Exploring eBay Car Sales Data

In this project, I am going to work with dataset of used cars from *eBay Kleinanzeigen* (classified section of German eBay webpage).

You can access the original dataset from [here (https://data.world/data-society/used-cars-data)](https://data.world/data-society/used-cars-data)

The original dataset differs from the dataset that I am working with in this analysis due to two reasons:

- 50,000 data points were selected from the full dataset to ensure high performance.
- Dataset was dirtied. (The original dataset is cleaner. Dataset was dirtied for a training of data cleaning skills.)

  Dictionary:
  - **dateCrawled** - When this ad was first crawled.
  - **name** - Name of the car.
  - **seller** - Whether the seller is private or dealer.
  - **offerType** - The type of listing.
  - **price** - The price on the ad to sell the car.
  - **abtest** - Whether the listing is included in an A/B test.
  - **vehicleType** - Type of the vehicle.
  - **yearOfRegistration** - The year in which the car was first registered.
  - **gearbox** - The transmission type.
  - **powerPS** - The power of the car in PS.
  - **model** - The car model name.
  - **odometer** - How many kilometers the car has driven.
  - **monthOfRegistration** - The month in which the car was first registered.
  - **fuelType** - Type of fuel that car uses.
  - **brand** - The brand of the car.
  - **notRepairedDamage** - If the car has a damage which is not yet repaired.
  - **dateCreated** - The date on which the eBay listing was created.
  - **nrOfPictures** - The number of pictures in the ad.
  - **postalCode** - The postal code for the location of the vehicle.
  - **lastSeenOnline** - When the crawler saw this ad last online.

  My goal of the project is to clean the data and analyze the car listings in this dataset.

Introduction - importing libraries and the dataset

In [1]:

```
import pandas as pd
import numpy as np

# autos = pd.read_csv('autos.csv')
# We can't read the file with default UTF-8 encoding

autos = pd.read_csv('autos.csv', encoding = 'Latin-1')
# Another encoding type also works if you want to try:
# autos = pd.read_csv('autos.csv', encoding = 'Windows-1252')
```

In [2]:

```
autos # The data is very messy and raw. I should start cleaning it.
autos.info() # There are also missing values in few columns.
autos.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 20 columns):
 #   Column               Non-Null Count  Dtype 
---  ------               --------------  ----- 
 0   dateCrawled          50000 non-null  object
 1   name                 50000 non-null  object
 2   seller               50000 non-null  object
 3   offerType            50000 non-null  object
 4   price                50000 non-null  object
 5   abtest               50000 non-null  object
 6   vehicleType          44905 non-null  object
 7   yearOfRegistration   50000 non-null  int64 
 8   gearbox              47320 non-null  object
 9   powerPS              50000 non-null  int64 
 10  model                47242 non-null  object
 11  odometer             50000 non-null  object
 12  monthOfRegistration  50000 non-null  int64 
 13  fuelType             45518 non-null  object
 14  brand                50000 non-null  object
 15  notRepairedDamage    40171 non-null  object
 16  dateCreated          50000 non-null  object
 17  nrOfPictures         50000 non-null  int64 
 18  postalCode           50000 non-null  int64 
 19  lastSeen             50000 non-null  object
dtypes: int64(5), object(15)
memory usage: 7.6+ MB
```

Out[2]:

	dateCrawled	name	seller	offerType	price	abtest	vehicleType	yearOfRegistration	gearbox	powe
0	2016-03-26 17:47:46	Peugeot_807_160_NAVTECH_ON_BOARD	privat	Angebot	$5,000	control	bus	2004	manuell	
1	2016-04-04 13:38:56	BMW_740i_4_4_Liter_HAMANN_UMBAU_Mega_Optik	privat	Angebot	$8,500	control	limousine	1997	automatik	
2	2016-03-26 18:57:24	Volkswagen_Golf_1.6_United	privat	Angebot	$8,990	test	limousine	2009	manuell	
3	2016-03-12 16:58:10	Smart_smart_fortwo_coupe_softouch/F1/Klima/Pan...	privat	Angebot	$4,350	control	kleinwagen	2007	automatik	
4	2016-04-01 14:38:50	Ford_Focus_1_6_Benzin_TÜV_neu_ist_sehr_gepfleg...	privat	Angebot	$1,350	test	kombi	2003	manuell	

We can see several issues in the dataset:

- 1. There are a lot of null values in some columns.
- 2. Column names are also not clear, as columns are written in camelcase.

Cleaning Column Names

In [3]:

```
autos.columns
```

Out[3]:

```
Index(['dateCrawled', 'name', 'seller', 'offerType', 'price', 'abtest',
       'vehicleType', 'yearOfRegistration', 'gearbox', 'powerPS', 'model',
       'odometer', 'monthOfRegistration', 'fuelType', 'brand',
       'notRepairedDamage', 'dateCreated', 'nrOfPictures', 'postalCode',
       'lastSeen'],
      dtype='object')
```

- I will change the columns from camelcase to snakecase
- I will change the wordings of the columns so they will be more accurate and understandable.

In [4]:

```python
autos.columns = ['date_crawled', 'name', 'seller', 'offer_type', 'price', 'ab_test',
        'vehicle_type', 'registration_year', 'gearbox', 'power_ps', 'model',
        'odometer', 'registration_month', 'fuel_type', 'brand',
        'unrepaired_damage', 'ad_created', 'num_photos', 'postal_code',
        'last_seen']
autos.head() # Columns are cleaner now.
```

Out[4]:

	date_crawled	name	seller	offer_type	price	ab_test	vehicle_type	registration_year	gearbox	pov
0	2016-03-26 17:47:46	Peugeot_807_160_NAVTECH_ON_BOARD	privat	Angebot	$5,000	control	bus	2004	manuell	
1	2016-04-04 13:38:56	BMW_740i_4_4_Liter_HAMANN_UMBAU_Mega_Optik	privat	Angebot	$8,500	control	limousine	1997	automatik	
2	2016-03-26 18:57:24	Volkswagen_Golf_1.6_United	privat	Angebot	$8,990	test	limousine	2009	manuell	
3	2016-03-12 16:58:10	Smart_smart_fortwo_coupe_softouch/F1/Klima/Pan...	privat	Angebot	$4,350	control	kleinwagen	2007	automatik	
4	2016-04-01 14:38:50	Ford_Focus_1_6_Benzin_TÜV_neu_ist_sehr_gepfleg...	privat	Angebot	$1,350	test	kombi	2003	manuell	

Initial Exploration and Cleaning

Now let's do some basic data exploration:

- Text columns where all or almost all values are the same should be removed, as they are not useful for the analysis.
- Numeric data stored in text format which can be cleaned and converted.

In [5]:

```python
autos.describe(include = 'all')
```

Out[5]:

	date_crawled	name	seller	offer_type	price	ab_test	vehicle_type	registration_year	gearbox	power_ps	model	odometer	reg
count	50000	50000	50000	50000	50000	50000	44905	50000.000000	47320	50000.000000	47242	50000	
unique	48213	38754	2	2	2357	2	8	NaN	2	NaN	245	13	
top	2016-04-02 11:37:04	Ford_Fiesta	privat	Angebot	$0	test	limousine	NaN	manuell	NaN	golf	150,000km	
freq	3	78	49999	49999	1421	25756	12859	NaN	36993	NaN	4024	32424	
mean	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2005.073280	NaN	116.355920	NaN	NaN	
std	NaN	NaN	NaN	NaN	NaN	NaN	NaN	105.712813	NaN	209.216627	NaN	NaN	
min	NaN	NaN	NaN	NaN	NaN	NaN	NaN	1000.000000	NaN	0.000000	NaN	NaN	
25%	NaN	NaN	NaN	NaN	NaN	NaN	NaN	1999.000000	NaN	70.000000	NaN	NaN	
50%	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2003.000000	NaN	105.000000	NaN	NaN	
75%	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2008.000000	NaN	150.000000	NaN	NaN	
max	NaN	NaN	NaN	NaN	NaN	NaN	NaN	9999.000000	NaN	17700.000000	NaN	NaN	

Columns `seller`, `offer_type` and `num_photos` look odd. These columns should ve investigated further.

In [6]:

```python
autos['seller'].value_counts()

```

Out[6]:

```
privat        49999
gewerblich        1
Name: seller, dtype: int64
```

In [7]:

```python
autos['offer_type'].value_counts()
```

Out[7]:

```
Angebot    49999
Gesuch         1
Name: offer_type, dtype: int64
```

In [8]:
```
autos['num_photos'].value_counts()
```

Out[8]:
```
0    50000
Name: num_photos, dtype: int64
```

- 1. `seller` and `offer_type` are columns where nearly all of the values are the same.
- 2. `num_photos` has 0 for every column.

We are going to drop these 3 columns as they are not useful for the data analysis.

In [9]:
```
autos = autos.drop(['seller', 'num_photos', 'offer_type'], axis = 1)
```

In [10]:
```
autos.columns # Columns are sucessfully dropped.
```

Out[10]:
```
Index(['date_crawled', 'name', 'price', 'ab_test', 'vehicle_type',
       'registration_year', 'gearbox', 'power_ps', 'model', 'odometer',
       'registration_month', 'fuel_type', 'brand', 'unrepaired_damage',
       'ad_created', 'postal_code', 'last_seen'],
      dtype='object')
```

Let's look which columns have numeric data written in the text format. I am going to clean data of such columns and turn it into numeric type.

In [11]:
```
autos.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 17 columns):
 #   Column              Non-Null Count  Dtype 
---  ------              --------------  ----- 
 0   date_crawled        50000 non-null  object
 1   name                50000 non-null  object
 2   price               50000 non-null  object
 3   ab_test             50000 non-null  object
 4   vehicle_type        44905 non-null  object
 5   registration_year   50000 non-null  int64 
 6   gearbox             47320 non-null  object
 7   power_ps            50000 non-null  int64 
 8   model               47242 non-null  object
 9   odometer            50000 non-null  object
 10  registration_month  50000 non-null  int64 
 11  fuel_type           45518 non-null  object
 12  brand               50000 non-null  object
 13  unrepaired_damage   40171 non-null  object
 14  ad_created          50000 non-null  object
 15  postal_code         50000 non-null  int64 
 16  last_seen           50000 non-null  object
dtypes: int64(4), object(13)
memory usage: 6.5+ MB
```

It can be observed that columns `price` and `odometer` columns have numeric values written in text form.

Let's start from `price` column.

In [12]:
```
autos['price'].value_counts() # We should remove $ and , signs
autos['price'] = autos['price'].str.replace('$','').str.replace(',','').astype(int)

```

```
C:\Users\Beibarys Nyussupov\AppData\Local\Temp\ipykernel_46968\222734509.py:2: FutureWarning: The default value of
regex will change from True to False in a future version. In addition, single character regular expressions will *n
ot* be treated as literal strings when regex=True.
  autos['price'] = autos['price'].str.replace('$','').str.replace(',','').astype(int)
```

In [13]:
```
autos['price'].head() # Completed!

```

Out[13]:
```
0    5000
1    8500
2    8990
3    4350
4    1350
Name: price, dtype: int32
```

Now, let's change values of `odometer` column.

In [14]:
```
autos['odometer'].value_counts()
```

Out[14]:
```
150,000km    32424
125,000km     5170
100,000km     2169
90,000km      1757
80,000km      1436
70,000km      1230
60,000km      1164
50,000km      1027
5,000km        967
40,000km       819
30,000km       789
20,000km       784
10,000km       264
Name: odometer, dtype: int64
```

In [15]:
```
autos['odometer'] = autos['odometer'].str.replace(',','').str.replace('km','').astype(int)
```

In [16]:
```
autos['odometer'].head() # Completed!
```

Out[16]:
```
0    150000
1    150000
2     70000
3     70000
4    150000
Name: odometer, dtype: int32
```

We cleaned values of `odometer` column and transformed these values into int-64 form. However, let's make a change to the name of the odometer column to make accurate description of these values.

In [17]:
```
autos.rename({'odometer':'odometer_km'}, axis = 1, inplace = True)

```

In [18]:
```
autos['odometer_km'].head()
```

Out[18]:
```
0    150000
1    150000
2     70000
3     70000
4    150000
Name: odometer_km, dtype: int32
```

Translating German words to English words

In [19]:
```
autos.head()
```

Out[19]:

	date_crawled	name	price	ab_test	vehicle_type	registration_year	gearbox	power_ps	model	odo
0	2016-03-26 17:47:46	Peugeot_807_160_NAVTECH_ON_BOARD	5000	control	bus	2004	manuell	158	andere	
1	2016-04-04 13:38:56	BMW_740i_4_4_Liter_HAMANN_UMBAU_Mega_Optik	8500	control	limousine	1997	automatik	286	7er	
2	2016-03-26 18:57:24	Volkswagen_Golf_1.6_United	8990	test	limousine	2009	manuell	102	golf	
3	2016-03-12 16:58:10	Smart_smart_fortwo_coupe_softouch/F1/Klima/Pan...	4350	control	kleinwagen	2007	automatik	71	fortwo	
4	2016-04-01 14:38:50	Ford_Focus_1_6_Benzin_TÜV_neu_ist_sehr_gepfleg...	1350	test	kombi	2003	manuell	0	focus	

In [20]:
```
autos['vehicle_type'].unique()
```

Out[20]:
```
array(['bus', 'limousine', 'kleinwagen', 'kombi', nan, 'coupe', 'suv',
       'cabrio', 'andere'], dtype=object)
```

First column that has entries with German language is - `vehicle_type` . It is an important column and we need to change entries written in German to English language.

`Kleinwagen` - Compact/Subcompact

`Kombi` - Wagon

`andere` - other

We also have NA values which should be grouped into other type of cars too.

In [21]:
```
before = autos['vehicle_type'].value_counts()
```

In [22]:
```
before
```

Out[22]:
```
limousine     12859
kleinwagen    10822
kombi          9127
bus            4093
cabrio         3061
coupe          2537
suv            1986
andere          420
Name: vehicle_type, dtype: int64
```

In [23]:
```
autos['vehicle_type'] = autos['vehicle_type'].replace('kleinwagen', 'compact/subcompact')
autos['vehicle_type'] = autos['vehicle_type'].replace('kombi', 'wagon')
autos['vehicle_type'] = autos['vehicle_type'].replace('andere', 'other')
autos['vehicle_type'] = autos['vehicle_type'].fillna('undefined')

```

In [24]:
```
after = autos['vehicle_type'].value_counts().sort_values(ascending = False)

```

In [25]:
```
after
```

Out[25]:
```
limousine             12859
compact/subcompact    10822
wagon                  9127
undefined              5095
bus                    4093
cabrio                 3061
coupe                  2537
suv                    1986
other                   420
Name: vehicle_type, dtype: int64
```

Everything in `vehichle_type` column was cleaned. Now every vehicle type is clear and understandable in English language.

Limousine, compact/subcompact, wagon, bus and cabrio are vehicle types with the most number of car listings on the website.

The next two columns that we need to clean are `gearbox` and `unrepaired_damage`

`automatik` - automatic

`manuell` - manual

`nein` - no

`ja` - yes

In [26]:
```
autos['gearbox'].unique()
```

Out[26]:
```
array(['manuell', 'automatik', nan], dtype=object)
```

In [27]:
```
autos['gearbox'].value_counts()
```

Out[27]:
```
manuell      36993
automatik    10327
Name: gearbox, dtype: int64
```

In [28]:
```
autos['gearbox'] = autos['gearbox'].replace('manuell', 'manual')
autos['gearbox'] = autos['gearbox'].replace('automatik', 'automatic')
autos['gearbox'] = autos['gearbox'].fillna('undefined')
```

In [29]:
```
autos['gearbox'].value_counts()
```

Out[29]:
```
manual       36993
automatic    10327
undefined     2680
Name: gearbox, dtype: int64
```

In [30]:
```
autos['unrepaired_damage'].unique()
```

Out[30]:
```
array(['nein', nan, 'ja'], dtype=object)
```

In [31]:
```
autos['unrepaired_damage'] = autos['unrepaired_damage'].replace('nein', 'no')
autos['unrepaired_damage'] = autos['unrepaired_damage'].replace('ja', 'yes')
autos['unrepaired_damage'] = autos['unrepaired_damage'].fillna('undefined')
```

In [32]:
```
autos['unrepaired_damage'].value_counts()
```

Out[32]:
```
no           35232
undefined     9829
yes           4939
Name: unrepaired_damage, dtype: int64
```

The rest two columns were cleaned and entries were transformed into understandable English language. Most of the cars on the market are on manual engine and do not have unrepaired damage.

Exploring the Odometer and Price Columns

In [33]:
```
autos['odometer_km'].value_counts().sort_index(ascending = True)
```

Out[33]:
```
5000        967
10000       264
20000       784
30000       789
40000       819
50000      1027
60000      1164
70000      1230
80000      1436
90000      1757
100000     2169
125000     5170
150000    32424
Name: odometer_km, dtype: int64
```

In [34]:
```
autos['odometer_km'].describe()
```

Out[34]:
```
count     50000.000000
mean     125732.700000
std       40042.211706
min        5000.000000
25%      125000.000000
50%      150000.000000
75%      150000.000000
max      150000.000000
Name: odometer_km, dtype: float64
```

Mileage values of cars are rounded, which means that sellers of vehicles had to choose from pre-existing options on the market to define the number of kilometres driven by each car. Furthermore, there are no listed cars with 0 mileage on the market. However it can be a drawback of prexisting options on eBay, as if there are cars that have a mileage below 5000 kilometres - sellers still had to choose 5000 due to limited options.

In [35]:
```
autos['price'].unique().shape # Number of unique values in price column
```

Out[35]:
```
(2357,)
```

In [36]:
```
autos['price'].value_counts().head(20).sort_index(ascending = True)
```

Out[36]:
```
0       1421
300      384
500      781
600      531
650      419
700      395
750      433
800      498
850      410
900      420
950      379
999      434
1000     639
1200     639
1500     734
2000     460
2200     382
2500     643
3500     498
4500     394
Name: price, dtype: int64
```

Again, values are very rounded. However, given that there are 2357 unique values in the price column, it means that sellers are just tended to round numbers and they are not limited to pre-existing options on eBay webpage.

In [37]:
```
autos['price'].describe()

```

Out[37]:
```
count    5.000000e+04
mean     9.840044e+03
std      4.811044e+05
min      0.000000e+00
25%      1.100000e+03
50%      2.950000e+03
75%      7.200000e+03
max      1.000000e+08
Name: price, dtype: float64
```

In [38]:

```
autos['price'].value_counts().sort_index(ascending = True).head(50)
```

Out[38]:

```
0      1421
1       156
2         3
3         1
5         2
8         1
9         1
10        7
11        2
12        3
13        2
14        1
15        2
17        3
18        1
20        4
25        5
29        1
30        7
35        1
40        6
45        4
47        1
49        4
50       49
55        2
59        1
60        9
65        5
66        1
70       10
75        5
79        1
80       15
89        1
90        5
99       19
100     134
110       3
111       2
115       2
117       1
120      39
122       1
125       8
129       1
130      15
135       1
139       1
140       9
Name: price, dtype: int64
```

In [39]:

```
autos['price'].value_counts().sort_index(ascending = False).head(20)
```

Out[39]:

```
99999999    1
27322222    1
12345678    3
11111111    2
10000000    1
3890000     1
1300000     1
1234566     1
999999      2
999990      1
350000      1
345000      1
299000      1
295000      1
265000      1
259000      1
250000      1
220000      1
198000      1
197000      1
Name: price, dtype: int64
```

In [40]:

```
(len(autos[autos['price'] > 350000]) / len(autos)) * 100 # Percentage of cars with a cell price higher than 350

```

Out[40]: 0.027999999999999997

There are plenty of car listings with a prices below 30 dollars and 1421 listings with 0 prices. There are also 14 listings with prices above 350 000 dollars.

Assuming that eBay is an auction website, there can be cars that where the opening equals to 1 dollar. However, I am going to remove all car listings with prices above 350 000 $ as prices increase regularly and then jummp up to very high and less realistic numbers.

In [41]:
```
autos = autos[autos['price'].between(1, 350000)]
```

In [42]:
```
autos['price'].describe() # We removed the outliers in prices
```

Out[42]:
```
count     48565.000000
mean       5888.935591
std        9059.854754
min           1.000000
25%        1200.000000
50%        3000.000000
75%        7490.000000
max      350000.000000
Name: price, dtype: float64
```

Exploring the date columns

Columns with date information:

- `date_crawled`
- `last_seen`
- `ad_created`
- `registration_month`
- `registration_year`

There is a mix of information that was created by crawler and information that was produced by website itself. Let's explore these columns a bit more.

In [43]:
```
autos[['date_crawled', 'ad_created', 'last_seen']][0:5]
```

Out[43]:

	date_crawled	ad_created	last_seen
0	2016-03-26 17:47:46	2016-03-26 00:00:00	2016-04-06 06:45:54
1	2016-04-04 13:38:56	2016-04-04 00:00:00	2016-04-06 14:45:08
2	2016-03-26 18:57:24	2016-03-26 00:00:00	2016-04-06 20:15:37
3	2016-03-12 16:58:10	2016-03-12 00:00:00	2016-03-15 03:16:28
4	2016-04-01 14:38:50	2016-04-01 00:00:00	2016-04-01 14:38:50

In [44]:
```
(autos['date_crawled'].str[:10].value_counts(normalize = True, dropna = False).sort_index(ascending = True)) * 1
```

Out[44]:
```
2016-03-05    2.532688
2016-03-06    1.404304
2016-03-07    3.601359
2016-03-08    3.329558
2016-03-09    3.308967
2016-03-10    3.218367
2016-03-11    3.257490
2016-03-12    3.691959
2016-03-13    1.566972
2016-03-14    3.654896
2016-03-15    3.428395
2016-03-16    2.960980
2016-03-17    3.162772
2016-03-18    1.291053
2016-03-19    3.477813
2016-03-20    3.788737
2016-03-21    3.737259
2016-03-22    3.298672
2016-03-23    3.222485
2016-03-24    2.934212
2016-03-25    3.160712
2016-03-26    3.220426
2016-03-27    3.109235
2016-03-28    3.486050
2016-03-29    3.409863
2016-03-30    3.368681
2016-03-31    3.183363
2016-04-01    3.368681
2016-04-02    3.547823
2016-04-03    3.860805
2016-04-04    3.648718
2016-04-05    1.309585
2016-04-06    0.317101
2016-04-07    0.140019
Name: date_crawled, dtype: float64
```

According to the data, the website was crawled daily in a period of one month from April to March 2016.

In [45]:
```
(autos['last_seen'].str[:10].value_counts(normalize = True, dropna = False).sort_index(ascending = True))* 100
```

Out[45]:
```
2016-03-05     0.107073
2016-03-06     0.432410
2016-03-07     0.539483
2016-03-08     0.741275
2016-03-09     0.959539
2016-03-10     1.066612
2016-03-11     1.237517
2016-03-12     2.378256
2016-03-13     0.889529
2016-03-14     1.260167
2016-03-15     1.587563
2016-03-16     1.645218
2016-03-17     2.808607
2016-03-18     0.735097
2016-03-19     1.583445
2016-03-20     2.065273
2016-03-21     2.063214
2016-03-22     2.137342
2016-03-23     1.853186
2016-03-24     1.976732
2016-03-25     1.921137
2016-03-26     1.680222
2016-03-27     1.564913
2016-03-28     2.085864
2016-03-29     2.234119
2016-03-30     2.477093
2016-03-31     2.378256
2016-04-01     2.279419
2016-04-02     2.491506
2016-04-03     2.520334
2016-04-04     2.448265
2016-04-05    12.476063
2016-04-06    22.180583
2016-04-07    13.194688
Name: last_seen, dtype: float64
```

`last_seen` column has a data recorded by crawler which shows the last date on which any car listing was seen. Looking on these values can help us with determining on what day the listing was removed, possibly because the car was sold.

Last three days of March (05, 06, 07) have disproportinate `last_seen` percentages, which are around 10 times bigger from other days. It is very unlikely that these 3 days experienced a rapid hike in sales and it is more likely that these values are related to crawling period ending and don't show spike in in car sales.

In [46]:
```
(autos['ad_created'].str[:10].value_counts(normalize = True, dropna = False).sort_index(ascending = True)) * 10
```

Out[46]:
```
2015-06-11    0.002059
2015-08-10    0.002059
2015-09-09    0.002059
2015-11-10    0.002059
2015-12-05    0.002059
                ...
2016-04-03    3.885514
2016-04-04    3.685782
2016-04-05    1.181921
2016-04-06    0.325337
2016-04-07    0.125605
Name: ad_created, Length: 76, dtype: float64
```

In [47]:
```
print('Number of dates crawled:', autos['date_crawled'].str[:10].unique().shape)
print('Number of dates ad_created:', autos['ad_created'].str[:10].unique().shape)
(autos['ad_created'].str[:10].value_counts(normalize = True, dropna = False).sort_index(ascending = True)) * 10
```

```
Number of dates crawled: (34,)
Number of dates ad_created: (76,)
```

Out[47]:
```
2015-06-11    0.002059
2015-08-10    0.002059
2015-09-09    0.002059
2015-11-10    0.002059
2015-12-05    0.002059
                ...
2016-04-03    3.885514
2016-04-04    3.685782
2016-04-05    1.181921
2016-04-06    0.325337
2016-04-07    0.125605
Name: ad_created, Length: 76, dtype: float64
```

There is a variety of dates when car listings were created. Some of them are within 1-2 months of the listing date, but some other dates are old and few of them even fall within around 8-9 months.

We can also turn dates in `date_crawled` , `ad_created` and `last_seen` columns to uniform numeric data.

In [48]:
```
autos['date_crawled'] = autos['date_crawled'].str.replace('-','').str.split(' ').str[0].astype(int)
autos['ad_created'] = autos['ad_created'].str.replace('-','').str.split(' ').str[0].astype(int)
autos['last_seen'] = autos['last_seen'].str.replace('-','').str.split(' ').str[0].astype(int)
```

In [49]:
```
autos['date_crawled'].head()
```

Out[49]:
```
0    20160326
1    20160404
2    20160326
3    20160312
4    20160401
Name: date_crawled, dtype: int32
```

In [50]:
```
autos['ad_created'].head()
```

Out[50]:
```
0    20160326
1    20160404
2    20160326
3    20160312
4    20160401
Name: ad_created, dtype: int32
```

In [51]:
```
autos['last_seen'].head()
```

Out[51]:
```
0    20160406
1    20160406
2    20160406
3    20160315
4    20160401
Name: last_seen, dtype: int32
```

In [52]:
```
autos.head()
```

Out[52]:

	date_crawled	name	price	ab_test	vehicle_type	registration_year	gearbox	power_ps	mo
0	20160326	Peugeot_807_160_NAVTECH_ON_BOARD	5000	control	bus	2004	manual	158	ande
1	20160404	BMW_740i_4_4_Liter_HAMANN_UMBAU_Mega_Optik	8500	control	limousine	1997	automatic	286	7
2	20160326	Volkswagen_Golf_1.6_United	8990	test	limousine	2009	manual	102	g
3	20160312	Smart_smart_fortwo_coupe_softouch/F1/Klima/Pan...	4350	control	compact/subcompact	2007	automatic	71	fort
4	20160401	Ford_Focus_1_6_Benzin_TÜV_neu_ist_sehr_gepfleg...	1350	test	wagon	2003	manual	0	foc

In [53]:
```
autos['registration_year'].describe()
```

Out[53]:
```
count    48565.000000
mean      2004.755421
std         88.643887
min       1000.000000
25%       1999.000000
50%       2004.000000
75%       2008.000000
max       9999.000000
Name: registration_year, dtype: float64
```

It is more likely that the date when the car was first registered will indicate the age of the car. Some strange values can be seen in above statistics. Minimum registration year is `1000` when cars even weren't used at that time and maximum is `9999` - a year which we even didn't reach yet.

Dealing with Incorrect Registration Year Data

Because a car can't be registered after the listing was seen, any vehicle with a registration year above 2016 is definitely inaccurate. However determining the earliest valid year is more complex. It is more likely to be somewhere in 1900s.

Let's count the number of listings with cars that fall outside the 1900-2016 interval and see if it's safe to tremove those rows entirely, or if we need more custom logic.

In [54]:
```
((~autos["registration_year"].between(1900,2016)).sum() / autos.shape[0]) * 100
```

Out[54]:
```
3.8793369710697
```

As the inaccurate dates are less than 4% of the data, we can safely remove such rows.

In [55]:
```
autos = autos[autos["registration_year"].between(1900,2016)]
(autos["registration_year"].value_counts(normalize = True, dropna = False).sort_index(ascending = False).head(25
```

Out[55]:
```
2016    2.613483
2015    0.839742
2014    1.420278
2013    1.720186
2012    2.806281
2011    3.476789
2010    3.403954
2009    4.466485
2008    4.744971
2007    4.877788
2006    5.719672
2005    6.289497
2004    5.790364
2003    5.781796
2002    5.325507
2001    5.646837
2000    6.760781
1999    6.205951
1998    5.062017
1997    4.179431
1996    2.941239
1995    2.628478
1994    1.347443
1993    0.910435
1992    0.792614
Name: registration_year, dtype: float64
```

In [56]:
```
(autos['registration_year'].value_counts(normalize = True).head(25))*100
```

Out[56]:
```
2000    6.760781
2005    6.289497
1999    6.205951
2004    5.790364
2003    5.781796
2006    5.719672
2001    5.646837
2002    5.325507
1998    5.062017
2007    4.877788
2008    4.744971
2009    4.466485
1997    4.179431
2011    3.476789
2010    3.403954
1996    2.941239
2012    2.806281
1995    2.628478
2016    2.613483
2013    1.720186
2014    1.420278
1994    1.347443
1993    0.910435
2015    0.839742
1992    0.792614
Name: registration_year, dtype: float64
```

As inaccurate rows were removed, it can be seen that most of the cars were first registered in past 22 years.

Exploring Price by Brand

In [57]:
```
(autos['brand'].value_counts(normalize = True)) * 100
```

Out[57]:
```
volkswagen        21.126368
bmw               11.004477
opel              10.758124
mercedes_benz      9.646323
audi               8.656627
ford               6.989996
renault            4.714980
peugeot            2.984083
fiat               2.564212
seat               1.827296
skoda              1.640925
nissan             1.527388
mazda              1.518819
smart              1.415994
citroen            1.400998
toyota             1.270324
hyundai            1.002549
sonstige_autos     0.981127
volvo              0.914719
mini               0.876159
mitsubishi         0.822604
honda              0.784045
kia                0.706926
alfa_romeo         0.664082
porsche            0.612669
suzuki             0.593389
chevrolet          0.569825
chrysler           0.351321
dacia              0.263490
daihatsu           0.250637
jeep               0.227073
subaru             0.214220
land_rover         0.209936
saab               0.164949
jaguar             0.156381
daewoo             0.149954
trabant            0.139243
rover              0.132816
lancia             0.107110
lada               0.057839
Name: brand, dtype: float64
```

In [58]:
```python
(autos['brand'].value_counts(normalize = True).head(7))*100
```

Out[58]:
```
volkswagen       21.126368
bmw              11.004477
opel             10.758124
mercedes_benz     9.646323
audi              8.656627
ford              6.989996
renault           4.714980
Name: brand, dtype: float64
```

There are a wide range of car brands that do not have significant percentages of listings, that's why I have filtered down the data to the manufacturers that have more than 4% of total listings.

German transport manufacturers represent top 5 brands with around 60% of overall listings. Volkswagen is the most popular car brand in the market having approximately 21% of overall listings.

In [59]:
```python
brand_counts = (autos['brand'].value_counts(normalize = True)) * 100
top_brands = brand_counts[brand_counts > 4].index
print(top_brands)
```

```
Index(['volkswagen', 'bmw', 'opel', 'mercedes_benz', 'audi', 'ford',
       'renault'],
      dtype='object')
```

In [60]:
```python
mean_prices = {}
for brand in top_brands:
    names = autos[autos['brand'] == brand]
    price = names['price'].mean()
    mean_prices[brand] = int(price)

mean_prices
```

Out[60]:
```
{'volkswagen': 5402,
 'bmw': 8332,
 'opel': 2975,
 'mercedes_benz': 8628,
 'audi': 9336,
 'ford': 3749,
 'renault': 2474}
```

According to the data:

```
- Audi, Bmw and Mercedes Benz are more expensive car brands.
- Ford, Openal and Renault are less expensive car manufacturers.
- Folkswagen - It is a car brand that is "something in between",
which also explains the popularity of given brand.
```

Exploring mileage

In [61]:
```python
bmp_series = pd.Series(mean_prices).sort_values(ascending = False)
price_df = pd.DataFrame(bmp_series, columns = ['mean_price($)'])
price_df
```

Out[61]:

	mean_price($)
audi	9336
mercedes_benz	8628
bmw	8332
volkswagen	5402
ford	3749
opel	2975
renault	2474

In [62]:

```python
mean_mileage = {}
for brand in top_brands:
    names = autos[autos['brand'] == brand]
    mileages = names['odometer_km'].mean()
    mean_mileage[brand] = int(mileages)

bmm_series = pd.Series(mean_mileage).sort_values(ascending  = False)


mileage_df = pd.DataFrame(bmm_series, columns = ['mean_mileage(km)'])
mileage_df
```

Out[62]:

	mean_mileage(km)
bmw	132572
mercedes_benz	130788
opel	129310
audi	129157
volkswagen	128707
renault	128071
ford	124266

In [63]:

```python
#Common dataframe
mileage_df['mean_price($)'] = price_df

mileage_df
```

Out[63]:

	mean_mileage(km)	mean_price($)
bmw	132572	8332
mercedes_benz	130788	8628
opel	129310	2975
audi	129157	9336
volkswagen	128707	5402
renault	128071	2474
ford	124266	3749

The range of kilometers passed doesn't really vary by car brand as prices do. Although, there is a weak trend that more expensive cars have more mileage than less expensive cars.