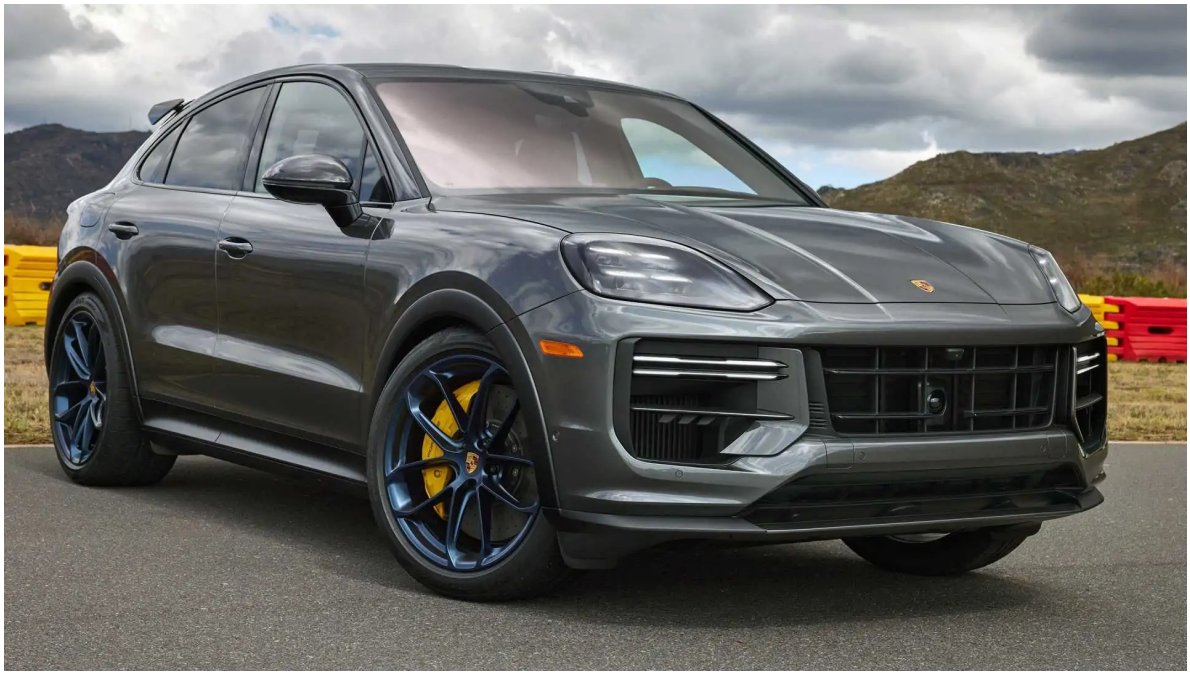
Analyzing Used Car Listings on eBay Kleinanzeigen

We'll work with a dataset of used cars from eBay Kleinanzeigen, a classifieds section of the German eBay website. The aim of this project is to clean the data and analyze the included used car listings.

The dataset was originally scraped and uploaded to Kaggle. The version of the dataset we are working with is a sample of 50,000 data points that was prepared by Dataquest including simulating a less-cleaned version of the data.



The data dictionary provided with data is as follows:

- dateCrawled - When this ad was first crawled. All field-values are taken from this date.
- name - Name of the car.
- seller - Whether the seller is private or a 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 - The vehicle Type.
- yearOfRegistration - The year in which which year the car was first registered.
- gearbox - The transmission type.
- powerPS - The power of the car in PS.
- model - The car model name.
- kilometer - How many kilometers the car has driven.
- monthOfRegistration - The month in which which year the car was first registered.
- fuelType - What type of fuel the 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.

The aim of this project is to clean the data and analyze the included used car listings. So let us begin by importing and reading the data.

Reading the data

In [208]:
```
import pandas as pd
import numpy as ny

autos = pd.read_csv("autos.csv", encoding = "Latin_1")
```

In [209]:
```
autos
```

Out[209]:

	dateCrawled	name	seller	offerType	pric
0	2016-03-26 17:47:46	Peugeot_807_160_NAVTECH_ON_BOARD	privat	Angebot	$5,00
1	2016-04-04 13:38:56	BMW_740i_4_4_Liter_HAMANN_UMBAU_Mega_Optik	privat	Angebot	$8,50
2	2016-03-26 18:57:24	Volkswagen_Golf_1.6_United	privat	Angebot	$8,99
3	2016-03-12 16:58:10	Smart_smart_fortwo_coupe_softouch/F1/Klima/Pan...	privat	Angebot	$4,35
4	2016-04-01 14:38:50	Ford_Focus_1_6_Benzin_TÜV_neu_ist_sehr_gepfleg...	privat	Angebot	$1,35
...	...	...	...	...	.
49995	2016-03-27 14:38:19	Audi_Q5_3.0_TDI_qu._S_tr.__Navi__Panorama__Xenon	privat	Angebot	$24,90
49996	2016-03-28 10:50:25	Opel_Astra_F_Cabrio_Bertone_Edition___TÜV_neu+...	privat	Angebot	$1,98
49997	2016-04-02 14:44:48	Fiat_500_C_1.2_Dualogic_Lounge	privat	Angebot	$13,20
49998	2016-03-08 19:25:42	Audi_A3_2.0_TDI_Sportback_Ambition	privat	Angebot	$22,90
49999	2016-03-14 00:42:12	Opel_Vectra_1.6_16V	privat	Angebot	$1,25

50000 rows × 20 columns

In [210]:
```
autos.info()
```

```
<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
```

In [211]:
```
autos.head()
```

Out[211]:

	dateCrawled	name	seller	offerType	price	abte
0	2016-03-26 17:47:46	Peugeot_807_160_NAVTECH_ON_BOARD	privat	Angebot	$5,000	conti
1	2016-04-04 13:38:56	BMW_740i_4_4_Liter_HAMANN_UMBAU_Mega_Optik	privat	Angebot	$8,500	conti
2	2016-03-26 18:57:24	Volkswagen_Golf_1.6_United	privat	Angebot	$8,990	te
3	2016-03-12 16:58:10	Smart_smart_fortwo_coupe_softouch/F1/Klima/Pan...	privat	Angebot	$4,350	conti
4	2016-04-01 14:38:50	Ford_Focus_1_6_Benzin_TÜV_neu_ist_sehr_gepfleg...	privat	Angebot	$1,350	te

From the above, we can observe:
- We have 5 columns with null values - vehicleType, gearbox, model, fuelType, notRepairedDamage - all of these contain less than 20% of null values.

- Data is in German, so in order to make it easier for the non-German speaking population, it might be a good idea to replace/translate certain columns to English.
- Most of the data is in the form of strings.

Renaming columns

In [212]:
```
autos.columns
```

Out[212]:
```
Index(['dateCrawled', 'name', 'seller', 'offerType', 'price', 'abtest',
       'vehicleType', 'yearOfRegistration', 'gearbox', 'powerPS', 'model',
       'odometer', 'monthOfRegistration', 'fuelType', 'brand',
       'notRepairedDamage', 'dateCreated', 'nrOfPictures', 'postalCode',
       'lastSeen'],
      dtype='object')
```

In [213]:
```
autos.columns = ['date_crawled', 'name', 'seller', 'offer_type', 'price', 'abt
       '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()
```

Out[213]:

	date_crawled	name	seller	offer_type	price	ab
0	2016-03-26 17:47:46	Peugeot_807_160_NAVTECH_ON_BOARD	privat	Angebot	$5,000	cor
1	2016-04-04 13:38:56	BMW_740i_4_4_Liter_HAMANN_UMBAU_Mega_Optik	privat	Angebot	$8,500	cor
2	2016-03-26 18:57:24	Volkswagen_Golf_1.6_United	privat	Angebot	$8,990	
3	2016-03-12 16:58:10	Smart_smart_fortwo_coupe_softouch/F1/Klima/Pan...	privat	Angebot	$4,350	cor
4	2016-04-01 14:38:50	Ford_Focus_1_6_Benzin_TÜV_neu_ist_sehr_gepfleg...	privat	Angebot	$1,350	

Now, we have column names in more understandable format, and in snakecase

Exploring and cleaning columns

In [214]:
```
autos.describe()
```

Out[214]:

	registration_year	power_ps	registration_month	num_photos	postal_code
count	50000.000000	50000.000000	50000.000000	50000.0	50000.000000
mean	2005.073280	116.355920	5.723360	0.0	50813.627300
std	105.712813	209.216627	3.711984	0.0	25779.747957
min	1000.000000	0.000000	0.000000	0.0	1067.000000
25%	1999.000000	70.000000	3.000000	0.0	30451.000000
50%	2003.000000	105.000000	6.000000	0.0	49577.000000
75%	2008.000000	150.000000	9.000000	0.0	71540.000000
max	9999.000000	17700.000000	12.000000	0.0	99998.000000

In [215]:
```
autos.describe(include='all')
```

Out[215]:

	date_crawled	name	seller	offer_type	price	abtest	vehicle_type	registration
count	50000	50000	50000	50000	50000	50000	44905	50000.0
unique	48213	38754	2	2	2357	2	8	
top	2016-03-08 10:40:35	Ford_Fiesta	privat	Angebot	$0	test	limousine	
freq	3	78	49999	49999	1421	25756	12859	
mean	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2005.0
std	NaN	NaN	NaN	NaN	NaN	NaN	NaN	105.7
min	NaN	NaN	NaN	NaN	NaN	NaN	NaN	1000.0
25%	NaN	NaN	NaN	NaN	NaN	NaN	NaN	1999.0
50%	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2003.0
75%	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2008.0

Let's do the low hanging fruit first - convert the price and odometer columns to numerical, and rename columns to include the unit of measure:

In [216]:
```
autos["price"].head()
```

Out[216]:
```
0    $5,000
1    $8,500
2    $8,990
3    $4,350
4    $1,350
Name: price, dtype: object
```

In [217]:
```
autos["price"] = (autos["price"]
                          .str.replace("$","")
                          .str.replace(",","")
                          .astype(int)
                          )
autos.rename({"price": "price_usd"}, axis=1, inplace=True)
autos["price_usd"].head()
```

Out[217]:
```
0    5000
1    8500
2    8990
3    4350
4    1350
Name: price_usd, dtype: int64
```

In [218]:
```
autos["odometer"].head()
```

Out[218]:
```
0    150,000km
1    150,000km
2     70,000km
3     70,000km
4    150,000km
Name: odometer, dtype: object
```

In [219]:
```
autos["odometer"] = (autos["odometer"]
                        .str.replace("km", "")
                        .str.replace(",", "")
                        .astype(int)
                    )
autos.rename({"odometer": "odometer_km"}, axis=1, inplace=True)
autos["odometer_km"].head()
```

Out[219]:
```
0    150000
1    150000
2     70000
3     70000
4    150000
Name: odometer_km, dtype: int64
```

In [220]:
```
autos.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 20 columns):
 #   Column              Non-Null Count  Dtype 
---  ------              --------------  ----- 
 0   date_crawled        50000 non-null  object
 1   name                50000 non-null  object
 2   seller              50000 non-null  object
 3   offer_type          50000 non-null  object
 4   price_usd           50000 non-null  int64 
 5   abtest              50000 non-null  object
 6   vehicle_type        44905 non-null  object
 7   registration_year   50000 non-null  int64 
 8   gearbox             47320 non-null  object
 9   power_ps            50000 non-null  int64 
 10  model               47242 non-null  object
 11  odometer_km         50000 non-null  int64 
 12  registration_month  50000 non-null  int64 
 13  fuel_type           45518 non-null  object
 14  brand               50000 non-null  object
 15  unrepaired_damage   40171 non-null  object
 16  ad_created          50000 non-null  object
 17  num_photos          50000 non-null  int64 
 18  postal_code         50000 non-null  int64 
 19  last_seen           50000 non-null  object
dtypes: int64(7), object(13)
memory usage: 7.6+ MB
```

Now that the low hanging fruit has been taken care of, let's explore column by columns to determine what else need to be done. We will list all the issues we observe, and then step by step we can start cleaning the columns.

In [221]:
```
autos.describe(include="all")
```

Out[221]:

	date_crawled	name	seller	offer_type	price_usd	abtest	vehicle_type	registra
count	50000	50000	50000	50000	5.000000e+04	50000	44905	500
unique	48213	38754	2	2	NaN	2	8	
top	2016-03-08 10:40:35	Ford_Fiesta	privat	Angebot	NaN	test	limousine	
freq	3	78	49999	49999	NaN	25756	12859	
mean	NaN	NaN	NaN	NaN	9.840044e+03	NaN	NaN	20
std	NaN	NaN	NaN	NaN	4.811044e+05	NaN	NaN	1
min	NaN	NaN	NaN	NaN	0.000000e+00	NaN	NaN	10
25%	NaN	NaN	NaN	NaN	1.100000e+03	NaN	NaN	19
50%	NaN	NaN	NaN	NaN	2.950000e+03	NaN	NaN	20
75%	NaN	NaN	NaN	NaN	7.200000e+03	NaN	NaN	20
max	NaN	NaN	NaN	NaN	1.000000e+08	NaN	NaN	99

Exploring the seller and offer_type columns

In [222]:
```
autos["seller"].value_counts()
```

Out[222]:
```
privat        49999
gewerblich        1
Name: seller, dtype: int64
```

We can see that the seller column contains 49999 identical values. We can drop that column since it is not valuable for our analysis.

In [223]:
```
autos["offer_type"].value_counts()
```

Out[223]:
```
Angebot    49999
Gesuch         1
Name: offer_type, dtype: int64
```

Same goes for the offer_type column. Let's drop those columns using the df.drop() method.

In [224]:
```
autos = autos.drop(labels = ["seller", "offer_type"], axis=1)
```

We can also observe that the 'num_photos' column looks weird (all = 0), so we will explore it:

In [225]:
```
autos["num_photos"].value_counts()
```

Out[225]:
```
0    50000
Name: num_photos, dtype: int64
```

In [226]:
```
autos["num_photos"].head()
```

Out[226]:
```
0    0
1    0
2    0
3    0
4    0
Name: num_photos, dtype: int64
```

All the values in num_photos column are 0, so we can drop that column since it has no value for us.

In [227]:
```
autos = autos.drop(labels = "num_photos", axis=1)
```

In [228]:
```
autos.columns
```

Out[228]:
```
Index(['date_crawled', 'name', 'price_usd', 'abtest', 'vehicle_type',
       'registration_year', 'gearbox', 'power_ps', 'model', 'odometer_km',
       'registration_month', 'fuel_type', 'brand', 'unrepaired_damage',
       'ad_created', 'postal_code', 'last_seen'],
      dtype='object')
```

In [229]:
```
autos.shape
```

Out[229]:
```
(50000, 17)
```

From the above, we can see we have successfuly dropped 3 unneccessary columns - our new column count is 17

Exploring price_usd and odometer_km columns

Not expensive, and not going around much either

OK. So, what's interesting in 'price' and 'odometer'?

For sure, I want:

1. A cheap car. Of course.
2. (But I'm curious with the price of the most expensive car too)
3. A car that has least mileage. Somehow, I'm under the impression of least mileage = better condition. Yea, I don't really know much about cars to be honest haha...
4. (But I'm also curious with the car with the most mileage.)

And let's see if there's any nonsensical price or mileage in our data set.

In [230]:
```
print(autos["odometer_km"].unique().shape)
print(autos["odometer_km"].describe())
autos["odometer_km"].value_counts().sort_index(ascending=False)
```

```
(13,)
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
```

Out[230]:
```
150000    32424
125000     5170
100000     2169
90000      1757
80000      1436
70000      1230
60000      1164
50000      1027
40000       819
30000       789
20000       784
10000       264
5000        967
Name: odometer_km, dtype: int64
```

When we explore the odometer_km, we can see that the values are rounded and equally incremented, which suggests users had to choose from predefined values. We can observe that approx - 65% of vehicles have high mileage (150,000km), and approx - 80% have mileage of 100,000km or higher.

In [231]:
```
print(autos["price_usd"].unique().shape)
print(autos["price_usd"].describe())
print(autos["price_usd"].value_counts().sort_index(ascending=False).head(10))
autos["price_usd"].value_counts().sort_index().head(10)
```

```
(2357,)
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_usd, dtype: float64
99999999    1
27322222    1
12345678    3
11111111    2
10000000    1
3890000     1
1300000     1
1234566     1
999999      2
999990      1
Name: price_usd, dtype: int64
```

Out[231]:
```
0     1421
1      156
2        3
3        1
5        2
8        1
9        1
10       7
11       2
12       3
Name: price_usd, dtype: int64
```

We can see that there are some unusually high prices - there are 15 vehicles listed with prices above or close to a million, we can probably drop those rows and retain everything priced at 350,000 and less, since that is much more realistic.

When we look at the ascending price order, we can see that there are values which incrementaly increase from 1 USD. Since eBay is an auction site, it might be possible that these sellers have started the auctions with low values. Since that assumption is viable.

In [232]:
```
autos = autos[autos["price_usd"].between(1, 350100)]
autos["price_usd"].describe()
```

Out[232]:
```
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_usd, dtype: float64
```

We can see that all the outliers have been successfully removed and our prices now range from 1 to 350,000 USD

Exploring and cleaning up registration data

Does time matter?

Other than prices, does time play an interesting role here?

Let's explore it and see if we can find something interesting.

In [233]:
```
autos[['date_crawled', 'ad_created', 'last_seen']].head()
```

Out[233]:

	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

To select the first 10 characters in each column, we can use Series.str[:10]:

In [234]:
```
print(autos['date_crawled']
      .str[:10]
      .value_counts(normalize=True, dropna=False)
      .sort_index()
      )
```

```
2016-03-05    0.025327
2016-03-06    0.014043
2016-03-07    0.036014
2016-03-08    0.033296
2016-03-09    0.033090
2016-03-10    0.032184
2016-03-11    0.032575
2016-03-12    0.036920
2016-03-13    0.015670
2016-03-14    0.036549
2016-03-15    0.034284
2016-03-16    0.029610
2016-03-17    0.031628
2016-03-18    0.012911
2016-03-19    0.034778
2016-03-20    0.037887
2016-03-21    0.037373
2016-03-22    0.032987
2016-03-23    0.032225
2016-03-24    0.029342
2016-03-25    0.031607
2016-03-26    0.032204
2016-03-27    0.031092
2016-03-28    0.034860
2016-03-29    0.034099
2016-03-30    0.033687
2016-03-31    0.031834
2016-04-01    0.033687
2016-04-02    0.035478
2016-04-03    0.038608
2016-04-04    0.036487
2016-04-05    0.013096
2016-04-06    0.003171
2016-04-07    0.001400
Name: date_crawled, dtype: float64
```

I don't see much interesting thing here. It's like the there was a daily crawl but that's it, the percentage are quite inconsistent.

In [235]:
```
print(autos['ad_created']
     .str[:10]
     .value_counts(normalize=True, dropna=False)
     .sort_index()
     )
```

```
2015-06-11    0.000021
2015-08-10    0.000021
2015-09-09    0.000021
2015-11-10    0.000021
2015-12-05    0.000021
                ...
2016-04-03    0.038855
2016-04-04    0.036858
2016-04-05    0.011819
2016-04-06    0.003253
2016-04-07    0.001256
Name: ad_created, Length: 76, dtype: float64
```

Ooookay, there's like a huge percentage gap there between Feburary and March 2016 in a period of just 2 weeks. Interesting!

In [236]:
```
print(autos['last_seen']
     .str[:10]
     .value_counts(normalize=True, dropna=False)
     .sort_index()
     )
```

```
2016-03-05    0.001071
2016-03-06    0.004324
2016-03-07    0.005395
2016-03-08    0.007413
2016-03-09    0.009595
2016-03-10    0.010666
2016-03-11    0.012375
2016-03-12    0.023783
2016-03-13    0.008895
2016-03-14    0.012602
2016-03-15    0.015876
2016-03-16    0.016452
2016-03-17    0.028086
2016-03-18    0.007351
2016-03-19    0.015834
2016-03-20    0.020653
2016-03-21    0.020632
2016-03-22    0.021373
2016-03-23    0.018532
```

The percentage spikes on the last three days, but I'm not sure why.

Do these cars have time-traveling features?

In [237]:
```
autos["registration_year"].describe()
```

Out[237]:
```
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
```

Minimum registration year is 1000, which is not posssible. Maximum registration year is 9999. I, for sure, am going to buy cars with any time-traveling features LOL

In [238]:
```
autos["registration_year"].value_counts().sort_index(ascending=False).head(20)
```

Out[238]:
```
9999       3
9000       1
8888       1
6200       1
5911       1
5000       4
4800       1
4500       1
4100       1
2800       1
2019       2
2018     470
2017    1392
2016    1220
2015     392
2014     663
2013     803
2012    1310
2011    1623
2010    1589
Name: registration_year, dtype: int64
```

In [239]:
```
autos.describe(include='all')
```

Out[239]:

	date_crawled	name	price_usd	abtest	vehicle_type	registration_year	gearbo
count	48565	48565	48565.000000	48565	43979	48565.000000	4622
unique	46882	37470	NaN	2	8	NaN	
top	2016-03-12 16:06:22	Ford_Fiesta	NaN	test	limousine	NaN	manue
freq	3	76	NaN	25019	12598	NaN	3610
mean	NaN	NaN	5888.935591	NaN	NaN	2004.755421	Na
std	NaN	NaN	9059.854754	NaN	NaN	88.643887	Na
min	NaN	NaN	1.000000	NaN	NaN	1000.000000	Na
25%	NaN	NaN	1200.000000	NaN	NaN	1999.000000	Na
50%	NaN	NaN	3000.000000	NaN	NaN	2004.000000	Na
75%	NaN	NaN	7490.000000	NaN	NaN	2008.000000	Na
max	NaN	NaN	350000.000000	NaN	NaN	9999.000000	Na

In [240]:
```
autos["last_seen"].max()
```

Out[240]:
```
'2016-04-07 14:58:50'
```

In [241]:
```
autos["date_crawled"].min()
```

Out[241]:
```
'2016-03-05 14:06:30'
```

Let's pick a good cutoff point of a realistic and valid year for car registration.

Since the max value of last_seen in 2016, registration years 2017 and above are not valid. In addition, we need to combine year and month of registration to not exceed March 2016, since that is the date ads have been crawled. Any rows with registration months/years newer than February 2016 should be removed.

In [242]:
```
autos["registration_year"].value_counts().sort_index().head(20)
```

Out[242]:
```
1000    1
1001    1
1111    1
1800    2
1910    5
1927    1
1929    1
1931    1
1934    2
1937    4
1938    1
1939    1
1941    2
1943    1
1948    1
1950    3
1951    2
1952    1
1953    1
1954    2
Name: registration_year, dtype: int64
```

For the lower limit of our registration interval, we can see that there are a couple of rows showing registration year from the beginning of the 20th century. Maybe those are old-times, so before we remove these rows entirely, it is worth to explore further. How about... 1886 for the minimum value? It's near the birth year of Mercedes-Benz car according to [Wikipedia (https://en.wikipedia.org/wiki/Car)](https://en.wikipedia.org/wiki/Car)

In [243]:
```
autos[autos["registration_year"] < 1911]
```

Out[243]:

	date_crawled	name	price_usd	abtest	vehicle_type	re
3679	2016-04-04 00:36:17	Suche_Auto	1	test	NaN	
10556	2016-04-01 06:02:10	UNFAL_Auto	450	control	NaN	
22316	2016-03-29 16:56:41	VW_Kaefer.__Zwei_zum_Preis_von_einem.	1500	control	NaN	
22659	2016-03-14 08:51:18	Opel_Corsa_B	500	test	NaN	
24511	2016-03-17 19:45:11	Trabant__wartburg__Ostalgie	490	control	NaN	
28693	2016-03-22 17:48:41	Renault_Twingo	599	control	kleinwagen	
30781	2016-03-25 13:47:46	Opel_Calibra_V6_DTM_Bausatz_1:24	30	test	NaN	
32585	2016-04-02 16:56:39	UNFAL_Auto	450	control	NaN	
45157	2016-03-11 22:37:01	Motorhaube	15	control	NaN	
49283	2016-03-15 18:38:53	Citroen_HY	7750	control	NaN	

We can see that the results are mixed - there are some oldtimers, but also there are some cars which clearly exist in that time period, e.g. Opel Corsa (1950s) or Renault Twingo (1910s). To ensure better data quality, we can remove all rows with registration years prior to 1911.

In [244]:
```
autos[(autos["registration_year"] == 2016) & (autos["registration_month"] > 2)
```

Out[244]:

	date_crawled	name	price_usd	abtest	vehi
135	2016-03-12 11:00:10	Opel_Meriva_B_Panoramadach__Sitz__und_Lenkradh...	8500	control	
256	2016-04-03 20:50:38	Passat_1.9TDI_4Motion_Highline	4250	test	
295	2016-03-28 03:36:22	Privat_anbiter	1000	control	
307	2016-03-15 22:50:48	Giessen_ford	2800	test	
437	2016-03-25 16:39:01	Mazda__klima_leder__Alufelgen	550	control	
...	...	...	...	...	
49547	2016-03-30 16:49:46	Smart_Passion_mit_Panorama_Dach	3600	test	
49852	2016-04-01 04:02:25	TOP__Golf_3_1.8l	1450	control	
49876	2016-03-22 17:57:24	Audi_a5_3.0_tdi_s_line	14700	control	
49919	2016-03-10 09:49:43	Fiat_Punto	180	test	
49938	2016-03-28 18:45:06	Mercedes_Benz_A_160_Avantgarde	2300	control	

795 rows × 17 columns

Let's see what is the ratio of these outliers:

In [245]:
```
((autos["registration_year"] == 2016) & (autos["registration_month"] > 2)).sum
```

Out[245]:
```
0.016369813651806855
```

In [246]:
```
(autos["registration_year"] < 1911).sum() / autos.shape[0]
```

Out[246]:
```
0.00020590960568310512
```

In [247]:
```
autos = autos[autos["registration_year"].between(1911,2016)]
```

In [248]:
```
autos = autos[~((autos["registration_year"] == 2016) & (autos["registration_mo
```

In [249]:
```
autos["registration_year"].value_counts(normalize = True).head(15)
```

Out[249]:
```
2000    0.068787
2005    0.063992
1999    0.063142
2004    0.058913
2003    0.058826
2006    0.058194
2001    0.057453
2002    0.054184
1998    0.051503
2007    0.049628
2008    0.048277
2009    0.045444
1997    0.042523
2011    0.035374
2010    0.034633
Name: registration_year, dtype: float64
```

In [250]:
```
autos.shape
```

Out[250]:
```
(45881, 17)
```

We have reduced our dataset to 45881 rows of data. Registration year distribution looks good, with majority of the data falling into the 1997+ year range.

Exploring Price by Brand

First, we will take a look at all the brands in the dataset and select the top brands by percentage.

In [251]:
```
autos["brand"].unique().shape
```

Out[251]:
```
(40,)
```

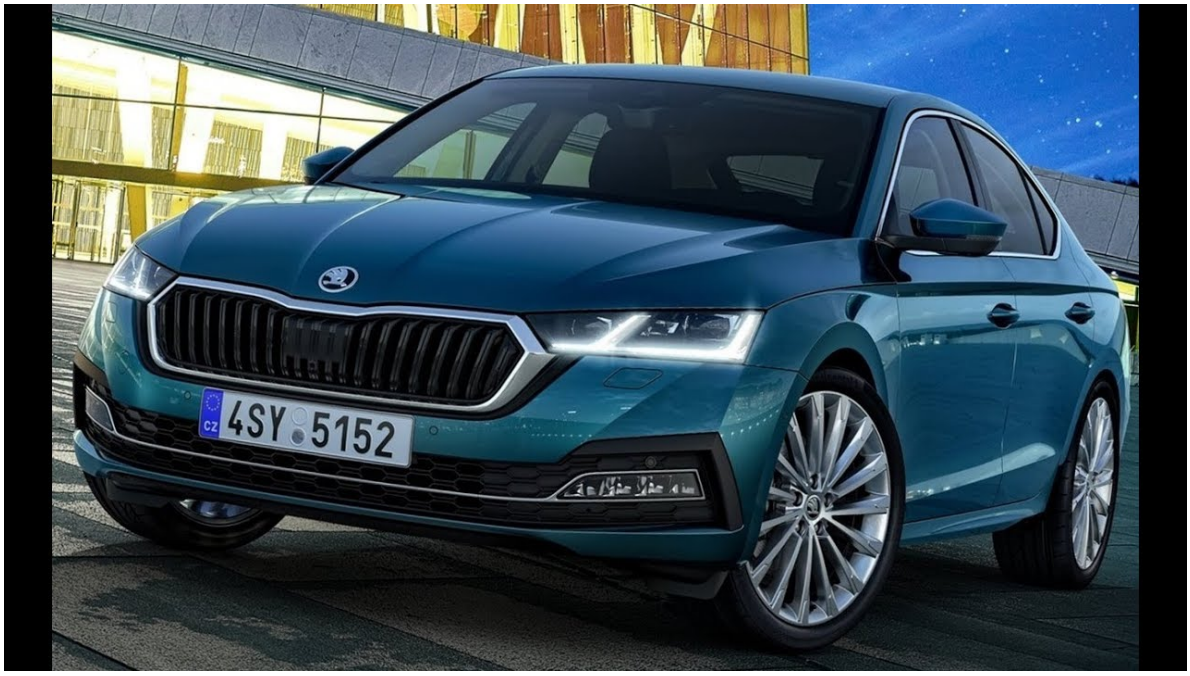
We can see there are 40 unique brands in the dataset. Let's see which those are:

In [252]:
```
autos["brand"].value_counts(normalize=True)
```

Out[252]:
```
volkswagen        0.210719
bmw               0.110743
opel              0.106951
mercedes_benz     0.096946
audi              0.086942
ford              0.069767
renault           0.047013
peugeot           0.029685
fiat              0.025544
seat              0.018134
skoda             0.016521
nissan            0.015213
mazda             0.015148
smart             0.014080
citroen           0.013971
toyota            0.012750
hyundai           0.010026
sonstige_autos    0.009808
volvo             0.009220
```

As expected, the majority of the brands that are offered are European (over 75%), seems like German brands dominated the top brands.

Also, this is the first time I heard about these car brands: 'skoda', 'sonstige_autos', 'dacia', 'saab', 'trabant', 'lancia', and 'lada'. So I decided to google them and... I'm totally missing out. These cars are super cool! Check out this pretty lava blue Skoda Superb!!



Well, we will take the top of the brands (accounts for more than 1%) for our price analysis:\

- volkswagen
- bmw
- opel

- mercedes_benz
- audi
- ford
- renault
- peugeot
- fiat
- seat
- skoda
- nissan
- mazda
- smart
- citroen
- toyota
- huyndai

Create an empty dictionary to hold the price data:

In [253]:
```
brand_mean_prices = {}
```

We will assign our normalized value count to a new variable, and then use the .index attribute to access the top market share brands:

In [254]:
```
brands_counts = autos["brand"].value_counts(normalize=True)
brands = brands_counts[brands_counts > .01].index #brands_counts[:15].sum()
brands
```

Out[254]:
```
Index(['volkswagen', 'bmw', 'opel', 'mercedes_benz', 'audi', 'ford', 'renaul
t',
       'peugeot', 'fiat', 'seat', 'skoda', 'nissan', 'mazda', 'smart',
       'citroen', 'toyota', 'hyundai'],
      dtype='object')
```

We will now loop over each brand and calcute the mean price. Then we will assign the brand as a key to the dictionary, and calculated mean price for each brand as a value to the key (as integer for better readability)

In [255]:
```
for b in brands:
    selected_rows = autos[autos["brand"] == b]
    mean_price = selected_rows["price_usd"].mean()
    brand_mean_prices[b] = int(mean_price)
```

In [256]:
```
brand_mean_prices
```

Out[256]:
```
{'volkswagen': 5453,
 'bmw': 8375,
 'opel': 2997,
 'mercedes_benz': 8682,
 'audi': 9357,
 'ford': 3797,
 'renault': 2493,
 'peugeot': 3111,
 'fiat': 2834,
 'seat': 4441,
 'skoda': 6375,
 'nissan': 4789,
 'mazda': 4164,
 'smart': 3603,
 'citroen': 3824,
 'toyota': 5200,
 'hyundai': 5437}
```

We can see that the #1 brand, Volkswagen, has a very attractive mean price - it is much cheaper than BMW, Mercedes or Audi, while more expensive on average than Opel, Renault or Peugeot. The attractive price and German origin most likely make it so popular.
On the other hand, BMW, Mercedes and Audi are most expensive, but still rank amongst the top 5.
Opel, Ford, Peugeot and Renault are less expensive than all the above mentioned brands, so it is realistic they will have a large portion of the market share.
Asian brands such as Renault, Mazda, Toyota and Huyndai with mid-range prices are at the bottom of the list.

Calculating the mean mileage

Using the same principle as above, we will calculate the mean mileage for each of our selected brands:

In [257]:
```
brand_mean_mileage = {}
```

In [258]:
```
for b in brands:
    selected_rows = autos[autos["brand"] == b]
    mean_mileage = selected_rows["odometer_km"].mean()
    brand_mean_mileage[b] = int(mean_mileage)

brand_mean_mileage
```

Out[258]:
```
{'volkswagen': 128526,
 'bmw': 132498,
 'opel': 129242,
 'mercedes_benz': 130683,
 'audi': 129251,
 'ford': 124039,
 'renault': 127950,
 'peugeot': 127063,
 'fiat': 116970,
 'seat': 120907,
 'skoda': 110916,
 'nissan': 118524,
 'mazda': 124079,
 'smart': 98769,
 'citroen': 119329,
 'toyota': 115777,
 'hyundai': 105847}
```

Create a new dataframe for comparison of price and mileage

We will first use pandas series constructor to convert both brand_mean_prices and brand_mean_mileage dictionaries to series objects:

In [259]:
```
mean_prices_series = pd.Series(brand_mean_prices).sort_values(ascending=False)
mean_mileage_series = pd.Series(brand_mean_mileage).sort_values(ascending=Fals
```

In [260]:
```
mean_prices_series #dict thành list
```

Out[260]:
```
audi             9357
mercedes_benz    8682
bmw              8375
skoda            6375
volkswagen       5453
hyundai          5437
toyota           5200
nissan           4789
seat             4441
mazda            4164
citroen          3824
ford             3797
smart            3603
peugeot          3111
opel             2997
fiat             2834
renault          2493
dtype: int64
```

In [261]:
```
mean_price_mileage_df = pd.DataFrame(mean_prices_series, columns = ["mean_pric
mean_price_mileage_df
```

Out[261]:

	mean_prices_series
audi	9357
mercedes_benz	8682
bmw	8375
skoda	6375
volkswagen	5453
hyundai	5437
toyota	5200
nissan	4789
seat	4441
mazda	4164
citroen	3824
ford	3797
smart	3603
peugeot	3111
opel	2997
fiat	2834
renault	2493

In [262]:
```
#mean_mileage_df = pd.DataFrame(mean_mileage_series, columns = ["mean_mileage_
#mean_mileage_df
mean_price_mileage_df['mean_mileage_series'] = mean_mileage_series
mean_price_mileage_df
```

Out[262]:

	mean_prices_series	mean_mileage_series
audi	9357	129251
mercedes_benz	8682	130683
bmw	8375	132498
skoda	6375	110916
volkswagen	5453	128526
hyundai	5437	105847
toyota	5200	115777
nissan	4789	118524
seat	4441	120907
mazda	4164	124079
citroen	3824	119329
ford	3797	124039

In [263]:
```
brand_info = mean_price_mileage_df
brand_info
```

Out[263]:

	mean_prices_series	mean_mileage_series
audi	9357	129251
mercedes_benz	8682	130683
bmw	8375	132498
skoda	6375	110916
volkswagen	5453	128526
hyundai	5437	105847
toyota	5200	115777
nissan	4789	118524
seat	4441	120907
mazda	4164	124079
citroen	3824	119329
ford	3797	124039
smart	3603	98769
peugeot	3111	127063
opel	2997	129242
fiat	2834	116970
renault	2493	127950

We have merged both series into one dataframe called brand_info with values sorted in a descending order. Now we can easily compare prices and mileage.

We cannot observe a large gap in mileage, but rather a trend that more expensive brands tend to have slightly higher mileage than less expensive brands. Exception is Skoda, which has quite low mileage for the mean price.

Since Mercedes, BMW and Audi mostly make limousines, it may be the reason why these brands have higher mean mileage - limousines are mostly used for long range travel, while cheaper vehicles will mostly be used within the city limits, for commuting.

Translating German to English

Since many people are monolingual who happen to be occasionally curious with raw data, let's translate the German words in this data into English, just to be safe.

In [264]:
```
autos.head()
```

Out[264]:

	date_crawled	name	price_usd	abtest	vehicle_typ
0	2016-03-26 17:47:46	Peugeot_807_160_NAVTECH_ON_BOARD	5000	control	bu
1	2016-04-04 13:38:56	BMW_740i_4_4_Liter_HAMANN_UMBAU_Mega_Optik	8500	control	limousin
2	2016-03-26 18:57:24	Volkswagen_Golf_1.6_United	8990	test	limousin
3	2016-03-12 16:58:10	Smart_smart_fortwo_coupe_softouch/F1/Klima/Pan...	4350	control	kleinwage
4	2016-04-01 14:38:50	Ford_Focus_1_6_Benzin_TÜV_neu_ist_sehr_gepfleg...	1350	test	kom

In [265]:
```
autos["vehicle_type"].unique()
```

Out[265]:
```
array(['bus', 'limousine', 'kleinwagen', 'kombi', nan, 'coupe', 'suv',
       'cabrio', 'andere'], dtype=object)
```

In [266]:
```
translation = ({'bus': 'buss',
                'limousine': 'limousine',
                'kleinwagen': 'small_car',
                'kombi': 'van',
                'coupe': 'coupe',
                'suv': 'suv',
                'cabrio': 'convertible',
                'andere':'other'})
```

In [267]:
```
autos["vehicle_type"] = autos["vehicle_type"].map(translation)
autos["vehicle_type"].value_counts()
```

Out[267]:
```
limousine      12591
small_car      10573
van             8925
buss            4031
convertible     3014
coupe           2460
suv             1962
other            390
Name: vehicle_type, dtype: int64
```

In [268]:
```
autos["fuel_type"].unique()
```

Out[268]:
```
array(['lpg', 'benzin', 'diesel', nan, 'cng', 'hybrid', 'elektro',
       'andere'], dtype=object)
```

In [269]:
```
translation = ({'lpg': 'lpg',
                'benzin': 'petrol',
                'diesel': 'diesel',
                'cng': 'cng',
                'hybrid': 'hybrid',
                'elektro': 'electric',
                'andere': 'other'})
```

In [270]:
```
autos["fuel_type"] = autos["fuel_type"].map(translation)
autos["fuel_type"].value_counts()
```

Out[270]:
```
petrol      28172
diesel      13932
lpg           643
cng            70
hybrid         36
electric       17
other          14
Name: fuel_type, dtype: int64
```

In [271]:
```
autos["gearbox"].unique()
```

Out[271]:
```
array(['manuell', 'automatik', nan], dtype=object)
```

In [272]:
```
autos['gearbox'] = autos['gearbox'].str.replace('manuell', 'manual').str.repla
autos["gearbox"].value_counts()
```

Out[272]:
```
manual       34081
automatic     9760
Name: gearbox, dtype: int64
```

In [273]:
```
autos["unrepaired_damage"].unique()
```

Out[273]:
```
array(['nein', nan, 'ja'], dtype=object)
```

In [274]:
```
autos['unrepaired_damage'] = autos['unrepaired_damage'].str.replace('ja', 'yes
autos["unrepaired_damage"].value_counts()
```

Out[274]:
```
no     33446
yes     4443
Name: unrepaired_damage, dtype: int64
```

Now I know some words in German!

#TeamCommon or #TeamUnique?

I can also see a combination of brand name and model type separated with underscores in the `name` column.

Let's see what's the common combos for these cars.

In [275]:
```
brand_model_combo = autos.groupby(["brand","model"]).count()   #df.groupby(['a
brand_model_combo
```

Out[275]:

brand	model	date_crawled	name	price_usd	abtest	vehicle_type	registration_year
	145	4	4	4	4	2	4
	147	78	78	78	78	73	78
alfa_romeo	**156**	87	87	87	87	84	87
	159	32	32	32	32	31	32
	andere	59	59	59	59	56	59
...	**...**	...	...	...	...	...	...
	v40	86	86	86	86	84	86
	v50	28	28	28	28	28	28
volvo	**v60**	3	3	3	3	3	3
	v70	90	90	90	90	87	90

In [276]:
```
common_combo = brand_model_combo["name"].sort_values(ascending=False)  #cột nà
common_combo
```

Out[276]:
```
brand       model
volkswagen  golf          3622
bmw         3er           2586
volkswagen  polo          1566
opel        corsa         1556
volkswagen  passat        1333
                          ...
lancia      kappa            2
rover       rangerover       1
            discovery        1
audi        200              1
ford        b_max            1
Name: name, Length: 289, dtype: int64
```

If you are in the mood to be in the #TeamUnique, you better avoid the `golf` Volkswagen or `3er` BMW and get to buy a `200` Audi or `b_max` Ford

---

More numeric data, please

Just for the fun of it, let's also convert the dates to be uniform numeric data, so `"2016-03-21"` becomes the integer `20160321` .

In [277]:
```
autos["date_crawled"] = autos["date_crawled"].str[:10].str.replace("-","").ast
```

In [278]:
```
autos["ad_created"] = autos["ad_created"].str[:10].str.replace("-","").astype(
```

In [279]:
```
autos["last_seen"] = autos["last_seen"].str[:10].str.replace("-","").astype(in
```

In [280]:
```
autos.head()
```

Out[280]:

	date_crawled	name	price_usd	abtest	vehicle_typ
0	20160326	Peugeot_807_160_NAVTECH_ON_BOARD	5000	control	bus
1	20160404	BMW_740i_4_4_Liter_HAMANN_UMBAU_Mega_Optik	8500	control	limousin
2	20160326	Volkswagen_Golf_1.6_United	8990	test	limousin
3	20160312	Smart_smart_fortwo_coupe_softouch/F1/Klima/Pan...	4350	control	small_c
4	20160401	Ford_Focus_1_6_Benzin_TÜV_neu_ist_sehr_gepfleg...	1350	test	va

Mileage group

Since we have concluded mileages are rounded up, they can easily illustrate for further analysis. We assume vehices with lower mileage will have lower mean prices, so let's see if that is correct.

We will start by taking a look once again at odometer value counts:

In [281]:
```
autos["odometer_km"].value_counts().sort_index() #odometer_group = autos.group
```

Out[281]:
```
5000         742
10000        239
20000        738
30000        756
40000        791
50000        988
60000       1117
70000       1175
80000       1359
90000       1655
100000      2032
125000      4803
150000     29486
Name: odometer_km, dtype: int64
```

We can see that there are 13 mileage categories, we will narrow them down into 3 group - low, medium and high:

In [282]:
```
odometer_price = autos.groupby("odometer_km")
odometer_price["price_usd"].mean().sort_values(ascending=False)
#odometer_price["price_usd"].mean().sort_values(ascending=False)
```

Out[282]:
```
odometer_km
10000     20574.305439
20000     18483.537940
30000     16644.611111
40000     15540.653603
50000     13844.735830
60000     12442.254252
70000     10987.248511
80000      9752.215600
90000      8515.038066
100000     8205.128937
5000       7267.716981
125000     6231.157402
150000     3806.961405
Name: price_usd, dtype: float64
```

From the above, we can see that our assumption was correct - mean prices drop significantly with the mileage.

Exploring damage effect on the price

Damaged cars are cheaper than non-damaged cars. That's the norm. But, by how much?

In [283]:
```
damage_group = autos.groupby("unrepaired_damage").count()
damage_group
```

Out[283]:

unrepaired_damage	date_crawled	name	price_usd	abtest	vehicle_type	registration_year	gear
no	33446	33446	33446	33446	33046	33446	32
yes	4443	4443	4443	4443	4244	4443	4

In [284]:
```
no_damage = autos[autos["unrepaired_damage"] == "no"] #autos.groupby("unrepair
damage = autos[autos["unrepaired_damage"] == "yes"]
```

In [285]:
```
print(no_damage["price_usd"].mean())
print(damage["price_usd"].mean())
damage_difference = no_damage["price_usd"].mean() - damage["price_usd"].mean()

print("On average, cars with damage are USD {:.2f}".format(damage_difference)
     + " cheaper than their non-damaged counterparts.")
```

```
7165.327034622975
2266.5109160477155
On average, cars with damage are USD 4898.82 cheaper than their non-damaged c
ounterparts.
```

We can see that, an expected, cars which have repaired damages are priced much higher ($4898.82) than cars which have damages left unrepaired

But, let's take a look which brands are more or less affected by unrepaired damege:

In [286]:
```
brands_unrepaired_vc = damage["brand"].value_counts(normalize=True).sort_value
brands_unrepaired = brands_unrepaired_vc.index
brands_unrepaired
```

Out[286]:
```
Index(['volkswagen', 'opel', 'ford', 'bmw', 'mercedes_benz', 'audi', 'renaul
t',
       'peugeot', 'fiat', 'nissan'],
      dtype='object')
```

In [287]:
```
unrepaired_brand_price = {}

for b in brands_unrepaired:
    selected_rows = damage[damage["brand"] == b]
    mean_price = selected_rows["price_usd"].mean()
    unrepaired_brand_price[b] = int(mean_price)
```

In [288]:
```
unrepaired_brand_price
```

Out[288]:
```
{'volkswagen': 2196,
 'opel': 1369,
 'ford': 1391,
 'bmw': 3554,
 'mercedes_benz': 4000,
 'audi': 3350,
 'renault': 1167,
 'peugeot': 1366,
 'fiat': 1166,
 'nissan': 1962}
```

In [289]:
```
ubp_series = pd.Series(unrepaired_brand_price).sort_values(ascending=False)
```

In [290]:
```
brands_repaired_vc = no_damage["brand"].value_counts(normalize=True).sort_valu
brands_repaired = brands_repaired_vc.index
brands_repaired
```

Out[290]:
```
Index(['volkswagen', 'bmw', 'mercedes_benz', 'opel', 'audi', 'ford', 'renaul
t',
       'peugeot', 'fiat', 'seat'],
      dtype='object')
```

In [291]:
```
repaired_brand_price = {}

for b in brands_repaired:
    selected_rows = no_damage[no_damage["brand"] == b]
    mean_price = selected_rows["price_usd"].mean()
    repaired_brand_price[b] = int(mean_price)
```

In [292]:
```
repaired_brand_price
```

Out[292]:
```
{'volkswagen': 6505,
 'bmw': 9467,
 'mercedes_benz': 9834,
 'opel': 3673,
 'audi': 10902,
 'ford': 4695,
 'renault': 3110,
 'peugeot': 3691,
 'fiat': 3452,
 'seat': 5220}
```

In [293]:
```
rbp_series = pd.Series(repaired_brand_price).sort_values(ascending=False)
```

In [294]:
```
damage_price_info = pd.DataFrame(ubp_series, columns = ["unrepaired_price"])
```

In [295]:
```
damage_price_info["repaired_price"] = rbp_series
```

In [301]:
```
damage_price_info["diff"] = (damage_price_info["unrepaired_price"] - damage_pr
damage_price_info["diff_%"] = (((damage_price_info["unrepaired_price"] - damag
```

In [302]:
```
damage_price_info.sort_values(by =["diff_%"])
```

Out[302]:

	unrepaired_price	repaired_price	diff	diff_%
ford	1391	4695.0	-3304.0	-70.0
audi	3350	10902.0	-7552.0	-69.0
volkswagen	2196	6505.0	-4309.0	-66.0
fiat	1166	3452.0	-2286.0	-66.0
opel	1369	3673.0	-2304.0	-63.0
peugeot	1366	3691.0	-2325.0	-63.0
bmw	3554	9467.0	-5913.0	-62.0
renault	1167	3110.0	-1943.0	-62.0
mercedes_benz	4000	9834.0	-5834.0	-59.0
nissan	1962	NaN	NaN	NaN

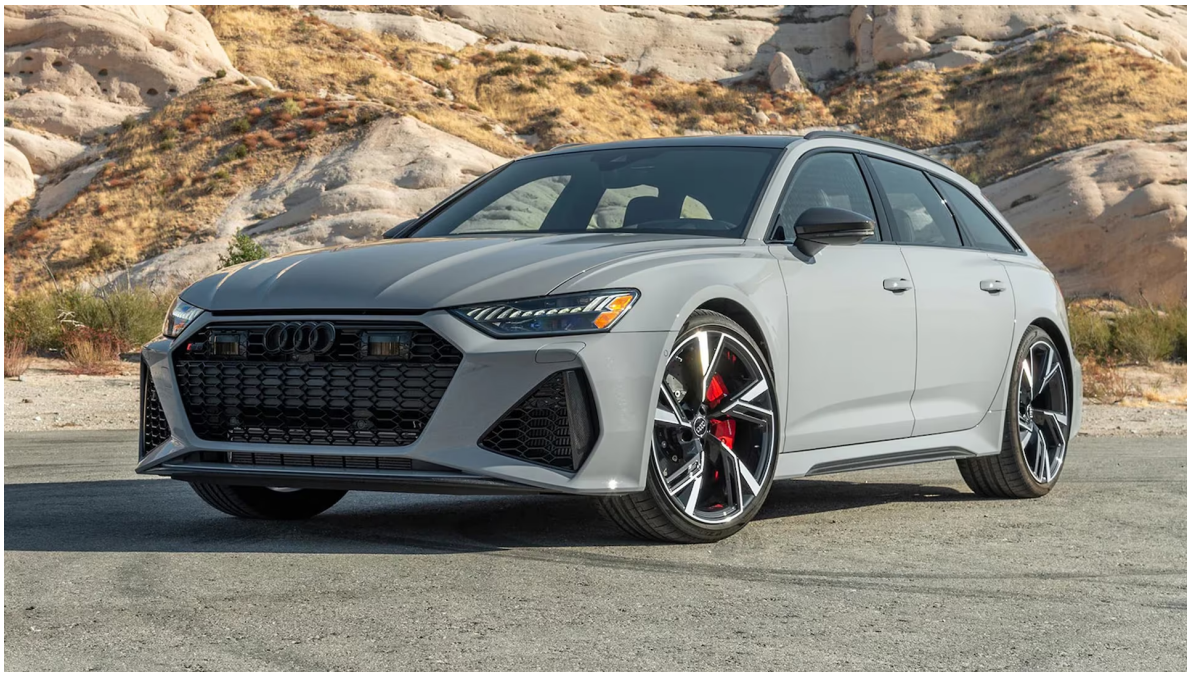
In [303]:
```
damage_price_info["diff_%"].mean().round(2)
```

Out[303]:
```
-64.44
```

On average, cars with damage are 59%-70% cheaper than their non-damaged counterparts in most brands.

Conclusion

Let me tell you something interesting from the analysis that we just did. Like, totally interesting. Some common sense stuff like Audi is too expensive for a car that is just into #TeamUnique; BMW, Mercedes-Benz and VW are among the top European car brands, damaged cars are cheaper than non-damaged cars, and how higher mileage could make car prices cheaper. That stuff can be easily deduced even if we don't use this data.



2023 Audi RS6 Model