model-evaluation-and-refinement

February 3, 2019

```
<a href="http://cocl.us/DA0101EN_NotbookLink_Top">
     <img src="https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/
</a>
```

Data Analysis with Python

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

```
<li><a href="#ref1">Model Evaluation </a></li>
<li><a href="#ref2">Over-fitting, Under-fitting and Model Selection </a></li>
<li><a href="#ref3">Ridge Regression </a></li>
<li><a href="#ref4">Grid Search</a></li>
```

In [1]:
```
import pandas as pd
import numpy as np

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

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

First lets only use numeric data

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

Out[4]:

	Unnamed: 0	Unnamed: 0.1	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]
```

Libraries for plotting

In [5]:
```
%%capture
! pip install ipywidgets
```

In [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/lib/python3.6/site-packages/IPython/html.py:14: ShimWarning: The `IPython
  "`IPython.html.widgets` has moved to `ipywidgets`.", ShimWarning)
```

Functions for plotting

In [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()
```

In [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='Predi
    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:

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

drop price data in x data

In [10]:
```
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 [65]:
```
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, ran


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
```

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.

Question #1):

Use the function "train_test_split" to split up the data set 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_train_1" , "x_test_1", "y_train_1" and "y_test_1".

```
In [51]: # Write your code below and press Shift+Enter to execute
         x_train_1, x_test_1, y_train_1, y_test_1 = train_test_split(x_data, y_data, test_size=0


         print("number of test samples :", x_test_1.shape[0])
         print("number of training samples:",x_train_1.shape[0])

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

Double-click here for the solution.

Let's import LinearRegression from the module linear_model.

```
In [13]: from sklearn.linear_model import LinearRegression
```

We create a Linear Regression object:

```
In [14]: lre=LinearRegression()
```

we fit the model using the feature horsepower

```
In [66]: lre.fit(x_train[['horsepower']], y_train)

Out[66]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
                  normalize=False)
```

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

```
In [67]: lre.score(x_test[['horsepower']], y_test)

Out[67]: 0.707688374146705
```

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

```
In [68]: lre.score(x_train[['horsepower']], y_train)

Out[68]: 0.6449517437659684
```

Question #2):

Find the R^2 on the test data using 90% of the data for training data

```
In [81]: # Write your code below and press Shift+Enter to execute
         # 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=0.10,
         lre.fit(x_train1[['horsepower']],y_train1)
         lre.score(x_test1[['horsepower']],y_test1), lre.score(x_train1[['horsepower']], y_train

Out[81]: (0.7340722810055448, 0.6479796296498318)
```

Double-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

Lets import model_selection from the module cross_val_score.

```
In [23]: 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.

```
In [82]: 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:

```
In [25]: Rcross

Out[25]: array([0.7746232 , 0.51716687, 0.74785353, 0.04839605])
```

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

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

The mean of the folds are 0.522009915042119 and the standard deviation is 0.2911839444756029
```

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

```
In [27]: -1 * cross_val_score(lre,x_data[['horsepower']], y_data,cv=4,scoring='neg_mean_squared_

Out[27]: array([20254142.84026704, 43745493.26505169, 12539630.34014931,
                17561927.72247591])
```

Question #3):

Calculate the average R^2 using two folds, find the average R^2 for the second fold utilizing the horsepower as a feature :

```
In [28]: # Write your code below and press Shift+Enter to execute
         Rcross2 = cross_val_score(lre, x_data[['horsepower']], y_data, cv=2)
         Rcross2[1]

Out[28]: 0.443196127755029
```

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

```
In [30]: 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:

```
In [31]: yhat = cross_val_predict(lre,x_data[['horsepower']], y_data,cv=4)
         yhat[0:5]

Out[31]: 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 [83]: lr = LinearRegression()
         lr.fit(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y_train)

Out[83]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
                  normalize=False)
```

Prediction using training data:

```
In [84]: yhat_train = lr.predict(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-m
         yhat_train[0:5]

Out[84]: array([11927.70699817, 11236.71672034,  6436.91775515, 21890.22064982,
                16667.18254832])
```

Prediction using test data:

```
In [85]: yhat_test = lr.predict(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg
         yhat_test[0:5]

Out[85]: 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.

```
In [86]: import matplotlib.pyplot as plt
         %matplotlib inline
         import seaborn as sns
```

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

```
In [87]: Title = 'Distribution  Plot of  Predicted Value Using Training Data vs Training Data Di
         DistributionPlot(y_train, yhat_train, "Actual Values (Train)", "Predicted Values (Train

/home/jupyterlab/conda/lib/python3.6/site-packages/scipy/stats/stats.py:1713: FutureWarning: Usi
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

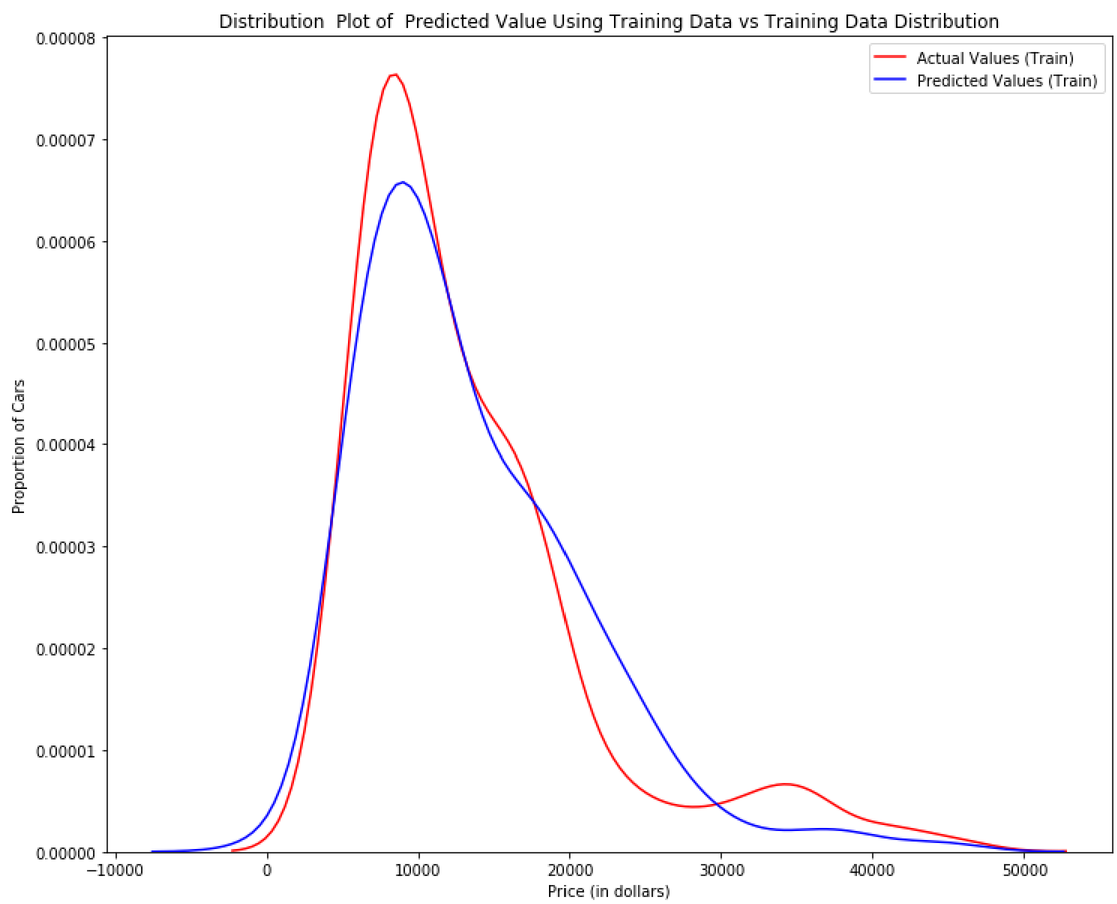
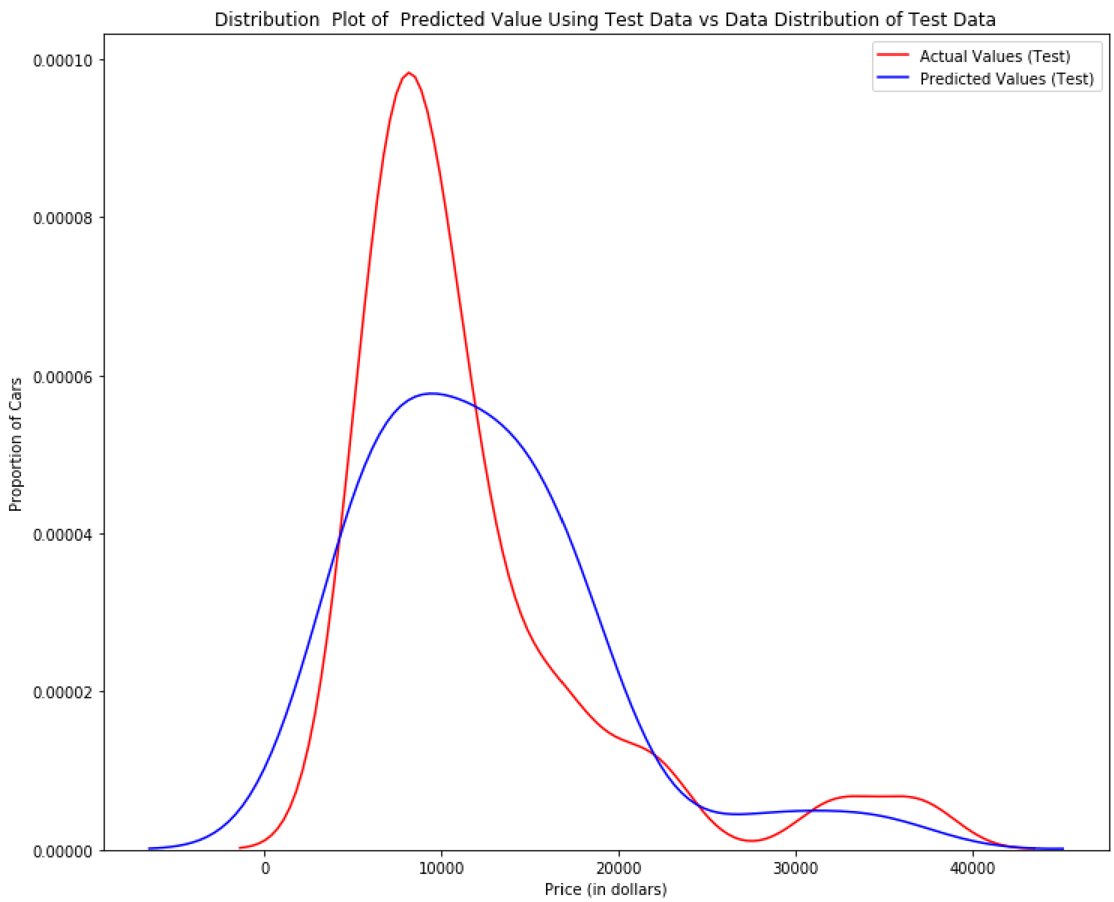


Figure 1: Plot of predicted values using the training data compared to the training data.

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.

```
In [89]: Title='Distribution  Plot of  Predicted Value Using Test Data vs Data Distribution of T
         DistributionPlot(y_test,yhat_test,"Actual Values (Test)","Predicted Values (Test)",Titl
```



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.

```
In [90]: 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:

```
In [92]: x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=0.45, ran
```

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

```
In [93]: pr = PolynomialFeatures(degree=5)
         x_train_pr = pr.fit_transform(x_train[['horsepower']])
         x_test_pr = pr.fit_transform(x_test[['horsepower']])
         pr
```

```
Out[93]: PolynomialFeatures(degree=5, include_bias=True, interaction_only=False)
```

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

```
In [94]: poly = LinearRegression()
         poly.fit(x_train_pr, y_train)
```

```
Out[94]: 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".

```
In [95]: yhat = poly.predict(x_test_pr)
         yhat[0:5]
```

```
Out[95]: array([ 6728.73877623,  7308.06173582, 12213.81078747, 18893.1290908 ,
                19995.81407813])
```

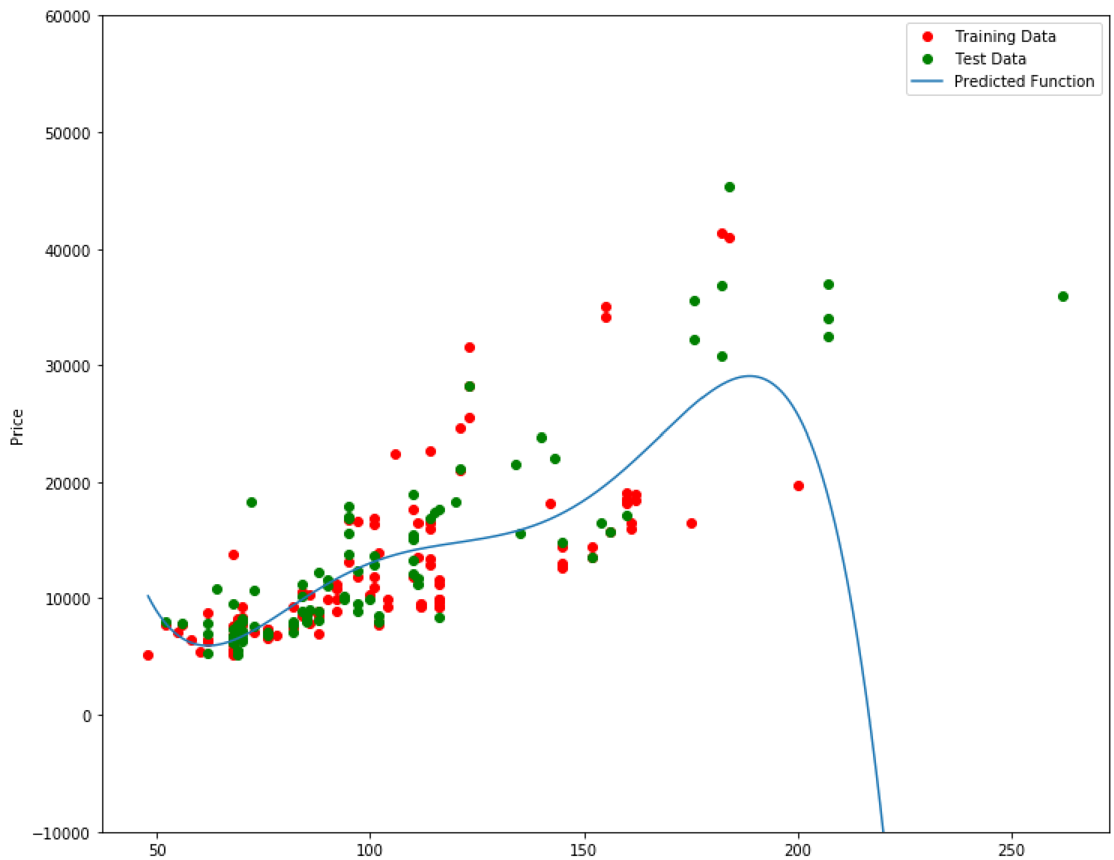
Let's take the first five predicted values and compare it to the actual targets.

```
In [96]: print("Predicted values:", yhat[0:4])
         print("True values:", y_test[0:4].values)
```

```
Predicted values: [ 6728.73877623  7308.06173582 12213.81078747 18893.1290908 ]
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 [97]: 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.

Rˆ2 of the training data:

```
In [98]: poly.score(x_train_pr, y_train)

Out[98]: 0.5567716902028981
```

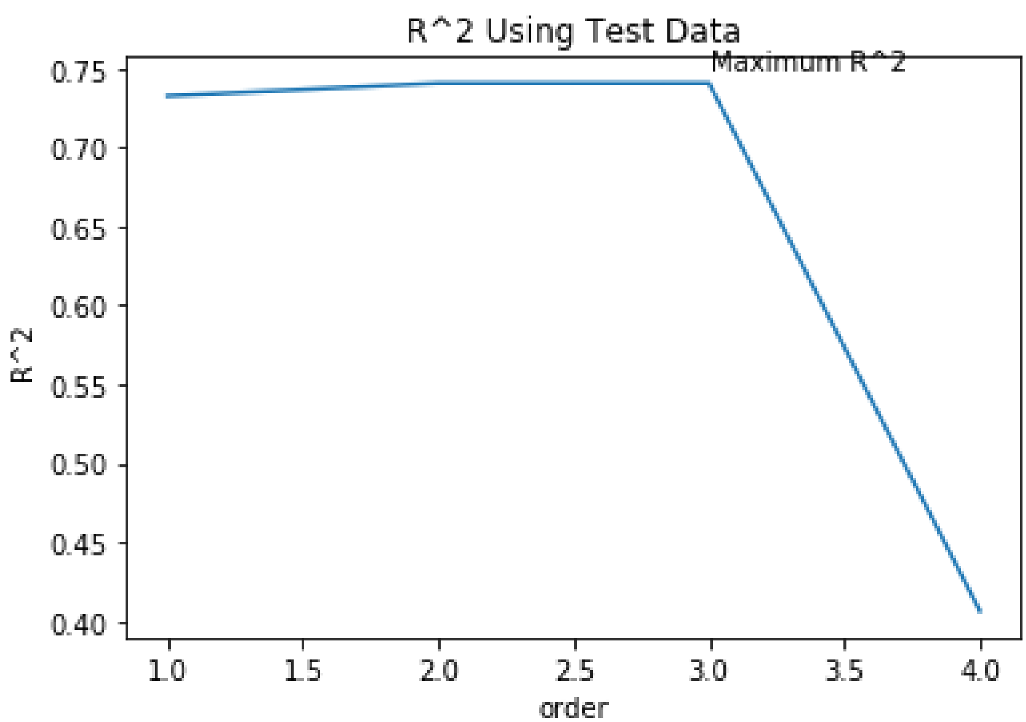
Rˆ2 of the test data:

```
In [99]: poly.score(x_test_pr, y_test)

Out[99]: -29.87162132967278
```

We see the Rˆ2 for the training data is 0.5567 while the Rˆ2 on the test data was -29.87. The lower the Rˆ2, the worse the model, a Negative Rˆ2 is a sign of overfitting.

Let's see how the Rˆ2 changes on the test data for different order polynomials and plot the results:

```
In [100]: 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[100]: 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.

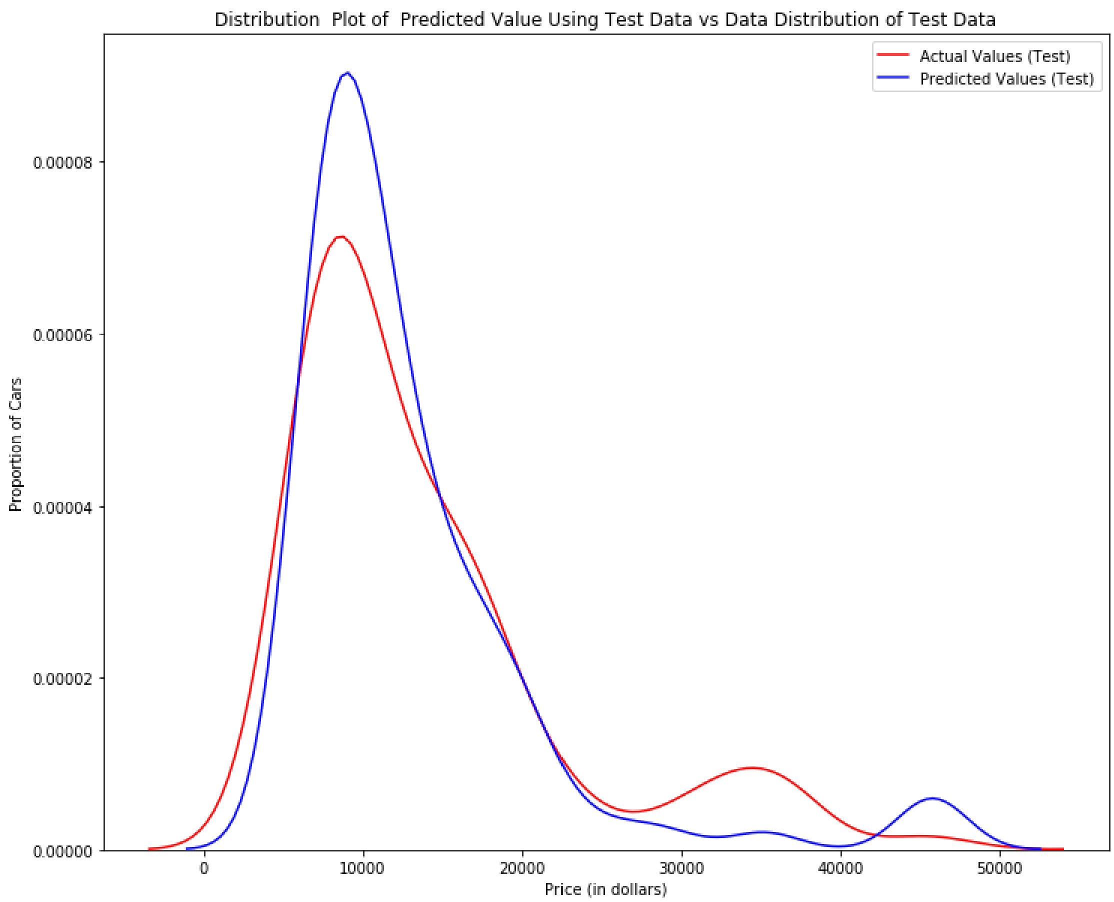
The following function will be used in the next section; please run the cell.

In [102]:
```
def f(order, test_data):
    x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, test_size=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, poly, p
```

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

In [117]:
```
#interact(f, order=(4, 6, 1), test_data=(0.05, 0.95, 0.05))
pr1 = PolynomialFeatures(degree=2)
x_train_pr1 = pr1.fit_transform(x_train[['horsepower', 'curb-weight', 'engine-size', '
x_test_pr1 = pr1.fit_transform(x_test[['horsepower', 'curb-weight', 'engine-size', 'hi
x_train_pr1.shape,  x_test_pr1.shape
poly1 = LinearRegression()
poly1.fit(x_train_pr1,y_train)
yhat_test1=poly1.predict(x_test_pr1)
Title='Distribution  Plot of  Predicted Value Using Test Data vs Data Distribution of
DistributionPlot(y_test, yhat_test1, "Actual Values (Test)", "Predicted Values (Test)"
```

```
/home/jupyterlab/conda/lib/python3.6/site-packages/scipy/stats/stats.py:1713: FutureWarning: Usi
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```



Question #4a):

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

Double-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" ?

Double-click here for the solution.

Question #4c):

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

Double-click here for the solution.

Question #4d):

Create a linear regression model "poly1" and train the object using the method "fit" using the polynomial features?

Double-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 output vs the test data?

Double-click here for the solution.

Question #4f):

Use the distribution plot to determine the two regions were the predicted prices are less accurate than the actual prices.

Double-click here for the solution.

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.

```
In [130]: pr=PolynomialFeatures(degree=2)
          x_train_pr=pr.fit_transform(x_train[['horsepower', 'curb-weight', 'engine-size', 'high
          x_test_pr=pr.fit_transform(x_test[['horsepower', 'curb-weight', 'engine-size', 'highwa
```

Let's import Ridge from the module linear models.

```
In [120]: from sklearn.linear_model import Ridge
```

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

```
In [121]: RigeModel=Ridge(alpha=0.1)
```

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

```
In [123]: RigeModel.fit(x_train_pr, y_train)
```

```
/home/jupyterlab/conda/lib/python3.6/site-packages/sklearn/linear_model/ridge.py:125: LinAlgWarn
Ill-conditioned matrix detected. Result is not guaranteed to be accurate.
Reciprocal condition number1.029716e-16
  overwrite_a=True).T
```

```
Out[123]: 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:

```
In [125]: yhat = RigeModel.predict(x_test_pr)
```

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

```
In [126]: print('predicted:', yhat[0:4])
          print('test set :', y_test[0:4].values)
```

```
predicted: [ 6567.83081933  9597.97151399 20836.22326843 19347.69543463]
test set : [ 6295. 10698. 13860. 13499.]
```

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

In [127]:
```
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

In [128]:
```
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()
```

Out[128]:
```
<matplotlib.legend.Legend at 0x7f0d30286518>
```

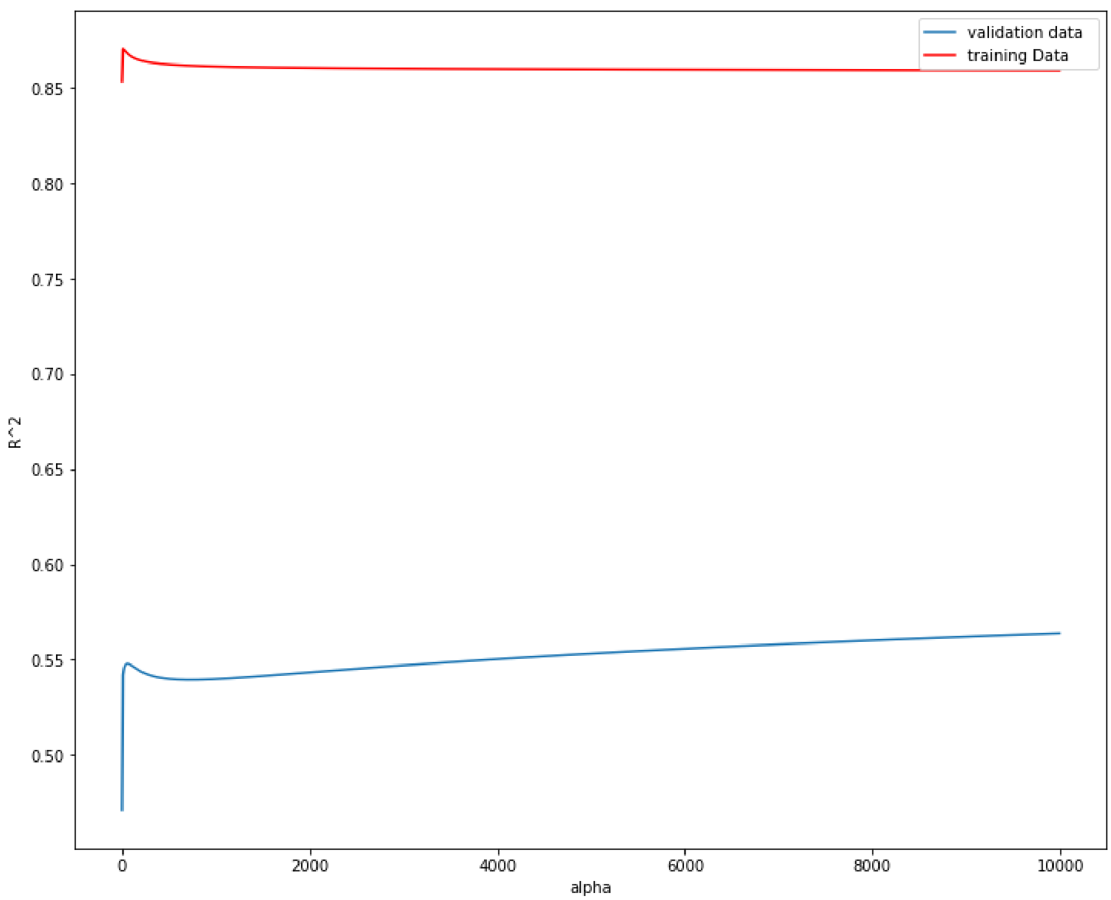


Figure 6:The blue line represents the R^2 of the test data, and the red line represents the R^2 of the training data. The x-axis represents the different values of Alfa

The red line in figure 6 represents the R^2 of the test data, as Alpha increases the R^2 decreases; therefore as Alfa increases the model performs worse on the test data. The blue line represents the R^2 on the validation data, as the value for Alfa increases the R^2 decreases.

Question #5):

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

```
In [133]: # Write your code below and press Shift+Enter to execute
          RigeModel1=Ridge(alpha=10)
          RigeModel1.fit(x_train_pr, y_train)
          #yhat = RigeModel1.predict(x_test_pr)
          #print('predicted:', yhat[0:4])
          #print('test set :', y_test[0:4].values)
          RigeModel1.score(x_test_pr, y_test)

Out[133]: 0.5418576440207072
```

Double-click here for the solution.

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.

```
In [135]: from sklearn.model_selection import GridSearchCV
```

We create a dictionary of parameter values:

```
In [136]: parameters1= [{'alpha': [0.001,0.1,1, 10, 100, 1000, 10000, 100000, 100000]}]
          parameters1

Out[136]: [{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 100000]}]
```

Create a ridge regions object:

```
In [137]: RR=Ridge()
          RR

Out[137]: 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

```
In [140]: Grid1 = GridSearchCV(RR, parameters1,cv=4)
```

Fit the model

```
In [141]: Grid1.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y_data)

/home/jupyterlab/conda/lib/python3.6/site-packages/sklearn/model_selection/_search.py:841: Depre
  DeprecationWarning)


Out[141]: 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:

```
In [142]: BestRR=Grid1.best_estimator_
          BestRR

Out[142]: 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

```
In [143]: BestRR.score(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y_te

Out[143]: 0.8411649831036149
```

Question #6):

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

```
In [150]: # Write your code below and press Shift+Enter to execute
          parameters2= [{'alpha': [0.001,0.1,1, 10, 100, 1000, 10000, 100000, 100000],'normalize
          Grid2 = GridSearchCV(RR, parameters2,cv=4)
          Grid2.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y_data)
          BestRR2=Grid2.best_estimator_
          BestRR2,BestRR2.score(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg

/home/jupyterlab/conda/lib/python3.6/site-packages/sklearn/model_selection/_search.py:841: Depre
  DeprecationWarning)


Out[150]: (Ridge(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=None,
              normalize=True, random_state=None, solver='auto', tol=0.001),
           0.840859719294301)
```

Double-click here for the solution.

Thank you for completing this notebook!

Get IBM Watson Studio free of charge!

```
<p><a href="http://cocl.us/NotebooksPython101bottom"><img src="https://s3-api.us-geo.objectstora
```

About the Authors:

This notebook was written by Mahdi Noorian PhD, Joseph Santarcangelo, Bahare Talayian, Eric Xiao, Steven Dong, Parizad, Hima Vsudevan and Fiorella Wenver.

Joseph Santarcangelo is a Data Scientist at IBM, and holds a PhD in Electrical Engineering. His research focused on using Machine Learning, Signal Processing, and Computer Vision to determine how videos impact human cognition. Joseph has been working for IBM since he completed his PhD.