Model Development

Estimated time needed: **30** minutes

Objectives

After completing this lab you will be able to:

- Develop prediction models

In this section, we will develop several models that will predict the price of the car using the variables or features. This is just an estimate but should give us an objective idea of how much the car should cost.

Some questions we want to ask in this module

- Do I know if the dealer is offering fair value for my trade-in?
- Do I know if I put a fair value on my car?

In data analytics, we often use **Model Development** to help us predict future observations from the data we have.

A model will help us understand the exact relationship between different variables and how these variables are used to predict the result.

Setup

Import libraries:

In [ ]:
```
#install specific version of libraries used in lab
#! mamba install pandas==1.3.3-y
#! mamba install numpy=1.21.2-y
#! mamba install sklearn=0.20.1-y
```

In [1]:
```
import piplite
await piplite.install('seaborn')
```

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Load the data and store it in dataframe `df`:

In [3]:
```
from pyodide.http import pyfetch

async def download(url, filename):
    response = await pyfetch(url)
    if response.status == 200:
        with open(filename, "wb") as f:
            f.write(await response.bytes())
```

In [4]:
```
file_path= "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/automob

await download(file_path, "usedcars.csv")
file_name="usedcars.csv"
```

In [5]:
```
df = pd.read_csv(file_name)
df.head()
```

Out[5]:

	symboling	normalized-losses	make	aspiration	num-of-doors	body-style	drive-wheels	engine-location	wheel-base	length	...	compression-ratio	horsepower	peak-rpm	city-mpg	highway-mpg	price
0	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	...	9.0	111.0	5000.0	21	27	13495.0
1	3	122	alfa-romero	std	two	convertible	rwd	front	88.6	0.811148	...	9.0	111.0	5000.0	21	27	16500.0
2	1	122	alfa-romero	std	two	hatchback	rwd	front	94.5	0.822681	...	9.0	154.0	5000.0	19	26	16500.0
3	2	164	audi	std	four	sedan	fwd	front	99.8	0.848630	...	10.0	102.0	5500.0	24	30	13950.0
4	2	164	audi	std	four	sedan	4wd	front	99.4	0.848630	...	8.0	115.0	5500.0	18	22	17450.0

5 rows × 29 columns

> Note: This version of the lab is working on JupyterLite, which requires the dataset to be downloaded to the interface.While working on the downloaded version

> of this notebook on their local machines(Jupyter Anaconda), the learners can simply skip the steps above, and simply use the URL directly in the pandas.read_csv() function. You can uncomment and run the statements in the cell below.

In [ ]:
```
#filepath = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DA0101EN-SkillsNetwork/labs/Data%20files/automo
#df = pd.read_csv(filepath, header=None)
```

1. Linear Regression and Multiple Linear Regression

Linear Regression

One example of a Data Model that we will be using is:

Simple Linear Regression
Simple Linear Regression is a method to help us understand the relationship between two variables:

- The predictor/independent variable (X)
- The response/dependent variable (that we want to predict)(Y)

The result of Linear Regression is a **linear function** that predicts the response (dependent) variable as a function of the predictor (independent) variable.

$$Y: Response\ Variable$$
$$X: Predictor\ Variables$$

Linear Function

$$Yhat = a + bX$$

- a refers to the **intercept** of the regression line, in other words: the value of Y when X is 0
- b refers to the **slope** of the regression line, in other words: the value with which Y changes when X increases by 1 unit

Let's load the modules for linear regression:

In [6]:
```
from sklearn.linear_model import LinearRegression
```

Create the linear regression object:

In [7]:
```
lm = LinearRegression()
lm
```

Out[7]:
```
LinearRegression()
```

How could "highway-mpg" help us predict car price?

For this example, we want to look at how highway-mpg can help us predict car price. Using simple linear regression, we will create a linear function with "highway-mpg" as the predictor variable and the "price" as the response variable.

In [8]:
```
X = df[['highway-mpg']]
Y = df['price']
```

Fit the linear model using highway-mpg:

In [9]:
```
lm.fit(X,Y)
```

Out[9]:
```
LinearRegression()
```

We can output a prediction:

In [10]:
```
Yhat=lm.predict(X)
Yhat[0:5]
```

Out[10]:
```
array([16236.50464347, 16236.50464347, 17058.23802179, 13771.3045085 ,
       20345.17153508])
```

What is the value of the intercept (a)?

In [11]:
```
lm.intercept_
```

Out[11]:
```
38423.30585815743
```

What is the value of the slope (b)?

In [12]:
```
lm.coef_
```

Out[12]:
```
array([-821.73337832])
```

What is the final estimated linear model we get?

As we saw above, we should get a final linear model with the structure:

$$Yhat = a + bX$$

Plugging in the actual values we get:

Price = 38423.31 - 821.73 x **highway-mpg**

Question #1 a):

Create a linear regression object called "lm1".

In [13]:
```
# Write your code below and press Shift+Enter to execute
lm1 = LinearRegression()
```

▸ Click here for the solution

Question #1 b):

Train the model using "engine-size" as the independent variable and "price" as the dependent variable?

In [14]:
```
# Write your code below and press Shift+Enter to execute
lm1.fit(df[["engine-size"]],df[["price"]])
```

Out[14]:
```
LinearRegression()
```

▸ Click here for the solution

Question #1 c):

Find the slope and intercept of the model.

Slope

In [15]:
```
# Write your code below and press Shift+Enter to execute
lm1.intercept_
```

Out[15]:
```
array([-7963.33890628])
```

Intercept

In [16]:
```
# Write your code below and press Shift+Enter to execute
lm1.coef_
```

Out[16]:
```
array([[166.86001569]])
```

▸ Click here for the solution

Question #1 d):

What is the equation of the predicted line? You can use x and yhat or "engine-size" or "price".

In [18]:
```
# Write your code below and press Shift+Enter to execute
yhat = lm1.predict(df[["engine-size"]])
yhat[0:5]
```

Out[18]:
```
array([[13728.4631336 ],
       [13728.4631336 ],
       [17399.38347881],
       [10224.40280408],
       [14729.62322775]])
```

▸ Click here for the solution

Multiple Linear Regression

What if we want to predict car price using more than one variable?

If we want to use more variables in our model to predict car price, we can use **Multiple Linear Regression**. Multiple Linear Regression is very similar to Simple Linear Regression, but this method is used to explain the relationship between one continuous response (dependent) variable and **two or more** predictor (independent) variables. Most of the real-world regression models involve multiple predictors. We will illustrate the structure by using four predictor variables, but these results can generalize to any integer:

$$
\begin{aligned}
Y &: Response\ Variable \\
X_1 &: Predictor\ Variable\ 1 \\
X_2 &: Predictor\ Variable\ 2 \\
X_3 &: Predictor\ Variable\ 3 \\
X_4 &: Predictor\ Variable\ 4
\end{aligned}
$$

$$
\begin{aligned}
a &: intercept \\
b_1 &: coefficients\ of\ Variable\ 1 \\
b_2 &: coefficients\ of\ Variable\ 2 \\
b_3 &: coefficients\ of\ Variable\ 3 \\
b_4 &: coefficients\ of\ Variable\ 4
\end{aligned}
$$

The equation is given by:

$$Yhat = a + b_1 X_1 + b_2 X_2 + b_3 X_3 + b_4 X_4$$

From the previous section we know that other good predictors of price could be:

- Horsepower
- Curb-weight
- Engine-size
- Highway-mpg

Let's develop a model using these variables as the predictor variables.

In [19]:
```
Z = df[['horsepower', 'curb-weight', 'engine-size', 'highway-mpg']]
```

Fit the linear model using the four above-mentioned variables.

In [20]:
```
lm.fit(Z, df['price'])
```

Out[20]:
```
LinearRegression()
```

What is the value of the intercept(a)?

In [21]:
```
lm.intercept_
```

Out[21]:
```
-15806.62462632922
```

What are the values of the coefficients (b1, b2, b3, b4)?

In [22]:
```
lm.coef_
```

Out[22]:
```
array([53.49574423,  4.70770099, 81.53026382, 36.05748882])
```

What is the final estimated linear model that we get?

As we saw above, we should get a final linear function with the structure:

$$Yhat = a + b_1 X_1 + b_2 X_2 + b_3 X_3 + b_4 X_4$$

What is the linear function we get in this example?

Price = -15678.742628061467 + 52.65851272 x **horsepower** + 4.69878948 x **curb-weight** + 81.95906216 x **engine-size** + 33.58258185 x **highway-mpg**

Question #2 a):

Create and train a Multiple Linear Regression model "lm2" where the response variable is "price", and the predictor variable is "normalized-losses" and "highway-mpg".

In [23]:
```
# Write your code below and press Shift+Enter to execute
lm2 = LinearRegression()
lm2.fit(df[["normalized-losses","highway-mpg"]],df["price"])
```

Out[23]:
```
LinearRegression()
```

▸ Click here for the solution

Question #2 b):

Find the coefficient of the model.

In [24]:
```
# Write your code below and press Shift+Enter to execute
lm2.coef_
```

Out[24]:
```
array([   1.49789586, -820.45434016])
```

▸ Click here for the solution

2. Model Evaluation Using Visualization

Now that we've developed some models, how do we evaluate our models and choose the best one? One way to do this is by using a visualization.

Import the visualization package, seaborn:

In [26]:
```
# import the visualization package: seaborn
import seaborn as sns
%matplotlib inline
```

Regression Plot

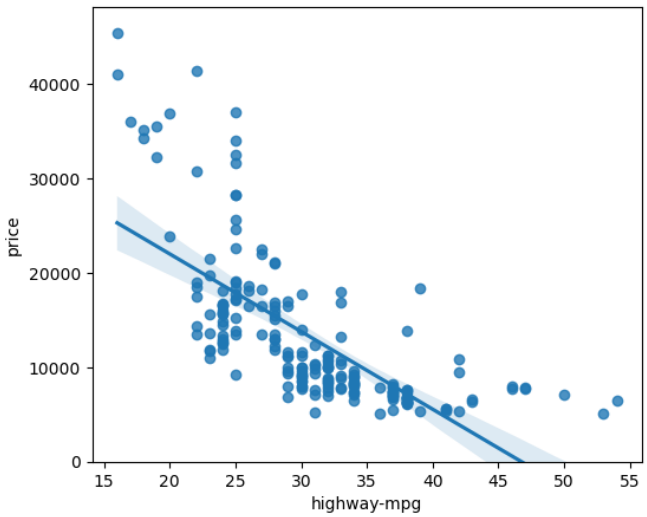
When it comes to simple linear regression, an excellent way to visualize the fit of our model is by using **regression plots**.

This plot will show a combination of a scattered data points (a **scatterplot**), as well as the fitted **linear regression** line going through the data. This will give us a reasonable estimate of the relationship between the two variables, the strength of the correlation, as well as the direction (positive or negative correlation).

Let's visualize **highway-mpg** as potential predictor variable of price:

In [29]:
```
width = 6
height = 5
plt.figure(figsize=(width, height))
sns.regplot(x="highway-mpg", y="price", data=df)
plt.ylim(0,)
```

Out[29]:
```
(0.0, 48157.8508372014)
```



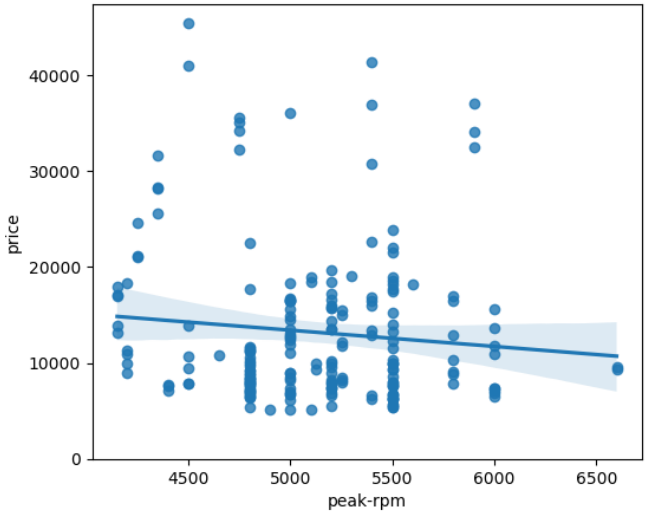
We can see from this plot that price is negatively correlated to highway-mpg since the regression slope is negative.

One thing to keep in mind when looking at a regression plot is to pay attention to how scattered the data points are around the regression line. This will give you a good indication of the variance of the data and whether a linear model would be the best fit or not. If the data is too far off from the line, this linear model might not be the best model for this data.

Let's compare this plot to the regression plot of "peak-rpm".

In [30]:
```
plt.figure(figsize=(width, height))
sns.regplot(x="peak-rpm", y="price", data=df)
plt.ylim(0,)
```

Out[30]: (0.0, 47414.1)



Comparing the regression plot of "peak-rpm" and "highway-mpg", we see that the points for "highway-mpg" are much closer to the generated line and, on average, decrease. The points for "peak-rpm" have more spread around the predicted line and it is much harder to determine if the points are decreasing or increasing as the "peak-rpm" increases.

Question #3:

Given the regression plots above, is "peak-rpm" or "highway-mpg" more strongly correlated with "price"? Use the method ".corr()" to verify your answer.

In [31]:
```
# Write your code below and press Shift+Enter to execute
df[["peak-rpm","highway-mpg","price"]].corr()
```

Out[31]:

	peak-rpm	highway-mpg	price
peak-rpm	1.000000	-0.058598	-0.101616
highway-mpg	-0.058598	1.000000	-0.704692
price	-0.101616	-0.704692	1.000000

▸ Click here for the solution

Residual Plot

A good way to visualize the variance of the data is to use a residual plot.

What is a **residual**?

The difference between the observed value (y) and the predicted value (Yhat) is called the residual (e). When we look at a regression plot, the residual is the distance from the data point to the fitted regression line.

So what is a **residual plot**?

A residual plot is a graph that shows the residuals on the vertical y-axis and the independent variable on the horizontal x-axis.

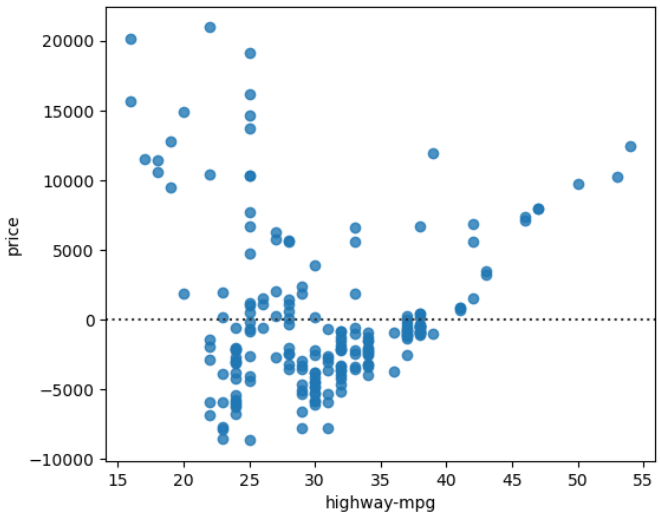
What do we pay attention to when looking at a residual plot?

We look at the spread of the residuals:

- If the points in a residual plot are **randomly spread out around the x-axis**, then a **linear model is appropriate** for the data.

Why is that? Randomly spread out residuals means that the variance is constant, and thus the linear model is a good fit for this data.

In [32]:
```
width = 6
height = 5
plt.figure(figsize=(width, height))
sns.residplot(x=df['highway-mpg'], y=df['price'])
plt.show()
```



What is this plot telling us?

We can see from this residual plot that the residuals are not randomly spread around the x-axis, leading us to believe that maybe a non-linear model is more appropriate for this data.

Multiple Linear Regression

How do we visualize a model for Multiple Linear Regression? This gets a bit more complicated because you can't visualize it with regression or residual plot.

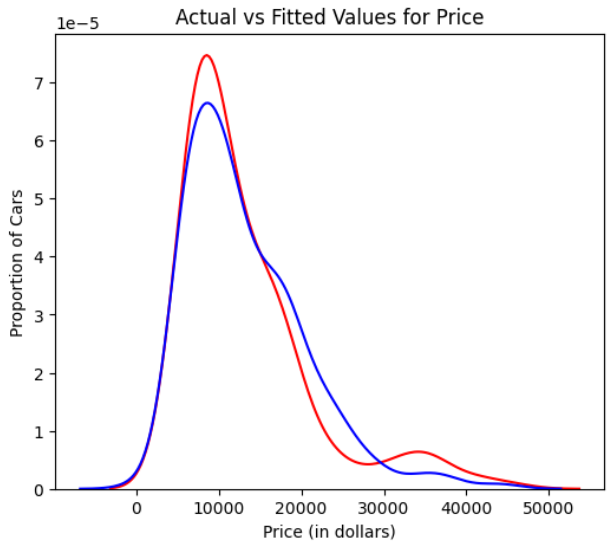
One way to look at the fit of the model is by looking at the **distribution plot**. We can look at the distribution of the fitted values that result from the model and compare it to the distribution of the actual values.

First, let's make a prediction:

In [33]:
```
Y_hat = lm.predict(Z)
```

In [34]:
```
plt.figure(figsize=(width, height))


ax1 = sns.distplot(df['price'], hist=False, color="r", label="Actual Value")
sns.distplot(Y_hat, hist=False, color="b", label="Fitted Values" , ax=ax1)


plt.title('Actual vs Fitted Values for Price')
plt.xlabel('Price (in dollars)')
plt.ylabel('Proportion of Cars')

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

```
<ipython-input-34-7377bca648c1>:4: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  ax1 = sns.distplot(df['price'], hist=False, color="r", label="Actual Value")
<ipython-input-34-7377bca648c1>:5: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.0.

Please adapt your code to use either `displot` (a figure-level function with
similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(Y_hat, hist=False, color="b", label="Fitted Values" , ax=ax1)
```



We can see that the fitted values are reasonably close to the actual values since the two distributions overlap a bit. However, there is definitely some room for improvement.

3. Polynomial Regression and Pipelines

Polynomial regression is a particular case of the general linear regression model or multiple linear regression models.

We get non-linear relationships by squaring or setting higher-order terms of the predictor variables.

There are different orders of polynomial regression:

Quadratic - 2nd Order

$$Yhat = a + b_1 X + b_2 X^2$$

Cubic - 3rd Order

$$Yhat = a + b_1 X + b_2 X^2 + b_3 X^3$$

Higher-Order:

$$Y = a + b_1 X + b_2 X^2 + b_3 X^3 \ldots$$

We saw earlier that a linear model did not provide the best fit while using "highway-mpg" as the predictor variable. Let's see if we can try fitting a polynomial model to the data instead.

We will use the following function to plot the data:

In [35]:
```
def PlotPolly(model, independent_variable, dependent_variabble, Name):
    x_new = np.linspace(15, 55, 100)
    y_new = model(x_new)

    plt.plot(independent_variable, dependent_variabble, '.', x_new, y_new, '-')
    plt.title('Polynomial Fit with Matplotlib for Price ~ Length')
    ax = plt.gca()
    ax.set_facecolor((0.898, 0.898, 0.898))
    fig = plt.gcf()
    plt.xlabel(Name)
    plt.ylabel('Price of Cars')

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

Let's get the variables:

In [36]:
```
x = df['highway-mpg']
y = df['price']
```

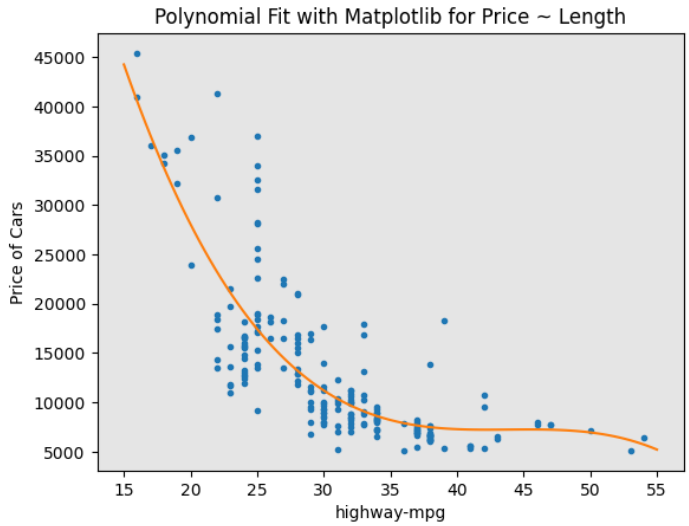
Let's fit the polynomial using the function **polyfit**, then use the function **poly1d** to display the polynomial function.

In [37]:
```
# Here we use a polynomial of the 3rd order (cubic)
f = np.polyfit(x, y, 3)
p = np.poly1d(f)
print(p)
```

```
        3         2
-1.557 x + 204.8 x - 8965 x + 1.379e+05
```

Let's plot the function:

In [38]:
```
PlotPolly(p, x, y, 'highway-mpg')
```



In [39]:
```
np.polyfit(x, y, 3)
```

Out[39]:
```
array([-1.55663829e+00,  2.04754306e+02, -8.96543312e+03,  1.37923594e+05])
```

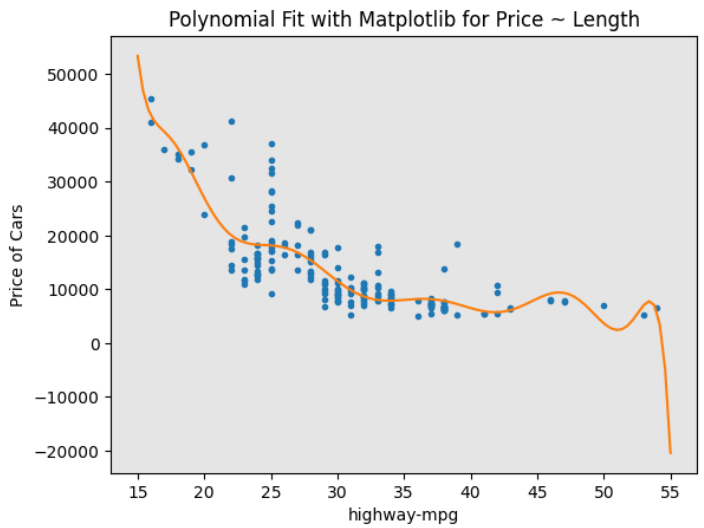
We can already see from plotting that this polynomial model performs better than the linear model. This is because the generated polynomial function "hits" more of the data points.

Question #4:

Create 11 order polynomial model with the variables x and y from above.

In [41]:
```
# Write your code below and press Shift+Enter to execute
f1 = np.polyfit(x, y, 11)
p1 = np.poly1d(f1)
print(p1)
PlotPolly(p1, x, y, 'highway-mpg')
```

```
           11             10            9           8         7
-1.243e-08 x  + 4.722e-06 x  - 0.0008028 x + 0.08056 x - 5.297 x
          6        5             4             3           2
 + 239.5 x - 7588 x + 1.684e+05 x - 2.565e+06 x + 2.551e+07 x - 1.491e+08 x + 3.879e+08
```



▸ Click here for the solution

The analytical expression for Multivariate Polynomial function gets complicated. For example, the expression for a second-order (degree=2) polynomial with two variables is given by:

$$Yhat = a + b_1 X_1 + b_2 X_2 + b_3 X_1 X_2 + b_4 X_1^2 + b_5 X_2^2$$

We can perform a polynomial transform on multiple features. First, we import the module:

In [42]:
```
from sklearn.preprocessing import PolynomialFeatures
```

We create a **PolynomialFeatures** object of degree 2:

In [43]:
```
pr=PolynomialFeatures(degree=2)
pr
```

Out[43]:
```
PolynomialFeatures()
```

In [44]:
```
Z_pr=pr.fit_transform(Z)
```

In the original data, there are 201 samples and 4 features.

In [45]:
```
Z.shape
```

Out[45]:
```
(201, 4)
```

After the transformation, there are 201 samples and 15 features.

In [46]:
```
Z_pr.shape
```

Out[46]:
```
(201, 15)
```

Pipeline

Data Pipelines simplify the steps of processing the data. We use the module **Pipeline** to create a pipeline. We also use **StandardScaler** as a step in our pipeline.

In [47]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
```

We create the pipeline by creating a list of tuples including the name of the model or estimator and its corresponding constructor.

In [48]:
```
Input=[('scale',StandardScaler()), ('polynomial', PolynomialFeatures(include_bias=False)), ('model',LinearRegression())]
```

We input the list as an argument to the pipeline constructor:

In [49]:
```
pipe=Pipeline(Input)
pipe
```

Out[49]:
```
Pipeline
  StandardScaler
  PolynomialFeatures
  LinearRegression
```

First, we convert the data type Z to type float to avoid conversion warnings that may appear as a result of StandardScaler taking float inputs.

Then, we can normalize the data, perform a transform and fit the model simultaneously.

In [50]:
```
Z = Z.astype(float)
pipe.fit(Z,y)
```

Out[50]:
```
Pipeline
  StandardScaler
  PolynomialFeatures
  LinearRegression
```

Similarly, we can normalize the data, perform a transform and produce a prediction simultaneously.

In [51]:
```
ypipe=pipe.predict(Z)
ypipe[0:4]
```

Out[51]:
```
array([13102.74784201, 13102.74784201, 18225.54572197, 10390.29636555])
```

Question #5:

Create a pipeline that standardizes the data, then produce a prediction using a linear regression model using the features Z and target y.

In [52]:
```
# Write your code below and press Shift+Enter to execute
```

```
Input = [('scale',StandardScaler()),('model',LinearRegression())]
pipe = Pipeline(Input)
pipe.fit(Z,y)
ypipe = pipe.predict(Z)
ypipe[0:10]
```

Out[52]:
```
array([13699.11161184, 13699.11161184, 19051.65470233, 10620.36193015,
       15521.31420211, 13869.66673213, 15456.16196732, 15974.00907672,
       17612.35917161, 10722.32509097])
```

▸ Click here for the solution

4. Measures for In-Sample Evaluation

When evaluating our models, not only do we want to visualize the results, but we also want a quantitative measure to determine how accurate the model is.

Two very important measures that are often used in Statistics to determine the accuracy of a model are:

- **R^2 / R-squared**
- **Mean Squared Error (MSE)**

R-squared

R squared, also known as the coefficient of determination, is a measure to indicate how close the data is to the fitted regression line.

The value of the R-squared is the percentage of variation of the response variable (y) that is explained by a linear model.

Mean Squared Error (MSE)

The Mean Squared Error measures the average of the squares of errors. That is, the difference between actual value (y) and the estimated value (ŷ).

Model 1: Simple Linear Regression

Let's calculate the R^2:

In [53]:
```
#highway_mpg_fit
lm.fit(X, Y)
# Find the R^2
print('The R-square is: ', lm.score(X, Y))
```

```
The R-square is:  0.4965911884339176
```

We can say that ~49.659% of the variation of the price is explained by this simple linear model "horsepower_fit".

Let's calculate the MSE:

We can predict the output i.e., "yhat" using the predict method, where X is the input variable:

In [54]:
```
Yhat=lm.predict(X)
print('The output of the first four predicted value is: ', Yhat[0:4])
```

```
The output of the first four predicted value is:  [16236.50464347 16236.50464347 17058.23802179 13771.3045085 ]
```

Let's import the function **mean_squared_error** from the module **metrics**:

In [55]:
```
from sklearn.metrics import mean_squared_error
```

We can compare the predicted results with the actual results:

In [56]:
```
mse = mean_squared_error(df['price'], Yhat)
print('The mean square error of price and predicted value is: ', mse)
```

```
The mean square error of price and predicted value is:  31635042.944639888
```

Model 2: Multiple Linear Regression

Let's calculate the R^2:

In [57]:
```
# fit the model
lm.fit(Z, df['price'])
# Find the R^2
print('The R-square is: ', lm.score(Z, df['price']))
```

```
The R-square is:  0.8093562806577457
```

We can say that ~80.896 % of the variation of price is explained by this multiple linear regression "multi_fit".

Let's calculate the MSE.

We produce a prediction:

In [58]:
```
Y_predict_multifit = lm.predict(Z)
```

We compare the predicted results with the actual results:

In [59]:
```
print('The mean square error of price and predicted value using multifit is: ', \
      mean_squared_error(df['price'], Y_predict_multifit))
```

```
The mean square error of price and predicted value using multifit is:  11980366.87072649
```

Model 3: Polynomial Fit

Let's calculate the R^2.

Let's import the function **r2_score** from the module **metrics** as we are using a different function.

In [60]:
```
from sklearn.metrics import r2_score
```

We apply the function to get the value of R^2:

In [61]:
```
r_squared = r2_score(y, p(x))
print('The R-square value is: ', r_squared)
```

```
The R-square value is:  0.702376909243598
```

We can say that ~67.419 % of the variation of price is explained by this polynomial fit.

MSE

We can also calculate the MSE:

In [62]:
```
mean_squared_error(df['price'], p(x))
```

Out[62]:
```
18703127.63915394
```

5. Prediction and Decision Making

Prediction

In the previous section, we trained the model using the method **fit**. Now we will use the method **predict** to produce a prediction. Lets import **pyplot** for plotting; we will also be using some functions from numpy.

In [63]:
```
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline
```

Create a new input:

In [64]:
```
new_input=np.arange(1, 100, 1).reshape(-1, 1)
```

Fit the model:

In [65]:
```
lm.fit(X, Y)
lm
```

Out[65]:
```
LinearRegression()
```

Produce a prediction:

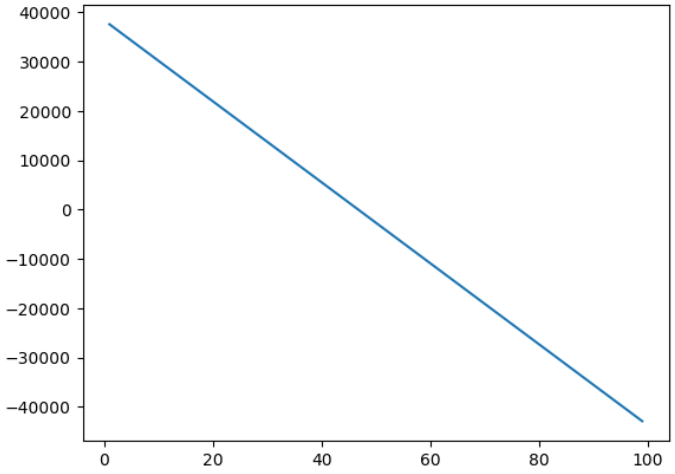
In [66]:
```
yhat=lm.predict(new_input)
yhat[0:5]
```

```
/lib/python3.12/site-packages/sklearn/base.py:493: UserWarning: X does not have valid feature names, but LinearRegression was fitted with feature names
  warnings.warn(
```

Out[66]:
```
array([37601.57247984, 36779.83910151, 35958.10572319, 35136.37234487,
       34314.63896655])
```

We can plot the data:

In [67]:
```
plt.plot(new_input, yhat)
plt.show()
```



Decision Making: Determining a Good Model Fit

Now that we have visualized the different models, and generated the R-squared and MSE values for the fits, how do we determine a good model fit?

- *What is a good R-squared value?*

When comparing models, **the model with the higher R-squared value is a better fit** for the data.

- *What is a good MSE?*

When comparing models, **the model with the smallest MSE value is a better fit** for the data.

Let's take a look at the values for the different models.

Simple Linear Regression: Using Highway-mpg as a Predictor Variable of Price.

- R-squared: 0.49659118843391759
- MSE: 3.16 x10^7

Multiple Linear Regression: Using Horsepower, Curb-weight, Engine-size, and Highway-mpg as Predictor Variables of Price.

- R-squared: 0.80896354913783497
- MSE: 1.2 x10^7

Polynomial Fit: Using Highway-mpg as a Predictor Variable of Price.

- R-squared: 0.6741946663906514
- MSE: 2.05 x 10^7

Simple Linear Regression Model (SLR) vs Multiple Linear Regression Model (MLR)

Usually, the more variables you have, the better your model is at predicting, but this is not always true. Sometimes you may not have enough data, you may run into numerical problems, or many of the variables may not be useful and even act as noise. As a result, you should always check the MSE and R^2.

In order to compare the results of the MLR vs SLR models, we look at a combination of both the R-squared and MSE to make the best conclusion about the fit of the model.

- **MSE**: The MSE of SLR is 3.16x10^7 while MLR has an MSE of 1.2 x10^7. The MSE of MLR is much smaller.
- **R-squared**: In this case, we can also see that there is a big difference between the R-squared of the SLR and the R-squared of the MLR. The R-squared for the SLR (~0.497) is very small compared to the R-squared for the MLR (~0.809).

This R-squared in combination with the MSE show that MLR seems like the better model fit in this case compared to SLR.

Simple Linear Model (SLR) vs. Polynomial Fit

- **MSE**: We can see that Polynomial Fit brought down the MSE, since this MSE is smaller than the one from the SLR.
- **R-squared**: The R-squared for the Polynomial Fit is larger than the R-squared for the SLR, so the Polynomial Fit also brought up the R-squared quite a bit.

Since the Polynomial Fit resulted in a lower MSE and a higher R-squared, we can conclude that this was a better fit model than the simple linear regression for predicting "price" with "highway-mpg" as a predictor variable.

Multiple Linear Regression (MLR) vs. Polynomial Fit

- **MSE**: The MSE for the MLR is smaller than the MSE for the Polynomial Fit.
- **R-squared**: The R-squared for the MLR is also much larger than for the Polynomial Fit.

Conclusion

Comparing these three models, we conclude that **the MLR model is the best model** to be able to predict price from our dataset. This result makes sense since we have 27 variables in total and we know that more than one of those variables are potential predictors of the final car price.

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.

Abhishek Gagneja

<!-- ## Change Log | Date (YYYY-MM-DD) | Version | Changed By | Change Description | |---|---|---|---| | 2023-09-28 | 2.3 | Abhishek Gagneja | Updated instructions | | 2020-10-30 | 2.2 | Lakshmi |Changed url of csv | | 2020-09-09 | 2.1 | Lakshmi | Fixes made in Polynomial Regression Equations | | 2020-08-27 | 2.0 | Lavanya | Moved lab to course repo in GitLab | --!>