Featurization, Model Selection & Tuning - Linear Regression

Why is regularization required ?

We are well aware of the issue of 'Curse of dimensionality', where the no. of columns are so huge that the no. of rows does not cover all the permutation and combinations that is applicable for this dataset. For eg: Data having 10 columns should have 10! rows but it has only 1000 rows

Therefore,when we depict this graphically there would be lot of white spaces as the datapoints for those regions may not be covered in the dataset.

If a linear regression model is tested over such a data, the model will tend to overfit this data by having sharp peaks & slopes. Such a model would have 100% training accuracy but would definitely fail in the test environment.

Thus arose the need of introducing slight errors in the form of giving smooth bends instead of sharp peaks (thereby reducing overfit).This is achieved by tweaking the model parameters (coefficients) and the hyperparameters (penalty factor).

Agenda

- Perform basic EDA
- Scale data and apply Linear, Ridge & Lasso Regression with Regularization
- Compare the r^2 score to determine which of the above regression methods gives the highest score
- Compute Root mean squared error (RMSE) which inturn gives a better score than r^2
- Finally use a scatter plot to graphically depict the correlation between actual and predicted mpg values

1. Import packages and observe dataset

In [1]:
```
# Import Numerical Libraries
import pandas as pd
import numpy as np
```

In [2]:
```
# Import Graphical Plotting libraries
import seaborn as sns
import matplotlib.pyplot as plt
```

In [3]:
```
# Import Linear Regression Machine Learning Libraries
from sklearn import preprocessing
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.metrics import r2_score
```

In [4]:
```
data = pd.read_csv(r'C:\Users\Windows10 Pro\Downloads\DataScience_AI\2025\3. Jun202
data
```

Out[4]:

	mpg	cyl	disp	hp	wt	acc	yr	origin	car_type	car_name
0	18.0	8	307.0	130	3504	12.0	70	1	0	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	0	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	0	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	0	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	0	ford torino
...	...	...	...	...	...	...	...	...	...	...
393	27.0	4	140.0	86	2790	15.6	82	1	1	ford mustang gl
394	44.0	4	97.0	52	2130	24.6	82	2	1	vw pickup
395	32.0	4	135.0	84	2295	11.6	82	1	1	dodge rampage
396	28.0	4	120.0	79	2625	18.6	82	1	1	ford ranger
397	31.0	4	119.0	82	2720	19.4	82	1	1	chevy s-10

398 rows × 10 columns

In [5]:
```
data.head()   # mpg is dependent (y) varible
```

Out[5]:

	mpg	cyl	disp	hp	wt	acc	yr	origin	car_type	car_name
0	18.0	8	307.0	130	3504	12.0	70	1	0	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	0	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	0	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	0	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	0	ford torino

In [6]:
```
# Drop Car Name
# Replace origin into 1,2,3,  don't forget dummies
# Replace ? with nan
# Replace all nan with median
```

```
data = data.drop(['car_name'], axis=1)
data['origin'] = data['origin'].replace({1:'america', 2: 'europe', 3:'asia'})
data = pd.get_dummies(data,columns = ['origin'],dtype=int)   # converts categroical
data = data.replace('?', np.nan)
data = data.apply(lambda x:x.fillna(x.median()), axis=0)
```

In [7]:
```
data.head()
```

Out[7]:

	mpg	cyl	disp	hp	wt	acc	yr	car_type	origin_america	origin_asia	origin_euro
0	18.0	8	307.0	130	3504	12.0	70	0	1	0	
1	15.0	8	350.0	165	3693	11.5	70	0	1	0	
2	18.0	8	318.0	150	3436	11.0	70	0	1	0	
3	16.0	8	304.0	150	3433	12.0	70	0	1	0	
4	17.0	8	302.0	140	3449	10.5	70	0	1	0	

2. Model Building

Here we would like to scale the data as the columns are varied which would result in 1 column dominating the others.

First we divide the data into independent (X) and dependent data (y) then we scale it.

Tip!:

*The reason we don't scale the entire data before and then divide it into train(X) & test(y) is because once you scale the data, the type(data_s) would be numpy.ndarray. It's impossible to divide this data when it's an array. *

Hence we divide type(data) pandas.DataFrame, then proceed to scaling it.

In [8]:
```
X = data.drop(['mpg'], axis=1)  # independent variable
y = data[['mpg']]   # independent variable
```

In [9]:
```
# Scaling data

X_s = preprocessing.scale(X)
X_s = pd.DataFrame(X_s,  columns = X.columns)  # converting scaled data into datafr

y_s = preprocessing.scale(y)
y_s = pd.DataFrame(y_s, columns = y.columns)   # ideally train, test data should be
```

In [16]:
```
# Split into Train, test set
X_train, X_test, y_train, y_test = train_test_split(X_s,y_s,test_size=0.20, random_
X_train.shape
```

Out[16]: (318, 10)

2.a Simple Linear Model

In [12]:
```
#Fit simple linear model and find coefficients
regression_model = LinearRegression()
regression_model.fit(X_train, y_train)

for idx, col_name in enumerate(X_train.columns):
    print('The coefficient for {} is {}'.format(col_name, regression_model.coef_[0]

intercept = regression_model.intercept_[0]
print('The intercept is {}'.format(intercept))
```

```
The coefficient for cyl is 0.24638776053571634
The coefficient for disp is 0.2917709209866447
The coefficient for hp is -0.18081621820393684
The coefficient for wt is -0.667553060986813
The coefficient for acc is 0.06537309205777046
The coefficient for yr is 0.3481770259426718
The coefficient for car_type is 0.3339231253960359
The coefficient for origin_america is -0.08117984631927032
The coefficient for origin_asia is 0.0698609820966492
The coefficient for origin_europe is 0.030003161242288048
The intercept is -0.018006831370923237
```

2.b Regularized Ridge Regression

In [13]:
```
#alpha factor here is lambda (penalty term) which helps to reduce the magnitude of

ridge_model = Ridge(alpha = 0.3)
ridge_model.fit(X_train, y_train)

print('Ridge model coef: {}'.format(ridge_model.coef_))
#As the data has 10 columns hence 10 coefficients appear here
```

```
Ridge model coef: [[ 0.24342352  0.28293268 -0.18074242 -0.65967997  0.06398366  0.3
4745486
   0.33096428 -0.08087356  0.06988696  0.0295866 ]]
```

2.c Regularized Lasso Regression

In [14]:
```
#alpha factor here is lambda (penalty term) which helps to reduce the magnitude of

lasso_model = Lasso(alpha = 0.1)
lasso_model.fit(X_train, y_train)

print('Lasso model coef: {}'.format(lasso_model.coef_))
#As the data has 10 columns hence 10 coefficients appear here
```

```
Lasso model coef: [-0.         -0.         -0.07247557 -0.45867691  0.          0.26
98134
  0.11341188 -0.04988145  0.          0.        ]
```

3. Score Comparison

In [17]:
```
#Model score - r^2 or coeff of determinant
#r^2 = 1-(RSS/TSS) = Regression error/TSS


#Simple Linear Model
print(regression_model.score(X_train, y_train))
print(regression_model.score(X_test, y_test))

print('**************************')
#Ridge
print(ridge_model.score(X_train, y_train))
print(ridge_model.score(X_test, y_test))

print('**************************')
#Lasso
print(lasso_model.score(X_train, y_train))
print(lasso_model.score(X_test, y_test))
```

```
0.8373422857977738
0.8474768646673948
**************************
0.837332956087454
0.847263786646594
**************************
0.8007202116330951
0.8283046020148332
```

Polynomial Features

If you wish to further compute polynomial features, you can use the below code.

In [18]:
```
#poly = PolynomialFeatures(degree = 2, interaction_only = True)

#Fit calculates u and std dev while transform applies the transformation to a parti
#Here fit_transform helps to fit and transform the X_s
#Hence type(X_poly) is numpy.array while type(X_s) is pandas.DataFrame
#X_poly = poly.fit_transform(X_s)
#Similarly capture the coefficients and intercepts of this polynomial feature model
```

4. Model Parameter Tuning

- r^2 is not a reliable metric as it always increases with addition of more attributes even if the attributes have no influence on the predicted variable. Instead we use adjusted r^2

which removes the statistical chance that improves r^2

(adjusted r^2 = r^2 - fluke)

- Scikit does not provide a facility for adjusted r^2... so we use statsmodel, a library that gives results similar to what you obtain in R language
- This library expects the X and Y to be given in one single dataframe

In [19]:
```
data_train_test = pd.concat([X_train, y_train], axis =1)
data_train_test.head()
```

Out[19]:

	cyl	disp	hp	wt	acc	yr	car_type	origin_ame
64	1.498191	1.196232	1.197027	1.376929	-0.750880	-1.085858	-1.062235	0.773
55	-0.856321	-0.925936	-1.160564	-1.343645	1.246054	-1.356642	0.941412	-1.292
317	-0.856321	-0.925936	-0.689046	-0.925095	0.084201	1.080415	0.941412	-1.292
102	-0.856321	-0.925936	-1.527300	-1.206493	1.972212	-0.815074	0.941412	-1.292
358	-0.856321	-0.705077	-0.793827	-0.396587	0.991899	1.351199	0.941412	-1.292

In [20]:
```
import statsmodels.formula.api as smf
ols1 = smf.ols(formula = 'mpg ~ cyl+disp+hp+wt+acc+yr+car_type+origin_america+origi
ols1.params
```

Out[20]:
```
Intercept        -0.018007
cyl               0.246388
disp              0.291771
hp               -0.180816
wt               -0.667553
acc               0.065373
yr                0.348177
car_type          0.333923
origin_america   -0.081180
origin_europe     0.030003
origin_asia       0.069861
dtype: float64
```

In [21]:
```
print(ols1.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                    mpg   R-squared:                       0.837
Model:                            OLS   Adj. R-squared:                  0.833
Method:                 Least Squares   F-statistic:                     176.2
Date:                Sun, 22 Jun 2025   Prob (F-statistic):          7.60e-116
Time:                        15:29:38   Log-Likelihood:                -160.75
No. Observations:                 318   AIC:                             341.5
Df Residuals:                     308   BIC:                             379.1
Df Model:                           9
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -0.0180      0.023     -0.786      0.433      -0.063       0.027
cyl            0.2464      0.096      2.571      0.011       0.058       0.435
disp           0.2918      0.112      2.606      0.010       0.071       0.512
hp            -0.1808      0.073     -2.481      0.014      -0.324      -0.037
wt            -0.6676      0.076     -8.797      0.000      -0.817      -0.518
acc            0.0654      0.037      1.768      0.078      -0.007       0.138
yr             0.3482      0.026     13.266      0.000       0.297       0.400
car_type       0.3339      0.061      5.518      0.000       0.215       0.453
origin_america -0.0812     0.018     -4.448      0.000      -0.117      -0.045
origin_europe  0.0300      0.019      1.600      0.111      -0.007       0.067
origin_asia    0.0699      0.019      3.752      0.000       0.033       0.106
==============================================================================
Omnibus:                       33.608   Durbin-Watson:                   1.862
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               58.482
Skew:                           0.625   Prob(JB):                     2.00e-13
Kurtosis:                       4.688   Cond. No.                     6.58e+15
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly spe
cified.
[2] The smallest eigenvalue is 4.21e-29. This might indicate that there are
strong multicollinearity problems or that the design matrix is singular.
```

In [22]:
```
#Lets check Sum of Squared Errors (SSE) by predicting value of y for test cases and
mse  = np.mean((regression_model.predict(X_test)-y_test)**2)

# root of mean_sq_error is standard deviation i.e. avg variance between predicted a
import math
rmse = math.sqrt(mse)
print('Root Mean Squared Error: {}'.format(rmse))
```

```
Root Mean Squared Error: 0.39853963361814243
```

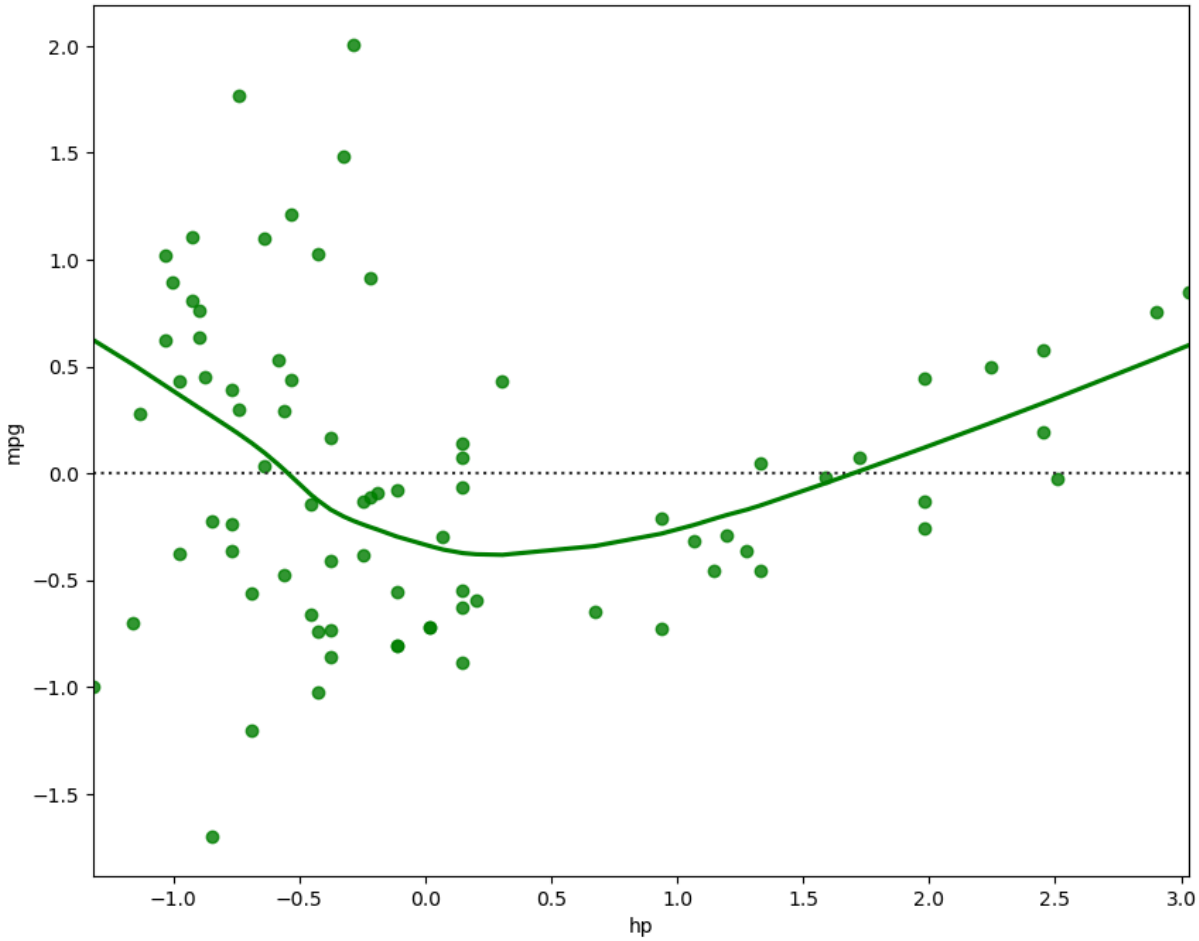
So there is an avg. mpg difference of 0.37 from real mpg

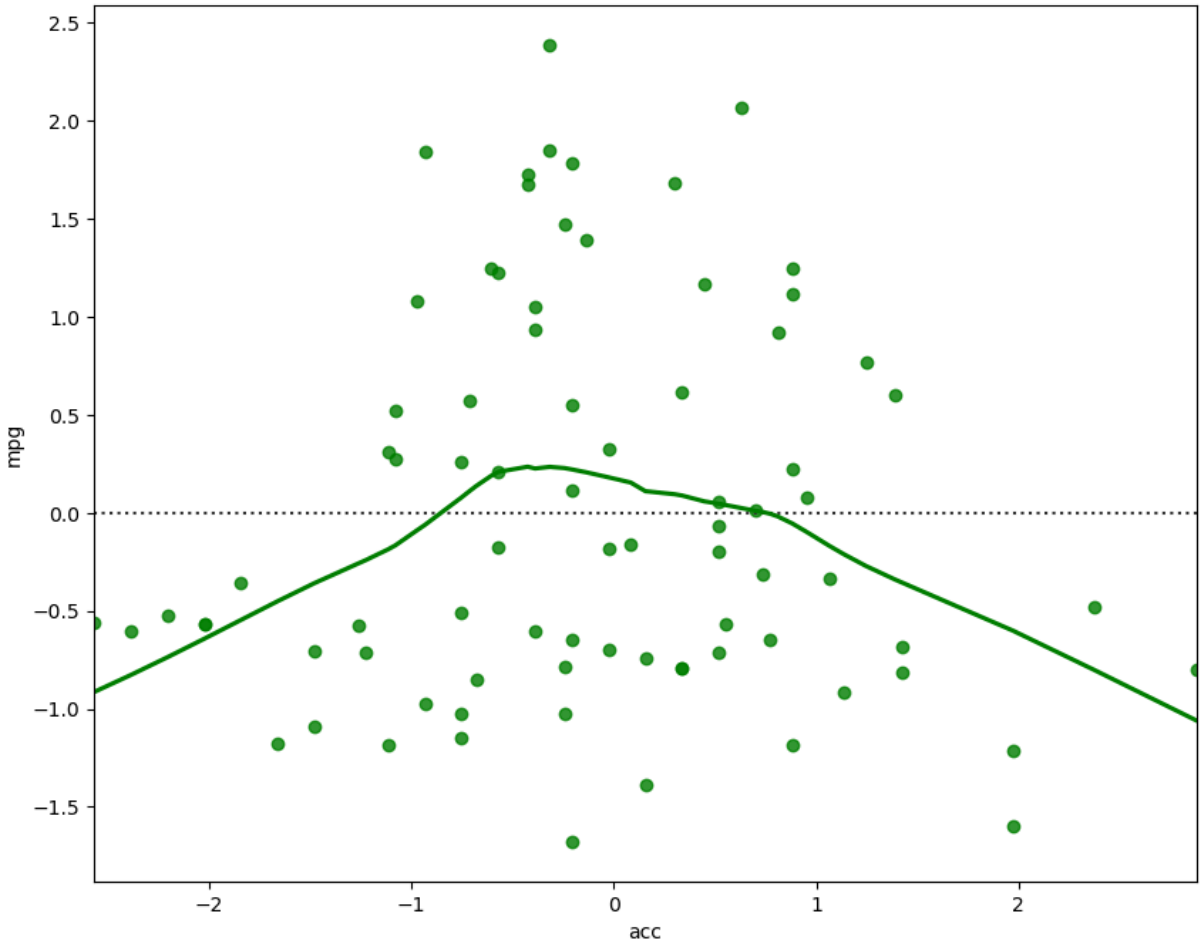
In [23]:
```
# Is OLS a good model ? Lets check the residuals for some of these predictor.

fig = plt.figure(figsize=(10,8))
sns.residplot(x= X_test['hp'], y= y_test['mpg'], color='green', lowess=True )
```

```
fig = plt.figure(figsize=(10,8))
sns.residplot(x= X_test['acc'], y= y_test['mpg'], color='green', lowess=True )
```

Out[23]: <Axes: xlabel='acc', ylabel='mpg'>

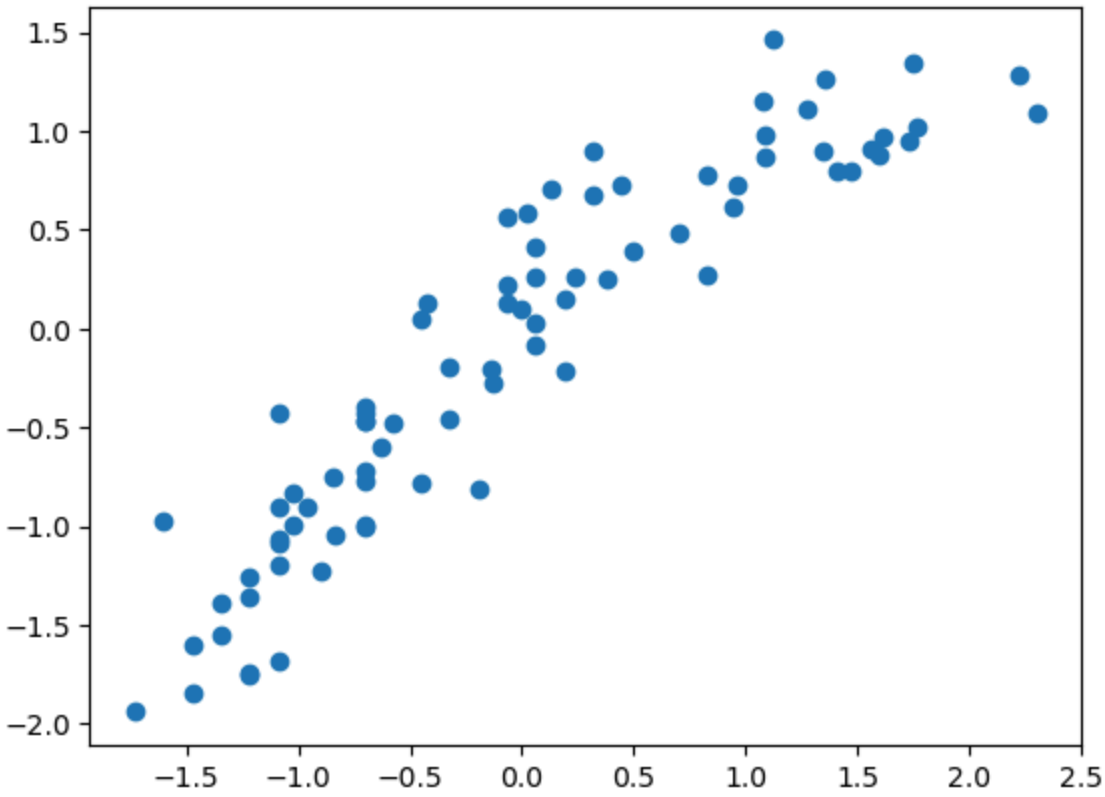




In [24]:
```
# predict mileage (mpg) for a set of attributes not in the training or test set
y_pred = regression_model.predict(X_test)

# Since this is regression, plot the predicted y value vs actual y values for the t
# A good model's prediction will be close to actual leading to high R and R2 values
#plt.rcParams['figure.dpi'] = 500
plt.scatter(y_test['mpg'], y_pred)
```

Out[24]:
```
<matplotlib.collections.PathCollection at 0x2253db012b0>
```



5. Inference

Both Ridge & Lasso regularization performs very well on this data, though Ridge gives a better score. The above scatter plot depicts the correlation between the actual and predicted mpg values.

This kernel is a work in progress.

In [ ]: