Encoding Categorical variables

Feature Engineering:
i. Feature Scaling
ii. Encoding Categorical Data

Data types:
(i) Nominal Data: Data in which there is no relationship or order in categories For ex: State (Maharashtra, Gujarat, Uttar Pradesh, Haryana, Rajasthan, Madhya Pradesh) Domain (Computer Science, Artificial Intelligence, Information Technology, Electronics)

(ii) Ordinal Data: Data in which there is relationship between categories. For ex: Graduate in High School Review (Excellent, Very Good, Good, Poor, Very Poor) this shows Excellent is at high level, Very Poor at low.

How to Encode Categorical Data? Mostly categorical data is in form of string and our ML algorithm expects numbers. As a Machine Learning Engineer, Data Scientist its our responsibilty to convert the category to numbers. There are lot of ways to convert categories to numbers, there are lot of techniques for encoding. Mainly: (i) Ordinal Encoding (ii) One-Hot Encoding (iii) Label Encoding

Label Encoding:

In ordinal encoding , in our data there is/are input columns X and output columns y

```
+---------+---------+---------+---------+
|   X1    |   X2    |   X3    |    Y    |
-----------------------------------------
|         |         |         |         |
-----------------------------------------
|         |         |         |         |
-----------------------------------------
|         |         |         |         |
-----------------------------------------
```

In this input columns (X), whenever we have values in column which is Ordinal Encoding there.
But if output variable (Y) is Categorical (like Classification), (0/1) then we don't go for Ordinal Encoding.
Then what to use?
So in this scenario, we use Label Encoding.
It does the same thing which Ordinal Encoding does, this is applied/designed just for output variable 'Y'.

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

```
df = pd.read_csv('customer_dataset.csv')
```

```
df
```

```
df.head()
```

	age	gender	review	education	purchased
0	30	Female	Average	School	No
1	68	Female	Poor	UG	No
2	70	Female	Good	PG	No
3	72	Female	Good	PG	No
4	16	Female	Average	UG	No

```
df.tail()
```

	age	gender	review	education	purchased
45	61	Male	Poor	PG	Yes
46	64	Female	Poor	PG	No
47	38	Female	Good	PG	Yes
48	39	Female	Good	UG	Yes
49	25	Female	Good	UG	No

```
df.isna().sum()
```

```
age          0
gender       0
review       0
education    0
purchased    0
dtype: int64
```

```
df.describe()
```

	age
count	50.000000
mean	54.160000
std	25.658161
min	15.000000
25%	30.250000
50%	57.000000
75%	74.000000
max	98.000000

Ordinal Encoding:

For converting 'X' i.e independent variables into numbers we use Ordinal Encoding.
Ex:
Education
High School (HS)
High School (HS)
Post Graduate (PG)
Post Graduate (PG)
Under Graduate (UG)
UnderGraduate (UG)

Here, we can easily see the relationship:
PG > UG > HS, hence 2 > 1 > 0. So we encode PG as 2, UG as 1 and HS as 0.

As we can observe here,
'Gender' is categorical, 'Review' is categorical, 'Education' is categorical. Except 'Age' which is non-categorical.
Gender --> Nominal categorical
Review --> Ordinal categorical
Education --> Ordinal categorical
Purchased --> Nominal categorical
So, we can apply:
One Hot Encoder on 'Gender'
Ordinal Encoder on 'Review' and 'Education'
Label Encoder on 'Purchased'

Now the thing is as we will have to do different encoding techniques on different individual columns (One-Hot, Ordinal, Nominal) and then asemble the models/result.
So to prevent fro these kind of problems we can use Column Transformer technique which is available in sci-kit learn.
Then building pipelines for individual data columns through Column Transformer.
This all I will pperform later
But for now, I will ignore columns 'Age' and 'Gender'
So left columns will be 'Review', 'Education' and 'Purchased'

Because there are two input columns which shows relationship, 'review' and 'education' and the target variable is 'purchased', So we will select columns 'review', 'education' and 'purchased'

```
df = df[['review', 'education', 'purchased']]
```

```
df
```

	review	education	purchased
0	Average	School	No
1	Poor	UG	No
2	Good	PG	No
3	Good	PG	No
4	Average	UG	No
5	Average	School	Yes
6	Good	School	No
7	Poor	School	Yes
8	Average	UG	No
9	Good	UG	Yes
10	Good	UG	Yes
11	Good	UG	Yes
12	Poor	School	No
13	Average	School	No

Now, I will implement Train Test Split on dataset in ratio 80% and 20% i.e, test_size=0.2

X_train will be 'Review' and 'Education'

y_train will be 'Purchased'

```
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import OrdinalEncoder
```

```
X = df[['review', 'education']]
y = df['purchased']
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

```
X_train
```

	review	education
12	Poor	School
24	Average	PG
25	Good	School
19	Poor	PG
47	Good	PG
13	Average	School
41	Good	PG
43	Poor	PG
20	Average	School
30	Average	UG
44	Average	UG
27	Poor	PG
4	Average	UG
7	Poor	School
17	Poor	UG
0	Average	School
38	Good	School
6	Good	School
9	Good	UG
49	Good	UG
33	Good	PG
48	Good	UG
35	Poor	School
32	Average	UG
18	Good	School
42	Good	PG
40	Good	School
1	Poor	UG
14	Poor	PG
26	Poor	PG
45	Poor	PG

```
y_train
```

```
12     No
24    Yes
25     No
19    Yes
47    Yes
13     No
41    Yes
43     No
20    Yes
30     No
44     No
27     No
4      No
7     Yes
17    Yes
0      No
38     No
6      No
9     Yes
49     No
33    Yes
48    Yes
35    Yes
```

```
32    Yes
18     No
42    Yes
40     No
1      No
14    Yes
26     No
45    Yes
29    Yes
37    Yes
8      No
34     No
22    Yes
10    Yes
15     No
28     No
23     No
Name: purchased, dtype: object
```

```
X_test
```

	review	education
39	Poor	PG
21	Average	PG
36	Good	UG
11	Good	UG
46	Poor	PG
16	Poor	UG
2	Good	PG
31	Poor	School
5	Average	School
3	Good	PG

```
y_test
```

```
39     No
21     No
36    Yes
11    Yes
46     No
16    Yes
2      No
31    Yes
5     Yes
3      No
Name: purchased, dtype: object
```

Lets implement Ordinal Encoder on values of categorical values of X_train

```
OE_A = OrdinalEncoder()
```

```
OE_A.fit(X_train)
```

```
▾ OrdinalEncoder
OrdinalEncoder()
```

```
OE_A.transform(X_train)
```

```
array([[2., 1.],
       [0., 0.],
       [1., 1.],
       [2., 0.],
       [1., 0.],
       [0., 1.],
       [1., 0.],
       [2., 0.],
       [0., 1.],
       [0., 2.],
       [0., 2.],
       [2., 0.],
       [0., 2.],
       [2., 1.],
       [2., 2.],
       [0., 1.],
       [1., 1.],
       [1., 1.],
       [1., 2.],
       [1., 2.],
       [1., 0.],
       [1., 2.],
       [2., 1.],
       [0., 2.],
       [1., 1.],
       [1., 0.],
       [1., 1.],
       [2., 2.],
       [2., 0.],
       [2., 0.],
       [2., 0.],
       [0., 2.],
       [0., 0.],
       [0., 2.],
       [0., 1.],
       [2., 0.],
       [1., 2.],
       [2., 2.],
       [2., 1.],
       [1., 1.]])
```

```
OE_A.transform(X_test)
```

```
array([[2., 0.],
       [0., 0.],
       [1., 2.],
       [1., 2.],
       [2., 0.],
       [2., 2.],
       [1., 0.],
       [2., 1.],
       [0., 1.],
       [1., 0.]])
```

```
OE_A.categories_
```

```
[array(['Average', 'Good', 'Poor'], dtype=object),
 array(['PG', 'School', 'UG'], dtype=object)]
```

```
OE = OrdinalEncoder(categories=[['Poor', 'Average', 'Good'], ['School', 'UG', 'PG']])
```

In the above line code,
OE = OrdinalEncoder(categories=[['Poor', 'Average', 'Good'], ['School', 'UG', 'PG']]) If I didn't had declared categories=[['Poor', 'Average', 'Good'], ['School', 'UG', 'PG']]
then it would had randomly categorized the data. So we give this in order like Poor < Average < Good. Same for School < UG < PG
Here we declared the level of each value in order.

Now, I will fit model on X_train

```
OE.fit(X_train)
```

```
OrdinalEncoder
OrdinalEncoder(categories=[['Poor', 'Average', 'Good'], ['School', 'UG', 'PG']])
```

Now, as I fit the model on X_train, lets transform the values into categorical values/numbers

X_train = OE.transform(X_train)
X_test = OE.transform(X_test)

```
X_train = OE.transform(X_train)
X_test = OE.transform(X_test)
```

```
X_train
```

```
array([[0., 0.],
       [1., 2.],
       [2., 0.],
       [0., 2.],
       [2., 2.],
       [1., 0.],
       [2., 2.],
       [0., 2.],
       [1., 0.],
       [1., 1.],
       [1., 1.],
       [0., 2.],
       [1., 1.],
       [0., 0.],
       [0., 1.],
       [1., 0.],
       [2., 0.],
       [2., 0.],
       [2., 1.],
       [2., 1.],
       [2., 2.],
       [2., 1.],
       [0., 0.],
       [1., 1.],
       [2., 0.],
       [2., 2.],
       [2., 0.],
       [0., 1.],
       [0., 2.],
       [0., 2.],
       [0., 2.],
       [1., 1.],
       [1., 2.],
       [1., 1.],
       [1., 0.],
       [0., 2.],
       [2., 1.],
       [0., 1.],
       [0., 0.],
       [2., 0.]])
```

OE.categories --> Gives categories of X_train data

```
OE.categories
```

```
[['Poor', 'Average', 'Good'], ['School', 'UG', 'PG']]
```

OE.categories_ --> Gives categories along with their data type of X_train data

```
OE.categories_
```

```
[array(['Poor', 'Average', 'Good'], dtype=object),
 array(['School', 'UG', 'PG'], dtype=object)]
```

Now, Lets apply LabelEncoder on column 'Purchased' i.e, y variable.
(Label Encoding is specifically used for Nomminal output or y variable data)
Encode target labels with values between 0 and n-classes -1.
This transformer must be used to encode target values, i.e., only 'y' and not 'X'

```
from sklearn.preprocessing import LabelEncoder

LE = LabelEncoder()
```

Stored our model label encoder into variable LE

Then I will fit this transformation technique on y_train

```
LE.fit(y_train)
```

```
LabelEncoder
LabelEncoder()
```

```
LE.classes_
```

```
array(['No', 'Yes'], dtype=object)
```

Then apply transformation technique on y_train and y_test

```
y_train = LE.transform(y_train)
y_test = LE.transform(y_test)
```

```
y_train
```

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

```
y_test
```

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

###

ONE HOT ENCODING

This technique is used to handle Nominal categorical data (No intrinsic order)

Work Flow:
One Hot Encoding --> Dummy variable Trap --> One Hot Encoding using frequent variables --> Examples

```
+----------+----------+
|  Color   |  Target  |
+----------+----------+
|  Yellow  |    0     |
|  Yellow  |    1     |
|   Blue   |    1     |
|  Yellow  |    1     |
|    Red   |    1     |
|  Yellow  |    0     |
|    Red   |    1     |
|    Red   |    0     |
|  Yellow  |    1     |
|   Blue   |    0     |
+----------+----------+
```

We can observe we have 3 types of color categories Yellow, Blue and Red and there is no relationship / intrinsic order among them.
And hence when we convert these into number, we won't use Ordinal Encoding else our Machine Learning Algorithm will think that one individual color is more important than other individuals.
Hence I encode it as:

```
+----------+----------+----------+----------+
|  Yellow  |   Blue   |    Red   |  Target  |
+----------+----------+----------+----------+
|    1     |    0     |    0     |    0     |
|    1     |    0     |    0     |    1     |
|    0     |    1     |    0     |    1     |
|    1     |    0     |    0     |    1     |
|    0     |    0     |    1     |    1     |
|    1     |    0     |    0     |    0     |
|    0     |    0     |    1     |    1     |
|    0     |    0     |    1     |    0     |
|    1     |    0     |    0     |    1     |
|    0     |    1     |    0     |    0     |
+----------+----------+----------+----------+
```

Kinda we converted into vectors
As:
[1, 0, 0] --> Yellow
[0, 1, 0] --> Blue
[0, 0, 1] --> Red

Even if I have 50 different categories, I will use this method.
I know Dimensionality will increase, but this is the only way.

After this we remove one column among these independent variable columns.
Like if we have n number of columns, we keep n-1 columns.
The reason is Multicollinearity.

Multicolinearity:

Our input columns (independent variables). Is there any mathematical relation?
If yes, then these columns are dependent on each other and this should be avoided.
In Machine Learning our input columns 'X' must be inependent eith each other.

If we observe in our table each row's summation comes out to 1. This thing creates difficulty when we are working with linear models like:
Linear Regressi, Logistic Regression, etc.
So it must be always remember that columns must not be dependent with each other.
For this reason we only keep n-1 columns.
Ex:
Yellow --> [1, 0, 0]
Blue --> [0, 1, 0]
Red --> [0, 0, 1]
Now the Question comes to our mind is:
Even if we removed column 'Yellow', how will we get to know that the color was 'Yellow'?
The answer is if both 'Blue' and 'Red' are 0 i.e., False, it means color is 'Yellow'.

If we observe, we can easily see there is multicolinearity among columns 'Yellow', 'Blue' and 'Red'. That is why it is called as Dummy variable Trap.
To prevent this there is only way to remove one column.

If suppose we are working with Car Dataset.

```
car_dataset = pd.read_csv('cars_dataset.csv')
```

```
car_dataset.head()
```

	brand	km_driven	fuel	owner	selling_price
0	Maruti	145500	Diesel	First Owner	450000
1	Skoda	120000	Diesel	Second Owner	370000
2	Honda	140000	Petrol	Third Owner	158000
3	Hyundai	127000	Diesel	First Owner	225000
4	Maruti	120000	Petrol	First Owner	130000

```
car_dataset.isna().sum()
```

```
brand            0
km_driven        0
fuel             0
owner            0
selling_price    0
dtype: int64
```

So I can say its a good dataset as there is not a single null value.

'brand' --> Nominal Categorical (As many as brand individuals are, the more will be Dimensionality and processing gets slow)
So in this scenario we keep some frequent categories whhich are important and others we declare as others.
For ex:
category 1 is Hyundai
category 2 is Honda category 3 is Toyota category 4 is Tata category 5 is others

So in this others

```
          others
            |
------------------------------------------------------------------------
|            |            |            |            |            |
Audi         BMW          Jaguar       Mercedes     Skoda        Mahindra
```

```
car_dataset.value_counts()
```

```
brand   km_driven  fuel    owner         selling_price
Jaguar  45000      Diesel  First Owner   3200000          34
Lexus   20000      Petrol  First Owner   5150000          34
Toyota  68089      Petrol  First Owner   2000000          32
Honda   56494      Petrol  First Owner   550000           32
Toyota  79328      Diesel  Second Owner  750000           31
                                                          ..
Jeep    12000      Diesel  First Owner   1500000           1
        10000      Petrol  First Owner   1520000           1
Jaguar  70000      Diesel  Second Owner  3000000           1
        35000      Diesel  First Owner   3500000           1
Volvo   72500      Diesel  Second Owner  1200000           1
Length: 6450, dtype: int64
```

So in our dataset
car_dataset['brand'].value_counts()
This gives us the count for each values of each individual brand
car_dataset['brand'].nunique()
This gives us overall different categories of brand
Same for other columns.

One Hot Encoding using Pandas

```
pd.get_dummies(car_dataset,columns=['fuel','owner'])
```

	brand	km_driven	selling_price	fuel_CNG	fuel_Diesel	fuel_LPG	fuel_Petrol	owner_First Owner	owner_Fourth & Above Owner	owner_Second Owner	owner_Test Drive Car	owner_Third Owner
0	Maruti	145500	450000	0	1	0	0	1	0	0	0	0
1	Skoda	120000	370000	0	1	0	0	0	0	1	0	0
2	Honda	140000	158000	0	0	0	1	0	0	0	0	1
3	Hyundai	127000	225000	0	1	0	0	1	0	0	0	0
4	Maruti	120000	130000	0	0	0	1	1	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...
8123	Hyundai	110000	320000	0	0	0	1	1	0	0	0	0
8124	Hyundai	119000	135000	0	1	0	0	0	1	0	0	0
8125	Maruti	120000	382000	0	1	0	0	1	0	0	0	0
8126	Tata	25000	290000	0	1	0	0	1	0	0	0	0
8127	Tata	25000	290000	0	1	0	0	1	0	0	0	0

8128 rows × 12 columns

pd.get_dummies(car_dataset, columns=['fuel', 'owner'])
This is ultimately OHE but here we didn't applied it on column 'brand' for now as it will increase more number of columns and hence dimensionality will increase.

K-1 One Hot Encoding

pd.get_dummies(car_dataset, columns=['fuel', 'owner'], drop_first=True)
So in this technique it will remove 1 column after One Hot Encoding. As I mentioned earlier.
So it will remove 1 column from each One Hot Encoding result of 'fuel' and 'owner'.

```
pd.get_dummies(car_dataset, columns=['fuel', 'owner'], drop_first=True)
```

	brand	km_driven	selling_price	fuel_Diesel	fuel_LPG	fuel_Petrol	owner_Fourth & Above Owner	owner_Second Owner	owner_Test Drive Car	owner_Third Owner
0	Maruti	145500	450000	1	0	0	0	0	0	0
1	Skoda	120000	370000	1	0	0	0	1	0	0
2	Honda	140000	158000	0	0	1	0	0	0	1
3	Hyundai	127000	225000	1	0	0	0	0	0	0
4	Maruti	120000	130000	0	0	1	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...
8123	Hyundai	110000	320000	0	0	1	0	0	0	0
8124	Hyundai	119000	135000	1	0	0	1	0	0	0
8125	Maruti	120000	382000	1	0	0	0	0	0	0
8126	Tata	25000	290000	1	0	0	0	0	0	0
8127	Tata	25000	290000	1	0	0	0	0	0	0

8128 rows × 10 columns

Problem:
For Data Analysis we don't use get_dummies, pandas.
But for Machine Learning we can't do this using pandas.
Reason:
Pandas don't remember that what or which column is at what position.
Solution:
We use class sci-kit learn for One Hot Encoding.
Firstly we proceed with train test split.
Then we import class One Hot Encoder from sci-kit learn.
from sklearn.preprocessing import OneHotEncoder

One Hot Encoding using sklearn

```
car_dataset
```

	brand	km_driven	fuel	owner	selling_price
0	Maruti	145500	Diesel	First Owner	450000
1	Skoda	120000	Diesel	Second Owner	370000
2	Honda	140000	Petrol	Third Owner	158000
3	Hyundai	127000	Diesel	First Owner	225000
4	Maruti	120000	Petrol	First Owner	130000
...	...	...	...	...	...
8123	Hyundai	110000	Petrol	First Owner	320000
8124	Hyundai	119000	Diesel	Fourth & Above Owner	135000
8125	Maruti	120000	Diesel	First Owner	382000
8126	Tata	25000	Diesel	First Owner	290000
8127	Tata	25000	Diesel	First Owner	290000

8128 rows × 5 columns

```
X_train_car, X_test_car, y_train_car, y_test_car = train_test_split(car_dataset.iloc[:, 0:4], car_dataset.iloc[:, -1], test_size=0.2, random_state=42)
```

```
X_train_car
```

	brand	km_driven	fuel	owner
6518	Tata	2560	Petrol	First Owner
6144	Honda	80000	Petrol	Second Owner
6381	Hyundai	150000	Diesel	Fourth & Above Owner
438	Maruti	120000	Diesel	Second Owner
5939	Maruti	25000	Petrol	First Owner
...	...	...	...	...
5226	Mahindra	120000	Diesel	First Owner
5390	Maruti	80000	Diesel	Second Owner
860	Hyundai	35000	Petrol	First Owner
7603	Maruti	27000	Diesel	First Owner
7270	Maruti	70000	Petrol	Second Owner

6502 rows × 4 columns

```
X_test_car
```

	brand	km_driven	fuel	owner
1971	Honda	110000	Petrol	Third Owner
4664	Tata	291977	Diesel	First Owner
5448	Maruti	70000	Diesel	First Owner
3333	Honda	120000	Petrol	Second Owner
2316	Maruti	69000	Diesel	Second Owner
...	...	...	...	...
1149	BMW	8500	Diesel	First Owner
5002	Maruti	40000	Petrol	First Owner
6008	Hyundai	54043	Petrol	First Owner
2283	Tata	70000	Petrol	First Owner
5428	Chevrolet	110000	Petrol	Second Owner

1626 rows × 4 columns

```
y_train_car
```

```
6518    520000
6144    300000
6381    380000
438     530000
5939    335000
         ...
5226    475000
5390    530000
860     576000
7603    770000
7270    155000
Name: selling_price, Length: 6502, dtype: int64
```

```
y_test_car
```

```
1971     198000
4664     500000
5448     425000
3333     150000
2316     525000
          ...
1149    5500000
5002     370000
6008     374000
2283     575000
5428     140000
Name: selling_price, Length: 1626, dtype: int64
```

```
from sklearn.preprocessing import OneHotEncoder
```

```
OHE = OneHotEncoder()
```

We are not applying One Hot Encoding on all columns as 'km_driven' is not categorical and for 'brand' we will be applying One Hot Encoding later (Reason is not to increase Dimensionality for now).
So I just apply One Hot Encoding on columns 'fuel' and 'owner'.
So firstly we have to seperate the columns 'fuel' and 'owner' from dataset and then apply One Hot Encoding and then asemble the result columns with dataset again.
This takes lot of time as to seperate first, then apply One Hot Encoding then again asemble.
So to save the time and efforts we can use Column Transformer where we can apply transformer technique as per our own individual column choice.
For now I will be using the lengthy method which is without Column Transformer.

```
OHE.fit_transform(X_train_car[['fuel', 'owner']])
```

```
<6502x9 sparse matrix of type '<class 'numpy.float64'>'
	with 13004 stored elements in Compressed Sparse Row format>
```

We take columns 'fuel' and 'owner' for training because 'km' is not Nominal Categorical and 'brand' will increase Dimensionality for now. and then fit transformation technique on selected columns, as here we can see it seems the output is in form of sparse matrix, hence I will make it in array format like:

```
X_train_car_new = OHE.fit_transform(X_train_car[['fuel', 'owner']]).toarray()
```

```
X_train_car_new
```

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

Same I did for X_test_car data

```
X_test_car_new = OHE.transform(X_test_car[['fuel', 'owner']]).toarray()
```

```
X_test_car_new
```

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

Now to append the X_train_car_new and X_test_car_new with the column 'brand' and 'km' I will use hstack.
we do,

```
np.hstack((X_train_car[['brand', 'km_driven']].values, X_train_car_new))
```

```
array([['Tata', 2560, 0.0, ..., 0.0, 0.0, 0.0],
       ['Honda', 80000, 0.0, ..., 1.0, 0.0, 0.0],
       ['Hyundai', 150000, 0.0, ..., 0.0, 0.0, 0.0],
       ...,
       ['Hyundai', 35000, 0.0, ..., 0.0, 0.0, 0.0],
       ['Maruti', 27000, 0.0, ..., 0.0, 0.0, 0.0],
       ['Maruti', 70000, 0.0, ..., 1.0, 0.0, 0.0]], dtype=object)
```

```
np.hstack((X_test_car[['brand', 'km_driven']].values, X_test_car_new))
```

```
array([['Honda', 110000, 0.0, ..., 0.0, 0.0, 1.0],
       ['Tata', 291977, 0.0, ..., 0.0, 0.0, 0.0],
       ['Maruti', 70000, 0.0, ..., 0.0, 0.0, 0.0],
       ...,
       ['Hyundai', 54043, 0.0, ..., 0.0, 0.0, 0.0],
       ['Tata', 70000, 0.0, ..., 0.0, 0.0, 0.0],
       ['Chevrolet', 110000, 0.0, ..., 1.0, 0.0, 0.0]], dtype=object)
```

To remove one column from each transformed column (as discussed earlier 'n-1')
we do some changes as:
OHE = OneHotEncoder(drop='first')
And then run the code again.

```
OHE_1 = OneHotEncoder(drop='first')
```

```
OHE_1.fit_transform(X_train_car[['fuel', 'owner']])
```

```
<6502x7 sparse matrix of type '<class 'numpy.float64'>'
	with 8718 stored elements in Compressed Sparse Row format>
```

```
X_train_car_1 = OHE_1.fit_transform(X_train_car[['fuel', 'owner']]).toarray()
```

```
X_train_car_1
```

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

```
OHE_1.transform(X_test_car[['fuel', 'owner']])
```

```
<1626x7 sparse matrix of type '<class 'numpy.float64'>'
	with 2192 stored elements in Compressed Sparse Row format>
```

```
X_test_car_1 = OHE_1.transform(X_test_car[['fuel', 'owner']]).toarray()
```

```
X_test_car_1
```

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

```
np.hstack((X_train_car[['brand', 'km_driven']].values, X_train_car_1))
```

```
array([['Tata', 2560, 0.0, ..., 0.0, 0.0, 0.0],
       ['Honda', 80000, 0.0, ..., 1.0, 0.0, 0.0],
       ['Hyundai', 150000, 1.0, ..., 0.0, 0.0, 0.0],
       ...,
       ['Hyundai', 35000, 0.0, ..., 0.0, 0.0, 0.0],
       ['Maruti', 27000, 1.0, ..., 0.0, 0.0, 0.0],
       ['Maruti', 70000, 0.0, ..., 1.0, 0.0, 0.0]], dtype=object)
```

```
np.hstack((X_test_car[['brand', 'km_driven']].values, X_test_car_1))
```

```
array([['Honda', 110000, 0.0, ..., 0.0, 0.0, 1.0],
       ['Tata', 291977, 1.0, ..., 0.0, 0.0, 0.0],
       ['Maruti', 70000, 1.0, ..., 0.0, 0.0, 0.0],
       ...,
       ['Hyundai', 54043, 0.0, ..., 0.0, 0.0, 0.0],
       ['Tata', 70000, 0.0, ..., 0.0, 0.0, 0.0],
       ['Chevrolet', 110000, 0.0, ..., 1.0, 0.0, 0.0]], dtype=object)
```

If we don't want sparse matrix then just implement 'sparse=False'
And to avoid float or double values just implement 'dtype=int32'

```
OHE_2 = OneHotEncoder(drop='first', sparse=False, dtype='int32')
```

```
OHE_2.fit_transform(X_train_car[['fuel', 'owner']])
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_output` in version 1.2 and will be removed in 1.4. `spa
  warnings.warn(
array([[0, 0, 1, ..., 0, 0, 0],
       [0, 0, 1, ..., 1, 0, 0],
       [1, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 1, ..., 0, 0, 0],
       [1, 0, 0, ..., 0, 0, 0],
       [0, 0, 1, ..., 1, 0, 0]], dtype=int32)
```

```
X_train_car_2 = OHE_2.fit_transform(X_train_car[['fuel', 'owner']])
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_output` in version 1.2 and will be removed in 1.4. `spa
  warnings.warn(
```

```
X_train_car_2
```

```
array([[0, 0, 1, ..., 0, 0, 0],
       [0, 0, 1, ..., 1, 0, 0],
       [1, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 1, ..., 0, 0, 0],
       [1, 0, 0, ..., 0, 0, 0],
       [0, 0, 1, ..., 1, 0, 0]], dtype=int32)
```

```
OHE_2.transform(X_test_car[['fuel', 'owner']])
```

```
array([[0, 0, 1, ..., 0, 0, 1],
       [1, 0, 0, ..., 0, 0, 0],
       [1, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 1, ..., 0, 0, 0],
       [0, 0, 1, ..., 0, 0, 0],
       [0, 0, 1, ..., 1, 0, 0]], dtype=int32)
```

```
X_test_car_2 = OHE_2.transform(X_test_car[['fuel', 'owner']])
```

```
X_test_car_2
```

```
array([[0, 0, 1, ..., 0, 0, 1],
       [1, 0, 0, ..., 0, 0, 0],
       [1, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 1, ..., 0, 0, 0],
       [0, 0, 1, ..., 0, 0, 0],
       [0, 0, 1, ..., 1, 0, 0]], dtype=int32)
```

```
np.hstack((X_train_car[['brand', 'km_driven']].values, X_train_car_2))
```

```
array([['Tata', 2560, 0, ..., 0, 0, 0],
       ['Honda', 80000, 0, ..., 1, 0, 0],
       ['Hyundai', 150000, 1, ..., 0, 0, 0],
       ...,
```

```
           ['Hyundai', 35000, 0, ..., 0, 0, 0],
           ['Maruti', 27000, 1, ..., 0, 0, 0],
           ['Maruti', 70000, 0, ..., 1, 0, 0]], dtype=object)
```

```
np.hstack((X_test_car[['brand', 'km_driven']].values, X_test_car_2))
```

```
array([['Honda', 110000, 0, ..., 0, 0, 1],
       ['Tata', 291977, 1, ..., 0, 0, 0],
       ['Maruti', 70000, 1, ..., 0, 0, 0],
       ...,
       ['Hyundai', 54043, 0, ..., 0, 0, 0],
       ['Tata', 70000, 0, ..., 0, 0, 0],
       ['Chevrolet', 110000, 0, ..., 1, 0, 0]], dtype=object)
```

Now waiting is over, let's encode the column 'brand'

One Hot Encoding with top categories
As I already discussed to reduce dimensionality I will be choosing frequent categories which seems more important and for rest I will be going as 'others'
So we will set threshold

```
counts = car_dataset['brand'].value_counts()
```

Stored all the counts of brand

```
counts
```

```
Maruti           2448
Hyundai          1415
Mahindra          772
Tata              734
Toyota            488
Honda             467
Ford              397
Chevrolet         230
Renault           228
Volkswagen        186
BMW               120
Skoda             105
Nissan             81
Jaguar             71
Volvo              67
Datsun             65
Mercedes-Benz      54
Fiat               47
Audi               40
Lexus              34
Jeep               31
Mitsubishi         14
Force               6
Land                6
Isuzu               5
Kia                 4
Ambassador          4
Daewoo              3
MG                  3
Ashok               1
Opel                1
Peugeot             1
Name: brand, dtype: int64
```

```
car_dataset['brand'].nunique()
```

```
32
```

```
threshold=100
```

Found out unique values of brand and set variable 'threshold' with value 100

```
repl = counts[counts<=threshold].index
```

```
pd.get_dummies(car_dataset['brand'].replace(repl, 'uncommon'))
```

	BMW	Chevrolet	Ford	Honda	Hyundai	Mahindra	Maruti	Renault	Skoda	Tata	Toyota	Volkswagen	uncommon
0	0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	0	0	0	0
2	0	0	0	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
8123	0	0	0	0	1	0	0	0	0	0	0	0	0
8124	0	0	0	0	1	0	0	0	0	0	0	0	0
8125	0	0	0	0	0	0	1	0	0	0	0	0	0
8126	0	0	0	0	0	0	0	0	0	1	0	0	0
8127	0	0	0	0	0	0	0	0	0	1	0	0	0

8128 rows × 13 columns

Here I fetched the brands whose count is less than 100 and stored in repl
And from the above count:
Maruti 2448
Hyundai 1415
Mahindra 772
Tata 734
Toyota 488
Honda 467
Ford 397
Chevrolet 230
Renault 228
Volkswagen 186
BMW 120
Skoda 105
These brand are having count more than 100 and rest others we count as uncommon.

```
repl # Stored brand whose count is less than 100
```

```
Index(['Nissan', 'Jaguar', 'Volvo', 'Datsun', 'Mercedes-Benz', 'Fiat', 'Audi',
       'Lexus', 'Jeep', 'Mitsubishi', 'Force', 'Land', 'Isuzu', 'Kia',
       'Ambassador', 'Daewoo', 'MG', 'Ashok', 'Opel', 'Peugeot'],
      dtype='object')
```

Now I will be applying One Hot Encoding on 'brand' just wherever count of the particular brand is less than 100 it will be said/renamed as uncommon (as less than 100 brand we stored in repl) So all the repl brand will be named as uncommon.

Now let's discuss the technique which takes less efforts and saves time.
Let's understand how Column Transformer works which I already mentioned above.

COLUMN TRANSFORMER

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

```
covid_dataset = pd.read_csv('covid.csv')
```

```
covid_dataset
```

	age	gender	fever	cough	city	has_covid
0	60	Male	103.0	Mild	Kolkata	No
1	27	Male	100.0	Mild	Delhi	Yes
2	42	Male	101.0	Mild	Delhi	No
3	31	Female	98.0	Mild	Kolkata	No
4	65	Female	101.0	Mild	Mumbai	No
...	...	...	...	...	...	...
95	12	Female	104.0	Mild	Bangalore	No
96	51	Female	101.0	Strong	Kolkata	Yes
97	20	Female	101.0	Mild	Bangalore	No
98	5	Female	98.0	Strong	Mumbai	No
99	10	Female	98.0	Strong	Kolkata	Yes

100 rows × 6 columns

```
covid_dataset.isna().sum()
```

```
age           0
gender        0
fever        10
cough         0
city          0
has_covid     0
dtype: int64
```

```
covid_dataset.describe()
```

	age	fever
count	100.000000	90.000000
mean	44.220000	100.844444
std	24.878931	2.054926
min	5.000000	98.000000
25%	20.000000	99.000000
50%	45.000000	101.000000
75%	66.500000	102.750000
max	84.000000	104.000000

Suppose we have data as:

```
+----------+----------+----------+----------+
|   Age    |   City   |  Gender  |  Review  |
+----------+----------+----------+----------+
```

where Age have 20 missing values.
Age --> Simple Imputer
City --> Nominal Encoding
Gender --> Nominal Encoding
Review --> Ordinal Encoding
Here we can observe that for individual columns we need to apply different transformer techniques.

```
Age   City   Gender    Review
 |     |       |         |
 |      \_____/          |
 |        |              |
[ ]    [       ]        [ ]
 \___________|___________/
             |
          [     ]
```

So to save the time we use column transformer

Simple Imputer:
Replace missing values using descriptive statistics (e.g., mean, median, most frequent) along each column or using constant value.
After importing dataset check value counts for each column (It will help us to show different categories).
So for column 'gender' and 'city' we will apply One Hot Encoding and for column 'cough' we will use ordinal encoding as it shows relationship or intrinsic order.

So as we have 10 missing values of fever,
gender --> One Hot Encoding
fever --> Simple Imputer
cough --> Ordinal Encoding
city --> One Hot Encoding
has_cough --> Label Encoding
(has_cough has label encoding because it is target variable and its nominal)
Lets train test split data with ratio 80%-20%.

```
X_train_covid, X_test_covid, y_train_covid, y_test_covid = train_test_split(covid_dataset[[ 'age',  'gender',   'fever',    'cough',    'city']], covid_dataset['has_covid'], test_size=0.2,
```

```
X_train_covid
```

	age	gender	fever	cough	city
55	81	Female	101.0	Mild	Mumbai
88	5	Female	100.0	Mild	Kolkata
26	19	Female	100.0	Mild	Kolkata
42	27	Male	100.0	Mild	Delhi
69	73	Female	103.0	Mild	Delhi
...	...	...	...	...	...
60	24	Female	102.0	Strong	Bangalore
71	75	Female	104.0	Strong	Delhi
14	51	Male	104.0	Mild	Bangalore
92	82	Female	102.0	Strong	Kolkata
51	11	Female	100.0	Strong	Kolkata

80 rows × 5 columns

```
X_test_covid
```

	age	gender	fever	cough	city
83	17	Female	104.0	Mild	Kolkata
53	83	Male	98.0	Mild	Delhi
70	68	Female	101.0	Strong	Delhi
45	72	Male	99.0	Mild	Bangalore
44	20	Male	102.0	Strong	Delhi
39	50	Female	103.0	Mild	Kolkata
22	71	Female	98.0	Strong	Kolkata
80	14	Female	99.0	Mild	Mumbai
10	75	Female	NaN	Mild	Delhi
0	60	Male	103.0	Mild	Kolkata
18	64	Female	98.0	Mild	Bangalore
30	15	Male	101.0	Mild	Delhi
73	34	Male	98.0	Strong	Kolkata
33	26	Female	98.0	Mild	Kolkata
90	59	Female	99.0	Strong	Delhi
4	65	Female	101.0	Mild	Mumbai
76	80	Male	100.0	Mild	Bangalore
77	8	Female	101.0	Mild	Kolkata
12	25	Female	99.0	Strong	Kolkata
31	83	Male	103.0	Mild	Kolkata

Without Using Column Trasformer

Simple Impuuter -> fever
import Simple Imputer from sci-kit learn
Then transform column 'fever' using simple imputer in both X_train_covid and X_test_covid and then store in variable from X_train_fever and X_test_fever

```
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import OrdinalEncoder
```

```
SI = SimpleImputer()
```

```
X_train_fever = SI.fit_transform(X_train_covid[['fever']])
```

```
X_train_fever
```

```
array([[101.],
       [100.],
       [100.],
       [100.],
       [103.],
       [103.],
       [102.],
       [101.],
       [101.],
       [101.],
       [ 98.],
       [104.],
       [103.],
       [104.],
       [100.],
       [101.],
       [104.],
       [102.],
       [102.],
       [103.],
       [104.],
       [102.],
       [101.],
       [104.],
       [102.],
```

```
       [101.],
       [ 99.],
       [101.],
       [104.],
       [102.],
       [100.],
       [ 98.],
       [ 98.],
       [101.],
       [100.],
       [100.],
       [101.],
       [104.],
       [101.],
       [103.],
       [101.],
       [ 98.],
       [ 99.],
       [ 99.],
       [101.],
       [ 99.],
       [101.],
       [104.],
       [ 98.],
       [101.],
       [103.],
       [101.],
       [ 98.],
       [ 99.],
       [ 98.],
       [ 99.],
       [102.],
       [101.],
```

```
X_test_fever = SI.fit_transform(X_test_covid[['fever']])
```

```
X_test_fever
```

```
array([[104.        ],
       [ 98.        ],
       [101.        ],
       [ 99.        ],
       [102.        ],
       [103.        ],
       [ 98.        ],
       [ 99.        ],
       [100.26315789],
       [103.        ],
       [ 98.        ],
       [101.        ],
       [ 98.        ],
       [ 98.        ],
       [ 99.        ],
       [101.        ],
       [100.        ],
       [101.        ],
       [ 99.        ],
       [103.        ]])
```

```
X_train_fever.shape
```

```
(80, 1)
```

Ordinal Encoding -> cough

Then transform column 'cough' using Ordinal Encoder from both X_train_covid, X_test_covid and store in variable X_train_cough, X_test_cough

```
OE = OrdinalEncoder(categories=[['Mild','Strong']])
```

```
X_train_cough = OE.fit_transform(X_train_covid[['cough']])
```

```
X_train_cough
```

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

```
       [0.],
       [1.],
```

```
X_test_cough = OE.fit_transform(X_test_covid[['cough']])
```

```
X_test_cough
```

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

```
X_train_cough.shape
```

```
(80, 1)
```

One Hot Encoding -> gender, city

Then transform column 'gender' and 'city' using One Hot Encoding from both X_train_gender_city and X_test_gender_city. Removing first column after encoding and setting sparse matrix as false.

```
OHE_A = OneHotEncoder(drop='first',sparse=False)
```

```
X_train_gender_city = OHE_A.fit_transform(X_train_covid[['gender','city']])
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_output` in version 1.2 and will be removed in 1.4. `spa
  warnings.warn(
```

```
X_train_gender_city
```

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

```
X_test_gender_city = OHE_A.fit_transform(X_test_covid[['gender','city']])
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_output` in version 1.2 and will be removed in 1.4. `spa
  warnings.warn(
```

```
X_train_gender_city
```

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

```
       [0., 0., 1., 0.],
       [0., 0., 0., 1.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [1., 0., 1., 0.],
       [1., 0., 1., 0.],
       [0., 0., 0., 0.],
       [1., 0., 1., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 1., 0.],
       [1., 0., 0., 1.],
       [1., 0., 0., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 0., 0.],
       [1., 1., 0., 0.],
       [1., 0., 0., 1.],
       [0., 0., 0., 0.],
       [1., 0., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 0., 0.],
       [1., 0., 0., 0.],
       [1., 0., 0., 0.],
       [1., 0., 0., 1.],
       [0., 0., 0., 0.],
       [1., 0., 0., 0.],
       [1., 0., 1., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.],
       [0., 0., 0., 0.],
       [1., 1., 0., 0.],
       [1., 0., 0., 0.],
       [0., 0., 0., 1.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.],
       [0., 0., 0., 0.],
       [0., 0., 1., 0.],
       [1., 0., 1., 0.],
       [1., 0., 0., 1.],
       [1., 0., 0., 0.],
       [0., 0., 0., 1.],
       [0., 0., 0., 0.],
       [1., 0., 1., 0.],
       [0., 1., 0., 0.],
```

```
X_train_gender_city.shape
```

```
(80, 4)
```

Concatenate all these
Step-1: Extract 'age' column
Step-2: Concatenate 'X_train_age', 'X_train_fever', 'X_train_cough', 'X_train_gender_city' and store concatenation in X_train_transformed.
Step-3: Concatenate 'X_test_age', 'X_test_fever', 'X_test_cough', 'X_test_gender_city' and store concatenation in X_test_transformed.

```
X_train_age = X_train_covid.drop(columns=['gender','fever','cough','city']).values
```

```
X_test_age = X_test_covid.drop(columns=['gender','fever','cough','city']).values
```

```
X_train_age.shape
```

```
(80, 1)
```

```
X_train_transformed = np.concatenate((X_train_age,X_train_fever,X_train_gender_city,X_train_cough),axis=1)
```

```
X_test_transformed = np.concatenate((X_test_age,X_test_fever,X_test_gender_city,X_test_cough),axis=1)
```

```
X_train_transformed.shape
```

```
(80, 7)
```

Using Column Transformer

import Column Transformer from sci-kit learn and store this in object, let's say CT.

```
from sklearn.compose import ColumnTransformer
```

Now, I will pass list for transformers

```
CT = ColumnTransformer(transformers=[])
```

In list we give name for each transformation technique we are applying on individual columns
Then I will pass remainder,
Remainder -> We don't apply transformation technique sometimes on all columns, so the remaining columns will have two options
(i) drop the columns (ii) remain those as it is
for drop:
CT = ColumnTransformer(transformers=[], remainder='drop')
for remaining it as it is
CT = ColumnTransformer(transformers=[], remainder='passthrough')

```
CT = ColumnTransformer(transformers=[
    ('tnf1',SimpleImputer(),['fever']),
    ('tnf2',OrdinalEncoder(categories=[['Mild','Strong']]),['cough']),
    ('tnf3',OneHotEncoder(sparse=False,drop='first'),['gender','city'])
],remainder='passthrough')
```

In 'tnf1' I simply applied SimpleImputer() technique on column 'fever'
In 'tnf2' I applied OrdinalEncoder on column 'cough' where we declared level for the categories as: Mild < Strong
In 'tnf3' I applied OneHotEncoder on columns 'gender' and 'city'and set sparse matrix as False and after getting implemented we set drop='first'
so that it will drop first column from each individual columns result after implementation of OneHotEncoder on 'gender' and 'city'.

```
CT.fit_transform(X_train_covid).shape
```

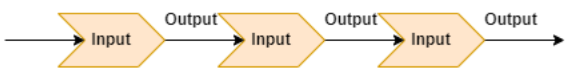
```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_output` in version 1.2 and will be removed in 1.4. `spa
  warnings.warn(
(80, 7)
```

```
CT.transform(X_test_covid).shape
```

```
(20, 7)
```

PIPELINES

Pipelines chains together multiple steps so that the output of each step is used as input to the next step.
Pipelines makes it easy to apply the same pre-processing to train and test.



Let's understand what efforts do we need if we don't use Pipeline

Without using Pipeline

So here I will be using Titanic Dataset.

Firstly I will import all the necessary libraries

```
import numpy as np
import pandas as pd
```

```
from sklearn.model_selection import train_test_split
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import Pipeline,make_pipeline
from sklearn.feature_selection import SelectKBest,chi2
from sklearn.tree import DecisionTreeClassifier
```

Now I will import dataset

```
titanic_dataset = pd.read_csv('titanic_dataset.csv')
```

```
titanic_dataset
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
...	...	...	...	...	...	...	...	...	...	...	...	...
886	887	0	2	Montvila, Rev. Juozas	male	27.0	0	0	211536	13.0000	NaN	S
887	888	1	1	Graham, Miss. Margaret Edith	female	19.0	0	0	112053	30.0000	B42	S
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 6607	23.4500	NaN	S
889	890	1	1	Behr, Mr. Karl Howell	male	26.0	0	0	111369	30.0000	C148	C
890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.7500	NaN	Q

891 rows × 12 columns

```
titanic_dataset.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

```
titanic_dataset.tail()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
886	887	0	2	Montvila, Rev. Juozas	male	27.0	0	0	211536	13.00	NaN	S
887	888	1	1	Graham, Miss. Margaret Edith	female	19.0	0	0	112053	30.00	B42	S
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 6607	23.45	NaN	S
889	890	1	1	Behr, Mr. Karl Howell	male	26.0	0	0	111369	30.00	C148	C
890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.75	NaN	Q

Let's find missing values for our dataset

```
titanic_dataset.isna().sum()
```

```
PassengerId     0
Survived        0
Pclass          0
Name            0
Sex             0
```

```
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

Now I will drop 'PassengerId', 'Name', 'Ticket', 'Cabin' as it won't show anything much informative

```
titanic_dataset.drop(columns=['PassengerId','Name','Ticket','Cabin'],inplace=True)
```

Now I will train, test, split the dataset
Target variable --> 'Survived'

```
X_train,X_test,y_train,y_test = train_test_split(titanic_dataset.drop(columns=['Survived']), titanic_dataset['Survived'], test_size=0.2, random_state=42)
```

```
X_train
```

	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
331	1	male	45.5	0	0	28.5000	S
733	2	male	23.0	0	0	13.0000	S
382	3	male	32.0	0	0	7.9250	S
704	3	male	26.0	1	0	7.8542	S
813	3	female	6.0	4	2	31.2750	S
...	...	...	...	...	...	...	...
106	3	female	21.0	0	0	7.6500	S
270	1	male	NaN	0	0	31.0000	S
860	3	male	41.0	2	0	14.1083	S
435	1	female	14.0	1	2	120.0000	S
102	1	male	21.0	0	1	77.2875	S

712 rows × 7 columns

```
X_test
```

	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
709	3	male	NaN	1	1	15.2458	C
439	2	male	31.0	0	0	10.5000	S
840	3	male	20.0	0	0	7.9250	S
720	2	female	6.0	0	1	33.0000	S
39	3	female	14.0	1	0	11.2417	C
...	...	...	...	...	...	...	...
433	3	male	17.0	0	0	7.1250	S
773	3	male	NaN	0	0	7.2250	C
25	3	female	38.0	1	5	31.3875	S
84	2	female	17.0	0	0	10.5000	S
10	3	female	4.0	1	1	16.7000	S

179 rows × 7 columns

```
y_train
```

```
331    0
733    0
382    0
704    0
813    0
      ..
106    1
270    0
860    0
435    1
102    0
Name: Survived, Length: 712, dtype: int64
```

```
y_test
```

```
709    1
439    0
840    0
720    1
39     1
      ..
433    0
773    0
25     1
84     1
10     1
Name: Survived, Length: 179, dtype: int64
```

Now I must check missing values for chosen features.

```
titanic_dataset.isnull().sum()
```

```
Survived      0
Pclass        0
Sex           0
Age         177
SibSp         0
Parch         0
Fare          0
Embarked      2
dtype: int64
```

As here we can see column 'Age' has 177 missing values and column 'Embarked' has 2 missing values.

Those columns which have missing values and which are not categorical must be transformed or filled using Simple Imputer.

```
SI_1 = SimpleImputer()
SI_2 = SimpleImputer(strategy='most_frequent')
```

SI_1 --> Simple Imputer where missing values will be filled with mean value.
SI_2 --> Simple Imputer where missing values will be filled with most frequent values.
Apply SI_1 on X_train['Age'] and store in any variable lets say X_train_age.
then apply SI_2 on X_train['Embarked'] and store in any variable lets say X_train_embarked and then transform.

```
X_train_age = SI_1.fit_transform(X_train[['Age']])
X_train_embarked = SI_2.fit_transform(X_train[['Embarked']])

X_test_age = SI_1.transform(X_test[['Age']])
X_test_embarked = SI_2.transform(X_test[['Embarked']])
```

```
X_train_age
```

```
array([[45.5       ],
       [23.        ],
       [32.        ],
       [26.        ],
       [ 6.        ],
       [24.        ],
       [45.        ],
       [29.        ],
       [29.49884615],
       [29.49884615],
       [42.        ],
       [36.        ],
       [33.        ],
       [17.        ],
       [29.        ],
       [50.        ],
       [35.        ],
       [38.        ],
       [34.        ],
       [17.        ],
       [11.        ],
       [61.        ],
       [30.        ],
       [ 7.        ],
       [63.        ],
       [20.        ],
       [29.49884615],
       [29.        ],
       [36.        ],
       [29.49884615],
       [50.        ],
       [27.        ],
       [30.        ],
       [33.        ],
       [29.49884615],
       [29.49884615],
       [ 2.        ],
       [25.        ],
       [51.        ],
       [25.        ],
       [29.49884615],
       [29.49884615],
       [24.        ],
       [18.        ],
       [29.49884615],
       [25.        ],
       [24.        ],
       [22.        ],
       [ 0.92      ],
       [24.        ],
       [26.        ],
       [34.        ],
       [21.        ],
       [29.49884615],
       [29.49884615],
       [29.49884615],
       [22.        ],
       [62.        ],
```

```
X_train_embarked
```

```
array([['S'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['C'],
       ['S'],
       ['C'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['C'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['C'],
       ['C'],
       ['S'],
       ['S'],
       ['C'],
       ['S'],
       ['C'],
       ['S'],
       ['Q'],
       ['Q'],
       ['S'],
       ['S'],
       ['S'],
       ['C'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
       ['S'],
```

```
       ['S'],
       ['C'],
       ['Q'],
       ['C'],
       ['S'],
       ['S'],
```

One Hot Encoding

As column Sex and Embarked needs One Hot Encoding because they are Nominal and which are not target variable.

So we will store OneHotEncoder in object OHE and set sparse as False and missing or unknown values as ignore.

Then I apply OHE on X_train['Sex'] and store it in X_train_Sex.

Then I apply OHE on X_train['Embarked'] and store it in and then transform.

```
OHE_sex = OneHotEncoder(sparse=False,handle_unknown='ignore')
OHE_embarked = OneHotEncoder(sparse=False,handle_unknown='ignore')
```

```
X_train_sex = OHE_sex.fit_transform(X_train[['Sex']])
X_train_embarked = OHE_embarked.fit_transform(X_train_embarked)

X_test_sex = OHE_sex.transform(X_test[['Sex']])
X_test_embarked = OHE_embarked.transform(X_test_embarked)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_output` in version 1.2 and will be removed in 1.4. `spa
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_output` in version 1.2 and will be removed in 1.4. `spa
  warnings.warn(
```

```
X_train_sex
```

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

```
X_train_embarked
```

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

```
X_test_sex
```

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

```
X_test_embarked
```

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

```
       [1., 0., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [1., 0., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [1., 0., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [1., 0., 0.],
       [0., 0., 1.],
       [0., 1., 0.],
       [1., 0., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 1., 0.],
       [1., 0., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [1., 0., 0.],
       [1., 0., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [1., 0., 0.],
```

```
X_train
```

	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
331	1	male	45.5	0	0	28.5000	S
733	2	male	23.0	0	0	13.0000	S
382	3	male	32.0	0	0	7.9250	S
704	3	male	26.0	1	0	7.8542	S
813	3	female	6.0	4	2	31.2750	S
...	...	...	...	...	...	...	...
106	3	female	21.0	0	0	7.6500	S
270	1	male	NaN	0	0	31.0000	S
860	3	male	41.0	2	0	14.1083	S
435	1	female	14.0	1	2	120.0000	S
102	1	male	21.0	0	1	77.2875	S

712 rows × 7 columns

Now I am going to create an object where I will be storing remaining columns and not affecting them / not making any changes in them.

```
X_train_rem = X_train.drop(columns=['Sex','Age','Embarked'])
```

```
X_test_rem = X_test.drop(columns=['Sex','Age','Embarked'])
```

Now let's concatenate the results.

```
X_train_transformed = np.concatenate((X_train_rem, X_train_age,X_train_sex, X_train_embarked), axis=1)
```

```
X_train_transformed
```

```
array([[1., 0., 0., ..., 0., 0., 1.],
       [2., 0., 0., ..., 0., 0., 1.],
       [3., 0., 0., ..., 0., 0., 1.],
       ...,
       [3., 2., 0., ..., 0., 0., 1.],
       [1., 1., 2., ..., 0., 0., 1.],
       [1., 0., 1., ..., 0., 0., 1.]])
```

```
X_test_transformed = np.concatenate((X_test_rem, X_test_age,X_test_sex, X_test_embarked), axis=1)
```

```
X_test_transformed
```

```
array([[3., 1., 1., ..., 1., 0., 0.],
       [2., 0., 0., ..., 0., 0., 1.],
       [3., 0., 0., ..., 0., 0., 1.],
       ...,
       [3., 1., 5., ..., 0., 0., 1.],
       [2., 0., 0., ..., 0., 0., 1.],
       [3., 1., 1., ..., 0., 0., 1.]])
```

Simply what I did is created transformation and concatenated its result with the remaining columns left.
Same we did for X_test

```
X_test_transformed.shape
```

```
(179, 10)
```

```
from sklearn.tree import DecisionTreeClassifier
```

just imported DecisionTreeClassifier and stored model in object DTC.

```
DTC = DecisionTreeClassifier()
```

Now let's fit the model for X_train_transformed and y_train

```
DTC.fit(X_train_transformed, y_train)
```

```
▾ DecisionTreeClassifier
DecisionTreeClassifier()
```

Let us create an object where we will store the predicted values for X_test_transformed, after X_train_transformed fit on DecisionTreeClassifier algorithm so that it will understand the data accordingly and will apply the same algorithm on it.
So we create an object named 'y_prediction' where I will be storing all the predicted values.
Then I will predict the predicted output for input **X_test_transformed**.

```
y_prediction = DTC.predict(X_test_transformed)
```

Lets look at our predicted values.

```
y_prediction
```

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

I will be now checking the accuracy so I will just import model accuracy_score from metrics which is available in sklearn.

Then apply this accuracy_score on y_train and y_prediction that how much of them is similar (in percentage).

```
from sklearn.metrics import accuracy_score
accuracy_score(y_test,y_prediction)
```

```
0.7877094972067039
```

Here we can see that accuracy is around 70%.

Now I want that after taking new input from user and based in that input I want to predict output.

To take our model on web page we use pickle.

Why is pickle used?
In simple words pickle I am using here for exporting models.

```
import pickle
```

```
import os
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
directory_path = '/content/drive/My Drive/Colab Notebooks/'
```

So I will be exporting transform of One Hot Enncoding of column Sex and Embarked only.
Because 'Age' and 'Embarked' although we used for transforming, but we did Simple Imputer transformation on it as there were missing values but while taking input from user we don't need Simple Imputer model and hence we won't export it.

```
pickle.dump(OHE_sex, open(directory_path + 'OHE_sex.pkl', 'wb'))
pickle.dump(OHE_embarked, open(directory_path + 'OHE_embarked.pkl', 'wb'))
pickle.dump(DTC, open(directory_path + 'DTC.pkl', 'wb'))
```

Let's go to new file...