

Lab 1: Regression Models

✓ Learning Objectives

At the end of the experiment, you will be able to:

- have an overview of the basics of Machine Learning
- understand the implementation of Train/Test Split
- develop an understanding of Least Squares, Learning Curves
- perform Linear Regression
- have an understanding of Regularization of Linear Models

✓ Introduction

Machine learning is a subfield of artificial intelligence (AI). The goal of machine learning is to understand the structure of data and model (fit) the data so that it can accurately predict the label or output for similar unseen data.

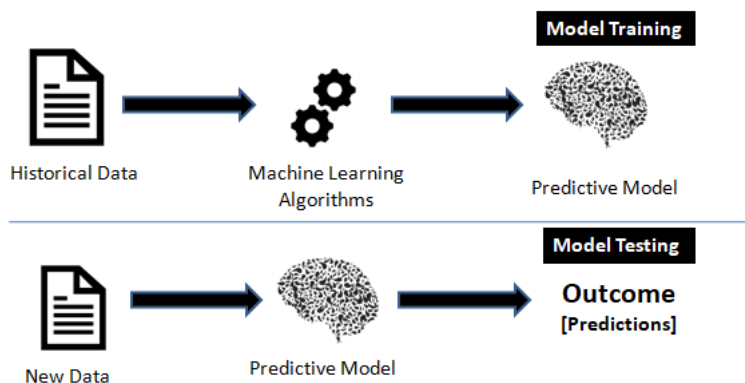
Machine Learning use cases:

Detecting tumors in brain scans, automatically classifying news articles, automatically flagging offensive comments on discussion forums, summarizing long documents automatically, creating a chatbot or a personal assistant, detecting credit card fraud, making your app react to voice commands, building an intelligent bot for a game.

Machine Learning Workflow:

1. Frame the ML problem by looking at the business need
2. Gather the data and do Data Munging/Wrangling for each subproblem
3. Explore different models, perform V&V and shortlist promising candidates
4. Fine-tune shortlisted models and combine them together to form the final solution
5. Present your solution
6. Deploy

Model training and testing



✓ Training, Validation, and Test Set

A machine learning algorithm splits the Dataset into two sets.

Splitting your dataset is essential for an unbiased evaluation of prediction performance. In most cases, it's enough to split your dataset randomly into two subsets:

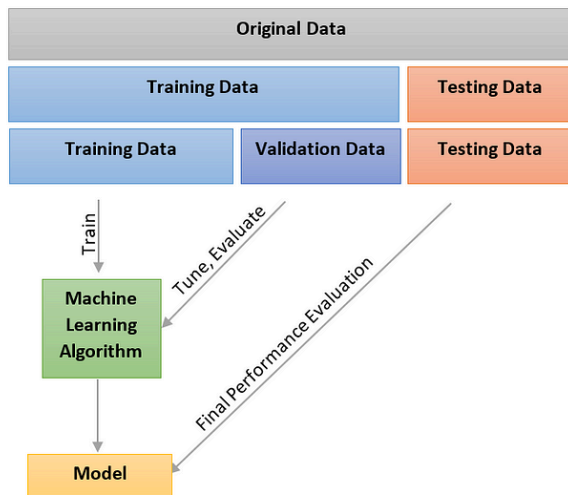
Training Dataset: The sample of data used to fit the model.

Test Dataset: The sample of data used to provide an unbiased evaluation of a final model fit on the training dataset.

We usually split the data as 80% for training stage and 20% for testing stage. 70% train and 30% test or 75% train and 25% test are also often used.

Validation Set: This is a separate section of your dataset that you will use during training to get a sense of how well your model is doing on data that are not being used in training.

In less complex cases, when you don't have to tune hyperparameters, it's okay to work with only the training and test sets.



✓ Prerequisites for using train_test_split()

We will use scikit-learn, or sklearn which has many packages for data science and machine learning.

Applying train_test_split()

You need to import:

1. train_test_split()
2. NumPy

We import NumPy because, in supervised machine learning applications, you'll typically work with two such sequences:

- A two-dimensional array with the inputs (x)
- A one-dimensional array with the outputs (y)

sklearn.model_selection.train_test_split(arrays, options)

- **arrays** is the sequence of lists, NumPy arrays, pandas DataFrames, or similar array-like objects that hold the data you want to split. All these objects together make up the dataset and must be of the same length.
- **options** are the optional keyword arguments that you can use to get desired behavior:
 - **train_size** is the number that defines the size of the training set.
 - **test_size** is the number that defines the size of the test set. You should provide either train_size or test_size.
 - If neither is given, then the default share of the dataset that will be used for testing is 0.25, or 25 percent.
 - If float (eg 0.25), it represents the proportion of the dataset to include in the test split and should be between 0.0 and 1.0.
 - If int (eg. 4), it represents the absolute number of test samples, eg. 4 samples of 12.
 - If None, the value is set to the complement of the train size.
 - If train_size is also None, it will be set to 0.25.
 - **random_state** is the object that controls randomization during splitting. It can be either an int or an instance of RandomState. The default value is None.
 - **shuffle** is the Boolean object (True by default) that determines whether to shuffle the dataset before applying the split.
 - **stratify** is an array-like object that, if not None, determines how to use a stratified split.

Setup Steps:

✓ Importing required packages

```
# Importing Standard Libraries
import numpy as np
import pandas as pd
```

```
import seaborn as sns
import matplotlib.pyplot as plt

# Importing sklearn Libraries
from sklearn import datasets
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import train_test_split, learning_curve
from sklearn.linear_model import LinearRegression
from sklearn import linear_model
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

✓ Let us use a small dataset to understand how to implement a train and test split

✓ Creating a simple dataset to work with

```
# inputs in the two-dimensional array X
X = np.arange(1, 25).reshape(12, 2)

# outputs in the one-dimensional array y
y = np.array([0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0])
```

```
print(X)
```

```
[[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]
 [13 14]
 [15 16]
 [17 18]
 [19 20]
 [21 22]
 [23 24]]
```

```
print(y)
```

```
#for 1 and 2 it corresponds to 0
```

```
[0 1 1 0 1 0 0 1 1 0 1 0]
```

✓ Splitting input and output datasets

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=4, random_state=4)
```

```
X_train
```

```
array([[17, 18],
       [ 5,  6],
       [23, 24],
       [ 1,  2],
       [ 3,  4],
       [11, 12],
       [15, 16],
       [21, 22]])
```

```
X_test
```

```
array([[ 7,  8],
       [ 9, 10],
       [13, 14],
       [19, 20]])
```

```
y_train
```

```
array([1, 1, 0, 0, 1, 0, 1, 1])
```

```
y_test
```

```
#Note: Finding the line that best represents the overall trend in the data, based on the relationship we're investigating
```

```
array([0, 1, 0, 0])
```

✓ Develop an understanding of Least Squares

Least Squares method is a statistical procedure to find the best fit for a set of data points by minimizing the sum of the offsets or residuals of points from the plotted curve.

Calculate Line Of Best Fit

A more accurate way of finding the line of best fit is the least square method.

Use the following steps to find the equation of line of best fit for a set of ordered pairs $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$.

Step 1: Calculate the slope 'm' by using the following formula:

$$m = \frac{\sum (x - \bar{x}) * (y - \bar{y})}{\sum (x - \bar{x})^2}$$

Step 2: Compute the y -intercept of the line by using the formula:

$$c = y - mx$$

Step 3: Substitute the values in the final equation

$$y = mx + c$$

- y: dependent variable
- m: the slope of the line
- x: independent variable
- c: y-intercept

As an example, we will try to find the least squares regression line for the below data set:

<i>HoursSpent</i>	<i>Grade</i>
6	82
10	88
2	56
4	64
6	77
7	92
0	23
1	41
8	80
5	59
3	47

x = HoursSpent

y = Grade

\bar{x} = 4.72

\bar{y} = 64.45

<i>HoursSpent</i>	<i>Grade</i>	$x - \bar{x}$	$y - \bar{y}$	$(x - \bar{x}) * (y - \bar{y})$
6	82	1.27	17.55	22.33
10	88	5.27	23.55	124.15
2	56	-2.73	-8.45	23.06
4	64	-0.73	-0.45	0.33
6	77	1.27	12.55	15.97
7	92	2.27	27.55	62.60
0	23	-4.73	-41.45	195.97
1	41	-3.73	-23.42	87.42
8	80	3.27	15.55	50.88
5	59	0.27	-5.45	-1.49
3	47	-1.73	-17.45	30.15

$$\sum (x - \bar{x}) * (y - \bar{y}) = 611.36$$

$$\sum (x - \bar{x})^2 = 94.18$$

$$m = \frac{611.36}{94.18}$$

$$m = 6.49$$

Calculate the intercept:

$$c = y - mx$$

$$c = 64.45 - (6.49 * 4.72)$$

$$c = 64.45 - 30.63$$

$$c = 30.18$$

Now that we have all the values to fit into the equation. If we want to know the predicted grade of someone who spends 2.35 hours on their essay, all we need to do is substitute that in for X.

$$y = (6.49 * X) + 30.18$$

$$y = (6.49 * 2.35) + 30.18$$

$$y = 45.43$$

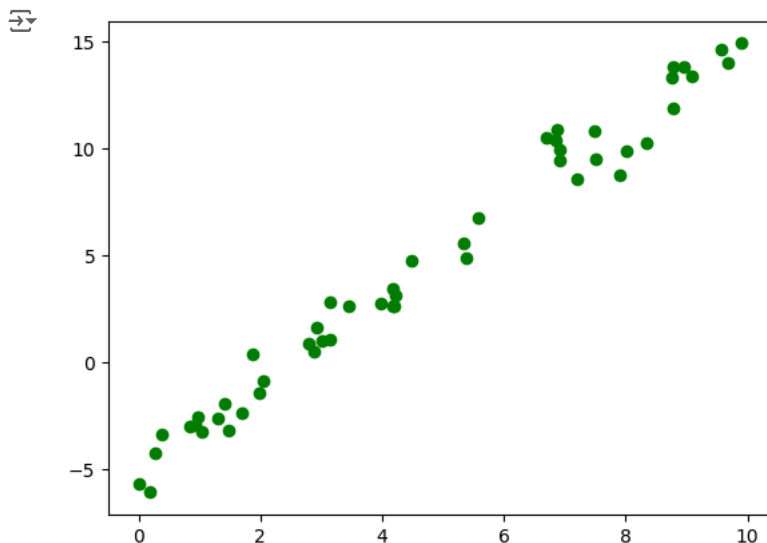
✓ Example: Ordinary least squares Linear Regression

Ordinary least squares (OLS) is a type of linear least squares method for estimating the unknown parameters in a linear regression model. OLS chooses the parameters of a linear function of a set of explanatory variables by the principle of least squares: minimizing the sum of the squares of the differences between the observed dependent variable (values of the variable being observed) in the given dataset and those predicted by the linear function of the independent variable.

Geometrically, this is seen as the sum of the squared distances, parallel to the axis of the dependent variable, between each data point in the set and the corresponding point on the regression surface—the smaller the differences, the better the model fits the data.

Generating Sample data

```
rng = np.random.RandomState(1)          # instantiate random number generator
x = 10 * rng.rand(50)                   # generate 50 random numbers from uniform distribution
y = 2 * x - 5 + rng.randn(50)           # use 50 random numbers from normal distribution as noise
plt.scatter(x, y, c='g');
```



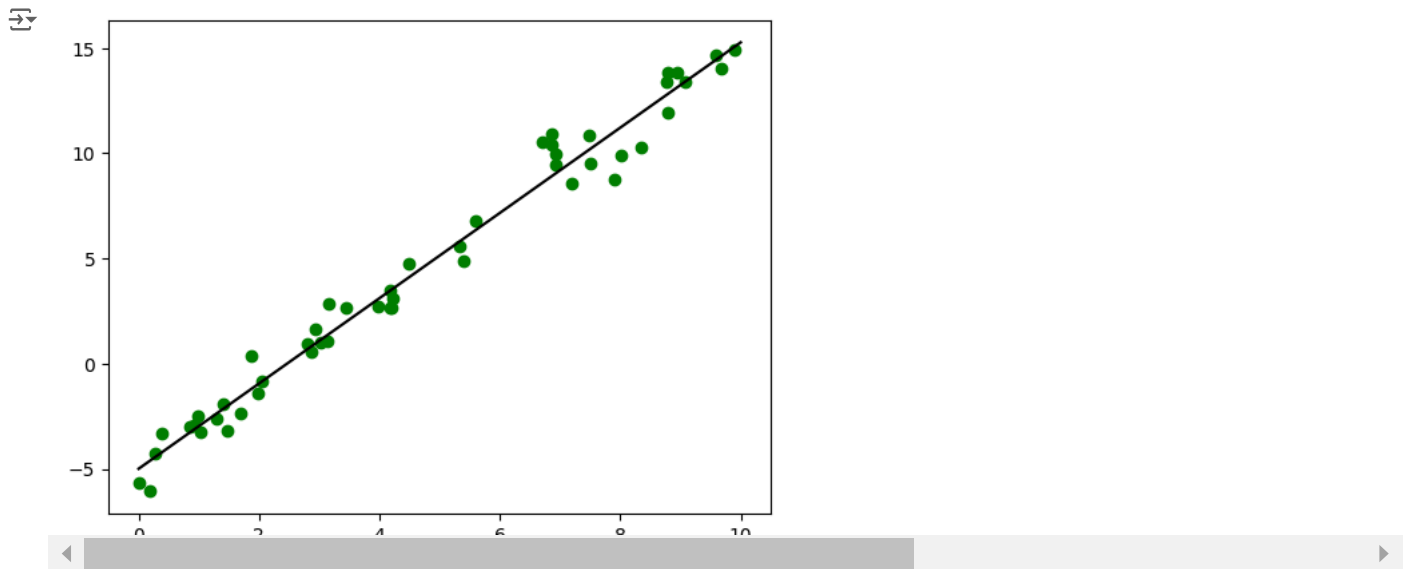
Using Scikit-Learn's Linear Regression estimator to fit the above data and construct the best-fit line

```
model = LinearRegression(fit_intercept=True)          # instantiate LinearRegression
model.fit(x[:, np.newaxis], y)                        # fit the model on data using 'x' as column vector

xfit = np.linspace(0, 10, 1000)                      # create 1000 points between 0 and 10
yfit = model.predict(xfit[:, np.newaxis])              # predict the values for dependent variable

plt.scatter(x, y, c='g')
plt.plot(xfit, yfit, 'k');
```

#When we say a model is being "fit," we usually mean that we are adjusting the model's parameters to best match the data.
 # "targets" (also known as "labels" or "outputs") refer to the values that you want to predict or classify based on the input data (features).

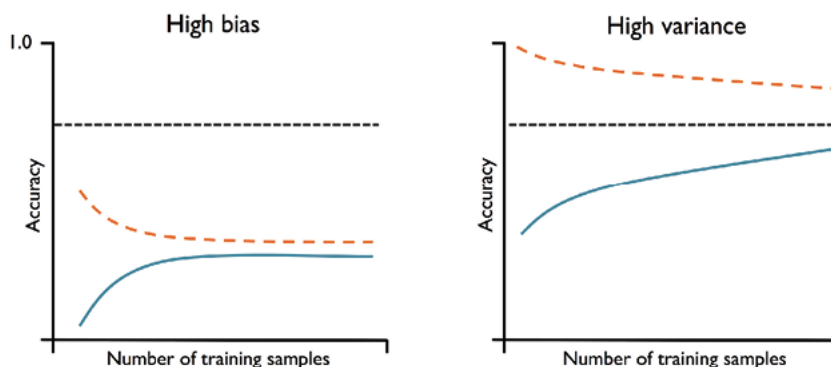


Learning Curves

Learning curve in machine learning is used to assess how models will perform with varying numbers of training samples. This is achieved by monitoring the training and validation scores (model accuracy) with an increasing number of training samples.

Below image showcases 'Learning curves representing high bias and high variance'.

- orange dashed line - represent training
- blue line - represent validation
- black dashed line - desired model accuracy



Example: Simple linear regression combined with the polynomial preprocessor

Polynomial Features

Polynomial features are those features created by raising existing features to an exponent.

For example, if a dataset had one input feature X , then a polynomial feature would be the addition of a new feature (column) where values were calculated by squaring the values in X , e.g. X^2 . This process can be repeated for each input variable in the dataset, creating a transformed version of each. And, if an input sample is two dimensional and of the form $[a, b]$, the degree-2 polynomial features are $[1, a, b, a^2, ab, b^2]$.

As such, polynomial features are a type of feature engineering, e.g. the creation of new input features based on the existing features.

The "degree" of the polynomial is used to control the number of features added, e.g. a degree of 3 will add two new variables for each input variable. Typically a small degree is used such as 2 or 3.

```
def PolynomialRegression(degree=2, **kwargs):
    return make_pipeline(PolynomialFeatures(degree), LinearRegression(**kwargs)) # using a pipeline to string these operations together
```

Creating data to fit into the model

```
def make_data(N, err=1.0, rseed=1):
    # randomly sample the data
```

```

rng = np.random.RandomState(rseed)
X = rng.rand(N, 1) ** 2
y = 10 - 1. / (X.ravel() + 0.1)
if err > 0:
    y += err * rng.randn(N)
return X, y

```

```
X, y = make_data(40)
```

✓ Learning curve in Scikit-Learn

learning curve for generated dataset with a second-order polynomial model and a ninth-order polynomial

```

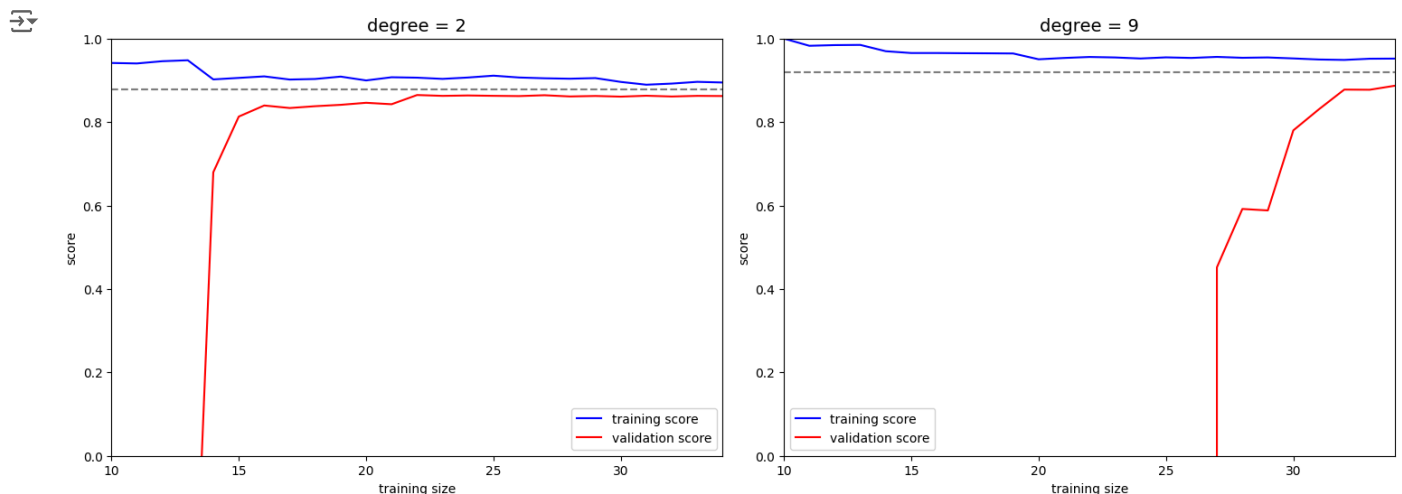
fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

for i, degree in enumerate([2, 9]):
    N, train_lc, val_lc = learning_curve(PolynomialRegression(degree),
                                       X, y, cv=7,
                                       train_sizes=np.linspace(0.3, 1, 25))

    ax[i].plot(N, np.mean(train_lc, 1), color='blue', label='training score')
    ax[i].plot(N, np.mean(val_lc, 1), color='red', label='validation score')
    ax[i].hlines(np.mean([train_lc[-1], val_lc[-1]]), N[0], N[-1],
                color='gray', linestyle='dashed')

    ax[i].set_ylim(0, 1)
    ax[i].set_xlim(N[0], N[-1])
    ax[i].set_xlabel('training size')
    ax[i].set_ylabel('score')
    ax[i].set_title('degree = {}'.format(degree), size=14)
    ax[i].legend(loc='best')

```



✓ Example: Machine Learning Workflow using Linear-Regression with Auto-MPG Dataset

✓ Dataset

In this example, we will be using the "Auto-MPG" dataset. From Github. (UC Irvine ML Repository)

The data concerns city-cycle fuel consumption in miles per gallon, to be predicted in terms of 3 multivalued discrete and 5 continuous attributes.

Attribute Information:

1. mpg: continuous
2. cylinders: multi-valued discrete
3. displacement: continuous
4. horsepower: continuous

5. weight: continuous
6. acceleration: continuous
7. model year: multi-valued discrete
8. origin: multi-valued discrete
9. car name: string (unique for each instance)

Number of instances: 398

Problem statement: Predict the fuel consumption in miles per gallon.

✓ Loading Data

```
# Read data
auto = pd.read_csv("/content/auto-mpg.csv") #file uploaded from device - of auto mpg
```

Displaying Dataframe

```
auto.head()
```



	mpg	cylinders	displacement	horsepower	weight	acceleration	model-year
0	18.0	8	307.0	130.0	3504	12.0	70
1	15.0	8	350.0	165.0	3693	11.5	70
2	18.0	8	318.0	150.0	3436	11.0	70
3	16.0	8	304.0	150.0	3433	12.0	70
4	17.0	8	280.0	140.0	3440	10.5	70

✓ Exploring the dataset

```
# print names of the features
print(auto.columns)
```



```
Index(['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
      'acceleration', 'model-year'],
      dtype='object')
```

```
# generating descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding Na
auto.describe()
```



	mpg	cylinders	displacement	horsepower	weight	acceleration	model-year
count	398.000000	398.000000	398.000000	396.000000	398.000000	398.000000	398.000000
mean	23.514573	5.454774	193.425879	104.189394	2970.424623	15.568090	76.010050
std	7.815984	1.701004	104.269838	38.402030	846.841774	2.757689	3.697627
min	9.000000	3.000000	68.000000	46.000000	1613.000000	8.000000	70.000000
25%	17.500000	4.000000	104.250000	75.000000	2223.750000	13.825000	73.000000
50%	23.000000	4.000000	148.500000	92.000000	2803.500000	15.500000	76.000000
75%	29.000000	8.000000	262.000000	125.000000	3608.000000	17.175000	79.000000
max	46.000000	8.000000	455.000000	330.000000	5140.000000	24.000000	82.000000


```
# summary of the DataFrame
auto.info()
```



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 7 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       396 non-null   float64
4   weight           398 non-null   int64
5   acceleration     398 non-null   float64
6   model-year       398 non-null   int64
dtypes: float64(4), int64(3)
memory usage: 21.9 KB
```


✓ Checking for Missing values

```
auto.isna().sum()
```




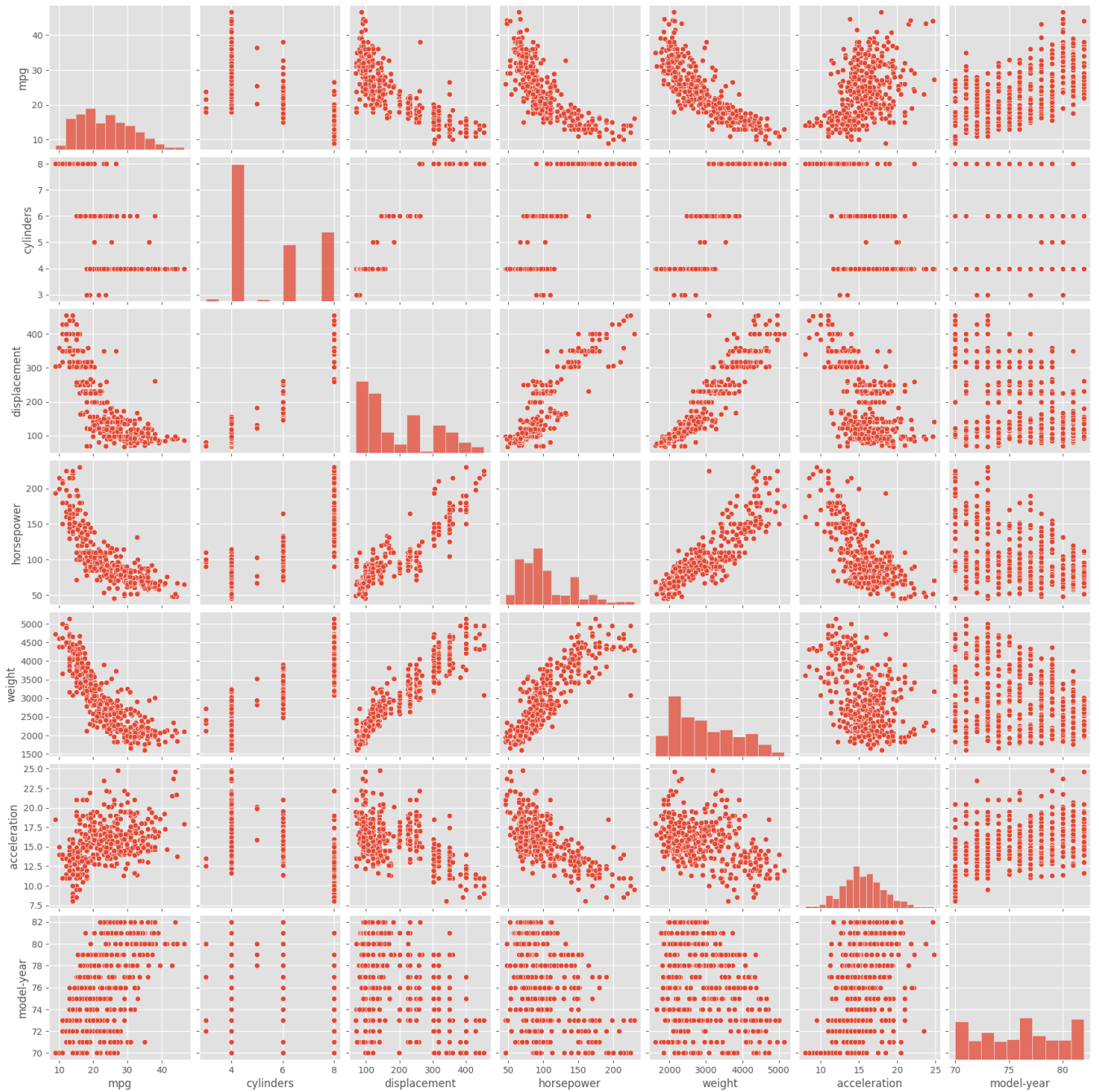
	0
mpg	0
cylinders	0
displacement	0
horsepower	2
weight	0
acceleration	0
model-year	0

✓ Visualization of Auto-MPG Dataset

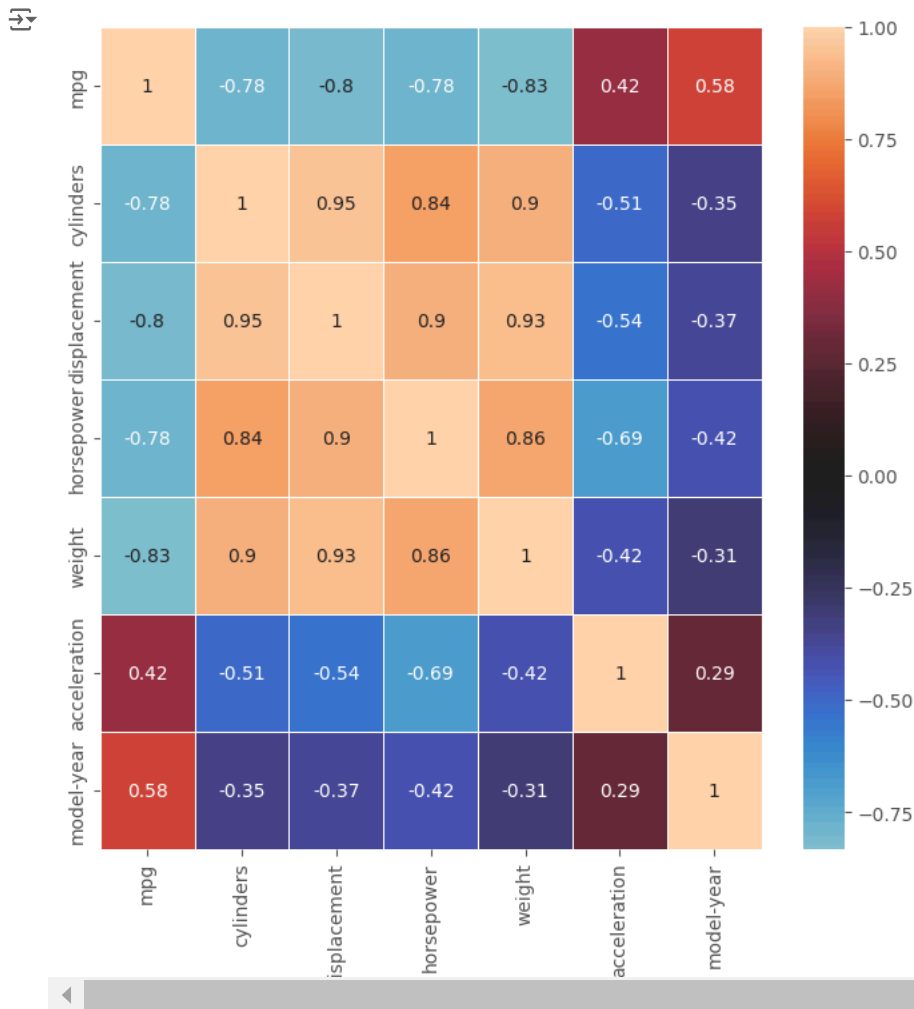
Creating a pairplot and a heatmap to check which features seems to be more correlated

```
# Pairplot
plt.style.use('ggplot')
sns.pairplot(auto)
```

 <seaborn.axisgrid.PairGrid at 0x7acab70220b0>



```
# Heatmap
plt.figure(figsize=(8, 8))
sns.heatmap(auto.corr(), annot=True, linewidth=0.5, center=0)
plt.show()
```



From the above plots, we can see that the features cylinders, displacement, and weight are highly correlated. We can use anyone of them for modeling.

Modeling and Prediction (Linear Regression)

```
auto.head()
```

	mpg	cylinders	displacement	horsepower	weight	acceleration	model-year
0	18.0	8	307.0	130.0	3504	12.0	70
1	15.0	8	350.0	165.0	3693	11.5	70
2	18.0	8	318.0	150.0	3436	11.0	70
3	16.0	8	304.0	150.0	3433	12.0	70
4	17.0	8	302.0	140.0	3440	10.5	70

```
# Datatypes of all features
auto.dtypes
```

mpg	float64
cylinders	int64
displacement	float64
horsepower	float64
weight	int64
acceleration	float64
model-year	int64

```
# Unique values in horsepower column
```

```
auto['horsepower'].unique()
```

```
array([130., 165., 150., 140., 198., 220., 215., 225., 190., 170., 160.,
       95., 97., 85., 88., 46., 87., 90., 113., 200., 210., 193.,
       nan, 100., 105., 175., 153., 180., 110., 72., 86., 70., 76.,
       65., 69., 60., 80., 54., 208., 155., 112., 92., 145., 137.,
       158., 167., 94., 107., 230., 49., 75., 91., 122., 67., 83.,
       78., 52., 61., 93., 148., 129., 96., 71., 98., 115., 53.,
       81., 79., 120., 152., 102., 108., 68., 58., 149., 89., 63.,
       48., 66., 139., 103., 125., 133., 138., 135., 142., 77., 62.,
       132., 84., 64., 74., 116., 82.])
```

```
# Removing '?' from horsepower column
```

```
auto = auto[auto['horsepower'] != '?']
```

```
auto['horsepower'].unique()
```

```
array([130., 165., 150., 140., 198., 220., 215., 225., 190., 170., 160.,
       95., 97., 85., 88., 46., 87., 90., 113., 200., 210., 193.,
       nan, 100., 105., 175., 153., 180., 110., 72., 86., 70., 76.,
       65., 69., 60., 80., 54., 208., 155., 112., 92., 145., 137.,
       158., 167., 94., 107., 230., 49., 75., 91., 122., 67., 83.,
       78., 52., 61., 93., 148., 129., 96., 71., 98., 115., 53.,
       81., 79., 120., 152., 102., 108., 68., 58., 149., 89., 63.,
       48., 66., 139., 103., 125., 133., 138., 135., 142., 77., 62.,
       132., 84., 64., 74., 116., 82.])
```

```
# Converting horsepower column datatype from string to float
```

```
auto['horsepower'] = auto['horsepower'].astype(float)
```

```
auto.dtypes
```

```

      0
mpg      float64
cylinders int64
displacement float64
horsepower float64
weight      int64
acceleration float64
model-year int64

```

```
auto.isna().sum()
```

```

      0
mpg      0
cylinders 0
displacement 0
horsepower 2
weight      0
acceleration 0
model-year 0

```

```
auto = auto.dropna()
```

```
# Prediction features
```


```
#X = auto[['displacement', 'horsepower', 'acceleration', 'model year', 'origin']]
```

```
X = auto[['displacement', 'horsepower', 'acceleration', 'model-year']]
```

```
# Target feature
```

```
y = auto['mpg']
```


```
X.head()
```



	displacement	horsepower	acceleration	model-year
0	307.0	130.0	12.0	70
1	350.0	165.0	11.5	70
2	318.0	150.0	11.0	70
3	304.0	150.0	12.0	70
4	269.0	140.0	10.5	70

```
# Splitting the Dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.33, random_state= 101)
```


```
X_train.head()
```



	displacement	horsepower	acceleration	model-year
125	198.0	95.0	16.5	74
224	302.0	130.0	14.9	77
61	122.0	86.0	16.5	72
379	98.0	70.0	17.3	82
445	232.0	64.0	10.0	74

```
# Instantiating LinearRegression() Model
lr = LinearRegression()
```

```
# Training/Fitting the Model
lr.fit(X_train, y_train)
```




LinearRegression	?	?
LinearRegression()		

Testing

```
# Making Predictions
pred = lr.predict(X_test)


# Evaluating Model's Performance
print('Mean Absolute Error:', mean_absolute_error(y_test, pred))
print('Mean Squared Error:', mean_squared_error(y_test, pred))
print('Mean Root Squared Error:', np.sqrt(mean_squared_error(y_test, pred)))
print('Coefficient of Determination:', r2_score(y_test, pred))
```



Mean Absolute Error: 2.7900063939976363
Mean Squared Error: 14.442906334553035
Mean Root Squared Error: 3.800382393201115
Coefficient of Determination: 0.7546220217389916

Predicting the value

```
pred = lr.predict(X_test)
print('Predicted fuel consumption(mpg):', pred[2])
print('Actual fuel consumption(mpg):', y_test.values[2])
```



Predicted fuel consumption(mpg): 32.745319120567956
Actual fuel consumption(mpg): 31.0

✓ Let us now apply the above learnings to perform a linear regression based price prediction, using a 'Real estate' dataset (Practice Ungraded)

Linear regression model implementation

- Fit the model
- Do the prediction
- Plot the straight line for the predicted data using linear regression model

Dataset

In this example, we will be using the “Real estate price prediction” dataset

- Transaction date (purchase)
- House age
- Distance to the nearest MRT station (metric not defined)
- Amount of convenience stores
- Location (latitude and longitude)
- House price of unit area

Problem statement: Predict the house price of unit area based on various features provided such as house age, location, etc.

Importing all the required libraries

```
# Your Standard Libraries
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Importing sklearn Libraries
from sklearn import datasets
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import train_test_split, learning_curve
from sklearn.linear_model import LinearRegression
from sklearn import linear_model
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

Importing the dataset

```
# Note that the dataset is already downloaded in the set-up stage at the start of the notebook
# Convert it into a pandas dataframe:
```

```
df = pd.read_csv('/content/Real_estate.csv')
```

```
# Taking only the selected two attributes from the dataset
df_binary = df[['X2 house age', 'Y house price of unit area']]
```

```
df.head()
```

	No	X1 transaction date	X2 house age	X3 distance to the nearest MRT station	X4 number of convenience stores	X5 latitude	X6 longitude	Y house price of unit area
0	1	2012.917	32.0	84.87882	10	24.98298	121.54024	37.9
1	2	2012.917	19.5	306.59470	9	24.98034	121.53951	42.2
2	3	2013.583	13.3	561.98450	5	24.98746	121.54391	47.3
3	4	2013.500	13.3	561.98450	5	24.98746	121.54391	54.8

Next steps: [Generate code with df](#) [View recommended plots](#) [New interactive sheet](#)

Dropping non-useful columns

```
#dropping columns
# YOUR CODE HERE
columns_to_drop = ['X1 transaction date', 'X3 distance to the nearest MRT station', 'X4 number of convenience stores', 'X5 latitude', '']
df = df.drop(columns=columns_to_drop)
```

Finding if there are any null values

```
# YOUR CODE HERE
null_values = df.isnull()
print(null_values)
```

```

No  X2 house age  Y house price of unit area
0   False       False                False
1   False       False                False
2   False       False                False
3   False       False                False
4   False       False                False
..   ...         ...                  ...
409 False       False                False
410 False       False                False
411 False       False                False
412 False       False                False
413 False       False                False

```

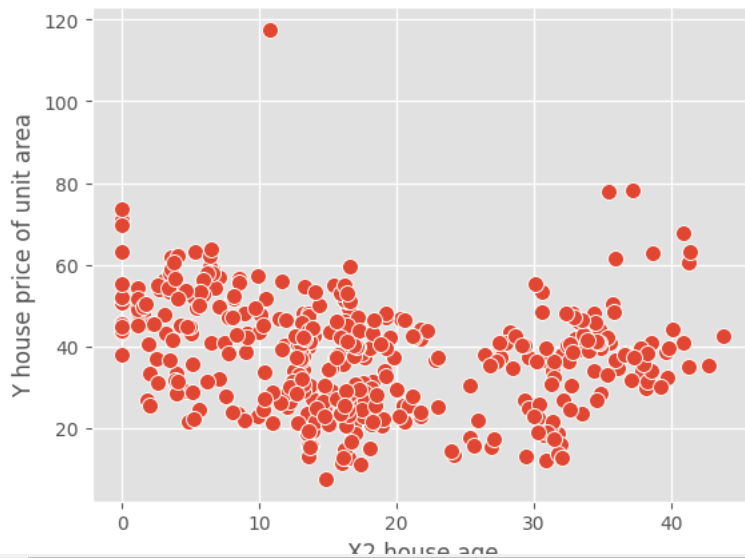
[414 rows x 3 columns]

✖ Exploring the data scatter

YOUR CODE HERE

```
sns.scatterplot(data=df, x='X2 house age', y='Y house price of unit area', s=75)
```

```
<Axes: xlabel='X2 house age', ylabel='Y house price of unit area'>
```



✖ Training our model

Separating the data into independent and dependent variables

```
X = df[['X2 house age']]
```

```
y = df['Y house price of unit area']
```

```
X.head()
```

YOUR CODE HERE

```

X2 house age
0           32.0
1           19.5
2           13.3
3           13.3
4            5.0

```

Splitting the data into training and testing data

YOUR CODE HERE

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 0.33, random_state= 101)
```

```
X_train.head()
```

	X2 house age	
90	0.0	
312	35.4	
383	29.1	
78	38.2	
101	0.0	

Next steps:

[Generate code with X_train](#)[View recommended plots](#)[New interactive sheet](#)

Training the Linear Regression model on the Training set

YOUR CODE HERE

lr = LinearRegression()

Training/Fitting the Model

YOUR CODE HERE

lr.fit(X_train, y_train)

LinearRegression

LinearRegression()

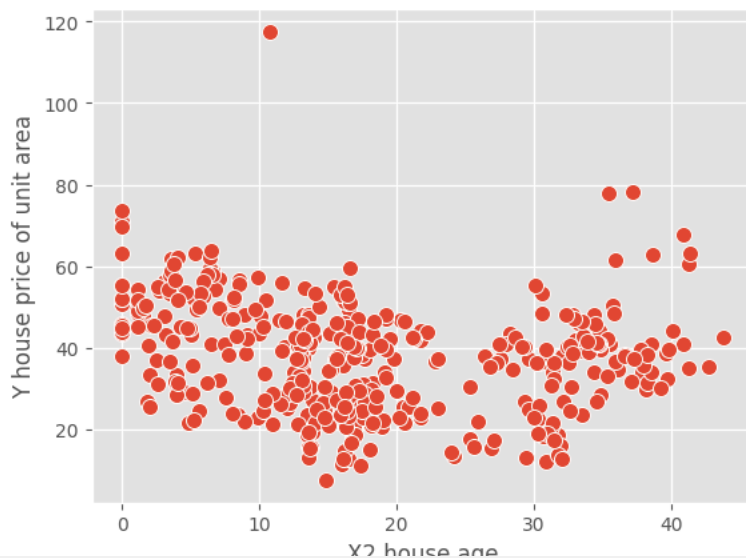
Exploring the results

Data scatter of predicted values

pred = lr.predict(X_test)

sns.scatterplot(data=df, x='X2 house age', y='Y house price of unit area', s=75)

<Axes: xlabel='X2 house age', ylabel='Y house price of unit area'>



Regularized Linear Models

A good way to reduce overfitting is to regularize the model (i.e., to constrain it): the fewer degrees of freedom it has, the harder it will be for it to overfit the data.

For a linear model, regularization is typically achieved by constraining the weights of the model.

Three different ways to constrain the weights:

1. Ridge Regression
2. Lasso Regression
3. Elastic Net

Ridge regression

Ridge regression or **Tikhonov regularization** is the regularization technique that performs L2 regularization. It modifies the loss function by adding the penalty (shrinkage quantity) equivalent to the square of the magnitude of coefficients.

- **sklearn.linear_model.Ridge** is the module used to solve a regression model where the loss function is the linear least squares function and regularization is L2.

✓ Ridge Regression with Scikit-Learn

```
n_samples, n_features = 15, 10
rng = np.random.RandomState(0)
y = rng.randn(n_samples)
X = rng.randn(n_samples, n_features)

rdg = linear_model.Ridge(alpha = 0.5)          # instantiate Ridge regressor
rdg.fit(X, y)
rdg.score(X,y)

↗ 0.7629498741931634
```

LASSO (Least Absolute Shrinkage and Selection Operator)

LASSO is the regularisation technique that performs L1 regularisation. It modifies the loss function by adding the penalty (shrinkage quantity) equivalent to the summation of the absolute value of coefficients.

- **sklearn.linear_model.Lasso** is a linear model, with an added regularisation term, used to estimate sparse coefficients.

✓ Lasso Regression with Scikit-Learn

uses coordinate descent as the algorithm to fit the coefficients

```
Lreg = linear_model.Lasso(alpha = 0.5)
Lreg.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
```

```
↗
└ Lasso ⓘ ?
  Lasso(alpha=0.5)
```

Once fitted, the model can predict new values as follows:

```
Lreg.predict([[0,1]])
```

```
↗ array([0.75])
```

```
#weight vectors
Lreg.coef_
```

```
↗ array([0.25, 0.  ])
```

```
#Calculating intercept
Lreg.intercept_
```

```
↗ 0.75
```

```
#Calculating number of iterations
Lreg.n_iter_
```

```
↗ 2
```

Elastic-Net Regression

Elastic-Net is a regularised regression method that linearly combines both penalties i.e. L1 and L2 of the Lasso and Ridge regression methods.

✓ Elastic Net Regression with Scikit-Learn

```
# uses coordinate descent as the algorithm to fit the coefficients
```

```
ENreg = linear_model.ElasticNet(alpha = 0.5, random_state = 0)
ENreg.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
```

```
ElasticNet
ElasticNet(alpha=0.5, random_state=0)
```

Once fitted, the model can predict new values as follows:

```
ENreg.predict([[0,1]])
```

```
array([0.73681643])
```

```
#weight vectors
```

```
ENreg.coef_
```

```
array([0.26318357, 0.26313923])
```

```
#Calculating intercept
```

```
ENreg.intercept_
```

```
0.47367720941913904
```

```
#Calculating number of iterations
```

```
ENreg.n_iter_
```

```
15
```

Understanding Significance of Alpha

Alpha is a parameter for regularization term, aka penalty term, that combats overfitting by constraining the size of the weights. Increasing alpha may fix high variance (a sign of overfitting) by encouraging smaller weights, resulting in a decision boundary plot that appears with lesser curvatures. Similarly, decreasing alpha may fix high bias (a sign of underfitting) by encouraging larger weights, potentially resulting in a more complicated decision boundary.

Lasso regression is a common modeling technique to do regularization. So, we will be applying varying levels of alpha to show the effect on the coefficients.

```
Lreg = linear_model.Lasso(alpha = 0.25)
Lreg.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
Lreg.coef_ #weight vectors
```

```
array([0.625, 0.   ])
```

```
Lreg = linear_model.Lasso(alpha = 0.5)
Lreg.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
Lreg.coef_ #weight vectors
```

```
array([0.25, 0.   ])
```

```
Lreg = linear_model.Lasso(alpha = 0.75)
Lreg.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
Lreg.coef_ #weight vectors
```

```
array([0., 0.])
```

From the above 3 code cells, we could see that the coefficient value has been decreasing as the value of alpha is increased.

<i>AlphaValues</i>	<i>CoefficientValues</i>
0.25	0.625
.5	.25
.75	0.

Theory Questions

1. What is the difference between the training set and the test set?

The training set is a subset of your data on which your model will learn how to predict the dependent variable with the independent variables.

The test set is the complimentary subset from the training set, on which you will evaluate your model to see if it manages to predict correctly the dependent variable with the independent variables.

2. Why do we split on the dependent variable?

We want to have well-distributed values of the dependent variable in the training and test set. For example, if we only had the same value of the dependent variable in the training set, our model wouldn't be able to learn any correlation between the independent and dependent variables.

3. What is the purpose of a validation set?

The Validation Set is a separate section of your dataset that you will use during training to get a sense of how well your model is doing on data that are not being used in training.

4. How do you choose the value of the regularization hyperparameter?

A common solution to this problem is called holdout validation: you simply hold out part of the training set to evaluate several candidate models and select the best one.

The new held out set is called the validation set (or sometimes the development set, or dev set).

5. If your model performs great on the training data but generalizes poorly to new instances, what is happening? Can you name three possible solutions?

If the model performs poorly to new instances, then it has overfitted on the training data. To solve this, we can do any of the following three: get more data, implement a simpler model, or eliminate outliers or noise from the existing data set.