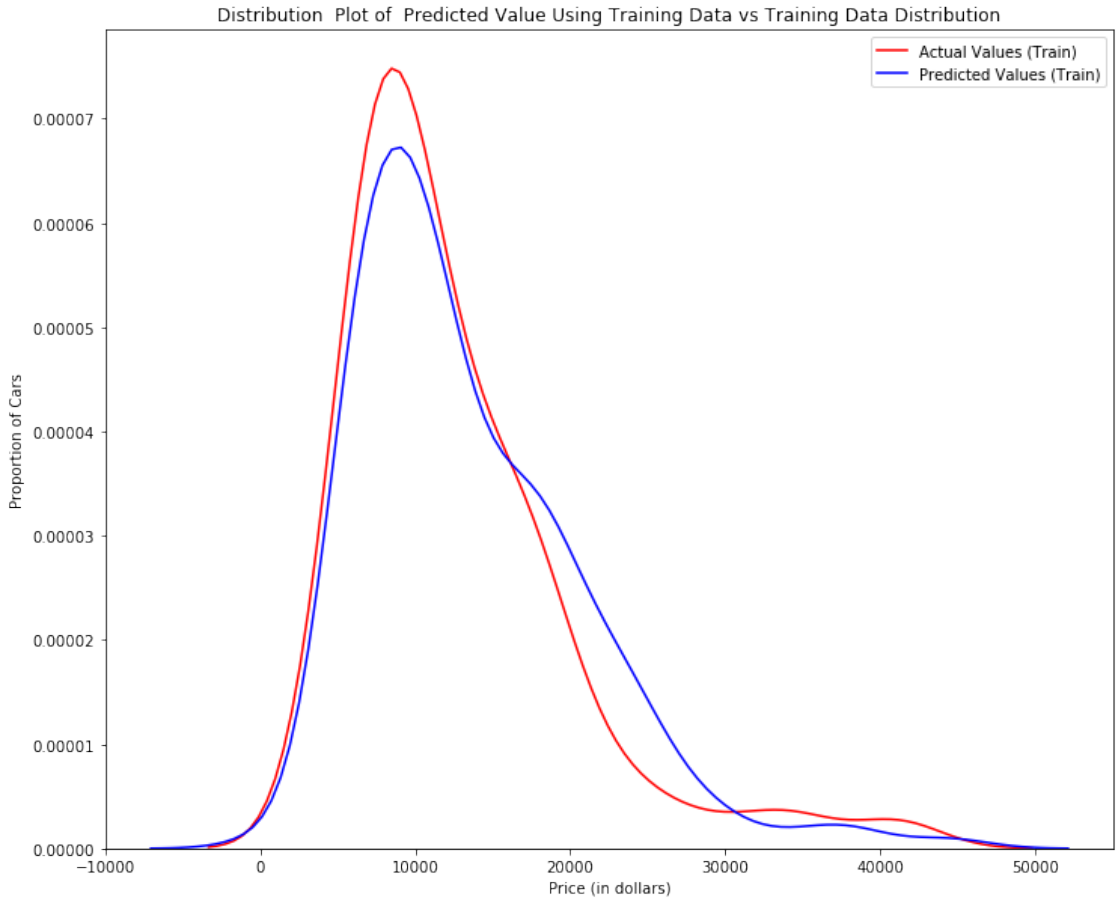
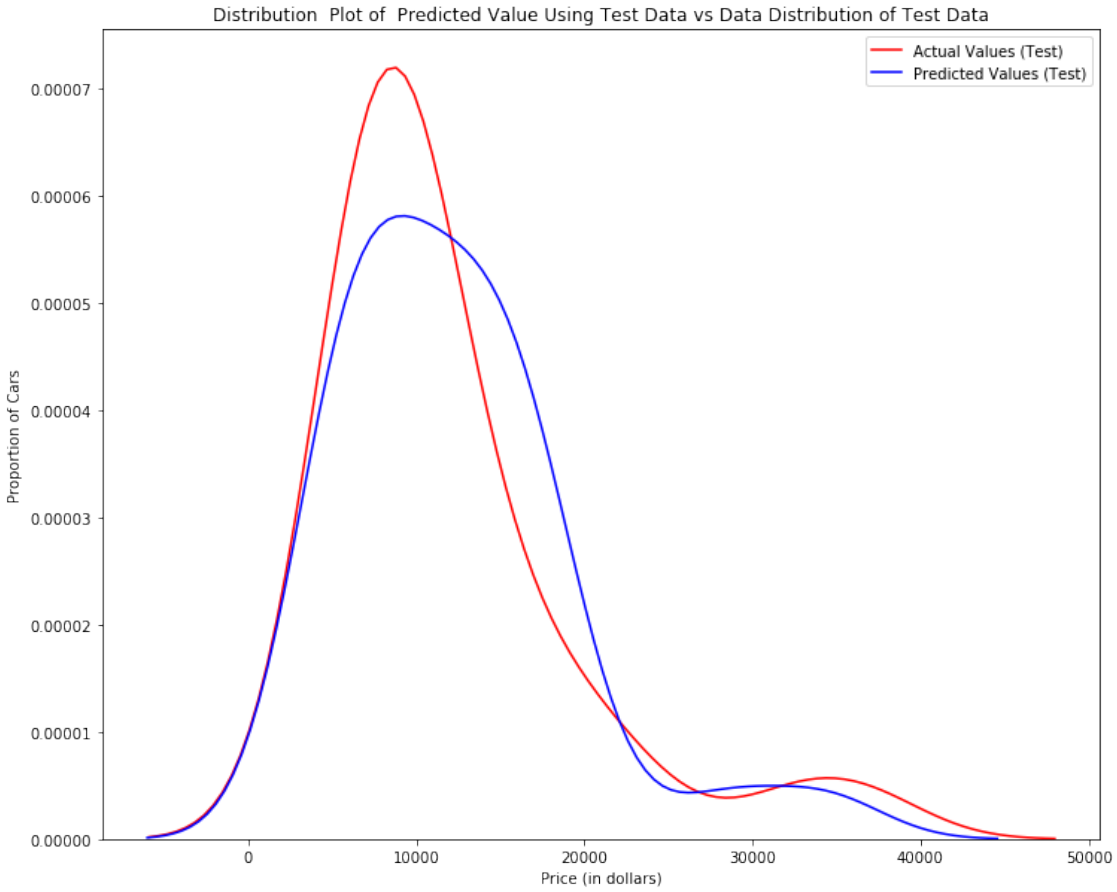


Figure 1: Plot of predicted values using the training data compared to the training data.

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.

[34]:
```
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)
```



Figur 2: Plot of predicted value using the test data compared to the test data.

Comparing Figure 1 and Figure 2; it is evident the distribution of the test data in Figure 1 is much better at fitting the data. This difference in Figure 2 is apparent where the ranges are from 5000 to 15 000. This is where the distribution shape is exceptionally different.

Let's see if polynomial regression also exhibits a drop in the prediction accuracy when analysing the test dataset.

[67]:
```
from sklearn.preprocessing import PolynomialFeatures
```

Overfitting

Overfitting occurs when the model fits the noise, not the underlying process.

Therefore when testing your model using the test-set, your model does not perform as well as 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 testing and the rest for training:

[68]:
```
x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.
↪45, random_state=1)
```

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

[75]:
```
pr = PolynomialFeatures(degree=5)
x_train_pr = pr.fit_transform(x_train[['horsepower']])
x_test_pr = pr.fit_transform(x_test[['horsepower']])
pr
```

[75]:
```
PolynomialFeatures(degree=5, include_bias=True, interaction_only=False)
```

Now let's create a linear regression model "poly" and train it.

[70]:
```
poly = LinearRegression()
poly.fit(x_train_pr, y_train)
```

[70]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

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

[71]:
```
yhat = poly.predict(x_test_pr)
yhat[0:5]
```

[71]:
```
array([11600.6592143 ,  7487.77266898,  9627.3278035 ,  8470.03573061,
       14359.24682451])
```

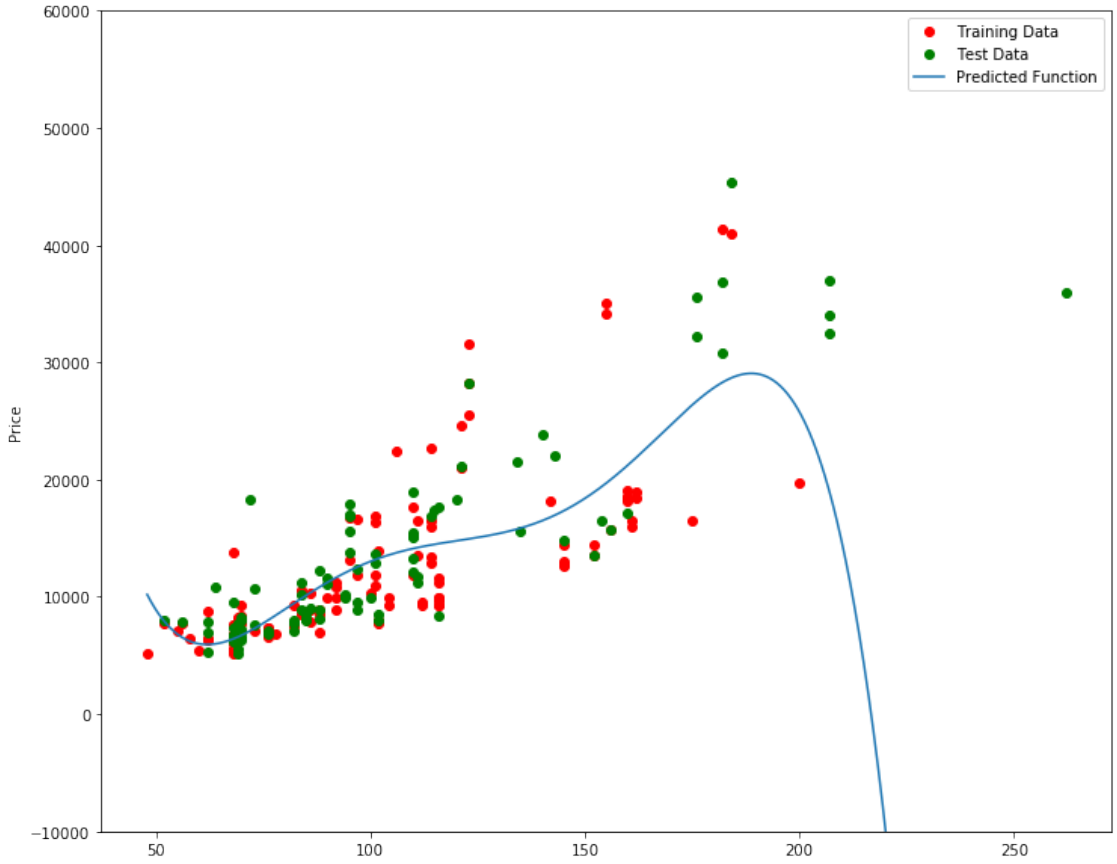
Let's take the first five predicted values and compare it to the actual targets.

[73]:
```
print("Predicted values:", yhat[0:4])
print("True values:", y_test[0:4].values)
```

```
Predicted values: [11600.6592143   7487.77266898  9627.3278035   8470.03573061]
True values: [ 9549.  6229. 10245.  7295.]
```

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.

[41]:
```
PollyPlot(x_train[['horsepower']], x_test[['horsepower']], y_train, y_test,␣
↪poly,pr)
```



Figur 4 A polynomial regression model, 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:

[42]:
```
poly.score(x_train_pr, y_train)
```

[42]:
```
0.556771690212023
```

R^2 of the test data:

[43]:
```
poly.score(x_test_pr, y_test)
```

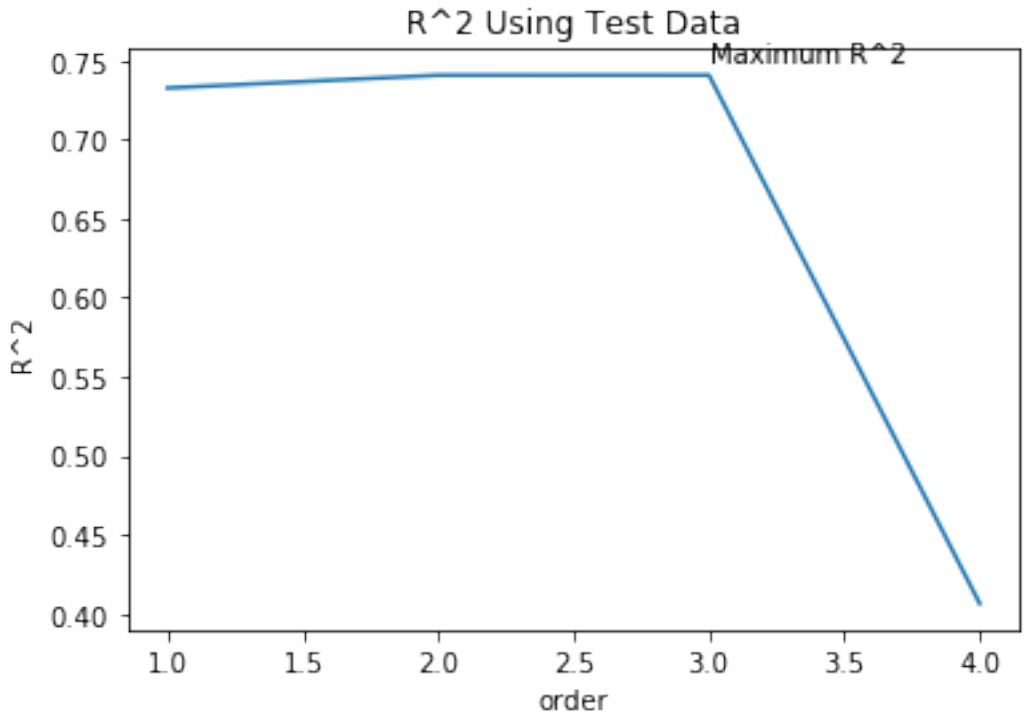
[43]:
```
-29.871340302044153
```

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 plot the results:

[44]:
```
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 ')
```

[44]:
```
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 four.

The following function will be used in the next section; please run the cell.

[76]:
```
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.

[77]:
```
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, d
```

[77]:
```
<function __main__.f(order, test_data)>
```

Part 3: Ridge regression

In this section, we will review Ridge Regression we will see how the parameter Alfa 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.

[78]:
```
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.

[79]:
```
from sklearn.linear_model import Ridge
```

Let's create a Ridge regression object, setting the **regularization parameter** to 0.1

[80]:
```
RigeModel=Ridge(alpha=0.1)
```

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

[81]:
```
RigeModel.fit(x_train_pr, y_train)
```

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/sklearn/linear_model/ridge.py:125: LinAlgWarning: Ill-conditioned
matrix (rcond=9.40677e-17): result may not be accurate.
  overwrite_a=True).T
```

[81]:
```
Ridge(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001)
```

Similarly, you can obtain a prediction:

[82]:
```
yhat = RigeModel.predict(x_test_pr)
```

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

[83]:
```
print('predicted:', yhat[0:4])
print('test set :', y_test[0:4].values)
```

```
predicted: [ 8982.359473    6123.81170413 11092.46584411  4225.74437131]
test set : [ 9549.  6229. 10245.  7295.]
```

We select the value of Alfa that minimizes the test error, for example, we can use a for loop.

[53]:
```
Rsqu_test = []
Rsqu_train = []
dummy1 = []
ALFA = 10 * np.array(range(0,1000))
for alfa in ALFA:
    RigeModel = Ridge(alpha=alfa)
    RigeModel.fit(x_train_pr, y_train)
    Rsqu_test.append(RigeModel.score(x_test_pr, y_test))
    Rsqu_train.append(RigeModel.score(x_train_pr, y_train))
```

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

[54]:
```
width = 12
height = 10
plt.figure(figsize=(width, height))

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

[54]:
```
<matplotlib.legend.Legend at 0x7f2f82a7bf28>
```

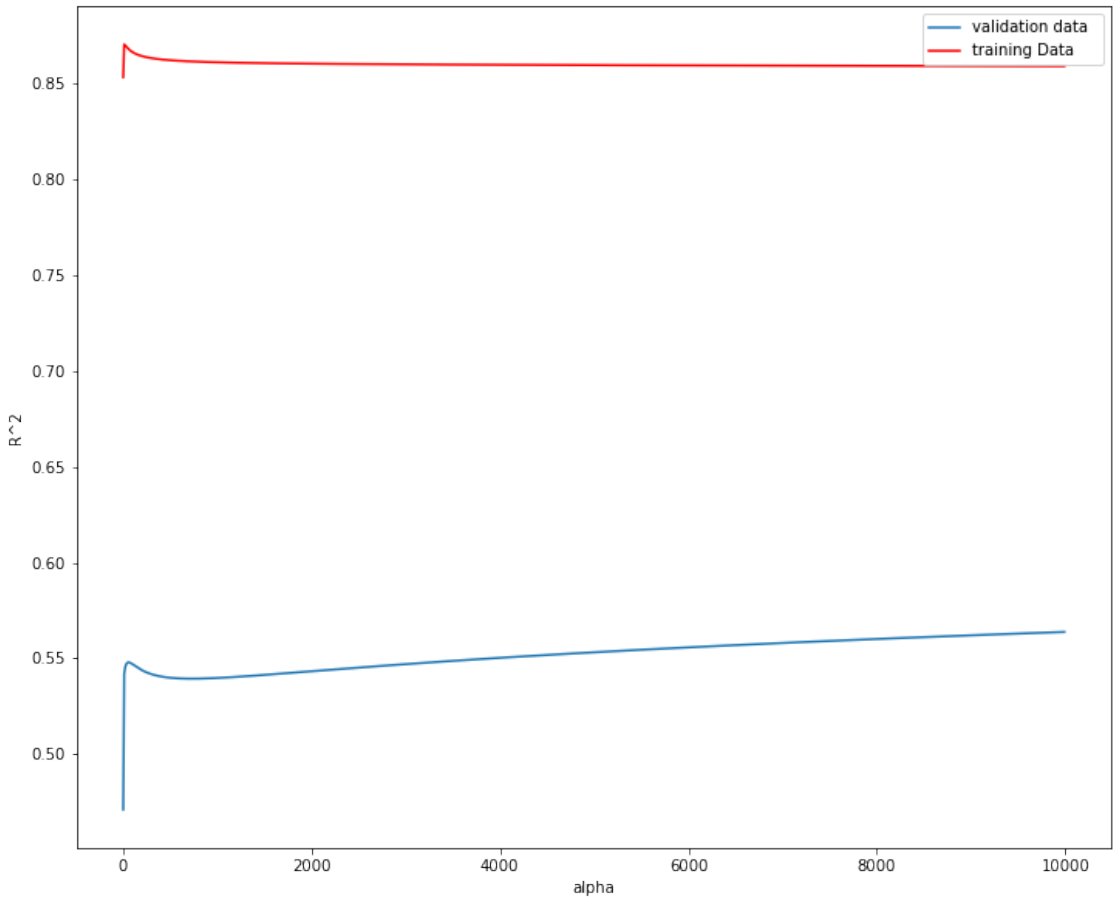


Figure 6:The blue line represents the R^2 of the test data, and the red line represents the R^2 of the training data. The x-axis represents the different values of Alfa

[85]:
```
Ridgemodel1=Ridge(alpha=10)
Ridgemodel1.fit(x_train_pr,y_train)
Ridgemodel1.score(x_test_pr,y_test)
```

[85]: 0.792821352363788

[86]:
```
Ridgemodel1.score(x_train_pr,y_train)
```

[86]: 0.8862832245101184

Part 4: Grid Search

The term Alfa 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.

[56]:
```
from sklearn.model_selection import GridSearchCV
```

We create a dictionary of parameter values:

[57]:
```
parameters1= [{'alpha': [0.001,0.1,1, 10, 100, 1000, 10000, 100000, 100000]}]
parameters1
```

[57]:
```
[{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 100000]}]
```

Create a ridge regions object:

[58]:
```
RR=Ridge()
RR
```

[58]:
```
Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001)
```

Create a ridge grid search object

[59]:
```
Grid1 = GridSearchCV(RR, parameters1,cv=4)
```

Fit the model

[60]:
```
Grid1.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']],␣
↪y_data)
```

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/sklearn/model_selection/_search.py:841: DeprecationWarning: The default
of the `iid` parameter will change from True to False in version 0.22 and will
be removed in 0.24. This will change numeric results when test-set sizes are
unequal.
  DeprecationWarning)
```

[60]:
```
GridSearchCV(cv=4, error_score='raise-deprecating',
       estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True,
max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid=[{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000,
100000]}],
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=0)
```

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:

[61]:
```
BestRR=Grid1.best_estimator_
BestRR
```

[61]:
```
Ridge(alpha=10000, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001)
```

We now test our model on the test data

[62]:
```
BestRR.score(x_test[['horsepower', 'curb-weight', 'engine-size',␣
↪'highway-mpg']], y_test)
```

[62]:
```
0.8411649831036152
```

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

[90]:
```
parameters2= [{'alpha': [0.001,0.1,1, 10, 100,␣
↪1000,10000,100000,100000],'normalize':[True,False]} ]
Grid2 = GridSearchCV(Ridge(), parameters2,cv=4)
Grid2.fit(x_data[['horsepower', 'curb-weight', 'engine-size',␣
↪'highway-mpg']],y_data)
```

```
/home/jupyterlab/conda/envs/python/lib/python3.6/site-
packages/sklearn/model_selection/_search.py:841: DeprecationWarning: The default
of the `iid` parameter will change from True to False in version 0.22 and will
be removed in 0.24. This will change numeric results when test-set sizes are
unequal.
  DeprecationWarning)
```

[90]:
```
GridSearchCV(cv=4, error_score='raise-deprecating',
       estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True,
max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid=[{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000,
100000], 'normalize': [True, False]}],
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=0)
```

[91]:
```
Grid2.best_estimator_
```

[91]:
```
Ridge(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=True, random_state=None, solver='auto', tol=0.001)
```

```
<p><a href="https://cocl.us/corsera_da0101en_notebook_bottom"><img src="https://s3-api.us-geo.
```