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 premulation 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 set test environment.

thus rows need of introducing slight errors in the form of giving smooth bonds 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 square 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 [15]:
```
# Import numerical Libraries
import numpy as np
```

```python
import pandas as pd

# Import graphical plotting libraries
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

# 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 [16]:
```python
data = pd.read_csv(r'D:\Samsom - All Data\Naresh IT Institute\New folder\car-mpg.csv')
data.head()
```

Out[16]:

	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 [17]:
```python
# Drop non-numeric column
data = data.drop(['car_name'], axis=1)

# Replace origin codes with region labels
data['origin'] = data['origin'].replace({1: 'america', 2: 'europe', 3: 'asia'})

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

# Replace '?' with NaN, then convert applicable columns to numeric
data = data.replace('?', np.nan)
data = data.apply(pd.to_numeric, errors='ignore')  # Ensures '?' replaced columns are numeric
```

```
# Fill missing values with median
data = data.fillna(data.median(numeric_only=True))
```

```
C:\Users\samua\AppData\Local\Temp\ipykernel_19864\2243124916.py:12: FutureWarning: errors='ignore' is deprecated and will raise in a future version. Use to_numeric without passing `errors` and catch exceptions explicitly instead
  data = data.apply(pd.to_numeric, errors='ignore')  # Ensures '?' replaced columns are numeric
```

In [18]:
```
data.head()
```

Out[18]:

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

we have to predict the mpg column given the features.

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(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 ton scaling it.

In [19]:
```
x = data.drop(['mpg'], axis = 1) # indepedent variable
y = data[['mpg']] # dependent variable
```

In [20]:
```
# Scaling the data
x_s = preprocessing.scale(x)
x_s = pd.DataFrame(x_s, columns = x.columns) # converting scaled data into dataframe

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

In [21]:
```
x_s
```

Out[21]:

	cyl	disp	hp	wt	acc	yr	car_type	origin_america	origin_asia	origin_europe
0	1.498191	1.090604	0.673118	0.630870	-1.295498	-1.627426	-1.062235	0.773559	-0.497643	-0.461968
1	1.498191	1.503514	1.589958	0.854333	-1.477038	-1.627426	-1.062235	0.773559	-0.497643	-0.461968
2	1.498191	1.196232	1.197027	0.550470	-1.658577	-1.627426	-1.062235	0.773559	-0.497643	-0.461968
3	1.498191	1.061796	1.197027	0.546923	-1.295498	-1.627426	-1.062235	0.773559	-0.497643	-0.461968
4	1.498191	1.042591	0.935072	0.565841	-1.840117	-1.627426	-1.062235	0.773559	-0.497643	-0.461968
...	...	...	...	...	...	...	...	...	...	...
393	-0.856321	-0.513026	-0.479482	-0.213324	0.011586	1.621983	0.941412	0.773559	-0.497643	-0.461968
394	-0.856321	-0.925936	-1.370127	-0.993671	3.279296	1.621983	0.941412	-1.292726	-0.497643	2.164651
395	-0.856321	-0.561039	-0.531873	-0.798585	-1.440730	1.621983	0.941412	0.773559	-0.497643	-0.461968
396	-0.856321	-0.705077	-0.662850	-0.408411	1.100822	1.621983	0.941412	0.773559	-0.497643	-0.461968
397	-0.856321	-0.714680	-0.584264	-0.296088	1.391285	1.621983	0.941412	0.773559	-0.497643	-0.461968

398 rows × 10 columns

In [22]:
```
y_s
```

Out[22]:

	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 [23]:
```
data.shape
```

Out[23]:
```
(398, 11)
```

In [24]:
```
x_train, x_test, y_train,y_test = train_test_split(x_s, y_s, test_size = 0.20, random_state = 0)
x_train.shape
```

Out[24]:
```
(318, 10)
```

2.a Simple Linear Model

In [25]:

```
# 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][idx]))

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

```
The coefficient for cyl is 0.24638776053571607
The coefficient for disp is 0.29177092098664514
The coefficient for hp is -0.18081621820393654
The coefficient for wt is -0.6675530609868133
The coefficient for acc is 0.06537309205777078
The coefficient for yr is 0.348177025942672
The coefficient for car_type is 0.3339231253960362
The coefficient for origin_america is -0.08117984631927024
The coefficient for origin_asia is 0.06986098209664919
The coefficient for origin_europe is 0.030003161242288134
The intercept is -0.018006831370923248
```

2.b Regularized Ridge Regression

In [26]:

```
# alpha factor here is lambda (penalty term) which helps to reduce the magnitude of coeff

ridge_model = Ridge(alpha = 0.4)
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: [[ 0.24242411  0.28008024 -0.18071842 -0.65711583  0.06353256  0.34721777
   0.32998816 -0.08077573  0.06989674  0.02945199]]
```

2.c Regularized Lasso Regression

In [27]:
```
# alpha factor here is lambda (penalty term) which helps to reduce the magnitude of coeff

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.          0.2698134
 0.11341188 -0.04988145  0.          0.        ]
```

Here we notice many coefficients are turned to 0 indicating drop of those dimensions form the model when we drop the dimeniaont that we eliminat the feature. overfitting reduce, good accuracy

3. Score Comparison

In [28]:
```
# 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
*************************
0.8373258758714116
0.8471902731156344
*************************
0.8007202116330951
0.8283046020148332
```

In [ ]: