

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 [23]: # Importing numerical library
import numpy as np
import pandas as pd
# importing graphical plotting libraries
import matplotlib.pyplot as plt

# Importing Linear regression ML 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 [16]: # reading dataset
data = pd.read_csv(r"C:\Users\mohap\Downloads\car-mpg.csv")

In [17]: data

Out[17]:

	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 [18]: data.head()

Out[18]:

	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 [19]:

```
# Droping car name due to categorical,
# replacing origion to 1,2,3 dont forgetting to get_dummies
# in dataset ?, so converting to nan
# replacing all nan to median
# Drop 'car_name' column if it exists
if 'car_name' in data.columns:
    data = data.drop(['car_name'], axis=1)

# Check if 'origin' column exists before processing
```

```

if 'origin' in data.columns:
    # Replace 'origin' values with categorical names
    data['origin'] = data['origin'].replace({1: 'america', 2: 'europe', 3: 'asia'})

    # Convert 'origin' to dummy variables
    data = pd.get_dummies(data, columns=['origin'])

# Replace '?' with NaN
data = data.replace('?', np.nan)

# Convert 'hp' to numeric, coercing errors to NaN
data['hp'] = pd.to_numeric(data['hp'], errors='coerce')

# Fill NaN values with the median
data = data.apply(lambda x: x.fillna(x.median()), axis=0)

```

In [20]: `data.head()`

Out[20]:

	mpg	cyl	disp	hp	wt	acc	yr	car_type	origin_america	origin_asia	origin_eur
0	18.0	8	307.0	130.0	3504	12.0	70	0	True	False	F
1	15.0	8	350.0	165.0	3693	11.5	70	0	True	False	F
2	18.0	8	318.0	150.0	3436	11.0	70	0	True	False	F
3	16.0	8	304.0	150.0	3433	12.0	70	0	True	False	F
4	17.0	8	302.0	140.0	3449	10.5	70	0	True	False	F

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 [21]:

```

x = data.drop(['mpg'], axis=1) # independent variable
y = data['mpg'] # dependent variable

```

In []:

```

# scaling data
x_s = preprocessing.scale(x) # scaling data
x_s = pd.DataFrame(x_s, columns=x.columns) # converting to dataframe

```

```
y_s = preprocessing.scale(y) # scaling data
# y is a Series, so use y.name as the column name in DataFrame
y_s = pd.DataFrame(y_s, columns=[y.name]) # ideally train, test should be in column
```

In []: x_s

Out[]:

	cyl	disp	hp	wt	acc	yr	car_type	origin_ame
0	1.498191	1.090604	0.673118	0.630870	-1.295498	-1.627426	-1.062235	0.773
1	1.498191	1.503514	1.589958	0.854333	-1.477038	-1.627426	-1.062235	0.773
2	1.498191	1.196232	1.197027	0.550470	-1.658577	-1.627426	-1.062235	0.773
3	1.498191	1.061796	1.197027	0.546923	-1.295498	-1.627426	-1.062235	0.773
4	1.498191	1.042591	0.935072	0.565841	-1.840117	-1.627426	-1.062235	0.773
...
393	-0.856321	-0.513026	-0.479482	-0.213324	0.011586	1.621983	0.941412	0.773
394	-0.856321	-0.925936	-1.370127	-0.993671	3.279296	1.621983	0.941412	-1.292
395	-0.856321	-0.561039	-0.531873	-0.798585	-1.440730	1.621983	0.941412	0.773
396	-0.856321	-0.705077	-0.662850	-0.408411	1.100822	1.621983	0.941412	0.773
397	-0.856321	-0.714680	-0.584264	-0.296088	1.391285	1.621983	0.941412	0.773

398 rows × 10 columns

In []: y_s

```
Out[ ]:      mpg
```

	mpg
0	-0.706439
1	-1.090751
2	-0.706439
3	-0.962647
4	-0.834543
...	...
393	0.446497
394	2.624265
395	1.087017
396	0.574601
397	0.958913

398 rows × 1 columns

```
In [25]: #Split into train, test set
```

```
X_train, X_test, y_train,y_test = train_test_split(x_s, y_s, test_size = 0.30, random_state=42)
```

```
Out[25]: (278, 10)
```

Simple Linear Model

```
In [27]: #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_[idx]))

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

```
The coefficient for cyl is 0.32102238569161057
The coefficient for disp is 0.324834309184838
The coefficient for hp is -0.22916950059437718
The coefficient for wt is -0.7112101905072288
The coefficient for acc is 0.014713682764190883
The coefficient for yr is 0.3755811949510743
The coefficient for car_type is 0.3814769484233099
The coefficient for origin_america is -0.07472247547584182
The coefficient for origin_asia is 0.044515252035678216
The coefficient for origin_europe is 0.04834854953945382
The intercept is 0.0192841161036397
```

Regularized linear regression

Ridge(L2)

```
In [29]: #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: [ 0.31649043  0.31320707 -0.22876025 -0.70109447  0.01295851  0.37
447352
 0.37725608 -0.07423624  0.04441039  0.04784031]
```

Regularized Lasso REgression

```
In [30]: #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.          -0.01690287 -0.51890013  0.          0.28
138241
 0.1278489 -0.01642647  0.          0.          ]
```

Score Comparision

```
In [32]: #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))

0.8343770256960538
0.8513421387780067
*****
0.8343617931312616
0.8518882171608508
*****
0.7938010766228453
0.8375229615977083

```

Polynomial Features

```

In [34]: 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 $r^2 = r^2 - \text{fluke}$)

- 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 [36]: # Convert y_train to a DataFrame before concatenation
y_train_df = pd.DataFrame(y_train, columns=[y.name]) # Use y.name to retain the co
data_train_test = pd.concat([X_train, y_train_df], axis=1)
data_train_test.head()

```

Out[36]:

	cyl	disp	hp	wt	acc	yr	car_type	origin_ame
350	-0.856321	-0.849116	-1.081977	-0.893172	-0.242570	1.351199	0.941412	0.773
59	-0.856321	-0.925936	-1.317736	-0.847061	2.879909	-1.085858	0.941412	-1.292
120	-0.856321	-0.695475	0.201600	-0.121101	-0.024722	-0.815074	0.941412	-1.292
12	1.498191	1.983643	1.197027	0.934732	-2.203196	-1.627426	-1.062235	0.773
349	-0.856321	-0.983552	-0.951000	-1.165111	0.156817	1.351199	0.941412	-1.292

In [38]: # Install statsmodels if not already installed
%pip install statsmodels

```
import statsmodels.formula.api as smf
ols1 = smf.ols(formula = 'mpg ~ cyl+disp+hp+wt+acc+yr+car_type+origin_americ
ols1.params
```

```
Defaulting to user installation because normal site-packages is not writeable
Collecting statsmodels
  Downloading statsmodels-0.14.4-cp312-cp312-win_amd64.whl.metadata (9.5 kB)
Requirement already satisfied: numpy<3,>=1.22.3 in c:\users\mohap\appdata\roaming\python\python312\site-packages (from statsmodels) (1.26.4)
Requirement already satisfied: scipy!=1.9.2,>=1.8 in c:\users\mohap\appdata\roaming\python\python312\site-packages (from statsmodels) (1.14.1)
Requirement already satisfied: pandas!=2.1.0,>=1.4 in c:\users\mohap\appdata\roaming\python\python312\site-packages (from statsmodels) (2.2.2)
Collecting patsy>=0.5.6 (from statsmodels)
  Downloading patsy-1.0.1-py2.py3-none-any.whl.metadata (3.3 kB)
Requirement already satisfied: packaging>=21.3 in c:\users\mohap\appdata\roaming\python\python312\site-packages (from statsmodels) (24.1)
Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\mohap\appdata\roaming\python\python312\site-packages (from pandas!=2.1.0,>=1.4->statsmodels) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in c:\users\mohap\appdata\roaming\python\python312\site-packages (from pandas!=2.1.0,>=1.4->statsmodels) (2024.1)
Requirement already satisfied: tzdata>=2022.7 in c:\users\mohap\appdata\roaming\python\python312\site-packages (from pandas!=2.1.0,>=1.4->statsmodels) (2024.1)
Requirement already satisfied: six>=1.5 in c:\users\mohap\appdata\roaming\python\python312\site-packages (from python-dateutil>=2.8.2->pandas!=2.1.0,>=1.4->statsmodels) (1.16.0)
Downloading statsmodels-0.14.4-cp312-cp312-win_amd64.whl (9.8 MB)
----- 0.0/9.8 MB ? eta -:---:-
----- 0.3/9.8 MB ? eta -:---:-
----- 0.8/9.8 MB 2.6 MB/s eta 0:00:04
----- 1.0/9.8 MB 1.6 MB/s eta 0:00:06
----- 1.3/9.8 MB 1.7 MB/s eta 0:00:06
----- 1.6/9.8 MB 1.6 MB/s eta 0:00:06
----- 2.1/9.8 MB 1.6 MB/s eta 0:00:05
----- 2.4/9.8 MB 1.6 MB/s eta 0:00:05
----- 3.1/9.8 MB 1.9 MB/s eta 0:00:04
----- 3.9/9.8 MB 2.2 MB/s eta 0:00:03
----- 4.7/9.8 MB 2.3 MB/s eta 0:00:03
----- 5.5/9.8 MB 2.4 MB/s eta 0:00:02
----- 6.3/9.8 MB 2.6 MB/s eta 0:00:02
----- 7.1/9.8 MB 2.7 MB/s eta 0:00:02
----- 7.6/9.8 MB 2.6 MB/s eta 0:00:01
----- 8.1/9.8 MB 2.6 MB/s eta 0:00:01
----- 8.9/9.8 MB 2.6 MB/s eta 0:00:01
----- 9.7/9.8 MB 2.7 MB/s eta 0:00:01
----- 9.8/9.8 MB 2.7 MB/s eta 0:00:00
Downloading patsy-1.0.1-py2.py3-none-any.whl (232 kB)
Installing collected packages: patsy, statsmodels
Successfully installed patsy-1.0.1 statsmodels-0.14.4
Note: you may need to restart the kernel to use updated packages.
```

```
Out[38]: Intercept      0.024749
          cyl        -0.232505
          disp       0.348422
          hp        -0.243993
          wt         0.123460
          acc       -0.119813
          yr        -0.056436
          car_type    0.262680
          origin_america 0.054458
          origin_europe   -0.047149
          origin_asia    -0.021072
          dtype: float64
```

```
In [39]: print(ols1.summary())
```

Dep. Variable:	mpg	R-squared:	0.040			
Model:	OLS	Adj. R-squared:	-0.006			
Method:	Least Squares	F-statistic:	0.8775			
Date:	Tue, 29 Apr 2025	Prob (F-statistic):	0.546			
Time:	18:29:21	Log-Likelihood:	-282.38			
No. Observations:	200	AIC:	584.8			
Df Residuals:	190	BIC:	617.7			
Df Model:	9					
Covariance Type:	nonrobust					
	coef	std err	t	P> t	[0.025	0.975]
Intercept	0.0247	0.094	0.264	0.792	-0.160	0.210
cyl	-0.2325	0.310	-0.750	0.454	-0.844	0.379
disp	0.3484	0.352	0.991	0.323	-0.345	1.042
hp	-0.2440	0.213	-1.143	0.255	-0.665	0.177
wt	0.1235	0.240	0.514	0.608	-0.350	0.597
acc	-0.1198	0.121	-0.988	0.325	-0.359	0.120
yr	-0.0564	0.112	-0.505	0.614	-0.277	0.164
car_type	0.2627	0.192	1.371	0.172	-0.115	0.641
origin_america	0.0545	0.067	0.812	0.418	-0.078	0.187
origin_europe	-0.0471	0.063	-0.750	0.454	-0.171	0.077
origin_asia	-0.0211	0.066	-0.318	0.751	-0.152	0.110
Omnibus:	9.043	Durbin-Watson:	1.824			
Prob(Omnibus):	0.011	Jarque-Bera (JB):	9.449			
Skew:	0.511	Prob(JB):	0.00887			
Kurtosis:	2.700	Cond. No.	1.02e+16			

Notes:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

[2] The smallest eigenvalue is 1.21e-29. This might indicate that there are strong multicollinearity problems or that the design matrix is singular.

```
In [40]: #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 and actual
import math
rmse = math.sqrt(mse)
print('Root Mean Squared Error: {}'.format(rmse))
```

Root Mean Squared Error: 0.3776693425408783

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

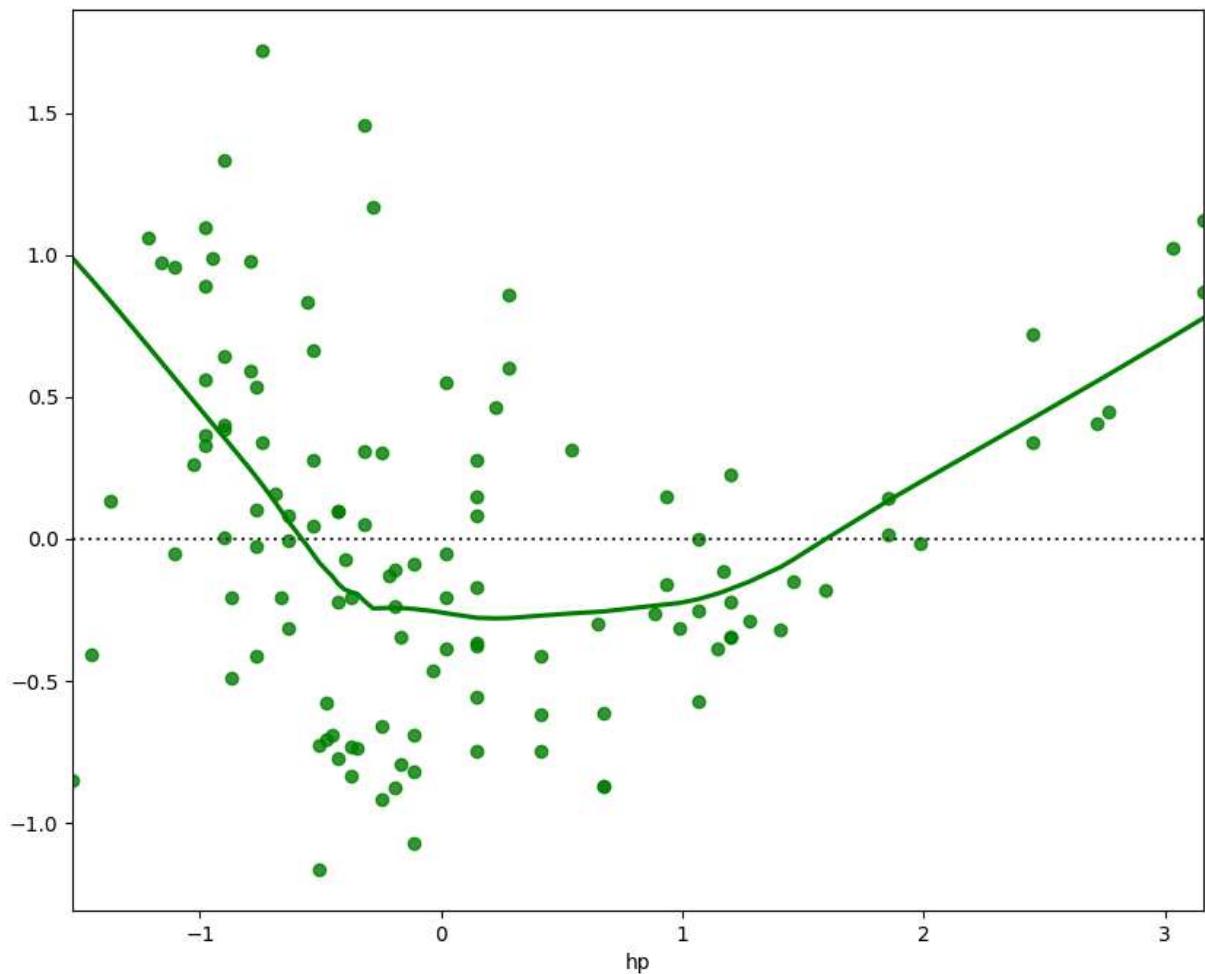
In [43]: *# Is OLS a good model ? Lets check the residuals for some of these predictor.*

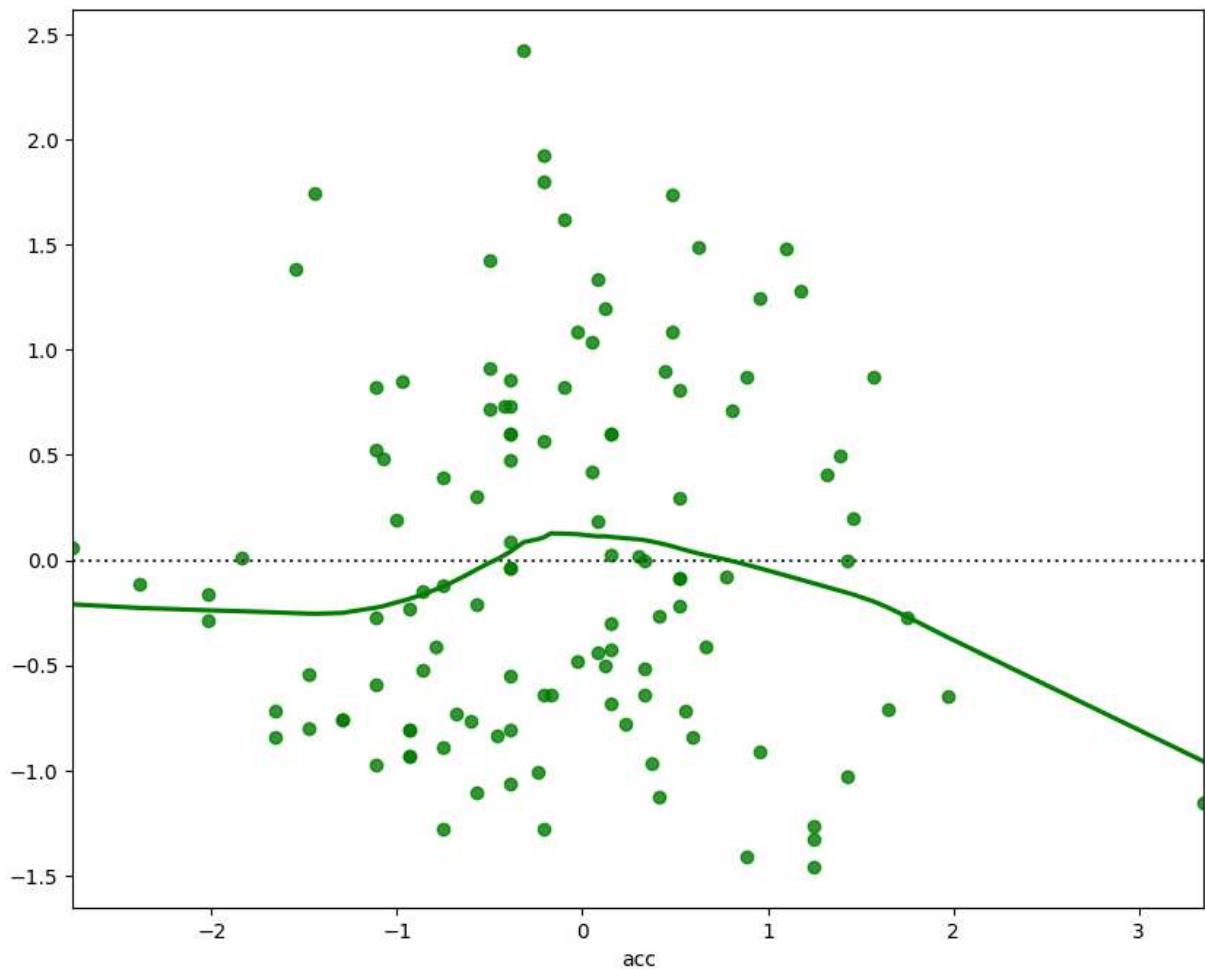
```
# Import Seaborn if not already imported
import seaborn as sns

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

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

Out[43]: <Axes: xlabel='acc'>





```
In [45]: # 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, y_pred)
```

```
Out[45]: <matplotlib.collections.PathCollection at 0x2fd57f1e4e0>
```

