Intel Student Developer Club, RCOEM

Machine Learning Bootcamp, 2022

Day - 2 : Supervised Learning 👋

Regularization :

Generalization :

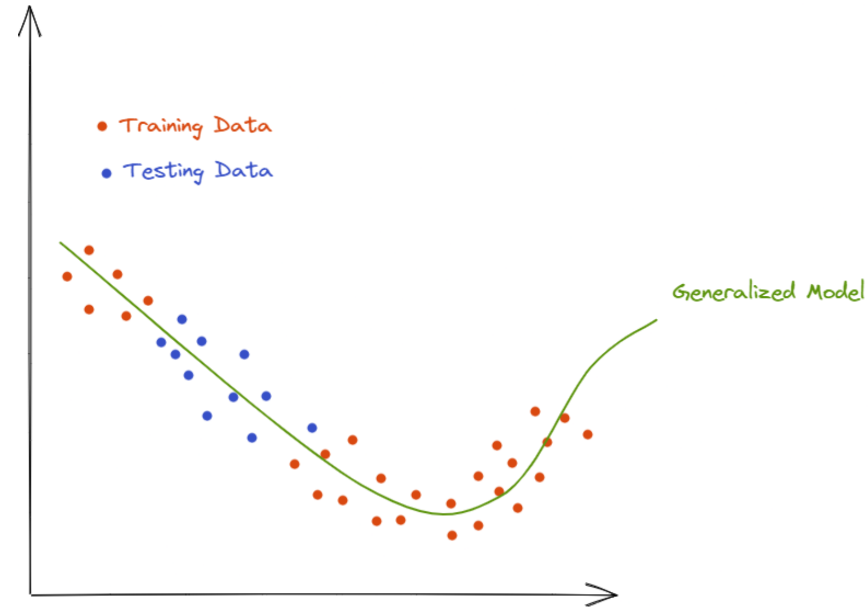
→ In supervised learning, we want to build a model on the training data and then be able to make accurate predictions on new, unseen data that has the same characteristics as the training set that we used. If a model is able to make accurate predictions on unseen data, we say it is able to *generalize* from the training set to the test set. We want to build a model that is able to generalize as accurately as possible.

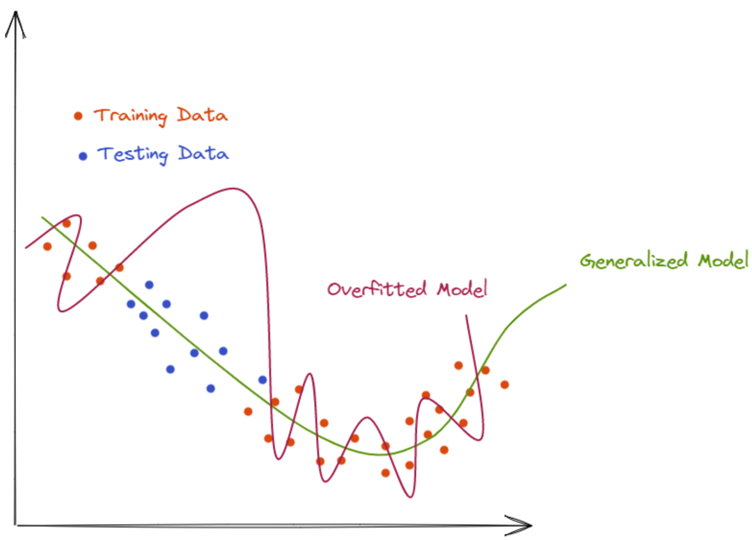
→ Usually we build a model in such a way that it can make accurate predictions on the training set. If the training and test sets have enough in common, we expect the model to also be accurate on the test set. However, there are some cases where this can go wrong. For example, if we allow ourselves to build very complex models, we can always be as accurate as we like on the training set.

Overfitting and Underfitting :

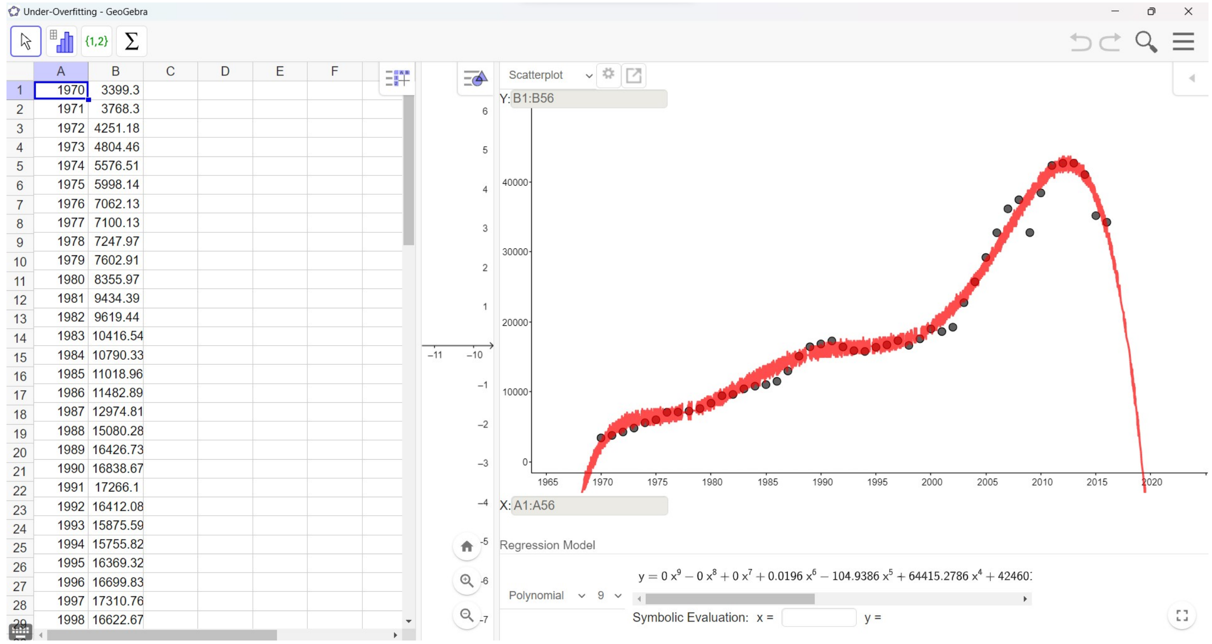
→ Building a model that is too complex for the amount of info we have is called ***Overfitting*** the model. *Overfitting* occurs when you fit a model too closely to the particularities of the training set and obtain the a model that works too well on the training set but is not able to generalize on new data.

→ On the other hand, if our model is very simple then we might not be able to catch all the aspects and variability in the data, and the model will perform badly even on the training data, this is called ***Underfitting***

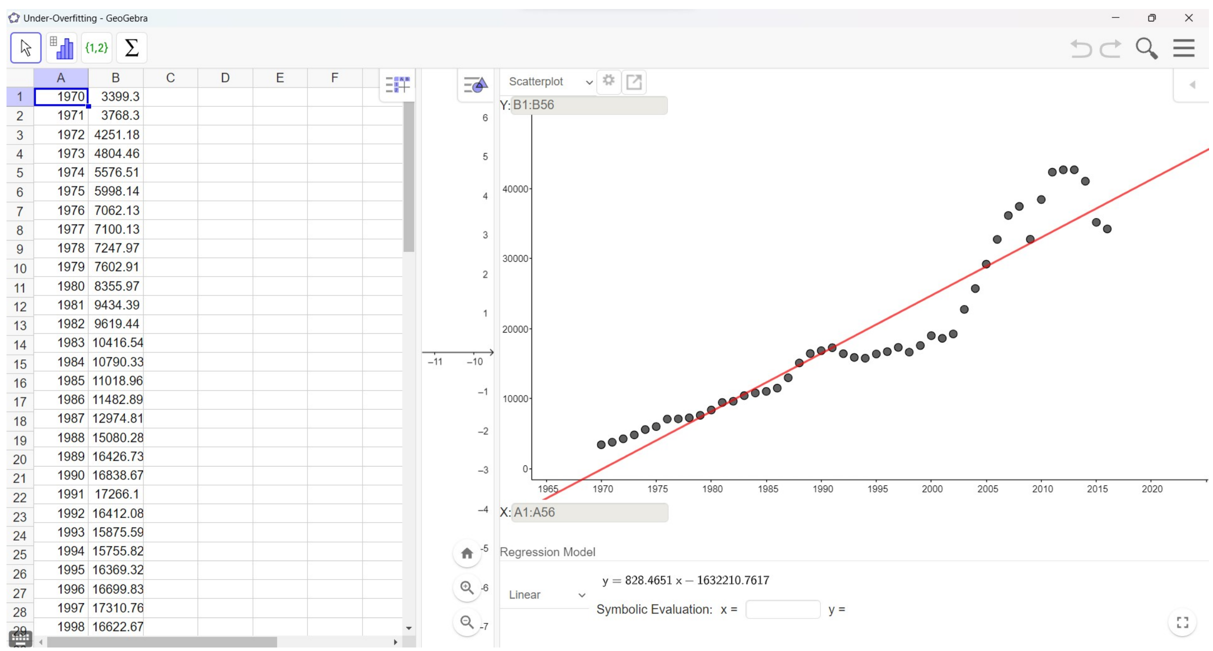




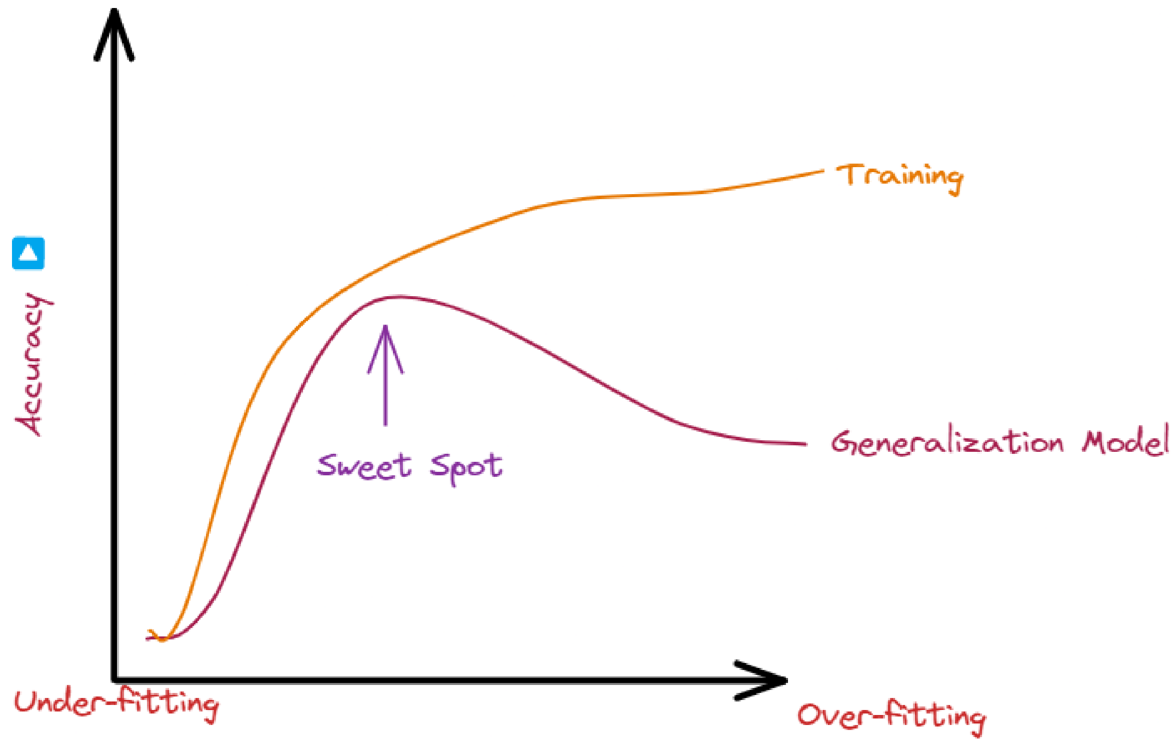
Lets now see a live example of Over and Underfitting



Over-Fitting



Under-Fitting

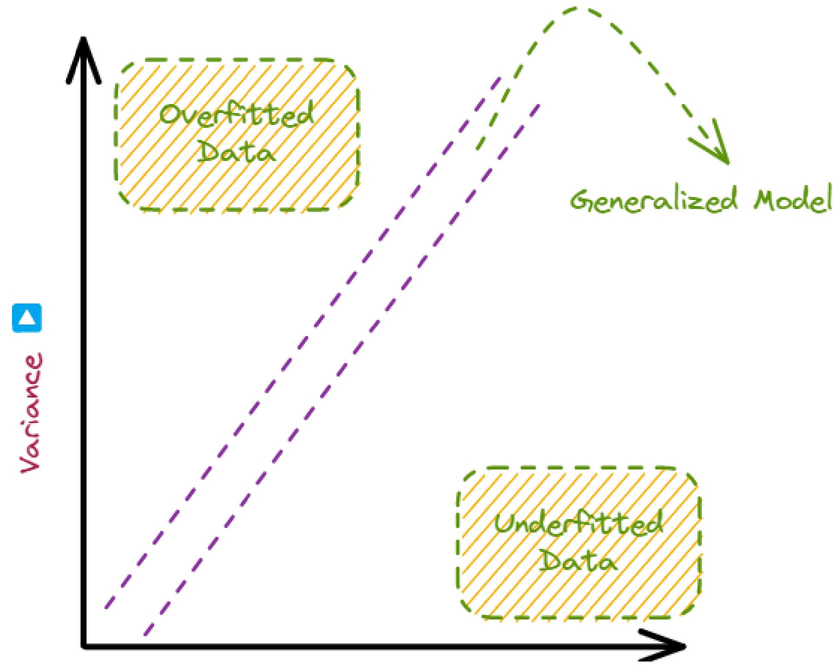


Bias and Variance

The inability of a machine learning model (for ex. Linear Regression) to capture the true relationship between data points is called ***Bias***.

The difference in fits between the datasets of a model is called ***Variance***.

Underfitting : High Bias, Low Variance ; This does not explore all the data points in the dataset
Overfitting : Low Bias, High Variance ; This fits great on training data but might not fit on testing data

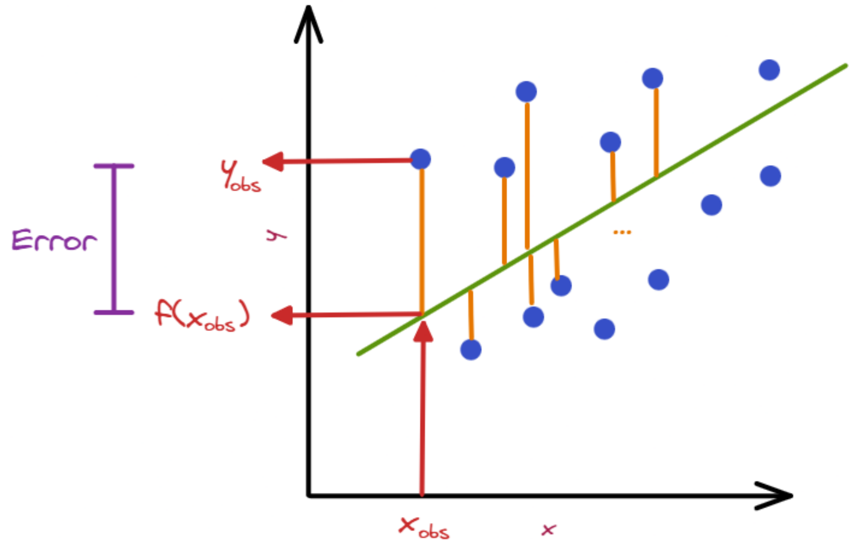


Introduction to Linear Regression :

The general formula for a linear regression model is as follows :-

$f(x) \equiv y = a + bx$

ie, for a value of x on the $x - axis$ we will get a corresponding value of y, which will be our outcome or prediction.



Residual Error :

$$\equiv f(x_{obs}) - y_{obs}$$
$$\equiv a + bx_{obs} - y_{obs}$$

$J(a,b) = 1/2m \sum[((a+bx_{obs}) - y_{obs})]^2$ → Here, we want to minimize the mean squared error.

KNN vs. Linear Regression :

A. Linear Regression :

- Fitting involves minimizing cost function, making it slower.
- Model has few parameters [memory efficient]
- Prediction involves calculation, ie, just putting x data in function to get value.

B. KNN :

- Fitting involves storing training data, ie, storing min value (fast).
- Model has many parameters [memory efficient].
- Prediction involves finding closest neighbors, $\therefore$ slow.

Let us now create our very own Linear Regression Model

In [ ]:

```
# Step 1 : Import the necessary libraries and datasets
```

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

from sklearn.datasets import load_boston
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
```

First We Will See a Simple Dataset

In [ ]:

```
canada_df = pd.read_csv('../Assets/canada_per_capita_income.csv')
canada_df.head()
```

Out[ ]:

	year	per capita income (US$)
0	1970	3399.299037
1	1971	3768.297935
2	1972	4251.175484
3	1973	4804.463248
4	1974	5576.514583

In [ ]:

```
canada_df.tail()
```

Out[ ]:

	year	per capita income (US$)
42	2012	42665.25597
43	2013	42676.46837
44	2014	41039.89360
45	2015	35175.18898
46	2016	34229.19363

In [ ]:

```
# Basic statistical values of our dataset can be known by using the describe method
```

```
canada_df.describe()
```

Out[ ]:

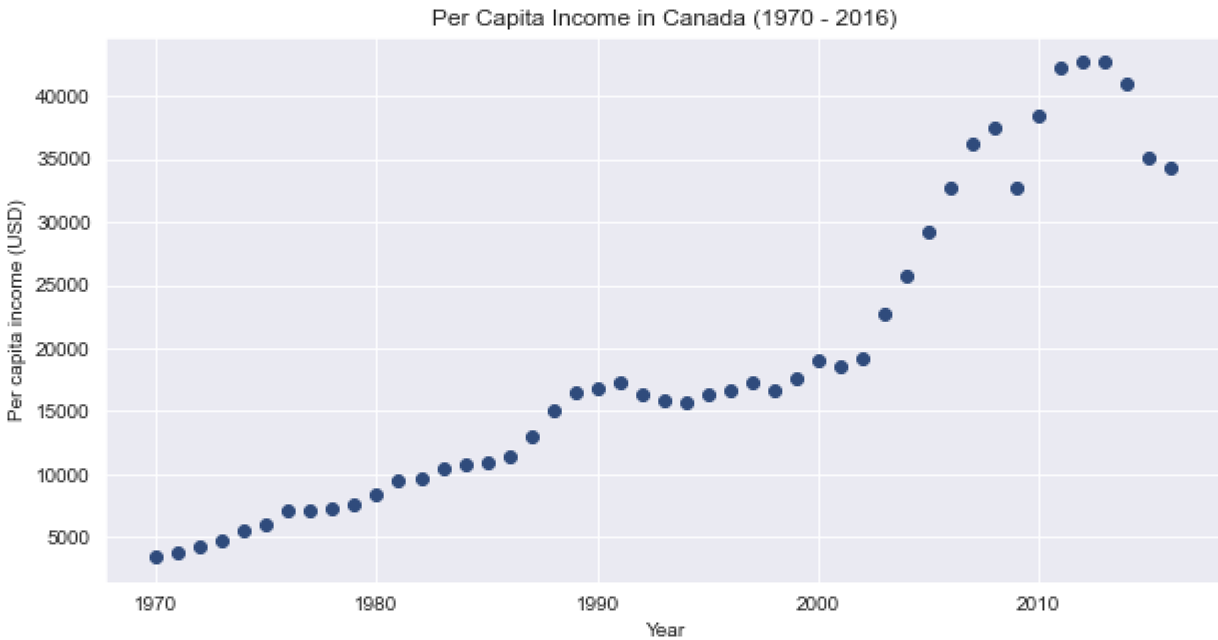
	year	per capita income (US$)
count	47.000000	47.000000
mean	1993.000000	18920.137063
std	13.711309	12034.679438
min	1970.000000	3399.299037
25%	1981.500000	9526.914515
50%	1993.000000	16426.725480
75%	2004.500000	27458.601420
max	2016.000000	42676.468370

In [ ]:

```
# Step 2: Data visualization using seaborn and matpotlib

sns.set_style('darkgrid')
colors = ['#003f5c', '#2f4b7c', '#665191', '#a05195', '#d45087', '#f95d6a', '#ff7c43','#ffa600']
```

In [ ]:

```
plt.figure(figsize=(10,5))

plt.scatter(canada_df['year'], canada_df['per capita income (US$)'], color=colors[1])

plt.title("Per Capita Income in Canada (1970 - 2016)")
plt.xlabel('Year')
plt.ylabel('Per capita income (USD)')

plt.show()
```



We previously saw the general form of a linear regression line which was :-

$f(x) \equiv y = a + bx$

> Note ☢: This is for a single feature dataset

where, a is the 'intercept' and b is the 'coefficient'.

In [ ]:

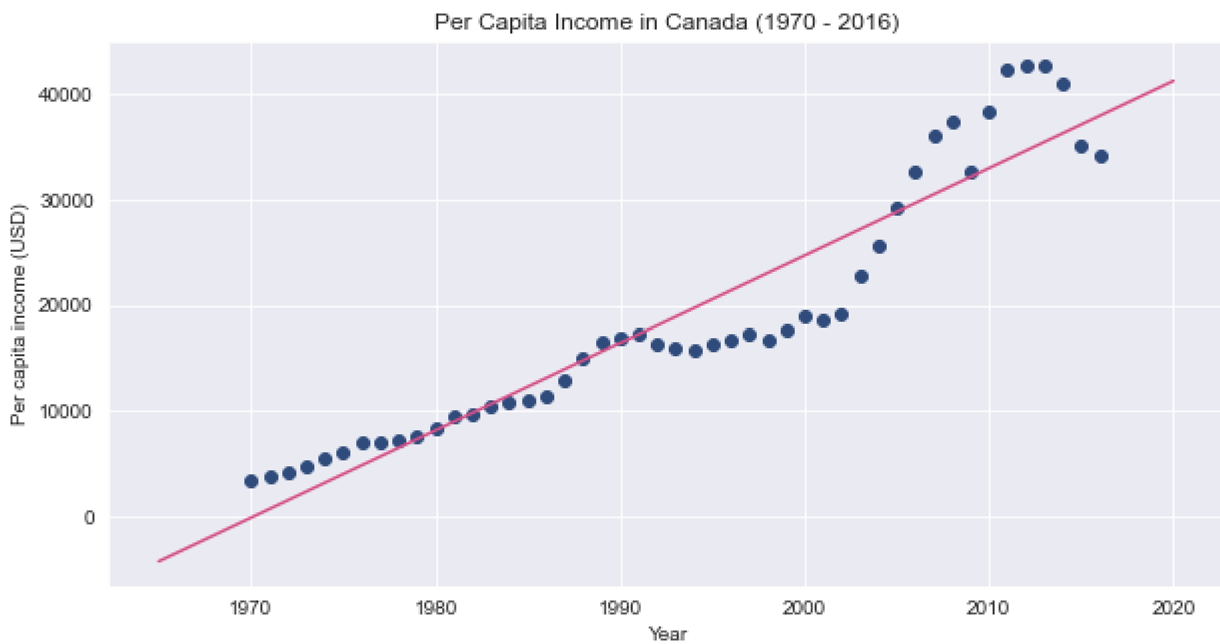
```
incReg = LinearRegression()
incReg.fit(canada_df[['year']], canada_df['per capita income (US$)'])

print(f'Coefficient : {incReg.coef_}\nIntercept : {incReg.intercept_}')
```

```
Coefficient : [828.46507522]
Intercept : -1632210.7578554575
```

In [ ]:

```
x = np.linspace(1965, 2020, 200)
y = incReg.coef_*x + incReg.intercept_


plt.figure(figsize=(10,5))
```

```
plt.scatter(canada_df['year'], canada_df['per capita income (US$)'], color=colors[1])
plt.plot(x, y, color=colors[4])

plt.title("Per Capita Income in Canada (1970 - 2016)")
plt.xlabel('Year')
plt.ylabel('Per capita income (USD)')

plt.show()
```



In [ ]:

```
# Lets try to predict the per capita income for the year 2023.

y_prediction = incReg.predict([[2023]])

print(f"Per capita income for the year 2023 will probably be : {y_prediction[0]:.4f}")
```

```
Per capita income for the year 2023 will probably be : 43774.0893
```

Like this we have trained our first linear regression machine learning model, along with that we have also tested and predicted using some never-before-seen data.

In [ ]:

```
boston = load_boston()
```

```
boston
```

Out[ ]:

```
{'data': array([[6.3200e-03, 1.8000e+01, 2.3100e+00, ..., 1.5300e+01, 3.9690e+02,
         4.9800e+00],
        [2.7310e-02, 0.0000e+00, 7.0700e+00, ..., 1.7800e+01, 3.9690e+02,
         9.1400e+00],
        [2.7290e-02, 0.0000e+00, 7.0700e+00, ..., 1.7800e+01, 3.9283e+02,
         4.0300e+00],
        ...,
        [6.0760e-02, 0.0000e+00, 1.1930e+01, ..., 2.1000e+01, 3.9690e+02,
         5.6400e+00],
        [1.0959e-01, 0.0000e+00, 1.1930e+01, ..., 2.1000e+01, 3.9345e+02,
         6.4800e+00],
        [4.7410e-02, 0.0000e+00, 1.1930e+01, ..., 2.1000e+01, 3.9690e+02,
         7.8800e+00]]),
 'target': array([24. , 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, 18.9, 15. ,
        18.9, 21.7, 20.4, 18.2, 19.9, 23.1, 17.5, 20.2, 18.2, 13.6, 19.6,
        15.2, 14.5, 15.6, 13.9, 16.6, 14.8, 18.4, 21. , 12.7, 14.5, 13.2,
        13.1, 13.5, 18.9, 20. , 21. , 24.7, 30.8, 34.9, 26.6, 25.3, 24.7,
        21.2, 19.3, 20. , 16.6, 14.4, 19.4, 19.7, 20.5, 25. , 23.4, 18.9,
        35.4, 24.7, 31.6, 23.3, 19.6, 18.7, 16. , 22.2, 25. , 33. , 23.5,
        19.4, 22. , 17.4, 20.9, 24.2, 21.7, 22.8, 23.4, 24.1, 21.4, 20. ,
        20.8, 21.2, 20.3, 28. , 23.9, 24.8, 22.9, 23.9, 26.6, 22.5, 22.2,
        23.6, 28.7, 22.6, 22. , 22.9, 25. , 20.6, 28.4, 21.4, 38.7, 43.8,
        33.2, 27.5, 26.5, 18.6, 19.3, 20.1, 19.5, 19.5, 20.4, 19.8, 19.4,
        21.7, 22.8, 18.8, 18.7, 18.5, 18.3, 21.2, 19.2, 20.4, 19.3, 22. ,
        20.3, 20.5, 17.3, 18.8, 21.4, 15.7, 16.2, 18. , 14.3, 19.2, 19.6,
        23. , 18.4, 15.6, 18.1, 17.4, 17.1, 13.3, 17.8, 14. , 14.4, 13.4,
        15.6, 11.8, 13.8, 15.6, 14.6, 17.8, 15.4, 21.5, 19.6, 15.3, 19.4,
        17. , 15.6, 13.1, 41.3, 24.3, 23.3, 27. , 50. , 50. , 50. , 22.7,
        25. , 50. , 23.8, 23.8, 22.3, 17.4, 19.1, 23.1, 23.6, 22.6, 29.4,
        23.2, 24.6, 29.9, 37.2, 39.8, 36.2, 37.9, 32.5, 26.4, 29.6, 50. ,
        32. , 29.8, 34.9, 37. , 30.5, 36.4, 31.1, 29.1, 50. , 33.3, 30.3,
        34.6, 34.9, 32.9, 24.1, 42.3, 48.5, 50. , 22.6, 24.4, 22.5, 24.4,
        20. , 21.7, 19.3, 22.4, 28.1, 23.7, 25. , 23.3, 28.7, 21.5, 23. ,
        26.7, 21.7, 27.5, 30.1, 44.8, 50. , 37.6, 31.6, 46.7, 31.5, 24.3,
        31.7, 41.7, 48.3, 29. , 24. , 25.1, 31.5, 23.7, 23.3, 22. , 20.1,
        22.2, 23.7, 17.6, 18.5, 24.3, 20.5, 24.5, 26.2, 24.4, 24.8, 29.6,
        42.8, 21.9, 20.9, 44. , 50. , 36. , 30.1, 33.8, 43.1, 48.8, 31. ,
        36.5, 22.8, 30.7, 50. , 43.5, 20.7, 21.1, 25.2, 24.4, 35.2, 32.4,
        32. , 33.2, 33.1, 29.1, 35.1, 45.4, 35.4, 46. , 50. , 32.2, 22. ,
        20.1, 23.2, 22.3, 24.8, 28.5, 37.3, 27.9, 23.9, 21.7, 28.6, 27.1,
        20.3, 22.5, 29. , 24.8, 22. , 26.4, 33.1, 36.1, 28.4, 33.4, 28.2,
        22.8, 20.3, 16.1, 22.1, 19.4, 21.6, 23.8, 16.2, 17.8, 19.8, 23.1,
        21. , 23.8, 23.1, 20.4, 18.5, 25. , 24.6, 23. , 22.2, 19.3, 22.6,
```

```
       19.8, 17.1, 19.4, 22.2, 20.7, 21.1, 19.5, 18.5, 20.6, 19. , 18.7,
       32.7, 16.5, 23.9, 31.2, 17.5, 17.2, 23.1, 24.5, 26.6, 22.9, 24.1,
       18.6, 30.1, 18.2, 20.6, 17.8, 21.7, 22.7, 22.6, 25. , 19.9, 20.8,
       16.8, 21.9, 27.5, 21.9, 23.1, 50. , 50. , 50. , 50. , 50. , 13.8,
       13.8, 15. , 13.9, 13.3, 13.1, 10.2, 10.4, 10.9, 11.3, 12.3,  8.8,
        7.2, 10.5,  7.4, 10.2, 11.5, 15.1, 23.2,  9.7, 13.8, 12.7, 13.1,
       12.5,  8.5,  5. ,  6.3,  5.6,  7.2, 12.1,  8.3,  8.5,  5. , 11.9,
       27.9, 17.2, 27.5, 15. , 17.2, 17.9, 16.3,  7. ,  7.2,  7.5, 10.4,
        8.8,  8.4, 16.7, 14.2, 20.8, 13.4, 11.7,  8.3, 10.2, 10.9, 11. ,
        9.5, 14.5, 14.1, 16.1, 14.3, 11.7, 13.4,  9.6,  8.7,  8.4, 12.8,
       10.5, 17.1, 18.4, 15.4, 10.8, 11.8, 14.9, 12.6, 14.1, 13. , 13.4,
       15.2, 16.1, 17.8, 14.9, 14.1, 12.7, 13.5, 14.9, 20. , 16.4, 17.7,
       19.5, 20.2, 21.4, 19.9, 19. , 19.1, 19.1, 20.1, 19.9, 19.6, 23.2,
       29.8, 13.8, 13.3, 16.7, 12. , 14.6, 21.4, 23. , 23.7, 25. , 21.8,
       20.6, 21.2, 19.1, 20.6, 15.2,  7. ,  8.1, 13.6, 20.1, 21.8, 24.5,
       23.1, 19.7, 18.3, 21.2, 17.5, 16.8, 22.4, 20.6, 23.9, 22. , 11.9]),
 'feature_names': array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
       'TAX', 'PTRATIO', 'B', 'LSTAT'], dtype='<U7'),
 'DESCR': ".. _boston_dataset:\n\nBoston house prices dataset\n---------------------------\n\n**Data Set Characteristics:**  \n\n    :Number of Instances: 506 \n\n    :Number of Attributes: 13 numeric/categorical predictive. Median Value (attribute 14) is usually the target.\n\n    :Attribute Information (in order):\n        - CRIM     per capita crime rate by town\n        - ZN       proportion of residential land zoned for lots over 25,000 sq.ft.\n        - INDUS    proportion of non-retail business acres per town\n        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)\n        - NOX      nitric oxides concentration (parts per 10 million)\n        - RM       average number of rooms per dwelling\n        - AGE      proportion of owner-occupied units built prior to 1940\n        - DIS      weighted distances to five Boston employment centres\n        - RAD      index of accessibility to radial highways\n        - TAX      full-value property-tax rate per $10,000\n        - PTRATIO  pupil-teacher ratio by town\n        - B        1000(Bk - 0.63)^2 where Bk is the proportion of black people by town\n        - LSTAT    % lower status of the population\n        - MEDV     Median value of owner-occupied homes in $1000's\n\n    :Missing Attribute Values: None\n\n    :Creator: Harrison, D. and Rubinfeld, D.L.\n\nThis is a copy of UCI ML housing dataset.\nhttps://archive.ics.uci.edu/ml/machine-learning-databases/housing/\n\n\nThis dataset was taken from the StatLib library which is maintained at Carnegie Mellon University.\n\nThe Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic\nprices and the demand for clean air', J. Environ. Economics & Management,\nvol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnostics\n...', Wiley, 1980.   N.B. Various transformations are used in the table on\npages 244-261 of the latter.\n\nThe Boston house-price data has been used in many machine learning papers that address regression\nproblems.   \n     \n.. topic:: References\n\n   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sources of Collinearity', Wiley, 1980. 244-261.\n   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the Tenth International Conference of Machine Learning, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.\n",
 'filename': 'c:\\Python39\\lib\\site-packages\\sklearn\\datasets\\data\\boston_house_prices.csv'}
```

- CRIM - per capita crime rate by town
- ZN - proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS - proportion of non-retail business acres per town.

- CHAS - Charles River dummy variable (1 if tract bounds river; 0 otherwise)
- NOX - nitric oxides concentration (parts per 10 million)
- RM - average number of rooms per dwelling
- DIS - weighted distances to five Boston employment centres
- RAD - index of accessibility to radial highways
- TAX - full-value property-tax rate per $10,000
- PTRATIO - pupil-teacher ratio by town
- B - 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
- LSTAT - % lower status of the population
- MEDV - Median value of owner-occupied homes in $1000's

In [ ]:

```
boston.feature_names
```

Out[ ]:

```
array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
       'TAX', 'PTRATIO', 'B', 'LSTAT'], dtype='<U7')
```

In [ ]:

```
# Lets look at a sample data point

boston.data[23]
```

Out[ ]:

```
array([  0.98843,   0.     ,   8.14   ,   0.     ,   0.538  ,   5.813  ,
       100.     ,   4.0952 ,   4.     , 307.     ,  21.     , 394.54   ,
        19.88   ])
```

In [ ]:

```
# Target value of the sample data point

boston.target[23]
```

Out[ ]:

```
14.5
```

In [ ]:

```
# Lets now try to convert our dataset to a pandas dataframe for better data analysis

boston_df = pd.DataFrame(boston.data, columns=boston.feature_names)
boston_df['Target'] = pd.DataFrame(boston.target)

boston_df.head()
```

Out[ ]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	Target
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

In [ ]:

```
# Basic statistical values of our dataset can be known by using the describe method

boston_df.describe()
```

Out[ ]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.237154	18.455534	356
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.537116	2.164946	91
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000	0
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000	17.400000	375
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.000000	19.050000	391
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.000000	20.200000	396
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000	396

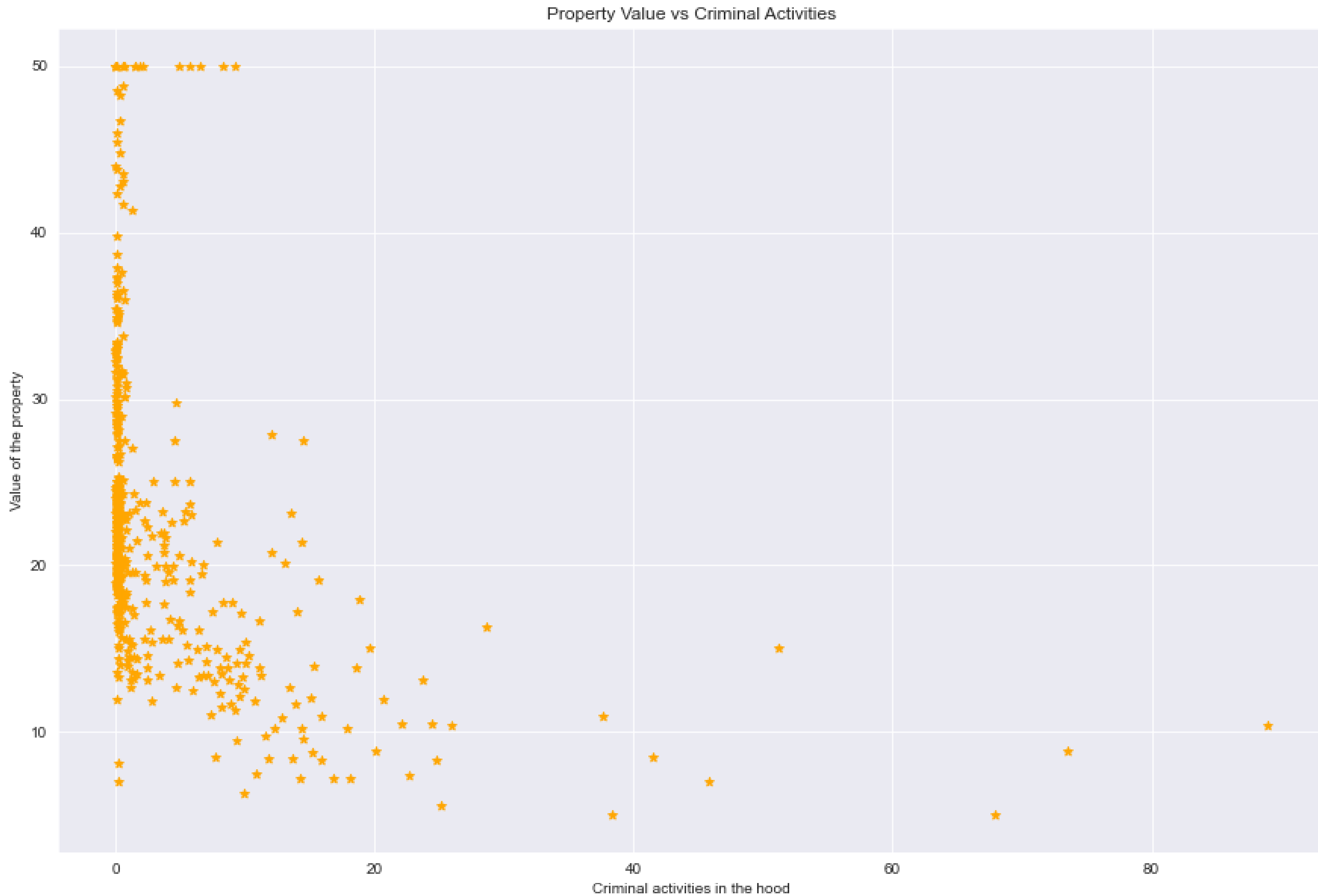
In [ ]:

```
# Step 2: Data visualization using seaborn and matpotlib

sns.set_style('darkgrid')
colors = ['#003f5c', '#2f4b7c', '#665191', '#a05195', '#d45087', '#f95d6a', '#ff7c43','#ffa600']
```

In [ ]:

```
# Let us see the effect of Crime on the property price.

plt.figure(figsize=(15,10))
```

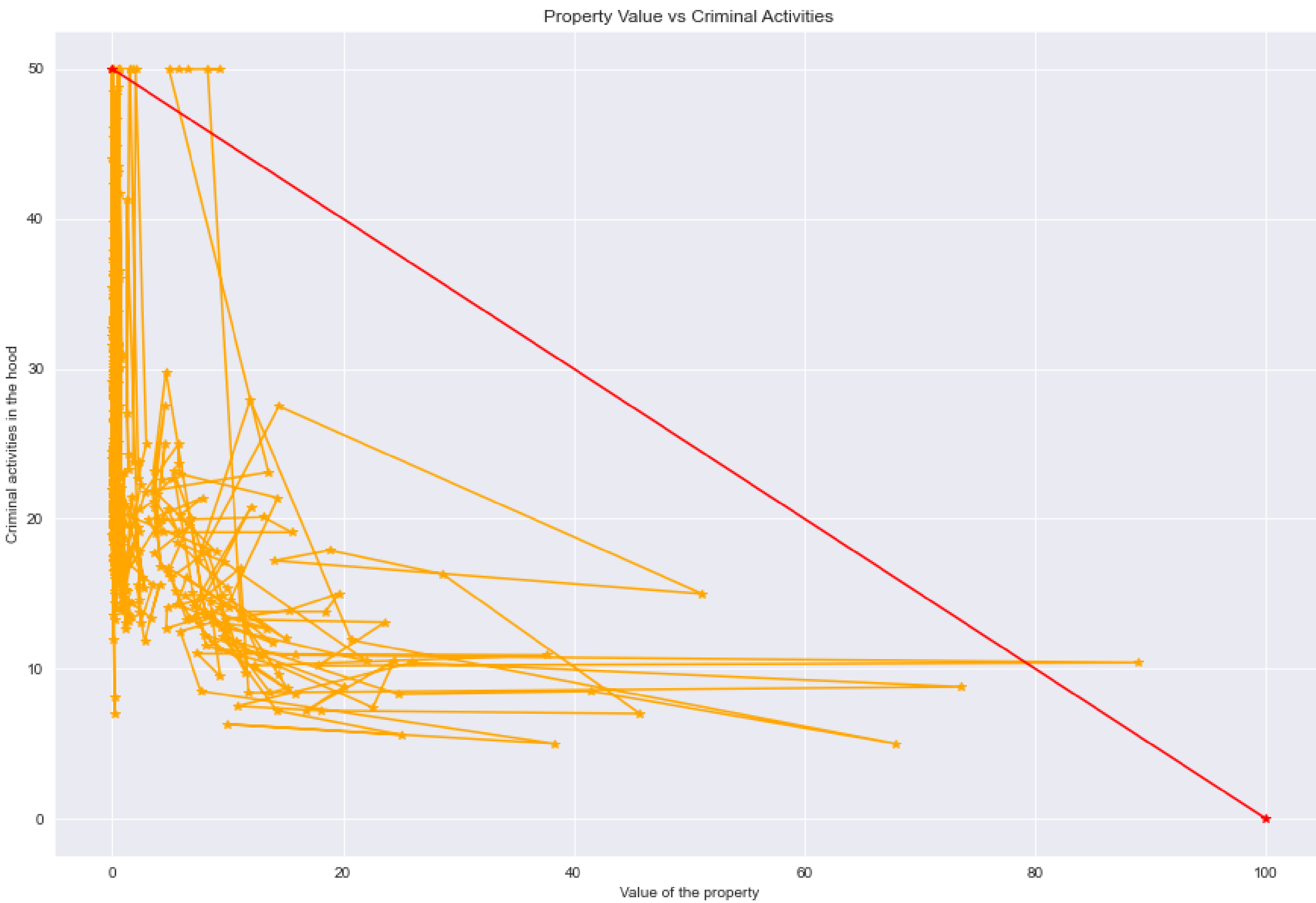
```python
plt.scatter(boston_df.CRIM, boston_df.Target, color=colors[7], marker='*')

plt.ylabel('Value of the property')
plt.xlabel('Criminal activities in the hood')
plt.title('Property Value vs Criminal Activities')

plt.show()
```



In [ ]:

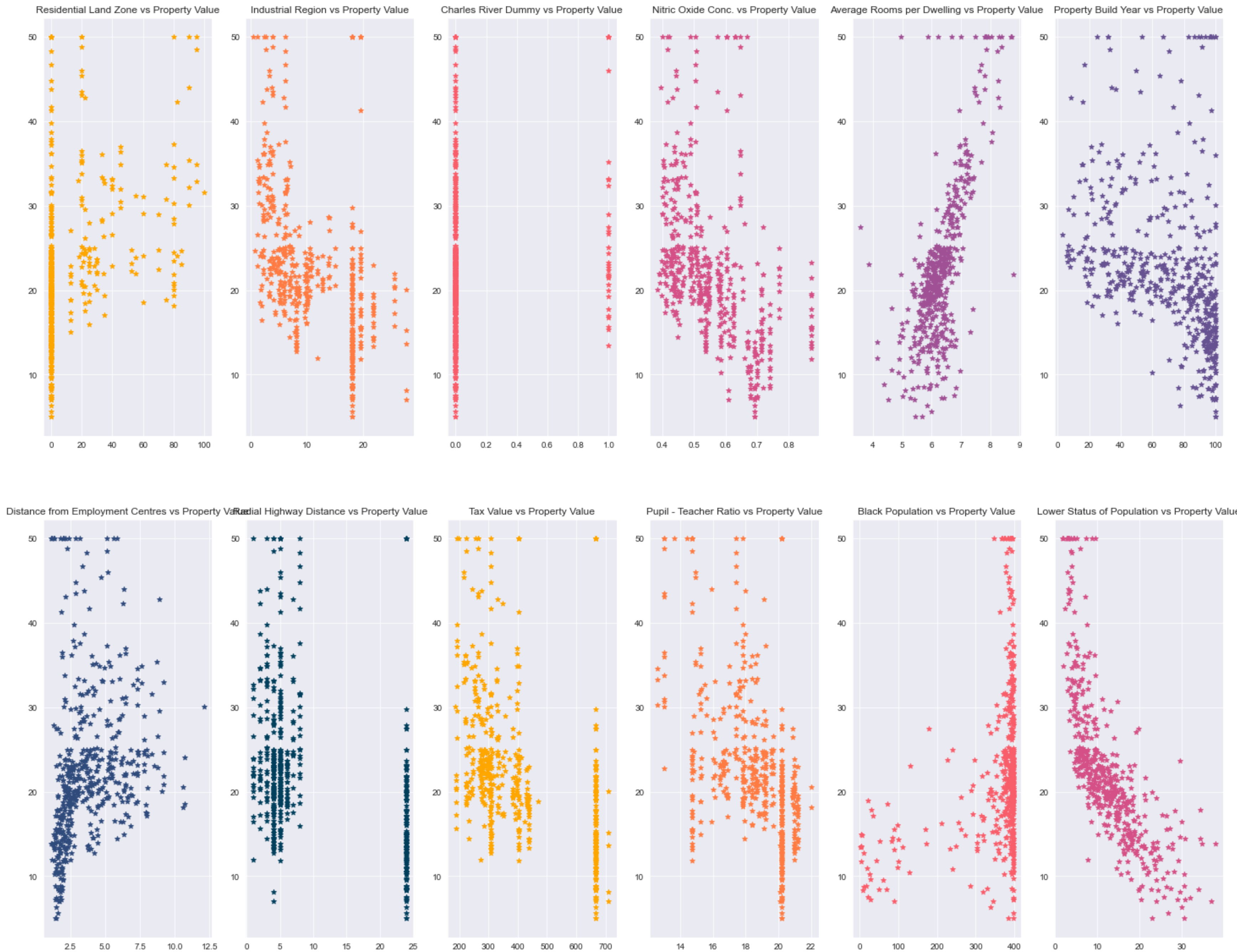
```
# This type of plots are redundant and inconclusive...

plt.figure(figsize=(15,10))

plt.plot(boston_df.CRIM, boston_df.Target, color=colors[7], marker='*')
plt.plot([0,100], [50, 0], color='red', marker='*')
```

```python
plt.xlabel('Value of the property')
plt.ylabel('Criminal activities in the hood')
plt.title('Property Value vs Criminal Activities')

plt.show()
```



In [ ]:

```python
# Likewise lets see how other features change the value of these properties

fig, ax = plt.subplots(ncols=6, nrows=2, figsize=(25,20))

ax[0,0].scatter(boston_df.ZN, boston_df.Target, marker='*', color=colors[7])    # Continuous Data
ax[0,0].set_title('Residential Land Zone vs Property Value')

ax[0,1].scatter(boston_df.INDUS, boston_df.Target, marker='*', color=colors[6]) # Continuous Data
ax[0,1].set_title('Industrial Region vs Property Value')

ax[0,2].scatter(boston_df.CHAS, boston_df.Target, marker='*', color=colors[5])  # Nominal / Categorical Data (1/0)
ax[0,2].set_title('Charles River Dummy vs Property Value')

ax[0,3].scatter(boston_df.NOX, boston_df.Target, marker='*', color=colors[4])   # Continuous Data
ax[0,3].set_title('Nitric Oxide Conc. vs Property Value')

ax[0,4].scatter(boston_df.RM, boston_df.Target, marker='*', color=colors[3])    # Continuous Data
ax[0,4].set_title('Average Rooms per Dwelling vs Property Value')

ax[0,5].scatter(boston_df.AGE, boston_df.Target, marker='*', color=colors[2])   # Continuous Data
ax[0,5].set_title('Property Build Year vs Property Value')

ax[1,0].scatter(boston_df.DIS, boston_df.Target, marker='*', color=colors[1])   # Continuous Data
ax[1,0].set_title('Distance from Employment Centres vs Property Value')

ax[1,1].scatter(boston_df.RAD, boston_df.Target, marker='*', color=colors[0])   # Ordinal Data (index or 0/1/2)
ax[1,1].set_title('Radial Highway Distance vs Property Value')

ax[1,2].scatter(boston_df.TAX, boston_df.Target, marker='*', color=colors[7])   # Continuous Data
ax[1,2].set_title('Tax Value vs Property Value')

ax[1,3].scatter(boston_df.PTRATIO, boston_df.Target, marker='*', color=colors[6])   # Continuous Data
ax[1,3].set_title('Pupil - Teacher Ratio vs Property Value')

ax[1,4].scatter(boston_df.B, boston_df.Target, marker='*', color=colors[5])     # Continuous Data
ax[1,4].set_title('Black Population vs Property Value')

ax[1,5].scatter(boston_df.LSTAT, boston_df.Target, marker='*', color=colors[4]) # Continuous Data
ax[1,5].set_title('Lower Status of Population vs Property Value')

plt.show()
```



In [ ]:

```
# Step 3 : Dividing our data into training and testing set and creating our model
```

```
X = boston_df.drop(['Target', 'RAD', 'CHAS'], axis=1)
y = boston_df.Target

print(f'Example Data : \n{X}\n-----------\nLabels : \n{y}')
```

```
Example Data :
        CRIM    ZN  INDUS    NOX     RM   AGE     DIS    TAX  PTRATIO       B  \
0    0.00632  18.0   2.31  0.538  6.575  65.2  4.0900  296.0     15.3  396.90
1    0.02731   0.0   7.07  0.469  6.421  78.9  4.9671  242.0     17.8  396.90
2    0.02729   0.0   7.07  0.469  7.185  61.1  4.9671  242.0     17.8  392.83
3    0.03237   0.0   2.18  0.458  6.998  45.8  6.0622  222.0     18.7  394.63
4    0.06905   0.0   2.18  0.458  7.147  54.2  6.0622  222.0     18.7  396.90
..       ...   ...    ...    ...    ...   ...     ...    ...      ...     ...
501  0.06263   0.0  11.93  0.573  6.593  69.1  2.4786  273.0     21.0  391.99
502  0.04527   0.0  11.93  0.573  6.120  76.7  2.2875  273.0     21.0  396.90
503  0.06076   0.0  11.93  0.573  6.976  91.0  2.1675  273.0     21.0  396.90
504  0.10959   0.0  11.93  0.573  6.794  89.3  2.3889  273.0     21.0  393.45
505  0.04741   0.0  11.93  0.573  6.030  80.8  2.5050  273.0     21.0  396.90

     LSTAT
0     4.98
1     9.14
2     4.03
3     2.94
4     5.33
..     ...
501   9.67
502   9.08
503   5.64
504   6.48
505   7.88

[506 rows x 11 columns]
-----------
Labels :
0      24.0
1      21.6
2      34.7
3      33.4
4      36.2
       ...
501    22.4
502    20.6
503    23.9
504    22.0
```

```
505    11.9
Name: Target, Length: 506, dtype: float64
```

In [ ]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42, test_size=0.25)
```

In [ ]:

```
linReg = LinearRegression()

linReg.fit(X_train, y_train)
```

Out[ ]:

```
LinearRegression()
```

We previously saw the general form of a linear regression line which was :-

$f(x) \equiv y = a + bx$

> Note ☢: This is for a single feature dataset

where, a is the 'intercept' and b is the 'coefficient'.

Whereas, for a multi-feature dataset the equation becomes :-

$f(x) \equiv y = a + b_1x_1 + b_2x_2 + b_3x_3 + b_4x_4 + \ldots + b_nx_n$

now lets see the equation our model created for each of the feature (individually)

In [ ]:

```
print(f"Coefficients : {linReg.coef_}\n\nIntercept : {linReg.intercept_}")
```

```
Coefficients : [-9.65063144e-02  2.35306584e-02  1.12922364e-02 -1.31790536e+01
  4.72737651e+00 -1.14251478e-02 -1.39775778e+00  1.00464496e-03
 -8.44004100e-01  1.29995400e-02 -5.26459515e-01]

Intercept : 23.290795392143302
```

> ☢ : We cannot possibly visualize this function as the number of variables is too high, going beyond the scope of natural dimensionality.

In [ ]:

```
# Let us check the accuracy of this model
```

```
linReg.score(X_test, y_test) * 100
```

Out[ ]:
```
65.09699144711765
```

In [ ]:
```
# Lets predict a value using the model we just created

X_predict = boston_df.drop(['Target', 'RAD', 'CHAS'], axis=1)

y_prediction = linReg.predict(X_predict.loc[[0]])  # Instead of this value (which was taken while curating this course),

print(f"Predicted property value : {y_prediction[0]}\nActual property value : {boston_df.Target[0]}")
```

```
Predicted property value : 31.192104356273376
Actual property value : 24.0
```

💀 This model is not at all accurate... 🥺 🥺

Accuracy in Linear regression models can be increased by using various regularization techniques which we will cover in some time.

In [ ]:
```
y_prediction = linReg.predict(X_test)
```

In [ ]:
```
# There is another parameter to calculate 'error' called the mean squared error, which can be calculated easily using num

print(f'Mean Squared Error (over the testing data) : {np.mean((y_prediction - y_test)**2)}')
```

```
Mean Squared Error (over the testing data) : 24.44157993899511
```

So, now we have successfully created our second linear regression model, and we have inferred that its accuracy is very less. In the upcoming sections we will see how we can increase our model's accuracy.

Advanced Linear Regression :

- Predictions from Linear Regression assume data points are normally distributed.
- Features and predicted data are often skewed.

> ♾ Data transformation can solve this issue.

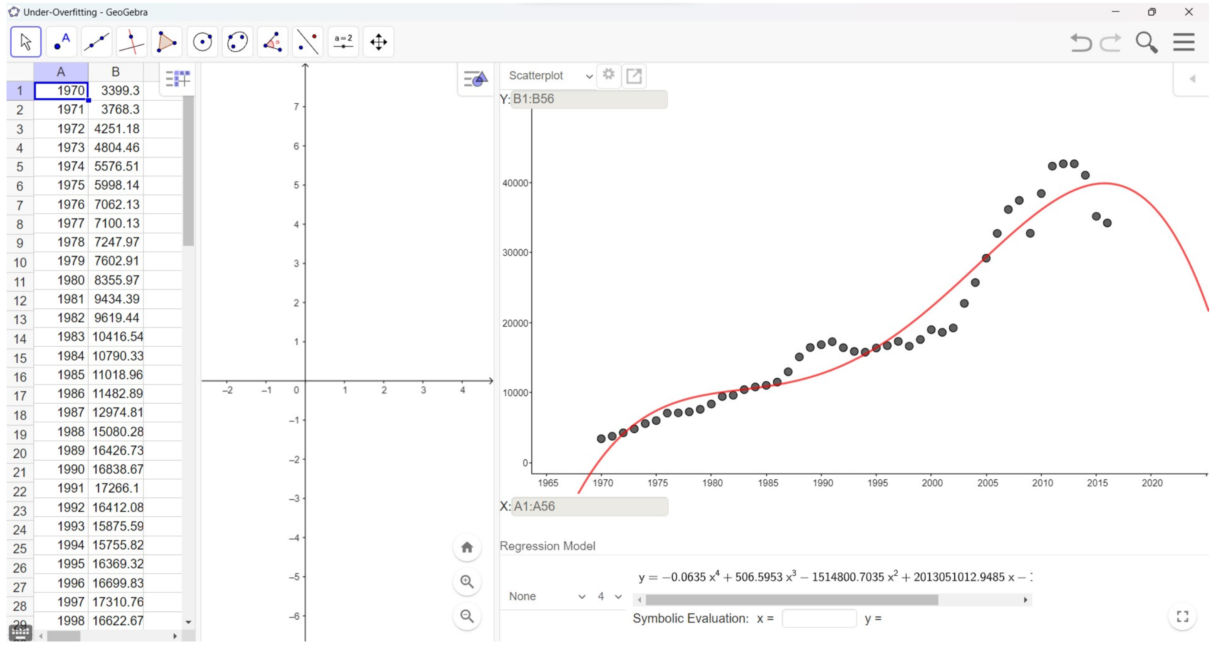
1. Addition of Polynomial Features :

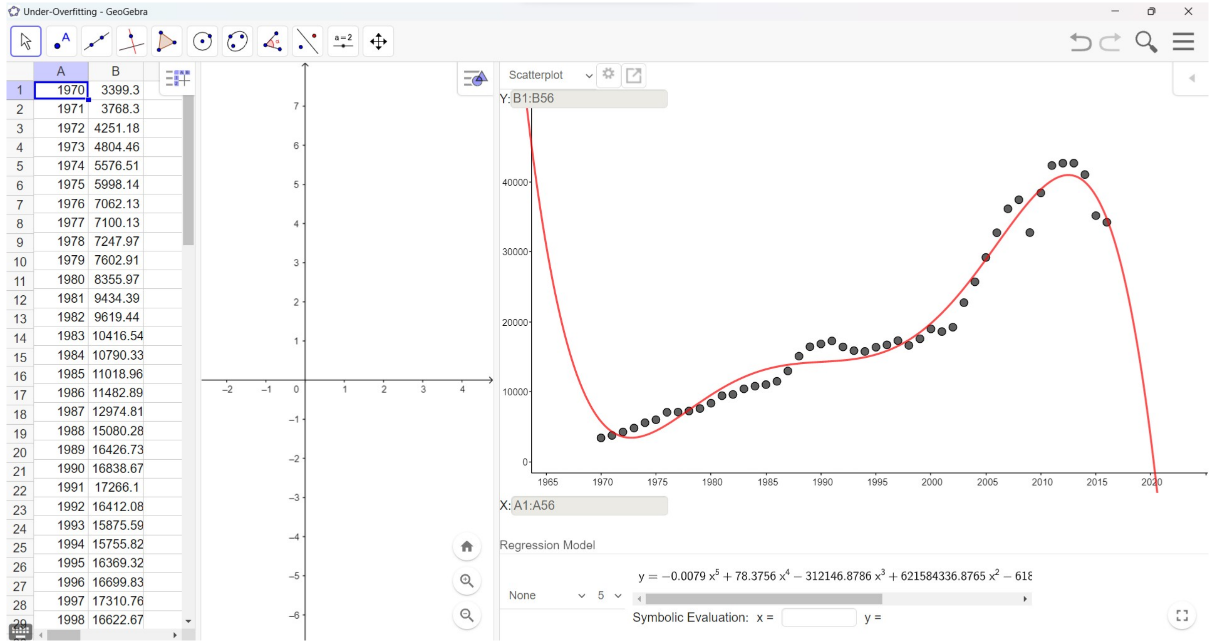
Capture higher order features of data by adding polynomial features, eg :

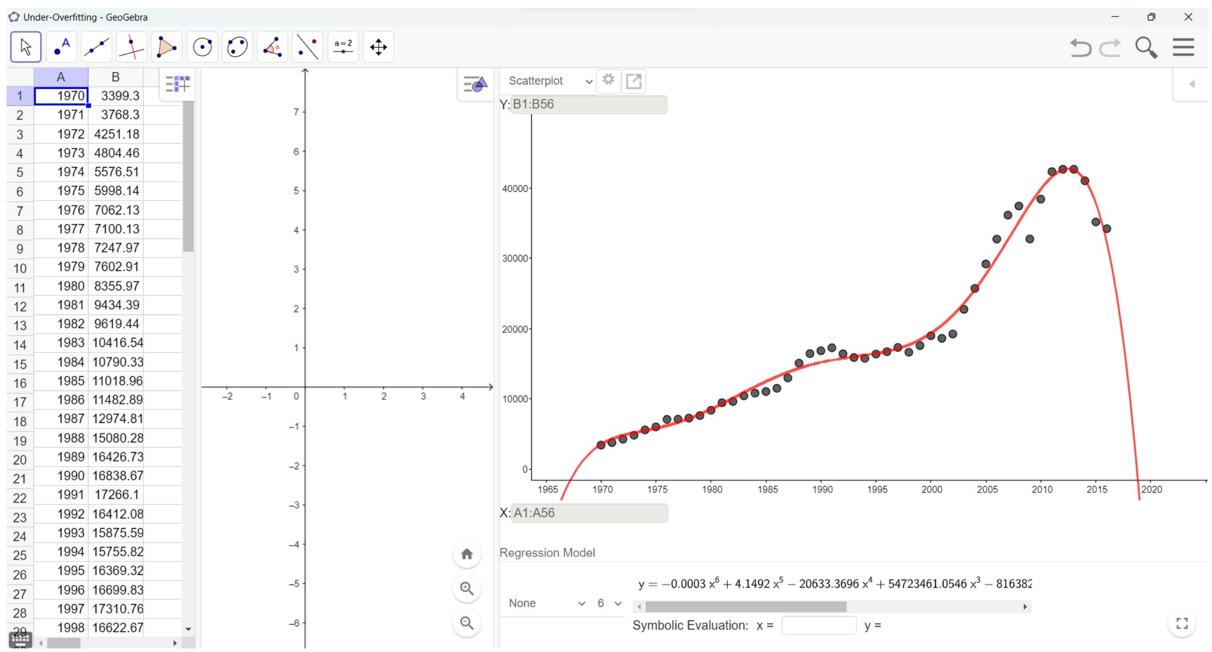
$f(x) = a + bx + cx^2 + dx^3 ... + px^n$ → for a single feature

OR

$f(x, y, z) = a + bx + cy + dz + exy + fyz + jx^2 ...$







In [ ]:

```
# On the per capita income data lets try to use a non linear model, we will take the polynomials degree such that it does
# Our probable candidates for fitting are 4th degree and 5th degree polynomials, lets check them one by one.

from sklearn.preprocessing import PolynomialFeatures
```

```python
polyFit = PolynomialFeatures(degree=4)
X_poly = polyFit.fit_transform(canada_df[['year']])

polyRig = LinearRegression()
polyRig.fit(X_poly, canada_df['per capita income (US$)'])
```

Out[ ]:

```
LinearRegression()
```

In [ ]:

```python
# Now lets plot our data alongwith the regression curve

plt.figure(figsize=(20,10))

plt.scatter(canada_df['year'], canada_df['per capita income (US$)'], color=colors[5], marker='*')
plt.plot(canada_df['year'], polyRig.predict(polyFit.fit_transform(canada_df[['year']])), color=colors[4])

plt.show()
```

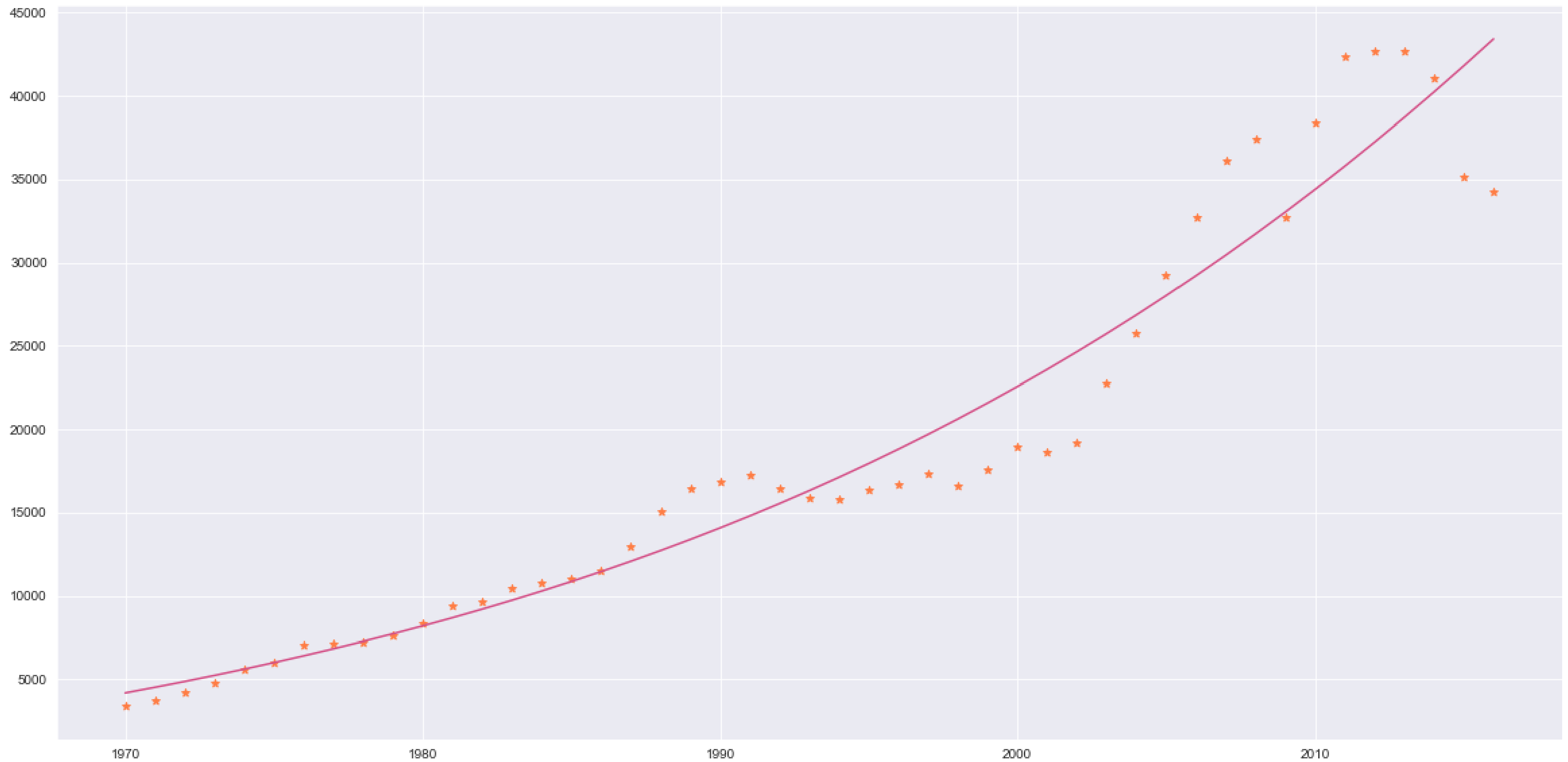
In [ ]:

```
# Lets predict the per capita income for the year 2023

y_prediction = polyRig.predict(polyFit.fit_transform([[2023]]))

print(f"Per capita income for the year 2023 will probably be : {y_prediction[0]:.4f}")
```

```
Per capita income for the year 2023 will probably be : 56017.2646
```

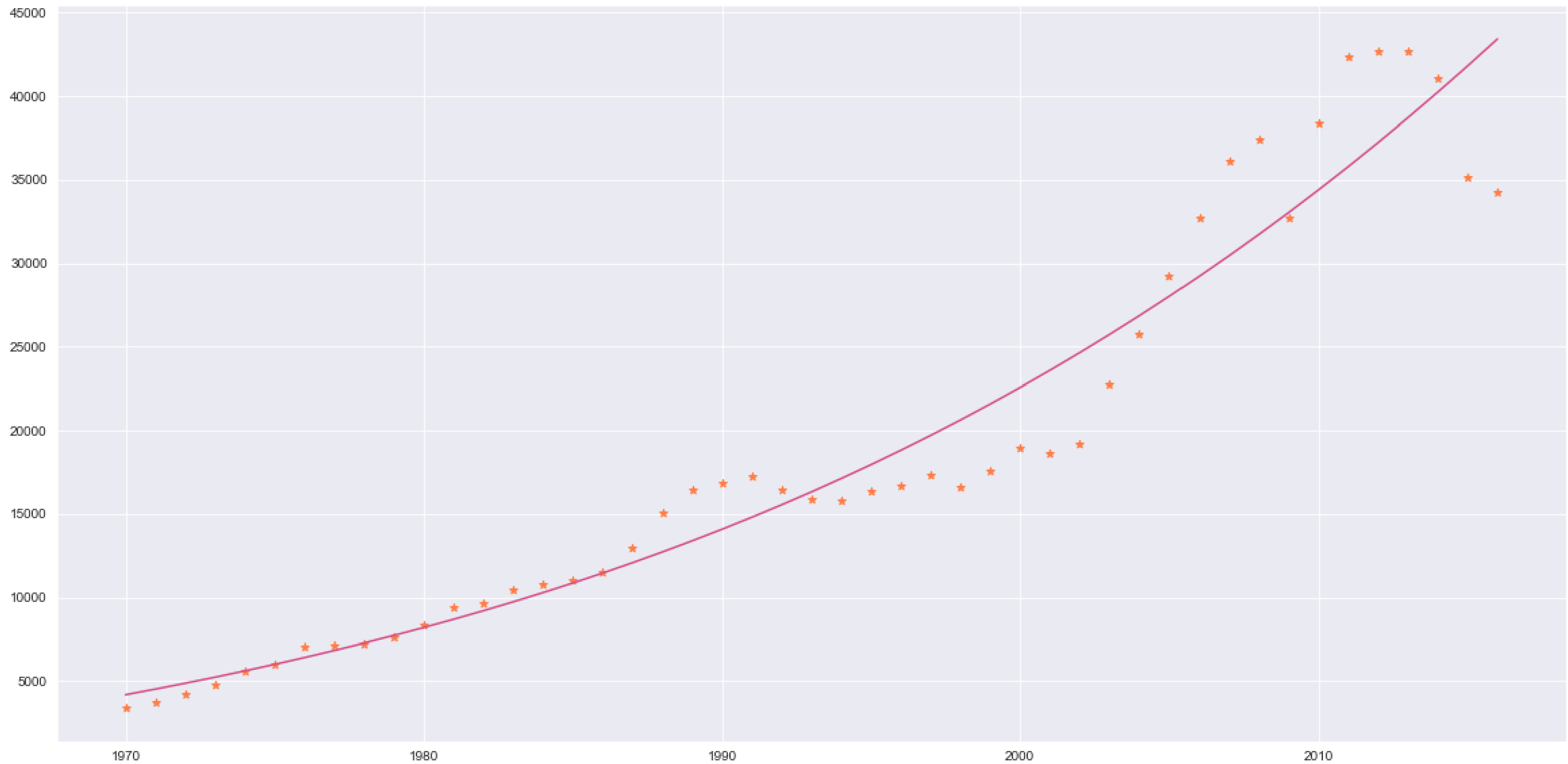
In [ ]:

```
# Now lets plot our data alongwith the regression curve
plt.figure(figsize=(20,10))

plt.scatter(canada_df['year'], canada_df['per capita income (US$)'], marker='*', color=colors[6])
plt.plot(canada_df['year'], polyRig.predict(polyFit.fit_transform(canada_df[['year']])), color=colors[4])

plt.show()
```



In [ ]:

```
# Now lets plot our data alongwith the regression curve
plt.figure(figsize=(20,10))

plt.scatter(canada_df['year'], canada_df['per capita income (US$)'], marker='*', color=colors[6])
plt.plot(canada_df['year'], polyRig.predict(polyFit.fit_transform(canada_df[['year']])), color=colors[4])

plt.show()
```



In [ ]:
```
# Lets predict the per capita income for the year 2023


y_prediction = polyRig.predict(polyFit.fit_transform([[2023]]))

print(f"Per capita income for the year 2023 will probably be : {y_prediction[0]:.4f}")
```

```
Per capita income for the year 2023 will probably be : 56017.2646
```

We can see that even if there is no notable change in the curvature of the graph, the result of both the predictions are different.

Prevention of Over and Under-fitting :-

1. Ridge Regression :- L2 Regression

This is also a linear model for regression, so the formula it uses to make predictions is the same as that for Linear Regression.

Prediction formula for Ridge regression is :-

$$\hat{y} = w[0]x[0] + w[1]x[1] + w[2]x[2] + ... + w[n]x[n] + b$$

↑ Feature!

> 🐎 In Ridge, the coeff $w[i]$ are chosen not only so that they predict well on the training set, but also fit relatively well on the additional testing data. We also want the magnitude of coefficient to be as small as possible (as near to 0 as possible)

Ridge regression shrinks the regression coefficients, so that variables, with minor contribution to the outcome, have their coefficients close to zero. The shrinkage of the coefficients is achieved by penalizing the regression model with a penalty term called L2-norm, which is the sum of the squared coefficients.

Smaller value of a feature's coefficient means that feature will have minimum effect on the predicted output. This constraint is used to implement **regularization**.

Training Score [Ridge Regression] < Training Score [Linear Regression]
Testing Score [Ridge Regression] > Testing Score [Linear Regression]

$J(a, b) = 1/2m \sum[((a + bx_{obs}) - y_{obs})]^2$ → Cost function of Linear Regression

$J(a, b) = 1/2m \sum[((a + bx_{obs}) - y_{obs})]^2 + \lambda \sum b_j^2$ → Cost function of Ridge Regression

Notice the additional term in equation 2, this is called *Penalty*, where λ is constant and it depends on the particular dataset we are using. Decreasing the value of λ decreases the training set performance but might help in generalization.

2. Lasso Regression :- L1 Regression

An alternative to Ridge regression for regularizing linear regression model. Lasso regression also restricts the coefficients to be close to zero, but in a slightly different way.

> 🐔 Consequence of L1 is that in this regression some coefficients are exactly 0. ie some features are completely ignored.

Less important features and unnecessary features are ignored to bring out effect from important features.

$J(a, b) = 1/2m \sum[((a + bx_{obs}) - y_{obs})]^2 + \lambda \sum |b_j|$ → Cost function of Lasso Regression

- Penalty selectively shrinks some coefficients.

- Can be used for feature selection.
- Slower to converge than Ridge regression.

$$\lambda \propto \frac{1}{features} \qquad \text{OR} \qquad features \propto \frac{1}{\lambda}$$

Lets see regularization of a model using Ridge Regression

In [ ]:
```
# Step 1 : Importing the required libraries and datasets

# We'll use the same boston dataset

from sklearn.linear_model import Ridge
```

In [ ]:
```
boston_df
```

Out[ ]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	Target
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
501	0.06263	0.0	11.93	0.0	0.573	6.593	69.1	2.4786	1.0	273.0	21.0	391.99	9.67	22.4
502	0.04527	0.0	11.93	0.0	0.573	6.120	76.7	2.2875	1.0	273.0	21.0	396.90	9.08	20.6
503	0.06076	0.0	11.93	0.0	0.573	6.976	91.0	2.1675	1.0	273.0	21.0	396.90	5.64	23.9
504	0.10959	0.0	11.93	0.0	0.573	6.794	89.3	2.3889	1.0	273.0	21.0	393.45	6.48	22.0
505	0.04741	0.0	11.93	0.0	0.573	6.030	80.8	2.5050	1.0	273.0	21.0	396.90	7.88	11.9

506 rows × 14 columns

In [ ]:
```
# Step 2 : Dividing our data into training and testing set and creating our model
```

```
X = boston_df.drop(['Target'], axis=1)
y = boston_df.Target

print(f'Example Data : \n{X}\n-----------\nLabels : \n{y}')
```

```
Example Data :
        CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
0    0.00632  18.0   2.31   0.0  0.538  6.575  65.2  4.0900  1.0  296.0
1    0.02731   0.0   7.07   0.0  0.469  6.421  78.9  4.9671  2.0  242.0
2    0.02729   0.0   7.07   0.0  0.469  7.185  61.1  4.9671  2.0  242.0
3    0.03237   0.0   2.18   0.0  0.458  6.998  45.8  6.0622  3.0  222.0
4    0.06905   0.0   2.18   0.0  0.458  7.147  54.2  6.0622  3.0  222.0
..       ...   ...    ...   ...    ...    ...   ...     ...  ...    ...
501  0.06263   0.0  11.93   0.0  0.573  6.593  69.1  2.4786  1.0  273.0
502  0.04527   0.0  11.93   0.0  0.573  6.120  76.7  2.2875  1.0  273.0
503  0.06076   0.0  11.93   0.0  0.573  6.976  91.0  2.1675  1.0  273.0
504  0.10959   0.0  11.93   0.0  0.573  6.794  89.3  2.3889  1.0  273.0
505  0.04741   0.0  11.93   0.0  0.573  6.030  80.8  2.5050  1.0  273.0

     PTRATIO       B  LSTAT
0       15.3  396.90   4.98
1       17.8  396.90   9.14
2       17.8  392.83   4.03
3       18.7  394.63   2.94
4       18.7  396.90   5.33
..       ...     ...    ...
501     21.0  391.99   9.67
502     21.0  396.90   9.08
503     21.0  396.90   5.64
504     21.0  393.45   6.48
505     21.0  396.90   7.88

[506 rows x 13 columns]
-----------
Labels :
0      24.0
1      21.6
2      34.7
3      33.4
4      36.2
       ...
501    22.4
502    20.6
503    23.9
```

```
504    22.0
505    11.9
Name: Target, Length: 506, dtype: float64
```

In [ ]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0, test_size=0.25)
```

In [ ]:
```
ridgeModel = Ridge().fit(X_train, y_train)

print(f"Training set score : {ridgeModel.score(X_train, y_train)*100}\nTesting set score : {ridgeModel.score(X_test, y_te
```

```
Training set score : 76.78858330771392
Testing set score : 62.66182204613857
```

Effect of alpha (λ) on the model score

$\lambda = 10$

In [ ]:
```
ridgeModel10 = Ridge(alpha=10).fit(X_train, y_train)

print(f"Training set score : {ridgeModel10.score(X_train, y_train)*100}\nTesting set score : {ridgeModel10.score(X_test,
```

```
Training set score : 76.23745182677773
Testing set score : 61.327730472069966
```

We can see that underfitting has started, neither the training score nor the testing score is good.

$\lambda = 1$

In [ ]:
```
ridgeModel1 = Ridge(alpha=1).fit(X_train, y_train)

print(f"Training set score : {ridgeModel1.score(X_train, y_train)*100}\nTesting set score : {ridgeModel1.score(X_test, y_
```

```
Training set score : 76.78858330771392
Testing set score : 62.66182204613857
```

$\lambda = 0.1$

In [ ]:
```
ridgeModel01 = Ridge(alpha=0.1).fit(X_train, y_train)

print(f"Training set score : {ridgeModel01.score(X_train, y_train)*100}\nTesting set score : {ridgeModel01.score(X_test,
```

```
Training set score : 76.97119401063664
```

```
Testing set score : 63.42855934691842
```

$\lambda = 0.0001$

```
In [ ]: ridgeModel00001 = Ridge(alpha=0.0001).fit(X_train, y_train)

        print(f"Training set score : {ridgeModel00001.score(X_train, y_train)*100}\nTesting set score : {ridgeModel00001.score(X_
```

```
Training set score : 76.9769948804977
Testing set score : 63.546264354434875
```

Not much but the relative accuracy has surely increased. The reason of this low accuracy is probably the size of the dataset, Linear models or any other model require lots and lots of data to create a truly accurate model.

So, let us now use the extended version of the Boston Housing Dataset which can be accessed using the mglearn library.

```
In [ ]: import mglearn

        X, y = mglearn.datasets.load_extended_boston()
```

```
In [ ]: X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

```
In [ ]: # Lets first check the accuracy of a simple linear model

        linReg = LinearRegression().fit(X_train, y_train)

        print(f"Training set score : {linReg.score(X_train, y_train)*100}\nTesting set score : {linReg.score(X_test, y_test)*100}
```

```
Training set score : 95.20519609032729
Testing set score : 60.74721959665881
```

Training score is pretty decent but the testing score straight up sucks!!!

```
In [ ]: # Let us now use the Ridge Regression Model

        ridgeModelExt = Ridge().fit(X_train, y_train)

        print(f"Training set score : {ridgeModelExt.score(X_train, y_train)*100}\nTesting set score : {ridgeModelExt.score(X_test
```

```
Training set score : 88.5796658517094
```

```
Testing set score : 75.27683481744748
```

Testing score has exponentially increased even with the default value of λ

$\lambda = 10$

In [ ]:
```
ridgeModelExt10 = Ridge(alpha=10).fit(X_train, y_train)

print(f"Training set score : {ridgeModelExt10.score(X_train, y_train)*100}\nTesting set score : {ridgeModelExt10.score(X_
```

```
Training set score : 78.82787115369615
Testing set score : 63.59411489177311
```

$\lambda = 1$

In [ ]:
```
ridgeModelExt1 = Ridge(alpha=1).fit(X_train, y_train)

print(f"Training set score : {ridgeModelExt1.score(X_train, y_train)*100}\nTesting set score : {ridgeModelExt1.score(X_te
```

```
Training set score : 88.5796658517094
Testing set score : 75.27683481744748
```

$\lambda = 0.1$

In [ ]:
```
ridgeModelExt01 = Ridge(alpha=0.1).fit(X_train, y_train)

print(f"Training set score : {ridgeModelExt01.score(X_train, y_train)*100}\nTesting set score : {ridgeModelExt01.score(X_
```

```
Training set score : 92.82273685001994
Testing set score : 77.22067936479662
```

$\lambda = 0.0001$

In [ ]:
```
ridgeModelExt00001 = Ridge(alpha=0.0001).fit(X_train, y_train)

print(f"Training set score : {ridgeModelExt00001.score(X_train, y_train)*100}\nTesting set score : {ridgeModelExt00001.sc
```

```
Training set score : 95.13789921673391
Testing set score : 61.831957815371055
```

Lets see regularization of a model using Lasso Regression

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

In [ ]:
```
lassoModel = Lasso().fit(X_train, y_train)

print(f"Training set score : {lassoModel.score(X_train, y_train)*100}\nTesting set score : {lassoModel.score(X_test, y_te
```

```
Training set score : 29.323768991114598
Testing set score : 20.937503255272272
Number of features used : 4
```

Very less number of features are used, and for the reference there are 104 features in this dataset... ☢️☢️💀💀

$\lambda = 0.01$

In [ ]:
```
lassoModel001 = Lasso(alpha=0.01, max_iter=10000).fit(X_train, y_train)

print(f"Training set score : {lassoModel001.score(X_train, y_train)*100}\nTesting set score : {lassoModel001.score(X_test
```

```
Training set score : 89.622265110865
Testing set score : 76.56571174549978
Number of features used : 33
```

max_iter → Maximum number of iterations taken for the solvers to converge.

$\lambda = 0.0001$

In [ ]:
```
lassoModel00001 = Lasso(alpha=0.0001, max_iter=100000).fit(X_train, y_train)

print(f"Training set score : {lassoModel00001.score(X_train, y_train)*100}\nTesting set score : {lassoModel00001.score(X_
```

```
Training set score : 95.07158754515463
Testing set score : 64.37467421273729
Number of features used : 96
```

Elastic Net Regression :

The best of both worlds! Usually, Ridge is the first choice between the two. But we can use both in Elastic Net or EN Regression.

$J(a, b) = 1/2m \sum[((a + bx_{obs}) - y_{obs})]^2 + \lambda \sum |b_j| + \lambda \sum b_j^2$ → Cost function

Notice both the penalty functions, each correspond to Lasso and Ridge respectively.

Importance of Feature Selection :

- Reducing the number of features is another way of preventing overfitting.
- For some models fewer features can improve fitting time.
- Identifying most critical features can improve model interpretability.

Lets see regularization of a model using Elastic Net Regression

In [ ]:

```
# This time we will be using another dataset called the diabetes dataset

from sklearn.datasets import load_diabetes
from sklearn.linear_model import ElasticNet
from sklearn.metrics import r2_score    # Another error metric

diabetes = load_diabetes()

diabetes
```

Out[ ]:

```
{'data': array([[ 0.03807591,  0.05068012,  0.06169621, ..., -0.00259226,
          0.01990842, -0.01764613],
        [-0.00188202, -0.04464164, -0.05147406, ..., -0.03949338,
         -0.06832974, -0.09220405],
        [ 0.08529891,  0.05068012,  0.04445121, ..., -0.00259226,
          0.00286377, -0.02593034],
        ...,
        [ 0.04170844,  0.05068012, -0.01590626, ..., -0.01107952,
         -0.04687948,  0.01549073],
        [-0.04547248, -0.04464164,  0.03906215, ...,  0.02655962,
          0.04452837, -0.02593034],
        [-0.04547248, -0.04464164, -0.0730303 , ..., -0.03949338,
         -0.00421986,  0.00306441]]),
 'target': array([151.,  75., 141., 206., 135.,  97., 138.,  63., 110., 310., 101.,
         69., 179., 185., 118., 171., 166., 144.,  97., 168.,  68.,  49.,
         68., 245., 184., 202., 137.,  85., 131., 283., 129.,  59., 341.,
         87.,  65., 102., 265., 276., 252.,  90., 100.,  55.,  61.,  92.,
        259.,  53., 190., 142.,  75., 142., 155., 225.,  59., 104., 182.,
        128.,  52.,  37., 170., 170.,  61., 144.,  52., 128.,  71., 163.,
        150.,  97., 160., 178.,  48., 270., 202., 111.,  85.,  42., 170.,
        200., 252., 113., 143.,  51.,  52., 210.,  65., 141.,  55., 134.,
         42., 111.,  98., 164.,  48.,  96.,  90., 162., 150., 279.,  92.,
         83., 128., 102., 302., 198.,  95.,  53., 134., 144., 232.,  81.,
        104.,  59., 246., 297., 258., 229., 275., 281., 179., 200., 200.,
        173., 180.,  84., 121., 161.,  99., 109., 115., 268., 274., 158.,
        107.,  83., 103., 272.,  85., 280., 336., 281., 118., 317., 235.,
         60., 174., 259., 178., 128.,  96., 126., 288.,  88., 292.,  71.,
```

```
       197., 186.,  25.,  84.,  96., 195.,  53., 217., 172., 131., 214.,
        59.,  70., 220., 268., 152.,  47.,  74., 295., 101., 151., 127.,
       237., 225.,  81., 151., 107.,  64., 138., 185., 265., 101., 137.,
       143., 141.,  79., 292., 178.,  91., 116.,  86., 122.,  72., 129.,
       142.,  90., 158.,  39., 196., 222., 277.,  99., 196., 202., 155.,
        77., 191.,  70.,  73.,  49.,  65., 263., 248., 296., 214., 185.,
        78.,  93., 252., 150.,  77., 208.,  77., 108., 160.,  53., 220.,
       154., 259.,  90., 246., 124.,  67.,  72., 257., 262., 275., 177.,
        71.,  47., 187., 125.,  78.,  51., 258., 215., 303., 243.,  91.,
       150., 310., 153., 346.,  63.,  89.,  50.,  39., 103., 308., 116.,
       145.,  74.,  45., 115., 264.,  87., 202., 127., 182., 241.,  66.,
        94., 283.,  64., 102., 200., 265.,  94., 230., 181., 156., 233.,
        60., 219.,  80.,  68., 332., 248.,  84., 200.,  55.,  85.,  89.,
        31., 129.,  83., 275.,  65., 198., 236., 253., 124.,  44., 172.,
       114., 142., 109., 180., 144., 163., 147.,  97., 220., 190., 109.,
       191., 122., 230., 242., 248., 249., 192., 131., 237.,  78., 135.,
       244., 199., 270., 164.,  72.,  96., 306.,  91., 214.,  95., 216.,
       263., 178., 113., 200., 139., 139.,  88., 148.,  88., 243.,  71.,
        77., 109., 272.,  60.,  54., 221.,  90., 311., 281., 182., 321.,
        58., 262., 206., 233., 242., 123., 167.,  63., 197.,  71., 168.,
       140., 217., 121., 235., 245.,  40.,  52., 104., 132.,  88.,  69.,
       219.,  72., 201., 110.,  51., 277.,  63., 118.,  69., 273., 258.,
        43., 198., 242., 232., 175.,  93., 168., 275., 293., 281.,  72.,
       140., 189., 181., 209., 136., 261., 113., 131., 174., 257.,  55.,
        84.,  42., 146., 212., 233.,  91., 111., 152., 120.,  67., 310.,
        94., 183.,  66., 173.,  72.,  49.,  64.,  48., 178., 104., 132.,
       220.,  57.]),
 'frame': None,
 'DESCR': '.. _diabetes_dataset:\n\nDiabetes dataset\n----------------\n\nTen baseline variables, age, sex, body mass ind
ex, average blood\npressure, and six blood serum measurements were obtained for each of n =\n442 diabetes patients, as we
ll as the response of interest, a\nquantitative measure of disease progression one year after baseline.\n\n**Data Set Cha
racteristics:**\n\n  :Number of Instances: 442\n\n  :Number of Attributes: First 10 columns are numeric predictive values
\n\n  :Target: Column 11 is a quantitative measure of disease progression one year after baseline\n\n  :Attribute Informa
tion:\n      - age     age in years\n      - sex\n      - bmi     body mass index\n      - bp      average blood pressure
\n      - s1      tc, total serum cholesterol\n      - s2      ldl, low-density lipoproteins\n      - s3      hdl, high-d
ensity lipoproteins\n      - s4      tch, total cholesterol / HDL\n      - s5      ltg, possibly log of serum triglycerid
es level\n      - s6      glu, blood sugar level\n\nNote: Each of these 10 feature variables have been mean centered and 
scaled by the standard deviation times `n_samples` (i.e. the sum of squares of each column totals 1).\n\nSource URL:\nhtt
ps://www4.stat.ncsu.edu/~boos/var.select/diabetes.html\n\nFor more information see:\nBradley Efron, Trevor Hastie, Iain J
ohnstone and Robert Tibshirani (2004) "Least Angle Regression," Annals of Statistics (with discussion), 407-499.\n(http
s://web.stanford.edu/~hastie/Papers/LARS/LeastAngle_2002.pdf)',
 'feature_names': ['age',
  'sex',
  'bmi',
  'bp',
```

```
 's1',
 's2',
 's3',
 's4',
 's5',
 's6'],
'data_filename': 'c:\\Python39\\lib\\site-packages\\sklearn\\datasets\\data\\diabetes_data.csv.gz',
'target_filename': 'c:\\Python39\\lib\\site-packages\\sklearn\\datasets\\data\\diabetes_target.csv.gz'}
```

In [ ]:

```
diabetes_df = pd.DataFrame(diabetes.data)

diabetes_df
```

Out[ ]:

	0	1	2	3	4	5	6	7	8	9
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	-0.002592	0.019908	-0.017646
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	-0.039493	-0.068330	-0.092204
2	0.085299	0.050680	0.044451	-0.005671	-0.045599	-0.034194	-0.032356	-0.002592	0.002864	-0.025930
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	0.034309	0.022692	-0.009362
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	-0.002592	-0.031991	-0.046641
...	...	...	...	...	...	...	...	...	...	...
437	0.041708	0.050680	0.019662	0.059744	-0.005697	-0.002566	-0.028674	-0.002592	0.031193	0.007207
438	-0.005515	0.050680	-0.015906	-0.067642	0.049341	0.079165	-0.028674	0.034309	-0.018118	0.044485
439	0.041708	0.050680	-0.015906	0.017282	-0.037344	-0.013840	-0.024993	-0.011080	-0.046879	0.015491
440	-0.045472	-0.044642	0.039062	0.001215	0.016318	0.015283	-0.028674	0.026560	0.044528	-0.025930
441	-0.045472	-0.044642	-0.073030	-0.081414	0.083740	0.027809	0.173816	-0.039493	-0.004220	0.003064

442 rows × 10 columns

In [ ]:

```
X_train, X_test, y_train, y_test = train_test_split(diabetes.data, diabetes.target, random_state=2, test_size=0.25)
```

$\lambda = 1$

In [ ]:

```
ENreg = ElasticNet(alpha=1,l1_ratio=0.9)
```

```
ENreg.fit(X_train,y_train)
y_pred = reg.predict(X_test)

print(f"R2 Score : {r2_score(y_test,y_pred)*100}")
```

```
R2 Score : 2.5841896057826497
```

$\lambda = 0.1$

In [ ]:

```
ENreg01 = ElasticNet(alpha=0.1,l1_ratio=0.9)

ENreg01.fit(X_train,y_train)
y_pred = ENreg01.predict(X_test)

print(f"R2 Score : {r2_score(y_test,y_pred)*100}")
```

```
R2 Score : 26.47045915740639
```

$\lambda = 0.01$

In [ ]:

```
ENreg001 = ElasticNet(alpha=0.01,l1_ratio=0.9)

ENreg001.fit(X_train,y_train)
y_pred = ENreg001.predict(X_test)

print(f"R2 Score : {r2_score(y_test,y_pred)*100}")
```

```
R2 Score : 43.545036098194025
```

$\lambda = 0.001$

In [ ]:

```
ENreg0001 = ElasticNet(alpha=0.001,l1_ratio=0.9)

ENreg0001.fit(X_train,y_train)
y_pred = ENreg0001.predict(X_test)

print(f"R2 Score : {r2_score(y_test,y_pred)*100}")
```

```
R2 Score : 44.73503374737765
```

Gradient Descent Method :

This is a method rather algorithm to minimize the cost/loss function.

Gradient descent (GD) is an iterative first-order optimisation algorithm used to find a local minimum/maximum of a given function. This method is commonly used in machine learning (ML) and deep learning(DL) to minimise a cost/loss function.

In this method, we start with a cost function $J(\beta)$, and reach the minimum value by decreasing the value of β.

But in this case if the number of features are more then the surface becomes a hyperplane (higher order) and the minimum cannot be found so easily.

$$\nabla J(\beta_0, \beta_1, \ldots, \beta_n) =< \frac{\delta J}{\delta \beta_0} + \frac{\delta J}{\delta \beta_1} + \ldots + \frac{\delta J}{\delta \beta_n} >$$

Let us now train our linear model and minimize its cost using Gradient Descent Method

In [ ]:

```
# We'll again use the extended Boston dataset for training and testing this model

from sklearn.linear_model import SGDRegressor

X, y = mglearn.datasets.load_extended_boston()
```

In [ ]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0, test_size=0.25)
```

$\lambda = 0.1$

In [ ]:

```
sgdReg01 = SGDRegressor(loss='squared_loss', alpha=0.1, penalty='l2').fit(X_train, y_train)

print(f"Training set score : {sgdReg01.score(X_train, y_train)*100}\nTesting set score: {sgdReg01.score(X_test, y_test)*1
```

```
Training set score : 69.23845963732708
Testing set score : 52.95276418208543
```

$\lambda = 0.01$

In [ ]:

```
sgdReg001 = SGDRegressor(loss='squared_loss', alpha=0.01, penalty='l2').fit(X_train, y_train)

print(f"Training set score : {sgdReg001.score(X_train, y_train)*100}\nTesting set score: {sgdReg001.score(X_test, y_test)
```

```
Training set score : 81.07375579725604
Testing set score: 66.75723670824871
```

$\lambda = 0.001$

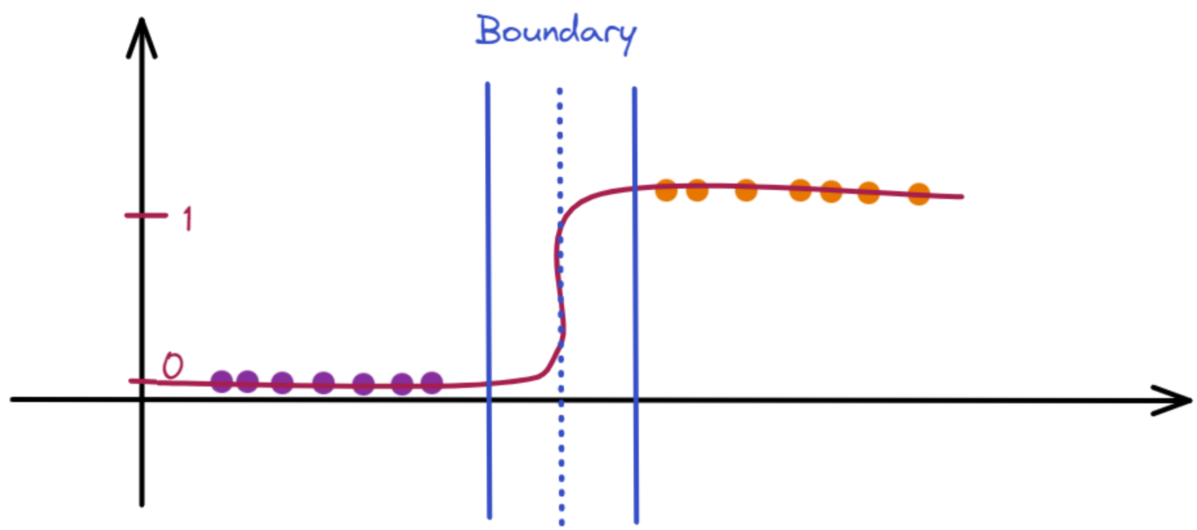
In [ ]:

```
sgdReg0001 = SGDRegressor(loss='squared_loss', alpha=0.001, penalty='l2').fit(X_train, y_train)

print(f"Training set score : {sgdReg0001.score(X_train, y_train)*100}\nTesting set score: {sgdReg0001.score(X_test, y_tes
```

```
Training set score : 83.10410859190111
Testing set score: 69.76460812770004
```

Logistic Regression :

Linear models are also extensively used for classification. Classification by logistic regression is pretty simple and straightforward.



Logistic Function is of the form :-

$f(x) = \frac{1}{1+e^{-x}}$

x is then replaced with the linear regression curve for the data and the classifier function alongwith the decision boundary is obtained.

$f(x) = \frac{1}{1+e^{-(a+bx)}}$, where $a + bx$ is the regression curve.

Lets now create our own logistic regression model.

In [ ]:

```
# As always we will first import our libraries and datasets
```

```python
from sklearn.linear_model import LogisticRegression

insurance_df = pd.read_csv('../Assets/insurance_data.csv')

insurance_df
```

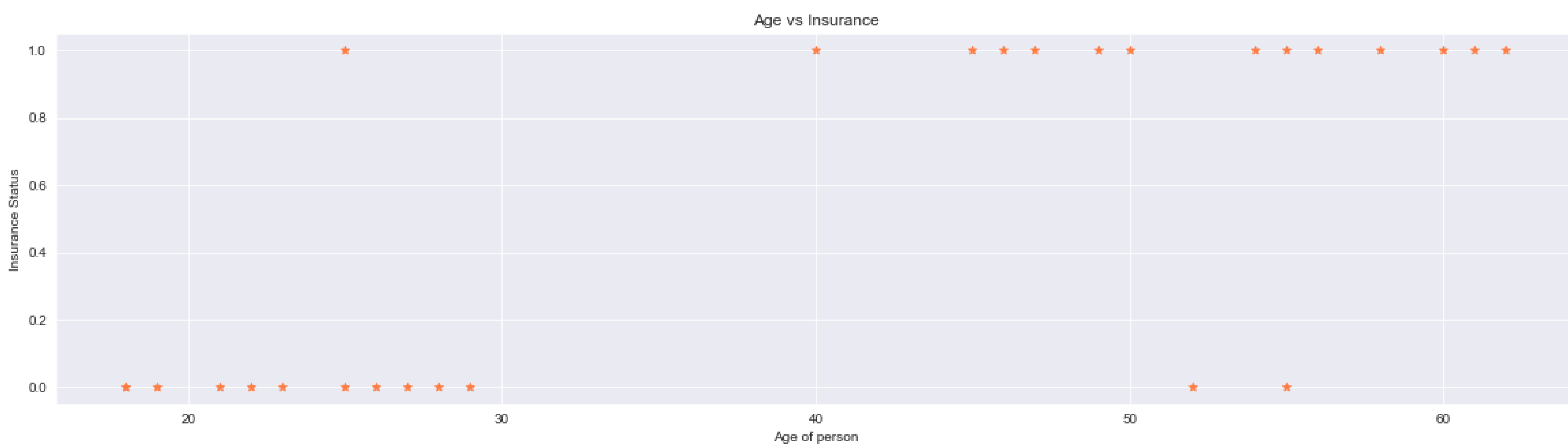
Out[ ]:

	age	bought_insurance
0	22	0
1	25	0
2	47	1
3	52	0
4	46	1
5	56	1
6	55	0
7	60	1
8	62	1
9	61	1
10	18	0
11	28	0
12	27	0
13	29	0
14	49	1
15	55	1
16	25	1
17	58	1
18	19	0
19	18	0
20	21	0

	age	bought_insurance
21	26	0
22	40	1
23	45	1
24	50	1
25	54	1
26	23	0

In [ ]:

```
plt.figure(figsize=(20,5))

plt.scatter(insurance_df['age'], insurance_df['bought_insurance'], color=colors[6], marker='*')

plt.xlabel("Age of person")
plt.ylabel('Insurance Status')
plt.title('Age vs Insurance')

plt.show()
```



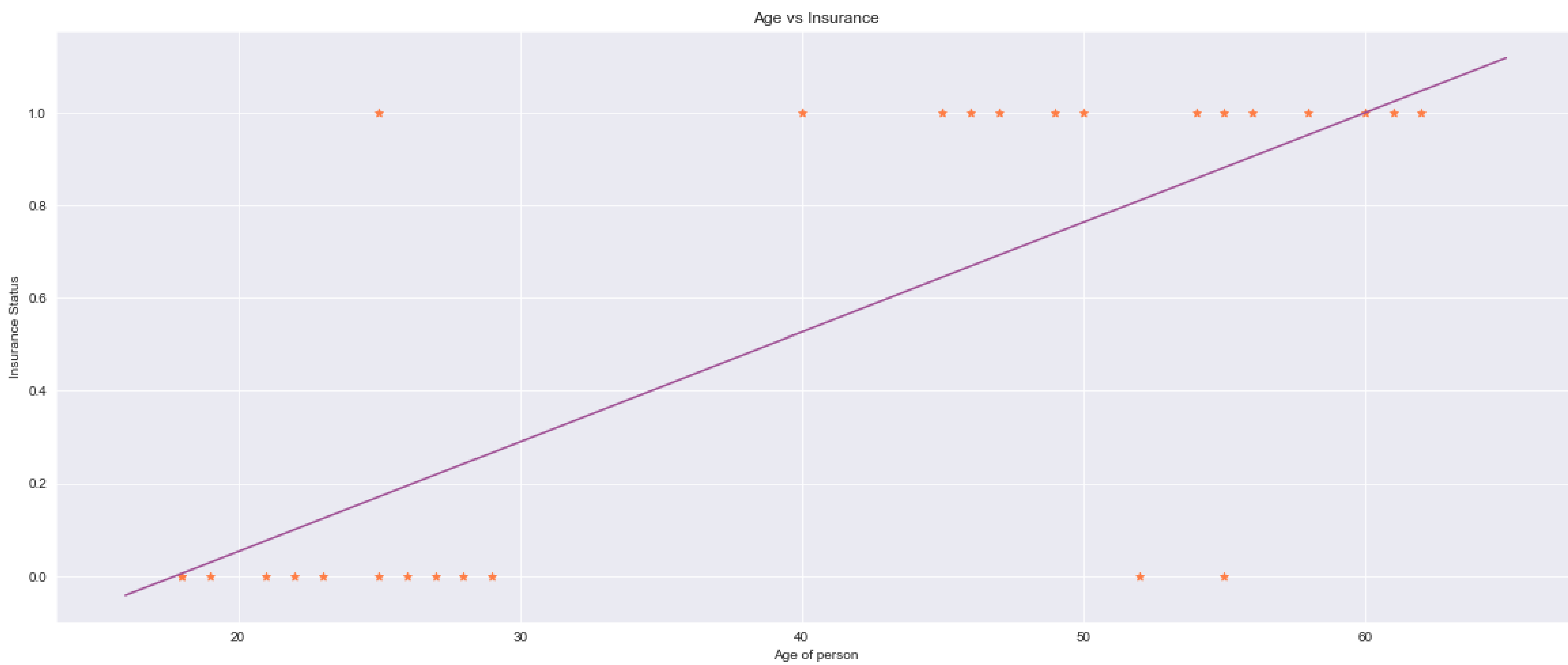
In [ ]:

```
insReg = LinearRegression()
insReg.fit(insurance_df[['age']], insurance_df['bought_insurance'])

print(f'Coefficient : {insReg.coef_[0]}\nIntercept : {insReg.intercept_}')
```

```
Coefficient : 0.023683938359706273
Intercept : -0.4209443697498303
```

In [ ]:

```python
x = np.linspace(16, 65, 2)
y = insReg.coef_*x + insReg.intercept_


plt.figure(figsize=(20, 8))

plt.scatter(insurance_df['age'], insurance_df['bought_insurance'], color=colors[6], marker='*')
plt.plot(x, y, color=colors[3])

plt.xlabel("Age of person")
plt.ylabel('Insurance Status')
plt.title('Age vs Insurance')

plt.show()
```



In [ ]:

```python
X_train, X_test, y_train, y_test = train_test_split(insurance_df[['age']], insurance_df['bought_insurance'], test_size=0.
```

In [ ]:

```
logReg = LogisticRegression().fit(X_train, y_train)

print(f'Coefficient : {logReg.coef_[0]}\nIntercept : {logReg.intercept_[0]}')
```

```
Coefficient : [0.11216016]
Intercept : -4.609606842692203
```

In [ ]:

```
# Lets check the accuracy of our model

print(f'Model Accuracy : {logReg.score(X_test, y_test)*100}')
```

```
Model Accuracy : 100.0
```

Daammmmnnn!!! 🦫🦫

In [ ]:

```
# Lets predict some values now

age = 43

y_prediction = logReg.predict([[age]])

print(f'Insurance status for person aged {age} : {y_prediction}')
```

```
Insurance status for person aged 43 : [1]
```

In [ ]: