Chapter 4 - Polynomial Regression

Solution Workbook for Student Practical Classes

(c) Dr Rob Collins 2024

Invalid Date

Table of contents

4 Polynomial Regression **4 - 3**
4.1 Introduction . . . 4 - 3
4.2 Instructions for Students . . . 4 - 3
4.3 Import Libraries . . . 4 - 4
4.4 Load the Data . . . 4 - 5
4.5 Review the data quantitatively . . . 4 - 6
4.6 Review the data visually . . . 4 - 9
4.7 Selecting the data we wish to use for modelling . . . 4 - 13
4.8 Splitting data for training and testing . . . 4 - 14
4.9 First model - linear regression . . . 4 - 16
4.9.1 Build the linear regression model . . . 4 - 16
4.9.2 Test the linear regression model . . . 4 - 16
4.10 Second model - Polynomial regression . . . 4 - 17
4.11 Final thoughts . . . 4 - 20

List of Figures

1 Master Artisan 'Learnists' Fitting a Complex Mathematical Curve (c. 1590) . 4 - 2
2 missingno bar chart for the Auto-mpg Data . . . 4 - 8
3 missingno 'bar' chart showing impact of dropping rows containing missing values . . . 4 - 8
4 Scatter matrix for the complete auto_mpg data-set . . . 4 - 9
5 Correlation matrix for the data-set . . . 4 - 11
6 3D Plot of mpg, displacement and weight . . . 4 - 13

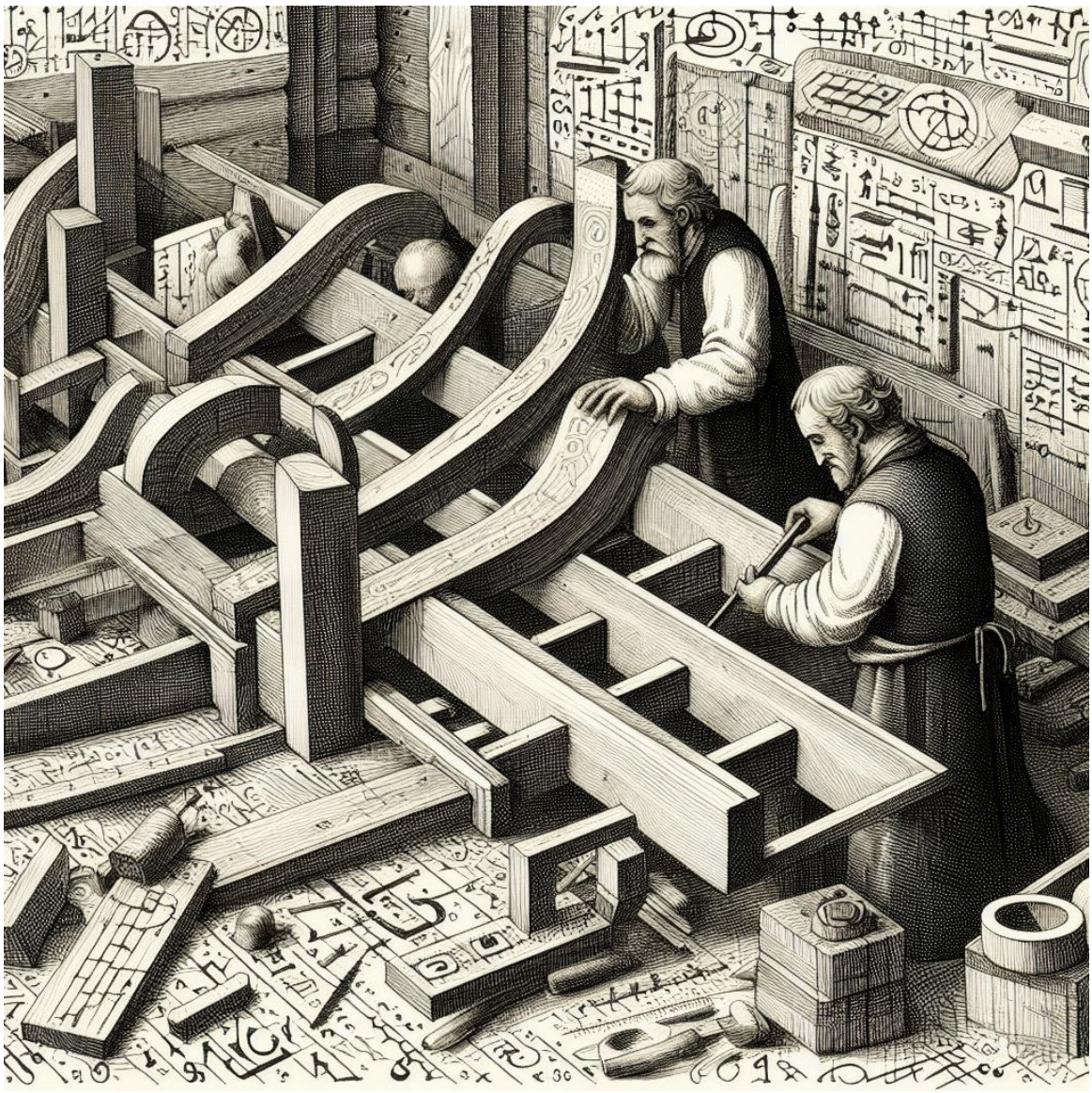

Figure 1: Master Artisan ‘Learnists’ Fitting a Complex Mathematical Curve (c. 1590)

4 Polynomial Regression

4.1 Introduction

In this workshop we will extend our previous work on Regression. In this session we will review 'polynomial regression' - that is, functions of the general form:

$y = \theta_0.x_0 + \theta_1.x_1 + \theta_2.x_1^2 + \theta_3.x_1^3 + ... + \theta_n.x_1^n$

where, as previously we set: $x_0 = 1$

The above general equation is a 'polynomial' (it has squared, cubed and fourth-power terms etc.) but it is only in one dimension. Or, to put it another way, in a data-set we were using to build the model, there would only be one feature (that is, one column). To make polynomial models even more general, they need to operate across multiple dimensions. Not only that, but they need to be capable of representing 'interactions' between dimensions.

Thus, for a second-order, polynomial model in two dimensions $x_1 and x_2$ we would see:

$y = \theta_0.x_0 + \theta_1.x_1 + \theta_2.x_2 + \theta_3.x_1^2 + \theta_4.x_2^2 + \theta_5.x_1.x_2$

Where the final '$\theta_5.x_1.x_2$' term enables us to model any interaction between $x_1 and x_2$.

The same logic holds in higher dimensions, and thus you should expect to see a model based on not only powers of each feature, but also terms relating to all permutations of features! This may sound horribly complicated! How on earth would somebody build a model with all of those complicated terms?

Inevitably, there is an easy answer. The answer, in some ways feels like a 'trick' to me - but is actually quite obvious when you see it. Not only that, but library functions within the Machine Learning libraries enable this type of model to be built quite easily.

The data-set used in this workshop is a slightly modified version of the 'auto-mpg' data, originally collected by R. Quinlan [https://archive.ics.uci.edu/dataset/9/auto+mpg] and discussed in the paper:

Quinlan, J.R., 1993, June. "Combining instance-based and model-based learning". In *Proceedings of the tenth international conference on machine learning* (pp. 236-243).

4.2 Instructions for Students

In this workbook there are regular 'callout' blocks indicating where you should add your own code. They look like this:

In those cases you are required to create the code for the block based on your learning on this course. In some cases the tutor has provided 'clues' towards the code you need to write and in a few places the complete code block.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine

2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class
8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

The numbers shown in the 'In [n]' text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

4.3 Import Libraries

As always we start by import the required libraries. In this case, those are:

- numpy : Conventionally named as 'np'
- pandas : Conventionally named as 'pd'

- matplotlib.pyplot : Conventionally named as plt
- scatter_matrix : Imported from pandas.plotting
- seaborn: Conventionally named as sns
- Axes3D : Imported from mpl_toolkits.mplot3d
- train_test_split : Imported from sklearn.model_selection
- LinearRegression : Imported from sklearn.linear_model
- mean_squared_error : Imported from sklearn.metrics
- r2_score : Imported from sklearn.metrics
- PolynomialFeatures : Imported from sklearn.preprocessing
- missingno : Conventionally named as msno

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pandas.plotting import scatter_matrix
import seaborn as sns
from mpl_toolkits.mplot3d import Axes3D
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import PolynomialFeatures
import missingno as msno
```

4.4 Load the Data

Add code to load then display the 'auto_mpg house price' data set from the provided data file 'CML_Auto_MPG_v1.txt'. Note that unlike the previous data files, the features (columns) in this data file are separated by 'tab' characters. You will need to modify the use of the pandas read_csv function to correctly read tab-separated data (or use some other method to fix this issue).

```
auto_mpg = pd.read_csv("CML_Auto_MPG_v1.txt", sep='\t')
print(f"auto_mpg.iloc[0:10, 0:4] = \n{ auto_mpg.iloc[0:20, 0:4]}")
```

```
auto_mpg.iloc[0:10, 0:4] =
     mpg  cylinders  displacement  horsepower
0   18.0        8.0         307.0       130.0
1   15.0        8.0         350.0       165.0
2   18.0        8.0         318.0       150.0
3   16.0        8.0         304.0       150.0
4   17.0        8.0         302.0       140.0
5   15.0        8.0         429.0       198.0
6   14.0        8.0         454.0       220.0
7   14.0        8.0         440.0       215.0
8   14.0        8.0         455.0       225.0
9   15.0        8.0         390.0       190.0
10  15.0        8.0         383.0       170.0
11  14.0        8.0         340.0       160.0
12  15.0        8.0         400.0       150.0
13  14.0        8.0         455.0       225.0
14  24.0        4.0         113.0        95.0
15  22.0        6.0         198.0        95.0
16  18.0        6.0         199.0        97.0
17  21.0        6.0         200.0        85.0
18  27.0        4.0          97.0        88.0
19  26.0        4.0          97.0        46.0
```

```
print(f"auto_mpg_no_nan.iloc[0:10, 5:] = \n{ auto_mpg.iloc[0:20, 5:]}")
```

```
auto_mpg_no_nan.iloc[0:10, 5:] =
    acceleration  year  origin                       car
0           12.0  70.0       1  chevrolet chevelle malibu
1           11.5  70.0       1          buick skylark 320
2           11.0  70.0       1         plymouth satellite
3           12.0  70.0       1              amc rebel sst
4           10.5  70.0       1                ford torino
5           10.0  70.0       1           ford galaxie 500
```

```
6            9.0  70.0       1              chevrolet impala
7            8.5  70.0       1             plymouth fury iii
8           10.0  70.0       1              pontiac catalina
9            8.5  70.0       1            amc ambassador dpl
10          10.0  70.0       1           dodge challenger se
11           8.0  70.0       1            plymouth 'cuda 340
12           9.5  70.0       1         chevrolet monte carlo
13          10.0  70.0       1       buick estate wagon (sw)
14          15.0  70.0       3         toyota corona mark ii
15          15.5  70.0       1                plymouth duster
16          15.5  70.0       1                     amc hornet
17          16.0  70.0       1                  ford maverick
18          14.5  70.0       3                   datsun pl510
19          20.5  70.0       2  volkswagen 1131 deluse sedan
```

4.5 Review the data quantitatively

It is often useful to review a brief summary of the data. The .describe() function of Pandas provides summary data for the dataframe, including:

- count : The number of items in the dataframe
- mean : The mean or 'average' value .. sum of values / number of values
- std : Standard deviation - a measure of spread in the data
- min : The smallest value
- 25%, 50%, 75% : Quartile values
- max : The largest value

Use the .describe() function to provide a summary of the data. Obviously, this type of 'statistical' summary only makes sense for certain types of data.

Note : In the following I spread the display over two cells so that they format correctly for this workbook. You may prefer simply to include the name of the dataframe as the last line in the cell - in this case you should obtain a scrollable full-width display of the data. If you wish to do as I have done, then you will need to use the pandas 'iloc[]' method to select a sub-set of the data-frame for display.

```
1 auto_mpg.iloc[:,0:4].describe()
```

	mpg	cylinders	displacement	horsepower
count	398.000000	396.000000	398.000000	392.000000
mean	23.514573	5.462121	193.425879	104.469388
std	7.815984	1.702144	104.269838	38.491160
min	9.000000	3.000000	68.000000	46.000000
25%	17.500000	4.000000	104.250000	75.000000
50%	23.000000	4.000000	148.500000	93.500000

	mpg	cylinders	displacement	horsepower
75%	29.000000	8.000000	262.000000	126.000000
max	46.600000	8.000000	455.000000	230.000000

```
auto_mpg.iloc[:,5:].describe()
```

	acceleration	year	origin
count	398.000000	396.000000	398.000000
mean	15.568090	76.005051	1.572864
std	2.757689	3.706303	0.802055
min	8.000000	70.000000	1.000000
25%	13.825000	73.000000	1.000000
50%	15.500000	76.000000	1.000000
75%	17.175000	79.000000	2.000000
max	24.800000	82.000000	3.000000

As in previous workshops, we should review the data to see if there are any missing values. In this case, we introduce a new and useful function 'isnull' that scans each column and indicates if it has any null elements:~

```
pd.isnull(auto_mpg).any()
```

```
mpg             False
cylinders        True
displacement    False
horsepower       True
weight           True
acceleration    False
year             True
origin          False
car             False
dtype: bool
```

Clearly there is some missing data in the data-set. Perhaps we should review this more carefully using missingno?

```
msno.bar(auto_mpg)
plt.show()
```

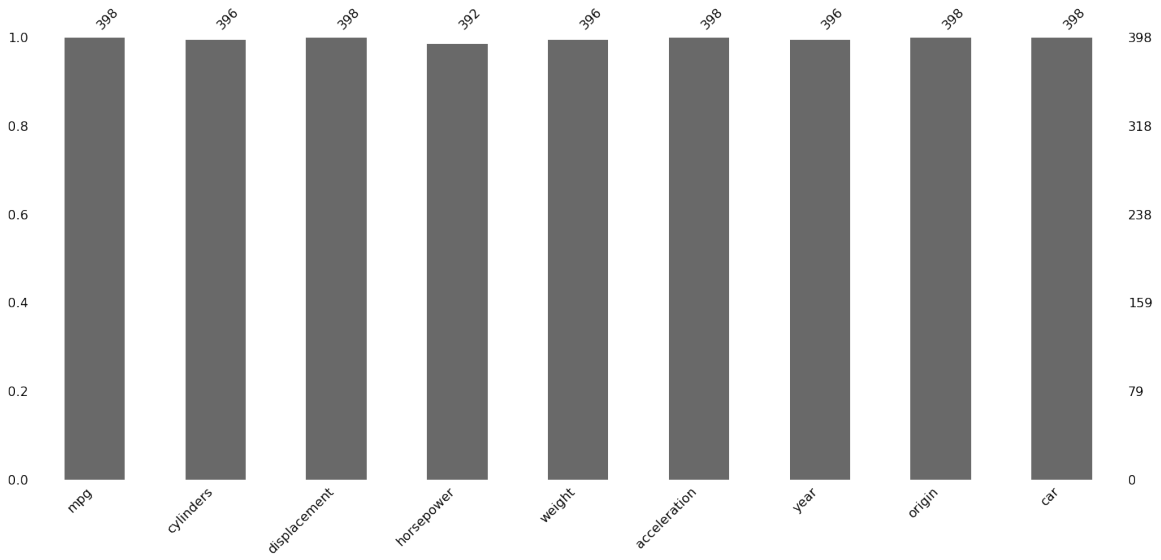

Figure 2: missingno bar chart for the Auto-mpg Data

In this case, the amount of missing data is a relatively small proportion of the complete data-set. For this reason, I am willing to delete all of the rows with missing data. Normally it is a better choice to impute missing values - and you could experiment doing that here if you wish.

```
auto_mpg_no_nan = auto_mpg.dropna(axis='index')
msno.bar(auto_mpg_no_nan)
plt.show()

```

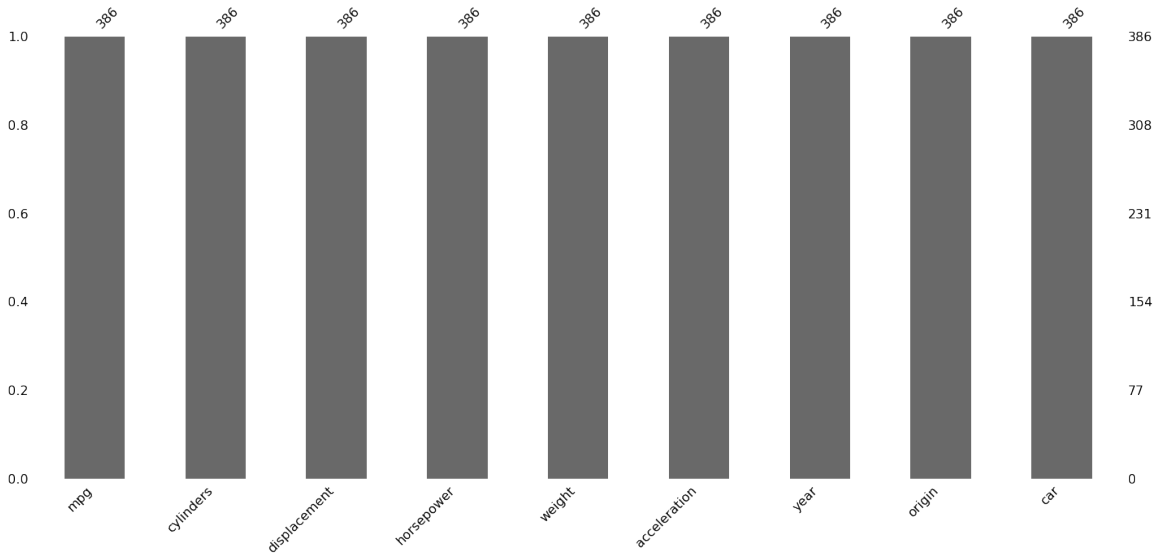

Figure 3: missingno 'bar' chart showing impact of dropping rows containing missing values

4.6 Review the data visually

We can start with the Pandas 'scatter_matrix' function we have used before to review all of the features and the correlations between them:

```
scatter_matrix(auto_mpg_no_nan, figsize=(12, 12));
plt.show()
```

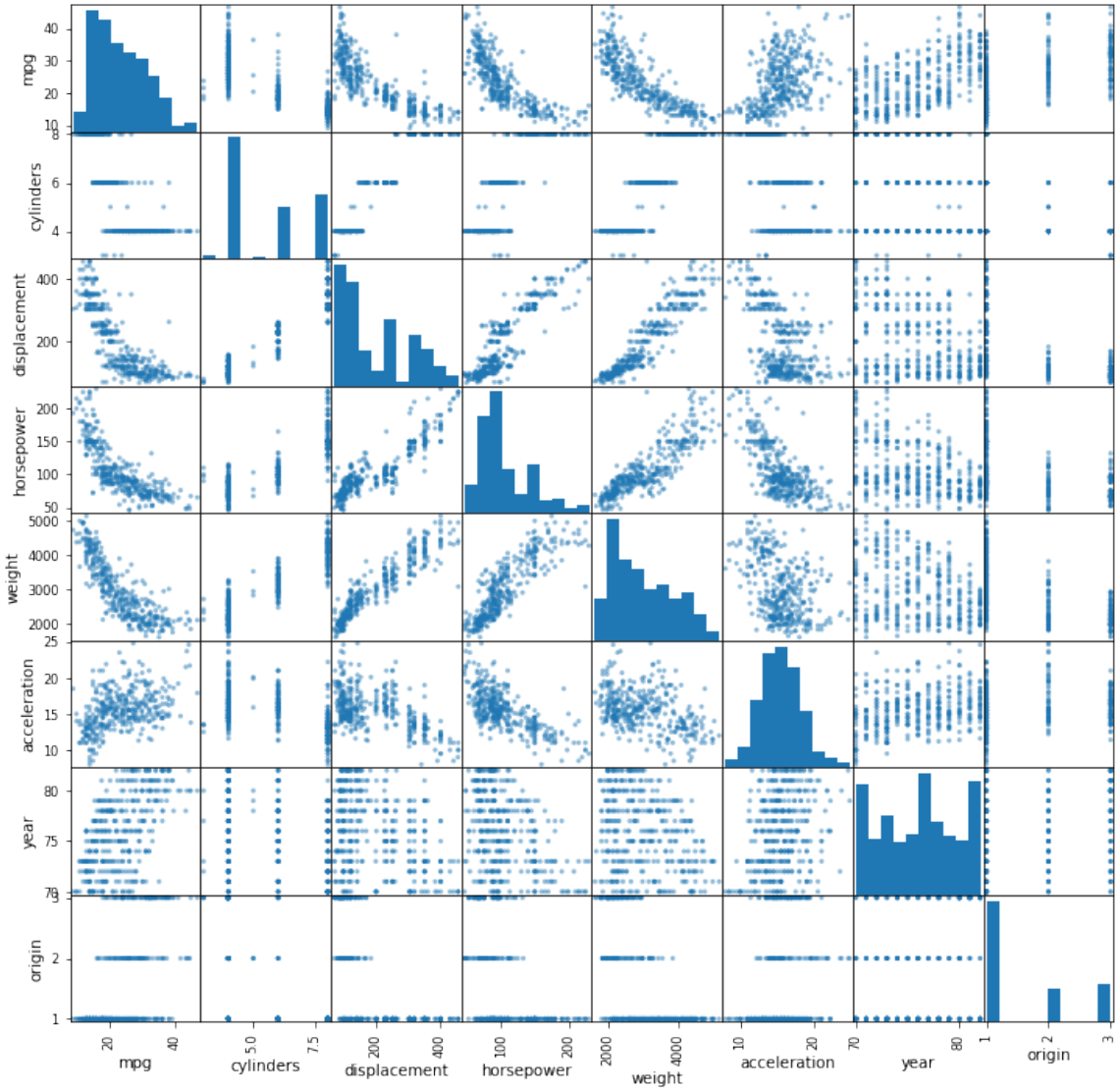

Figure 4: Scatter matrix for the complete auto_mpg data-set

As previously, you are encouraged to study this plot and determine what it means to you in relation to this modelling activity. I notice the following things:

- There do not appear to be any obvious extreme outliers in the data
- Acceleration appears to be normally distributed
- Horsepower and weight are somewhat normal - but rather skewed
- Year is clearly a discreet value (representing the year of manufacture). There is a slight upward trend in mpg over the time-period of the data
- It is not clear what 'origin' represents. It has three discreet values and may represent the place of manufacture (US, Europe, Asia?) (Note I have not been able to determine the definition of this feature - it is not recorded in the original paper relating to this data-set published by Quinlan(1993))
- The four features 'acceleration', 'weight', 'horsepower' and 'displacement' all appear to have a non-linear relationship with the dependent variable we are interested in 'mpg'
- Some of the correlations are positive (e.g. 'weight' vs 'displacement') whereas other are negative (E.g.'horsepower' v 'mpg' and 'weight' v 'mpg')

Note that there is also evidence of cross-correlation between these last four variables - and this seems reasonable when thinking about the physics / mechanics of the system. Statistically, we might conclude that each of these features is in conveying some of the same information - but in different ways. In a later module we will consider this issue in more depth: 'Principle Component Analysis' (PCA) provides a method for extracting key information from multiple variables and 'compressing' them into a smaller number of independent variables. For now, we will continue without using that technique.

Since we are going to create a regression model, it is useful to understand the correlation between the features and the independent variable. As well as the Correlatioon Matrix there is another method for reviewing the correlation between features, as follows:

Add code to display a **correlation matrix** for the data-set. This chart is clearer if the absolute value of each correlation is plotted. This is because, regardless of whether there is a negative or positive correlation .. a stronger correlation will provide a better basis for modelling.

- Compute the correlation of the dataset using the '.corr()' function.
- Compute the absolute value for each element in this matrix using the numpy 'absolute' function
- Plot this correlation using the seaborn 'heatmap()' method.

```
%matplotlib inline
correlation_matrix = np.absolute(auto_mpg_no_nan.corr().round(2))
sns.set(rc={'figure.figsize':(10,10)})
ax = sns.heatmap(correlation_matrix, annot=True, cmap='Blues')
bottom, top = ax.get_ylim()
ax.set_ylim(bottom + 0.5, top - 0.5)
plt.show()
```

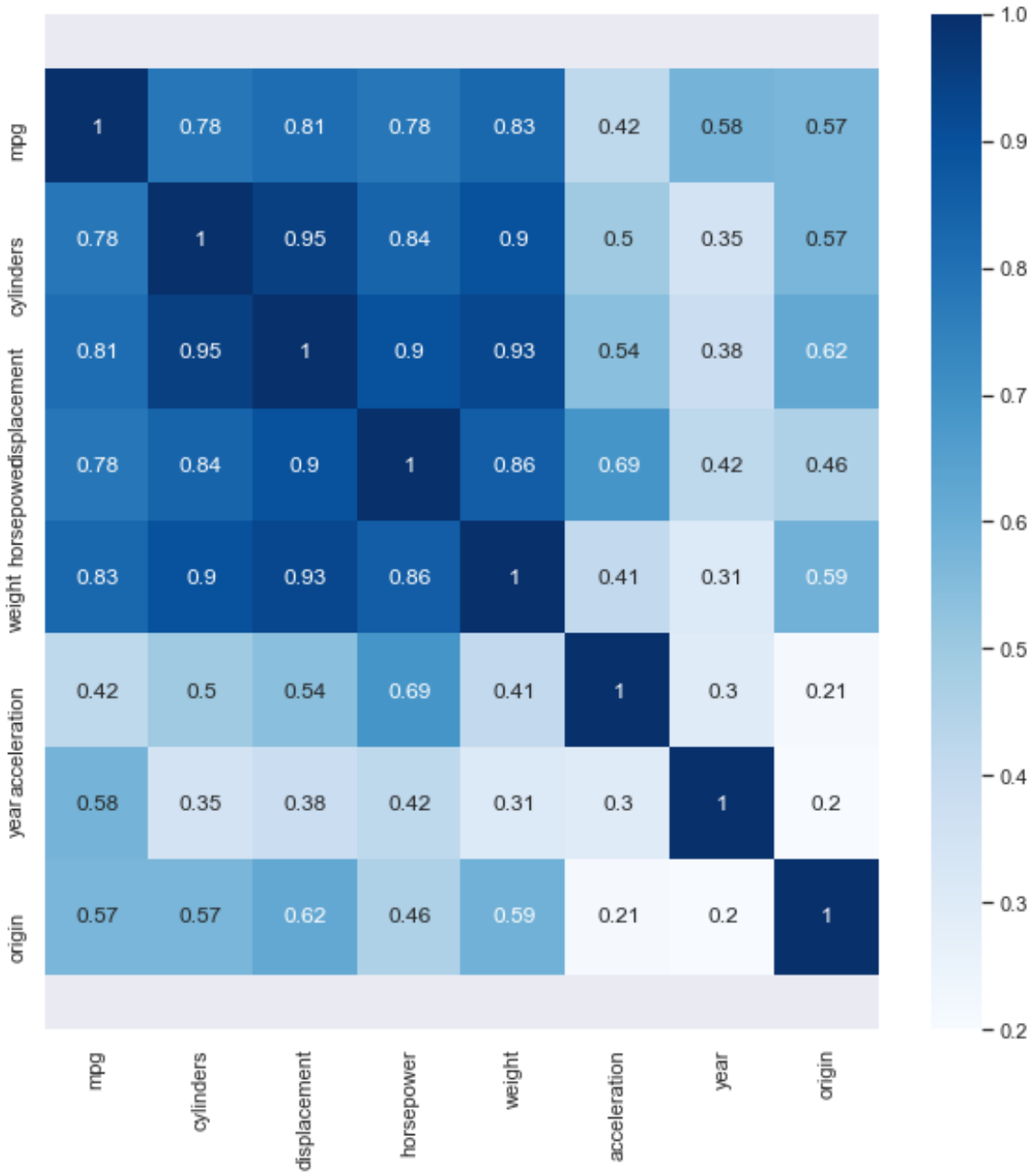

Figure 5: Correlation matrix for the data-set

The correlation coefficients are show in the matrix. Stronger correlations are indicated with darker blue cells.

You can experiment with the way that the heat map is displayed. The 'cmap' option in the 'heatmap' function defines the colours that are used for the plot. Many different sets of colours are available and these are documented here: Other colour map options

at https://matplotlib.org/3.1.1/gallery/color/colormap_reference.html. Experiment with some of the other possible cmap pallets and decide which one you prefer.

From this correlation map we see that, as in the previous section, there is evidence of strong correlation between MPG and four other variables: 'acceleration', 'weight', 'horsepower' and 'displacement'. These are therefore good candidates for building our model.

It is interesting to review these features in a 3D plot. Set the vertical dimension to represent MPG.

Note I have used the command

```
%matplotlib inline
```

So that the chart displays neatly in this workbook. However, you should use the command:

```
%matplotlib auto
```

To create a 'pop-out' 3D window that can be rotated and zoomed.

We can also set the colour of a data point based on the value of MPG. This provides another visual clue to help understand the data.

Rotate the 3D view and look at the 'shape' of the data. This is the thing we are going to try to model. It is clearly multi-dimensional and also appears to be non-linear. It will thus require a polynomial model.

Experiment with the cmap values to create a visualisation that you prefer. Also, experiment by plotting other variables on the 3D chart.

```
# Uncomment next  line if you want the graph to appear in the page
%matplotlib inline

# Uncomment next line if you want a 3D pop-out
# This command does not seem to work on a Virtual Machine via
# a Guacamole server but should work correctly on a local
# machine - depending on your configuration

# %matplotlib auto
```

```
fig = plt.figure(figsize=(7,7))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(auto_mpg_no_nan['displacement'],
           auto_mpg_no_nan['weight'],
           auto_mpg_no_nan['mpg'],
           c=auto_mpg_no_nan['mpg'],
           cmap ='gist_heat' )
```

```
8
9 plt.show()
```

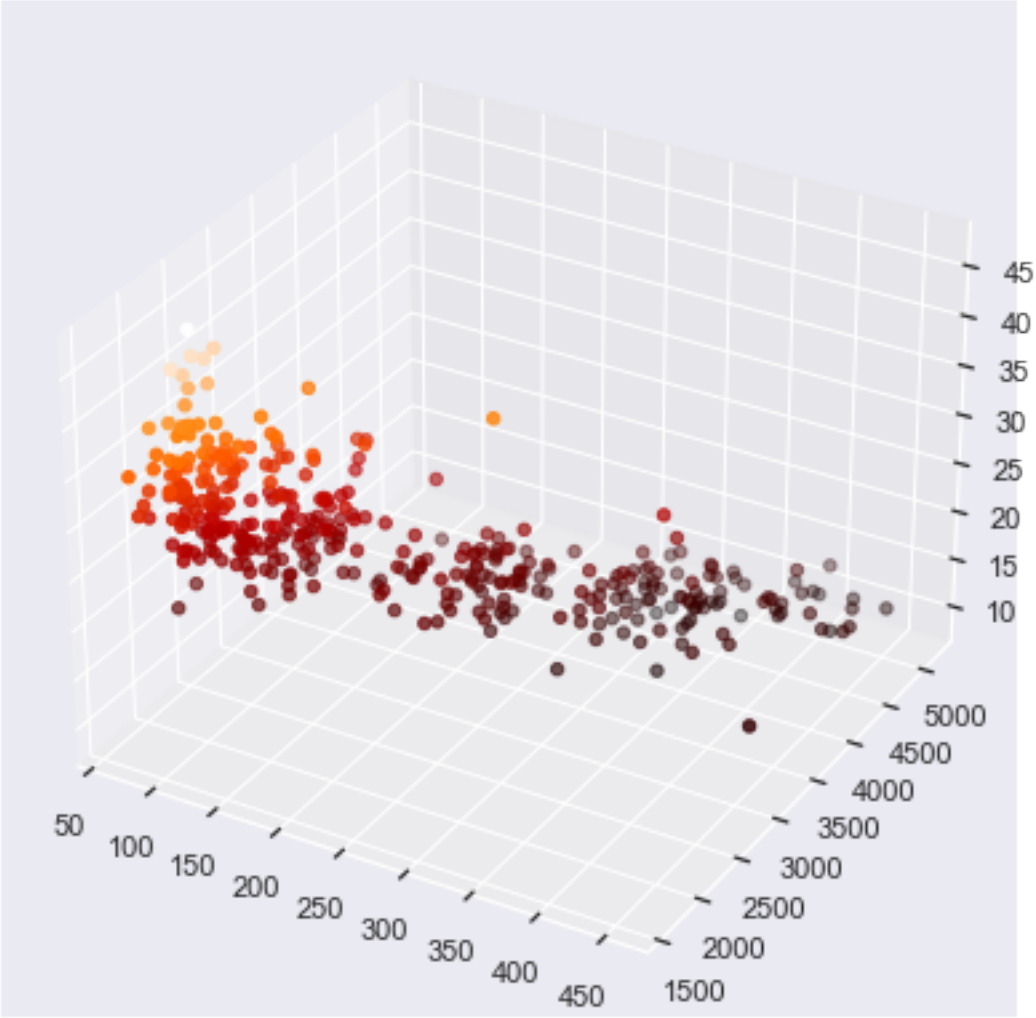

Figure 6: 3D Plot of mpg, displacement and weight

4.7 Selecting the data we wish to use for modelling

Add code to create a sub-set of the overall dataframe by copying the four correlated features to a new dataframe. Call that dataframe 'X'

```
1 feature_names = ['acceleration', 'weight', 'horsepower', 'displacement']
2 X = auto_mpg_no_nan[feature_names].copy()
3 print(f" X.iloc[0:10,:] = \n {X.iloc[0:10:]}")
```

```
 X.iloc[0:10,:] =
    acceleration  weight  horsepower  displacement
0          12.0  3504.0       130.0         307.0
1          11.5  3693.0       165.0         350.0
2          11.0  3436.0       150.0         318.0
3          12.0  3433.0       150.0         304.0
4          10.5  3449.0       140.0         302.0
5          10.0  4341.0       198.0         429.0
6           9.0  4354.0       220.0         454.0
7           8.5  4312.0       215.0         440.0
8          10.0  4425.0       225.0         455.0
9           8.5  3850.0       190.0         390.0
```

```
Y = auto_mpg_no_nan['mpg']
print(f" Y.iloc[0:10,:] = \n {Y.iloc[0:10:]}")
```

```
 Y.iloc[0:10,:] =
 0    18.0
1    15.0
2    18.0
3    16.0
4    17.0
5    15.0
6    14.0
7    14.0
8    14.0
9    15.0
Name: mpg, dtype: float64
```

4.8 Splitting data for training and testing

We want to be able to test how good our model is for prediction. In particular, we need to be assured that is is not over-fitting the data.

Add code to split the model into training and a testing sub-sets using sklearn's 'train_test_split' method.

Note Because this function creates a random selection, you are likely to see different rows selected for your train and test sets. To deal with this issue, I added a parameter 'random_state = 4' to the train_test_split function. This ensure that every run, whilst still being 'randomized', produces the same result. This is often used for teaching and demonstration purposes but is not recommended for production runs of your model building.

```
X_train, X_test, Y_train, Y_test = train_test_split(
    X,
    Y,
    test_size = 0.2,
    random_state=4)
print(f"X_train[0:5] = \n{X_train[0:5]}\n")
print(f"X_test[0:5] = \n{X_test[0:5]}\n")
print(f"Y_train[0:5] = \n{Y_train[0:5]}\n")
print(f"Y_test[0:5] = \n{Y_test[0:5]}\n")
```

```
X_train[0:5] =
     acceleration  weight  horsepower  displacement
65           13.0  4129.0       153.0         351.0
132          17.0  2542.0        75.0         140.0
392          17.3  2950.0        90.0         151.0
62           12.0  4274.0       165.0         350.0
363          15.8  3415.0       110.0         231.0

X_test[0:5] =
     acceleration  weight  horsepower  displacement
254          15.8  2965.0        85.0         200.0
150          15.5  2391.0        93.0         108.0
74           16.0  4294.0       140.0         302.0
157          14.0  4440.0       145.0         350.0
264          11.2  3205.0       139.0         302.0

Y_train[0:5] =
65     14.0
132    25.0
392    27.0
62     13.0
363    22.4
Name: mpg, dtype: float64

Y_test[0:5] =
254    20.2
150    26.0
74     13.0
157    15.0
264    18.1
Name: mpg, dtype: float64
```

4.9 First model - linear regression

To give us a baseline to judge our polynomial regression model we first generate a linear model and and produce metrics that can help us jude the quality of the model.

4.9.1 Build the linear regression model

Building a linear model is trivial using sklearn:

- Instantiate a model object based on the 'LinearRegression()' class
- Call the '.fit' method of that object, passing as parameters the X_train and Y_train data

```
lin_model = LinearRegression()
lin_model.fit(X_train, Y_train)
```

```
LinearRegression()
```

4.9.2 Test the linear regression model

The sklearn library provides a range of functions for scoring different types of models:

https://scikit-learn.org/stable/modules/model_evaluation.html

Two useful and common quality measure for regression are:

- The R^2 score or 'Coefficient of Determination'
  - (See https://en.wikipedia.org/wiki/Coefficient_of_determination)
- The 'root mean squared error' or RMSE

Both of these can easily be computed using sklearn functions as follows.

It is interesting to compare how well the model fitted to the training data as compared with the performance of the model on test data (which the model has not 'seen'). So we will compute these scores first for the training data and then for the test data:

```
y_train_predict = lin_model.predict(X_train)
r2_train = r2_score(Y_train, y_train_predict)
rmse_train = (np.sqrt(mean_squared_error(Y_train, y_train_predict)))
```

Now do the same thing for the test data:

```
y_test_predict = lin_model.predict(X_test)
r2_test = r2_score(Y_test, y_test_predict)
rmse_test = (np.sqrt(mean_squared_error(Y_test, y_test_predict)))
```

And print the results for comparison:

```
print("R2:")
print(f"  Train = {r2_train:.2f}")
print(f"  Test  = {r2_test:.2f}")
print("RMSE:")
print(f"  Train = {rmse_train:.2f}")
print(f"  Test  = {rmse_test:.2f}")
```

```
R2:
  Train = 0.70
  Test  = 0.71
RMSE:
  Train = 4.26
  Test  = 4.09
```

You should see that the values for R2 and RMSE are quite similar. This indicates that we are not seeing 'over-fitting'. This is in-line with our expectations given that this is a linear model and we already have some clues that there is a more complex relationship between features.

4.10 Second model - Polynomial regression

Now comes the 'trick' that enables us to easily generate a polynomial model for given data.

sklearn provides a function that generates an expanded data-set. It creates new features that are combinations of the other features. So that if we have a feature 'x' and we use the function to generate Polynomial Features of degree 3, then the function will add new data columns (features) for x^2 and x^3.

Even better, if we have two features .. say 'x' and 'z', and we use the function to generate Polynomial Features, for example, order 2, then it will generate new features for x^2, z^2 and $x.z$ (the last of these being the interaction term described in the first section above.

So let's experiment with the sklearn.PolynomialFeatures function and generate an expanded data-set for variables in X_train.

We first generate the features and take a look at the extended set of feature names:

Note: In my code above, I defined the list of 'interesting' features in a variable called 'feature_names'. I re-use that variable in the following cells.

```
poly_features = PolynomialFeatures(degree=2)
X_train_poly = poly_features.fit_transform(X_train)
print(poly_features.get_feature_names_out(feature_names))
```

```
['1' 'acceleration' 'weight' 'horsepower' 'displacement' 'acceleration^2'
 'acceleration weight' 'acceleration horsepower'
 'acceleration displacement' 'weight^2' 'weight horsepower'
 'weight displacement' 'horsepower^2' 'horsepower displacement'
 'displacement^2']
```

We can also look at the actual data that has been produced:

Note In the following I restricted the display of values to using 2 decimal points using the following command:

```
np.set_printoptions(precision=2, suppress=True)
```

```
1 np.set_printoptions(precision=2, suppress=True, floatmode='fixed')
2 print(f"X_train_poly[0:5] = \n{X_train_poly[0:5]}")
```

```
X_train_poly[0:5] =
[[       1.00       13.00     4129.00      153.00      351.00      169.00
     53677.00     1989.00     4563.00 17048641.00   631737.00  1449279.00
     23409.00    53703.00   123201.00]
 [       1.00       17.00     2542.00       75.00      140.00      289.00
     43214.00     1275.00     2380.00  6461764.00   190650.00   355880.00
      5625.00    10500.00    19600.00]
 [       1.00       17.30     2950.00       90.00      151.00      299.29
     51035.00     1557.00     2612.30  8702500.00   265500.00   445450.00
      8100.00    13590.00    22801.00]
 [       1.00       12.00     4274.00      165.00      350.00      144.00
     51288.00     1980.00     4200.00 18267076.00   705210.00  1495900.00
     27225.00    57750.00   122500.00]
 [       1.00       15.80     3415.00      110.00      231.00      249.64
     53957.00     1738.00     3649.80 11662225.00   375650.00   788865.00
     12100.00    25410.00    53361.00]]
```

Notice that, rather conveniently, the PolynomialFeatures function has also added a column of '1's to the data. This follows the pattern introduced in the last session for dealing with the constant term.

Also, notice that the coefficients are generally larger for the higher-power parameters. In a later section we will return to this as it is indicative of a problem that can occur with higher-order models. For now, just think about the relative size of the parameters and what issues that might cause.

Having generated an extended set of polynomial features, we can now simply apply a linear regression model to the new extended set! This effectively gives us terms representing polynomial versions of the features and hence is a polynomial model.

The beauty of this is that we can just re-use our linear model fitting tools.

```
polynomial_model = LinearRegression()
polynomial_model.fit(X_train_poly, Y_train)
print("Model coefficients   = \n" , polynomial_model.coef_)
print(f"Constant term (bias) = {polynomial_model.intercept_:.2f}")
```

```
Model coefficients   =
 [ 0.00 -2.87 -0.00 -0.21 -0.11  0.04  0.00 -0.00 -0.00 -0.00  0.00  0.00
 -0.00  0.00 -0.00]
Constant term (bias) = 85.64
```

As previously, we can apply this model to both the original training data and to the test data:

```
y_train_predicted = polynomial_model.predict(X_train_poly)
y_test_predicted = polynomial_model.predict(
    poly_features.fit_transform(X_test))
```

Then measure the model quality for each of these cases:

```
rmse_train = np.sqrt(mean_squared_error(Y_train, y_train_predicted))
r2_train = r2_score(Y_train, y_train_predicted)

rmse_test = np.sqrt(mean_squared_error(Y_test, y_test_predicted))
r2_test = r2_score(Y_test, y_test_predicted)

print("R2:")
print(f"  Train = {r2_train:.2f}")
print(f"  Test  = {r2_test:.2f}")
print("RMSE:")
print(f"  Train = {rmse_train:.2f}")
print(f"  Test  = {rmse_test:.2f}")
```

```
R2:
  Train = 0.76
  Test  = 0.72
RMSE:
  Train = 3.81
  Test  = 4.05
```

Notice that both the R2 and RMSE values are somewhat better in the case of the polynomial model than they were above in the linear model.

But there is also some evidence here that the model is moving towards over-fitting the data. The two scores are very much improved for the training data (they indicate a good fit) but only improved by a small amount for the testing data.

In other words, the second-order polynomial model is better than the original model - but not as much better as we might think without careful testing with new data!

Experiment by generating models of higher-degree polynomials. That is, try substituting different values for the 'degree' in the line:

```
poly_features = PolynomialFeatures(degree=2)
```

Observe the behaviour and determine at which degree the model starts to over-fit the data. That is, the polynomial degree at which the model actually starts to perform less well with new test data.

4.11 Final thoughts

Having built this model you might consider going back and experimenting with various things:

- Rather than deleting rows with missing values - why not try to impute missing values?
- Did I make a good decision about the features I selected for modelling?
  - Could some of the other features be used?
  - Maybe we don't need all of these 4 features - they could be rather redundant.