ETL_cars

September 14, 2024

1 Vehicle Sales Data

A Project by Andrea Frittella | LinkedIn Profile

This project starts from a dataset containing information about vehicle sales in the United States. The dataset was obtained from Kaggle and contains information about 558837 used cars listed for sale.

It encompasses details such as the year (model year), maker, model, trim (vehicle setup/optionals), body type, transmission type, VIN, state of registration, condition rating, odometer reading, exterior and interior colors, seller information, MMR (Manheim Market Report) values, selling prices, sale dates.

We will start by loading the dataset and performing some basic data cleaning and exploration. After that, we will analyze the data to answer some questions and extract insights.

1.1 Table of Contents (with links to sections, usable in PDF format conversion)

1.2 Data Loading and Cleaning

[227]:
```python
#libs for data analysis
import pandas as pd
import numpy as np

#libs for data visualization
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
from scipy import stats
import seaborn as sns
import polars as pl
```

```python
#libs for file handling
import os

#libs for time handling
import time
from datetime import datetime, timedelta
from dateutil.relativedelta import relativedelta

#warning ignoring
import warnings
warnings.filterwarnings("ignore")
```

Let's start by loading the data into a dataframe and taking a look at the first few rows.

[228]:
```python
df_source = pd.read_csv(os.getcwd() + '/source_data/cars.csv')
df_source.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 558837 entries, 0 to 558836
Data columns (total 16 columns):
 #   Column        Non-Null Count   Dtype
---  ------        --------------   -----
 0   year          558837 non-null  int64
 1   make          548536 non-null  object
 2   model         548438 non-null  object
 3   trim          548186 non-null  object
 4   body          545642 non-null  object
 5   transmission  493485 non-null  object
 6   vin           558833 non-null  object
 7   state         558837 non-null  object
 8   condition     547017 non-null  float64
 9   odometer      558743 non-null  float64
 10  color         558088 non-null  object
 11  interior      558088 non-null  object
 12  seller        558837 non-null  object
 13  mmr           558799 non-null  float64
 14  sellingprice  558825 non-null  float64
 15  saledate      558825 non-null  object
dtypes: float64(4), int64(1), object(11)
memory usage: 68.2+ MB
```

We need to rework a little the dataset and start to work on a dataframe copy.

[229]:
```python
#create a new dataframe "df_edited" and rename some columns to give more context
df_edited = df_source.rename(columns={'year':'my','body':'category','state':
↪'state_sale','make':'maker','color':'ext_color','interior':'int_color',␣
↪'sellingprice':'selling_price','saledate':'sale_date'}, inplace=False)
```

```
df_edited.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 558837 entries, 0 to 558836
Data columns (total 16 columns):
 #   Column         Non-Null Count   Dtype
---  ------         --------------   -----
 0   my             558837 non-null  int64
 1   maker          548536 non-null  object
 2   model          548438 non-null  object
 3   trim           548186 non-null  object
 4   category       545642 non-null  object
 5   transmission   493485 non-null  object
 6   vin            558833 non-null  object
 7   state_sale     558837 non-null  object
 8   condition      547017 non-null  float64
 9   odometer       558743 non-null  float64
 10  ext_color      558088 non-null  object
 11  int_color      558088 non-null  object
 12  seller         558837 non-null  object
 13  mmr            558799 non-null  float64
 14  selling_price  558825 non-null  float64
 15  sale_date      558825 non-null  object
dtypes: float64(4), int64(1), object(11)
memory usage: 68.2+ MB
```

Having a look at the info, there is a lot of null values in the columns 'transmission' and some others. Let's evaluate the quota of null values in the dataset and decide how to handle them.

We will also check for duplicates and drop them if necessary.

[230]:
```
#evaluate the quota of null values in the dataset per each column. We will use␣
↪as the max not-null value the number as reference.
dic_quota_null = {}
for column in df_edited.columns:
    dic_quota_null[column] = df_edited[column].isnull().sum() / df_edited.
↪notnull().sum().max() #the quotient of this ratio is the max not-null value␣
↪considering all the columns

for i in dic_quota_null:
    print(i, '-', round(dic_quota_null[i]*100,2), '%')
```

```
my - 0.0 %
maker - 1.84 %
model - 1.86 %
trim - 1.91 %
category - 2.36 %
transmission - 11.69 %
```

```
vin - 0.0 %
state_sale - 0.0 %
condition - 2.12 %
odometer - 0.02 %
ext_color - 0.13 %
int_color - 0.13 %
seller - 0.0 %
mmr - 0.01 %
selling_price - 0.0 %
sale_date - 0.0 %
```

As shown in the previous output, the highest quota is from "transmission" column, with 11.69% of null values. The second highest is the "category" column, with 2.36% of nulla values.

Taking this in consideration, we will drop all the rows with null, from all the columns, as the quota is not too high and we will not lose too much data.

[231]:
```
#drop the rows with null values in the whole dataset, creating a new dataframe
↪"df_clean_wip"
df_clean = df_edited.dropna()
df_clean.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 472325 entries, 0 to 558836
Data columns (total 16 columns):
 #   Column         Non-Null Count   Dtype
---  ------         --------------   -----
 0   my             472325 non-null  int64
 1   maker          472325 non-null  object
 2   model          472325 non-null  object
 3   trim           472325 non-null  object
 4   category       472325 non-null  object
 5   transmission   472325 non-null  object
 6   vin            472325 non-null  object
 7   state_sale     472325 non-null  object
 8   condition      472325 non-null  float64
 9   odometer       472325 non-null  float64
 10  ext_color      472325 non-null  object
 11  int_color      472325 non-null  object
 12  seller         472325 non-null  object
 13  mmr            472325 non-null  float64
 14  selling_price  472325 non-null  float64
 15  sale_date      472325 non-null  object
dtypes: float64(4), int64(1), object(11)
memory usage: 61.3+ MB
```

The combination of all the line dropping from all the columns left us with 472,325 rows, which is still a good amount of data to work with: 84.52% of the original dataset (remaining rows / initial rows = 558837).

Now let's focus on the last column "sale date", converting its type as datetime, and removing the time part: it's not relevant for our analysis.

[232]:
```
#let's convert the column "sale_date" to datetime format, with utc parameter as␣
↪true since the data is in GMT/UTC
df_clean.loc[:, 'sale_date'] = pd.to_datetime(df_clean['sale_date'], utc=True,␣
↪errors='coerce')

#let's create a new column "sale_date_no_time" with only the date part of the␣
↪column "sale_date"
df_clean.loc[:, 'sale_date_no_time'] = df_clean['sale_date'].apply(lambda x: x.
↪date() if pd.notnull(x) else None)

#let's convert the column "sale_date_no_time" to datetime format
df_clean['sale_date_no_time'] = pd.to_datetime(df_clean['sale_date_no_time'],␣
↪format='%Y-%m-%d')

#let's drop the column "sale_date" since we have the column "sale_date_no_time"
df_clean.drop(columns=['sale_date'], inplace=True)

df_clean.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 472325 entries, 0 to 558836
Data columns (total 16 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   my                 472325 non-null  int64
 1   maker              472325 non-null  object
 2   model              472325 non-null  object
 3   trim               472325 non-null  object
 4   category           472325 non-null  object
 5   transmission       472325 non-null  object
 6   vin                472325 non-null  object
 7   state_sale         472325 non-null  object
 8   condition          472325 non-null  float64
 9   odometer           472325 non-null  float64
 10  ext_color          472325 non-null  object
 11  int_color          472325 non-null  object
 12  seller             472325 non-null  object
 13  mmr                472325 non-null  float64
 14  selling_price      472325 non-null  float64
 15  sale_date_no_time  472325 non-null  datetime64[ns]
dtypes: datetime64[ns](1), float64(4), int64(1), object(10)
memory usage: 61.3+ MB
```

It might be useful to create a new column "age_months" for the vehicles, expressed in months, which will be the difference between the date of the sale and the date of the vehicle. To be able to do this, we need to convert the “year” column to datetime type.

[233]:
```
#convert the "year" column to datetime
df_clean['my'] = pd.to_datetime(df_clean['my'], format='%Y')
df_clean.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 472325 entries, 0 to 558836
Data columns (total 16 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   my                 472325 non-null  datetime64[ns]
 1   maker              472325 non-null  object
 2   model              472325 non-null  object
 3   trim               472325 non-null  object
 4   category           472325 non-null  object
 5   transmission       472325 non-null  object
 6   vin                472325 non-null  object
 7   state_sale         472325 non-null  object
 8   condition          472325 non-null  float64
 9   odometer           472325 non-null  float64
 10  ext_color          472325 non-null  object
 11  int_color          472325 non-null  object
 12  seller             472325 non-null  object
 13  mmr                472325 non-null  float64
 14  selling_price      472325 non-null  float64
 15  sale_date_no_time  472325 non-null  datetime64[ns]
dtypes: datetime64[ns](2), float64(4), object(10)
memory usage: 61.3+ MB
```

[234]:
```
#let's create a new column "age_month" with the age of the car in months␣
↪(float), using the difference between the sale date and the year of the car
avg_day_per_month = 365.25/12
df_clean['age_months'] = round((df_clean['sale_date_no_time'] - df_clean['my']).
↪dt.days / avg_day_per_month,2)
df_clean[['my','sale_date_no_time','age_months']].head()
```

[234]:

	my	sale_date_no_time	age_months
0	2015-01-01	2014-12-16	-0.53
1	2015-01-01	2014-12-16	-0.53
2	2014-01-01	2015-01-14	12.42
3	2015-01-01	2015-01-28	0.89
4	2014-01-01	2014-12-18	11.53

We see that the “age” column has been created correctly, but it has some negative values. This could seem wrong, but it’s not, actually: some vehicles are produced some weeks before the official model year starting date, so it’s normal to have some negative values. We will just assign the value 0 to these rows.

[235]:
```python
#assign value 0 to the column "age_months" for the cars with negative values
df_clean['age_months'] = df_clean['age_months'].apply(lambda x: 0 if x < 0 else␣
↪x)
df_clean[['my','sale_date_no_time','age_months']].head()
```

[235]:
```
          my sale_date_no_time  age_months
0 2015-01-01        2014-12-16        0.00
1 2015-01-01        2014-12-16        0.00
2 2014-01-01        2015-01-14       12.42
3 2015-01-01        2015-01-28        0.89
4 2014-01-01        2014-12-18       11.53
```

1.3 Data Exploration and more cleaning

In this section, we will explore the data to get a better understanding of the dataset, checking the uniquity of the values in some columns, to exclude the ones that are not useful for our analysis, or that has no sense to not be or not to be unique.

1. VIN values

[236]:
```python
#let's check the VIN column to see if there are any duplicates
result = df_clean['vin'].unique().size == df_clean['vin'].size

print('There are no duplicates in the VIN column:', result)
```

```
There are no duplicates in the VIN column: False
```

Comparing the length of the list of unique values with the length of 'vin' column, the result is "False". It means that there are duplicates in the VIN column.

This does not necessarly mean that the duplicates presence is wrong: some vehicles could have been sold more than once. But we need to verify that all the values from sale date and odomter tied to the duplicated vins value are unique: even if the same vechile could have been sold more than once in the same day, we will exclude this sceanrio, admitting only one sale per day, per vehicle. Same for the odometer: we will assume that the same vehicle could have been sold more than once, in a different day, but the odometer value shall be different.

On the other hand, there are some other columns values that shall to be the same, for the same VIN: - model year, - maker, - model, - trim, - category, - transmission, - external and intarnal color.

Those columns describe the vehicle itself, and they should not change between two sales of the same vehicle, represented by the same VIN.

[237]:
```python
#let's create a list of the values of duplicated VIN values
duplicated_vins = df_clean[df_clean['vin'].duplicated()]['vin'].unique().
↪tolist()

print('We have', len(duplicated_vins), 'duplicated VINs in the dataset.')
print('The first 5 duplicated VINs are:', duplicated_vins[:5])
```

```
We have 6421 duplicated VINs in the dataset.
The first 5 duplicated VINs are: ['jnrar05y4xw049475', '5npec4ab7dh504364',
'3c4fy58834t214497', '1ftpw14v89ka34567', '3fahp08z09r134348']
```

2. Sale Date and other investigations

[238]:
```python
#let's create a dictionary with the duplicated VIN values as keys and the␣
↪amount of each duplicated value of the sale_date_no_time as value
dic_duplicated_vins = {} #emtpy dictionary

for i in duplicated_vins:
    dic_duplicated_vins[i] = df_clean[df_clean['vin'] ==␣
↪i]['sale_date_no_time'].duplicated().sum()

#craete a list of the VIN keys with a value greater than 0
vins_of_interest = [key for key, value in dic_duplicated_vins.items() if value␣
↪> 0]

print('We have', len(vins_of_interest), 'duplicated VINs with more than one␣
↪sale date in the dataset.')
print('The first 3 duplicated VINs with more than one sale date are:',␣
↪vins_of_interest[:3])
```

```
We have 69 duplicated VINs with more than one sale date in the dataset.
The first 3 duplicated VINs with more than one sale date are:
['1ftpw14588fa92105', '1zvft80n255109966', '2c3la63h26h278454']
```

[239]:
```python
#let's check if the vin of interest have all at least 1 sale date with the same␣
↪value
df_clean[df_clean['vin'].isin(vins_of_interest)][['vin','sale_date_no_time']].
↪sort_values('sale_date_no_time').head(10)
```

[239]:

	vin	sale_date_no_time
3744	2c3la63h26h278454	2014-12-31
45472	1ftpw14588fa92105	2014-12-31
47447	1zvft80n255109966	2014-12-31
57321	1ftpw14588fa92105	2014-12-31
72120	2c3la63h26h278454	2014-12-31
67284	1zvft80n255109966	2014-12-31
101939	2g2fs32k1y2122983	2015-01-07
95105	2g2fs32k1y2122983	2015-01-07
90230	1ftfw1cv5afb30053	2015-01-07
130513	wdbuf70j23a235692	2015-01-13

The presence of items from "vins_of_interest" list demonstrate that there are some rows with a duplicated VINs that have also the same sale date duplicates. This is not possible, as we assumed that the same vehicle could have been sold more than once, but not in the same day. We will drop these rows.

[240]:
```
#let's save only the first rows that are in the list "vins_of_interest" and
↪that have duplicated value in the column "sale_date_no_time"
df_clean = df_clean.groupby('vin').apply(lambda x: x.
↪drop_duplicates(subset=['sale_date_no_time'], keep='first')).
↪reset_index(drop=True)
```

At this point we need to investigate on the other values of the duplicated vins from the vins of interest.

[241]:
```
df_clean[df_clean['vin'].isin(vins_of_interest)][['vin','sale_date_no_time']].
↪sort_values('sale_date_no_time').head(10)
```

[241]:
```
                      vin sale_date_no_time
182672  1zvft80n255109966        2014-12-31
193361  2c3la63h26h278454        2014-12-31
79110   1ftpw14588fa92105        2014-12-31
223843  2g2fs32k1y2122983        2015-01-07
69397   1ftfw1cv5afb30053        2015-01-07
173983  1n6ad07w57c401276        2015-01-13
452320  wdbuf70j23a235692        2015-01-13
367585  jn1cv6ap0dm715990        2015-01-14
84038   1ftzx0727ykb48432        2015-01-14
111602  1gccs148358279742        2015-01-19
```

Here is important to note that we have performed this crucial and complex cleaning: before the operation, the “sale_date_no_time” value” 2014-12-31” was shared among 6 entries, with 3 unique vin values (A, B, C):

vin	sale_date_no_time
A: 2c3la63h26h278454	2014-12-31
B: 1ftpw14588fa92105	2014-12-31
C: 1zvft80n255109966	2014-12-31
A: 1ftpw14588fa92105	2014-12-31
B: 2c3la63h26h278454	2014-12-31
C: 1zvft80n255109966	2014-12-31

After the cleaning, we have not mistakenly removed (absolute) duplicates values of “sale_date_no_time” column: we correclty admitted (absolute) “sale_date_no_time” duplicates, but only for different “vin” values:

vin	sale_date_no_time
C: 1zvft80n255109966	2014-12-31
A: 2c3la63h26h278454	2014-12-31
B: 1ftpw14588fa92105	2014-12-31

[242]:
```
df_clean.describe()
```

[242]:

```
                                 my      condition       odometer  \
count                        472256  472256.000000  472256.000000
mean   2010-03-19 02:54:41.115327232      30.775442   66694.574030
min              1990-01-01 00:00:00       1.000000       1.000000
25%              2008-01-01 00:00:00      24.000000   28134.750000
50%              2012-01-01 00:00:00      35.000000   51078.500000
75%              2013-01-01 00:00:00      41.000000   96583.250000
max              2015-01-01 00:00:00      49.000000  999999.000000
std                             NaN      13.286012   51934.169802

                 mmr  selling_price               sale_date_no_time  \
count  472256.000000  472256.000000                          472256
mean    13837.860186   13691.326331  2015-03-06 09:03:35.699959040
min        25.000000       1.000000             2014-01-01 00:00:00
25%      7425.000000    7200.000000             2015-01-21 00:00:00
50%     12300.000000   12200.000000             2015-02-16 00:00:00
75%     18300.000000   18200.000000             2015-05-20 00:00:00
max    182000.000000  230000.000000             2015-07-20 00:00:00
std      9532.263021    9612.998462                             NaN

          age_months
count  472256.000000
mean       59.573152
min         0.000000
25%        25.490000
50%        40.970000
75%        86.050000
max       305.410000
std        45.722912
```

There are some strange rows in the resulting dataframe: - min odomter value is 1, which is not possible. We want to allow only values greater than 50, - min selling price is 1, which is not possible. We want to allow only values greater than 100.

[243]:

```
#let's drop the rows that does not respect our analysis criteria
df_clean = df_clean[(df_clean['odometer'] >= 50) & (df_clean['selling_price']␣
 ↪>= 100)]
df_clean.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 471013 entries, 0 to 472255
Data columns (total 17 columns):
 #   Column              Non-Null Count   Dtype
---  ------              --------------   -----
 0   my                  471013 non-null  datetime64[ns]
 1   maker               471013 non-null  object
 2   model               471013 non-null  object
 3   trim                471013 non-null  object
```

```
 4   category          471013 non-null  object
 5   transmission      471013 non-null  object
 6   vin               471013 non-null  object
 7   state_sale        471013 non-null  object
 8   condition         471013 non-null  float64
 9   odometer          471013 non-null  float64
 10  ext_color         471013 non-null  object
 11  int_color         471013 non-null  object
 12  seller            471013 non-null  object
 13  mmr               471013 non-null  float64
 14  selling_price     471013 non-null  float64
 15  sale_date_no_time 471013 non-null  datetime64[ns]
 16  age_months        471013 non-null  float64
dtypes: datetime64[ns](2), float64(5), object(10)
memory usage: 64.7+ MB
```

Let's check now if there are still some other strange min value in other columns.

[244]:
```
df_clean.describe()
```

[244]:
```
                                  my      condition       odometer  \
count                         471013  471013.000000  471013.000000
mean   2010-03-22 19:54:23.611832832      30.820205   66870.501702
min              1990-01-01 00:00:00       1.000000      50.000000
25%              2008-01-01 00:00:00      24.000000   28293.000000
50%              2012-01-01 00:00:00      35.000000   51253.000000
75%              2013-01-01 00:00:00      42.000000   96715.000000
max              2015-01-01 00:00:00      49.000000  999999.000000
std                              NaN      13.254520   51889.444899

                 mmr  selling_price              sale_date_no_time  \
count  471013.000000  471013.000000                         471013
mean    13846.265814   13709.700295  2015-03-06 09:34:23.109935360
min        25.000000     100.000000            2014-01-01 00:00:00
25%      7450.000000    7200.000000            2015-01-21 00:00:00
50%     12350.000000   12200.000000            2015-02-16 00:00:00
75%     18300.000000   18200.000000            2015-05-21 00:00:00
max    182000.000000  230000.000000            2015-07-13 00:00:00
std      9529.211854    9603.696452                            NaN

          age_months
count  471013.000000
mean       59.452027
min         0.000000
25%        25.490000
50%        40.970000
75%        86.010000
max       305.410000
```

```
std        45.586947
```

The mmr values seems ok: mmr is a value that is calculated by the Manheim Market Report, and it's not a real value, but a reference value. It's normal to have some low values, as it's a reference value, and it's not the real selling price.

It might be a good idea to create a new column "gain_pct" that will be the difference between the selling price and the mmr value, divided by the mmr value. This will give us an idea of the gain percentage of the selling price, compared to the mmr value.

[245]:
```
df_clean['gain_pct'] = round(((df_clean['selling_price'] - df_clean['mmr']) /
 ↪df_clean['mmr']) * 100,2)
df_clean['gain_pct'].describe()
```

[245]:
```
count    471013.000000
mean         -0.583275
std          34.820589
min         -97.710000
25%          -7.190000
50%          -0.360000
75%           5.560000
max        8033.330000
Name: gain_pct, dtype: float64
```

The gain percentage is calculated correctly, but there is a min and max value that are very large: -97.71% as min and 8033.33% as max. We need to filter, allowing only values between -50% and 200%.

[246]:
```
df_clean = df_clean[(df_clean['gain_pct'] >= -50) & (df_clean['gain_pct'] <=
 ↪200)]
df_clean['gain_pct'].describe()
```

[246]:
```
count    460358.000000
mean          0.224983
std          17.848041
min         -50.000000
25%          -6.490000
50%           0.000000
75%           5.720000
max         200.000000
Name: gain_pct, dtype: float64
```

[247]:
```
df_clean.describe()
df_clean.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 460358 entries, 0 to 472255
Data columns (total 18 columns):
 #   Column             Non-Null Count   Dtype
```

```
---  ------             --------------  -----
 0   my                 460358 non-null  datetime64[ns]
 1   maker              460358 non-null  object
 2   model              460358 non-null  object
 3   trim               460358 non-null  object
 4   category           460358 non-null  object
 5   transmission       460358 non-null  object
 6   vin                460358 non-null  object
 7   state_sale         460358 non-null  object
 8   condition          460358 non-null  float64
 9   odometer           460358 non-null  float64
 10  ext_color          460358 non-null  object
 11  int_color          460358 non-null  object
 12  seller             460358 non-null  object
 13  mmr                460358 non-null  float64
 14  selling_price      460358 non-null  float64
 15  sale_date_no_time  460358 non-null  datetime64[ns]
 16  age_months         460358 non-null  float64
 17  gain_pct           460358 non-null  float64
dtypes: datetime64[ns](2), float64(6), object(10)
memory usage: 66.7+ MB
```

Now all the columns seems to have a coherent value, and we can proceed with the analysis.

Note: the cleaned dataset is representative of 82.38% of the original dataset.

1.4 Data Analysis

Let's imagine some business questions that we need to answer for a Company that is interested in the vehicle sales market.

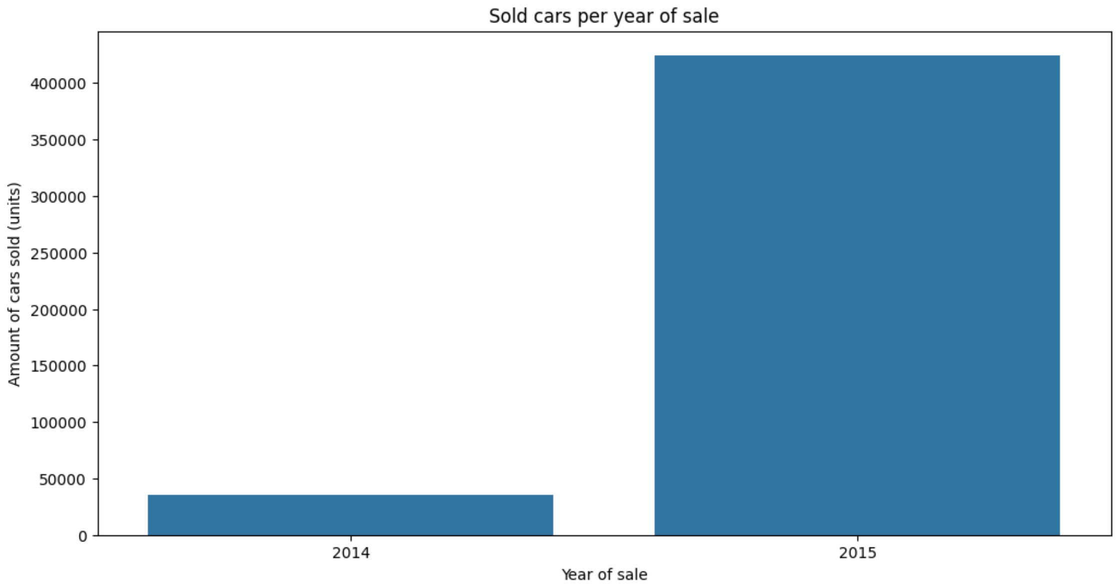
1. Market Trends Analysis

Business Need: Understand how the used car market has evolved over time.

Possible Reports: - Sales Volume by Year and Make: Analyze the distribution of vehicle sales by model year and make. This will help identify trends in the popularity of certain brands and models over time. - Price Trends Over Time: Track the average selling prices and MMR values across different years to see how vehicle values have changed. - Seasonal Sales Patterns: Investigate if there are specific times of the year when vehicle sales peak or dip.

[248]:
```
#let's make some plots to visualize the data. We will use the seaborn library␣
↪for this task.
df_clean['sale_year'] = df_clean['sale_date_no_time'].dt.year

#let's create a graph to visualize the sale volumes (amount of cars sold) per␣
↪year of sale
plt.figure(figsize=(12,6))
sns.countplot(data=df_clean, x='sale_year')
```

```
plt.title('Sold cars per year of sale')
plt.xlabel('Year of sale')
plt.ylabel('Amount of cars sold (units)')
plt.show()
```



This analysis is not really helpful, as the dataset is not complete: we have only data from 2014 and 2015, and the dataset is not complete for the whole year 2014. We will skip analysis based on sale year comparison.

Let's try to analyze the sales volume by month of the year.

[249]:
```
df_clean['sale_month'] = df_clean['sale_date_no_time'].dt.month
df_clean.groupby('sale_year')['sale_month'].unique()
```

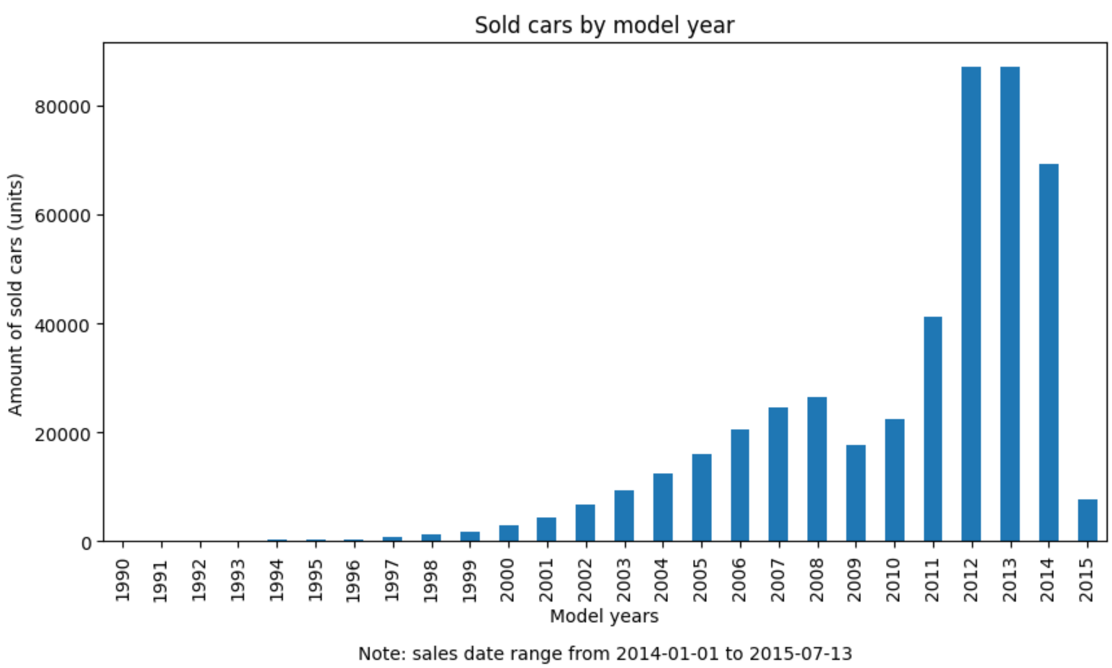
[249]:
```
sale_year
2014                [12, 1, 2]
2015    [5, 2, 3, 6, 1, 7, 4]
Name: sale_month, dtype: object
```

There are holes in the months: there are no data from March to november in 2014, and from August to December in 2015. To use this set to analyze the sales volume by month, might be not the best idea. We will skip this analysis, based on the year of the sale.

Let's check the amount of sales based on the vehicle model year.

[250]:
```
#before to start let's check the value range of the "sale_date_no_time" column
min_date = df_clean['sale_date_no_time'].min().strftime('%Y-%m-%d')
max_date = df_clean['sale_date_no_time'].max().strftime('%Y-%m-%d')
```

```
df_clean['my'] = df_clean['my'].dt.year #to convert the column "my" to year␣
↪format as integer value

#let's create a grpah to check the distribution of the sold cars by year, using␣
↪, and a bar graph
df_clean['my'].value_counts().sort_index().plot(kind='bar', figsize=(10,5))
plt.title('Sold cars by model year')
plt.xlabel('Model years')
plt.ylabel('Amount of sold cars (units)')
note =f'Note: sales date range from {min_date} to {max_date}'
plt.text(x=0.5, y=-0.225, s=note, ha='center', va='center', transform=plt.gca().
↪transAxes)
plt.show()

print('Top 5 model years with more cars sold:')
df_clean['my'].value_counts().head(5)
```




```
Top 5 model years with more cars sold:
```

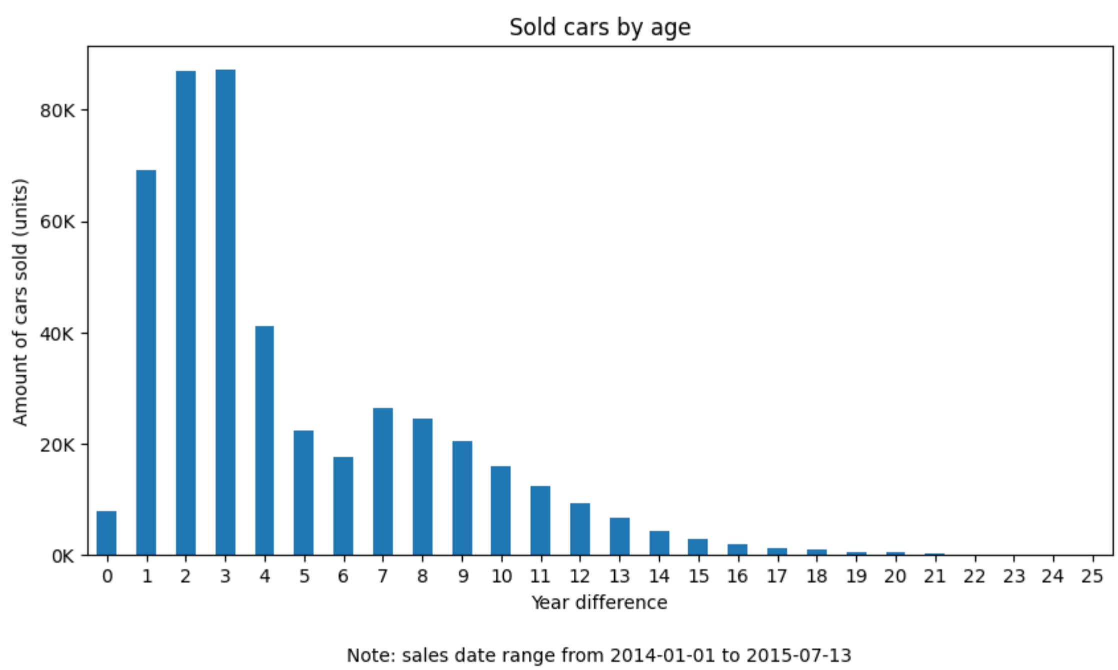
[250]:

```
my
2012    87130
2013    87066
2014    69162
2011    41157
2008    26361
```

```
Name: count, dtype: int64
```

We can create a better graph that will show the amount of sales based per each year difference with 2015.

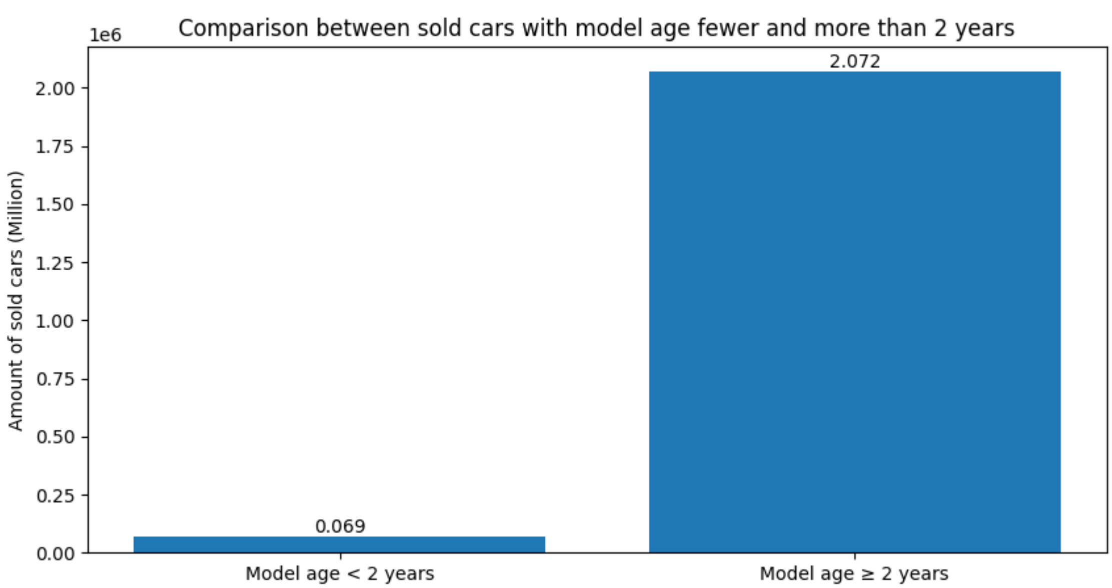
[251]:
```
#let's create a graph the amount of sold cars by year difference between the␣
↪2015 and the model year
diff_years = 2015 - df_clean['my']
diff_years.value_counts().sort_index().plot(kind='bar', figsize=(10,5))
plt.title('Sold cars by age')
plt.xlabel('Year difference')
plt.ylabel('Amount of cars sold (units)')

formatter = ticker.FuncFormatter(lambda x, pos: '{:,.0f}K'.format(x/1000))
plt.gca().yaxis.set_major_formatter(formatter)

note =f'Note: sales date range from {min_date} to {max_date}'
plt.text(x=0.5, y=-0.2, s=note, ha='center', va='center', transform=plt.gca().
↪transAxes)
plt.xticks(rotation=0)
plt.show()
```



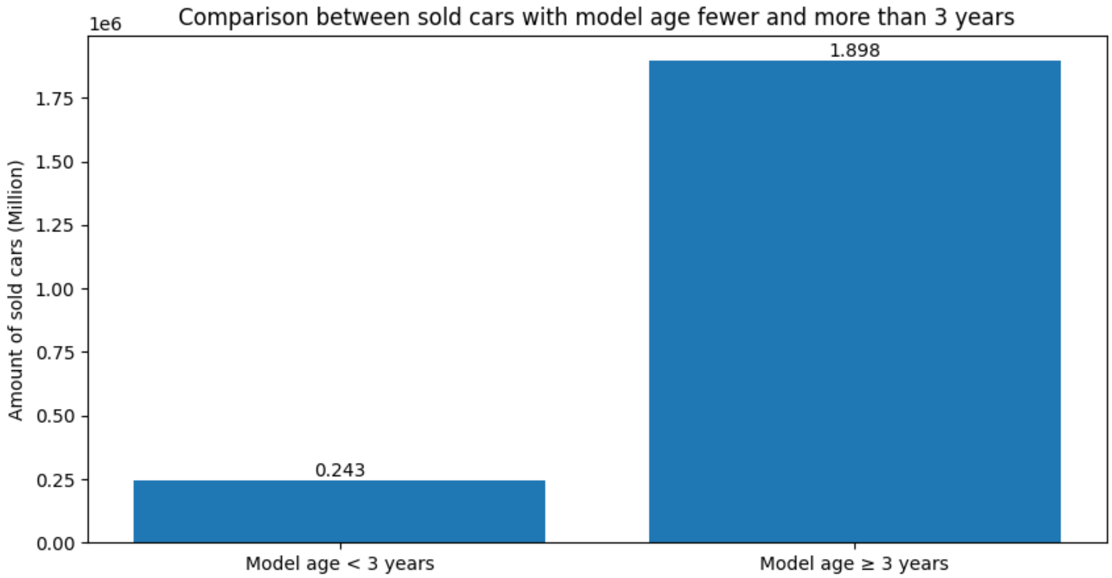
[252]:
```
#let's compare the total amount of sold cars with an year difference fewer and␣
↪more than x years
x = range(2,6,1)
```

```
for x in x:
    more_equal_amount = diff_years[diff_years >= x].sum()
    fewer_amount = diff_years[diff_years < x].sum()

    print(f'Amount of sold cars with an age fewer than {x} years:',␣
 ↪fewer_amount, '(quota:', round(fewer_amount/
 ↪(fewer_amount+more_equal_amount),2)*100, '%)')
    print(f'Amount of sold cars with an age more than {x} years:',␣
 ↪more_equal_amount, '(quota:', round(more_equal_amount/
 ↪(fewer_amount+more_equal_amount),2)*100, '%)')

    #let's create a graph to comapre in the same image the amount of sold cars␣
 ↪fewer and more than x years, adding the values in the bars
    plt.figure(figsize=(10,5))
    plt.bar([f'Model age < {x} years',f'Model age   {x} years'],[fewer_amount,␣
 ↪more_equal_amount])
    plt.text(0, fewer_amount, round(fewer_amount/1e6,3), ha='center',␣
 ↪va='bottom')
    plt.text(1, more_equal_amount, round(more_equal_amount/1e6,3), ha='center',␣
 ↪va='bottom')
    plt.title('Comparison between sold cars with model age fewer and more than␣
 ↪'+ str(x) + ' years')
    plt.ylabel('Amount of sold cars (Million)')
    plt.show()
```

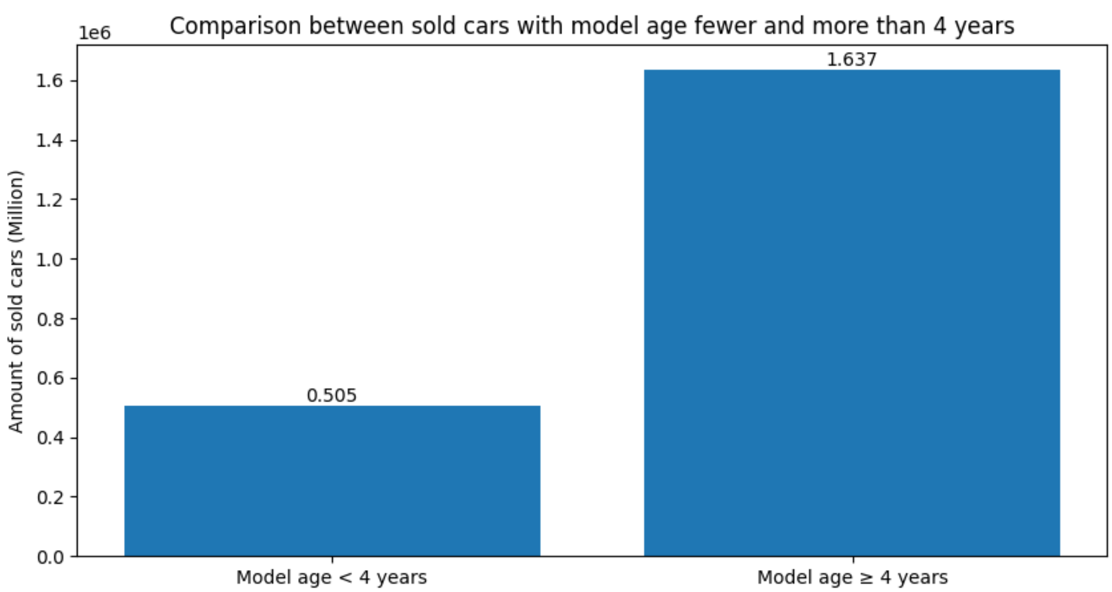
```
Amount of sold cars with an age fewer than 2 years: 69162 (quota: 3.0 %)
Amount of sold cars with an age more than 2 years: 2072257 (quota: 97.0 %)
```



```
Amount of sold cars with an age fewer than 3 years: 243294 (quota: 11.0 %)
Amount of sold cars with an age more than 3 years: 1898125 (quota: 89.0 %)
```

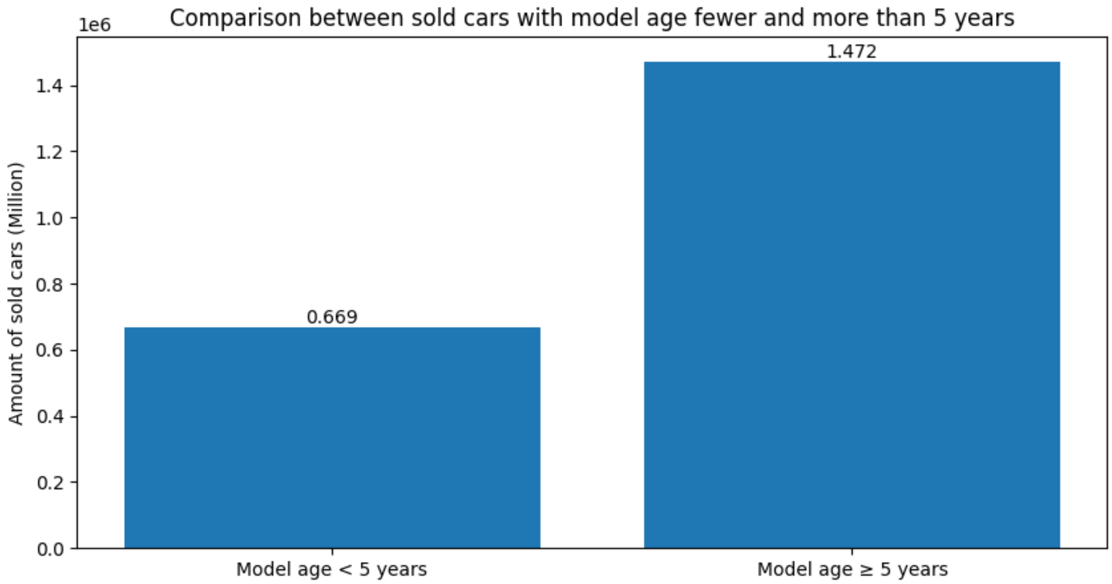



```
Amount of sold cars with an age fewer than 4 years: 504684 (quota: 24.0 %)
Amount of sold cars with an age more than 4 years: 1636735 (quota: 76.0 %)
```




```
Amount of sold cars with an age fewer than 5 years: 669312 (quota: 31.0 %)
```

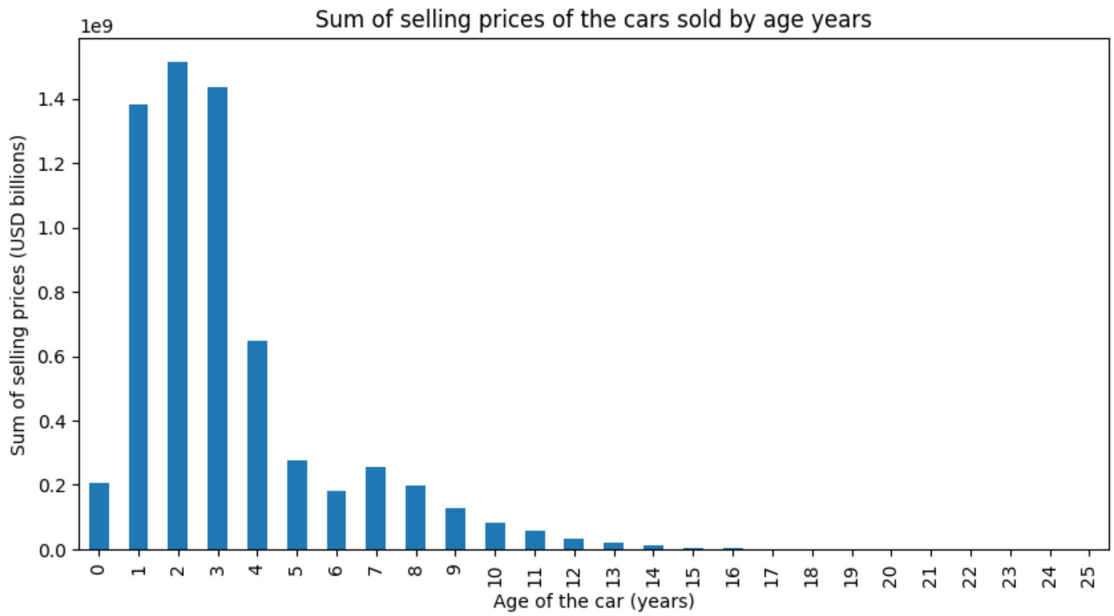
```
Amount of sold cars with an age more than 5 years: 1472107 (quota: 69.0 %)
```



We can say that around one third (31%) of all the sales are from vehicles that have an age of 4 years or less. Indicating that the most interesting age for used cards sales is max 4 years.

In case we consider the most profitable vehicles, we can analyze the selling prices values, distributed by the age of the vehicle. Let's do it.

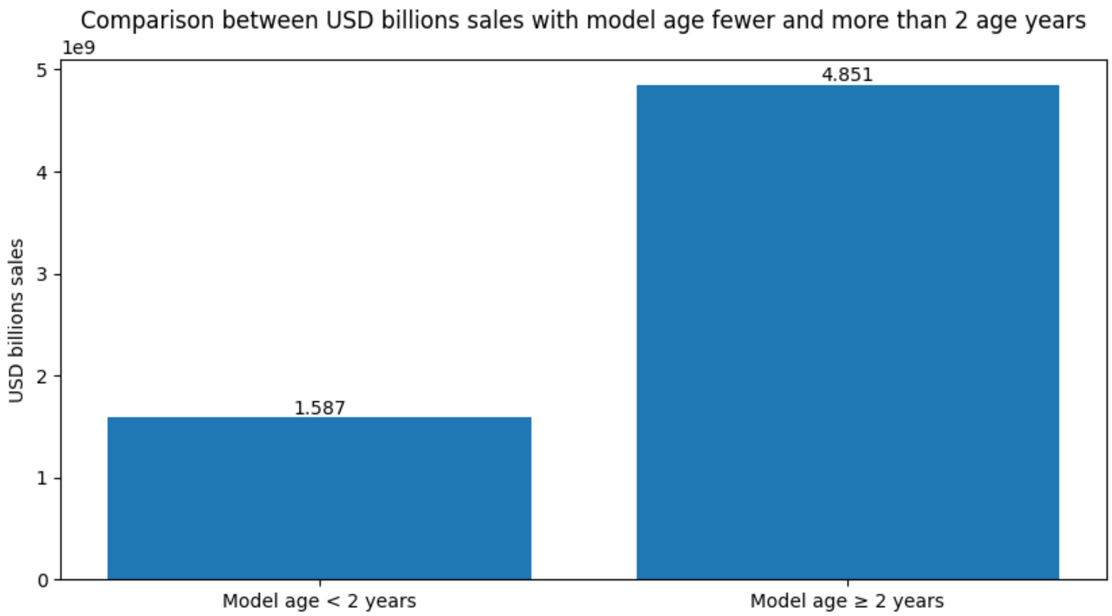
[253]:
```
#this time let's create a new column "age_years" in the df, with the age of the␣
↪car in years (int), using the difference between a fix value of 2015 and the␣
↪sale date and the year of the car
df_clean['age_years'] = diff_years

#let's create a graph to visualize the sum of all the selling prices of the␣
↪cars sold by year, grouped by age_years values
df_clean.groupby('age_years')['selling_price'].sum().plot(kind='bar',␣
↪figsize=(10,5))
plt.title('Sum of selling prices of the cars sold by age years')
plt.xlabel('Age of the car (years)')
plt.ylabel('Sum of selling prices (USD billions)')
plt.show()
```



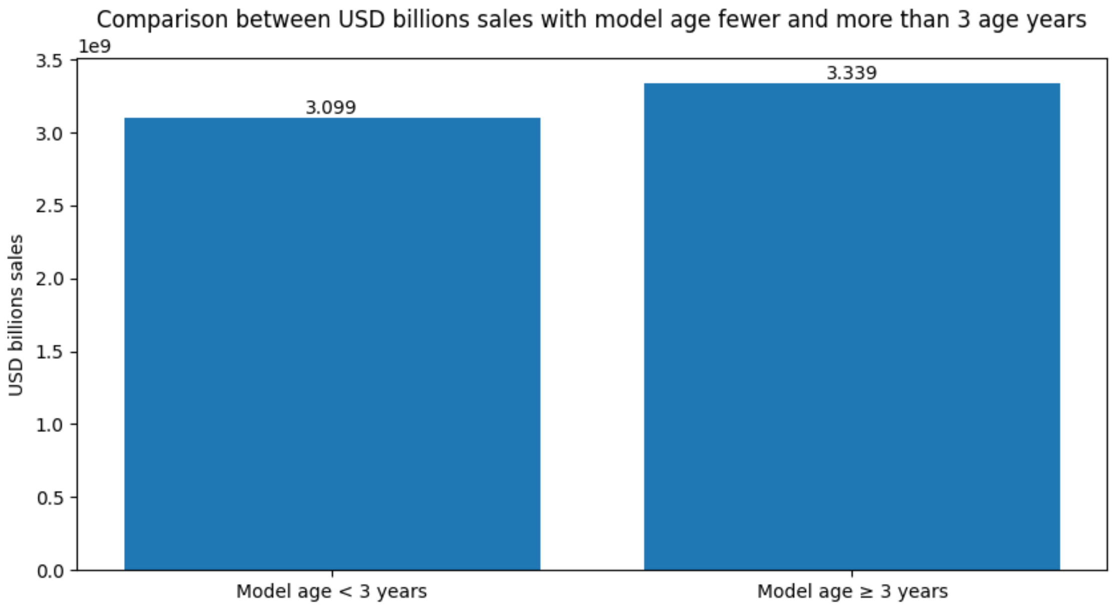
[254]:
```
#let's compare the total amount of sold cars with an year difference fewer and more than x years
x = range(2,6,1)

for x in x:
    more_equal_prices = df_clean[df_clean['age_years'] >= x]['selling_price'].sum()
    fewer_prices_prices = df_clean[df_clean['age_years'] < x]['selling_price'].sum()

    print(f'Amount of USD billions sales for models with age fewer than {x} years:', fewer_prices_prices, '(quota: ', round(fewer_prices_prices/(fewer_prices_prices+more_equal_prices),2), ')')
    print(f'Amount of USD billions sales for models with age fewer than {x} years:', more_equal_prices, '(quota: ', round(more_equal_prices/(fewer_prices_prices+more_equal_prices),2), ')')

    #let's create a graph to comapre in the same image the amount of sold cars fewer and more than x years, adding the values in the bars
    plt.figure(figsize=(10,5))
    plt.bar([f'Model age < {x} years',f'Model age   {x} years'],[fewer_prices_prices, more_equal_prices])
    plt.text(0, fewer_prices_prices, round(fewer_prices_prices/1e9,3), ha='center', va='bottom')
    plt.text(1, more_equal_prices, round(more_equal_prices/1e9,3), ha='center', va='bottom')
```

```
    plt.title('Comparison between USD billions sales with model age fewer and␣
↪more than '+ str(x) + ' age years')
    plt.ylabel('USD billions sales')
    plt.show()
```

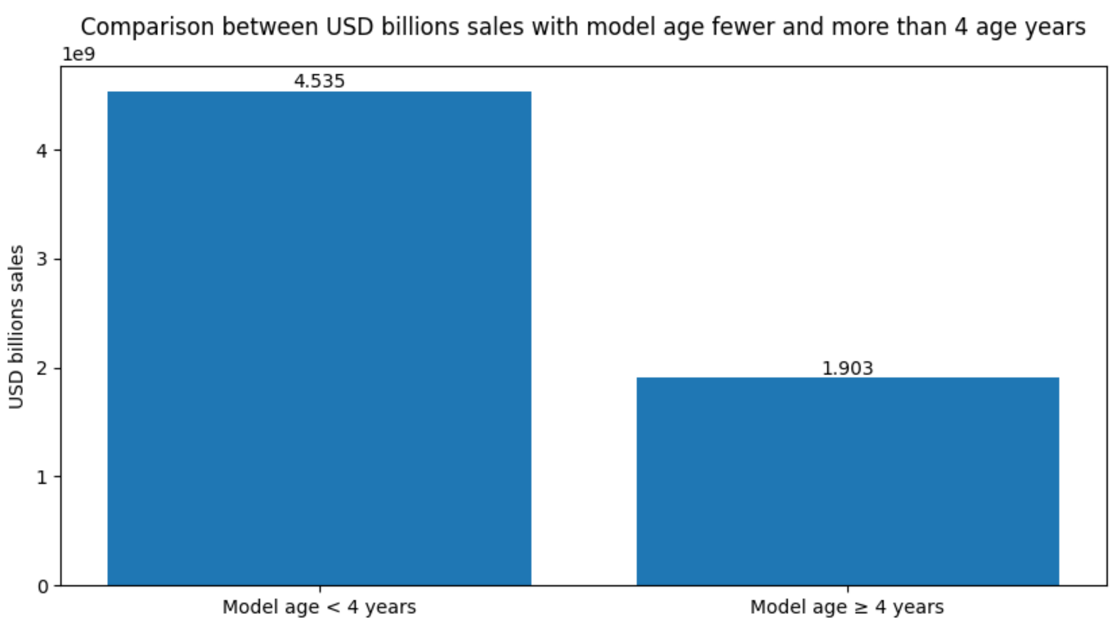
```
Amount of USD billions sales for models with age fewer than 2 years:
1586946225.0 (quota:  0.25 )
Amount of USD billions sales for models with age fewer than 2 years:
4851087645.0 (quota:  0.75 )
```




```
Amount of USD billions sales for models with age fewer than 3 years:
3099336781.0 (quota:  0.48 )
Amount of USD billions sales for models with age fewer than 3 years:
3338697089.0 (quota:  0.52 )
```

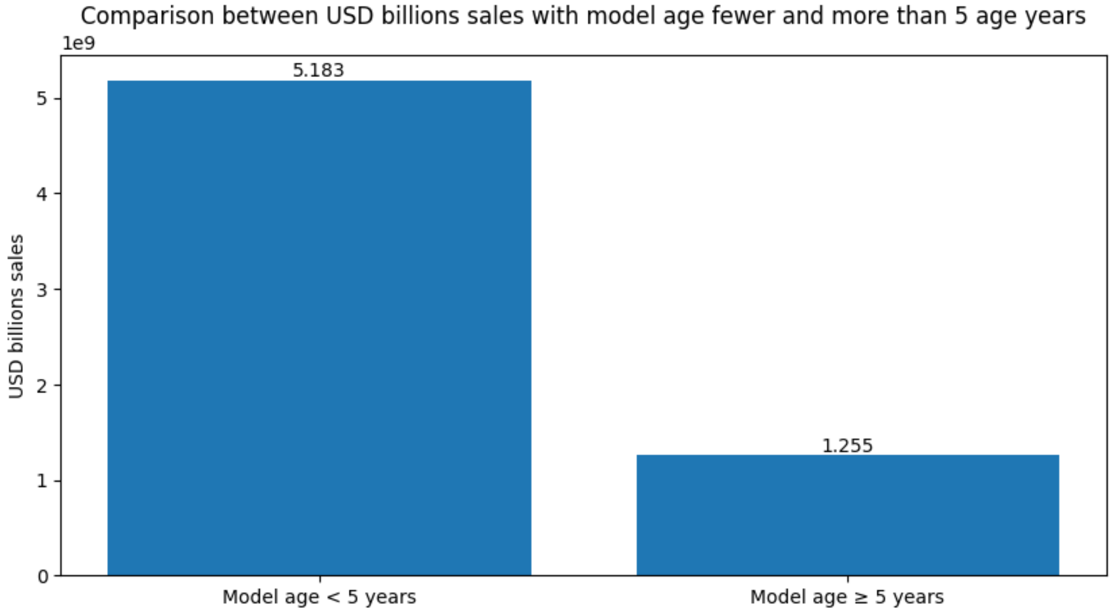



```
Amount of USD billions sales for models with age fewer than 4 years:
4534561168.0 (quota:  0.7 )
Amount of USD billions sales for models with age fewer than 4 years:
1903472702.0 (quota:  0.3 )
```




```
Amount of USD billions sales for models with age fewer than 5 years:
5182912731.0 (quota:  0.81 )
```

```
Amount of USD billions sales for models with age fewer than 5 years:
1255121139.0 (quota:  0.19 )
```



Due to the resulting graph we can affirm that around half of the USD generated by the sales are from vehicles that are 2 years old or less.

So, the gold age for the best sales is 2 years old.

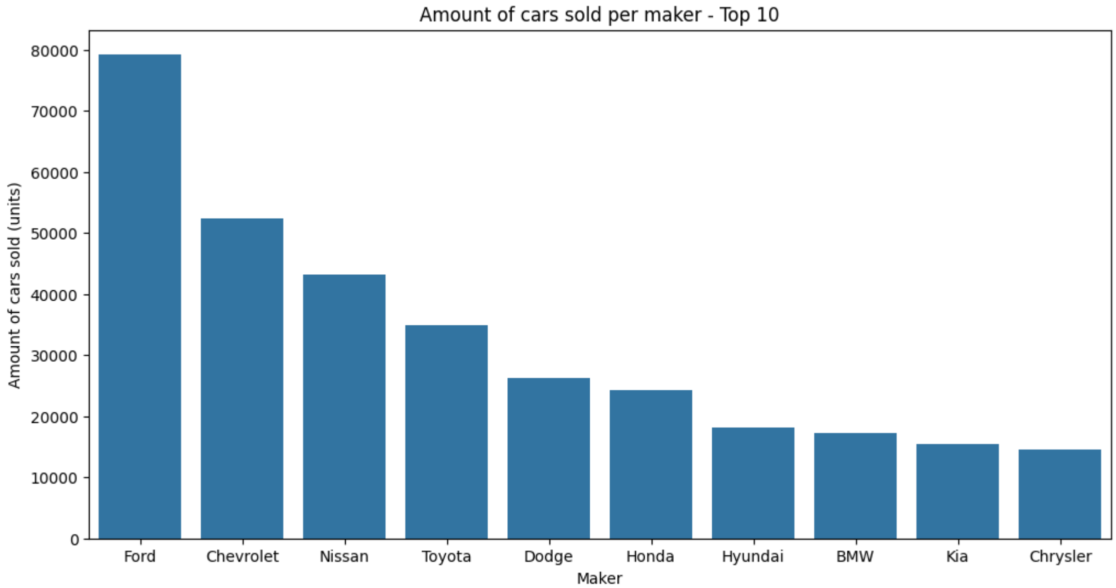
---

2. Customer Preferences

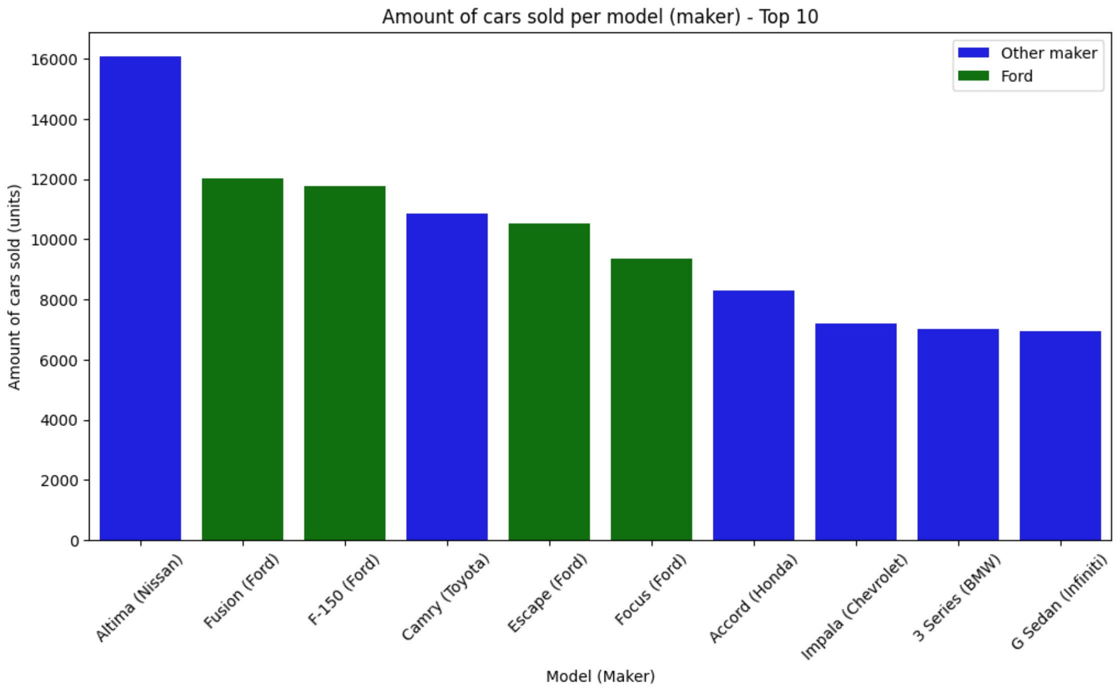
Business Need: Identify what customers are looking for in used cars to optimize inventory and marketing strategies.

Possible Reports: - Popular Makes and Models: Determine which car makers and models are the most popular based on sales volume. - Condition vs. Price Correlation: Analyze how the condition of the vehicle (condition rating) correlates with its selling price to understand customer sensitivity to vehicle condition. - Preferred Features: Look into trends regarding transmission type, body type, and trim options that are in high demand.

[255]:
```
#let's graph the amount of cars sold per maker, showing the top 10 makers
plt.figure(figsize=(12,6))
sns.countplot(data=df_clean, x='maker', order=df_clean['maker'].value_counts().
 ↪index[:10])
plt.title('Amount of cars sold per maker - Top 10')
plt.xlabel('Maker')
plt.ylabel('Amount of cars sold (units)')
plt.show()
```

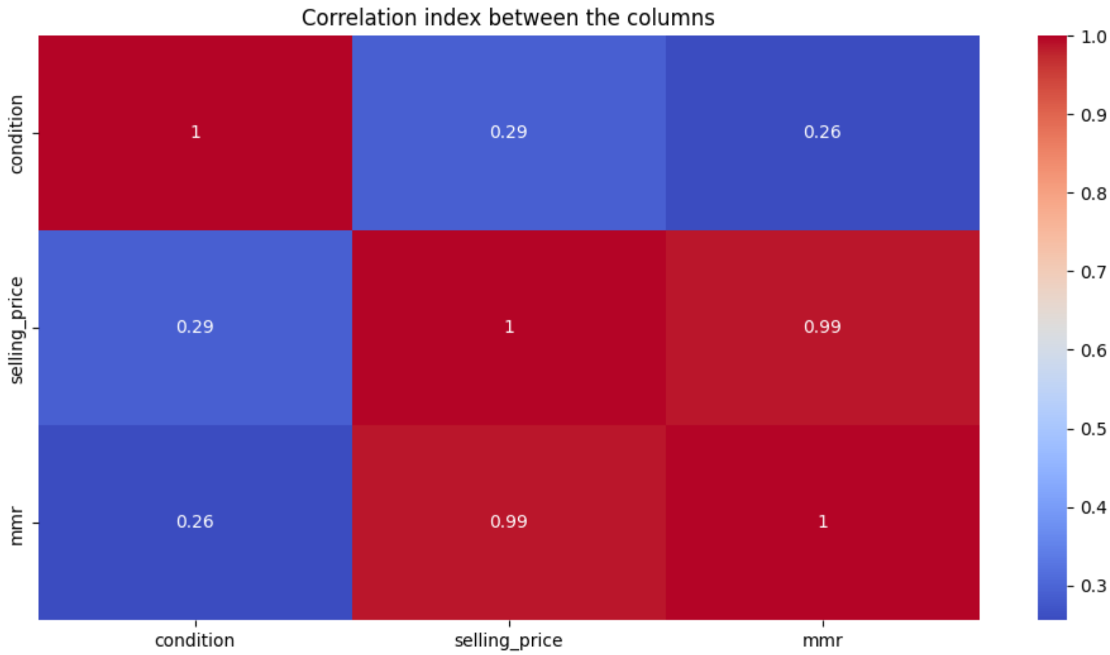


[256]:
```
#le'ts create a new column "model_maker" with the model and the maker of the car
df_clean['model_maker'] = df_clean['model'] + ' (' + df_clean['maker'] + ')'

#let's graph the amount of cars sold per model, showing the top 10 models
plt.figure(figsize=(12,6))
sns.countplot(data=df_clean, x='model_maker', order=df_clean['model_maker'].
 ↪value_counts().index[:10], palette=['green' if 'Ford' in x else 'blue' for x␣
 ↪in df_clean['model_maker'].value_counts().index[:10]])
plt.title('Amount of cars sold per model (maker) - Top 10')
plt.xticks(rotation=45)
plt.xlabel('Model (Maker)')
plt.ylabel('Amount of cars sold (units)')
plt.legend(['Other maker','Ford'])
plt.show()

#let's drop the column "model_maker" since we will not use it anymore
df_clean.drop(columns=['model_maker'], inplace=True)
```



From the previous 2 graphs, is clear why Ford is the top maker in the dataset: 4 models (red bars) are in the top 6 model of the most sold ones. The most Ford's sold model is the Fusion model, which is also the 2nd most sold model in whole dataset.

[257]:
```
#caluculate the correlation between the columns "condition" and "selling_price"
corr = df_clean[['condition','selling_price','mmr']].corr()

#let's create a heatmap to visualize the correlation between the columns
plt.figure(figsize=(12,6))
sns.heatmap(corr, annot=True, cmap='coolwarm')
plt.title('Correlation index between the columns')
plt.show()
```

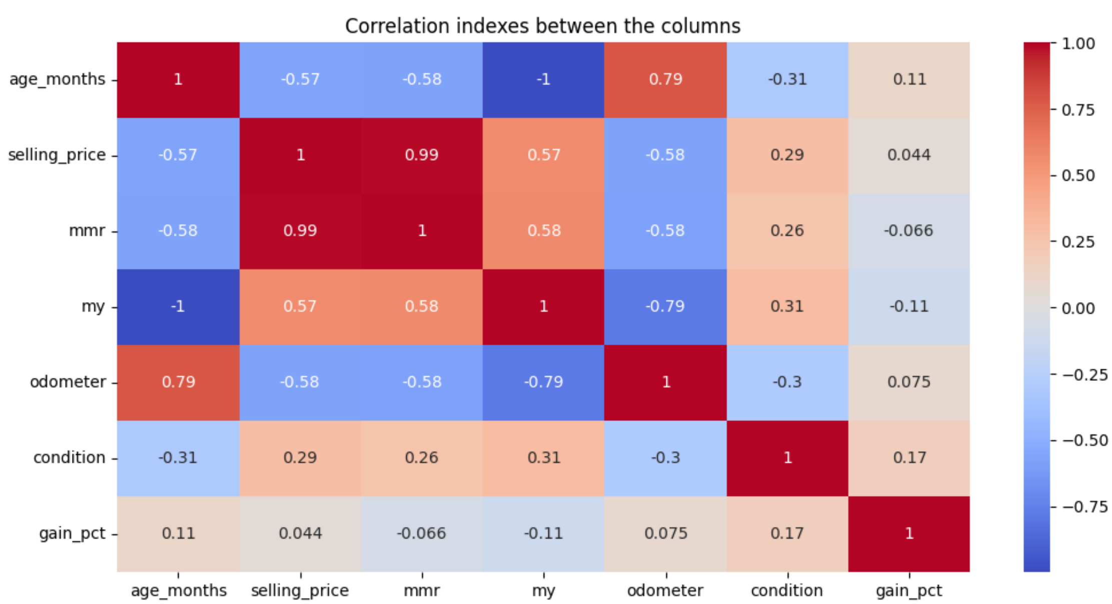


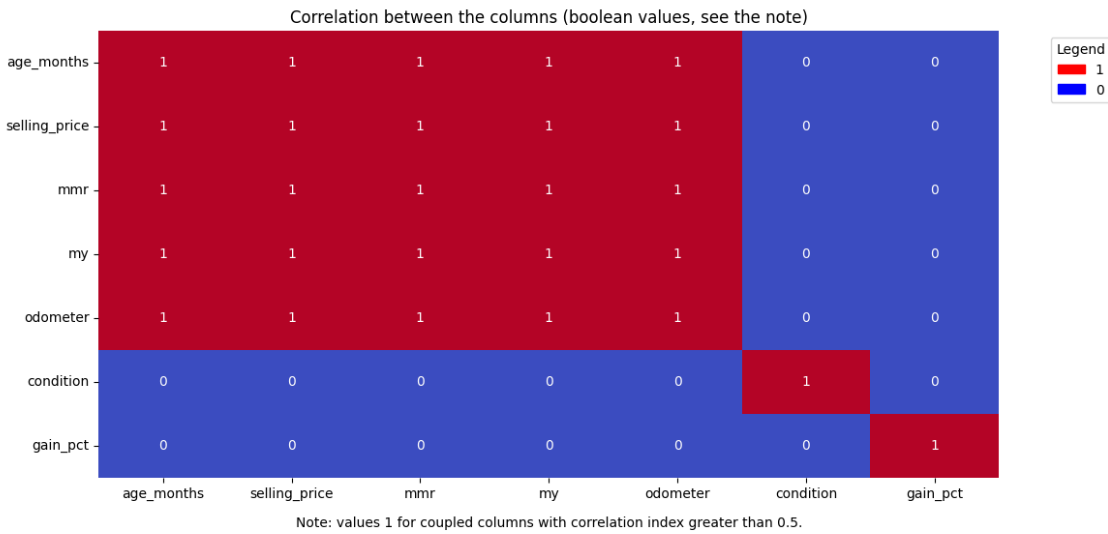
There is a very low correlation between the condition rating and the selling price. This is not surprising, as the condition rating is a subjective value, and it's not always related to the real condition of the vehicle. Same story for mmr value, since it's a reference value, and it's not always related to the real selling price.

To be sure, let's check stronger and obvious correlations.

[258]:
```
corr_multi =␣
 ↪df_clean[['age_months','selling_price','mmr','my','odometer','condition','gain_pct']].
 ↪corr()

corr_coeff = .5

#let's create a heatmap to visualize the correlation between the columns
plt.figure(figsize=(12,6))
sns.heatmap(corr_multi, annot=True, cmap='coolwarm')
plt.title('Correlation indexes between the columns')
plt.show()

#let's create a heatmap to visualize the correlation between the columns with a␣
 ↪correlation greater than the coefficient corr_coeff

import matplotlib.patches as mpatches

corr_multi_filtered = abs(corr_multi) > corr_coeff
plt.figure(figsize=(12,6))
```

```
sns.heatmap(corr_multi_filtered, annot=True, cmap='coolwarm', cbar=False)
plt.title('Correlation between the columns (boolean values, see the note)')
note =f'Note: values 1 for coupled columns with correlation index greater than␣
  ↪{corr_coeff}.'
plt.text(x=0.5, y=-0.1, s=note, ha='center', va='center', transform=plt.gca().
  ↪transAxes)

legend_labels = [mpatches.Patch(color='red', label='1'), mpatches.
  ↪Patch(color='blue', label='0')]
plt.legend(handles=legend_labels, title='Legend', bbox_to_anchor=(1.05, 1),␣
  ↪loc='upper left')

plt.show()

#let's create a new dataframe with the columns that have a correlation greater␣
  ↪than coeff x
columns = corr_multi_filtered.columns
columns_filtered = []
for i in range(len(columns)):
    for j in range(i+1,len(columns)):
        if corr_multi_filtered.iloc[i,j]:
            columns_filtered.append(columns[i])
            columns_filtered.append(columns[j])

columns_filtered = list(set(columns_filtered))

print('The columns with correlation index greater than', corr_coeff, 'are:',␣
  ↪columns_filtered, 'with the following correlation matrix:')
print(corr_multi.loc[columns_filtered,columns_filtered])
```






```
The columns with correlation index greater than 0.5 are: ['odometer',
'selling_price', 'my', 'age_months', 'mmr'] with the following correlation
matrix:
               odometer  selling_price        my  age_months       mmr
odometer       1.000000      -0.577332 -0.785528    0.785486 -0.580889
selling_price -0.577332       1.000000  0.565887   -0.565101  0.986621
my            -0.785528       0.565887  1.000000   -0.999051  0.576314
age_months     0.785486      -0.565101 -0.999051    1.000000 -0.575428
mmr           -0.580889       0.986621  0.576314   -0.575428  1.000000
```

---

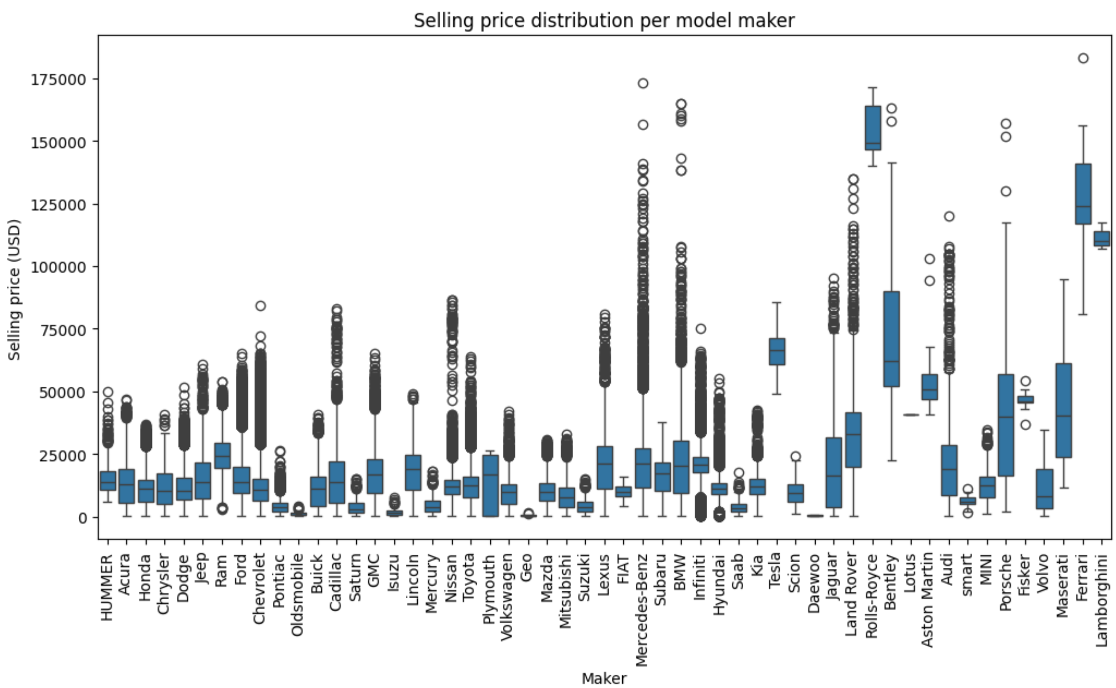
3. Pricing Strategy

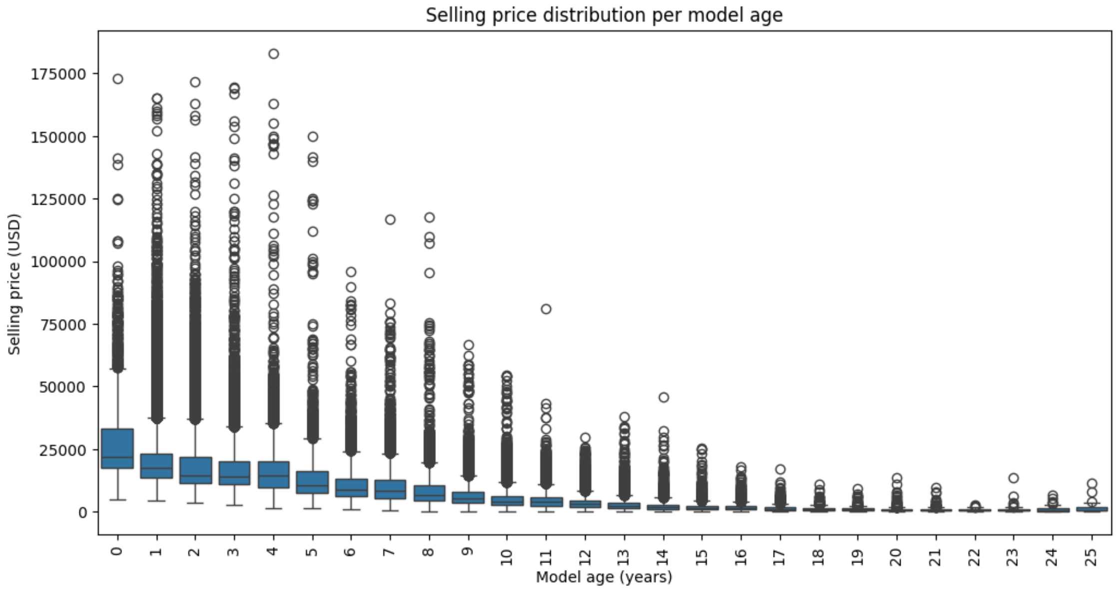
Business Need: develop a competitive pricing strategy that maximizes profit while remaining attractive to buyers.

Possible Reports: - price Distribution Analysis: examine the distribution of selling prices across different car makers, models, and years to identify typical pricing ranges. - geographical Pricing Differences: investigate if there are significant differences in selling prices based on the state of registration or location of the seller.

[259]:
```
#let's export the dataframe to a csv file
df_clean.to_csv(os.getcwd() + '/clean_data/cars_clean.csv', index=False)

#let's create some graph with selling price ditribution per maker and model year
list_var = ['maker','age_years']

for i in list_var:
```

```python
    plt.figure(figsize=(12,6))
    sns.boxplot(data=df_clean, x=i, y='selling_price')
    if i == 'maker':
        plt.title(f'Selling price distribution per model maker')
        plt.xlabel('Maker')
        plt.ylabel('Selling price (USD)')
    else:
        plt.title(f'Selling price distribution per model age')
        plt.xlabel('Model age (years)')
        plt.ylabel('Selling price (USD)')
    plt.xticks(rotation=90)
    plt.show()
```





The above graph clearly show the distribution of the selling price, per car model maker and model year. I've used a boxplot since it's the best way to show the distribution of the data, condensed in little space, and to show the outliers data. In the same time is possible lto have a quick idea of the median, the 25th and 75th percentile, for each model maker and model year. Is also shown the range of data values of the not-outliers data.

Let's have a check to the z-scores of the selling prices, to see the amount of outliers in the dataset.

[260]:

```
#identifying outliers using z-score
abs_z_scores = np.abs(stats.zscore(df_clean[['selling_price']]))

# identify outliers as z-scores greater than x standard deviations from the mean
outliers_coefficient = 3
outliers = (abs_z_scores > outliers_coefficient).any(axis = 1)

df_outliers = df_clean[outliers]
print(df_outliers.groupby(['maker',])['selling_price'].count().
 ↪sort_values(ascending=False).head(5))
print(df_outliers.groupby(['age_years',])['selling_price'].count().
 ↪sort_values(ascending=False).head(5))
```

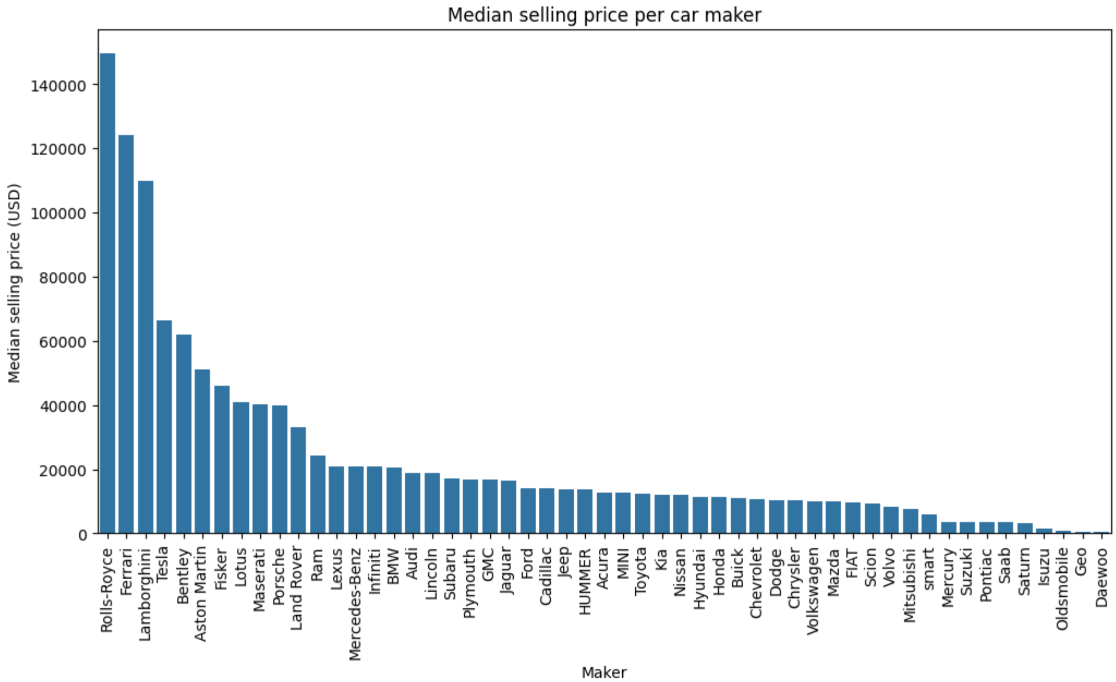
```
maker
BMW              1259
Mercedes-Benz     921
Chevrolet         563
Porsche           537
Ford              420
Name: selling_price, dtype: int64
age_years
```

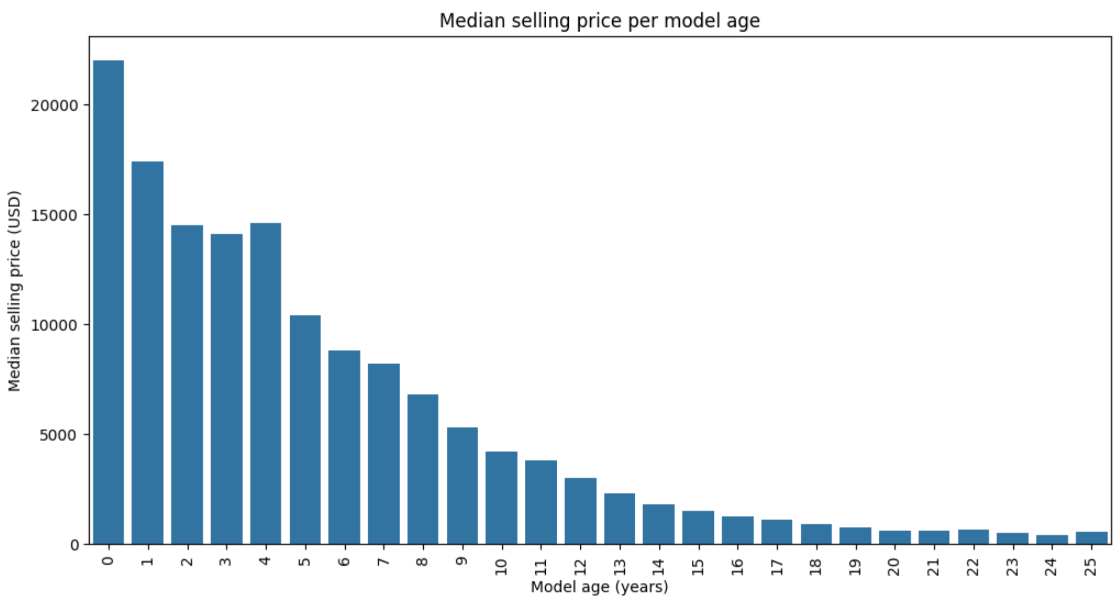
```
1    2028
2    1407
3     957
0     955
4     231
Name: selling_price, dtype: int64
```

In the majority of the cases, there is a considerable amount of outliers than have more than have the z-score more than 3 times the standard deviation, that may affect the mean value. So the median value is a better representation of the central tendency of the data.

So here are the median selling prices, per car maker and model year:

[261]:
```python
median_selling_price_per_car_maker = df_clean.groupby('maker')['selling_price'].
 ↪median().sort_values(ascending=False)
#print(median_selling_price_per_car_maker)

#let's create a graph to visualize the median selling price per car maker
plt.figure(figsize=(12,6))
sns.barplot(x=median_selling_price_per_car_maker.index,␣
 ↪y=median_selling_price_per_car_maker)
plt.title('Median selling price per car maker')
plt.xlabel('Maker')
plt.ylabel('Median selling price (USD)')
plt.xticks(rotation=90)
plt.show()

median_selling_price_per_model_age = df_clean.
 ↪groupby('age_years')['selling_price'].median().sort_values(ascending=False)
#print(median_selling_price_per_model_year)

#let's create a graph to visualize the median selling price per model year
plt.figure(figsize=(12,6))
sns.barplot(x=median_selling_price_per_model_age.index,␣
 ↪y=median_selling_price_per_model_age)
plt.title('Median selling price per model age')
plt.xlabel('Model age (years)')
plt.ylabel('Median selling price (USD)')
plt.xticks(rotation=90)
plt.show()
```

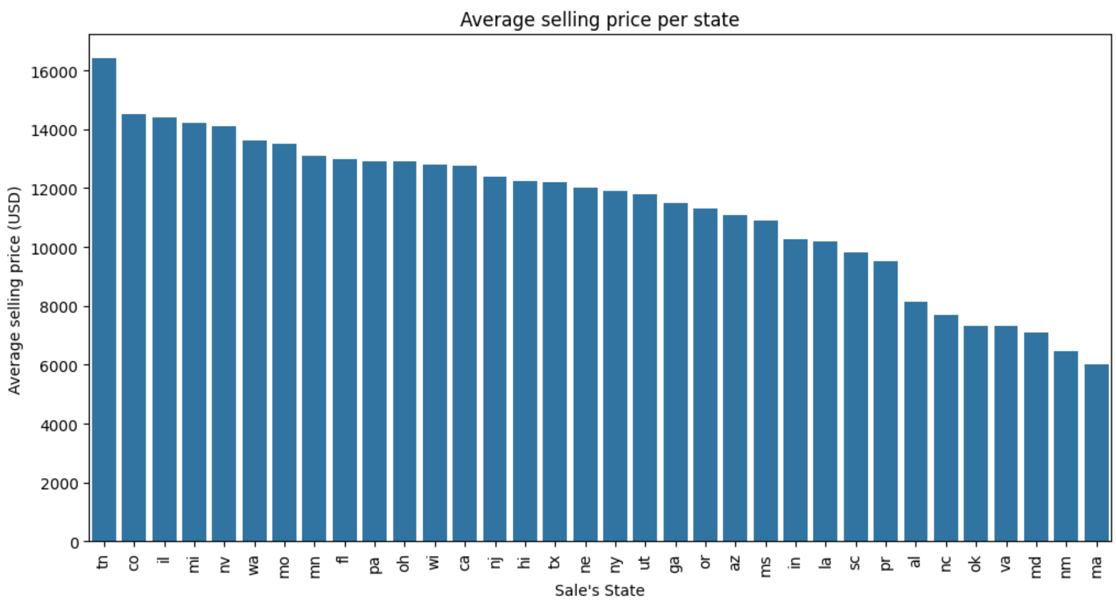




Now let's check the geographical pricing differences.

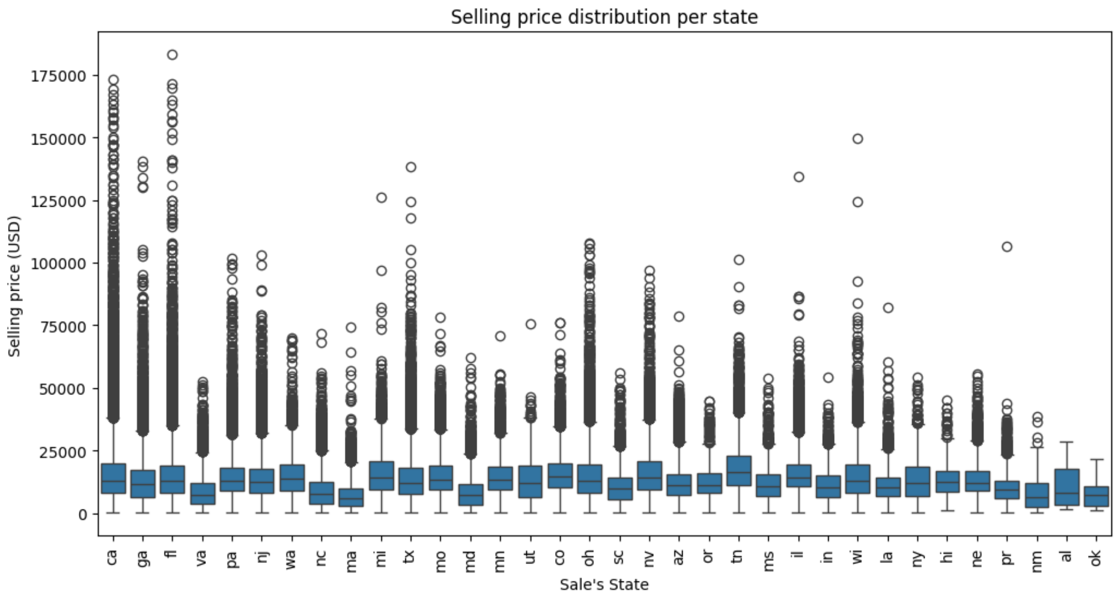
[262]:
```
#graph the average selling price per state
median_prices = df_clean.groupby('state_sale')['selling_price'].median().
 ↪reset_index
```

```
median_prices = median_prices().sort_values('selling_price', ascending=False)

plt.figure(figsize=(12,6))
sns.barplot(data=median_prices, x='state_sale', y='selling_price')
plt.title('Average selling price per state')
plt.xticks(rotation=90)
plt.xlabel("Sale's State")
plt.ylabel('Average selling price (USD)')
plt.show()
```



At this point is possible to list the most profitable states, based on the selling prices of the sales. But how is better to evaluate the profitability of a state? We could use the median selling price, or the mean selling price to compare each state's peformance. What is the better choice? Let's evalute the outliers in the selling prices for states.

[263]:
```
#let's create a boxplot to visualize the selling price median distribution per␣
↪state
plt.figure(figsize=(12,6))
sns.boxplot(data=df_clean, x='state_sale', y='selling_price')
plt.title('Selling price distribution per state')
plt.xlabel("Sale's State")
plt.ylabel('Selling price (USD)')
plt.xticks(rotation=90)
plt.show()
```



From the previous boxplot, showing the distribution of the selling price per state, is clear that the median selling price is a better indicator of the profitability of a state, as the mean selling price is influenced by the outliers…and there are a lot of them in some states. For this reason it's better to use the median selling price to evaluate the profitability of a state.

[264]:
```
#let's create a list of profitable states made by all the states where the␣
 ↪median selling price is higher than the global media of all the states
global_median_price = df_clean['selling_price'].median()
print('The global median price is:', global_median_price)

median_prices_per_state = df_clean.groupby('state_sale')['selling_price'].
 ↪median().reset_index().sort_values('selling_price', ascending=False)
median_prices_per_state.rename(columns={'selling_price':
 ↪'median_selling_price'}, inplace=True)

profitable_states =␣
 ↪median_prices_per_state[median_prices_per_state['median_selling_price'] >␣
 ↪global_median_price]['state_sale'].tolist()
profitable_states =␣
 ↪median_prices_per_state[median_prices_per_state['state_sale'].
 ↪isin(profitable_states)].sort_values('median_selling_price', ascending=False)
profitable_states.rename(columns={'selling_price':'median_selling_price'},␣
 ↪inplace=True)

print(f'\nThe profitable states (with more than {global_median_price}) are:')
profitable_states
```

```
The global median price is: 12400.0
```

```
The profitable states (with more than 12400.0) are:
```

[264]:

	state_sale	median_selling_price
28	tn	16400.0
3	co	14500.0
7	il	14400.0
12	mi	14200.0
20	nv	14100.0
32	wa	13600.0
14	mo	13500.0
13	mn	13100.0
4	fl	13000.0
25	pa	12900.0
22	oh	12900.0
33	wi	12800.0
2	ca	12750.0

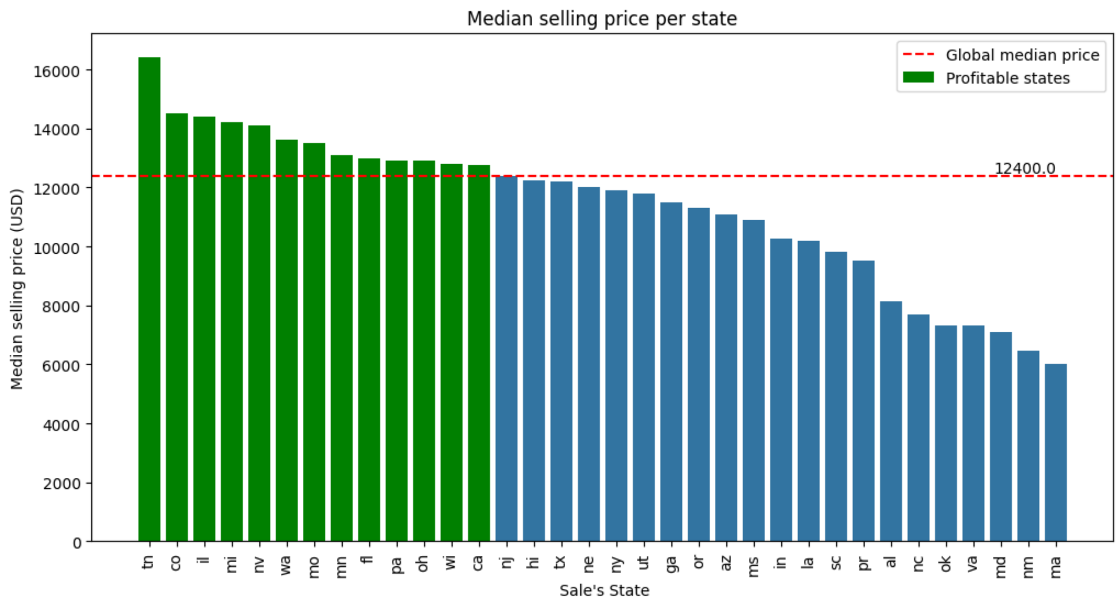
Now we can affirm that due to our analysis, the most profitable states are:

[265]:
```
list_profitable_state = profitable_states['state_sale'].tolist()
list_profitable_state
```

[265]:
```
['tn', 'co', 'il', 'mi', 'nv', 'wa', 'mo', 'mn', 'fl', 'pa', 'oh', 'wi', 'ca']
```

[266]:
```
#let's create a graph the media selling price per state
plt.figure(figsize=(12,6))
sns.barplot(data=median_prices_per_state, x='state_sale',␣
 ↪y='median_selling_price')
plt.title('Median selling price per state')
plt.xticks(rotation=90)
plt.xlabel(" Sale's State")
plt.ylabel('Median selling price (USD)')

#let's add a line to show the global median price in the graph
plt.axhline(global_median_price, color='red', linestyle='--', label='Global␣
 ↪median price')
#let's highlight the profitable states in the graph
plt.bar(profitable_states['state_sale'],␣
 ↪profitable_states['median_selling_price'], color='green', label='Profitable␣
 ↪states')
#let's add the value of the global median price above the line on the right side
plt.text(x=len(median_prices_per_state)-1, y=global_median_price,␣
 ↪s=round(global_median_price,2), ha='right', va='bottom')

#add the legend to the graph
plt.legend()
```

```
plt.show()
```



In this last graph is shown in green the most profitable states, based on the global median selling price: the most profitable states are the ones with an higher median selling price, compared to the global median selling price of whole data set.

---

4. Vehicle Depreciation Analysis

Business Need: Understand how different vehicles depreciate over time to inform resale value projections.

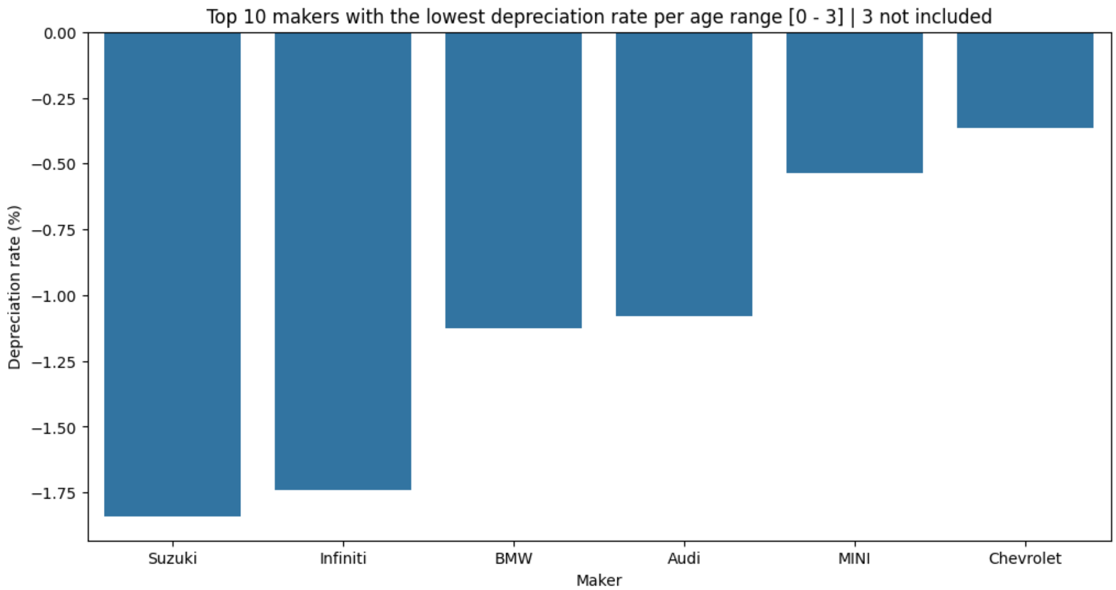
Possible Reports: - Depreciation by Maker: calculate the depreciation rates for different makes and models based on the difference between MMR values and original sale prices over time. - Impact of Odometer Reading on Depreciation: analyze how odometer readings affect the depreciation rate, focusing on different segments of the market (e.g., luxury vs. economy cars). - Residual Value Predictions: create a model to predict the residual value of a vehicle after a certain period, based on factors like make, model, year, condition, and mileage.

Let’s start with the depreciation by maker.

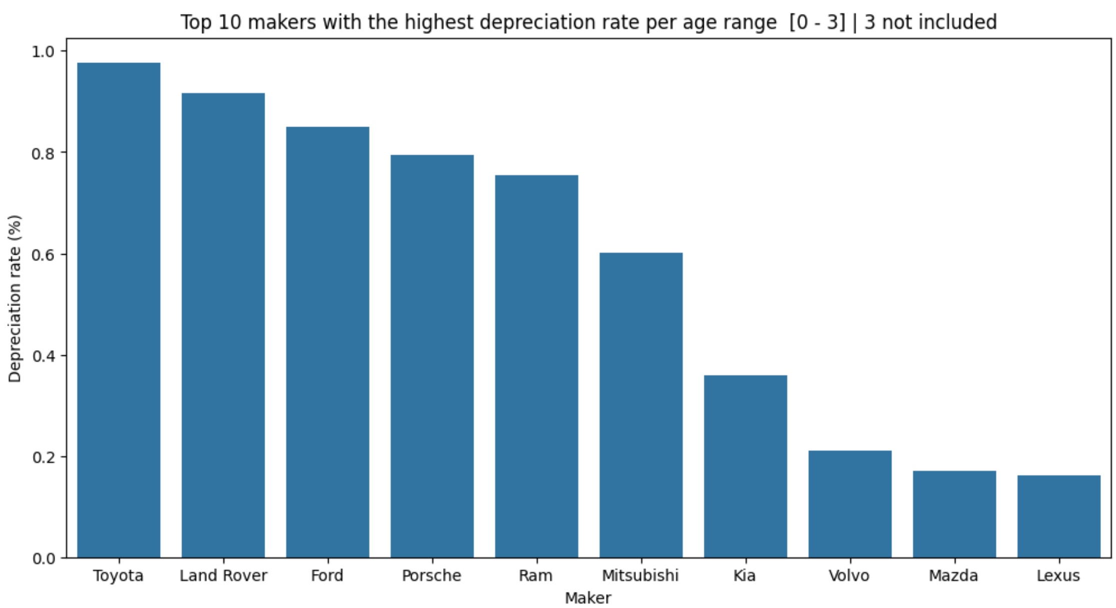
[267]:
```
#let's create sub-dataframes due to the age of the car
age_ranges = [(0, 3), (3, 6), (6, 9)]
df_age_dict = {}

for start, end in age_ranges:
    df_age_dict[f'df_age_{start}_{end}'] = df_clean[(df_clean['age_years'] >=␣
 ↪start) & (df_clean['age_years'] < end)]
```

```
#let's calculate the depreciation rate per maker and model each for the fisrt X␣
↪makers
x = 10
df_depreciation_negative_dict = {}
df_depreciation_positive_dict = {}

for key, df_age in df_age_dict.items():
    df_depreciation_grouped = df_age.groupby(['maker'])[['selling_price',␣
↪'mmr']].median().reset_index()
    df_depreciation_grouped['depreciation_pctg'] =␣
↪round(((df_depreciation_grouped['mmr'] -␣
↪df_depreciation_grouped['selling_price']) / df_depreciation_grouped['mmr'])␣
↪* 100, 3)
    df_depreciation_grouped = df_depreciation_grouped.
↪sort_values('depreciation_pctg', ascending=True)

    df_depreciation_negative_dict[key] =␣
↪df_depreciation_grouped[df_depreciation_grouped['depreciation_pctg'] < 0]
    df_depreciation_positive_dict[key] =␣
↪df_depreciation_grouped[df_depreciation_grouped['depreciation_pctg'] > 0]

#let's graph the depreciation rate per maker for the first X makers
for key in df_age_dict.keys():
    df_depreciation_negative_head = df_depreciation_negative_dict[key].head(x).
↪sort_values('depreciation_pctg', ascending=True)
    df_depreciation_positive_head = df_depreciation_positive_dict[key].head(x).
↪sort_values('depreciation_pctg', ascending=False)

    plt.figure(figsize=(12,6))
    sns.barplot(data=df_depreciation_negative_head, x='maker',␣
↪y='depreciation_pctg')
    plt.title(f'Top {x} makers with the lowest depreciation rate per age range␣
↪[{key[7:8]} - {key[9:10]}] | {key[9:10]} not included')
    plt.xlabel('Maker')
    plt.ylabel('Depreciation rate (%)')
    plt.xticks(rotation=0)
    plt.show()

    plt.figure(figsize=(12,6))
    sns.barplot(data=df_depreciation_positive_head, x='maker',␣
↪y='depreciation_pctg')
    plt.title(f'Top {x} makers with the highest depreciation rate per age range␣
↪ [{key[7:8]} - {key[9:10]}] | {key[9:10]} not included')
    plt.xlabel('Maker')
    plt.ylabel('Depreciation rate (%)')
    plt.xticks(rotation=0)
```

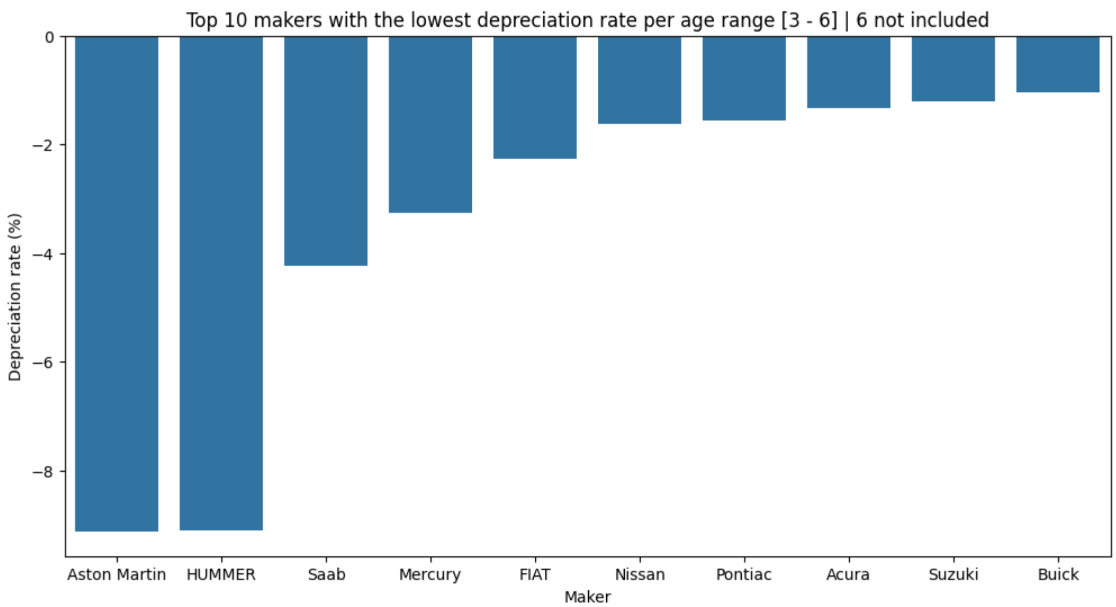
```
plt.show()
```

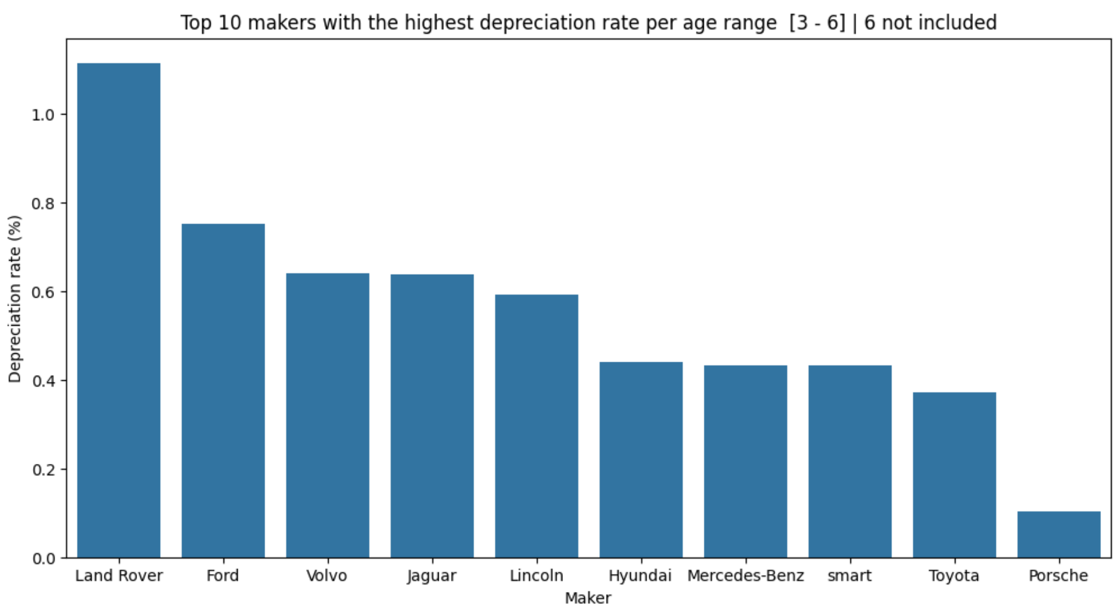


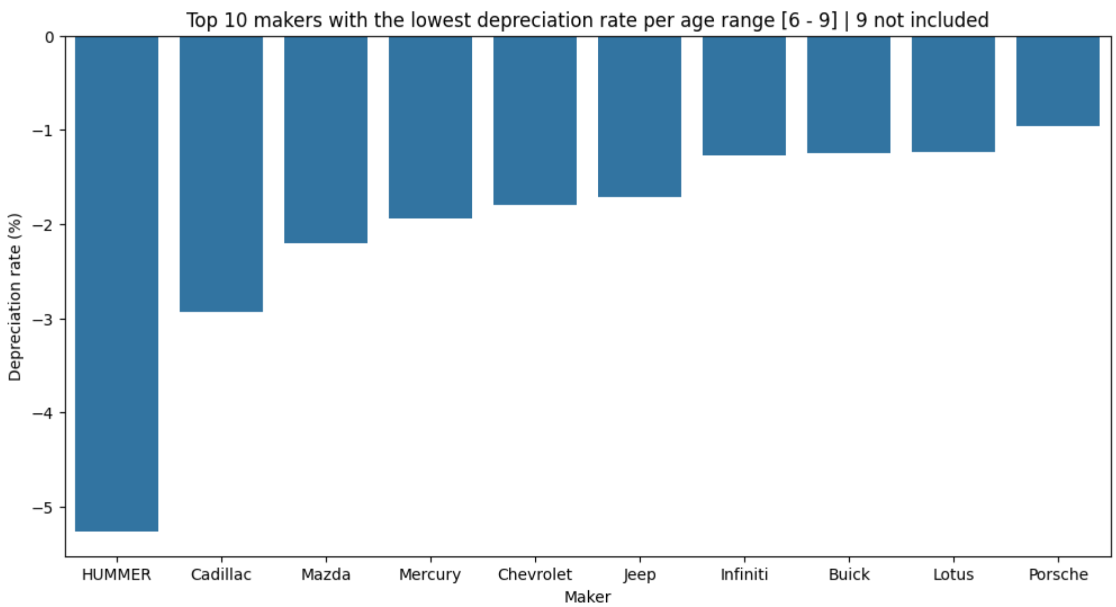
Top 10 makers with the lowest depreciation rate per age range [0 - 3] | 3 not included
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
−1.75
Depreciation rate (%)
Suzuki
Infiniti
BMW
Audi
MINI
Chevrolet
Maker

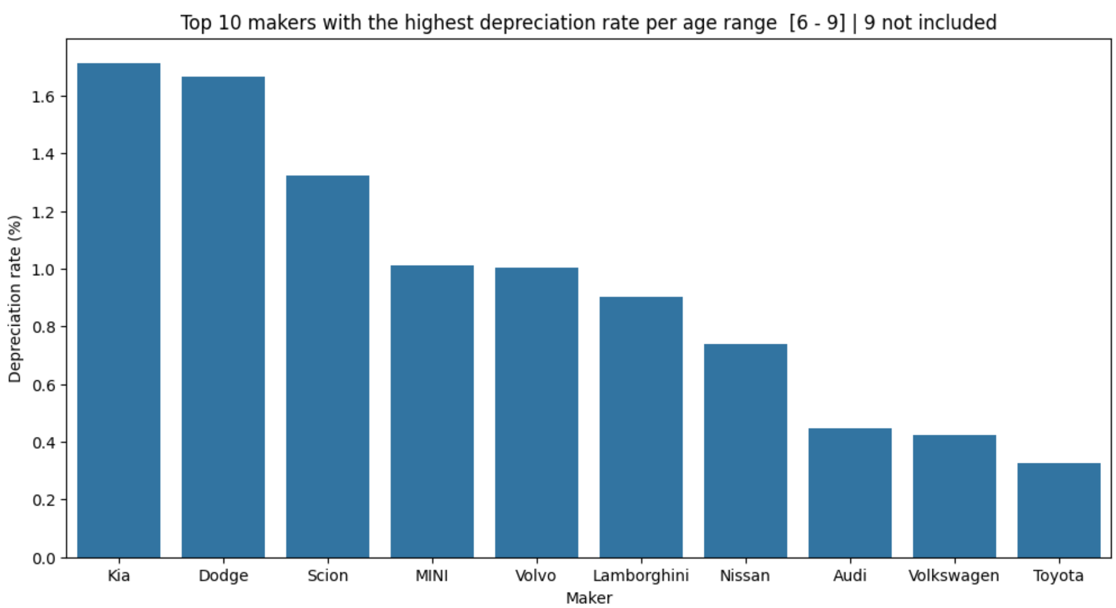


Top 10 makers with the highest depreciation rate per age range [0 - 3] | 3 not included
1.0
0.8
0.6
0.4
0.2
0.0
Depreciation rate (%)
Toyota
Land Rover
Ford
Porsche
Ram
Mitsubishi
Kia
Volvo
Mazda
Lexus
Maker


Top 10 makers with the lowest depreciation rate per age range [3 - 6] | 6 not included
0
−2
−4
−6
−8
Depreciation rate (%)
Aston Martin
HUMMER
Saab
Mercury
FIAT
Nissan
Pontiac
Acura
Suzuki
Buick
Maker


Top 10 makers with the highest depreciation rate per age range [3 - 6] | 6 not included
1.0
0.8
0.6
0.4
0.2
0.0
Depreciation rate (%)
Land Rover
Ford
Volvo
Jaguar
Lincoln
Hyundai
Mercedes-Benz
smart
Toyota
Porsche
Maker



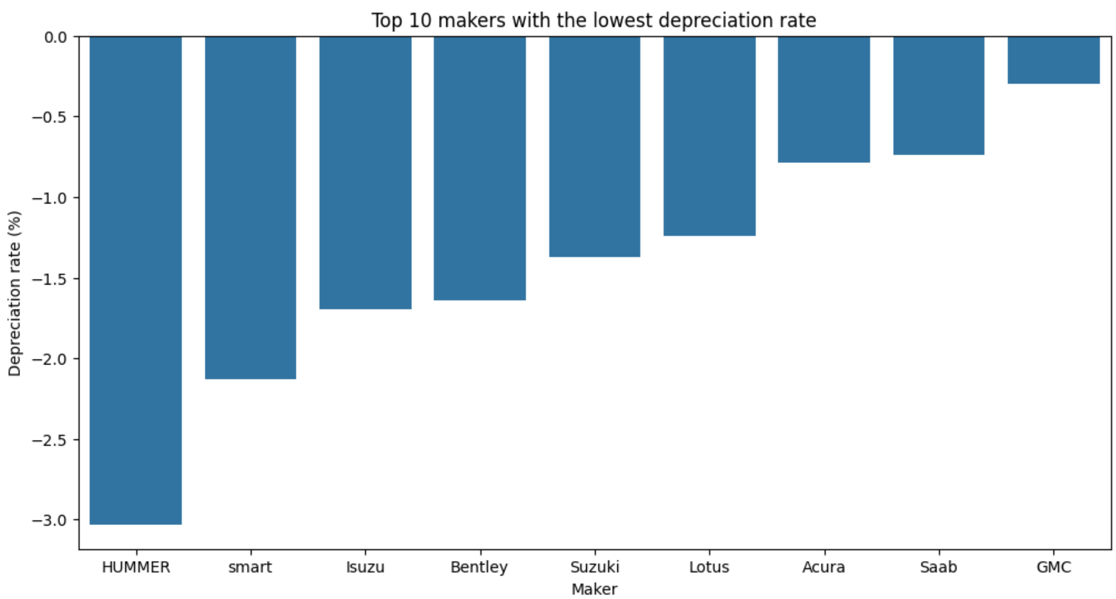


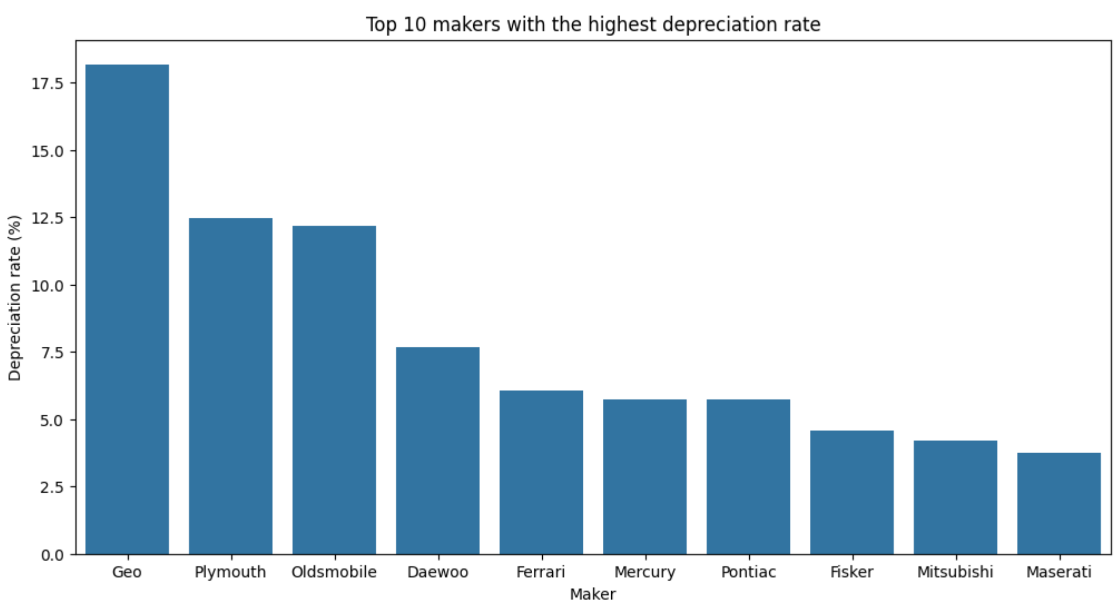
From the previous graphs is shown the depreciation percentages of the vehicles, based on the age of the vehicle, grouped by model maker and model age.

The depreciation is calculated as the difference between the selling price and the mmr value, divided by the mmr value. Since the depreciation is heavily influnced by the model age, I've clustered the data in 3 groups: 0-3 years, 3-6 years and 6-9 years.

Let's now check at the top 10 makers, with lowest and highest depreciation rates, without considering the model age.

[268]:
```
#let's now check at the top 10 makers, whit lowest and highest depreciation␣
↪rates, without considering the model age
df_depreciation = df_clean.groupby(['maker'])[['selling_price', 'mmr']].
↪median().reset_index()

df_depreciation['depreciation_pctg'] = round(((df_depreciation['mmr'] -␣
↪df_depreciation['selling_price']) / df_depreciation['mmr']) * 100, 3)

dfdepreiation_positive_head =␣
↪df_depreciation[df_depreciation['depreciation_pctg'] > 0].
↪sort_values('depreciation_pctg', ascending=False).head(x)

dfdepreiation_negative_head =␣
↪df_depreciation[df_depreciation['depreciation_pctg'] < 0].
↪sort_values('depreciation_pctg', ascending=True).head(x)

#let's graph the top x makers with the lowest and highest depreciation rate
plt.figure(figsize=(12,6))
sns.barplot(data=dfdepreiation_negative_head, x='maker', y='depreciation_pctg')
plt.title(f'Top {x} makers with the lowest depreciation rate')
plt.xlabel('Maker')
plt.ylabel('Depreciation rate (%)')
plt.xticks(rotation=0)

plt.figure(figsize=(12,6))
sns.barplot(data=dfdepreiation_positive_head, x='maker', y='depreciation_pctg')
plt.title(f'Top {x} makers with the highest depreciation rate')
plt.xlabel('Maker')
plt.ylabel('Depreciation rate (%)')
plt.xticks(rotation=0)
plt.show()
```





1.5 Answers and Conclusions

1. Market Trends:

- About a third of sales are vehicles that are 4 years or younger, indicating that newer vehicles (4 years or younger) dominate the used market.

- Sales of vehicles that are 2 years or younger account for about 50% of total sales, indicating that these vehicles generate the most revenue.

2. Customer Preferences:

- Most Popular Makes and Models: Ford is the top-selling brand with 4 models in the top 6. The top-selling model is the Ford Fusion.
- Vehicle Price and Condition: The correlation between vehicle condition and sales price is low, indicating that condition assessment does not have much impact on the final price.

3. Pricing Strategy:

- Vehicles that are 2 years or younger generate 75% of sales by value.
- The most profitable states for car sales are Tennessee, Colorado and Illinois, where median prices exceed the global average of $12,400.

4. Vehicle Depreciation:

- Vehicles between 0 and 3 years old show less loss of value than older vehicles.
- Some brands, such as BMW and Mercedes-Benz, tend to retain their value better, while other brands show higher rates of depreciation.