Troubleshooting errors

DIV/0!

Reason:Your formula or function is asking the spreadsheet to divide something that is impossible to divide. Its value is either '0' or blank.
Solution1: Divide by another number or add a value to the blank cell
Solution2: You could use the QUOTIENT function in =IF(B3,QUOTIENT(B2,B3),0) to divide B2 by B3, and make the solution 0 if the dividing parameter (B3) is 0

N/A

Reason:It tells you that the numbers in your formula or function can’t be found by the spreadsheet application
Solution1:Simply add the data being looked up in the formula that is causing the error

NAME?

Reason: It means that your spreadsheet needs help understanding your formula or function
Solution1: check the spelling and make sure you use the full name of any formula

NULL!

Reason: You have specified two or more ranges that are supposed to overlap, but don’t actually intersect
Solution1: Use a colon (:) to separate the first cell from the last cell when you refer to a continuous range of cells in a formula
Solution2: Use a comma (,) between ranges to tell your spreadsheet that they are actually separate.E.G; =count(G1:G10,B1:B10)

NUM!

Reason: The spreadsheet can’t perform a calculation as written.
Solution: Return to your formula and double-check it

REF!

Reason: It means that there are spaces, characters, or text in your formula or function, which should actually be numbers Your formula or function is referencing a cell that is not valid.
Solution: Make sure you are giving your formula or function only valid cells by double-checking the reference

VALUE!

Reason: It means that there are spaces, characters, or text in your formula or function, which should actually be numbers
Solution: You can solve this problem by making sure they are either eliminated or numerical

Importing Spreadsheet file using URL.

=IMPORTDATA function and paste the URL into the parentheses. Remember to put the URL in quotes!
IMPORTDATA('URL')

Field length enables an analyst to determine how many characters can be typed into a spreadsheet field. An analyst can use field length as part of the data-validation process

VLOOKUP searches for a value in a column in order to return a corresponding piece of information

Sort sheet sorts all of the data in a spreadsheet by a specific sorted column. Data across rows is kept together during the sort.

```
A data analyst sorts a spreadsheet range between cells F19
and G82. They sort in ascending order by the second
column, Column G.


The first part of the function sorts the data in the
specified range. The 2 represents the second column. And a
TRUE statement sorts in ascending order


=SORT(F19:G82, 2, TRUE)
```

Excel DATEVALUE function - change text to date

```
=DATEVALUE(A1)
Where A1 is a cell with a date stored as a text string
```

Because the Excel DATEVALUE function converts a text date to a serial number, you will have to make that number look like a date by [applying the Date format]to it.

Convert string to number in Excel

```
=VALUE(A2)
To convert a column of text values, you enter the formula
in the first cell, and drag the fill handle to copy the
formula down the column
```

Combine text from two or more cells into one cell

```
CONCAT(A2, " Family")
Adds 'family' to cell A2

Use commas to separate the cells you are combining and use
quotation marks to add spaces, commas, or other text
```

The function =LEN(B8) will display the number of characters in cell B8. The LEN function returns the length of a string of text by counting the number of characters it contains

VLOOKUP, or Vertical Lookup
searches for a certain value in a spreadsheet column and returns

a corresponding piece of information from the row in which the searched value is found

VLOOKUP(search_key, range, index, [is_sorted])

search_key

- The value to search for.
- For example, 42, "Cats", or I24.

range

- The range to consider for the search.
- The first column in the range is searched for the key specified in search_key.

index

- The column index of the value to be returned, where the first column in range is numbered 1.
- If index is not between 1 and the number of columns in range, #VALUE! is returned.

is_sorted

- Indicates whether the column to be searched (the first column of the specified range) is sorted. TRUE by default.
- It's recommended to set is_sorted to FALSE. If set to FALSE, an exact match is returned. If there are multiple matching values, the content of the cell corresponding to the first value found is returned, and #N/A is returned if no such value is found.
- If is_sorted is TRUE or omitted, the nearest match (less than or equal to the search key) is returned. If all values in the

search column are greater than the search key, #N/A is returned.

When do you need to use VLOOKUP?

- Populating data in a spreadsheet
- Merging data from one spreadsheet with data in another

To change the text string in spreadsheet cell F8 to a numerical value, the correct syntax is =VALUE(F8)

Functions with multiple conditions

	A	B	C
1	Expense	Price	Date
2	Fuel	$48.00	12/14/2020
3	Food	$12.34	12/14/2020
4	Taxi	$21.57	12/14/2020
5	Coffee	$2.50	12/15/2020
6	Fuel	$36.00	12/15/2020
7	Taxi	$15.88	12/15/2020
8	Coffee	$4.15	12/15/2020
9	Food	$6.75	12/15/2020

SUMIF

The basic syntax of a SUMIF function is

```
=SUMIF(range, criterion, sum_range)
```

The first range is where the function will search for the condition that you have set. The criterion is the condition you are applying

and the sum_range is the range of cells that will be included in the calculation.

SUMIFS

SUMIF and SUMIFS are very similar, but SUMIFS can include multiple conditions.

```
=SUMIFS(sum_range, criteria_range1, criterion1,
[criteria_range2, criterion2, ...])
```

The square brackets let you know that this is optional. The ellipsis at the end of the statement lets you know that you can have as many repetition of these parameters as needed. For example, if you wanted to calculate the sum of the fuel costs for one date in this table, you could create a SUMIF statement with multiple conditions, like this:

```
=SUMIFS(B1:B9,A1:A9,"fuel",C1:C9,"12/15/2020")
```

COUNTIF

```
=COUNTIF(range, criterion)
```

Just like SUMIF, you set the range and then the condition that needs to be met. For example, if you wanted to count the number of times Food came up in the Expenses column, you could use a COUNTIF function like this:

```
=COUNTIF(A1:A9,"Food")
```

COUNTIFS

```
=COUNTIFS(criteria_range1, criterion1, [criteria_range2, criterion2, ...])
```

The criteria_range and criterion are in the same order, and you can add more conditions to the end of the function. So, if you wanted to find the number of times Coffee appeared in the Expenses column on 12/15/2020, you could use COUNTIFS to apply those conditions, like this:

```
=COUNTIFS(A1:A9,"Coffee",C1:C9,"12/15/2020")
```

An array is a collection of values in spreadsheet cells.

Model Evaluation and Refinement

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()
```

Code V1

```
#### Libraries for plotting
from ipywidgets import interact, interactive, fixed,
interact_manual

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()
```

Part 1: Training and Testing

An important step in testing your model is to split your data into training and testing data.

Code V2

```
#### Libraries for plotting
from ipywidgets import interact, interactive, fixed,
interact_manual

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()

#### 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)

#### 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.

print("number of test samples :", x_test.shape[0])
print("number of training samples:",x_train.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)
#### 0.3635875575078824

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

lre.score(x_train[['horsepower']], y_train)
#### 0.6619724197515103
```

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

Code V3

```
#### Libraries for plotting
from ipywidgets import interact, interactive, fixed,
interact_manual

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()

#### 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)

#### 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.

print("number of test samples :", x_test.shape[0])
print("number of training samples:",x_train.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)
#### 0.3635875575078824

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

lre.score(x_train[['horsepower']], y_train)
#### 0.6619724197515103

#### 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')

#### 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

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

Code V4

```
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()

#### Libraries for plotting
from ipywidgets import interact, interactive, fixed,
interact_manual

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()

#### 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)

#### 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.

print("number of test samples :", x_test.shape[0])
print("number of training samples:",x_train.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)
#### 0.3635875575078824

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

lre.score(x_train[['horsepower']], y_train)
#### 0.6619724197515103

#### 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')

#### 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]

#### 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

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)


#### 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)
```

```
####  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

#### 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
```

Part 3: Ridge Regression

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.

Code V5

```
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()

#### Libraries for plotting
```

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

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()

#### 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)

#### 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.

print("number of test samples :", x_test.shape[0])
print("number of training samples:",x_train.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)
#### 0.3635875575078824

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

lre.score(x_train[['horsepower']], y_train)
#### 0.6619724197515103

#### 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')

#### 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]

#### 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

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)


#### 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)

####  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

#### 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

####  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
```

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

Code V6

```
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()

#### Libraries for plotting
from ipywidgets import interact, interactive, fixed,
interact_manual

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()

#### 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)

#### 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.

print("number of test samples :", x_test.shape[0])
print("number of training samples:",x_train.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)
#### 0.3635875575078824

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

lre.score(x_train[['horsepower']], y_train)
#### 0.6619724197515103

#### 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')

#### 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]

#### 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

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)


#### 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)

####  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
```

```
#### 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

####  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

####  Let's import **GridSearchCV** from the module **model_selection**
```

```
from sklearn.model_selection import GridSearchCV

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)
```