**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 Contents**

* [Model Evaluation](https://jupyterlab-4-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/akulasamson/files/labs/DA0101EN/https%3A/%23ref1?_xsrf=2%7C34313e40%7C6238ca29ce5b2e428c5db70dff082614%7C1630087457)
* [Over-fitting, Under-fitting and Model Selection](https://jupyterlab-4-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/akulasamson/files/labs/DA0101EN/https%3A/%23ref2?_xsrf=2%7C34313e40%7C6238ca29ce5b2e428c5db70dff082614%7C1630087457)
* [Ridge Regression](https://jupyterlab-4-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/akulasamson/files/labs/DA0101EN/https%3A/%23ref3?_xsrf=2%7C34313e40%7C6238ca29ce5b2e428c5db70dff082614%7C1630087457)
* [Grid Search](https://jupyterlab-4-labs-prod-jupyterlab-us-east-0.labs.cognitiveclass.ai/user/akulasamson/files/labs/DA0101EN/https%3A/%23ref4?_xsrf=2%7C34313e40%7C6238ca29ce5b2e428c5db70dff082614%7C1630087457)

This dataset was hosted on IBM Cloud object. Click [HERE](https://cocl.us/DA101EN_object_storage?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-Channel-SkillsNetworkCoursesIBMDeveloperSkillsNetworkDA0101ENSkillsNetwork20235326-2021-01-01) for free storage.

[ ]:



**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)

[ ]:



df.to\_csv('module\_5\_auto.csv')

First, let's only use numeric data:

[ ]:



df**=**df.\_get\_numeric\_data()

df.head()

Libraries for plotting:

[ ]:



**%%**capture

**!** pip install ipywidgets

[ ]:



**from** ipywidgets **import** interact, interactive, fixed, interact\_manual

**Functions for Plotting**

[ ]:



**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()

[ ]:



**def** PollyPlot(xtrain, xtest, y\_train, y\_test, lr,poly\_transform):

width **=** 12

height **=** 10

plt.figure(figsize**=**(width, height))

*#training data*

*#testing data*

*# lr: linear regression object*

*#poly\_transform: polynomial transformation object*

xmax**=**max([xtrain.values.max(), xtest.values.max()])

​

xmin**=**min([xtrain.values.min(), xtest.values.min()])

​

x**=**np.arange(xmin, xmax, 0.1)

​

​

plt.plot(xtrain, y\_train, 'ro', label**=**'Training Data')

plt.plot(xtest, y\_test, 'go', label**=**'Test Data')

plt.plot(x, lr.predict(poly\_transform.fit\_transform(x.reshape(**-**1, 1))), label**=**'Predicted Function')

plt.ylim([**-**10000, 60000])

plt.ylabel('Price')

plt.legend()

**Part 1: Training and Testing**

An important step in testing your model is to split your data into training and testing data. We will place the target data **price** in a separate dataframe **y\_data**:

[ ]:



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])

​

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.

**Question #1):**

**Use the function "train\_test\_split" to split up the dataset 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\_train1" , "x\_test1", "y\_train1" and "y\_test1".**

[ ]:



*# 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**.

[ ]:



**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)

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

[ ]:



lre.score(x\_test[['horsepower']], y\_test)

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)

**Question #2):**

**Find the R^2 on the test data using 40% of the dataset for testing.**

[ ]:



*# 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**

Let's import **model\_selection** from the module **cross\_val\_score**.

[ ]:



**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

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())

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')

**Question #3):**

**Calculate the average R^2 using two folds, then find the average R^2 for the second fold utilizing the "horsepower" feature:**

[ ]:



*# 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, 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]

**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.

[ ]:



lr **=** LinearRegression()

lr.fit(x\_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y\_train)

Prediction using training data:

[ ]:



yhat\_train **=** lr.predict(x\_train[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])

yhat\_train[0:5]

Prediction using test data:

[ ]:



yhat\_test **=** lr.predict(x\_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']])

yhat\_test[0:5]

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.

Comparing Figure 1 and Figure 2, it is evident that the distribution of the test data in Figure 1 is much better at fitting the data. This difference in Figure 2 is apparent in the range of 5000 to 15,000. This is where the shape of the distribution is extremely different. Let's see if polynomial regression also exhibits a drop in the prediction accuracy when analysing the test dataset.

[ ]:



**from** sklearn.preprocessing **import** PolynomialFeatures

**Overfitting**

Overfitting occurs when the model fits the noise, but not the underlying process. Therefore, when testing your model using the test set, your model does not perform as well since it is modelling noise, not the underlying process that generated the relationship. Let's create a degree 5 polynomial model.

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

[ ]:



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

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

[ ]:



poly **=** LinearRegression()

poly.fit(x\_train\_pr, y\_train)

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]

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)

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)

R^2 of the test data:

[ ]:



poly.score(x\_test\_pr, y\_test)

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 ')

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))

**Question #4a):**

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

[ ]:



*# 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".**

[ ]:



*# 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".**

[ ]:



*# 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". Train the object using the method "fit" using the polynomial features.**

[ ]:



*# 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 test output vs. the actual test data.**

[ ]:



*# 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, describe (in words) the two regions where the predicted prices are less accurate than the actual prices.**

[ ]:



*# 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**

**Did you know? IBM Watson Studio lets you build and deploy an AI solution, using the best of open source and IBM software and giving your team a single environment to work in.**[**Learn more here.**](https://cocl.us/ibm_watson_studio_infobox)

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)

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)

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.

[ ]:



**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)

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

[ ]:



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()

**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.

**Question #5):**

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

[ ]:



*# 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 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

Create a Ridge regression object:

[ ]:



RR**=**Ridge()

RR

Create a ridge grid search object:

[ ]:



Grid1 **=** GridSearchCV(RR, parameters1,cv**=**4, iid**=None**)

In order to avoid a deprecation warning due to the iid parameter, we set the value of iid to "None".

Fit the model:

[ ]:



Grid1.fit(x\_data[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y\_data)

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

We now test our model on the test data:

[ ]:



BestRR.score(x\_test[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']], y\_test)

**Question #6):**

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

[ ]:



*# 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\_