

Notebook - Used Car Price Prediction Linear Regression

January 7, 2020

1 Project: Linear Regression Price Predictor for Used Cars

```
[1]: import pandas as pd
import numpy as np

from sklearn.linear_model import LinearRegression, Lasso, LassoCV, Ridge, \
↳ RidgeCV, ElasticNet, ElasticNetCV

import statsmodels.api as sm
import statsmodels.formula.api as smf
import patsy

import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
sns.set_style('whitegrid')
```

1.0.1 Section 1: DataFrame Loading, Cleaning

```
[2]: # Returns you all the variables in the current environment

# dir()

# Refer to this link: https://stackoverflow.com/questions/633127/
↳ viewing-all-defined-variables
# To find out out to view all defined variables in the current environment
```

```
[3]: df_main = pd.read_csv('cars.csv')
df_main.sample(5)
```

```
[3]:
```

	manufacturer_name	model_name	transmission	body_type	\
15099	Toyota	innova	Số tay	MPV	
1338	Mercedes-Benz	s-class	Số tự động	Sedan	
19282	Toyota	vios	Số tay	Sedan	
20329	Kia	cerato	Số tự động	Sedan	
2829	Ford	ranger	Số tự động	Pick-up	Truck

	production_year	number_of_seat	odometer_value	color_body	engine_fuel	\
15099	2015	7	170000.0	NaN	Xăng	
1338	2018	5	20000.0	Đen	Xăng	
19282	2010	5	98500.0	Bạc	Xăng	
20329	2010	5	108000.0	Đen	Xăng	
2829	2017	5	55000.0	Màu khác	Diesel	

	engine_type	engine_capacity	drivetrain	price	\
15099	NaN	NaN	NaN	4.900000e+08	
1338	NaN	NaN	RFD - Dẫn động cầu sau	4.539000e+09	
19282	NaN	NaN	FWD - Dẫn động cầu trước	3.000000e+08	
20329	NaN	NaN	FWD - Dẫn động cầu trước	3.800000e+08	
2829	NaN	NaN	4WD - Dẫn động 4 bánh	7.900000e+08	

	url
15099	http://oto.com.vn/mua-ban-xe-toyota-innova-ha...
1338	http://oto.com.vn/mua-ban-xe-mercedes-benz-s-c...
19282	http://oto.com.vn/mua-ban-xe-toyota-vios-phu-t...
20329	http://oto.com.vn/mua-ban-xe-kia-cerato-ha-noi...
2829	http://oto.com.vn/mua-ban-xe-ford-ranger-bac-n...

```
[4]: df_main.columns
```

```
[4]: Index(['manufacturer_name', 'model_name', 'transmission', 'body_type',
        'production_year', 'number_of_seat', 'odometer_value', 'color_body',
        'engine_fuel', 'engine_type', 'engine_capacity', 'drivetrain', 'price',
        'url'],
        dtype='object')
```

```
[5]: df_main.describe()
```

```
[5]:
```

	production_year	number_of_seat	odometer_value	engine_type	\
count	20540.000000	20540.000000	1.719800e+04	0.0	
mean	2013.443427	5.433009	7.035604e+04	NaN	
std	4.853385	1.231451	1.859967e+05	NaN	
min	1965.000000	0.000000	1.000000e+00	NaN	
25%	2010.000000	5.000000	3.000000e+04	NaN	
50%	2015.000000	5.000000	5.500000e+04	NaN	
75%	2017.000000	5.000000	1.080000e+05	NaN	
max	2019.000000	47.000000	1.800000e+07	NaN	

	engine_capacity	price
count	1138.000000	2.054000e+04
mean	1591.660808	1.455116e+09
std	491.286989	3.845170e+09
min	20.000000	0.000000e+00

25%	1496.000000	3.700000e+08
50%	1496.000000	4.760000e+08
75%	1496.000000	8.800000e+08
max	5700.000000	1.790000e+10

```
[6]: df_clean = df_main.drop(['color_body',
                             'engine_fuel', 'engine_type', 'engine_capacity', 'drivetrain',
                             'url'],axis=1) # Dropping columns that I used for my own reference

# We see that we have NA entries in this dataset.
# We want to drop these NA values or they will pose problems for us later
# The null values can be attributed to the fact that some data is not keyed in
# the listing itself,
# or formatting issues due to the varying ways of which people organize the
# information of the car in a single listing

df_clean.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20540 entries, 0 to 20539
Data columns (total 8 columns):
manufacturer_name    20540 non-null object
model_name           20540 non-null object
transmission         19288 non-null object
body_type            20539 non-null object
production_year      20540 non-null int64
number_of_seat       20540 non-null int64
odometer_value       17198 non-null float64
price                20540 non-null float64
dtypes: float64(2), int64(2), object(4)
memory usage: 1.3+ MB
```

```
[7]: # We now have 20540 rows
```

```
df_clean.dropna(inplace=True)
df_clean.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 17027 entries, 0 to 20539
Data columns (total 8 columns):
manufacturer_name    17027 non-null object
model_name           17027 non-null object
transmission         17027 non-null object
body_type            17027 non-null object
production_year      17027 non-null int64
number_of_seat       17027 non-null int64
odometer_value       17027 non-null float64
```

```
price          17027 non-null float64
dtypes: float64(2), int64(2), object(4)
memory usage: 1.2+ MB
```

```
[8]: # Taking a look at our data
df_clean.sample(5)
```

```
[8]:      manufacturer_name model_name transmission body_type production_year \
5703      Rolls-Royce      phantom    Số tự động      Sedan            2011
11748           Toyota      camry    Số tự động      Sedan            2007
12734           Toyota      vios      Số tay      Sedan            2010
1182    Mercedes-Benz    c-class    Số tự động      Sedan            2016
18897           Toyota      camry    Số tự động      Sedan            2016

      number_of_seat  odometer_value      price
5703                4        30000.0  1.790000e+10
11748                5       150000.0  4.320000e+08
12734                5        98500.0  3.000000e+08
1182                5        45000.0  1.090000e+09
18897                5        36000.0  9.800000e+08
```

```
[9]: print(df_clean.columns, '\n', len(df_clean.columns))

# We have 12 features in our columns
```

```
Index(['manufacturer_name', 'model_name', 'transmission', 'body_type',
      'production_year', 'number_of_seat', 'odometer_value', 'price'],
      dtype='object')
8
```

1.0.2 Section 2: Data Categorizing

1.1 Section 2.1: One-hot encoding TRANSMISSION Column

```
[10]: # Here, we see that there only two options for transmission - Auto or Manual (i.
      ↪e., Auto or not).
      # Therefore, we can do 1-hot encoding for this

df_clean['transmission'].value_counts()
```

```
[10]: Số tự động      11396
Số tay      5625
Số hỗn hợp      6
Name: transmission, dtype: int64
```

```
[11]: # Transmission conversion -> 1 for auto, 0 for manual
```

```
df_clean['transmission_convert'] = df_clean['transmission'].apply(lambda x: 1
↳ if x == 'Số tự động' else 0)
df_clean.drop('transmission',axis=1,inplace=True)
df_clean.rename(columns={'transmission_convert':"transmission"}, inplace=True)
↳ # Renaming column back
df_clean.sample(5)
```

```
[11]:      manufacturer_name model_name  body_type  production_year \
7282          Toyota  fortuner      SUV      2019
8927      Rolls-Royce  phantom      Sedan      2011
8304          Kia    cerato      Sedan      2010
17594         Toyota    vios      Sedan      2015
20164          Kia    morning  Hatchback      2019

      number_of_seat  odometer_value      price  transmission
7282              7         7000.0  1.030000e+09              1
8927              4        30000.0  1.790000e+10              1
8304              5       108000.0  3.800000e+08              1
17594             5        62000.0  3.700000e+08              0
20164             5         9612.0  3.700000e+08              1
```

```
[12]: indexNames = df_clean[ df_clean['price'] == 0 ].index
df_clean.drop(indexNames , inplace=True)
df_clean.describe()
```

```
[12]:      production_year  number_of_seat  odometer_value      price \
count      17026.000000      17026.000000      17026.000000  1.702600e+04
mean        2014.233819         5.380242      68174.348878  1.689566e+09
std          4.207260         1.169015      57817.070771  4.178914e+09
min         1965.000000         2.000000         1.000000  3.600000e+07
25%         2010.000000         5.000000      30000.000000  3.700000e+08
50%         2016.000000         5.000000      55000.000000  4.900000e+08
75%         2018.000000         5.000000      108000.000000  8.850000e+08
max         2019.000000        47.000000     1000000.000000  1.790000e+10

      transmission
count      17026.000000
mean         0.669329
std         0.470468
min         0.000000
25%         0.000000
50%         1.000000
75%         1.000000
max         1.000000
```

```
[13]: # Performing whitespace stripping prior to dtype manipulation
#df_clean['body_type'].apply(str.strip)
```

```
#df_clean['body_type'].apply(str.lstrip)

# Inspection of the type of Vehicles
df_clean['body_type'].value_counts()
df_clean['engine_capacity'].value_counts()
```

```
[13]: Sedan          9885
      SUV           2501
      Hatchback     2264
      MPV           1145
      Pick-up Truck  1111
      Van/Minivan    69
      CUV           32
      Truck         13
      City Car       2
      Sport Car      2
      Coupe          1
      Special Purpose 1
      Name: body_type, dtype: int64
```

```
[14]: df_clean.head()
```

```
[14]:  manufacturer_name  model_name  body_type  production_year  number_of_seat  \
0             Kia      cerato      Sedan          2018           5
1          Toyota    innova      MPV          2016           7
2          Toyota    innova      MPV          2014           8
3          Toyota  corolla-altis    Sedan          2009           5
4             Kia        rio      Sedan          2015           5

      odometer_value      price  transmission
0          15000.0  608000000.0             1
1          151000.0  583000000.0             0
2           82000.0  520000000.0             1
3          127000.0  470000000.0             1
4           70000.0  366000000.0             0
```

1.2 Section 2.2: Datetime conversion

```
[15]: # Converting reg_date to datetime, and Manufactured year to int

#df_clean['REG_DATE'] = pd.to_datetime(df_clean['REG_DATE'])
df_clean['production_year'] = df_clean['production_year'].astype(int)
df_clean[['production_year']].dtypes
```

```
[15]: production_year    int32
      dtype: object
```

```
[16]: df_clean.dtypes
```

```
[16]: manufacturer_name    object
      model_name          object
      body_type           object
      production_year      int32
      number_of_seat       int64
      odometer_value       float64
      price                float64
      transmission         int64
      dtype: object
```

```
[17]: # ONLY RUN THIS CELL ONCE!
      #df_main['SCRAPE_DATE'] = \
      #pd.to_datetime(df_main['SCRAPE_DATE']).dt.year # Convert scrape date to
      ↪integer to perform operations on them
```

```
[18]: #df_main['SCRAPE_DATE'] # Checking dtype
```

1.2.1 Section 2.2.1: Adding a Car Age Column

```
[19]: # Converting current scrape date from main dataframe to datetime object using
      ↪pandas
      from datetime import date

      # Obtaining number of years from year of manufacture to current year (metric
      ↪for how new the car is)
      current_year = date.today().year
      df_clean['car_age'] = current_year - df_clean['production_year'] # Obtaining
      ↪values for age of car
      df_clean['car_age'].astype(int)
```

```
[19]: 0          2
      1          4
      2          6
      3         11
      4          5
      ..
      20535       3
      20536       3
      20537       9
      20538       1
      20539      13
      Name: car_age, Length: 17026, dtype: int32
```

```
[20]: # Rearranging Columns
#df_clean = df_clean[['BRAND', 'PRICE', 'DEPRE_VALUE_PER_YEAR', 'MILEAGE_KM',
→ 'COE_FROM_SCRAPE_DATE', 'DAYS_OF_COE_LEFT',
#
→ 'REG_DATE', 'MANUFACTURED_YEAR', 'CAR_AGE',
→ 'DEREG_VALUE_FROM_SCRAPE_DATE', 'OMV', 'ARF',
#
→ 'ENGINE_CAPACITY_CC', 'ROAD_TAX_PER_YEAR',
→ 'CURB_WEIGHT_KG',
#
→ 'NO_OF_OWNERS', 'VEHICLE_TYPE', 'TRANSMISSION']]
df_clean.head()
```

```
[20]: manufacturer_name    model_name body_type production_year number_of_seat \
0          Kia          cerato      Sedan            2018             5
1        Toyota        innova      MPV              2016             7
2        Toyota        innova      MPV              2014             8
3        Toyota  corolla-altis      Sedan            2009             5
4          Kia           rio       Sedan            2015             5

odometer_value    price transmission car_age
0         15000.0  608000000.0           1         2
1         151000.0  583000000.0           0         4
2          82000.0  520000000.0           1         6
3         127000.0  470000000.0           1        11
4          70000.0  366000000.0           0         5
```

1.3 Section 2.3: BODY_TYPE To Dummy Variables

```
[21]: # Making Dummy Variables out of Vehicle Types:

x_vehtype_dummy = patsy.
→ dmatrix('body_type', data=df_clean, return_type='dataframe')
x_vehtype_dummy.head()

# Do we drop the "Intercept" column?
```

```
[21]: Intercept  body_type[T.City Car]  body_type[T.Coupe] \
0          1.0             0.0             0.0
1          1.0             0.0             0.0
2          1.0             0.0             0.0
3          1.0             0.0             0.0
4          1.0             0.0             0.0

body_type[T.Hatchback]  body_type[T.MPV]  body_type[T.Pick-up Truck] \
0             0.0             0.0             0.0
1             0.0             1.0             0.0
2             0.0             1.0             0.0
3             0.0             0.0             0.0
```


4	0.0	0.0	0.0
---	-----	-----	-----

	body_type[T.SUV]	body_type[T.Sedan]	body_type[T.Special Purpose]	\
0	0.0	1.0	0.0	
1	0.0	0.0	0.0	
2	0.0	0.0	0.0	
3	0.0	1.0	0.0	
4	0.0	1.0	0.0	

	body_type[T.Sport Car]	body_type[T.Truck]	body_type[T.Van/Minivan]
0	0.0	0.0	0.0
1	0.0	0.0	0.0
2	0.0	0.0	0.0
3	0.0	0.0	0.0
4	0.0	0.0	0.0

```
[22]: #df_clean_memory = df_clean.memory_usage(index=True).sum()
x_vehtype_dummy_memory = x_vehtype_dummy.memory_usage(index=True).sum()
#print("df_clean_memory dataset uses ",df_clean_memory/ 1024**2," MB")
#print("x_vehtype_dummy_memory dataset uses ",x_vehtype_dummy_memory/ 1024**2,"
↪MB")
df_clean2 = df_clean.join(x_vehtype_dummy)
df_clean2
```

```
[22]: manufacturer_name  model_name  body_type  production_year  \
0          Kia          cerato      Sedan      2018
1        Toyota          innova      MPV      2016
2        Toyota          innova      MPV      2014
3        Toyota  corolla-altis      Sedan      2009
4          Kia             rio      Sedan      2015
...
20535        Ford          ranger  Pick-up Truck      2017
20536        Toyota          fortuner      SUV      2017
20537    Rolls-Royce          phantom      Sedan      2011
20538        Toyota          fortuner      SUV      2019
20539        Toyota          camry      Sedan      2007
```

	number_of_seat	odometer_value	price	transmission	car_age	\
0	5	15000.0	6.080000e+08	1	2	
1	7	151000.0	5.830000e+08	0	4	
2	8	82000.0	5.200000e+08	1	6	
3	5	127000.0	4.700000e+08	1	11	
4	5	70000.0	3.660000e+08	0	5	
...	
20535	5	55000.0	7.900000e+08	1	3	
20536	7	55000.0	8.850000e+08	0	3	
20537	4	30000.0	1.790000e+10	1	9	

20538	7	7000.0	1.030000e+09	1	1
20539	5	150000.0	4.320000e+08	1	13

	Intercept	...	body_type[T.Coupe]	body_type[T.Hatchback]	\
0	1.0	...	0.0	0.0	
1	1.0	...	0.0	0.0	
2	1.0	...	0.0	0.0	
3	1.0	...	0.0	0.0	
4	1.0	...	0.0	0.0	
...	
20535	1.0	...	0.0	0.0	
20536	1.0	...	0.0	0.0	
20537	1.0	...	0.0	0.0	
20538	1.0	...	0.0	0.0	
20539	1.0	...	0.0	0.0	

	body_type[T.MPV]	body_type[T.Pick-up Truck]	body_type[T.SUV]	\
0	0.0	0.0	0.0	
1	1.0	0.0	0.0	
2	1.0	0.0	0.0	
3	0.0	0.0	0.0	
4	0.0	0.0	0.0	
...	
20535	0.0	1.0	0.0	
20536	0.0	0.0	1.0	
20537	0.0	0.0	0.0	
20538	0.0	0.0	1.0	
20539	0.0	0.0	0.0	

	body_type[T.Sedan]	body_type[T.Special Purpose]	\
0	1.0	0.0	
1	0.0	0.0	
2	0.0	0.0	
3	1.0	0.0	
4	1.0	0.0	
...	
20535	0.0	0.0	
20536	0.0	0.0	
20537	1.0	0.0	
20538	0.0	0.0	
20539	1.0	0.0	

	body_type[T.Sport Car]	body_type[T.Truck]	body_type[T.Van/Minivan]
0	0.0	0.0	0.0
1	0.0	0.0	0.0
2	0.0	0.0	0.0
3	0.0	0.0	0.0

4	0.0	0.0	0.0
...
20535	0.0	0.0	0.0
20536	0.0	0.0	0.0
20537	0.0	0.0	0.0
20538	0.0	0.0	0.0
20539	0.0	0.0	0.0

[17026 rows x 21 columns]

1.4 Section 2.4: Car Brand Categorization. Includes:

- Splitting them into Dummy Variables
- Indexing them into price range categories (perhaps better metric over vehicle types)
- Converting lesser-known brands into “others”

[23]: *# Renaming Brand Names to their actual names*

```
#df_clean2.loc[df_clean2['BRAND'] == 'Aston', 'BRAND'] = 'Aston Martin'
#df_clean2.loc[df_clean2['BRAND'] == 'Land', 'BRAND'] = 'Land Rover'
#df_clean2.loc[df_clean2['BRAND'] == 'Alfa', 'BRAND'] = 'Alfa Romeo'
#df_clean2.head()
```

[24]: *# Cleaning whitespaces from the values in "Brand" to prevent any messup later*
#df_clean2['manufacturer_name'].apply(str.strip)

```
# Checking the number of brands in the dataset
print("# Of rows in DataFrame in Brands Column:\n", df_clean2.loc[:
    ↪, 'manufacturer_name'])
print("\nValue Counts of Brands:\n", df_clean2.loc[:, 'manufacturer_name'].
    ↪value_counts())
print("\n# of Brands:", len(df_clean2.loc[:, 'manufacturer_name']).
    ↪value_counts()))
```

```
# New Column ATAS
# New Column Budget Cars
# Top 20 brands
# Am I comfortable with grouping uncommon cars into others?
```

Of rows in DataFrame in Brands Column:

0	Kia
1	Toyota
2	Toyota
3	Toyota
4	Kia
...	
20535	Ford

```
20536      Toyota
20537  Rolls-Royce
20538      Toyota
20539      Toyota
Name: manufacturer_name, Length: 17026, dtype: object
```

Value Counts of Brands:

Toyota	8693
Kia	2228
Mercedes-Benz	1207
Hyundai	1182
Ford	1178
Mazda	1098
Rolls-Royce	1046
Lexus	70
Honda	55
Chevrolet	54
Mitsubishi	32
LandRover	28
BMW	25
Nissan	23
Audi	23
Daewoo	16
Suzuki	13
Cadillac	7
Porsche	7
Bentley	5
Isuzu	4
Acura	4
Jaguar	4
Peugeot	3
Renault	3
Zotye	2
Cửu Long	2
Volkswagen	2
Mini	2
Maserati	1
Ssangyong	1
Samco	1
Subaru	1
Thaco	1
Infiniti	1
Hino	1
Fuso	1
FAW	1
Fiat	1

```
Name: manufacturer_name, dtype: int64
```

of Brands: 39

1.5 Section 3: Data Visualization

- EDA
- Correlation Matrix
- Pairplots

1.6 Section 3.1: Preliminary Correlation Exploration

1.6.1 Section 3.1.1: Analysis without Car Brands and Vehicle Types for Feature Dropping

```
[25]: df_clean2.columns
```

```
[25]: Index(['manufacturer_name', 'model_name', 'body_type', 'production_year',  
          'number_of_seat', 'odometer_value', 'price', 'transmission', 'car_age',  
          'Intercept', 'body_type[T.City Car]', 'body_type[T.Coupe]',  
          'body_type[T.Hatchback]', 'body_type[T.MPV]',  
          'body_type[T.Pick-up Truck]', 'body_type[T.SUV]', 'body_type[T.Sedan]',  
          'body_type[T.Special Purpose]', 'body_type[T.Sport Car]',  
          'body_type[T.Truck]', 'body_type[T.Van/Minivan]'],  
          dtype='object')
```

```
[26]: df_price_no_brands = df_clean2[['model_name', 'body_type', 'production_year',  
          'number_of_seat', 'odometer_value', 'price', 'transmission',  
          'car_age']]  
df_price_no_brands.head()
```

```
[26]:
```

	model_name	body_type	production_year	number_of_seat	odometer_value	\
0	cerato	Sedan	2018	5	15000.0	
1	innova	MPV	2016	7	151000.0	
2	innova	MPV	2014	8	82000.0	
3	corolla-altis	Sedan	2009	5	127000.0	
4	rio	Sedan	2015	5	70000.0	

	price	transmission	car_age
0	608000000.0	1	2
1	583000000.0	0	4
2	520000000.0	1	6
3	470000000.0	1	11
4	366000000.0	0	5

```
[27]: # Corr Matrix  
df_price_no_brands.corr()
```

```
[27]:
```

	production_year	number_of_seat	odometer_value	price	\
production_year	1.000000	0.236846	-0.722899	-0.154397	
number_of_seat	0.236846	1.000000	0.112451	-0.277573	
odometer_value	-0.722899	0.112451	1.000000	-0.207174	
price	-0.154397	-0.277573	-0.207174	1.000000	
transmission	-0.101326	-0.298632	-0.142373	0.199955	
car_age	-1.000000	-0.236846	0.722899	0.154397	

	transmission	car_age
production_year	-0.101326	-1.000000
number_of_seat	-0.298632	-0.236846
odometer_value	-0.142373	0.722899
price	0.199955	0.154397
transmission	1.000000	0.101326
car_age	0.101326	1.000000

```
[28]: # How each feature relates to price
df_price_no_brands.corr()['price'].sort_values(ascending=False)
```

```
[28]: price          1.000000
transmission    0.199955
car_age         0.154397
production_year -0.154397
odometer_value  -0.207174
number_of_seat  -0.277573
Name: price, dtype: float64
```

```
[29]: # Corr Matrix Heatmap Visualization

sns.set(style="white")

# Generate a mask for the upper triangle
mask = np.zeros_like(df_price_no_brands.corr(), dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

# Set up the matplotlib figure to control size of heatmap
fig, ax = plt.subplots(figsize=(15,15))

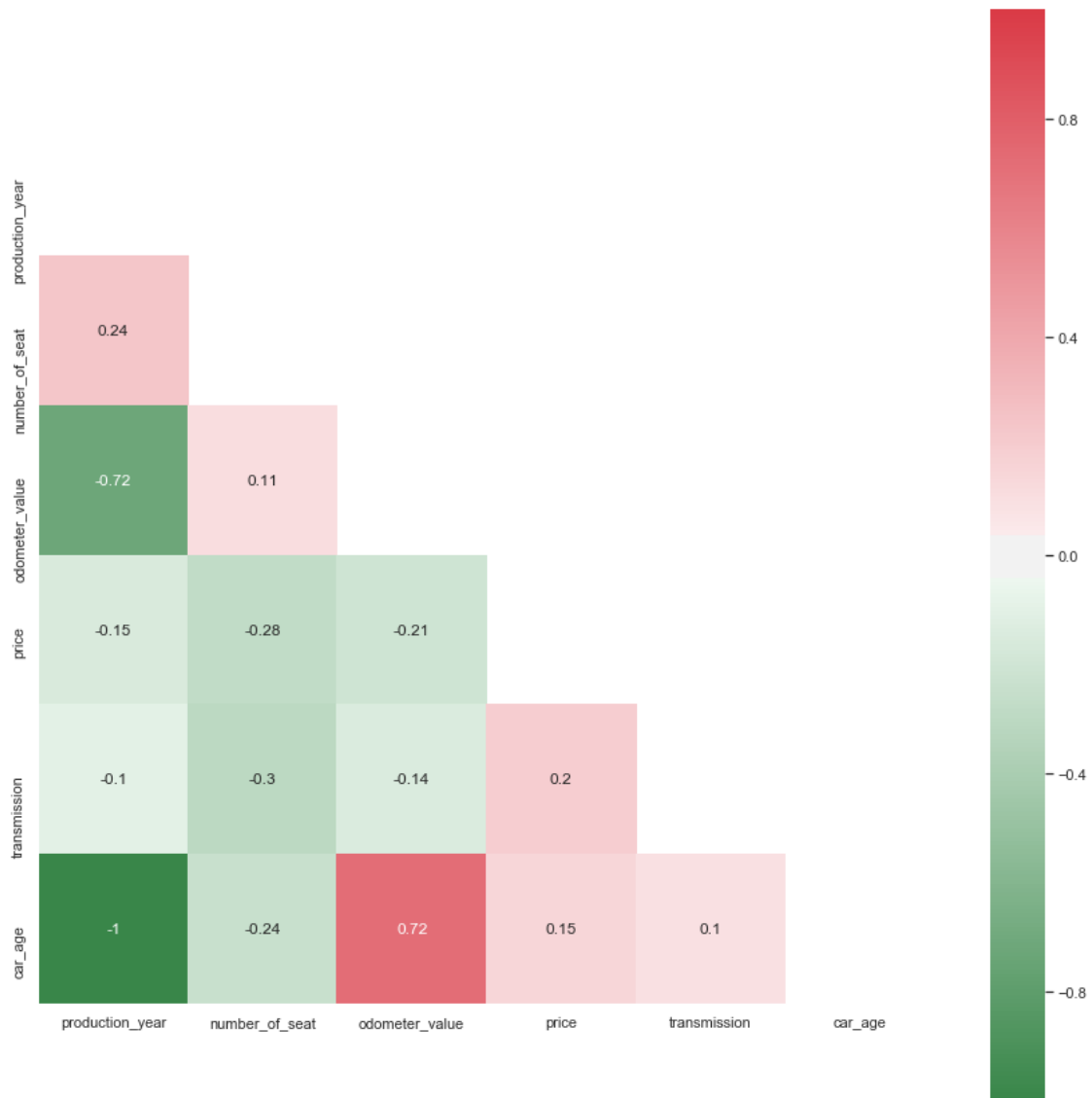
# Create a custom color palette
cmap = \
sns.diverging_palette(133, 10,
                      as_cmap=True) # as_cmap returns a matplotlib colormap
    ↳ object rather than a list of colors
# Green = Good (low correlation), Red = Bad (high correlation) between the
    ↳ independent variables

# Plot the heatmap
```

```
sns.heatmap(df_price_no_brands.corr(), mask=mask, annot=True,
             square=True, cmap=cmap, vmin=-1, vmax=1, ax=ax);

# Prevent Heatmap Cut-Off Issue
bottom, top = ax.get_ylim()
ax.set_ylim(bottom + 0.5, top - 0.5)
```

[29]: (6.0, 0.0)



1.6.2 Section 3.1.2: Removing Independent Variables with High Correlation to each other

```
[30]: df_price_no_brands.drop(['production_year', 'body_type'], axis=1, inplace=True)
df_price_no_brands.columns
```

D:\Application\Anaconda2\envs\py3\lib\site-packages\pandas\core\frame.py:4117:

SettingWithCopyWarning:

A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
errors=errors,

```
[30]: Index(['model_name', 'number_of_seat', 'odometer_value', 'price',
          'transmission', 'car_age'],
          dtype='object')
```

1.6.3 Section 3.1.3: Re-Visualizing New Correlation Matrix (with a few features dropped)

From the above Corr Matrix, we can observe that a few Independent Variables are highly correlated with each other. Interestingly, this makes sense due to how a few of the independent variables are calculated. Therefore, some of these features can be dropped.

1. **OMV and ARF:** The Additional Registration Fee (ARF) is a tax imposed upon registration of a vehicle. It is calculated based on a percentage of the Open Market Value (OMV) of the vehicle. From the correlation matrix, they have a correlation of **0.96**. Therefore, we can just keep one of this. Therefore, **ARF** will be dropped.

2. **Engine CC and Road Tax:** Road tax increases as engine CC increases. Therefore, having Road Tax in it would be introducing multicollinearity into the model, since both explain the same thing anyway. From the correlation matrix, they have a correlation of **0.97**. Therefore, **road tax** column will be dropped.

3. **Manufactured Year and Car Age:** Obviously, manufactured year can be dropped, since Car Age is derived from year of manufacture. And since car age is more intuitive, **Manufactured Year** column will be dropped. From the correlation matrix, they have a correlation of **-1**.

4. **Dereg Value:**

The deregistration value of a vehicle is amount one will get back from the government upon deregistering the vehicle for use in Singapore. It is the sum of the COE rebate and the PARF rebate. The COE rebate is a percentage of the original COE value. The PARF rebate is a percentage of the original OMV (Open Market Value). Therefore, the **dereg value will be dropped**. From the corr matrix, OMV and dereg value have a correlation of **0.91**.

5. **Depreciation:** Annual depreciation is the amount the owner loses on the value of the vehicle per year based on the assumption that the vehicle is deregistered only at the end of its 10-yr COE life-span. The value of the car body has not been taken into account. The formula used to derive the figure is as follows:

Depreciation = (List Price - Minimum PARF value) ÷ remaining years of COE

The minimum Preferential Additional Registration Fee (PARF) value is a sum of money that the vehicle owner gets for deregistering his vehicle at the end of its 10-yr COE. It is a function of the Open Market Value (OMV) or Additional Registration Fee (ARF). Commercial vehicles and cars with renewed COE have no minimum PARF value. Depreciation and OMV have a correlation of **0.91**. **Depreciation** will be dropped.

```
[31]: # Re-visualizing the correlation matrix

sns.set(style="white")

# Creating the data
data = df_price_no_brands.corr()

# Generate a mask for the upper triangle
mask = np.zeros_like(data, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

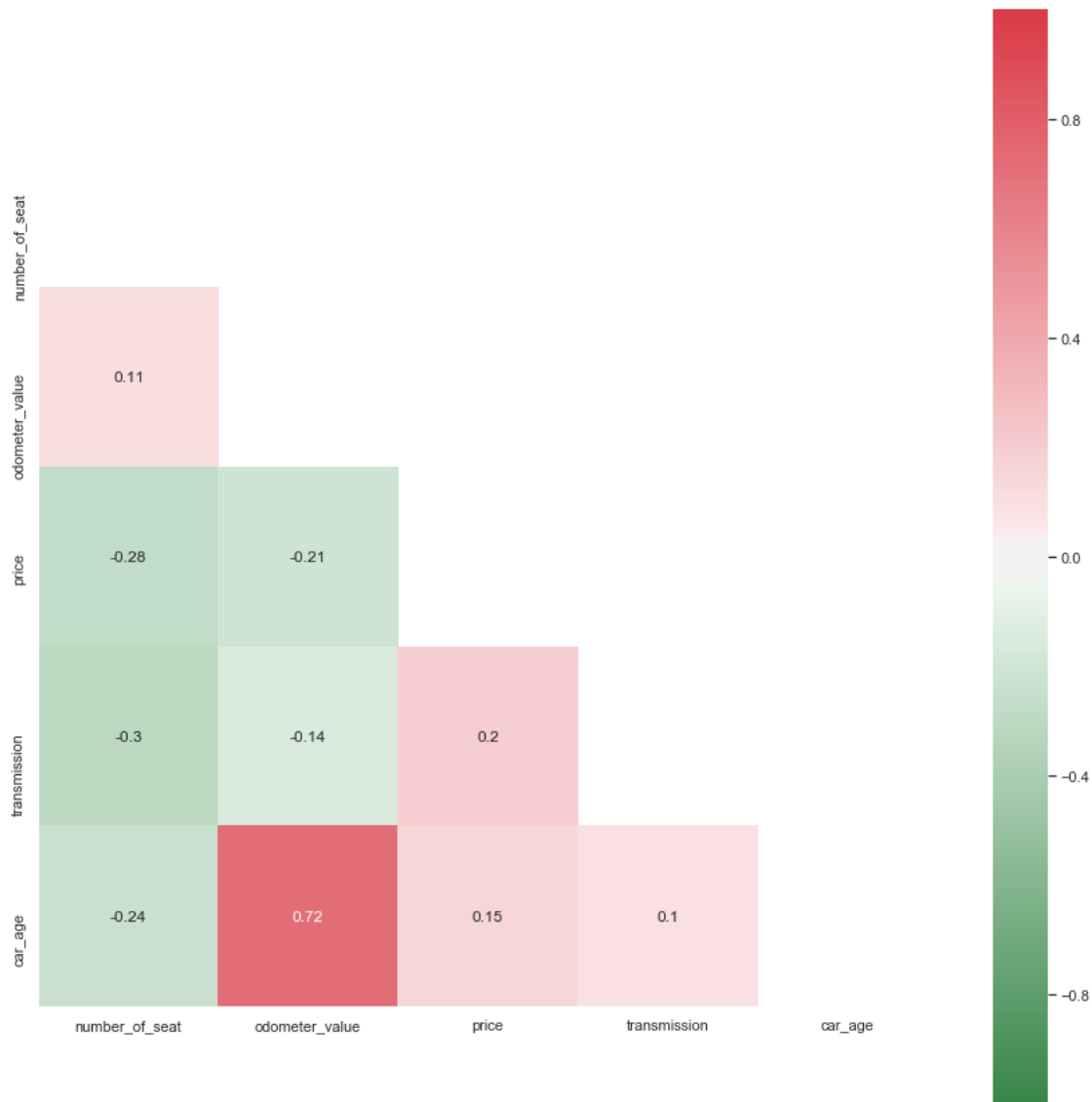
# Set up the matplotlib figure to control size of heatmap
fig, ax = plt.subplots(figsize=(15,15))

# Create a custom color palette
cmap = \
sns.diverging_palette(133, 10,
                      as_cmap=True) # as_cmap returns a matplotlib colormap
↳ object rather than a list of colors
# Green = Good (low correlation), Red = Bad (high correlation) between the
↳ independent variables

# Plot the heatmap
sns.heatmap(data, mask=mask, annot=True,
            square=True, cmap=cmap, vmin=-1, vmax=1, ax=ax);

# Prevent Heatmap Cut-Off Issue
bottom, top = ax.get_ylim()
ax.set_ylim(bottom + 0.5, top - 0.5)
```

```
[31]: (5.0, 0.0)
```



```
[32]: # Correlations of the independent variables (features) to dependent variable ↵
      ↪(target, price)
      df_price_no_brands.corr()['price'].sort_values(ascending=False)
```

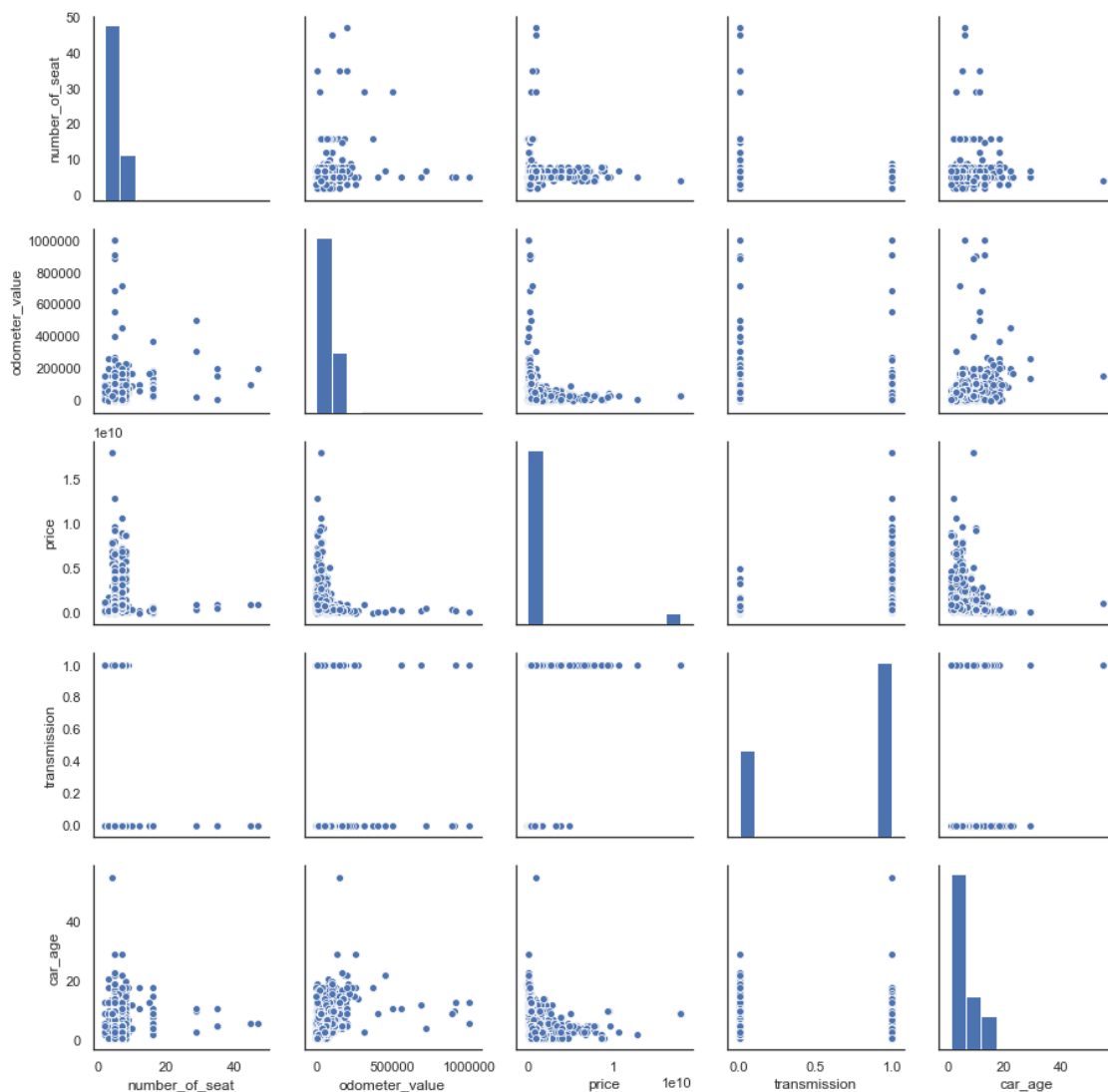
```
[32]: price          1.000000
      transmission    0.199955
      car_age         0.154397
      odometer_value  -0.207174
      number_of_seat  -0.277573
      Name: price, dtype: float64
```

Section 3.1.3.1: Pairplot after Feature Selection

```
[33]: # Performing a pairplot to visualize the data trends of the variables

# We can see that price and mileage hold a negative linear relationship
# COE from the scrape date doesn't seem to have a very clear relationship here
# Days of COE seems to have a slight linear r/s
# Car age doesn't seem to have a very distinct relationship here. But
    ↳ generally, the younger the car, the higher the price
# OMV has a clear increasing linear rs with price with price
# Engine capacity seems to also have a increasing linear r/s with price, with
    ↳ a few outliers in the center
# Perhaps it's because a lot of the higher-priced cars (higher brands) are
    ↳ produced in that engine capacity range?
# Curb weight seems to have a linear r/s too.

sns.pairplot(df_price_no_brands);
```



```
[34]: df_price_no_brands.columns
```

```
[34]: Index(['model_name', 'number_of_seat', 'odometer_value', 'price',  
        'transmission', 'car_age'],  
        dtype='object')
```

1.6.4 Section 3.2: Preliminary Model fitting to check R^2 Value and $P > |t|$ values of Price and the leftover Independent Variables

```
[35]: # Slicing Data into Independent Variables (Features) and Dependent Variable  
      ↪ (Target)  
      #df_price_no_brands.dropna(inplace=True)  
      X = df_price_no_brands[ ['number_of_seat', 'odometer_value', 'transmission',  
                              'car_age'] ].astype(float)  
      X = sm.add_constant(X)  
      y = df_price_no_brands['price'].astype(int)
```

```
C:\Users\nam.nguyen\AppData\Roaming\Python\Python37\site-  
packages\numpy\core\fromnumeric.py:2389: FutureWarning: Method .ptp is  
deprecated and will be removed in a future version. Use numpy.ptp instead.  
    return ptp(axis=axis, out=out, **kwargs)
```

```
[36]: X.head()
```

```
[36]:
```

	const	number_of_seat	odometer_value	transmission	car_age
0	1.0	5.0	15000.0	1.0	2.0
1	1.0	7.0	151000.0	0.0	4.0
2	1.0	8.0	82000.0	1.0	6.0
3	1.0	5.0	127000.0	1.0	11.0
4	1.0	5.0	70000.0	0.0	5.0

```
[37]: y.head()
```

```
[37]: 0    608000000  
      1    583000000  
      2    520000000  
      3    470000000  
      4    366000000  
      Name: price, dtype: int32
```

```
[38]: # model / fit / summarize  
      import statsmodels.api as sm  
  
      lsm = sm.OLS(y, X)
```

```
results = lsm.fit()
results.summary()
```

```
[38]: <class 'statsmodels.iolib.summary.Summary'>
```

```
"""
                                OLS Regression Results
=====
Dep. Variable:                price    R-squared:                0.295
Model:                        OLS      Adj. R-squared:           0.295
Method:                       Least Squares    F-statistic:            1778.
Date:                         Tue, 07 Jan 2020    Prob (F-statistic):      0.00
Time:                         10:49:19    Log-Likelihood:         -3.6918e+05
No. Observations:             17026    AIC:                    7.384e+05
Df Residuals:                 17021    BIC:                    7.384e+05
Df Model:                     4
Covariance Type:              nonrobust
=====
==
                                coef    std err          t      P>|t|      [0.025
0.975]
-----
--
const                5.674e+07    3.02e+07     1.879     0.060    -2.46e+06
1.16e+08
number_of_seat      9.707e+07    4.78e+06    20.301     0.000     8.77e+07
1.06e+08
odometer_value      6404.3877    137.269     46.656     0.000     6135.325
6673.450
transmission         1.404e+08    1.11e+07    12.677     0.000     1.19e+08
1.62e+08
car_age             -1.243e+08    1.92e+06   -64.822     0.000    -1.28e+08
-1.21e+08
=====
Omnibus:                7813.001    Durbin-Watson:           2.046
Prob(Omnibus):           0.000    Jarque-Bera (JB):        45536.917
Skew:                    -2.166    Prob(JB):                 0.00
Kurtosis:                9.740    Cond. No.                 5.70e+05
=====

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 5.7e+05. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

1.6.5 Section 3.2.1: Optimizing R^2 Value

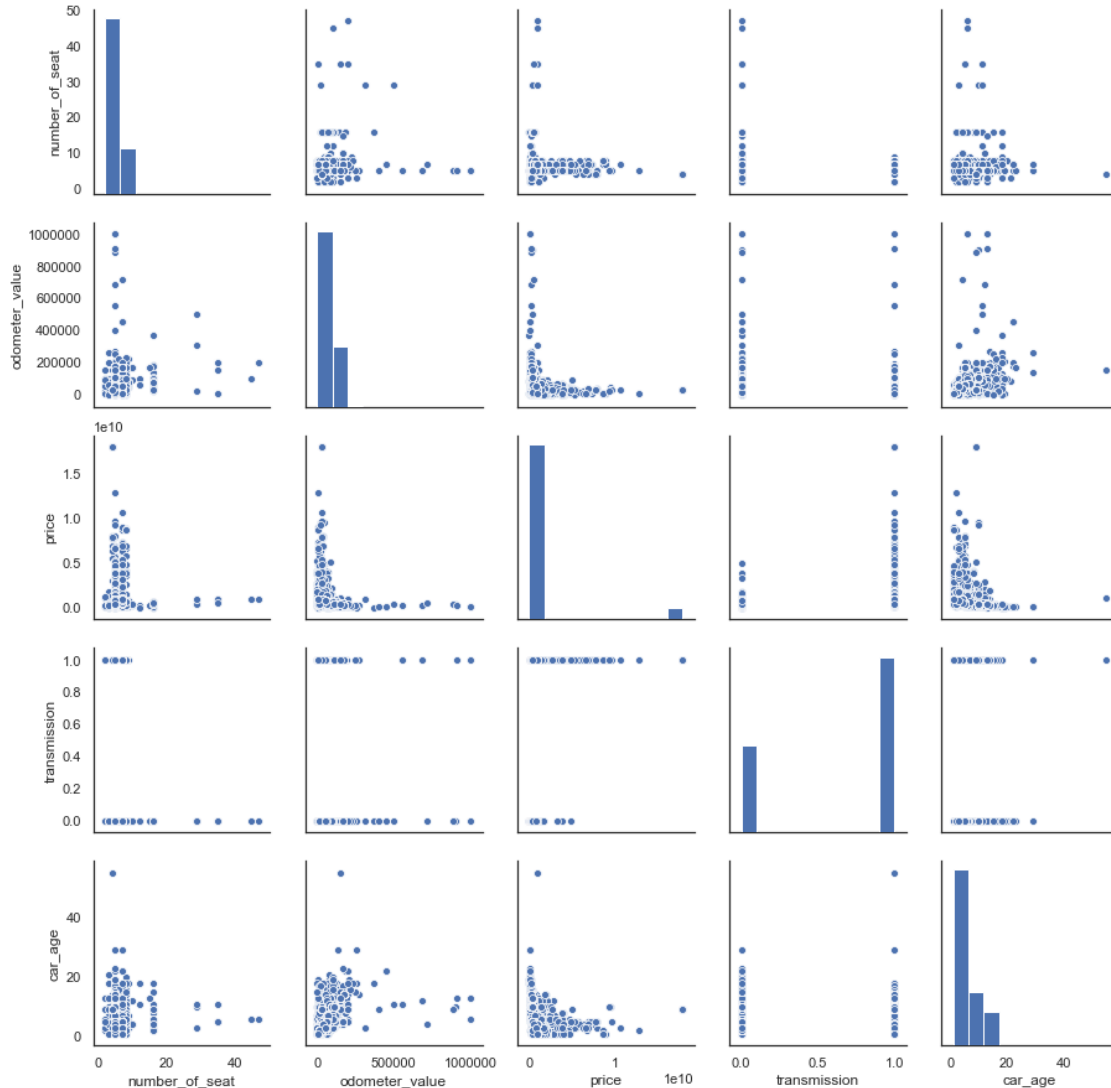
Section 3.2.1.1: Checking Distributions & Pairplots of all Variables

Pairplot of Price vs Independent Variables (without any transformation)

```
[39]: # Performing a pairplot to visualize the data trends of the variables

# We can see that price and mileage hold a negative linear relationship
# COE from the scrape date doesn't seem to have a very clear relationship here
# Days of COE seems to have a slight linear r/s
# Car age doesn't seem to have a very distinct relationship here. But ↵
    ↪generally, the younger the car, the higher the price
# OMV has a clear increasing linear rs with price with price
# Engine capacity seems to also have a increasing linear r/s with price, with ↵
    ↪a few outliers in the center
# Perhaps it's because a lot of the higher-priced cars (higher brands) are ↵
    ↪produced in that engine capacity range?
# Curb weight seems to have a linear r/s too.

sns.pairplot(df_price_no_brands);
```



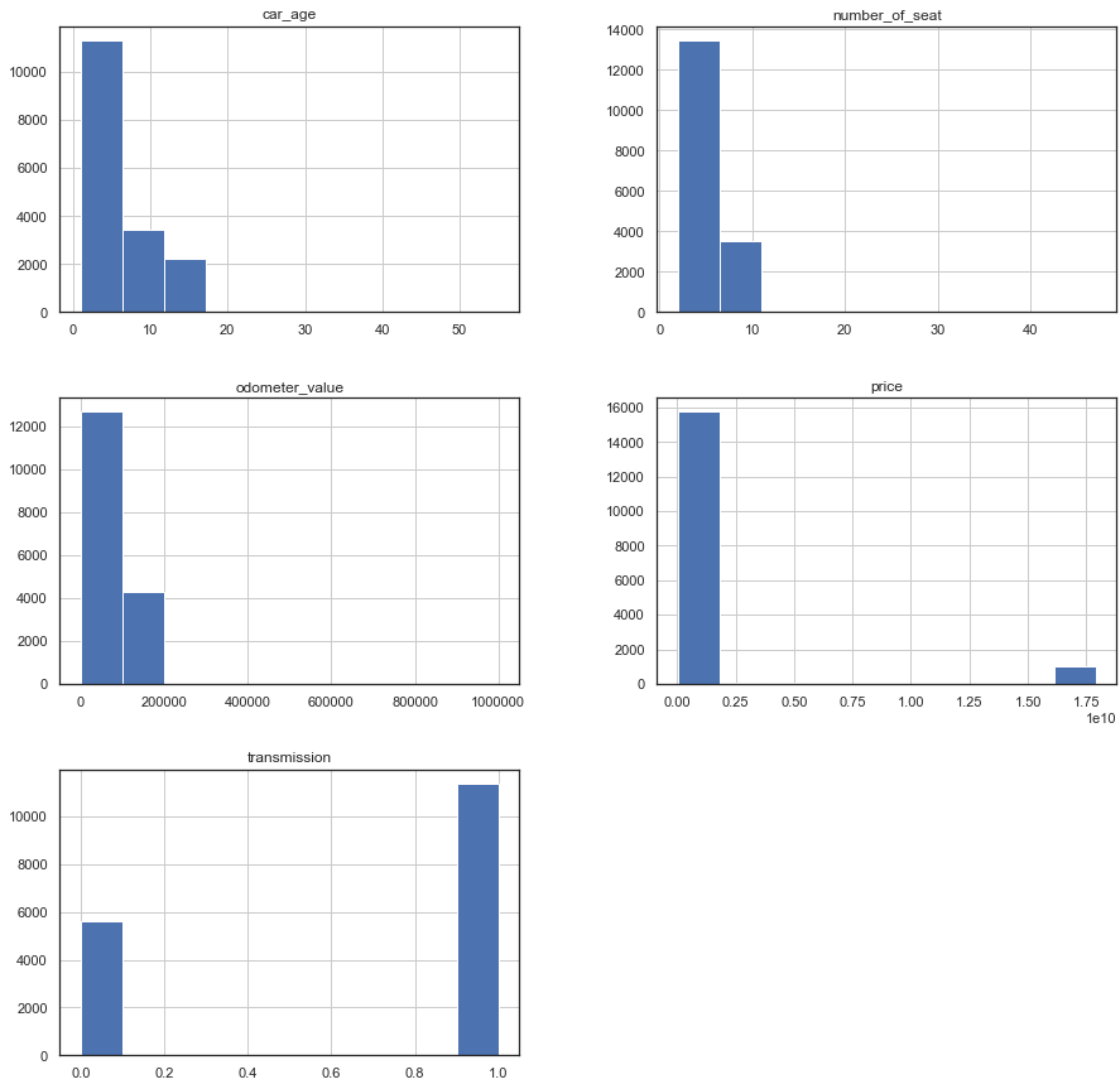
Histogram of all Variables (Columns) in DataFrame

```
[40]: fig, ax = plt.subplots(figsize=(15,15))
      pd.DataFrame.hist(df_price_no_brands,ax=ax)
```

D:\Application\Anaconda2\envs\py3\lib\site-packages\ipykernel_launcher.py:2:
UserWarning: To output multiple subplots, the figure containing the passed axes
is being cleared

```
[40]: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x0000021E6C38D088>,
              <matplotlib.axes._subplots.AxesSubplot object at 0x0000021E6C241888>],
             [<matplotlib.axes._subplots.AxesSubplot object at 0x0000021E6E8EE348>,
              <matplotlib.axes._subplots.AxesSubplot object at 0x0000021E6BBE2E08>],
```

```
[<matplotlib.axes._subplots.AxesSubplot object at 0x0000021E6BBBF0C8>,
 <matplotlib.axes._subplots.AxesSubplot object at 0x0000021E6E983748>]],
 dtype=object)
```

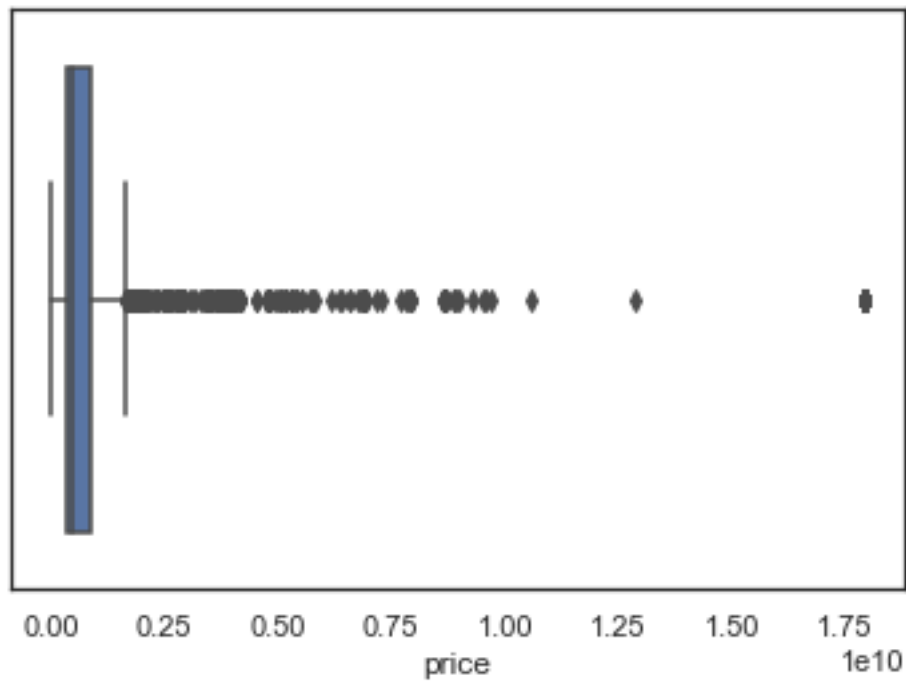


```
[41]: # From the above graphs, it would make sense to apply log transform on the
      ↪ following variables to make them
      # more normally distributed
      # Mileage
      # Engine Cap
      # Price
      # OMV
```


1.7 Distribution of Price

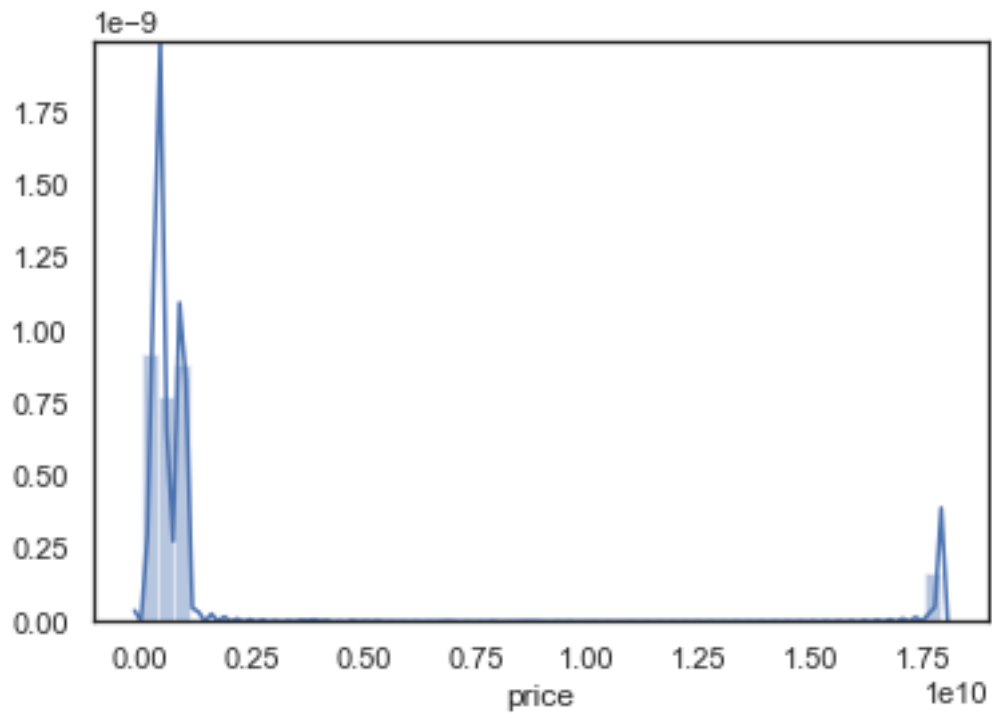
```
[42]: sns.boxplot(df_price_no_brands['price']) #
```

```
[42]: <matplotlib.axes._subplots.AxesSubplot at 0x21e6c633508>
```



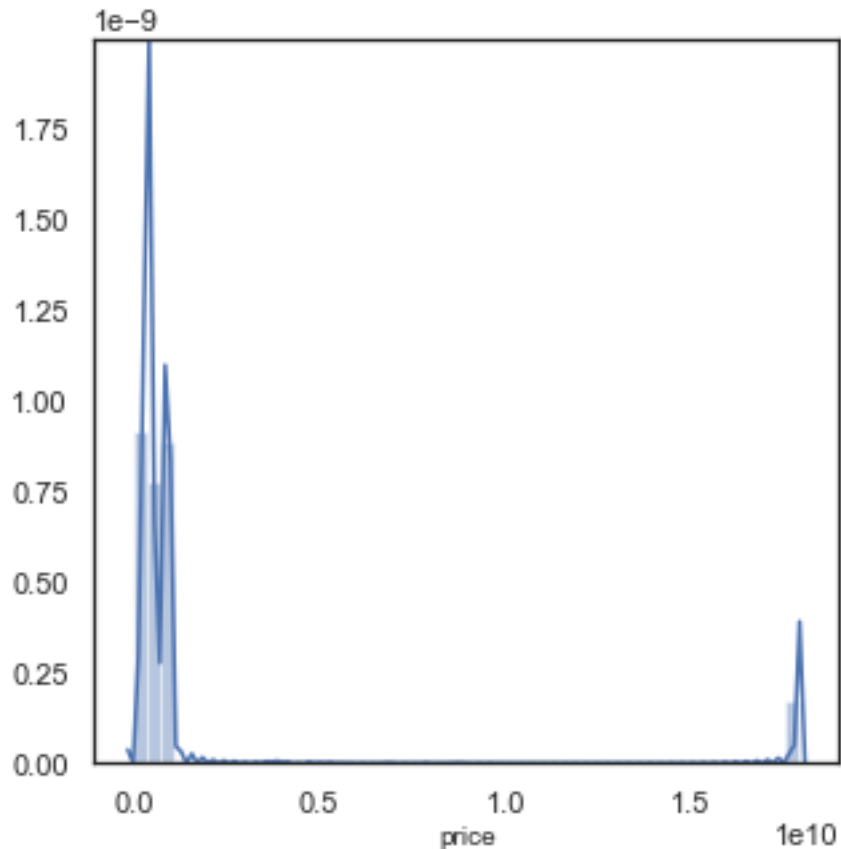
```
[43]: sns.distplot(df_price_no_brands['price']) # Your dependent variable 'must' be
↳ normally distributed
```

```
[43]: <matplotlib.axes._subplots.AxesSubplot at 0x21e6c633208>
```



```
[44]: # We see that price is right-skewed. Therefore, we can try applying a log onto
      ↪ price, then visualize the data again.
```

```
[45]: fig, ax = plt.subplots(figsize=(5,5))
      sns.distplot(df_price_no_brands['price'],ax=ax)
      plt.xlabel('price',size=10)
      plt.savefig("price_no_log.png",transparent=True)
```



1.7.1 Section 3.2.1: Logging Mileage ONLY

```
[46]: # Creating a copy of the dataframe to work log on
df_price_no_brands_only_mileage_logged = df_price_no_brands.copy()
```

```
[47]: # Log Mileage
df_price_no_brands_only_mileage_logged["odometer_value_log"] =
    df_price_no_brands_only_mileage_logged['odometer_value'].apply(np.log)
df_price_no_brands_only_mileage_logged
```

```
[47]:
```

	model_name	number_of_seat	odometer_value	price \
0	cerato	5	15000.0	6.080000e+08
1	innova	7	151000.0	5.830000e+08
2	innova	8	82000.0	5.200000e+08
3	corolla-altis	5	127000.0	4.700000e+08
4	rio	5	70000.0	3.660000e+08
...
20535	ranger	5	55000.0	7.900000e+08
20536	fortuner	7	55000.0	8.850000e+08

20537	phantom	4	30000.0	1.790000e+10
20538	fortuner	7	7000.0	1.030000e+09
20539	camry	5	150000.0	4.320000e+08

	transmission	car_age	odometer_value_log
0	1	2	9.615805
1	0	4	11.925035
2	1	6	11.314475
3	1	11	11.751942
4	0	5	11.156251
...
20535	1	3	10.915088
20536	0	3	10.915088
20537	1	9	10.308953
20538	1	1	8.853665
20539	1	13	11.918391

[17026 rows x 7 columns]

[48]: *# Rearranging columns*

```
df_price_no_brands_only_mileage_logged = \
df_price_no_brands_only_mileage_logged[['price', 'odometer_value_log',
    'car_age', 'transmission']]
```

[49]: `df_price_no_brands_only_mileage_logged.columns`
`df_price_no_brands_only_mileage_logged.describe()`

	price	odometer_value_log	car_age	transmission
count	1.702600e+04	17026.000000	17026.000000	17026.000000
mean	1.689566e+09	10.704260	5.766181	0.669329
std	4.178914e+09	1.053606	4.207260	0.470468
min	3.600000e+07	0.000000	1.000000	0.000000
25%	3.700000e+08	10.308953	2.000000	0.000000
50%	4.900000e+08	10.915088	4.000000	1.000000
75%	8.850000e+08	11.589887	10.000000	1.000000
max	1.790000e+10	13.815511	55.000000	1.000000

[50]: *# Slicing Data into Independent Variables (Features) and Dependent Variable ↪ (Target)*

```
X = df_price_no_brands_only_mileage_logged[ [ 'odometer_value_log',
    'car_age', 'transmission' ] ].astype(float)
X = sm.add_constant(X)
y = df_price_no_brands_only_mileage_logged['price'].astype(int)
```

```
# model / fit / summarize
```

```
import statsmodels.api as sm
```

```
lsm = sm.OLS(y, X)
results = lsm.fit()
results.summary()
```

C:\Users\nam.nguyen\AppData\Roaming\Python\Python37\site-packages\numpy\core\fromnumeric.py:2389: FutureWarning: Method .ptp is deprecated and will be removed in a future version. Use numpy.ptp instead.

```
return ptp(axis=axis, out=out, **kwargs)
```

[50]: <class 'statsmodels.iolib.summary.Summary'>

```
"""
```

OLS Regression Results

```
=====
Dep. Variable:          price    R-squared:                0.225
Model:                  OLS      Adj. R-squared:            0.225
Method:                 Least Squares    F-statistic:          1648.
Date:                  Tue, 07 Jan 2020    Prob (F-statistic):      0.00
Time:                  10:49:30    Log-Likelihood:        -3.6999e+05
No. Observations:      17026    AIC:                   7.400e+05
Df Residuals:          17022    BIC:                   7.400e+05
Df Model:               3
Covariance Type:        nonrobust
=====
=====
```

	coef	std err	t	P> t	[0.025
	0.975]				
const	-2.329e+09	7.29e+07	-31.943	0.000	-2.47e+09
odometer_value_log	3.152e+08	7.2e+06	43.807	0.000	3.01e+08
car_age	-1.217e+08	1.79e+06	-68.188	0.000	-1.25e+08
transmission	7.356e+07	1.15e+07	6.376	0.000	5.09e+07

```
=====
Omnibus:                6905.208    Durbin-Watson:          2.041
Prob(Omnibus):           0.000    Jarque-Bera (JB):       38017.599
Skew:                   -1.886    Prob(JB):               0.00
Kurtosis:               9.274    Cond. No.               181.
=====
```

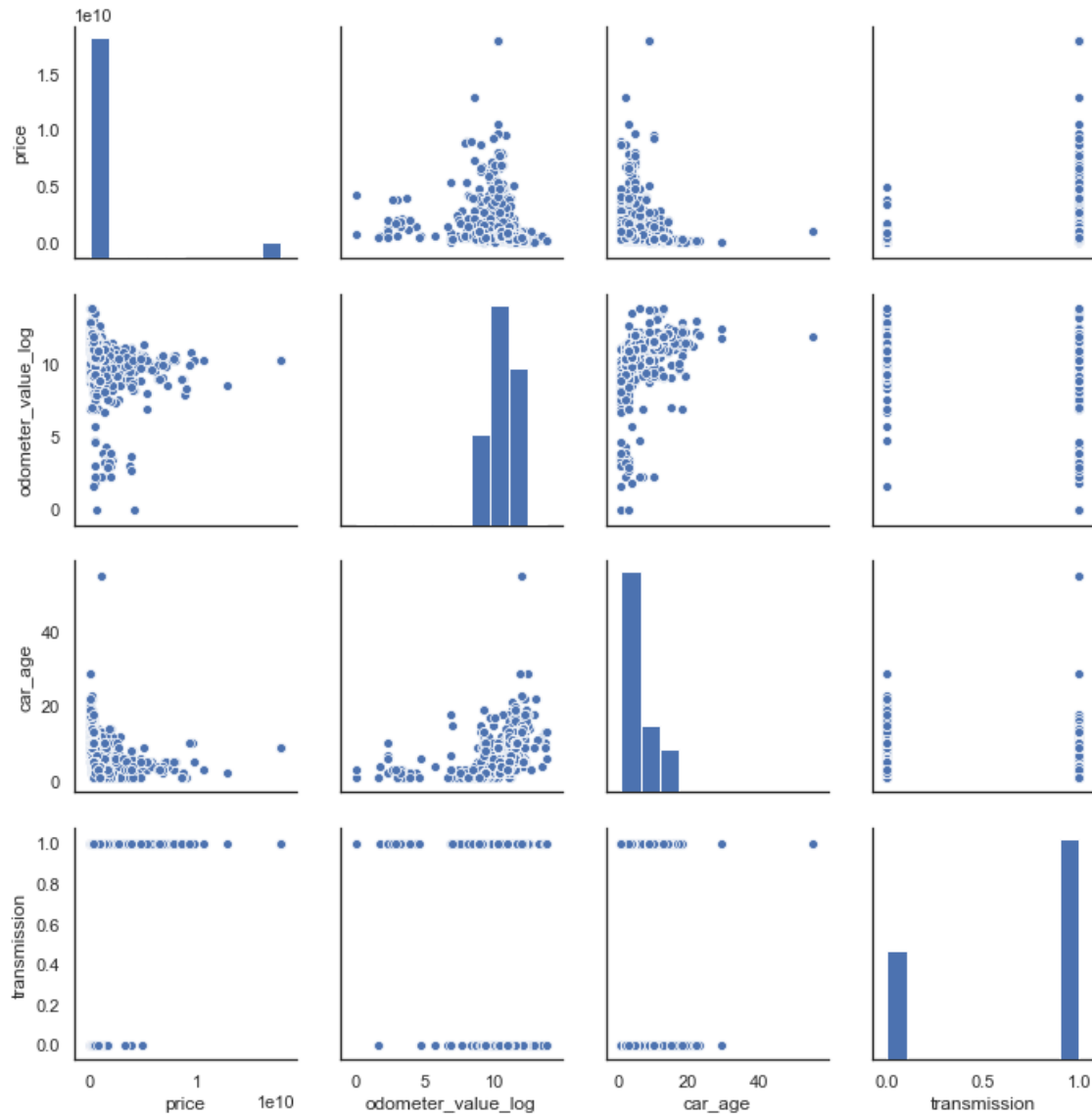
Warnings:

[1] Standard Errors assume that the covariance matrix of the errors is correctly

```
specified.  
"""
```

```
[51]: # Visualizing Pairplots of Price vs Other Features and Only Mileage logged
```

```
sns.pairplot(df_price_no_brands_only_mileage_logged);
```



1.7.2 Section 3.2.2: Normal Price with Logged Mileage and Squared Engine CC

```
[52]: # Creating a copy of the dataframe to work log on
      #df_price_no_brands_mileage_logged_squared_engine_cap = \
      ↪df_price_no_brands_only_mileage_logged.copy()

[53]: # Square Engine CC
      #df_price_no_brands_mileage_logged_squared_engine_cap["engine_squared"] = \
      #df_price_no_brands_mileage_logged_squared_engine_cap['engine_capacity'].
      ↪apply(lambda x: x**2)

[54]: #df_price_no_brands_mileage_logged_squared_engine_cap.columns

[55]: # Rearrange columns
      #df_price_no_brands_mileage_logged_squared_engine_cap = \
      #df_price_no_brands_mileage_logged_squared_engine_cap[['PRICE', 'MILEAGE_LOG', \
      ↪'COE_FROM_SCRAPE_DATE', 'DAYS_OF_COE_LEFT',
      #      'CAR_AGE', 'OMV', 'ENGINE_SQUARED', 'CURB_WEIGHT_KG',
      #      'NO_OF_OWNERS', 'TRANSMISSION']]

[56]: # Slicing Data into Independent Variables (Features) and Dependent Variable ↪
      ↪(Target)
      #X = df_price_no_brands_mileage_logged_squared_engine_cap[ \
      ↪['odometer_value_log', 'car_age',
      #      'transmission', 'engine_squared']] .astype(float)
      #X = sm.add_constant(X)
      #y = df_price_no_brands_only_mileage_logged['price'].astype(int)

      # model / fit / summarize
      #import statsmodels.api as sm

      #lsm = sm.OLS(y, X)
      #results = lsm.fit()
      #results.summary()

[57]: # Visualizing Pairplots of the distributions

      #sns.pairplot(df_price_no_brands_mileage_logged_squared_engine_cap);

[58]: # Viewing Corr Matrix of Price vs Independent Variables (only logged mileage ↪
      ↪and squared engine CC)

      #sns.set(style="white")
      #data = df_price_no_brands_mileage_logged_squared_engine_cap.corr()

      # Generate a mask for the upper triangle
```

```

#mask = np.zeros_like(data, dtype=np.bool)
#mask[np.triu_indices_from(mask)] = True

# Set up the matplotlib figure to control size of heatmap
#fig, ax = plt.subplots(figsize=(15,15))

# Create a custom color palette
#cmap = \
#sns.diverging_palette(133, 10,
#                      as_cmap=True) # as_cmap returns a matplotlib colormap
#    ↳ object rather than a list of colors
# Green = Good (low correlation), Red = Bad (high correlation) between the
#    ↳ independent variables

# Plot the heatmap
#sns.heatmap(data, mask=mask, annot=True,
#            square=True, cmap=cmap, vmin=-1, vmax=1, ax=ax);

# Prevent Heatmap Cut-Off Issue
#bottom, top = ax.get_ylim()
#ax.set_ylim(bottom + 0.5, top - 0.5)

```

1.7.3 Section 3.2.3: Logged Price with Logged Mileage and Squared Engine CC

```

[59]: #df_logged_price_no_brands_mileage_logged_squared_engine_cap = \
#    ↳ df_price_no_brands_mileage_logged_squared_engine_cap.copy()

[60]: # Logging Price
#df_logged_price_no_brands_mileage_logged_squared_engine_cap['price_log'] = \
#df_logged_price_no_brands_mileage_logged_squared_engine_cap['price'].apply(np.
#    ↳ log)

[61]: # Rearrange columns
#df_logged_price_no_brands_mileage_logged_squared_engine_cap = \
#df_logged_price_no_brands_mileage_logged_squared_engine_cap[['price_log',
#    ↳ 'odometer_value_log', 'car_age',
#    'transmission', 'engine_squared']]

[62]: # Slicing Data into Independent Variables (Features) and Dependent Variable
#    ↳ (Target)
#X = df_logged_price_no_brands_mileage_logged_squared_engine_cap[
#    ↳ ['odometer_value_log', 'car_age',
#    'transmission', 'engine_squared']] .astype(float)
#X = sm.add_constant(X)
#y = df_logged_price_no_brands_mileage_logged_squared_engine_cap['price_log'].
#    ↳ astype(float)

```



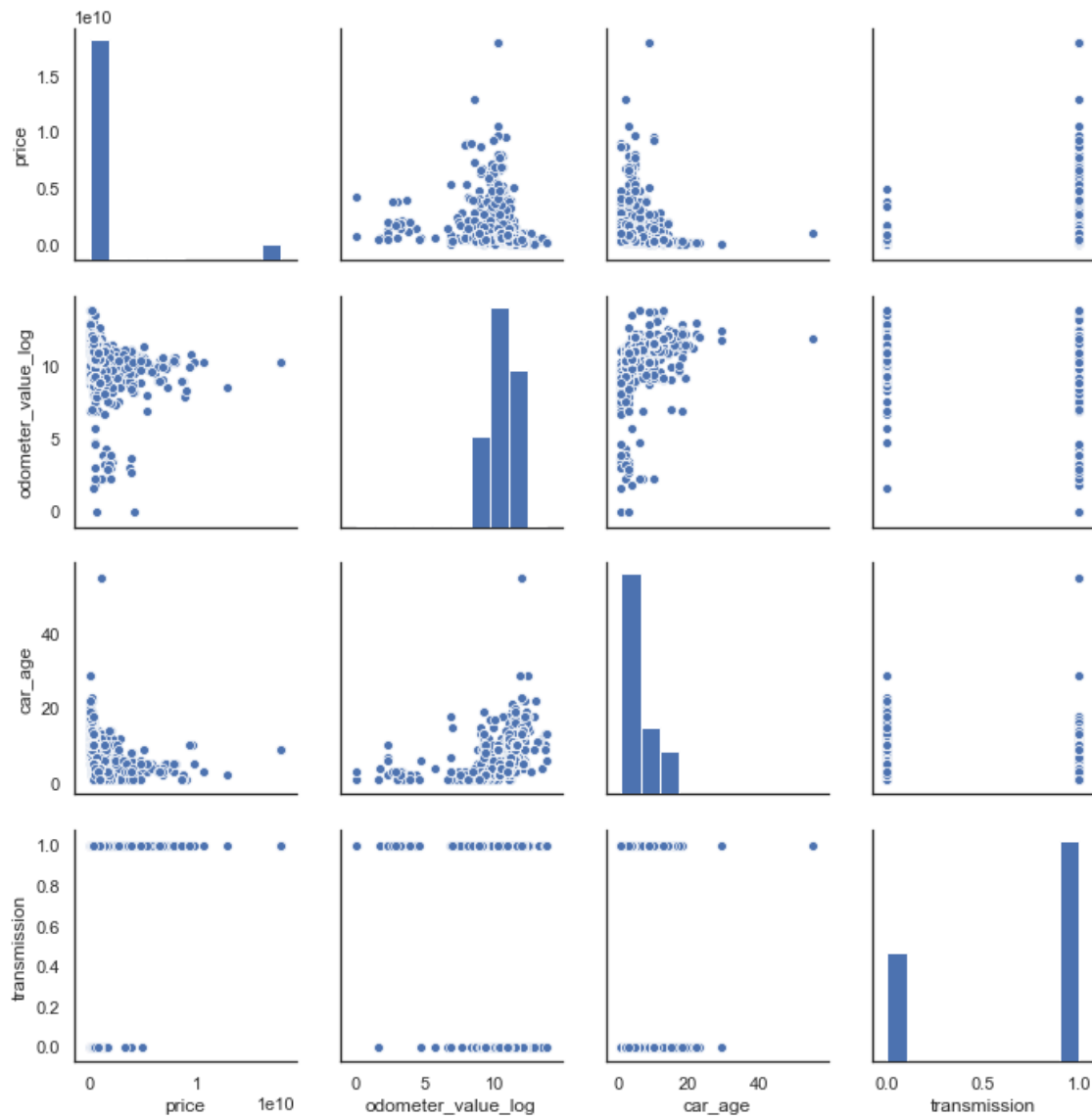
```
# model / fit / summarize
# import statsmodels.api as sm

# lsm = sm.OLS(y, X)
# results = lsm.fit()
# results.summary()
```

1.7.4 Section 3.2.3: Logged Price with Logged Mileage only (no engine squared)

```
[63]: df_logged_price_no_brands_only_mileage_logged = ↳
↳ df_price_no_brands_only_mileage_logged.copy()
```

```
[64]: sns.pairplot(df_logged_price_no_brands_only_mileage_logged);
plt.savefig("log_price_and_mileage.png")
```



```
[65]: # Logging Price
df_logged_price_no_brands_only_mileage_logged['price'] =
    ↪ df_logged_price_no_brands_only_mileage_logged['price'].apply(np.log)

# Renaming column
df_logged_price_no_brands_only_mileage_logged.rename(columns={'price':
    ↪ 'price_log'}, inplace=True)
```

```
[66]: df_logged_price_no_brands_only_mileage_logged.columns
```

```
[66]: Index(['price_log', 'odometer_value_log', 'car_age', 'transmission'],
dtype='object')
```

```
[67]: df_logged_price_no_brands_only_mileage_logged.head()
```

```
[67]:   price_log  odometer_value_log  car_age  transmission
0   20.225685         9.615805         2             1
1   20.183698        11.925035         4             0
2   20.069339        11.314475         6             1
3   19.968243        11.751942        11             1
4   19.718144        11.156251         5             0
```

```
[68]: df_logged_price_no_brands_only_mileage_logged.dropna(inplace=True)
df_logged_price_no_brands_only_mileage_logged.describe()
```

```
[68]:   price_log  odometer_value_log  car_age  transmission
count  17026.000000        17026.000000  17026.000000  17026.000000
mean    20.313010         10.704260     5.766181     0.669329
std      0.986852         1.053606     4.207260     0.470468
min     17.399029         0.000000     1.000000     0.000000
25%     19.729014         10.308953     2.000000     0.000000
50%     20.009916         10.915088     4.000000     1.000000
75%     20.601098         11.589887    10.000000     1.000000
max     23.608067         13.815511    55.000000     1.000000
```

```
[69]: # Slicing Data into Independent Variables (Features) and Dependent Variable
      ↪ (Target)
X = df_logged_price_no_brands_only_mileage_logged[['odometer_value_log',
      ↪ 'car_age', 'transmission']].astype(float)
X = sm.add_constant(X)
y = df_logged_price_no_brands_only_mileage_logged['price_log'].astype(float)

# model / fit / summarize
import statsmodels.api as sm

lsm = sm.OLS(y, X)
results = lsm.fit()
results.summary()
```

```
[69]: <class 'statsmodels.iolib.summary.Summary'>
      """
```

```

                        OLS Regression Results
=====
Dep. Variable:          price_log   R-squared:                0.142
Model:                  OLS        Adj. R-squared:             0.142
Method:                 Least Squares   F-statistic:            937.1
Date:                  Tue, 07 Jan 2020   Prob (F-statistic):      0.00
Time:                  10:49:43          Log-Likelihood:         -22632.
No. Observations:      17026            AIC:                  4.527e+04
```

```

Df Residuals:          17022    BIC:          4.530e+04
Df Model:              3
Covariance Type:      nonrobust
=====
=====
              coef      std err          t      P>|t|      [0.025
0.975]
-----
const          23.1053      0.101     229.692      0.000      22.908
23.302
odometer_value_log -0.2934      0.010    -29.552      0.000     -0.313
-0.274
car_age         0.0118      0.002      4.780      0.000      0.007
0.017
transmission     0.4192      0.016     26.336      0.000      0.388
0.450
=====
Omnibus:          6772.075    Durbin-Watson:          2.037
Prob(Omnibus):    0.000    Jarque-Bera (JB):      26406.582
Skew:             1.998    Prob(JB):              0.00
Kurtosis:         7.611    Cond. No.              181.
=====

```

Warnings:

```

[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
"""

```

1.7.5 R^2 Summary from Linear Regression Models

Price vs Original Independent Variables:

R^2 : **0.295**

R^2 Adjusted: **0.295**

df_price_no_brands

Logged Price vs Independent Variables (Logged Mileage): R^2 : **0.142**

R^2 Adjusted: **0.142**

df_logged_price_no_brands_only_mileage_logged

1.8 Section 3.3: Analysis of Car Brands, Vehicle Types, Brand Categorization

1.8.1 3.3.1: Further breaking down dataframe into finalized features (with log price, log mileage)

```
[70]: df_finalized_features = df_clean2.copy()
```

```
[71]: # Applying log to the desired features
df_finalized_features['price'] = df_clean2['price'].apply(np.log)
df_finalized_features['odometer_value'] = df_clean2['odometer_value'].apply(np.
    ↪ log)

# Renaming features
df_finalized_features.rename(columns={'price': 'price_log',
    ↪ 'odometer_value':
    ↪ 'odometer_value_log'}, inplace=True)
```

```
[72]: df_finalized_features.columns
```

```
[72]: Index(['manufacturer_name', 'model_name', 'body_type', 'production_year',
    ↪ 'number_of_seat', 'odometer_value_log', 'price_log', 'transmission',
    ↪ 'car_age', 'Intercept', 'body_type[T.City Car]', 'body_type[T.Coupe]',
    ↪ 'body_type[T.Hatchback]', 'body_type[T.MPV]',
    ↪ 'body_type[T.Pick-up Truck]', 'body_type[T.SUV]', 'body_type[T.Sedan]',
    ↪ 'body_type[T.Special Purpose]', 'body_type[T.Sport Car]',
    ↪ 'body_type[T.Truck]', 'body_type[T.Van/Minivan]'],
    ↪ dtype='object')
```

```
[73]: # Rearranging Columns and removing unwanted variables
#df_finalized_features = df_finalized_features[['BRAND', 'PRICE_LOG',
    ↪ 'MILEAGE_LOG', 'COE_FROM_SCRAPE_DATE', 'DAYS_OF_COE_LEFT',
    ↪ 'CAR_AGE', 'OMV', 'ENGINE_CAPACITY_CC', 'CURB_WEIGHT_KG',
    ↪ 'NO_OF_OWNERS', 'TRANSMISSION', 'Intercept', 'VEHICLE_TYPE[T.Luxury
    ↪ Sedan]',
    ↪ 'VEHICLE_TYPE[T.MPV]', 'VEHICLE_TYPE[T.Mid-Sized Sedan]',
    ↪ 'VEHICLE_TYPE[T.SUV]', 'VEHICLE_TYPE[T.Sports Car]',
    ↪ 'VEHICLE_TYPE[T.Stationwagon]]]
```

```
[74]: df_logged_price_no_brands_only_mileage_logged.columns
```

```
[74]: Index(['price_log', 'odometer_value_log', 'car_age', 'transmission'],
    ↪ dtype='object')
```

```
[75]: df_finalized_features.columns
```

```
[75]: Index(['manufacturer_name', 'model_name', 'body_type', 'production_year',
    ↪ 'number_of_seat', 'odometer_value_log', 'price_log', 'transmission',
    ↪ 'car_age', 'Intercept', 'body_type[T.City Car]', 'body_type[T.Coupe]',
    ↪ 'body_type[T.Hatchback]', 'body_type[T.MPV]',
    ↪ 'body_type[T.Pick-up Truck]', 'body_type[T.SUV]', 'body_type[T.Sedan]',
    ↪ 'body_type[T.Special Purpose]', 'body_type[T.Sport Car]',
    ↪ 'body_type[T.Truck]', 'body_type[T.Van/Minivan]'],
    ↪ dtype='object')
```

1.8.2 3.3.2: Joining Brand Dummy Variables into Main Dataframe

```
[76]: # Creating a new DataFrame for this Brand Categorization
df_categorized_car_brands = df_finalized_features.copy()
```

```
[77]: print(df_categorized_car_brands['manufacturer_name'].value_counts())
print(len(df_categorized_car_brands['manufacturer_name'].value_counts()))
```

Toyota	8693
Kia	2228
Mercedes-Benz	1207
Hyundai	1182
Ford	1178
Mazda	1098
Rolls-Royce	1046
Lexus	70
Honda	55
Chevrolet	54
Mitsubishi	32
LandRover	28
BMW	25
Nissan	23
Audi	23
Daewoo	16
Suzuki	13
Cadillac	7
Porsche	7
Bentley	5
Isuzu	4
Acura	4
Jaguar	4
Peugeot	3
Renault	3
Zotye	2
Cửu Long	2
Volkswagen	2
Mini	2
Maserati	1
Ssangyong	1
Samco	1
Subaru	1
Thaco	1
Infiniti	1
Hino	1
Fuso	1
FAW	1
Fiat	1

Name: manufacturer_name, dtype: int64

```
[78]: # Creating the relevant columns
df_categorized_car_brands['EXOTIC'] = 0 # Create EXOTIC column
df_categorized_car_brands["ULTRA_LUXURY"] = 0
df_categorized_car_brands["LUXURY"] = 0
df_categorized_car_brands["MID_LEVEL"] = 0
df_categorized_car_brands["ECONOMY"] = 0

[79]: # Labelling Car Brands into Exotic
df_categorized_car_brands.loc[(df_clean2['manufacturer_name'] == "Aston_
    ↪Martin") |
    (df_clean2['manufacturer_name'] == "Ferrari") |
    (df_clean2['manufacturer_name'] == "Lamborghini") |
    (df_clean2['manufacturer_name'] == "McLaren") |
    (df_clean2['manufacturer_name'] == "Hummer"),
    'EXOTIC'] = 1

# Labelling Car Brands into Ultra Luxury
df_categorized_car_brands.loc[(df_clean2['manufacturer_name'] == "Bentley") |
    (df_clean2['manufacturer_name'] == "Land Rover") |
    (df_clean2['manufacturer_name'] == "Maserati") |
    (df_clean2['manufacturer_name'] == "Porsche") |
    (df_clean2['manufacturer_name'] == "Rolls-Royce"),
    "ULTRA_LUXURY"] = 1

# Labelling Car Brands into Luxury
df_categorized_car_brands.loc[(df_clean2['manufacturer_name'] == "Audi") |
    (df_clean2['manufacturer_name'] == "BMW") |
    (df_clean2['manufacturer_name'] == "Jeep") |
    (df_clean2['manufacturer_name'] == "Lexus") |
    (df_clean2['manufacturer_name'] == "Lotus") |
    (df_clean2['manufacturer_name'] == "Mercedes-Benz") |
    (df_clean2['manufacturer_name'] == "Volvo") |
    (df_clean2['manufacturer_name'] == "Peugeot"),
    "LUXURY"] = 1

# Labelling Car Brands into Mid-Level
df_categorized_car_brands.loc[(df_clean2['manufacturer_name'] == "Infiniti") |
    (df_clean2['manufacturer_name'] == "MINI") |
    (df_clean2['manufacturer_name'] == "Volkswagen") |
    (df_clean2['manufacturer_name'] == "Renault") |
    (df_clean2['manufacturer_name'] == "Peugeot"),
    "MID_LEVEL"] = 1
```

```
# (df_clean2['manufacturer_name'] == "Opel") & "Alfa Romeo" will be considered
↳ as "Others" because it is not a very common brand in Singapore
```

```
# Labelling Car Brands into Economy
```

```
df_categorized_car_brands.loc[(df_clean2['manufacturer_name'] == "Chevrolet") |
                              (df_clean2['manufacturer_name'] == "Citroen") |
                              (df_clean2['manufacturer_name'] == "Ford") |
                              (df_clean2['manufacturer_name'] == "Honda") |
                              (df_clean2['manufacturer_name'] == "Hyundai") |
                              (df_clean2['manufacturer_name'] == "Kia") |
                              (df_clean2['manufacturer_name'] == "Mazda") |
                              (df_clean2['manufacturer_name'] == "Mitsubishi") |
                              (df_clean2['manufacturer_name'] == "Nissan") |
                              (df_clean2['manufacturer_name'] == "Suzuki") |
                              (df_clean2['manufacturer_name'] == "Toyota"),
                              "ECONOMY"] = 1
```

```
# (df_clean2['manufacturer_name'] == "Ssangyong") will be considered as
↳ "Others" because it is not a common brand in Singapore
```

```
# Changing Uncommon Car brands to "Others"
```

```
df_categorized_car_brands.loc[(df_clean2['manufacturer_name'] == 'Opel') |
                              (df_clean2['manufacturer_name'] == 'Ssangyong') |
                              (df_clean2['manufacturer_name'] == 'Proton') |
                              (df_clean2['manufacturer_name'] == 'Daihatsu') |
                              (df_clean2['manufacturer_name'] == 'Fiat') |
                              (df_clean2['manufacturer_name'] == 'Alfa Romeo') |
                              (df_clean2['manufacturer_name'] == 'Skoda') |
                              (df_clean2['manufacturer_name'] == 'Hummer') |
                              (df_clean2['manufacturer_name'] == 'Aston Martin') |
                              (df_clean2['manufacturer_name'] == 'Lotus') |
                              (df_clean2['manufacturer_name'] == 'Ford') |
                              (df_clean2['manufacturer_name'] == 'Jeep'),
                              'manufacturer_name'] = "Others"
```

```
# Group uncommon cars into "Others". There are too many brands to work with.
```

```
[80]: print(df_categorized_car_brands['manufacturer_name'].value_counts())
      print(len(df_categorized_car_brands['manufacturer_name'].value_counts()))
```

Toyota	8693
Kia	2228
Mercedes-Benz	1207

Hyundai	1182
Others	1180
Mazda	1098
Rolls-Royce	1046
Lexus	70
Honda	55
Chevrolet	54
Mitsubishi	32
LandRover	28
BMW	25
Nissan	23
Audi	23
Daewoo	16
Suzuki	13
Cadillac	7
Porsche	7
Bentley	5
Jaguar	4
Isuzu	4
Acura	4
Peugeot	3
Renault	3
Cửu Long	2
Zotye	2
Volkswagen	2
Mini	2
Infiniti	1
Thaco	1
FAW	1
Maserati	1
Hino	1
Subaru	1
Fuso	1
Samco	1

Name: manufacturer_name, dtype: int64
37

Brand Dummy Variables Creation

```
[81]: x_brand_dummy = patsy.dmatrix('manufacturer_name',  
    ↪ data=df_categorized_car_brands, return_type='dataframe')  
x_brand_dummy.head()
```

```
[81]:
```

	Intercept	manufacturer_name[T.Audi]	manufacturer_name[T.BMW]	\
0	1.0	0.0	0.0	
1	1.0	0.0	0.0	
2	1.0	0.0	0.0	
3	1.0	0.0	0.0	

4	1.0	0.0	0.0
---	-----	-----	-----

	manufacturer_name[T.Bentley]	manufacturer_name[T.Cadillac]	\
0	0.0	0.0	
1	0.0	0.0	
2	0.0	0.0	
3	0.0	0.0	
4	0.0	0.0	

	manufacturer_name[T.Chevrolet]	manufacturer_name[T.Cửu Long]	\
0	0.0	0.0	
1	0.0	0.0	
2	0.0	0.0	
3	0.0	0.0	
4	0.0	0.0	

	manufacturer_name[T.Daewoo]	manufacturer_name[T.FAW]	\
0	0.0	0.0	
1	0.0	0.0	
2	0.0	0.0	
3	0.0	0.0	
4	0.0	0.0	

	manufacturer_name[T.Fuso]	...	manufacturer_name[T.Porsche]	\
0	0.0	...	0.0	
1	0.0	...	0.0	
2	0.0	...	0.0	
3	0.0	...	0.0	
4	0.0	...	0.0	

	manufacturer_name[T.Renault]	manufacturer_name[T.Rolls-Royce]	\
0	0.0	0.0	
1	0.0	0.0	
2	0.0	0.0	
3	0.0	0.0	
4	0.0	0.0	

	manufacturer_name[T.Samco]	manufacturer_name[T.Subaru]	\
0	0.0	0.0	
1	0.0	0.0	
2	0.0	0.0	
3	0.0	0.0	
4	0.0	0.0	

	manufacturer_name[T.Suzuki]	manufacturer_name[T.Thaco]	\
0	0.0	0.0	
1	0.0	0.0	

2	0.0	0.0
3	0.0	0.0
4	0.0	0.0

	manufacturer_name[T.Toyota]	manufacturer_name[T.Volkswagen]	\
0	0.0	0.0	
1	1.0	0.0	
2	1.0	0.0	
3	1.0	0.0	
4	0.0	0.0	

	manufacturer_name[T.Zotye]
0	0.0
1	0.0
2	0.0
3	0.0
4	0.0

[5 rows x 37 columns]

```
[82]: df_categorized_car_brands.drop('Intercept',axis=1,inplace=True)# Drop intercept
      ↳because already have intercept from previous vehicle type
df_categorized_car_brands = df_categorized_car_brands.join(x_brand_dummy)
df_categorized_car_brands
```

```
[82]:
```

	manufacturer_name	model_name	body_type	production_year	\
0	Kia	cerato	Sedan	2018	
1	Toyota	innova	MPV	2016	
2	Toyota	innova	MPV	2014	
3	Toyota	corolla-altis	Sedan	2009	
4	Kia	rio	Sedan	2015	
...	
20535	Others	ranger	Pick-up Truck	2017	
20536	Toyota	fortuner	SUV	2017	
20537	Rolls-Royce	phantom	Sedan	2011	
20538	Toyota	fortuner	SUV	2019	
20539	Toyota	camry	Sedan	2007	

	number_of_seat	odometer_value_log	price_log	transmission	car_age	\
0	5	9.615805	20.225685	1	2	
1	7	11.925035	20.183698	0	4	
2	8	11.314475	20.069339	1	6	
3	5	11.751942	19.968243	1	11	
4	5	11.156251	19.718144	0	5	
...	
20535	5	10.915088	20.487544	1	3	
20536	7	10.915088	20.601098	0	3	

20537	4	10.308953	23.608067	1	9
20538	7	8.853665	20.752825	1	1
20539	5	11.918391	19.883936	1	13

	body_type[T.City Car]	...	manufacturer_name[T.Porsche]	\
0	0.0	...	0.0	
1	0.0	...	0.0	
2	0.0	...	0.0	
3	0.0	...	0.0	
4	0.0	...	0.0	
...	
20535	0.0	...	0.0	
20536	0.0	...	0.0	
20537	0.0	...	0.0	
20538	0.0	...	0.0	
20539	0.0	...	0.0	

	manufacturer_name[T.Renault]	manufacturer_name[T.Rolls-Royce]	\
0	0.0	0.0	
1	0.0	0.0	
2	0.0	0.0	
3	0.0	0.0	
4	0.0	0.0	
...	
20535	0.0	0.0	
20536	0.0	0.0	
20537	0.0	1.0	
20538	0.0	0.0	
20539	0.0	0.0	

	manufacturer_name[T.Samco]	manufacturer_name[T.Subaru]	\
0	0.0	0.0	
1	0.0	0.0	
2	0.0	0.0	
3	0.0	0.0	
4	0.0	0.0	
...	
20535	0.0	0.0	
20536	0.0	0.0	
20537	0.0	0.0	
20538	0.0	0.0	
20539	0.0	0.0	

	manufacturer_name[T.Suzuki]	manufacturer_name[T.Thaco]	\
0	0.0	0.0	
1	0.0	0.0	
2	0.0	0.0	

```

3          0.0          0.0
4          0.0          0.0
...
20535      0.0          0.0
20536      0.0          0.0
20537      0.0          0.0
20538      0.0          0.0
20539      0.0          0.0

```

```

      manufacturer_name[T.Toyota]  manufacturer_name[T.Volkswagen]  \
0          0.0          0.0
1          1.0          0.0
2          1.0          0.0
3          1.0          0.0
4          0.0          0.0
...
20535      0.0          0.0
20536      1.0          0.0
20537      0.0          0.0
20538      1.0          0.0
20539      1.0          0.0

```

```

      manufacturer_name[T.Zotye]
0          0.0
1          0.0
2          0.0
3          0.0
4          0.0
...
20535      0.0
20536      0.0
20537      0.0
20538      0.0
20539      0.0

```

[17026 rows x 62 columns]

1.8.3 3.3.4 : Only Brand Segregation

```
[83]: df_categorized_car_brands.columns
```

```
[83]: Index(['manufacturer_name', 'model_name', 'body_type', 'production_year',
'number_of_seat', 'odometer_value_log', 'price_log', 'transmission',
'car_age', 'body_type[T.City Car]', 'body_type[T.Coupe]',
'body_type[T.Hatchback]', 'body_type[T.MPV]',
'body_type[T.Pick-up Truck]', 'body_type[T.SUV]', 'body_type[T.Sedan]',
'body_type[T.Special Purpose]', 'body_type[T.Sport Car]'],
dtype='object', length=20)
```

```
'body_type[T.Truck]', 'body_type[T.Van/Minivan]', 'EXOTIC',
'ULTRA_LUXURY', 'LUXURY', 'MID_LEVEL', 'ECONOMY', 'Intercept',
'manufacturer_name[T.Audi]', 'manufacturer_name[T.BMW]',
'manufacturer_name[T.Bentley]', 'manufacturer_name[T.Cadillac]',
'manufacturer_name[T.Chevrolet]', 'manufacturer_name[T.Cửu Long]',
'manufacturer_name[T.Daewoo]', 'manufacturer_name[T.FAW]',
'manufacturer_name[T.Fuso]', 'manufacturer_name[T.Hino]',
'manufacturer_name[T.Honda]', 'manufacturer_name[T.Hyundai]',
'manufacturer_name[T.Infiniti]', 'manufacturer_name[T.Isuzu]',
'manufacturer_name[T.Jaguar]', 'manufacturer_name[T.Kia]',
'manufacturer_name[T.LandRover]', 'manufacturer_name[T.Lexus]',
'manufacturer_name[T.Maserati]', 'manufacturer_name[T.Mazda]',
'manufacturer_name[T.Mercedes-Benz]', 'manufacturer_name[T.Minis]',
'manufacturer_name[T.Mitsubishi]', 'manufacturer_name[T.Nissan]',
'manufacturer_name[T.Others]', 'manufacturer_name[T.Peugeot]',
'manufacturer_name[T.Porsche]', 'manufacturer_name[T.Renault]',
'manufacturer_name[T.Rolls-Royce]', 'manufacturer_name[T.Samco]',
'manufacturer_name[T.Subaru]', 'manufacturer_name[T.Suzuki]',
'manufacturer_name[T.Thaco]', 'manufacturer_name[T.Toyota]',
'manufacturer_name[T.Volkswagen]', 'manufacturer_name[T.Zotye]'],
dtype='object')
```

```
[84]: df_categorized_car_brands[['manufacturer_name', 'model_name', 'body_type',
    ↳ 'production_year', 'Intercept',
    ↳ 'number_of_seat', 'odometer_value_log', 'price_log', 'transmission',
    ↳ 'car_age', 'EXOTIC', 'ULTRA_LUXURY', 'LUXURY',
    ↳ 'MID_LEVEL', 'ECONOMY']]
```

```
[84]:
```

	manufacturer_name	model_name	body_type	production_year	\
0	Kia	cerato	Sedan	2018	
1	Toyota	innova	MPV	2016	
2	Toyota	innova	MPV	2014	
3	Toyota	corolla-altis	Sedan	2009	
4	Kia	rio	Sedan	2015	
...	
20535	Others	ranger	Pick-up Truck	2017	
20536	Toyota	fortuner	SUV	2017	
20537	Rolls-Royce	phantom	Sedan	2011	
20538	Toyota	fortuner	SUV	2019	
20539	Toyota	camry	Sedan	2007	

	Intercept	number_of_seat	odometer_value_log	price_log	transmission	\
0	1.0	5	9.615805	20.225685	1	
1	1.0	7	11.925035	20.183698	0	
2	1.0	8	11.314475	20.069339	1	
3	1.0	5	11.751942	19.968243	1	
4	1.0	5	11.156251	19.718144	0	

...
20535	1.0		5	10.915088	20.487544	1
20536	1.0		7	10.915088	20.601098	0
20537	1.0		4	10.308953	23.608067	1
20538	1.0		7	8.853665	20.752825	1
20539	1.0		5	11.918391	19.883936	1

	car_age	EXOTIC	ULTRA_LUXURY	LUXURY	MID_LEVEL	ECONOMY
0	2	0	0	0	0	1
1	4	0	0	0	0	1
2	6	0	0	0	0	1
3	11	0	0	0	0	1
4	5	0	0	0	0	1
...
20535	3	0	0	0	0	1
20536	3	0	0	0	0	1
20537	9	0	1	0	0	0
20538	1	0	0	0	0	1
20539	13	0	0	0	0	1

[17026 rows x 15 columns]

```
[85]: # Finding out new R^2 from log transformations of Log Price and Log Independent
      ↪ Variables (except Mileage)

      # Slicing Variables
      #df_categorized_car_brands.dropna(inplace=True)
      X = df_categorized_car_brands[['odometer_value_log', 'transmission',
      ↪ 'car_age', 'EXOTIC', 'ULTRA_LUXURY', 'LUXURY',
      ↪ 'MID_LEVEL', 'ECONOMY']]
      X = sm.add_constant(X)
      y = df_categorized_car_brands['price_log'].astype(float)

      # Initially my coefficients were difficult to interpret.
      # Therefore I transformed it using log for better explanation purposes

      # model / fit / summarize
      import statsmodels.api as sm

      lsm = sm.OLS(y, X)
      results = lsm.fit()
      results.summary()
```

```
[85]: <class 'statsmodels.iolib.summary.Summary'>
      """
```

OLS Regression Results

=====

```

Dep. Variable:          price_log    R-squared:                0.858
Model:                  OLS          Adj. R-squared:           0.858
Method:                 Least Squares    F-statistic:             1.473e+04
Date:                  Tue, 07 Jan 2020    Prob (F-statistic):      0.00
Time:                  10:49:43          Log-Likelihood:          -7294.2
No. Observations:      17026            AIC:                    1.460e+04
Df Residuals:          17018            BIC:                    1.467e+04
Df Model:               7
Covariance Type:       nonrobust

```

```

=====
=====

```

	coef	std err	t	P> t	[0.025
0.975]					

const	20.3377	0.058	348.707	0.000	20.223
20.452					
odometer_value_log	0.0873	0.004	20.334	0.000	0.079
0.096					
transmission	0.3163	0.007	48.207	0.000	0.303
0.329					
car_age	-0.0964	0.001	-81.884	0.000	-0.099
-0.094					
EXOTIC	-8.213e-16	7.55e-17	-10.872	0.000	-9.69e-16
-6.73e-16					
ULTRA_LUXURY	2.8997	0.043	66.779	0.000	2.815
2.985					
LUXURY	-0.8710	0.043	-20.190	0.000	-0.956
-0.786					
MID_LEVEL	-0.1835	0.127	-1.445	0.149	-0.432
0.065					
ECONOMY	-0.8510	0.042	-20.236	0.000	-0.933
-0.769					

Omnibus:	4696.531		Durbin-Watson:		1.927
Prob(Omnibus):	0.000		Jarque-Bera (JB):		64009.643
Skew:	0.945		Prob(JB):		0.00
Kurtosis:	12.309		Cond. No.		1.43e+18
=====					

Warnings:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

[2] The smallest eigenvalue is 1.31e-30. This might indicate that there are strong multicollinearity problems or that the design matrix is singular.

"""

1.8.4 R² Summary from Linear Regression Models

Price vs Original Independent Variables:

R²: **0.225**

R² Adjusted: **0.225**

df_price_no_brands

Logged Price vs Independent Variables (Logged Mileage): R²: **0.142**

R² Adjusted: **0.142**

df_logged_price_no_brands_only_mileage_logged

Logged Price vs Independent Variables (Logged Mileage) + Categorized Car Brands: R²: **0.858**

R² Adjusted: **0.858**

df_categorized_car_brands

[]:

1.9 Section 4.2: Cross-Validation Using Models other than LR

1.9.1 Section 4.2.1 : Using LassoCV to find best Alpha Value for L1 Regularization

```
[86]: from sklearn.metrics import mean_squared_error, r2_score
      from sklearn.linear_model import LinearRegression, Lasso, Ridge, ElasticNet
      from sklearn.linear_model import LassoCV, RidgeCV, ElasticNetCV
      from sklearn.preprocessing import StandardScaler, PolynomialFeatures

[87]: from sklearn.model_selection import train_test_split
      from sklearn.model_selection import KFold #Kfold will allow you to do cross_
      validation

      X = df_categorized_car_brands[['odometer_value_log', 'transmission',
      car_age', 'EXOTIC', 'ULTRA_LUXURY', 'LUXURY',
      'MID_LEVEL', 'ECONOMY']]
      y = df_categorized_car_brands['price_log']

      # hold out 20% of the data for final testing

      X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size = 0.
      2, random_state=10)
      X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,
      test_size = 0.2, random_state=20)

[88]: ## Scale the data (a MUST if you're doing regularization)
      std = StandardScaler()
      std.fit(X_train.values)

      ## Scale the Predictors on both the train/validation (the whole 80%) and test_
      set (the whole 20%)
      X_train_scaled = std.transform(X_train.values)
```

```

X_val_scaled = std.transform(X_val.values)

# LassoCV does 2 things for you. It trains your model, and it also chooses the
↳ best lambda/alpha for you.
# But of course, you have to feed it a list of lambdas to try.

# The best part about LambdaCV is that it does all 3 for you:
# Fit
# Finding best lambda
# Doing Cross-Validation

from sklearn.model_selection import KFold    #Kfold will allow you to do cross
↳ validation

# Run the cross validation, find the best alpha, refit the model on all the
↳ data with that alpha

alphavec = 10**np.linspace(-3,3,200) # Defining a vector of lambdas (alpha) to
↳ try from
kf = KFold(n_splits=5, shuffle=True, random_state = 1000) # Creating a
↳ partitioned randomized-state data

lasso_model = LassoCV(alphas = alphavec, cv=kf) # If you want to use Ridge,
↳ use RidgeCV
lasso_model.fit(X_train_scaled, y_train)

# This is the best alpha value it found - not far from the value
# selected using simple validation
lasso_model.alpha_

```

[88]: 0.001

```

[89]: # These are the (standardized) coefficients found when it refit using that best
↳ alpha
list(zip(X_train.columns, lasso_model.coef_))

```

```

[89]: [('odometer_value_log', 0.08470542643925862),
      ('transmission', 0.14717164938035499),
      ('car_age', -0.39641272099307434),
      ('EXOTIC', 0.0),
      ('ULTRA_LUXURY', 0.7648896286058525),
      ('LUXURY', -0.16175046659662895),
      ('MID_LEVEL', -0.0035497269716694884),
      ('ECONOMY', -0.20246361609994537)]

```

```
[90]: def RMSE(y_true, y_pred):
        return np.sqrt(np.mean((y_true - y_pred)**2))

# Make predictions on the test set using the new model
# (the model is already using the best alpha. It is a LassoCV initialization)
val_set_pred = lasso_model.predict(X_val_scaled)

# Find the MAE and R^2 on the test set using this model
print(f"LassoCV Best Lambda (alpha): {lasso_model.alpha_}")
print(f"LassoCV RMSE: {RMSE(y_val, val_set_pred)}")
print(f"LassoCV R^2 Score: {r2_score(y_val, val_set_pred)}")
```

LassoCV Best Lambda (alpha): 0.001
LassoCV RMSE: 0.38373913501898876
LassoCV R^2 Score: 0.8435895300489282

1.9.2 Section 4.2.2: Using RidgeCV to find best Alpha Value for L2 Regularization

```
[91]: from sklearn.metrics import mean_squared_error, r2_score
from sklearn.linear_model import LinearRegression, Lasso, Ridge, ElasticNet
from sklearn.linear_model import LassoCV, RidgeCV, ElasticNetCV
from sklearn.preprocessing import StandardScaler, PolynomialFeatures

from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold #Kfold will allow you to do cross-
    validation

X = df_categorized_car_brands[['odometer_value_log', 'transmission',
    'car_age', 'EXOTIC', 'ULTRA_LUXURY', 'LUXURY',
    'MID_LEVEL', 'ECONOMY']]
y = df_categorized_car_brands['price_log']

# hold out 20% of the data for final testing

X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size = 0.
    2, random_state=10)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,
    test_size = 0.2, random_state=20)
```

```
[92]: # Use RidgeCV to find the optimal ALPHA value for L2 regularization

## Scale the data (a MUST if you're doing regularization)
std = StandardScaler()
std.fit(X_train.values)
```

```

# Scale the Predictors on both the train and validation set (for RidgeCV)
X_train_scaled = std.transform(X_train.values)
X_val_scaled = std.transform(X_val.values)

# Run the cross-validation, find the best alpha, refit the model on all the
↳ data with that alpha (RidgeCV does this for you)
alphavec = 10 ** np.linspace(-3,3,200) # alpha varies from 0.001 to 1000
kf = KFold(n_splits=5, shuffle=True, random_state=1000)

ridge_model = RidgeCV(alphas=alphavec, cv=kf)
ridge_model.fit(X_train_scaled, y_train) # Fit your scaled train input and
↳ your y train values

# This is the best alpha value found
ridge_model.alpha_

```

[92]: 25.23539170434766

```

[93]: # display all coefficients in the model with optimal alpha
list(zip(X_train.columns, ridge_model.coef_))

```

```

[93]: [('odometer_value_log', 0.0876069492402216),
      ('transmission', 0.1487559385087927),
      ('car_age', -0.3988572527834823),
      ('EXOTIC', 0.0),
      ('ULTRA_LUXURY', 0.7373005062921066),
      ('LUXURY', -0.19249255412014488),
      ('MID_LEVEL', -0.006751763592785005),
      ('ECONOMY', -0.2424662145775657)]

```

```

[94]: def RMSE(y_true, y_pred):
      return np.sqrt(np.mean((y_true - y_pred)**2))

# Make predictions on the test set using the new model and save it into a
↳ variable
val_set_pred = lasso_model.predict(X_val_scaled)

# Find the MAE and R^2 on the test set using this model
print(f"Best Lambda (alpha) RidgeCV: {ridge_model.alpha_}")
print(f"RidgeCV MAE: {RMSE(y_val, val_set_pred)}")
print(f"RidgeCV R^2 Score: {r2_score(y_val, val_set_pred)}")

```

```

Best Lambda (alpha) RidgeCV: 25.23539170434766
RidgeCV MAE: 0.38373913501898876
RidgeCV R^2 Score: 0.8435895300489282

```

1.9.3 Section 4.2.3: Using ElasticnetCV to find best Alpha Value

```
[95]: from sklearn.metrics import mean_squared_error, r2_score
from sklearn.linear_model import LinearRegression, Lasso, Ridge, ElasticNet
from sklearn.linear_model import LassoCV, RidgeCV, ElasticNetCV
from sklearn.preprocessing import StandardScaler, PolynomialFeatures

from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold #Kfold will allow you to do cross-
    validation

X = df_categorized_car_brands[['odometer_value_log', 'transmission',
    'car_age', 'EXOTIC', 'ULTRA_LUXURY', 'LUXURY',
    'MID_LEVEL', 'ECONOMY']]
y = df_categorized_car_brands['price_log']

# hold out 20% of the data for final testing

X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size = 0.
    2, random_state=10)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,
    test_size = 0.2, random_state=20)
```

```
[96]: # Using ElasticNetCV to find the optimal ALPHA value
# Scale the data as before (scaling is a must for regularization)
std = StandardScaler()
std.fit(X_train.values) # (60% of the data)

# Scale the Predictors on both the train and validation set
X_train_scaled = std.transform(X_train.values)
X_val_scaled = std.transform(X_val.values)

# Run the cross-validation, find the best alpha, refit the model and all the
    data using that alpha (ElasticNetCV does this for you)
alphavec = 10 ** np.linspace(-3,3,200) # alpha varies from 0.001 to 1000
kf = KFold(n_splits=5, shuffle=True, random_state=1000)

elasticnet_model = ElasticNetCV(alphas = alphavec, cv=kf)
elasticnet_model.fit(X_train_scaled, y_train) # Fitting standardscaled input
    and true y values into model to train it

elasticnet_model.alpha_
```

[96]: 0.001

```
[97]: # display all coefficients in the model with optimal alpha
list(zip(X_train.columns, elasticnet_model.coef_))
```

```
[97]: [('odometer_value_log', 0.08709220005311995),
      ('transmission', 0.14816022550936267),
      ('car_age', -0.3987944732203305),
      ('EXOTIC', 0.0),
      ('ULTRA_LUXURY', 0.7526732782206628),
      ('LUXURY', -0.17604684112050104),
      ('MID_LEVEL', -0.005087419957232235),
      ('ECONOMY', -0.22129368702501492)]
```

```
[98]: def RMSE(y_true, y_pred):
      return np.sqrt(np.mean((y_true - y_pred)**2))

      # Use this model to do prediction on a validation data set
      val_set_pred = elasticnet_model.predict(X_val_scaled)

      # Find the MAE and R^2 on the test set using this model
      print(f'Best Lambda (Alpha) ElasticNetCV: {elasticnet_model.alpha_}')
      print(f'ElasticNetCV RMSE: {RMSE(y_val, val_set_pred)}') # mae is a defined_
      ↪function above
      print(f'ElasticNetCV R^2 Score: {r2_score(y_val, val_set_pred)}') # r2_score_
      ↪is an imported module
```

```
Best Lambda (Alpha) ElasticNetCV: 0.001
ElasticNetCV RMSE: 0.38350590470075996
ElasticNetCV R^2 Score: 0.843779599679487
```

1.9.4 Section 4.2.3: Summary from the above Train-Validation Sets

LassoCV Best Lambda (alpha): 0.001 LassoCV RMSE: 0.38373913501898876 LassoCV R² Score: 0.8435895300489282

Best Lambda (alpha) RidgeCV: 25.23539170434766 RidgeCV MAE: 0.38373913501898876 RidgeCV R² Score: 0.8435895300489282

Best Lambda (Alpha) ElasticNetCV: 0.001 ElasticNetCV RMSE: 0.38350590470075996 ElasticNetCV R² Score: 0.843779599679487

1.9.5 Section 4.3: Picking Model with the best R² score (Train and Cross-Validation)

```
[99]: from sklearn.metrics import mean_squared_error, r2_score
      from sklearn.linear_model import LinearRegression, Lasso, Ridge, ElasticNet
      from sklearn.linear_model import LassoCV, RidgeCV, ElasticNetCV
      from sklearn.preprocessing import StandardScaler, PolynomialFeatures
```

```

from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold #Kfold will allow you to do cross_
    ↪validation

X = df_categorized_car_brands[[ 'odometer_value_log', 'transmission',
    ↪'car_age', 'EXOTIC', 'ULTRA_LUXURY', 'LUXURY',
    ↪'MID_LEVEL', 'ECONOMY']]
y = df_categorized_car_brands['price_log']

# Create 80% of train data. The code below will automate the cross-validation
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size = 0.
    ↪2, random_state=10)

```

```

[100]: ## cross validation using KFold (on the 100% dataset, without manually_
    ↪splitting)
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold

def RMSE(y_true, y_pred):
    return np.sqrt(np.mean((y_true - y_pred)**2))

#Feature transform/scaling so that we can run our ridge/lasso/elasticnet model
scaler = StandardScaler()
X_train_val_scaled = scaler.fit_transform(X_train_val.values)

#Feature transform/scaling so that we can run our poly model
poly = PolynomialFeatures(degree=2)
X_train_val_poly = poly.fit_transform(X_train_val.values)

kf = KFold(n_splits=5, shuffle=True, random_state = 1000)

lm = LinearRegression()
cvs_lm = cross_val_score(lm, X_train_val, y_train_val, cv=kf, scoring='r2')
print("Linear Regression Cross Val Score: {}".format(cvs_lm))
# print(f'ElasticNetCV RMSE: {round(RMSE(y_val, val_set_pred),3)}')
print('Linear regression cv R^2:', round(np.mean(cvs_lm),3), '+-', round(np.
    ↪std(cvs_lm),3), '\n' )

lm_ridge = Ridge(alpha= 25.23539170434766 )
cvs_ridge = cross_val_score(lm_ridge, X_train_val_scaled, y_train_val, cv=kf,
    ↪scoring='r2')
print("Ridge Cross Val Score: {}".format(cvs_ridge))
# print(f'ElasticNetCV RMSE: {round(RMSE(y_val, val_set_pred),3)}')
print('Ridge regression cv R^2:', round(np.mean(cvs_ridge),3), '+-', round(np.
    ↪std(cvs_ridge),3), '\n' )

```

```

lm_lasso = Lasso(alpha=0.001)
cvs_lasso = cross_val_score(lm_lasso, X_train_val_scaled, y_train_val, cv=kf,
    ↪scoring='r2')
print("Lasso Cross Val Score: {}".format(cvs_lasso))
# print(f'ElasticNetCV RMSE: {round(RMSE(y_val, val_set_pred),3)}')
print('Lasso regression cv R^2:', round(np.mean(cvs_lasso),3), '+-', round(np.
    ↪std(cvs_lasso),3), '\n' )

lm_elasticnet = ElasticNet(alpha=0.001 )
cvs_elasticnet = cross_val_score(lm_elasticnet, X_train_val_scaled,
    ↪y_train_val, cv=kf, scoring='r2')
print("Elastic Net Cross Val Score: {}".format(cvs_elasticnet))
# print(f'ElasticNetCV RMSE: {round(RMSE(y_val, val_set_pred),3)}')
print('ElasticNet regression cv R^2:', round(np.mean(cvs_elasticnet),3), '+-',
    ↪round(np.std(cvs_elasticnet),3), '\n' )

lm_poly = LinearRegression()
cvs_poly = cross_val_score(lm_poly, X_train_val_poly, y_train_val, cv=kf,
    ↪scoring='r2')
print("Poly Regression Cross Val Score: {}".format(cvs_poly))
# print(f'ElasticNetCV RMSE: {round(RMSE(y_val, val_set_pred),3)}')
print('Degree 2 polynomial Regression cv R^2:', round(np.mean(cvs_poly),3),
    ↪'+-', round(np.std(cvs_poly),3) )

```

Linear Regression Cross Val Score: [0.85352277 0.86759166 0.84863217 0.85471994 0.84724817]

Linear regression cv R^2: 0.854 +- 0.007

Ridge Cross Val Score: [0.85342813 0.86762171 0.84882679 0.85501083 0.84722059]

Ridge regression cv R^2: 0.854 +- 0.007

Lasso Cross Val Score: [0.85350301 0.86721532 0.84914978 0.85352675 0.84771586]

Lasso regression cv R^2: 0.854 +- 0.007

Elastic Net Cross Val Score: [0.85350324 0.86742934 0.84894665 0.8542459

0.84748116]

ElasticNet regression cv R^2: 0.854 +- 0.007

Poly Regression Cross Val Score: [0.90227071 0.90362012 0.89025832 0.89516535

0.89900252]

Degree 2 polynomial Regression cv R^2: 0.898 +- 0.005


```
[101]: # From the code above, it seems like Linear Regression provides similar results_
        ↪as compared to the rest.
        # Therefore, will choose to use linear regression for it's simplicity and ease_
        ↪of use
```

2 Section 5: Model Testing (On whole DataSet)

2.1 Section 5.1: Training Model on 80% DataSet

```
[102]: df_categorized_car_brands.columns
```

```
[102]: Index(['manufacturer_name', 'model_name', 'body_type', 'production_year',
            'number_of_seat', 'odometer_value_log', 'price_log', 'transmission',
            'car_age', 'body_type[T.City Car]', 'body_type[T.Coupe]',
            'body_type[T.Hatchback]', 'body_type[T.MPV]',
            'body_type[T.Pick-up Truck]', 'body_type[T.SUV]', 'body_type[T.Sedan]',
            'body_type[T.Special Purpose]', 'body_type[T.Sport Car]',
            'body_type[T.Truck]', 'body_type[T.Van/Minivan]', 'EXOTIC',
            'ULTRA_LUXURY', 'LUXURY', 'MID_LEVEL', 'ECONOMY', 'Intercept',
            'manufacturer_name[T.Audi]', 'manufacturer_name[T.BMW]',
            'manufacturer_name[T.Bentley]', 'manufacturer_name[T.Cadillac]',
            'manufacturer_name[T.Chevrolet]', 'manufacturer_name[T.Cũu Long]',
            'manufacturer_name[T.Daewoo]', 'manufacturer_name[T.FAW]',
            'manufacturer_name[T.Fuso]', 'manufacturer_name[T.Hino]',
            'manufacturer_name[T.Honda]', 'manufacturer_name[T.Hyundai]',
            'manufacturer_name[T.Infiniti]', 'manufacturer_name[T.Isuzu]',
            'manufacturer_name[T.Jaguar]', 'manufacturer_name[T.Kia]',
            'manufacturer_name[T.LandRover]', 'manufacturer_name[T.Lexus]',
            'manufacturer_name[T.Maserati]', 'manufacturer_name[T.Mazda]',
            'manufacturer_name[T.Mercedes-Benz]', 'manufacturer_name[T.Mini]',
            'manufacturer_name[T.Mitsubishi]', 'manufacturer_name[T.Nissan]',
            'manufacturer_name[T.Others]', 'manufacturer_name[T.Peugeot]',
            'manufacturer_name[T.Porsche]', 'manufacturer_name[T.Renault]',
            'manufacturer_name[T.Rolls-Royce]', 'manufacturer_name[T.Samco]',
            'manufacturer_name[T.Subaru]', 'manufacturer_name[T.Suzuki]',
            'manufacturer_name[T.Thaco]', 'manufacturer_name[T.Toyota]',
            'manufacturer_name[T.Volkswagen]', 'manufacturer_name[T.Zotye]'],
            dtype='object')
```

```
[103]: from sklearn.metrics import mean_squared_error, r2_score
        from sklearn.linear_model import LinearRegression, Lasso, Ridge, ElasticNet
        from sklearn.linear_model import LassoCV, RidgeCV, ElasticNetCV
        from sklearn.preprocessing import StandardScaler, PolynomialFeatures

        from sklearn.model_selection import train_test_split
```

```

from sklearn.model_selection import KFold #Kfold will allow you to do cross
↳validation

X = df_categorized_car_brands[['odometer_value_log', 'transmission',
↳'car_age', 'Intercept', 'EXOTIC', 'ULTRA_LUXURY', 'LUXURY',
↳'MID_LEVEL', 'ECONOMY']]
#X = sm.add_constant(X)
y = df_categorized_car_brands['price_log']

# Create 80% of train data. The code below will automate the cross-validation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
↳random_state=20)

```

```

[104]: # model / fit / summarize
import statsmodels.api as sm

lm_model = LinearRegression()
lm_model = sm.OLS(y_train, X_train) # no need sm.add_constant because there's
↳already an intercept
results = lm_model.fit()
results.summary()

```

[104]: <class 'statsmodels.iolib.summary.Summary'>

```

"""
                                OLS Regression Results
=====
Dep. Variable:                  price_log    R-squared:                0.857
Model:                            OLS       Adj. R-squared:            0.857
Method:                 Least Squares    F-statistic:                1.166e+04
Date:                Tue, 07 Jan 2020    Prob (F-statistic):          0.00
Time:                  10:49:48    Log-Likelihood:            -5816.0
No. Observations:                  13620    AIC:                      1.165e+04
Df Residuals:                      13612    BIC:                      1.171e+04
Df Model:                            7
Covariance Type:                  nonrobust
=====
=====
                                coef    std err          t      P>|t|      [0.025
0.975]
-----
-----
odometer_value_log    0.0986    0.005    20.025    0.000    0.089
0.108
transmission          0.3180    0.007    43.278    0.000    0.304
0.332
car_age              -0.1001    0.001   -74.934    0.000   -0.103
-0.097

```

Intercept	20.2943	0.066	308.872	0.000	20.166
20.423					
EXOTIC	-3.762e-15	3.97e-17	-94.650	0.000	-3.84e-15
-3.68e-15					
ULTRA_LUXURY	2.8605	0.048	59.911	0.000	2.767
2.954					
LUXURY	-0.9081	0.047	-19.165	0.000	-1.001
-0.815					
MID_LEVEL	0.0300	0.153	0.196	0.845	-0.271
0.331					
ECONOMY	-0.9093	0.046	-19.720	0.000	-1.000
-0.819					

Omnibus:	3150.158	Durbin-Watson:	2.005
Prob(Omnibus):	0.000	Jarque-Bera (JB):	30622.838
Skew:	0.833	Prob(JB):	0.00
Kurtosis:	10.155	Cond. No.	1.50e+18

Warnings:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

[2] The smallest eigenvalue is 9.54e-31. This might indicate that there are strong multicollinearity problems or that the design matrix is singular.

""

2.1.1 R² Summary from Linear Regression Models

Best Data Set (80% of data on train, 20% on test)

Logged Price vs Independent Variables (Logged Mileage) + Categorized Car Brands: R²: **0.857**

R² Adjusted: **0.857**

df_categorized_car_brands

Price vs Original Independent Variables:

R²: **0.295**

R² Adjusted: **0.295**

df_price_no_brands

Logged Price vs Independent Variables (Logged Mileage): R²: **0.142**

R² Adjusted: **0.142**

df_logged_price_no_brands_only_mileage_logged

Logged Price vs Independent Variables (Logged Mileage) + Categorized Car Brands: R²: **0.858**

R² Adjusted: **0.858**

df_categorized_car_brands

2.2 Section 5.2: Testing Model on 20% DataSet

```
[105]: from sklearn.metrics import mean_squared_error, r2_score
#Mean Absolute Error (MAE)
def mae(y_true, y_pred):
    return np.mean(np.abs(y_pred - y_true))

# Create prediction variable for test set
model_test_pred = results.predict(X_test)

# Check accuracy of test
print("Linear Regression Test Scores:\n")
print("Linear Regression MAE: {}".format(mae(y_test, model_test_pred))) # MAE
print("Linear Regression MSE: {}".format(mean_squared_error(y_test,
    ↪model_test_pred))) # Mean Squared Error (MSE)
print("Linear Regression RMSE: {}".format(RMSE(y_test, model_test_pred))) #
    ↪Root Mean squared error
print("Linear Regression R2 Score: {}".format(r2_score(y_test,
    ↪model_test_pred)))# R2 Score
```

Linear Regression Test Scores:

Linear Regression MAE: 0.24679311662778855

Linear Regression MSE: 0.14010916191613537

Linear Regression RMSE: 0.37431158399939396

Linear Regression R2 Score: 0.862591947195551

3 Section 6: Checking Linear Regression Assumptions

3.0.1 Plot 3 Graphs

- residue
- QQ plot

```
[106]: import pandas as pd
import numpy as np
import seaborn as sns
import patsy
import scipy.stats as stats

import statsmodels.api as sm
import statsmodels.formula.api as smf
from sklearn import preprocessing
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
%matplotlib inline
```

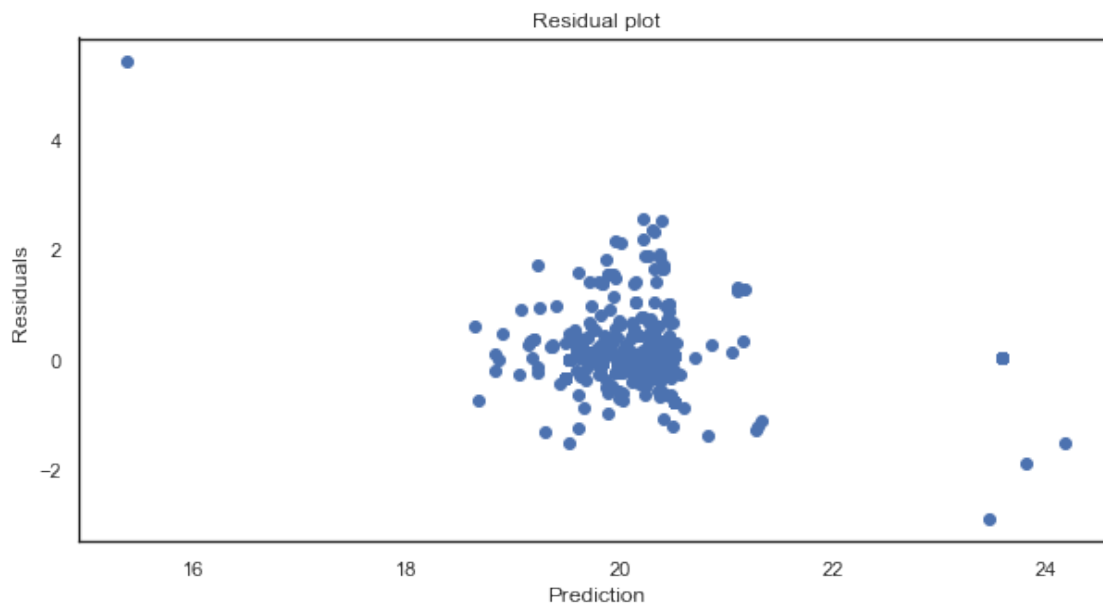
3.1 Section 6.1: Plotting the Residuals

```
[107]: # Defining Graph size
plt.figure(figsize=(10,5))

# Defining the residue and model predicted results
df_categorized_car_brands['PREDICTIONS'] = results.predict(X_test)
df_categorized_car_brands['RESIDUE'] = y_test - model_test_pred

# Plot your predicted values on the x-axis, and your residuals on the y-axis on
↳Residue Plot
plt.scatter(df_categorized_car_brands['PREDICTIONS'],
↳df_categorized_car_brands['RESIDUE'])
plt.title("Residual plot")
plt.xlabel("Prediction")
plt.ylabel("Residuals")
```

```
[107]: Text(0, 0.5, 'Residuals')
```



3.2 Section 6.2: QQ Plot

```
[108]: # Defining the residue and model predicted results
df_categorized_car_brands['PREDICTIONS'] = results.predict(X_test)
df_categorized_car_brands['RESIDUE'] = y_test - model_test_pred

# diagnose/inspect residual normality using QQplot:
plt.figure(figsize=(10,10))
```

```
stats.probplot(df_categorized_car_brands['RESIDUE'], dist="norm", plot=plt)
plt.title("Normal Q-Q plot")
```

```
D:\Application\Anaconda2\envs\py3\lib\site-
packages\scipy\stats\_distn_infrastructure.py:901: RuntimeWarning: invalid value
encountered in greater
```

```
    return (a < x) & (x < b)
```

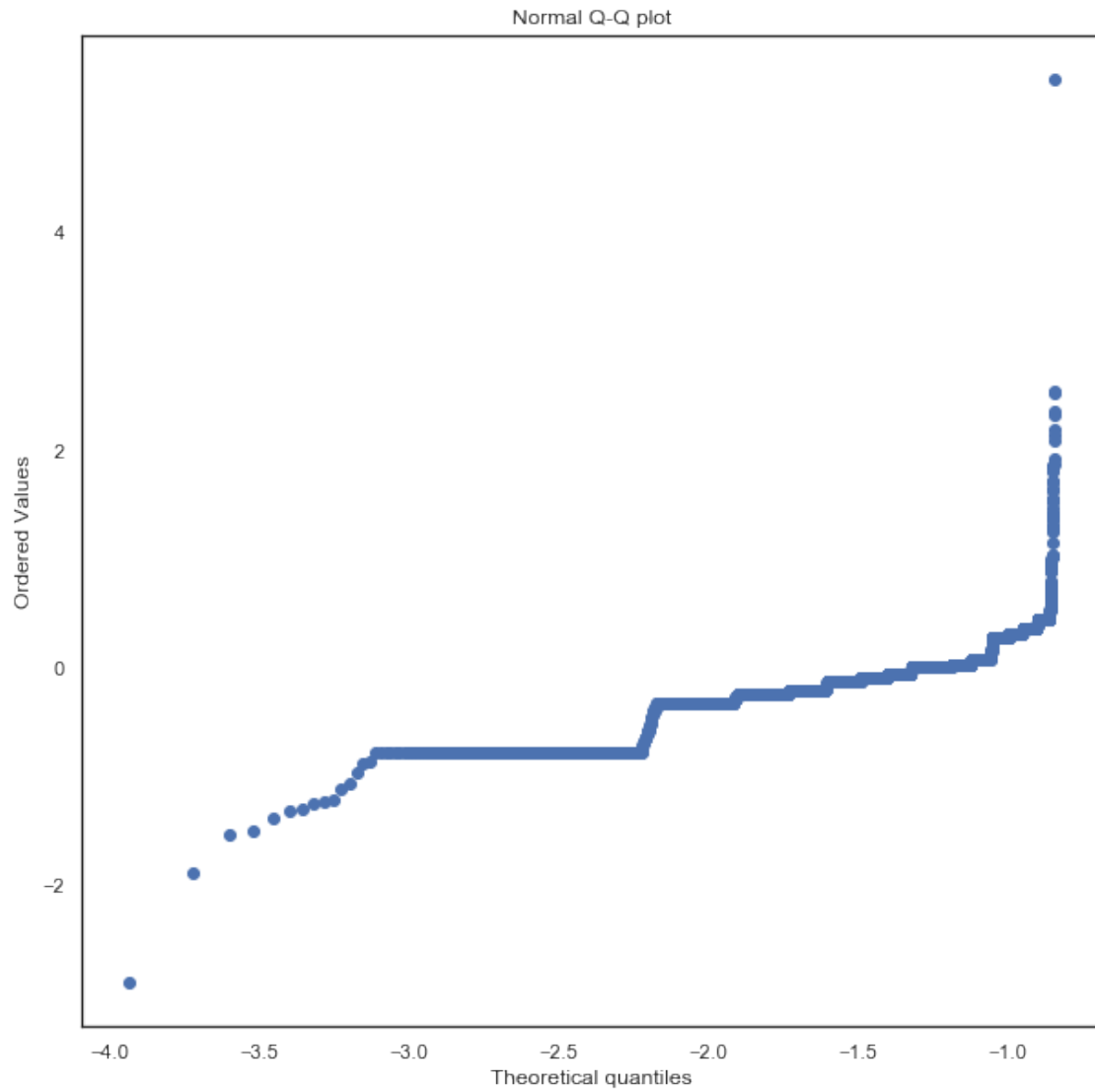
```
D:\Application\Anaconda2\envs\py3\lib\site-
packages\scipy\stats\_distn_infrastructure.py:901: RuntimeWarning: invalid value
encountered in less
```

```
    return (a < x) & (x < b)
```

```
D:\Application\Anaconda2\envs\py3\lib\site-
packages\scipy\stats\_distn_infrastructure.py:1892: RuntimeWarning: invalid
value encountered in less_equal
```

```
    cond2 = cond0 & (x <= _a)
```

```
[108]: Text(0.5, 1.0, 'Normal Q-Q plot')
```



[]: