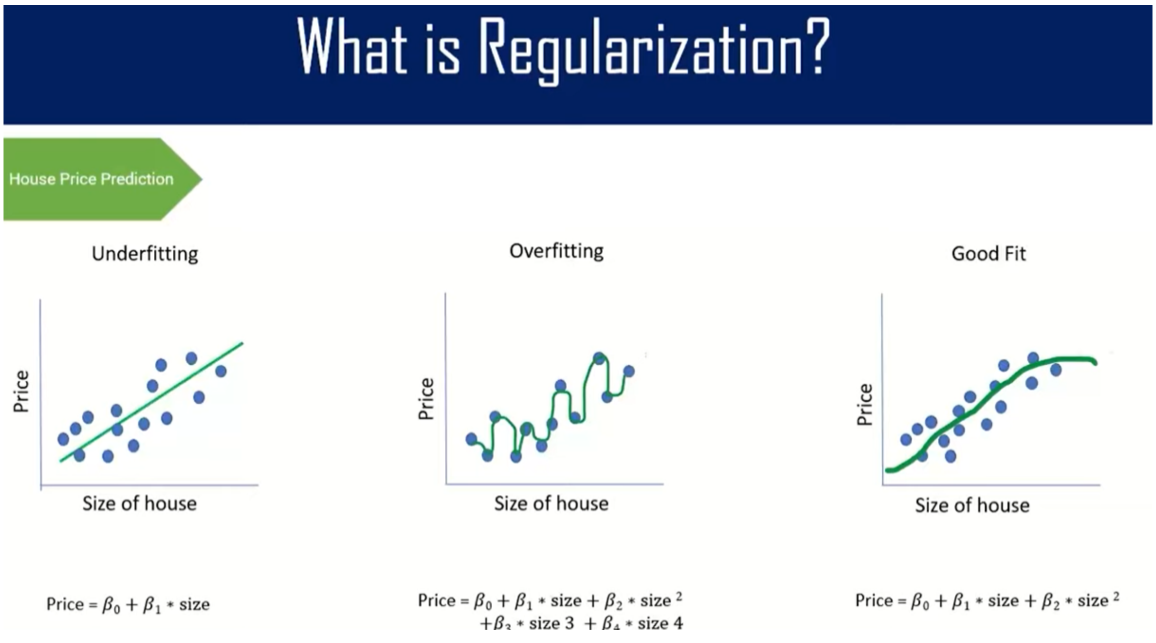
Ridge and Lasso Regression

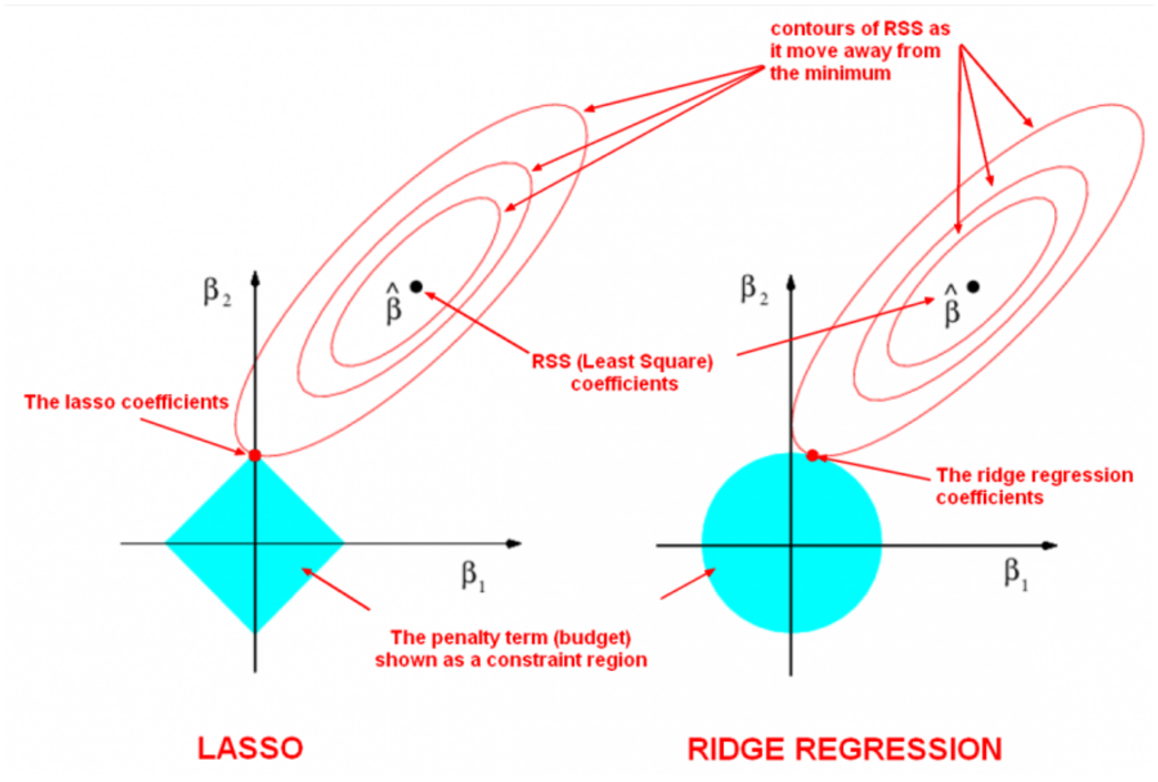
(Code: Subhajit Das)

What is Ridge and Lasso Regression?

Ridge Regression and **Lasso Regression** are two types of regularized linear regression methods that are used to prevent overfitting.

1. **Ridge Regression**: It is an adaptation of the popular and widely used linear regression algorithm. It enhances regular linear regression by slightly changing its cost function, which results in less overfit models. Ridge regression is a method of estimating the coefficients in scenarios where the independent variables are highly correlated[2]. It has been used in many fields including econometrics, chemistry, and engineering.
2. **Lasso Regression**: Lasso (Least Absolute Shrinkage and Selection Operator) is a regression analysis method that performs both variable selection and regularization in order to enhance the prediction accuracy and interpretability of the resulting statistical model. Lasso regression is a regularization technique. It is used over regression methods for a more accurate prediction. This model uses shrinkage, where data values are shrunk towards a central point as the mean.




contours of RSS as it move away from the minimum
β_2
$\hat{\beta}$
RSS (Least Square) coefficients
The lasso coefficients
β_1
The penalty term (budget) shown as a constraint region
The ridge regression coefficients
LASSO
RIDGE REGRESSION

Where we can use Ridge and Lasso Regression?

Ridge and Lasso Regression are used in various fields for different applications:

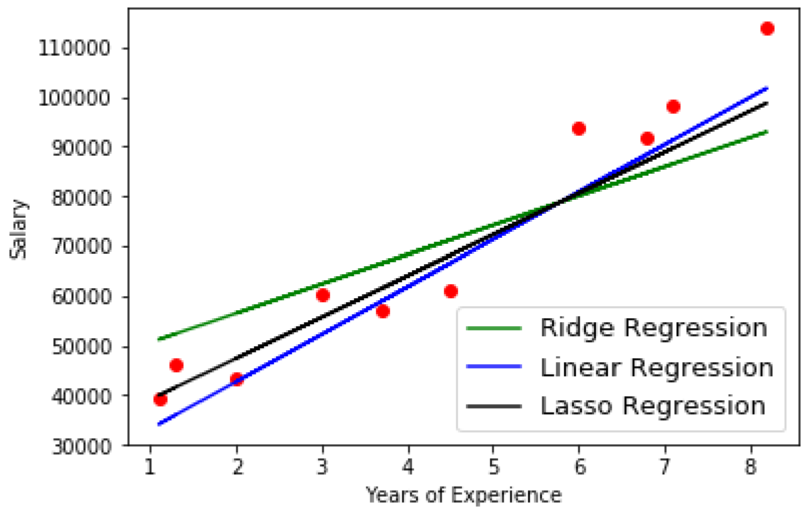
Ridge Regression:

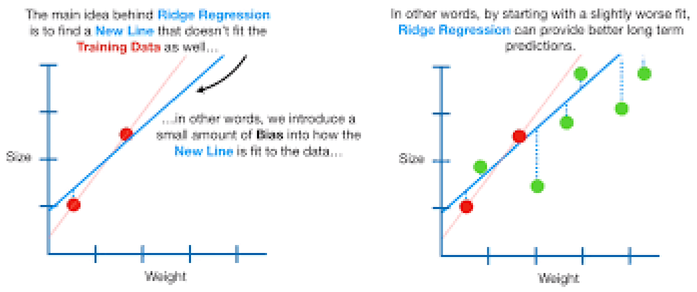
1. **Multicollinearity**: Ridge regression is a machine learning instrument that helps to analyze multiple regression data sets with multicollinearity.
2. **Creating Parsimonious Models**: It is used for the purpose of creating parsimonious models when the number of predictor variables in a given set exceeds the number of observations or when the dataset has multicollinearity.
3. **Mitigating Overfitting**: Ridge regression is a powerful technique for mitigating overfitting in machine learning models.
4. **Econometrics, Chemistry, and Engineering**: It has been used in many fields including econometrics, chemistry, and engineering.

Lasso Regression:

1. **Shrinkage or Regularization**: Lasso regression is used to shrink or regularize coefficients to avoid overfitting and make them work better on different datasets.
2. **High Multicollinearity**: This type of regression is used when the dataset shows high multicollinearity.
3. **Automate Variable Elimination and Feature Selection**: It is used when you want to automate variable elimination and feature selection.
4. **Constructing Forecasting Models**: Lasso regression is used in constructing forecasting models of sectoral probabilities of default in an advanced emerging market economy.

Both methods are particularly useful when dealing with datasets where features are correlated.




The main idea behind Ridge Regression is to find a New Line that doesn't fit the Training Data as well...
...in other words, we introduce a small amount of Bias into how the New Line is fit to the data...
Size
Weight
In other words, by starting with a slightly worse fit, Ridge Regression can provide better long term predictions.
Size
Weight

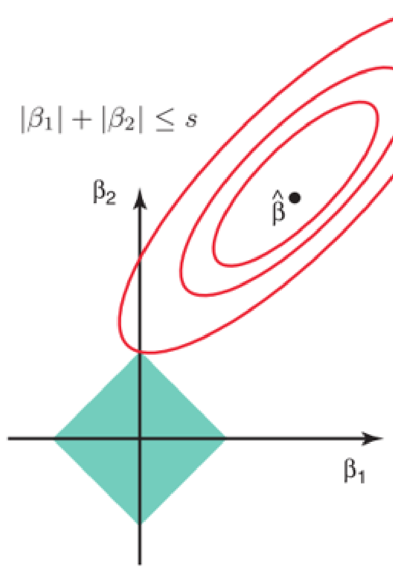
How Ridge and Lasso Regression works:

Ridge Regression:

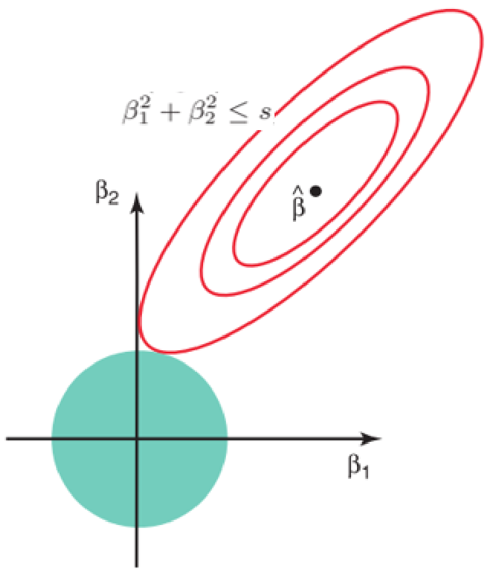
1. **Linear Regression**: Ridge regression is an adaptation of linear regression.
2. **Cost Function**: It enhances regular linear regression by slightly changing its cost function, which results in less overfit models.
3. **Highly Correlated Variables**: Ridge regression is a method of estimating the coefficients in scenarios where the independent variables are highly correlated.
4. **Many Fields**: It has been used in many fields including econometrics, chemistry, and engineering.

Lasso Regression:

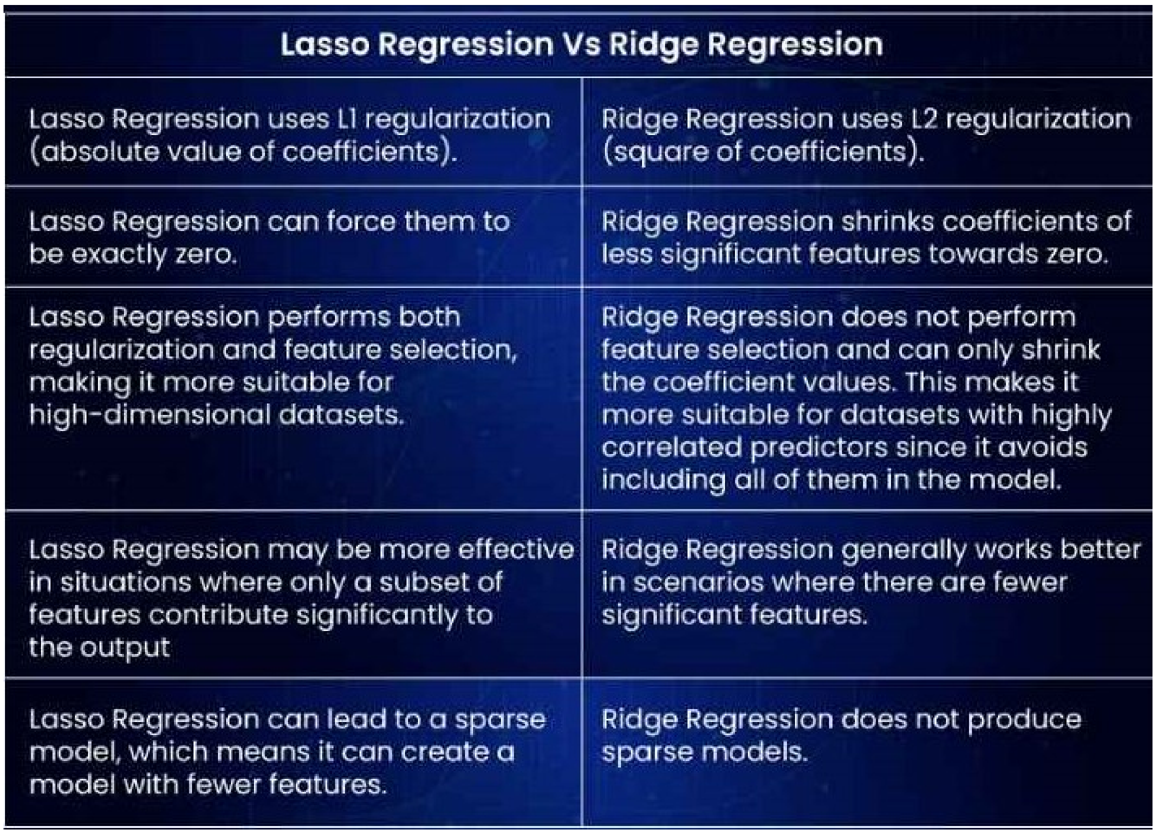
1. **Variable Selection and Regularization**: Lasso (Least Absolute Shrinkage and Selection Operator) is a regression analysis method that performs both variable selection and regularization in order to enhance the prediction accuracy and interpretability of the resulting statistical model.
2. **Regularization Technique**: Lasso regression is a regularization technique. It is used over regression methods for a more accurate prediction.
3. **Shrinkage**: This model uses shrinkage, where data values are shrunk towards a central point as the mean.



Lasso Regression

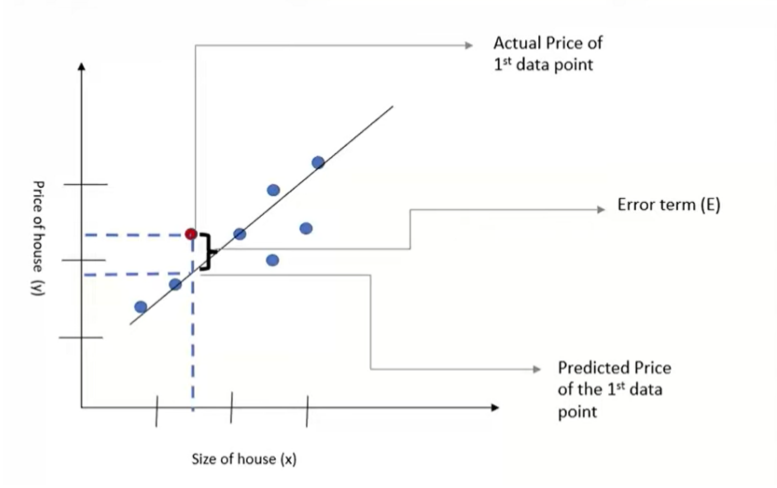


Ridge Regression



Lasso Regression Vs Ridge Regression	
Lasso Regression uses L1 regularization (absolute value of coefficients).	Ridge Regression uses L2 regularization (square of coefficients).
Lasso Regression can force them to be exactly zero.	Ridge Regression shrinks coefficients of less significant features towards zero.
Lasso Regression performs both regularization and feature selection, making it more suitable for high-dimensional datasets.	Ridge Regression does not perform feature selection and can only shrink the coefficient values. This makes it more suitable for datasets with highly correlated predictors since it avoids including all of them in the model.
Lasso Regression may be more effective in situations where only a subset of features contribute significantly to the output	Ridge Regression generally works better in scenarios where there are fewer significant features.
Lasso Regression can lead to a sparse model, which means it can create a model with fewer features.	Ridge Regression does not produce sparse models.

Cost Function for Linear Regression


Actual Price of 1st data point
Error term (E)
Predicted Price of the 1st data point
Price of house (y)
Size of house (x)

Error Term (E) = ($y_{actual} - y_{predicted}$)

$E_1 = (y_{1\ actual} - y_{1\ predicted})$

$E_2 = (y_{2\ actual} - y_{2\ predicted})$

⋮

$E_n = (y_{n\ actual} - y_{n\ predicted})$

Square the error terms

$E_n = (y_{n\ actual} - y_{n\ predicted})^2$

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

Auto-mpg Dataset

In [ ]:
```
auto_df = pd.read_csv("/content/drive/MyDrive/ML and DL DataSets/13_auto-mpg
auto_df.head()
```

Out[2]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	ford torino

In [ ]:
```
auto_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   mpg           398 non-null    float64
 1   cylinders     398 non-null    int64  
 2   displacement  398 non-null    float64
 3   horsepower    398 non-null    object 
 4   weight        398 non-null    int64  
 5   acceleration  398 non-null    float64
 6   model year    398 non-null    int64  
 7   origin        398 non-null    int64  
 8   car name      398 non-null    object 
dtypes: float64(3), int64(4), object(2)
memory usage: 28.1+ KB
```

In [ ]:
```
auto_df.dtypes
```

Out[4]:
```
mpg             float64
cylinders         int64
displacement    float64
horsepower       object
weight            int64
acceleration    float64
model year        int64
origin            int64
car name         object
dtype: object
```

Data Cleaning (Horsepower)

In [ ]:
```
# Viwing the abnormal horsepower data
auto_df[auto_df.horsepower.str.isdigit()==False]
```

Out[5]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
32	25.0	4	98.0	?	2046	19.0	71	1	ford pinto
126	21.0	6	200.0	?	2875	17.0	74	1	ford maverick
330	40.9	4	85.0	?	1835	17.3	80	2	renault lecar deluxe
336	23.6	4	140.0	?	2905	14.3	80	1	ford mustang cobra
354	34.5	4	100.0	?	2320	15.8	81	2	renault 18i
374	23.0	4	151.0	?	3035	20.5	82	1	amc concord dl

In [ ]:
```
# Replace ? with nan
# Replaces all instances of ‘?’ in the ‘horsepower’ column with np.nan (whic
auto_df['horsepower'] = auto_df['horsepower'].replace('?', np.nan)
```

In [ ]:
```
# Filling with median value
auto_df['horsepower'] = auto_df['horsepower'].fillna(auto_df['horsepower'].m
```

In [ ]:
```
# Viewing the dataset
auto_df.head(6)
```

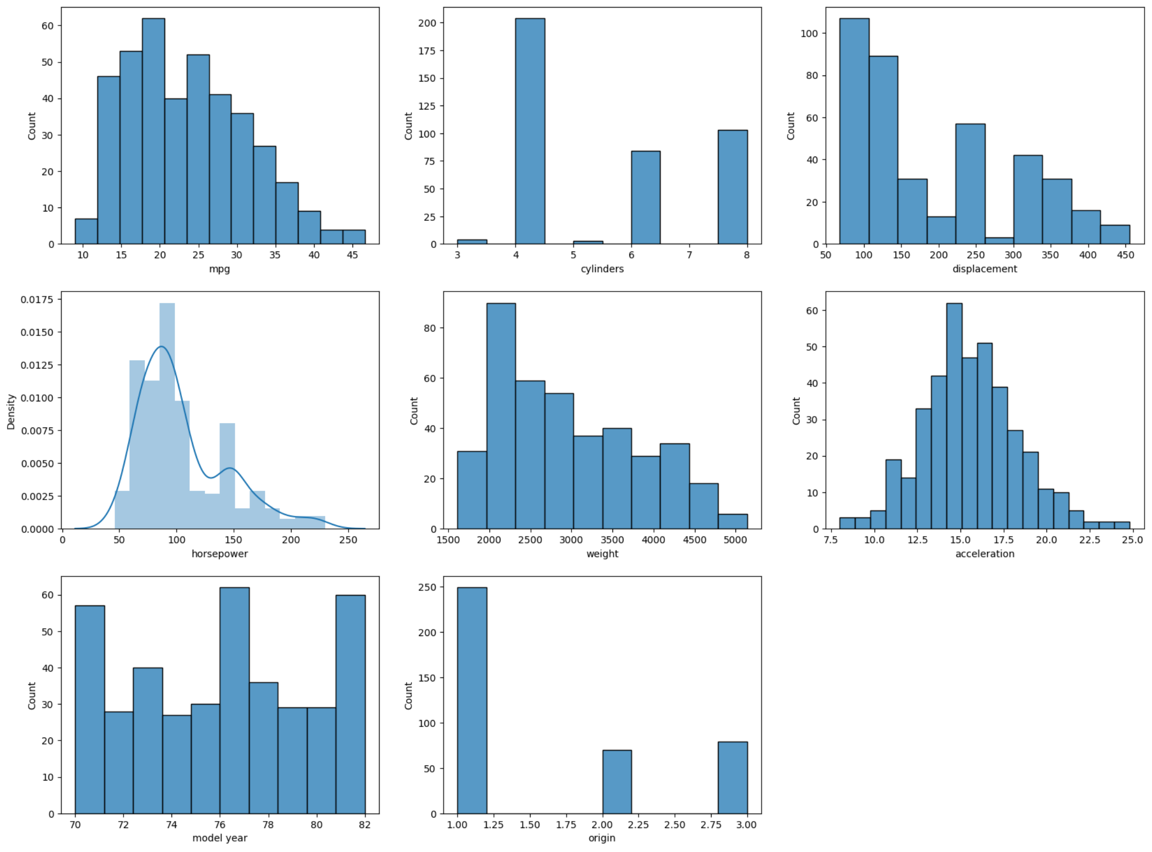
Out[8]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin	car name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	ford torino
5	15.0	8	429.0	198	4341	10.0	70	1	ford galaxie 500

Data Visualizations

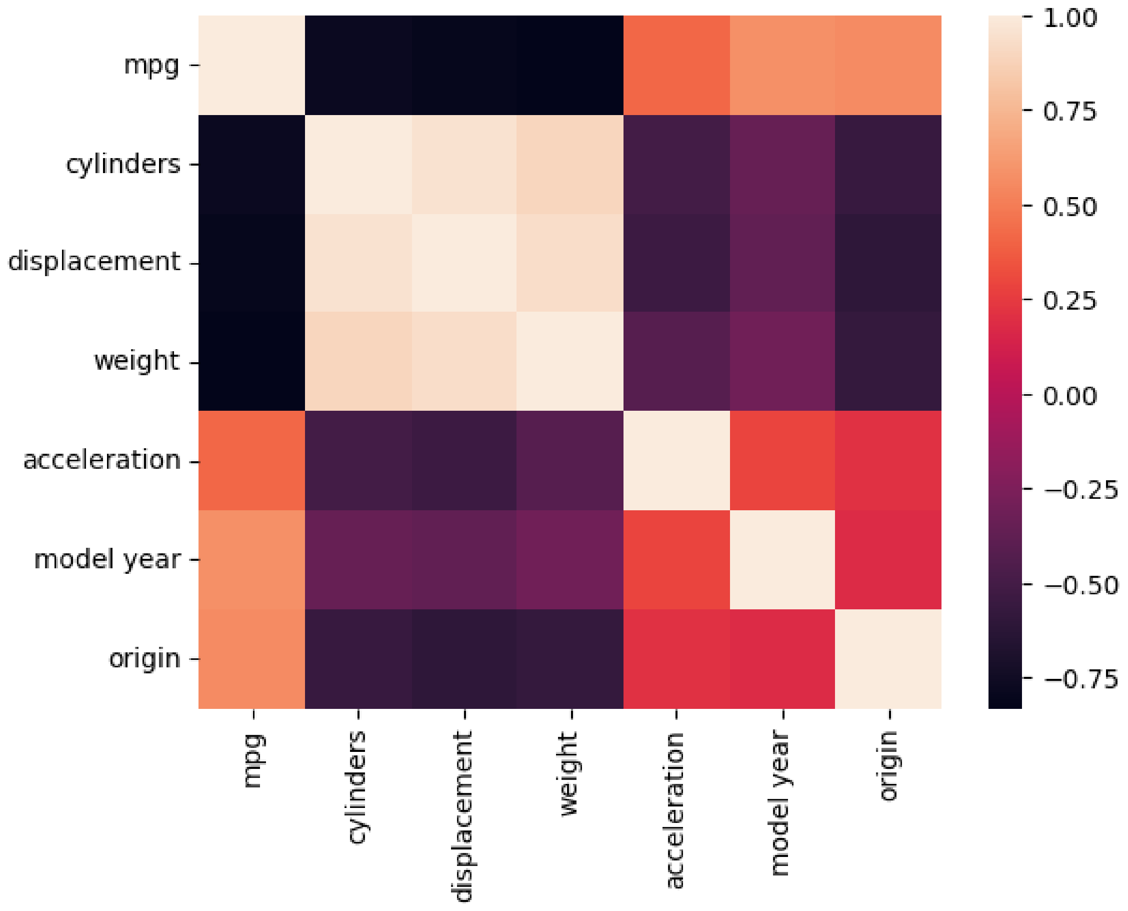
In [ ]:
```python
plt.figure(figsize = (20, 15))

plt.subplot(3, 3, 1)
sns.histplot(auto_df['mpg'])

plt.subplot(3, 3, 2)
sns.histplot(auto_df['cylinders'])

plt.subplot(3, 3, 3)
sns.histplot(auto_df['displacement'])

plt.subplot(3, 3, 4)
sns.distplot(auto_df['horsepower']) # sns.distplot() automatically removes N

plt.subplot(3, 3, 5)
sns.histplot(auto_df['weight'])

plt.subplot(3, 3, 6)
sns.histplot(auto_df['acceleration'])

plt.subplot(3, 3, 7)
sns.histplot(auto_df['model year'])

plt.subplot(3, 3, 8)
sns.histplot(auto_df['origin'])

plt.show()
```

```
<ipython-input-9-870524a27db6>:13: 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 wi
th
similar flexibility) or `histplot` (an axes-level function for histogram
s).

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

  sns.distplot(auto_df['horsepower']) # sns.distplot() automatically remov
es NaN values and can handle non-numeric values better. In histplot() it c
an raise Type Error
```



In [ ]:
```
# Building the Correlation Matrix
# Creating a heatmap of the correlation matrix of the DataFrame. This can be
sns.heatmap(auto_df.corr(numeric_only=True)) # More closer to 1, it denotes
```

Out[10]:
```
<Axes: >
```



In [ ]:
```
auto_df.corr()
```

```
<ipython-input-11-4682980c0113>:1: FutureWarning: The default value of num
eric_only in DataFrame.corr is deprecated. In a future version, it will de
fault to False. Select only valid columns or specify the value of numeric_
only to silence this warning.
  auto_df.corr()
```

Out[11]:

	mpg	cylinders	displacement	weight	acceleration	model year	origin
mpg	1.000000	-0.775396	-0.804203	-0.831741	0.420289	0.579267	0.563450
cylinders	-0.775396	1.000000	0.950721	0.896017	-0.505419	-0.348746	-0.562543
displacement	-0.804203	0.950721	1.000000	0.932824	-0.543684	-0.370164	-0.609409
weight	-0.831741	0.896017	0.932824	1.000000	-0.417457	-0.306564	-0.581024
acceleration	0.420289	-0.505419	-0.543684	-0.417457	1.000000	0.288137	0.205873
model year	0.579267	-0.348746	-0.370164	-0.306564	0.288137	1.000000	0.180662
origin	0.563450	-0.562543	-0.609409	-0.581024	0.205873	0.180662	1.000000

Pre-processing

In [ ]:
```
auto_df = auto_df.drop('car name', axis = 1)
auto_df.tail()
```

Out[12]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model year	origin
393	27.0	4	140.0	86	2790	15.6	82	1
394	44.0	4	97.0	52	2130	24.6	82	2
395	32.0	4	135.0	84	2295	11.6	82	1
396	28.0	4	120.0	79	2625	18.6	82	1
397	31.0	4	119.0	82	2720	19.4	82	1

In [ ]:
```
auto_df.shape
```

Out[13]:
```
(398, 8)
```

Separating features and labels

In [ ]:
```
x = auto_df.drop(['mpg'], axis = 1)
x.tail()
```

Out[14]:

	cylinders	displacement	horsepower	weight	acceleration	model year	origin
393	4	140.0	86	2790	15.6	82	1
394	4	97.0	52	2130	24.6	82	2
395	4	135.0	84	2295	11.6	82	1
396	4	120.0	79	2625	18.6	82	1
397	4	119.0	82	2720	19.4	82	1

In [ ]:
```
y = auto_df['mpg']
y.head()
```

Out[15]:
```
0    18.0
1    15.0
2    18.0
3    16.0
4    17.0
Name: mpg, dtype: float64
```

Scaling the data

In [ ]:
```
# The preprocessing.scale function in the sklearn.preprocessing module is us
''' This function is helpful with largely sparse datasets. It subtracts the
    This brings all your values onto one scale, eliminating the sparsity. I
from sklearn import preprocessing
```

In [ ]:
```
# Scaling the columns of data
x_scaled = preprocessing.scale(x)

y_scaled = preprocessing.scale(y)
```

Train-Test Split

In [ ]:
```
from sklearn.model_selection import train_test_split
```

In [ ]:
```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3)
```

Using Linear Regression Model

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

In [ ]:
```
lin = LinearRegression()
```

In [ ]:
```
# Fit the model
lin.fit(x_train, y_train)
```

Out[22]:
```
LinearRegression()
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [ ]:
```
y_train.head(6)
```

Out[23]:
```
240    30.5
323    27.9
150    26.0
11     14.0
151    31.0
348    37.7
Name: mpg, dtype: float64
```

In [ ]:
```
y_pred = lin.predict(x_train)
print(y_pred)
```

```
[28.53490984 26.28613872 26.50726894 13.67984538 27.86063695 34.47252849
 13.85360683 28.19751439 35.96067005 20.94517545 24.44819919 12.0869313
 23.41305456 14.46977584 32.99884688 32.03563901  9.19608154 30.28331467
 34.02618318 27.21622859 25.29523263 31.15779505 16.90882141 25.8421224
 22.64339605 18.54382513 28.14747579 31.91500348 22.01769412 31.95606783
 35.33830173 10.17098932 18.33366776 19.82757819 30.52249937  6.19954595
 27.03106362 26.22440974 26.93472562 14.7395863  13.29147787 34.54278193
 32.34542217 28.996812   30.54979055 26.70805776 15.5321049  13.36762781
  9.75613138 21.29843217 17.41583521 17.05760572 27.40068432 24.28166699
 28.87878367 29.26944875 15.46962744 26.41373483 11.27056568 10.41112511
 29.95513157 25.25601398 20.38086796 29.55960944 26.04146212 29.16379389
 22.04034527 32.97239378 26.69021878 21.39534528 25.51979664 28.9345504
 21.57474056 14.8582928  23.9343194  17.41653346 28.81759963 25.07938803
 16.56339474 21.33418477 29.27008606 25.13017534 24.53028465 25.64574162
 28.97336806 20.2285668  12.15194534 18.06333491 33.30338941 32.31468575
 26.63390489 29.68138477 24.22291834 10.78851442 27.83946234  9.09223056
 25.13664539 31.74083828 24.57866778 28.67785843 13.22191297 26.0475577
 25.74525101 22.17988202 28.23139455 14.5354821  18.911368   26.46080583
 12.45181328 26.33733136 27.93676396 28.64295037 21.98552469 23.5869094
 17.53552405 22.00326414 25.91990706 19.590008   30.30324949 31.01960361
 13.3484041  15.06415935 15.63936237 30.08017232 35.21504008 15.31326002
 20.26829145 22.01963544 24.42767573 25.30183618 22.12714857 23.35366014
 30.45920399 27.09178629 22.88957129 22.28226428 25.2533599  11.84820991
 29.00489634 13.11170811 26.80091902 13.07210884 11.70022384 23.97118972
 15.81854016 30.57079598 29.08708201 21.80953912 21.62836836 10.00338399
 13.73432862 15.2394583  33.94603436 34.66012861  7.70213365 16.99872932
  7.03180127 26.33905377 24.0254366  31.37408771 24.79672349 15.32850246
 12.62462584 27.95236219 33.42344243 18.85401641 31.86000547 30.78370832
 25.87251781 19.88930923 29.23548103 25.01301062  8.87554956 21.84717709
 28.84328772 25.99626998 25.04180698 21.14783347 29.33770695 24.921558
 29.01588537 23.28129623 33.19713731 19.3532966  32.68959619 30.23385339
 10.67808896 27.6117363  11.37235513 26.11357327 24.15868481 26.34521465
 31.74086941 11.44002135 10.85830632 34.50479294 17.34631697 33.40779944
 33.52419102 31.64630803 17.85365051 10.03849312 29.43896318 31.86720008
 25.34180987 30.98609671 29.87070148 17.16032448  6.45876949 20.8951387
 20.37707603 31.08953604 25.68151146 17.78253548 27.57189588 26.56075495
 32.09433461 27.04797127 17.87012547 24.64520115 20.13141434 25.78420078
 19.25402826 24.52281134 25.35984756 15.59699848 12.32906178 11.65181683
 24.4123948  32.79729224 31.99085279 15.08136877 22.34399318 30.06242317
 34.12681481 20.78681766 12.69410216 32.09360465  6.89385522 28.00556627
 16.33954136 25.47730016 23.03672828 23.96269788 20.74055436 16.70193254
 35.17932952 33.72773472 25.70127001 32.55186648 18.26837309 27.2121144
 21.20962749 21.22293771 24.45985665 26.80421083 32.34887668 10.92123657
 29.90082585 24.69810669 20.13703252 28.86251383 21.80395955 29.22205189
 19.50027025 25.76630265 15.31556615 15.10483566 30.37223856 20.96662503
 18.73741992 19.47931033 30.69242771 28.20020225 25.48879724 12.86841615
 23.97173929 24.89805007]
```

In [ ]:
```
y_test.head(6)
```

Out[25]:
```
292    18.5
126    21.0
273    23.9
116    16.0
331    33.8
335    35.0
Name: mpg, dtype: float64
```

In [ ]:
```python
y_pred = lin.predict(x_test)
print(y_pred)
```

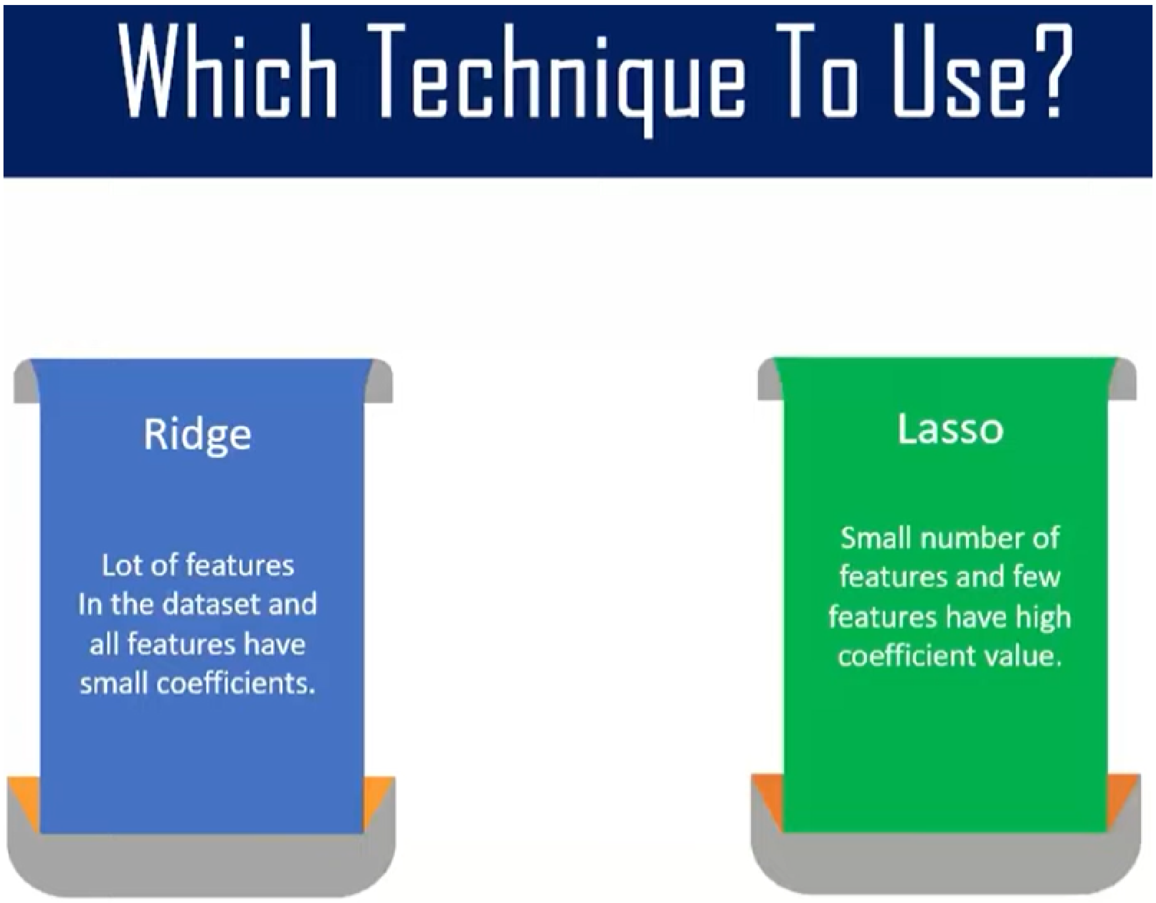
```
[19.74811136 22.41832206 29.30880283 11.16564753 33.35607929 29.22824733
 24.56772491 10.48206356 28.91259555 17.78493535 23.22890722 35.3715484
 22.04959799 27.21599107 12.9721788  27.02740383 24.91065367 25.68071581
 20.37590151 22.47318453 29.5178074  23.98293056 27.04789403 25.65747649
 23.21775326 21.14736844 13.71448257 24.69620349 33.59046558 35.14677951
 19.82710107 11.69526319 20.55529118 30.24274267 30.39944513 27.03003703
 36.15472598 25.00232736 23.68744254 10.20085691 26.35375803 14.75228637
 22.62868821 25.0047747  24.02661471 24.38207978 26.86786396 23.42022823
 19.11105447 25.47284241 24.77565884 30.7131393  19.82153565 28.59873753
 30.03701073 22.2219976  12.23169375 13.93826403 29.56654497 12.42222198
 23.64427896 17.06647607 23.58783071 14.60981599 23.46560036 29.92966684
 10.02071088 21.46313241 35.61460532 29.48268049 22.24799085 22.060838
 20.05485101 28.80092479 28.18360275 22.69992163 30.97439779 15.08291952
 32.70066505 16.08417777 21.71089449 28.09681809 20.68758359 31.01418588
 21.96887949 24.76027714 12.393679   25.57439875 34.29116934 33.97459897
 31.98086973 33.60772128 20.19761137 16.42155341 27.12067245  7.3677356
 35.31430977 21.08826774 22.794407   19.42786851 22.32815187 23.80346087
 21.89257441 25.6847995  30.90160101 29.61884352 29.68801852 27.29546508
 27.62641056 10.33773145 20.73128081 24.1867582  32.09519094 22.56100763
 19.78175629 15.15410912 29.54976262 20.24623354 32.95988597 10.77177122]
```

In [ ]:
```python
# Finding m / slope of line
for m, col_name in enumerate(x_train):
    print("The coefficient for {} is {}".format(col_name, lin.coef_[m]))
# In this case, a coefficient of -0.63.. means that for every one unit incre
```

```
The coefficient for cylinders is -0.679346387332016
The coefficient for displacement is 0.029875991756258456
The coefficient for horsepower is -0.033465375138656085
The coefficient for weight is -0.006522805022109741
The coefficient for acceleration is 0.1928289231949515
The coefficient for model year is 0.7034440270134991
The coefficient for origin is 1.2971592068824689
```

In [ ]:
```python
# Finding y_intercept of model
# The y-intercept of a linear regression model, also known as the bias or co
c = lin.intercept_
print('The y_intercept of this Linear Regression is {}'.format(c))
# y-intercept of -16.54 means that if all the independent var in your model
```

```
The y_intercept of this Linear Regression is -14.228829857898504
```

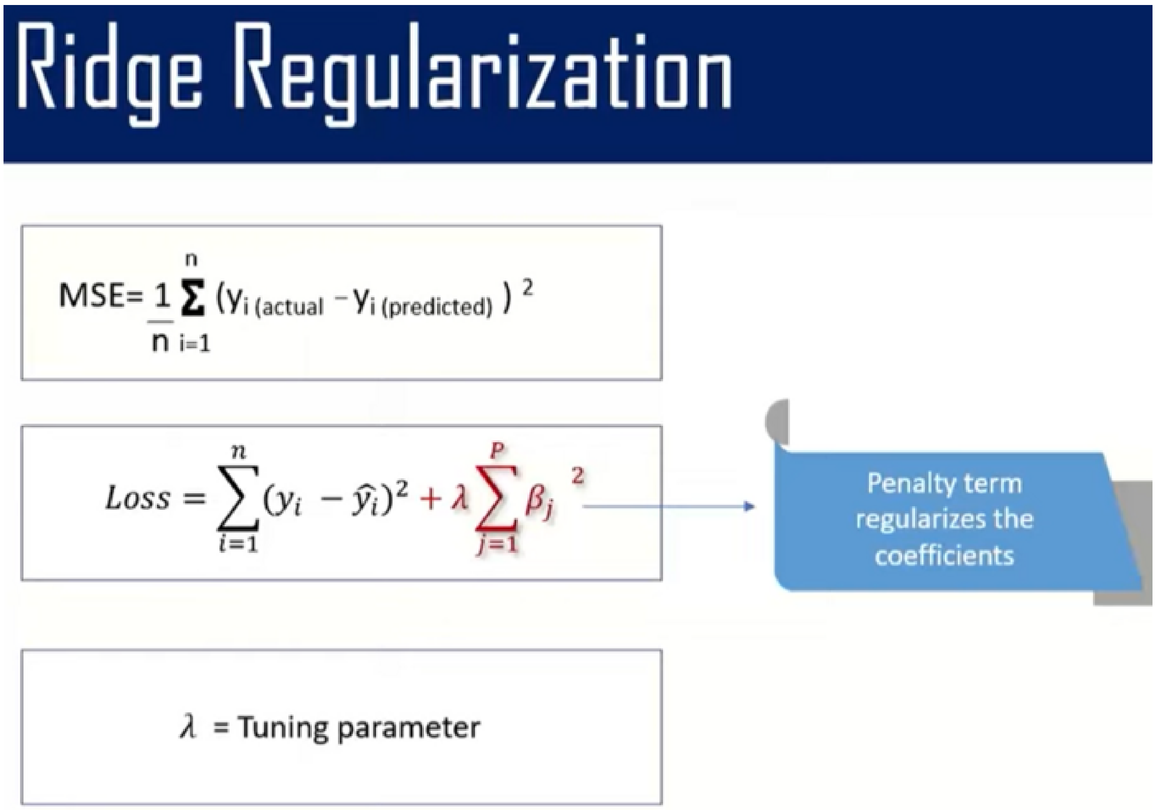

Which Technique To Use?
Ridge
Lot of features In the dataset and all features have small coefficients.
Lasso
Small number of features and few features have high coefficient value.

Parameters used in Ridge and Lasso Regression:

The parameters used in Ridge and Lasso Regression include:

1. **Dependent Variable (Y)**: This is the variable that we want to predict or forecast.
2. **Independent Variables (X)**: These are the variables that we use to predict or forecast the dependent variable.
3. **Intercept (b0)**: This is the predicted value of Y when X is 0.
4. **Slope (b1)**: This is the regression coefficient, which represents the change in the dependent variable for a unit change in the independent variable.
5. **Error Term (ϵ)**: This is the difference between the observed and predicted values.
6. **alpha**: This is a regularization parameter that controls the strength of the penalty term.
7. **fit_intercept**: This specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.
8. **normalize**: This parameter is used to normalize the input variables (X) before regression.
9. **copy_X**: This parameter is used to copy the input variables (X). If False, the input variables may be overwritten during the normalization process.
10. **max_iter**: This is the maximum number of iterations for the solver to converge.
11. **tol**: This is the tolerance for the solution.
12. **warm_start**: When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution.
13. **positive**: When set to True, forces the coefficients to be positive.
14. **random_state**: This parameter is used for shuffling the data.
15. **selection**: If set to ‘random’, a random coefficient is updated every iteration rather than looping over features sequentially by default.

Implementing Ridge Regression (L2) [Reduce Overfiting]



Ridge Regularization

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_{i\,(actual)} - y_{i\,(predicted)})^2$$

$$Loss = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda\sum_{j=1}^{P}\beta_j^{\;2}$$

Penalty term regularizes the coefficients

λ = Tuning parameter

```
In [ ]: from sklearn.linear_model import Ridge
```

```
In [ ]: # Alpha is the regularization strength. It improves the conditioning of the
        rid = Ridge(alpha = 0.4) # The default value is 1.0, but you might need to t
```

```
In [ ]: # Fit the model
        rid.fit(x_train, y_train)
```

```
Out[31]: Ridge(alpha=0.4)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [ ]:
```
y_pred_rid = rid.predict(x_test)
print(y_pred_rid)
```

```
[19.74615416 22.42274078 29.30252987 11.1653999  33.35008765 29.22612737
 24.57144764 10.48303769 28.91720204 17.78316309 23.23342328 35.36615773
 22.04629884 27.21539006 12.97598077 27.0212687  24.91254406 25.67923986
 20.37829171 22.46986068 29.50871845 23.98235584 27.04043377 25.6573163
 23.21750628 21.14632497 13.71805231 24.69574181 33.58940592 35.14198401
 19.83262749 11.69878879 20.5544918  30.24387606 30.40008207 27.02669369
 36.15051501 25.00671146 23.69077975 10.19985767 26.36001612 14.75457798
 22.63064221 25.00730015 24.03358605 24.37637196 26.86701445 23.42167724
 19.10903955 25.4649236  24.77704476 30.70619866 19.81964741 28.59862554
 30.03307729 22.22090788 12.23129461 13.94004862 29.57118831 12.41690501
 23.64455789 17.07185734 23.58561779 14.61501383 23.46864333 29.92282523
 10.01367787 21.46382767 35.60932638 29.48460498 22.24865698 22.05827366
 20.05881558 28.80471509 28.18439721 22.70162971 30.98023414 15.0823848
 32.70375234 16.08605814 21.712371   28.10181253 20.68657536 31.00433724
 21.97969785 24.76862199 12.396603   25.57588951 34.28641435 33.96749441
 31.97561319 33.60297327 20.20047678 16.42426849 27.12241018  7.37457761
 35.3090264  21.08947299 22.79600179 19.42856106 22.32675478 23.79881999
 21.89287643 25.6862545  30.8978782  29.6246988  29.68733128 27.2961458
 27.62588238 10.33279132 20.73562889 24.1897224  32.09100848 22.56259585
 19.78075943 15.15819731 29.54956975 20.24624809 32.95538617 10.76769569]
```

In [ ]:
```
# Finding m / slope of line
for m, col_name in enumerate(x_train):
    print("Ridge model coefficient for {} is {}".format(col_name, rid.coef_[
# Coefficient of 0.020 means that for every one unit increase in the ind var
# In ridge regression coefficient can't be zero
```

```
Ridge model coefficient for cylinders is -0.6751253889822624
Ridge model coefficient for displacement is 0.029774473258055555
Ridge model coefficient for horsepower is -0.03339760977219361
Ridge model coefficient for weight is -0.00652452195552349
Ridge model coefficient for acceleration is 0.19280010448618864
Ridge model coefficient for model year is 0.7033059366008064
Ridge model coefficient for origin is 1.2916467139033134
```

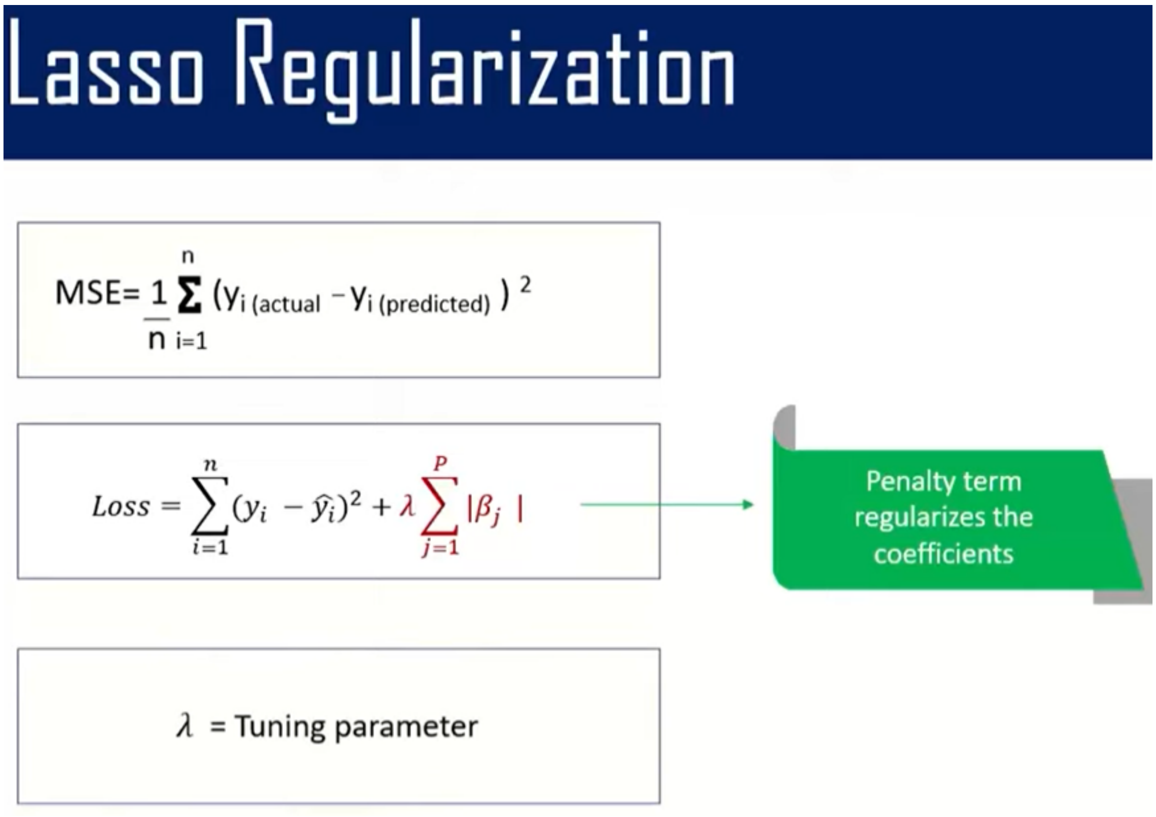
In [ ]:
```
# Finding y_intercept of model
c = rid.intercept_
print('The y_intercept of this Ridge Regression is {}'.format(c))
```

```
The y_intercept of this Ridge Regression is -14.214612196732098
```

In [ ]:
```
''' Here we can see, there is no huge differance in co-eddicient of Linear a
```

Out[35]:
```
' Here we can see, there is no huge differance in co-eddicient of Linear a
nd Ridge(L2) Regression '
```

Implementing Lasso Regression (L1) [Feature Selection]



Lasso Regularization

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_{i\,(actual} - y_{i\,(predicted)})^2$$

$$Loss = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda\sum_{j=1}^{P}|\beta_j|$$

Penalty term regularizes the coefficients

λ = Tuning parameter

```
In [ ]: from sklearn.linear_model import Lasso
```

```
In [ ]: # Alpha is the regularization strength. It improves the conditioning of the
        las = Lasso(alpha = 0.4) # The default value is 1.0, but you might need to t
```

```
In [ ]: # Fit the model
        las.fit(x_train, y_train)

Out[38]: Lasso(alpha=0.4)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [ ]:
```
y_pred_las = las.predict(x_test)
print(y_pred_las)
```

```
[19.2126628  23.19812449 28.05051845 10.99611689 31.9106092  28.92706139
 25.53787201 10.69968835 30.28699481 17.61753592 24.38616798 33.93121197
 21.19635501 26.65502658 13.55226498 25.67520339 25.21587569 25.30856816
 20.82915178 21.62647509 27.75827052 24.12698079 25.40680854 25.76506355
 23.56220043 20.95370599 14.43793786 24.44039824 32.94446349 33.72275513
 21.00600168 12.29714484 20.30603781 30.58211682 30.44736976 26.26528966
 35.08204849 25.82241572 24.60145793 10.10320929 27.30499517 15.22746515
 23.06439795 25.80987343 25.74024377 23.44163911 26.89067866 23.64986201
 18.56760041 24.21484425 25.29554561 29.3176094  19.37460668 28.85158835
 29.01326229 22.0193104  12.23381564 14.29184981 30.7489456  11.58044626
 23.5939624  17.75631325 23.01927812 15.71272613 24.01038863 28.5635979
  8.88579946 21.56709728 34.23939211 30.2742995  22.4250748  21.83162229
 20.69247937 29.40935673 28.19570039 23.01856855 32.36667992 15.0124885
 32.91739353 16.40041962 22.11585873 29.40568494 20.50400665 29.16801217
 24.11813159 26.4327399  13.0041357  25.8394813  33.4272092  32.48363629
 30.85348655 32.54881719 20.66443627 16.86526948 27.62366044  8.44985256
 34.1453687  21.44503331 23.07401401 19.59072509 22.43319063 23.02938872
 21.69493136 26.27625425 30.01863505 30.4234102  29.29856711 27.86661827
 27.60830584  9.51886183 21.62521487 24.7621375  30.939533   23.19697579
 19.94239031 16.10837013 29.7418507  20.48336386 32.0819309  10.14629845]
```

In [ ]:
```
# Finding m / slope of line
for m, col_name in enumerate(x_train):
    print("Lasso model coefficient for {} is {}".format(col_name, las.coef_[
```

```
Lasso model coefficient for cylinders is -0.0
Lasso model coefficient for displacement is 0.009480059923844144
Lasso model coefficient for horsepower is -0.027364135720170345
Lasso model coefficient for weight is -0.006486451078414584
Lasso model coefficient for acceleration is 0.08095778709571876
Lasso model coefficient for model year is 0.655140788339108
Lasso model coefficient for origin is 0.10843474338531872
```

In [ ]:
```
# Finding y_intercept of model
c = las.intercept_
print('The y_intercept of this Lasso Regression is {}'.format(c))
```

```
The y_intercept of this Lasso Regression is -7.4559294197358135
```

Compare the Score

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

In [ ]:
```
# Viewing the Linear Regression Score
print(lin.score(x_train, y_train))
print(lin.score(x_test, y_test))
```

```
0.8231951593929733
0.7982782196387228
```

In [ ]:
```
# Calculating MSE and RMSE
# The lower the MSE, the better the model’s predictions match the actual val
print(mean_squared_error(y_test, y_pred)) # MSE 9.841823535225 indicates the
print(np.sqrt(mean_squared_error(y_test, y_pred))) # Square of MSE
```

```
12.230223441820602
3.4971736362126205
```

In [ ]:
```
# Viewing the Ridge Regression Score
print(rid.score(x_train, y_train))
print(rid.score(x_test, y_test))
```

```
0.823194921676305
0.7982741268009738
```

In [ ]:
```
print(mean_squared_error(y_test, y_pred_rid))
print(np.sqrt(mean_squared_error(y_test, y_pred_rid)))
```

```
12.23047158716263
3.4972091140168655
```

In [ ]:
```
# Viewing the Lasso Regression Score
print(las.score(x_train, y_train))
print(las.score(x_test, y_test))
```

```
0.8125624525047934
0.7876938769671812
```

In [ ]:
```
print(mean_squared_error(y_test, y_pred_las))
print(np.sqrt(mean_squared_error(y_test, y_pred_las)))
```

```
12.871943317710612
3.5877490600250472
```