THE MACHINE LEARNING JOURNEY

CHAPTER-04

ONE HOT ENCODING

Machine Learning Models only understand numeric data. So what do we do when we have textual data?

→ We convert them into numerical data.

To do this, there are 3 ways →

- Label Encoding.
- Dummy Variables.
- One Hot Encoding.

@Shreyan

Label Encoding →

What is it?

↳ A process of converting categorical text data into numerical form.

This is essential because most ML algorithms can only handle numerical values.

How does it work?

↳ Each unique category (label) in the data is assigned a numerical value.

Use cases:

→ for preprocessing to prepare data for algorithms that require numerical input.

→ eg. regression, SVM, K-NN.

Main disadvantage:

→ might introduce a problem called ordinality.

→ this means the model might misunderstand the data to be in some kind of order (0<1<2) which isn't the case with categorical data.

Alternative to beat this:

One Hot Encoding (OHE)

Dummy Variables:

@Shreyan

What is it?

It is an intermediate variable by which the categorical (textual) data can be represented.

How does it work?

Suppose we have a categorical variable like "Season" with 3 categories — Summer, Monsoon, Winter, we can represent it as →

	Summer	Monsoon	Winter
Summer	1	0	0
Monsoon	0	1	0
Winter	0	0	0

So, when Summer is 1, it is summer and not monsoon/winter, so we can represent it as (1,0,0).

The Dummy Variable TRAP!

- Avoid/omit one variable to avoid multicollinearity, a scenario where one variable can be predicted from the others.

  eg. if we drop Winter column, we can still understand the rest.

  i.e – (1,0) → this is summer.
  (0,1) → this is monsoon.
  (0,0) → this indicates winter

Encode categorical variables using Dummy Variables Method

In [1]:
```
import pandas as pd
from sklearn.linear_model import LinearRegression

import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
df = pd.read_csv("homeprices.csv")
df
```

Out[2]:

	town	area	price
0	Kolkata	2600	550000
1	Kolkata	3000	565000
2	Kolkata	3200	610000
3	Kolkata	3600	680000
4	Kolkata	4000	725000
5	Hyderabad	2600	585000
6	Hyderabad	2800	615000
7	Hyderabad	3300	650000
8	Hyderabad	3600	710000
9	Bangalore	2600	575000
10	Bangalore	2900	600000
11	Bangalore	3100	620000
12	Bangalore	3600	695000

We now convert these categorical columns into numeric type using get_dummies() method

In [3]:
```
#create dummy variables
dummies = pd.get_dummies(df.town)
dummies
```

Out[3]:

	Bangalore	Hyderabad	Kolkata
0	False	False	True
1	False	False	True
2	False	False	True
3	False	False	True
4	False	False	True
5	False	True	False
6	False	True	False
7	False	True	False
8	False	True	False
9	True	False	False
10	True	False	False
11	True	False	False
12	True	False	False

Each row represents a house from your dataset. The columns Bangalore, Hyderabad, and Kolkata are the dummy variables. 'True' in any of these columns indicates that the house is located in that town, while 'False' means it's not in that town. For example, the first row (index 0) has a 'True' in the Kolkata column, indicating that this house is in Kolkata.

But let us get the binary form instead of boolean values so we do -

In [4]:
```
dummies = dummies.astype(int)
dummies
```

Out[4]:

	Bangalore	Hyderabad	Kolkata
0	0	0	1
1	0	0	1
2	0	0	1
3	0	0	1
4	0	0	1
5	0	1	0
6	0	1	0
7	0	1	0
8	0	1	0
9	1	0	0
10	1	0	0
11	1	0	0
12	1	0	0

you can drop any columns you want

To avoid dummy variable trap, let's remove 'Hyderabad'

In [5]:
```
dummies = dummies.drop(['Hyderabad'], axis='columns')
dummies
```

Out[5]:

	Bangalore	Kolkata
0	0	1
1	0	1
2	0	1
3	0	1
4	0	1
5	0	0
6	0	0
7	0	0
8	0	0
9	1	0
10	1	0
11	1	0
12	1	0

Now, the dataset contains only two dummy variables: Bangalore and Kolkata.

or 0

The presence of a house in Hyderabad is implicitly indicated by 'False' in both these columns. This approach helps to avoid the dummy variable trap, which can cause multicollinearity in the model.

In [6]:
```
merged = pd.concat([df,dummies], axis='columns')
merged
```

Out[6]:

	town	area	price	Bangalore	Kolkata
0	Kolkata	2600	550000	0	1
1	Kolkata	3000	565000	0	1
2	Kolkata	3200	610000	0	1
3	Kolkata	3600	680000	0	1
4	Kolkata	4000	725000	0	1
5	Hyderabad	2600	585000	0	0
6	Hyderabad	2800	615000	0	0
7	Hyderabad	3300	650000	0	0
8	Hyderabad	3600	710000	0	0
9	Bangalore	2600	575000	1	0
10	Bangalore	2900	600000	1	0
11	Bangalore	3100	620000	1	0
12	Bangalore	3600	695000	1	0

In this merged DataFrame, the columns Bangalore and Kolkata are the dummy variables representing the towns. A value of 1 indicates the house is in that town, and a 0 indicates it is not. Note that Hyderabad is represented implicitly (when both Bangalore and Kolkata are 0). This DataFrame is now ready for use in machine learning models, where the town names have been effectively encoded.

We do not require 'town' variable as it is replaced by dummy variables, hence we drop 'town'

In [7]:
```
final = merged.drop(['town'], axis='columns')
final
```

Out[7]:

	area	price	Bangalore	Kolkata
0	2600	550000	0	1
1	3000	565000	0	1
2	3200	610000	0	1
3	3600	680000	0	1
4	4000	725000	0	1
5	2600	585000	0	0
6	2800	615000	0	0
7	3300	650000	0	0
8	3600	710000	0	0
9	2600	575000	1	0
10	2900	600000	1	0
11	3100	620000	1	0
12	3600	695000	1	0

'Price' is the target column to be predicted, so we drop it.

In [8]:
```
x = final.drop(['price'],axis='columns')
x
```

Out[8]:

	area	Bangalore	Kolkata
0	2600	0	1
1	3000	0	1
2	3200	0	1
3	3600	0	1
4	4000	0	1
5	2600	0	0
6	2800	0	0
7	3300	0	0
8	3600	0	0
9	2600	1	0
10	2900	1	0
11	3100	1	0
12	3600	1	0

@Shreyan

In [9]:
```
y = final['price']
y
```

← we store the price column in a separate variable.

Out[9]:
```
0     550000
1     565000
2     610000
3     680000
4     725000
5     585000
6     615000
7     650000
8     710000
9     575000
10    600000
11    620000
12    695000
Name: price, dtype: int64
```

Internal Handling in Linear Regression Models:

Some linear regression implementations, such as those in certain machine learning libraries, are designed to detect and handle multicollinearity internally. They can automatically exclude one of the dummy variables to avoid the dummy variable trap, ensuring that the model does not suffer from multicollinearity issues. This means that even if you include all dummy variables (one for each category of the categorical variable) in your model, the algorithm might implicitly drop one of them during the training process. It's a form of built-in safeguard against multicollinearity that simplifies model specification, as you don't need to manually drop one of the dummy variables.

Implication:

You can provide all dummy variables to the model without manually omitting one to prevent multicollinearity. The model will adjust itself by internally excluding one dummy variable. However, it's always good practice to understand the data you're working with and manually handle dummy variables, as this practice can provide more control over the model and ensure clarity in your model-building process.

```
In [14]: model = LinearRegression()
         #Train the model
         model.fit(x,y)
```

→ Apply Linear Regression and train the model

```
Out[14]: LinearRegression
         LinearRegression()
```

@Shreyan

Predict the price of house with 2800sqft area located at Bangalore

Parameters: [Area, Kolkata, Bangalore] ← this is what 'x' contains

```
In [11]: model.predict([[2800,0,1]])
Out[11]: array([565089.22812299])
```

= Rs. 565089

Predict the price of house with 3400sqft area at Hyderabad

```
In [12]: model.predict([[3400,0,0]])
Out[12]: array([681241.66845839])
```

= Rs. 681241

Finding the accuracy of the model

```
In [13]: model.score(x,y)
Out[13]: 0.9573929037221873
```

= 95.74%.

One-Hot-Encoding:

A technique used to convert categorical variables into a form that helps in better predictions for a ML model.

It creates new binary columns for each category in the original data.

eg. If we have a feature "Color" with 3 categories — 'Red', 'Blue', 'Green'.

OHE will create 3 new features —

Color_Red, Color_Blue, Color_Green.

So, if a particular observation is Red, it will be represented as [1, 0, 0].

Here is where OHE supercedes Label Encoding, i.e - LE might assign 'Red' as 1, 'Blue' as 2, 'Green' as 3.

This can lead to confusion and misinterpretation by models as having an ordinal relationship, as 3 > 2 > 1, so Green > Blue > Red.

@Shreyan

OHE - Behind The Scenes (#BTS)

The term "**one-hot**" in "one-hot encoding" originates from the domain of digital circuit design. In this context, "one-hot" refers to a group of bits among which the only valid combinations of values are those w unith a single high (1) bit and all the others low (0). Essentially, only one bit is "hot" (active or set to 1) at a time.

In the context of one-hot encoding for categorical data in machine learning:

Representation: The technique creates a binary column for each category and marks the column as '1' for the row where the category is present, and '0' for rows where it's not. This mirrors the digital circuit's one-hot concept, where only one bit is active at a time.

Example: Suppose you have a categorical feature with three categories: A, B, and C. One-hot encoding this feature would create three new binary features (columns). For a row where the original feature's value is A, the encoded pattern would be 1, 0, 0. Here, only the first bit (or feature column) is "hot" (set to 1).

Encode categorical variables using One Hot Encoding Method

In [1]:
```
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.linear_model import LinearRegression

import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
df = pd.read_csv("homeprices.csv")
df
```

Out[2]:

	town	area	price
0	Kolkata	2600	550000
1	Kolkata	3000	565000
2	Kolkata	3200	610000
3	Kolkata	3600	680000
4	Kolkata	4000	725000
5	Hyderabad	2600	585000
6	Hyderabad	2800	615000
7	Hyderabad	3300	650000
8	Hyderabad	3600	710000
9	Bangalore	2600	575000
10	Bangalore	2900	600000
11	Bangalore	3100	620000
12	Bangalore	3600	695000

this is generally not required, but let us use it for now.

To use OHE, let us first use Label Encoding -

In [3]:
```
le = LabelEncoder()
```

In [4]:
```
df.town = le.fit_transform(df.town)
df
```

Out[4]:

	town	area	price
0	2	2600	550000
1	2	3000	565000
2	2	3200	610000
3	2	3600	680000
4	2	4000	725000
5	1	2600	585000
6	1	2800	615000
7	1	3300	650000
8	1	3600	710000
9	0	2600	575000
10	0	2900	600000
11	0	3100	620000
12	0	3600	695000

Kolkata – 2
Hyderabad – 1
Bangalore – 0.

The town column in the DataFrame has been successfully encoded using Label Encoding. The towns have been converted to numerical values:

Kolkata is encoded as **2** Hyderabad is encoded as **1** Bangalore is encoded as **0**

The **LabelEncoder** in scikit-learn assigns labels based on the **alphabetical order** of the categories by default. It doesn't assign labels based on the **order in which categories appear** in the data or any other criterias. Here's how the labeling works in our case:

We have three towns: Bangalore, Hyderabad, and Kolkata. Alphabetically, "Bangalore" comes first, so it gets labeled as 0."Hyderabad" comes next, so it gets labeled as 1. "Kolkata" is last alphabetically, so it gets labeled as 2.

In [5]:
```
#retreive training data
x = df[['town', 'area']]
x
```

'x' is the training data here.

Out[5]:

	town	area
0	2	2600
1	2	3000
2	2	3200
3	2	3600
4	2	4000
5	1	2600
6	1	2800
7	1	3300
8	1	3600
9	0	2600
10	0	2900
11	0	3100
12	0	3600

@Shreyan

In [6]:
```
#retreive target data
y= df.price
y
```

'y' is the target data.

Out[6]:
```
0     550000
1     565000
2     610000
3     680000
4     725000
5     585000
6     615000
7     650000
8     710000
9     575000
10    600000
11    620000
12    695000
Name: price, dtype: int64
```

IMP→ **The OneHotEncoder instance has to be created with the *handle_unknown='ignore'* option. This option ensures that if the encoder encounters a category in the test data that wasn't present in the training data, it will ignore it rather than throwing an error.**

In [7]:
```
ohe = OneHotEncoder(handle_unknown='ignore')
```

In [8]:
```
x1 = ohe.fit_transform(df[['town']])
x1
```

Out[8]:
```
<13x3 sparse matrix of type '<class 'numpy.float64'>'
        with 13 stored elements in Compressed Sparse Row format>
```

1. ohe.fit_transform(df[['town']]): This part of the code uses the OneHotEncoder object ohe to both fit and transform the town column of your DataFrame.

2. fit part of fit_transform learns how many unique values are in the town column and assigns a new binary column to each unique value. In this case, it identifies three unique towns: Kolkata, Hyderabad, and Bangalore. transform part of fit_transform then takes each row in the town column and encodes it as a binary array where only the column corresponding to the town's value is 1, and all others are 0.

The result of this operation is a sparse matrix which efficiently represents a large matrix with mostly 0 values.

In [9]:
```
x1 = pd.DataFrame(x1.toarray())
x1
```

Out[9]:

	0	1	2
0	0.0	0.0	1.0
1	0.0	0.0	1.0
2	0.0	0.0	1.0
3	0.0	0.0	1.0
4	0.0	0.0	1.0
5	0.0	1.0	0.0
6	0.0	1.0	0.0
7	0.0	1.0	0.0
8	0.0	1.0	0.0
9	1.0	0.0	0.0
10	1.0	0.0	0.0
11	1.0	0.0	0.0
12	1.0	0.0	0.0

x1.toarray(): This converts the sparse matrix obtained from the OneHotEncoder into a regular (dense) numpy array. This array now represents the one-hot encoded form of the town column.

pd.DataFrame(...): This converts the numpy array into a pandas DataFrame, making it easier to view and manipulate. Each column in this DataFrame corresponds to one unique value in the original town column, with rows representing the one-hot encoded binary values.

In [10]:
```
#to avoid dummy variable trap, drop 0th column
x1 = x1.iloc[:,1:]
x1
```

Out[10]:

	1	2
0	0.0	1.0
1	0.0	1.0
2	0.0	1.0
3	0.0	1.0
4	0.0	1.0
5	1.0	0.0
6	1.0	0.0
7	1.0	0.0
8	1.0	0.0
9	0.0	0.0
10	0.0	0.0
11	0.0	0.0
12	0.0	0.0

In this above DataFrame:

The column labeled **1 represents Hyderabad.** The column labeled **2 represents Kolkata.**

The absence of both (0 in both columns) implicitly indicates Bangalore, the category represented by the dropped column. This approach is used to prevent multicollinearity in the model.

In [11]:
```
#add these columns to x
x = pd.concat([x,x1], axis="columns")
x
```

Out[11]:

	town	area	1	2
0	2	2600	0.0	1.0
1	2	3000	0.0	1.0
2	2	3200	0.0	1.0
3	2	3600	0.0	1.0
4	2	4000	0.0	1.0
5	1	2600	1.0	0.0
6	1	2800	1.0	0.0
7	1	3300	1.0	0.0
8	1	3600	1.0	0.0
9	0	2600	0.0	0.0
10	0	2900	0.0	0.0
11	0	3100	0.0	0.0
12	0	3600	0.0	0.0

In this DataFrame:

Area represents the area of the houses. Hyderabad and Kolkata are the one-hot encoded columns for the towns.

A 1.0 indicates the presence of the house in that town, and 0.0 indicates its absence.

In [12]:
```
x.drop('town', axis=1, inplace=True)
x
```

Out[12]:

	area	1	2
0	2600	0.0	1.0
1	3000	0.0	1.0
2	3200	0.0	1.0
3	3600	0.0	1.0
4	4000	0.0	1.0
5	2600	1.0	0.0
6	2800	1.0	0.0
7	3300	1.0	0.0
8	3600	1.0	0.0
9	2600	0.0	0.0
10	2900	0.0	0.0
11	3100	0.0	0.0
12	3600	0.0	0.0

Convert all column names in 'x' to strings

Error Handling →

You may encounter a TypeError when trying to fit the linear regression model using the sklearn library.The error message will suggest that the feature names in your DataFrame x are of mixed types (some are strings and some are integers).The sklearn library requires all feature names to be of the same type, preferably strings, for compatibility.

In [13]:
```
x.columns = x.columns.astype(str)
x
```

Out[13]:

	area	1	2
0	2600	0.0	1.0
1	3000	0.0	1.0
2	3200	0.0	1.0
3	3600	0.0	1.0
4	4000	0.0	1.0
5	2600	1.0	0.0
6	2800	1.0	0.0
7	3300	1.0	0.0
8	3600	1.0	0.0
9	2600	0.0	0.0
10	2900	0.0	0.0
11	3100	0.0	0.0
12	3600	0.0	0.0

@Shreyan

In [14]:
```
model = LinearRegression()
model.fit(x,y) #train the model
```

Out[14]:
```
LinearRegression
LinearRegression()
```

Predict the price of house with 2800sqft area located at Kolkata

In [15]:
```
model.predict([[2800,0,1]])
```

Out[15]:
```
array([565089.22812299])
```

Predict the price of house with 3400sqft area located at Bangalore

In [16]:
```
model.predict([[3400,0,0]])
```

Out[16]:
```
array([666914.10449365])
```

Accuracy score is

In [17]:
```
model.score(x,y)
```

Out[17]:
```
0.9573929037221872
```
→ 95.74%.

Let us now check out a problem →

Problem :

'carprices.csv' contains car sale prices for 3 different models. First, plot data points on a regression plot to see if Linear Regression Model can be applied. Then build a model that can answer the following questions:

1. Predict price of a Mercedes Benz that is **4yrs old with mileage 45000.**
2. Predict price of a BMW X5 that is **7 years old with mileage 86000.**
3. What is the accuracy of your model ?

In [1]:
```
import pandas as pd
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')

from sklearn.preprocessing import OneHotEncoder
from sklearn.linear_model import LinearRegression
```

→ We will do this without Label Encoding.

In [2]:
```
df = pd.read_csv("carprices.csv")
df
```

Out[2]:

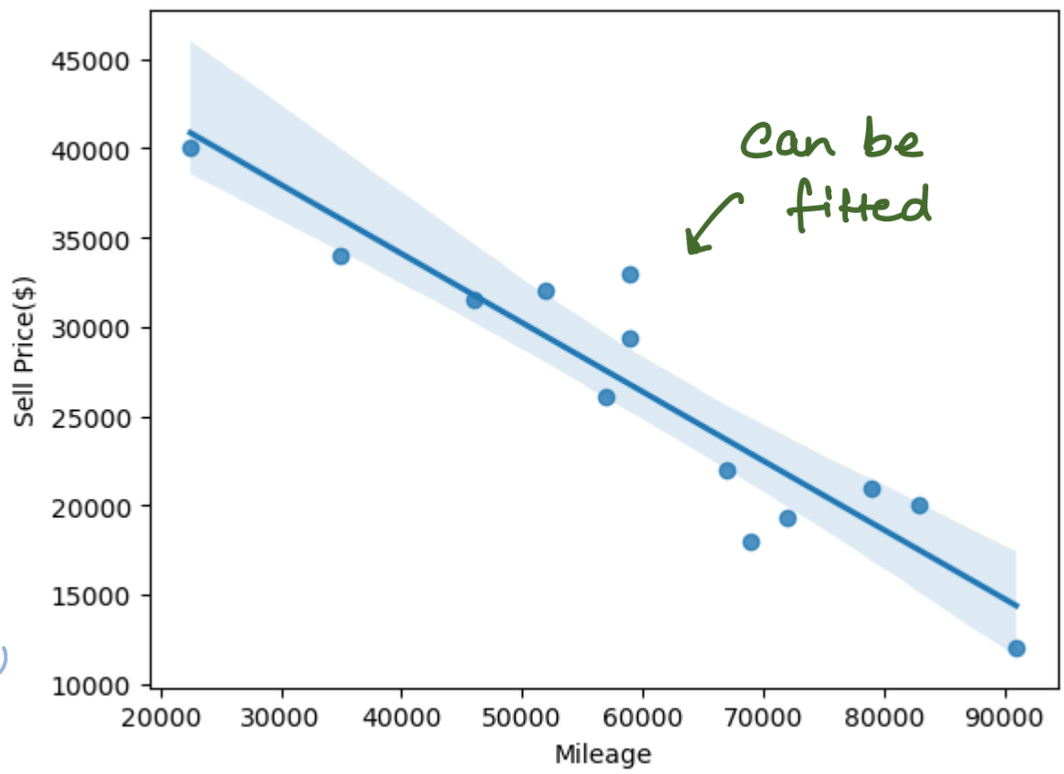
	Car Model	Mileage	Sell Price($)	Age(yrs)
0	BMW X5	69000	18000	6
1	BMW X5	35000	34000	3
2	BMW X5	57000	26100	5
3	BMW X5	22500	40000	2
4	BMW X5	46000	31500	4
5	Audi A5	59000	29400	5
6	Audi A5	52000	32000	5
7	Audi A5	72000	19300	6
8	Audi A5	91000	12000	8
9	Mercedes Benz C Class	67000	22000	6
10	Mercedes Benz C Class	83000	20000	7
11	Mercedes Benz C Class	79000	21000	7
12	Mercedes Benz C Class	59000	33000	5

Let us check using seaborn plots **if we can train this model using Linear Regression**

@Shreyan

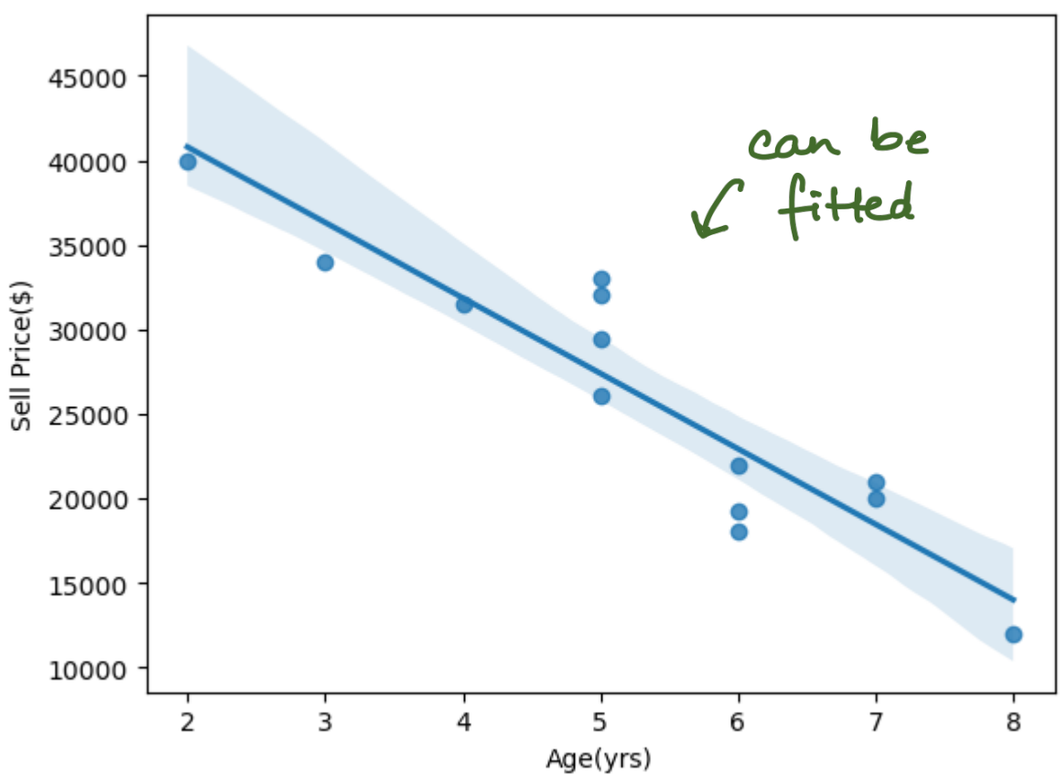
In [3]:
```
sns.regplot(data=df, x='Mileage', y='Sell Price($)')
```

Out[3]:
```
<Axes: xlabel='Mileage', ylabel='Sell Price($)'>
```



In [4]:
```
sns.regplot(data=df, x='Age(yrs)', y='Sell Price($)')
```

Out[4]:
```
<Axes: xlabel='Age(yrs)', ylabel='Sell Price($)'>
```



In [5]:
```
ohe = OneHotEncoder()
x = ohe.fit_transform(df[['Car Model']])
x
```

Out[5]:
```
<13x3 sparse matrix of type '<class 'numpy.float64'>'
        with 13 stored elements in Compressed Sparse Row format>
```

In [6]:
```
x=pd.DataFrame(x.toarray())
x
```

Out[6]:

	0	1	2
0	0.0	1.0	0.0
1	0.0	1.0	0.0
2	0.0	1.0	0.0
3	0.0	1.0	0.0
4	0.0	1.0	0.0
5	1.0	0.0	0.0
6	1.0	0.0	0.0
7	1.0	0.0	0.0
8	1.0	0.0	0.0
9	0.0	0.0	1.0
10	0.0	0.0	1.0
11	0.0	0.0	1.0
12	0.0	0.0	1.0

Now since we can drop a column to avoid dummy variable trap, so —

In [7]:
```
#to avoid dummy variable trap
x = x.iloc[:,1:]
x
```

we have dropped the first column.

Out[7]:

	1	2
0	1.0	0.0
1	1.0	0.0
2	1.0	0.0
3	1.0	0.0
4	1.0	0.0
5	0.0	0.0
6	0.0	0.0
7	0.0	0.0
8	0.0	0.0
9	0.0	1.0
10	0.0	1.0
11	0.0	1.0
12	0.0	1.0

@Shreyan

In [8]:
```
x1 = df[['Mileage', 'Age(yrs)']]
x1
```

Out[8]:

	Mileage	Age(yrs)
0	69000	6
1	35000	3
2	57000	5
3	22500	2
4	46000	4
5	59000	5
6	52000	5
7	72000	6
8	91000	8
9	67000	6
10	83000	7
11	79000	7
12	59000	5

We concat the 2 columns — 'x' and 'x1'.

In [9]:
```
x1 = pd.concat([x1, x], axis="columns")
x1
```

Out[9]:

	Mileage	Age(yrs)	1	2
0	69000	6	1.0	0.0
1	35000	3	1.0	0.0
2	57000	5	1.0	0.0
3	22500	2	1.0	0.0
4	46000	4	1.0	0.0
5	59000	5	0.0	0.0
6	52000	5	0.0	0.0
7	72000	6	0.0	0.0
8	91000	8	0.0	0.0
9	67000	6	0.0	1.0
10	83000	7	0.0	1.0
11	79000	7	0.0	1.0
12	59000	5	0.0	1.0

@Shreyan

In [10]:
```
y = df['Sell Price($)']
y
```

← we store the price list in 'y'

Out[10]:
```
0     18000
1     34000
2     26100
3     40000
4     31500
5     29400
6     32000
7     19300
8     12000
9     22000
10    20000
11    21000
12    33000
Name: Sell Price($), dtype: int64
```

In [11]:
```
x1.columns = x1.columns.astype(str)
```
← to avoid mismatch error.

In [12]:
```
model = LinearRegression()
#train the model
model.fit(x1, y)
```

Out[12]:
```
▾ LinearRegression

LinearRegression()
```

Parameters for predict are - [[Mileage, Years, BMW, Mercedes, Audi]]

In [17]:
```
model.predict([[45000, 4, 0, 1]]) #for Mercedes
```
← So for 45000 kms, 4 yrs, Mercedes

Out[17]:
```
array([36991.31721061])
```

In [14]:
```
model.predict([[86000, 7, 1, 0]]) #for BMW
```
← for 86000, 7yrs, BMW

Out[14]:
```
array([11080.74313219])
```

In [15]:
```
model.score(x1,y)
```

Out[15]:
```
0.9417050937281082
```
→ 94.17%.

@Shreyan

SHREYAN BASU RAY

OPENSOURCE MAINTAINER @ SAGE.AI

SUPPORT ME → github.com/sponsors/Shreyan1