Model Evaluation and Refinement

Estimated time needed: **30** minutes

Objectives

After completing this lab you will be able to:

- Evaluate and refine prediction models

Table of content

This dataset was hosted on IBM Cloud object click HERE for free storage.

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

# Import clean data
path = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/module_5_auto.csv'
df = pd.read_csv(path)
```

[2]:
```
df.to_csv('module_5_auto.csv')
```

First lets only use numeric data

[3]:
```
df=df._get_numeric_data()
df.head()
```

[3]:

	Unnamed: 0	Unnamed: 0.1	symboling	normalized-losses	wheel-base	length	width	height	curb-weight	engine-size	...	stroke	compression-ratio	horsepower	peak-rpm	city-mpg	highway-mpg	price	city-L/100km	diesel	gas
0	0	0	3	122	88.6	0.811148	0.890278	48.8	2548	130	...	2.68	9.0	111.0	5000.0	21	27	13495.0	11.190476	0	1
1	1	1	3	122	88.6	0.811148	0.890278	48.8	2548	130	...	2.68	9.0	111.0	5000.0	21	27	16500.0	11.190476	0	1
2	2	2	1	122	94.5	0.822681	0.909722	52.4	2823	152	...	3.47	9.0	154.0	5000.0	19	26	16500.0	12.368421	0	1
3	3	3	2	164	99.8	0.848630	0.919444	54.3	2337	109	...	3.40	10.0	102.0	5500.0	24	30	13950.0	9.791667	0	1
4	4	4	2	164	99.4	0.848630	0.922222	54.3	2824	136	...	3.40	8.0	115.0	5500.0	18	22	17450.0	13.055556	0	1

5 rows × 21 columns

Libraries for plotting

[4]:
```
%%capture
! pip install ipywidgets
```

[5]:
```
from ipywidgets import interact, interactive, fixed, interact_manual
```

Functions for plotting

[6]:
```
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()
```

[7]:
```
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:

[8]:
```
y_data = df['price']
```

drop price data in x data

[9]:
```
x_data=df.drop('price',axis=1)
```

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

[10]:
```
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 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".

[11]:
```
# Write your code below and press Shift+Enter to execute
```

▼Click here for the solution
```
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])
```

Let's import LinearRegression from the module linear_model.

[12]:
```
from sklearn.linear_model import LinearRegression
```

We create a Linear Regression object:

[13]:
```
lre=LinearRegression()
```

we fit the model using the feature horsepower

[14]:
```
lre.fit(x_train[['horsepower']], y_train)
```

[14]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

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

[15]:
```
lre.score(x_test[['horsepower']], y_test)
```

[15]:
```
0.3635875575078824
```

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

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

[16]:
```
0.6619724197515103
```

Question #2):

Find the R^2 on the test data using 40% of the data for training data

[17]:
```
# Write your code below and press Shift+Enter to execute
```

▼Click here for the solution
```
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)
```

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.

[18]:
```
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.

[19]:
```
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:

[20]:
```
Rcross
```

[20]:
```
array([0.7746232 , 0.51716687, 0.74785353, 0.04839605])
```

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

[21]:
```
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 0.2911839444756029
```

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

[22]:
```
-1 * cross_val_score(lre,x_data[['horsepower']], y_data,cv=4,scoring='neg_mean_squared_error')
```

[22]:
```
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 :

[23]:
```
# Write your code below and press Shift+Enter to execute
```

▼Click here for the solution

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

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 for testing and the other folds are used for training. First import the function:

[24]:
```
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:

[25]:
```
yhat = cross_val_predict(lre,x_data[['horsepower']], y_data,cv=4)
yhat[0:5]
```

[25]:
```
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.

[26]:
```
lr = LinearRegression()
lr.fit(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y_train)
```

[26]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

Prediction using training data:

[27]:
```
yhat_train = lr.predict(x_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])
yhat_train[0:5]
```

[27]:
```
array([ 7426.6731551 , 28323.75090803, 14213.38819709,  4052.34146983,
       34500.19124244])
```

Prediction using test data:

[28]:
```
yhat_test = lr.predict(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])
yhat_test[0:5]
```

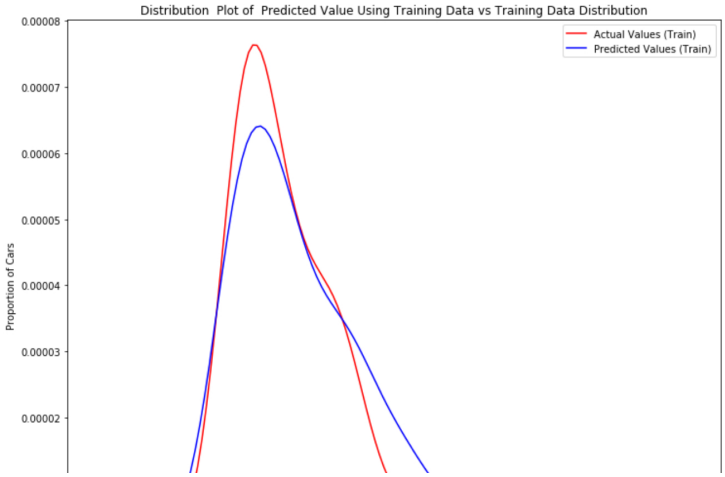
[28]:
```
array([11349.35089149,  5884.11059106, 11208.6928275 ,  6641.07786278,
       15565.79920282])
```

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

[29]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

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

[30]:
```
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)
```



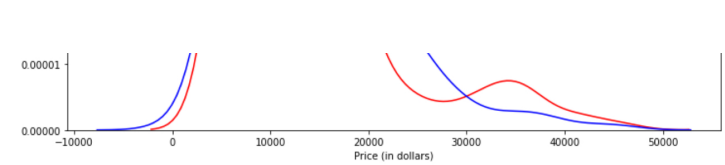
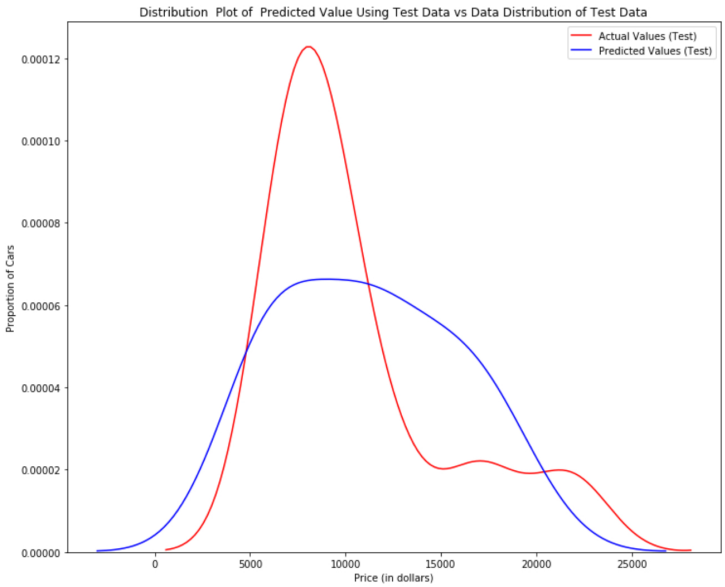


Figure 1: Plot of predicted values using the training data compared to the training data.

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.

[31]:
```
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.

[32]:
```
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 training and the rest for testing:

[33]:
```
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 'horse power'.

[34]:
```
pr = PolynomialFeatures(degree=5)
x_train_pr = pr.fit_transform(x_train[['horsepower']])
x_test_pr = pr.fit_transform(x_test[['horsepower']])
pr
```

[34]:
```
PolynomialFeatures(degree=5, include_bias=True, interaction_only=False,
                   order='C')
```

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

[35]:
```
poly = LinearRegression()
poly.fit(x_train_pr, y_train)
```

[35]:
```
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".

[36]:
```
yhat = poly.predict(x_test_pr)
yhat[0:5]
```

[36]:
```
array([ 6728.77492727,  7308.09738048, 12213.83912148, 18893.06269972,
       19995.73316497])
```

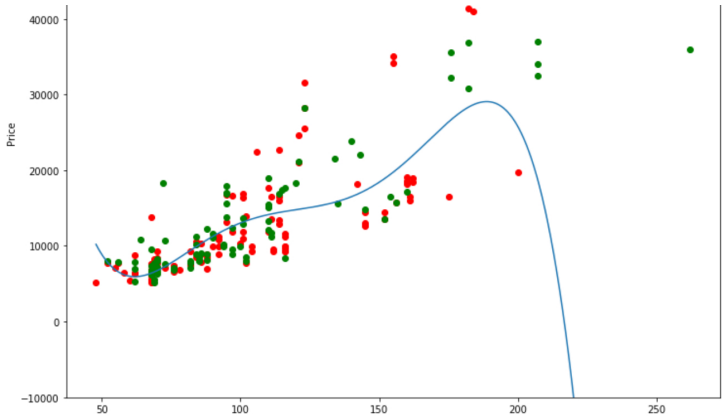
Let's take the first five predicted values and compare it to the actual targets.

[37]:
```
print("Predicted values:", yhat[0:4])
print("True values:", y_test[0:4].values)
```

```
Predicted values: [ 6728.77492727  7308.09738048 12213.83912148 18893.06269972]
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.

[38]:
```
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:

[39]:
```
poly.score(x_train_pr, y_train)
```

[39]:
```
0.5567716899817778
```

R^2 of the test data:

[40]:
```
poly.score(x_test_pr, y_test)
```

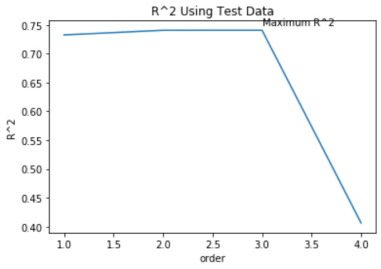
[40]:
```
-29.871838229908324
```

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:

[41]:
```
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 ')
```

[41]:
```
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.

[42]:
```
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.

[43]:
```
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…
```

[43]:
```
<function __main__.f(order, test_data)>
```

Question #4a):

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

[44]:
```
# Write your code below and press Shift+Enter to execute
```

▼Click here for the solution

```
pr1=PolynomialFeatures(degree=2)
```

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" ?

[45]:
```
# Write your code below and press Shift+Enter to execute
```

▼Click here for the solution

```
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']])
```

Question #4c):

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

[46]:
```
# Write your code below and press Shift+Enter to execute
```

▼Click here for the solution

```
x_train_pr1.shape #there are now 15 features
```

Question #4d):

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

[47]:
```
# Write your code below and press Shift+Enter to execute
```

▼Click here for the solution

```
poly1=LinearRegression().fit(x_train_pr1,y_train)
```

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?

[48]:
```
# Write your code below and press Shift+Enter to execute
```

▼Click here for the solution

```
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)
```

Question #4f):

Using the distribution plot above, explain in words about the two regions were the predicted prices are less accurate than the actual prices

[49]:
```
# Write your code below and press Shift+Enter to execute
```

▼Click here for the solution

```
#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 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.

[50]:
```
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.

[51]:
```
from sklearn.linear_model import Ridge
```

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

[52]:
```
RigeModel=Ridge(alpha=0.1)
```

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

[53]:
```
RigeModel.fit(x_train_pr, y_train)
```

```
C:\Users\Simran Garg\Anaconda3\lib\site-packages\sklearn\linear_model\ridge.py:147: LinAlgWarning: Ill-conditioned matrix (rcond=1.02972e-16): result may not be accurate.
  overwrite_a=True).T
```

[53]:
```
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:

[54]:
```
yhat = RigeModel.predict(x_test_pr)
```

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

[55]:
```
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 Alpha that minimizes the test error, for example, we can use a for loop.

[56]:
```
Rsqu_test = []
Rsqu_train = []
dummy1 = []
Alpha = 10 * np.array(range(0,1000))
for alpha in Alpha:
    RigeModel = Ridge(alpha=alpha)
    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

[57]:
```
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()
```

[57]:
```
<matplotlib.legend.Legend at 0x681b538278>
```

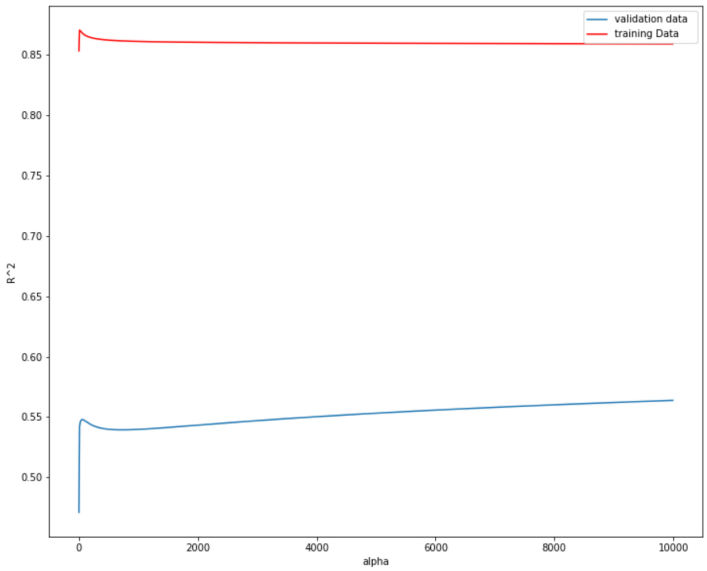


Figure 6:The blue line represents the R^2 of the validation data, and the red line represents the R^2 of the training data. The x-axis represents the different values of Alpha.

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 6 represents the R^2 of the training data.
As Alpha increases the R^2 decreases.
Therefore as Alpha increases the model performs worse on the training data.

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

Question #5):

Perform Ridge regression 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.

[58]:
```
# Write your code below and press Shift+Enter to execute
```

▼Click here for the solution
```
RigeModel = Ridge(alpha=10)
RigeModel.fit(x_train_pr, y_train)
RigeModel.score(x_test_pr, y_test)
```

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.

[59]:
```
from sklearn.model_selection import GridSearchCV
```

We create a dictionary of parameter values:

[60]:
```
parameters1= [{'alpha': [0.001,0.1,1, 10, 100, 1000, 10000, 100000, 100000]}]
parameters1
```

[60]:
```
[{'alpha': [0.001, 0.1, 1, 10, 100, 1000, 10000, 100000, 100000]}]
```

Create a ridge regions object:

[61]:
```
RR=Ridge()
RR
```

[61]:
```
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

[62]:
```
Grid1 = GridSearchCV(RR, parameters1,cv=4)
```

Fit the model

[63]:
```
Grid1.fit(x_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y_data)
```

```
C:\Users\Simran Garg\Anaconda3\lib\site-packages\sklearn\model_selection\_search.py:813: 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)
```

[63]:
```
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),
             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=False,
             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:

[64]:
```
BestRR=Grid1.best_estimator_
BestRR
```

[64]:
```
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

[65]:
```
BestRR.score(x_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y_test)
```

[65]:
```
0.8411649831036149
```

Question #6):

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

[66]:
```
# Write your code below and press Shift+Enter to execute
```

▼Click here for the solution

```
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)
Grid2.best_estimator_
```

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
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