main

April 9, 2022

1 Car prices prediction

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sb
import pickle
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPRegressor
```

1.1 Dataset Exploration :

[2]:
```
DataSet = pd.read_csv("DataSet.csv")

DataSet.head()
```

[2]:
```
                           name  year  selling_price  km_driven    fuel  \
0        Maruti Swift Dzire VDI  2014         450000     145500  Diesel
1  Skoda Rapid 1.5 TDI Ambition  2014         370000     120000  Diesel
2      Honda City 2017-2020 EXi  2006         158000     140000  Petrol
3     Hyundai i20 Sportz Diesel  2010         225000     127000  Diesel
4        Maruti Swift VXI BSIII  2007         130000     120000  Petrol

  seller_type transmission         owner     mileage   engine   max_power  \
0  Individual       Manual   First Owner   23.4 kmpl  1248 CC      74 bhp
1  Individual       Manual  Second Owner  21.14 kmpl  1498 CC  103.52 bhp
2  Individual       Manual   Third Owner   17.7 kmpl  1497 CC      78 bhp
3  Individual       Manual   First Owner   23.0 kmpl  1396 CC      90 bhp
4  Individual       Manual   First Owner   16.1 kmpl  1298 CC    88.2 bhp

                    torque  seats
0           190Nm@ 2000rpm    5.0
1      250Nm@ 1500-2500rpm    5.0
2    12.7@ 2,700(kgm@ rpm)    5.0
3  22.4 kgm at 1750-2750rpm    5.0
```

```
   4     11.5@ 4,500(kgm@ rpm)    5.0
```

[3]:
```
DataSet.shape
```

[3]:
```
(8128, 13)
```

[4]:
```
DataSet.dtypes
```

[4]:
```
name              object
year               int64
selling_price      int64
km_driven          int64
fuel              object
seller_type       object
transmission      object
owner             object
mileage           object
engine            object
max_power         object
torque            object
seats            float64
dtype: object
```

[5]:
```
DataSet.isnull().sum()
```

[5]:
```
name               0
year               0
selling_price      0
km_driven          0
fuel               0
seller_type        0
transmission       0
owner              0
mileage          221
engine           221
max_power        215
torque           222
seats            221
dtype: int64
```

1.1.1 We can see that this dataset contains some rows that have empty cells (NaN), In this case we are going to get rid of those rows but we can for example replace those cells with their colmn's mean value

[6]:
```
DataSet = DataSet.dropna()
DataSet.isnull().sum().sum()
```

[6]:
```
0
```

1.1.2 Here, we create 2 functions :

- Numerise : which will help us replace string columns with number (for example : for the transmission column manual is replaced with 0 and automatic is replaced with 1)
- ExtractNum : which will help us extract numerical features from cell (for example : 32 kmpl becomes 32)

[7]:
```
def Numerise(DF, Column):
    x = DF[Column]
    col = pd.factorize(x)
    return col
```

[8]:
```
def ExtractNum(DF, Column):
    x = DF[Column]
    x = x.astype('str').str.extract(r'(\d+.)').astype(float)
    return np.array(x)
```

[9]:
```
DataSet.dtypes
```

[9]:
```
name              object
year               int64
selling_price      int64
km_driven          int64
fuel              object
seller_type       object
transmission      object
owner             object
mileage           object
engine            object
max_power         object
torque            object
seats            float64
dtype: object
```

1.1.3 And finally here we are creating a new DataFrame containing nothing but numerical values, some columns are discarded either because their features are represented inconsistently(Torque), or because they offer no use in our case(Name)

[10]:
```
NewDataSet = pd.DataFrame({"year" : np.array(DataSet["year"]),
                           "SellingPrice" : np.array(DataSet["selling_price"]),
                           "kmDriven" : np.array(DataSet["km_driven"]),
                           "fuel" : Numerise(DataSet, "fuel")[0],
                           "Seller" : Numerise(DataSet, "seller_type")[0],
                           "transmission" : Numerise(DataSet,␣
 ↪"transmission")[0],
                           "owner" : Numerise(DataSet, "owner")[0],
                           "mileage" : ExtractNum(DataSet, "mileage").flatten(),
                           "engine" : ExtractNum(DataSet, "engine").flatten(),
                           "maxPower" : ExtractNum(DataSet, "max_power").
 ↪flatten(),
                           "Seats" : np.array(DataSet["seats"]),
                          })
NewDataSet
```

[10]:

	year	SellingPrice	kmDriven	fuel	Seller	transmission	owner \
0	2014	450000	145500	0	0	0	0
1	2014	370000	120000	0	0	0	1
2	2006	158000	140000	1	0	0	2
3	2010	225000	127000	0	0	0	0
4	2007	130000	120000	1	0	0	0
…	…	…	…	…	…	…	…
7901	2013	320000	110000	1	0	0	0
7902	2007	135000	119000	0	0	0	3
7903	2009	382000	120000	0	0	0	0
7904	2013	290000	25000	0	0	0	0
7905	2013	290000	25000	0	0	0	0

	mileage	engine	maxPower	Seats
0	23.0	1248.0	74.0	5.0
1	21.0	1498.0	103.0	5.0
2	17.0	1497.0	78.0	5.0
3	23.0	1396.0	90.0	5.0
4	16.0	1298.0	88.0	5.0
…	…	…	…	…
7901	18.0	1197.0	82.0	5.0
7902	16.0	1493.0	110.0	5.0
7903	19.0	1248.0	73.0	5.0
7904	23.0	1396.0	70.0	5.0
7905	23.0	1396.0	70.0	5.0

[7906 rows x 11 columns]

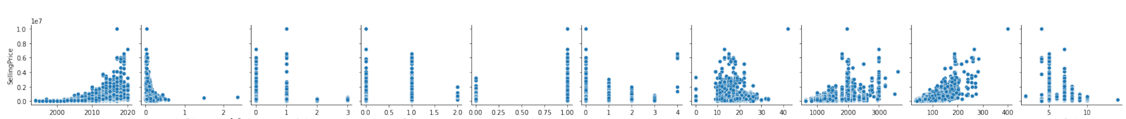
[11]:
```
NewDataSet.describe()
```

[11]:
```
              year  SellingPrice      kmDriven         fuel       Seller  \
count  7906.000000  7.906000e+03  7.906000e+03  7906.000000  7906.000000
mean   2013.983936  6.498137e+05  6.918866e+04     0.473817     0.199722
std       3.863695  8.135827e+05  5.679230e+04     0.545591     0.468575
min    1994.000000  2.999900e+04  1.000000e+00     0.000000     0.000000
25%    2012.000000  2.700000e+05  3.500000e+04     0.000000     0.000000
50%    2015.000000  4.500000e+05  6.000000e+04     0.000000     0.000000
75%    2017.000000  6.900000e+05  9.542500e+04     1.000000     0.000000
max    2020.000000  1.000000e+07  2.360457e+06     3.000000     2.000000

       transmission        owner      mileage       engine     maxPower  \
count   7906.000000  7906.000000  7906.000000  7906.000000  7906.000000
mean       0.131672     0.447255    18.981027  1458.708829    91.271060
std        0.338155     0.710854     4.064364   503.893057    35.732781
min        0.000000     0.000000     0.000000   624.000000    32.000000
25%        0.000000     0.000000    16.000000  1197.000000    68.000000
50%        0.000000     0.000000    19.000000  1248.000000    82.000000
75%        0.000000     1.000000    22.000000  1582.000000   102.000000
max        1.000000     4.000000    42.000000  3604.000000   400.000000

             Seats
count  7906.000000
mean      5.416393
std       0.959208
min       2.000000
25%       5.000000
50%       5.000000
75%       5.000000
max      14.000000
```

1.1.4 And here, we scatter plot the selling price with all the other parameters to inspect for relations between them.

[12]:
```
sb.pairplot(data=NewDataSet,
                  x_vars=['year', 'kmDriven', 'fuel', 'Seller', 'transmission',␣
 ↪'owner', 'mileage', 'engine', 'maxPower', 'Seats'],
                  y_vars=['SellingPrice'])
```

[12]: <seaborn.axisgrid.PairGrid at 0x7ff9fb7d8460>



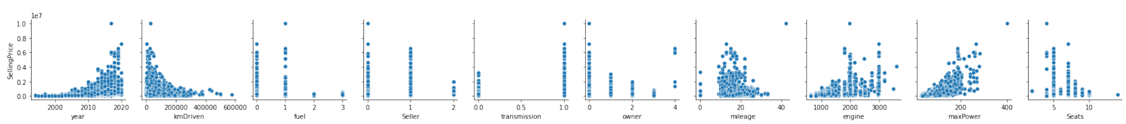
1.1.5 We can see from the graph above, that some of the values just doesn't make sense or are just impossible in real life (for example: kmDriven > 1000000), this is very common in hand typed datasets where human caused error is present, especially with large datasets, we can just remove those values as shown below :

[13]:
```
NewDataSet = NewDataSet[NewDataSet["kmDriven"]<=700000]
NewDataSet.shape
```

[13]:
```
(7904, 11)
```

[14]:
```
sb.pairplot(data=NewDataSet,
                    x_vars=['year', 'kmDriven', 'fuel', 'Seller', 'transmission',␣
 ↪'owner', 'mileage', 'engine', 'maxPower', 'Seats'],
                    y_vars=['SellingPrice'])
```

[14]:
```
<seaborn.axisgrid.PairGrid at 0x7ff9f94332b0>
```



1.1.6 We can see from the graph below that the Selling price have some relation with the following columns :

- Year
- kmDriven
- owner
- mileage
- engine
- maxPower
- Seats

1.2 Preparing for Training :

1.2.1 Before we train our model we first have to separate our dataset to 2 seperate DataFrames:

- Features : which contains the input for the model
- Labels : the expected output (in our case :
- the features contains : Year, kmDriven, owner, mileage, engine, maxPower, seats
- the label is the selling price)

[15]:
```
X = NewDataSet.drop(["SellingPrice", "fuel", "Seller", "transmission"], axis =␣
 ↪1)
Y = pd.DataFrame({"SellingPrice" : NewDataSet["SellingPrice"]})
```

[16]:
```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state=1)
```

Instansiating our neural network model inside a pipeline, this pipeline contains a StandardScaler which allows us to normalize the input for our model, it also contains an MLPRegressor (Multi-Layer Perceptron Regressor) with 2 hidden Layers each containing 64 nodes.

After that we fit our model.

[ ]:
```
regressor = make_pipeline(
    StandardScaler(),
    MLPRegressor(
        hidden_layer_sizes = (64, 64),
        activation = "relu",
        solver = "adam",
        alpha = 0.0005,
        learning_rate='constant',
        learning_rate_init=0.005,
        max_iter = 10000
    )
)

regressor.fit(X_train, Y_train)
```

1.2.2 As we can see below our model has an over-all accuracy of 97.4% when tested with the test data we created earlier :

[18]:
```
(regressor.score(X_test, Y_test))**0.5
```

[18]:
```
0.9740203924808798
```

1.2.3 Here, we make a direct comparison between the expected selling price and the predicted selling price :

[19]:
```
results = pd.DataFrame({
    "Expected" : (Y_test["SellingPrice"]),
    "Predicted" :(regressor.predict(X_test))
})

results.head()
```

[19]:

	Expected	Predicted
1838	150000	138089.089721
6988	725000	554629.026174

```
1886    200000  179149.096178
7902    135000  217936.653645
1508    750000  750274.346333
```

1.2.4 All in all, the model is quite accurate for most cases, but is a bit hit-or-miss depending on the input, as we can see below, most of the time, the model will work as expected !

[20]:
```python
results.describe()
```

[20]:
```
            Expected     Predicted
count  1.976000e+03  1.976000e+03
mean   6.456362e+05  6.444082e+05
std    7.917996e+05  7.618389e+05
min    3.500000e+04  2.300291e+04
25%    2.737500e+05  2.824326e+05
50%    4.500000e+05  4.847263e+05
75%    6.750000e+05  6.717931e+05
max    6.223000e+06  6.451111e+06
```

1.2.5 We Use Pickle to save the Model Object as a binary file that you can use in other projects as shown below :

[21]:
```python
pickle.dump(regressor, open("Regressor", 'wb'))
```

[22]:
```python
LoadedModel = pickle.load(open("Regressor", 'rb'))
LoadedModel.predict(X_test[:1])
```

[22]:
```
array([138089.08972082])
```