Huseyin Can Minareci - Noam Shmuel

June 10, 2020

0.1 Faculty of Economic Sciences University of Warsaw

Machine Learning 1: Classification Methods
Dr. Maciej Wilamowski

1 Used Car Price Prediction Analysis

Huseyin Can Minareci & Noam Shmuel

2 Introduction

In this project, we aim to find the best regression model for dataset in order to be able to predict used cars prices.
We have used public dataset from Kaggle which was scraped with Scrapy by Orges Leka from Ebay-Kleinanzeigen. It contains more than 370000 cars and 20 features of each of them like price, power, model etc…
Original dataset had plenty missing values, outliers and mistakes, to be able to work on it we had to clean, impute and change their format.
We select the features carefully and extract new features from raw data.
In order to achieve this goal we used Linear Model, K Nearest Neighbor and Random Forest.

3 Data Preparation

[1]:
```
# Importing the packages which we are going to use
import pandas as pd
import numpy as np
import math
import random
import time
import matplotlib
from math import sqrt
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets, linear_model, preprocessing, svm, neighbors,␣
↪metrics
from sklearn.preprocessing import StandardScaler, Normalizer, MinMaxScaler
from datetime import datetime
```

```
import statsmodels.api as sm
import statsmodels.formula.api as smf
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import KFold, train_test_split, GridSearchCV

%matplotlib inline
```

[2]:
```
# reading dataset
df = pd.read_csv('autos.csv', sep=',', header=0, encoding='cp1252')
df.sample(5)
```

[2]:
```
                dateCrawled                                          name  \
300022  2016-03-25 18:46:09            Opel_Astra_2.0_DTI_Caravan_Elegance
326791  2016-03-17 20:41:51                          Ford_Fiesta_1.4_Ghia
220614  2016-03-20 14:47:23  BMW_E46_318i_Touring_143PS_PDC_Klimaautomatik
224538  2016-03-23 09:55:40                            Ford_Focus_1.4_16V
157406  2016-03-31 15:53:13          Volkswagen_Touran_1.9_TDI_Comfortline

        seller offerType  price   abtest vehicleType  yearOfRegistration  \
300022  privat   Angebot   1690     test       kombi                2003
326791  privat   Angebot   5500  control  kleinwagen                2009
220614  privat   Angebot   3800     test       kombi                2004
224538  privat   Angebot   4499     test   limousine                2007
157406  privat   Angebot   4899  control         bus                2004

        gearbox  powerPS   model  kilometer  monthOfRegistration fuelType  \
300022  manuell      101   astra     150000                    6   diesel
326791  manuell       97  fiesta     125000                    3   benzin
220614  manuell      143     3er     150000                    1   benzin
224538  manuell       80   focus     100000                    8   benzin
157406  manuell      105  touran     150000                    8   diesel

             brand notRepairedDamage          dateCreated  nrOfPictures  \
300022        opel              nein  2016-03-25 00:00:00             0
326791        ford              nein  2016-03-17 00:00:00             0
220614         bmw              nein  2016-03-20 00:00:00             0
224538        ford              nein  2016-03-23 00:00:00             0
157406  volkswagen              nein  2016-03-31 00:00:00             0

        postalCode             lastSeen
300022       90602  2016-04-07 00:17:10
326791       41468  2016-04-07 09:15:50
220614       76131  2016-04-06 17:47:01
224538       22946  2016-04-07 10:16:30
```

 157406   67227 2016-04-06 08:46:23

The variables explained as follows: price : the price on the advert to sell the car, This is the dependent variable in all of the upcoming models.
dateCrawled : when advert was first crawled, all field-values are taken from this date
name : headline, which the owner of the car gave to the advert
seller : who is selling the car(private or dealer)
offerType : offer(car to sell) or request(car to buy)

abtest : ebay-intern variable (a/b testing)
vehicleType : one of eight vehicle-categories
yearOfRegistration : at which year the car was first registered
gearbox : manual or automatic
powerPS : the power of the car in PS

model : the car's model
kilometer : how many kilometres the car has driven
monthOfRegistration : at which month the car was first registered
fuelType : one of seven fuel-categories
brand : the car's brand

notRepairedDamage : if the car has a damage which is not repaired yet
dateCreated : the date for which the advert at 'ebay Kleinanzeigen' was created
nrOfPictures : number of pictures in the advert
postalCode : where in germany the car is located
lastSeenOnline : when the crawler saw this advert last online

3.0.1 Duplicates and outliers

Since data is crawled by scraper we may have some duplicates it will be better to check and drop them as first step.

[3]:
```python
# Removing the duplicates
dfl = df.drop_duplicates(['name','price','vehicleType','yearOfRegistration'
                                                      ␣
 ↪,'gearbox','powerPS','model','kilometer','monthOfRegistration','fuelType'
                         ,'notRepairedDamage'])
left = 100 * dfl['name'].count() / df['name'].count()
print("The amount of data left:", left, "%")
```

```
The amount of data left: 97.64270795202515 %
```

[4]:
```python
df.describe()
```

[4]:
```
                 price  yearOfRegistration        powerPS      kilometer  \
count  3.715280e+05       371528.000000  371528.000000  371528.000000
```

```
mean   1.729514e+04        2004.577997      115.549477  125618.688228
std    3.587954e+06          92.866598      192.139578   40112.337051
min    0.000000e+00        1000.000000        0.000000    5000.000000
25%    1.150000e+03        1999.000000       70.000000  125000.000000
50%    2.950000e+03        2003.000000      105.000000  150000.000000
75%    7.200000e+03        2008.000000      150.000000  150000.000000
max    2.147484e+09        9999.000000    20000.000000  150000.000000

       monthOfRegistration  nrOfPictures   postalCode
count        371528.000000      371528.0  371528.00000
mean              5.734445           0.0   50820.66764
std               3.712412           0.0   25799.08247
min               0.000000           0.0    1067.00000
25%               3.000000           0.0   30459.00000
50%               6.000000           0.0   49610.00000
75%               9.000000           0.0   71546.00000
max              12.000000           0.0   99998.00000
```

It can be clearly seen from summary statistics that there are outliers(yearOfRegistration, powerPS, price) in the data. We should remove them in order to continue.

[5]:
```python
print("Cars more expensive than 40000:", (dfl.price > 40000).sum())
print("Cars cheaper than 250:",(dfl.price < 250).sum())
print("Cars which has less than 10PS:" , (dfl.powerPS < 10).sum())
print("Cars which has more than 400PS:" , (dfl.powerPS > 400).sum())
print("Cars newer than 2017:" , (dfl.yearOfRegistration >= 2017).sum())
print("Cars older than 1990:" , (dfl.yearOfRegistration < 1990).sum())
```

```
Cars more expensive than 40000: 2692
Cars cheaper than 250: 19530
Cars which has less than 10PS: 39579
Cars which has more than 400PS: 1864
Cars newer than 2017: 14483
Cars older than 1990: 10495
```

There are only 2692 cars which are more expensive than 40000 since we have over 370000 cars we can easily call them as outliers and remove them from our dataset. Also we don't want to take into account the obvious mistakes like cars cheaper than 250.

Looks like we have a lot of cars which has 0 PS since something like that it is not possible we need to remove them from our dataset as well.

[6]:
```python
# Removing the outliers
dfl = dfl[
        (dfl.yearOfRegistration <= 2017)
      & (dfl.yearOfRegistration >= 1990)
      & (dfl.price >= 250)
      & (dfl.price <= 40000)
```

```
      & (dfl.powerPS >= 10)
      & (dfl.powerPS <= 400)]
left = 100 * dfl['name'].count() / df['name'].count()
print("The amount of data left:", left, "%")
```

```
The amount of data left: 79.97378394091427 %
```

We check again summary statistics to see if outliers removed from dataset succesfully.

[7]:
```
dfl.describe()
```

[7]:

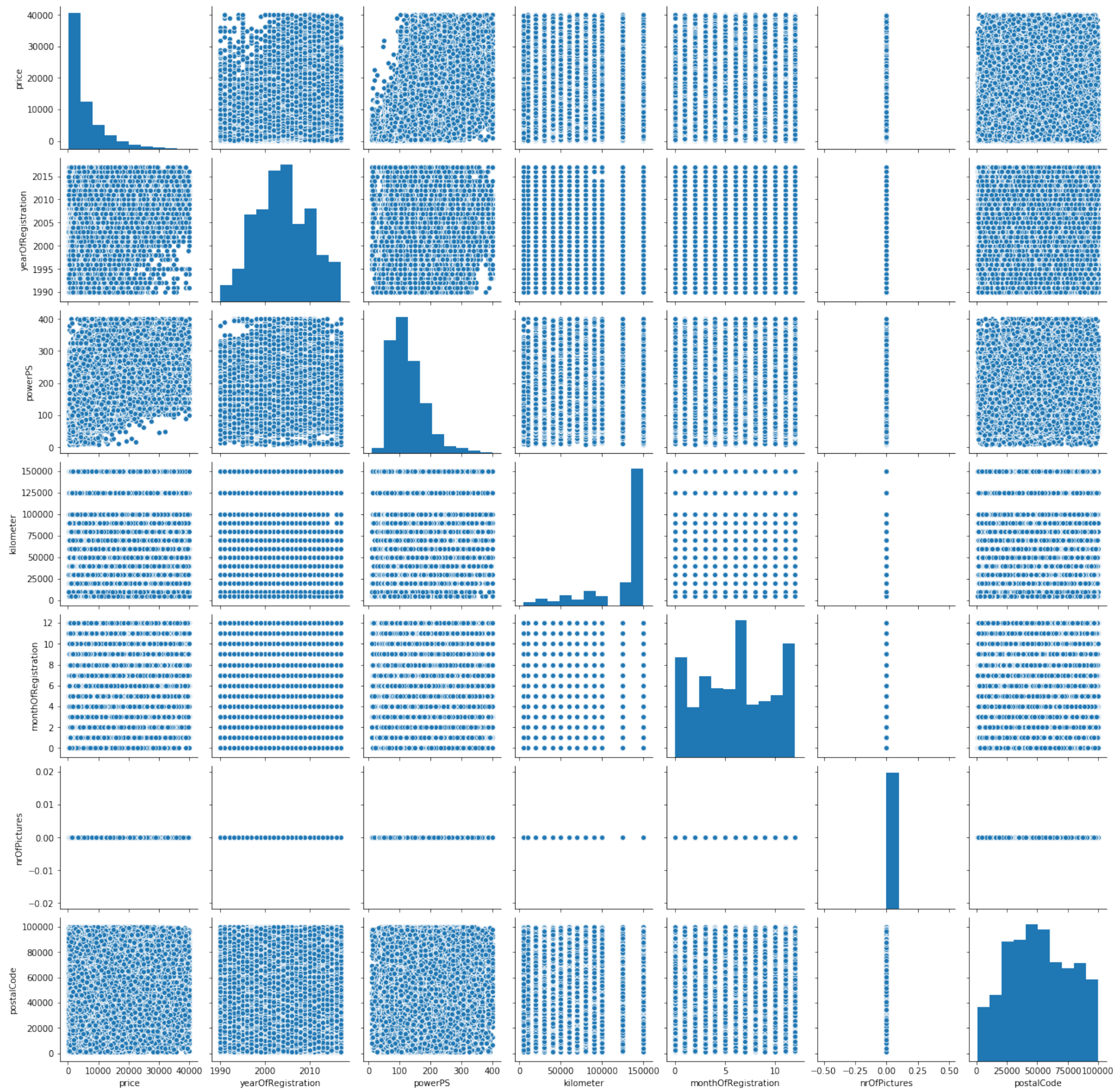
	price	yearOfRegistration	powerPS	kilometer
count	297125.000000	297125.000000	297125.000000	297125.000000
mean	5805.787638	2004.062714	124.808094	126377.063525
std	6364.402007	5.932930	55.794467	38409.786388
min	250.000000	1990.000000	10.000000	5000.000000
25%	1450.000000	2000.000000	82.000000	125000.000000
50%	3499.000000	2004.000000	116.000000	150000.000000
75%	7800.000000	2008.000000	150.000000	150000.000000
max	40000.000000	2017.000000	400.000000	150000.000000

	monthOfRegistration	nrOfPictures	postalCode
count	297125.000000	297125.0	297125.000000
mean	6.016505	0.0	51641.176182
std	3.577724	0.0	25686.994168
min	0.000000	0.0	1067.000000
25%	3.000000	0.0	31246.000000
50%	6.000000	0.0	50765.000000
75%	9.000000	0.0	72393.000000
max	12.000000	0.0	99998.000000

3.0.2 Exploratory Data Analysis

[8]:
```
sns.pairplot(dfl)
```

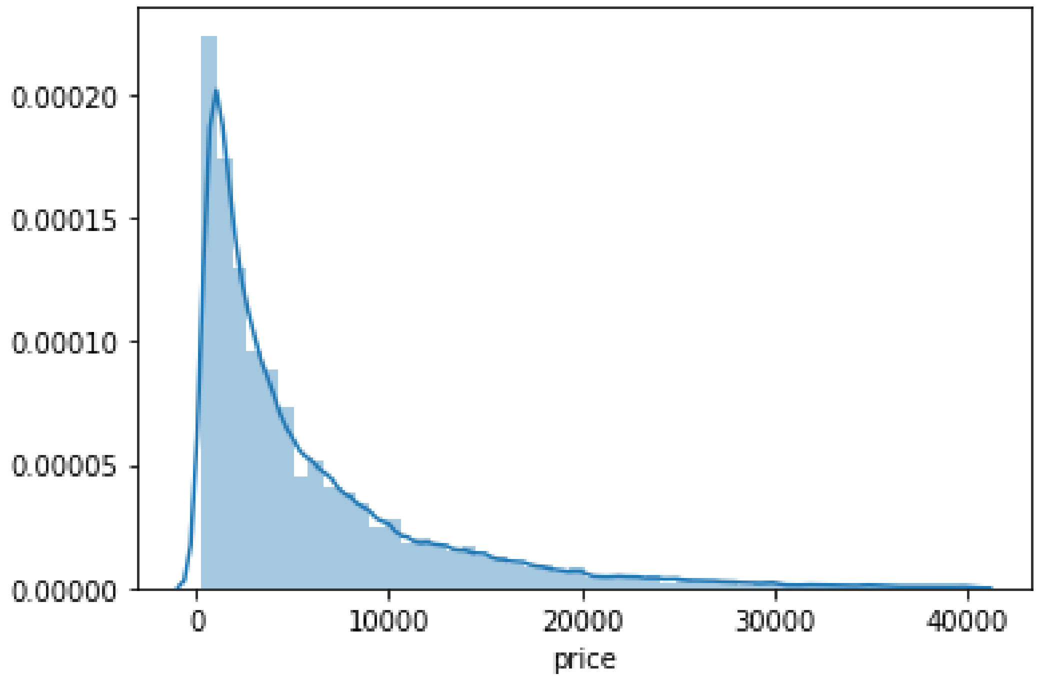
[8]:
```
<seaborn.axisgrid.PairGrid at 0x1ba0d39cc50>
```



From pairwise plots we can see how our variables looks like and go deeper to the necessary ones.

[9]:
```
sns.distplot(dfl["price"])
```

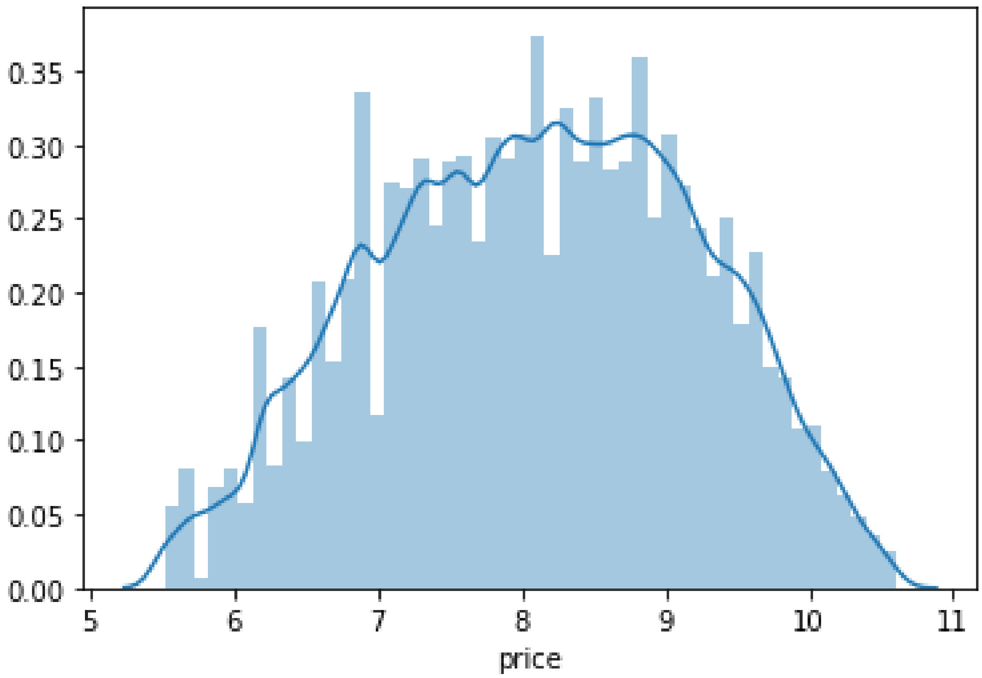
[9]: <matplotlib.axes._subplots.AxesSubplot at 0x1ba185b3128>



We have make a log transformation on price in in order to get rid of right skewness.

[10]:
```
dfl['priceNormal'] = dfl['price']
dfl['price'] = np.log(dfl['price'])
sns.distplot(dfl["price"])
```

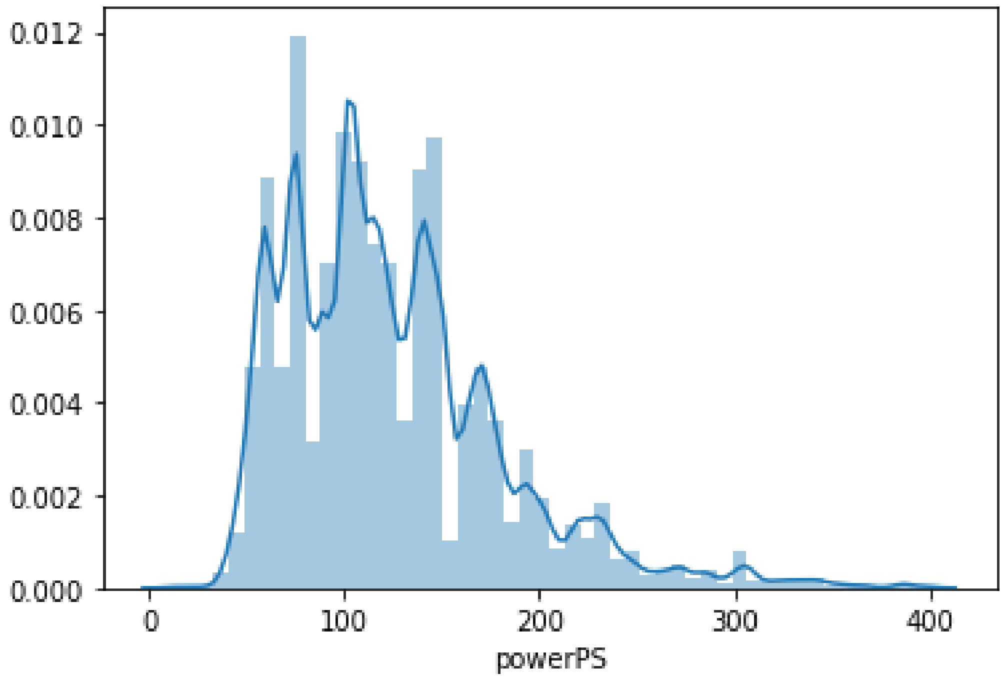
[10]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1ba186be4e0>
```



Since we did log transformation to our dependent variable we need to do also to the skewed independent variables

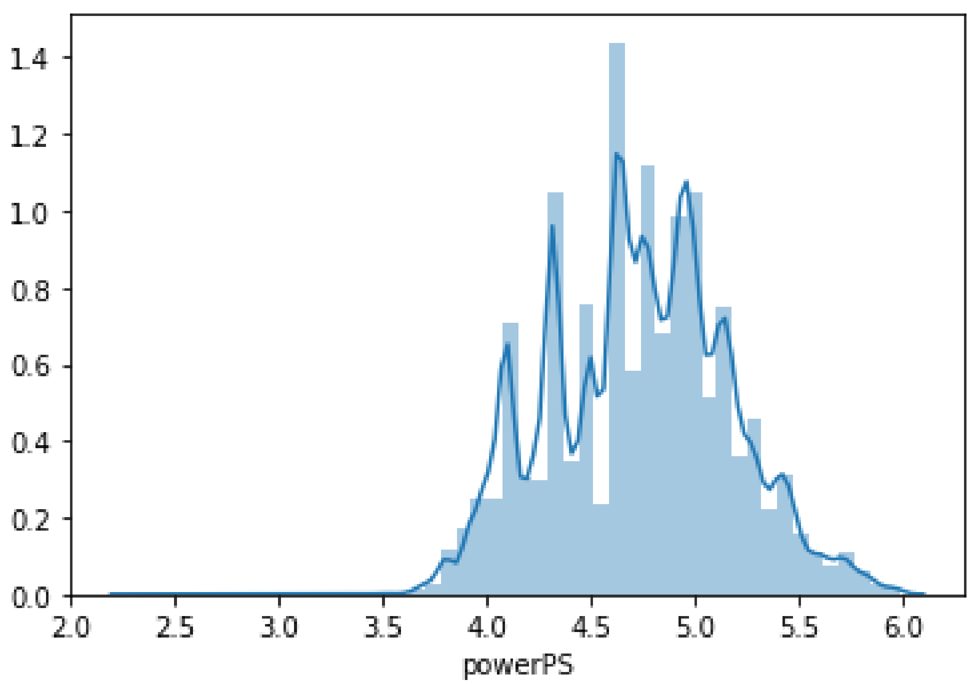
[11]:
```
sns.distplot(dfl["powerPS"])
```

[11]: `<matplotlib.axes._subplots.AxesSubplot at 0x1ba187a5390>`



[12]:
```
dfl['powerPS'] = np.log(dfl['powerPS'])
sns.distplot(dfl["powerPS"])
```

[12]: <matplotlib.axes._subplots.AxesSubplot at 0x1ba18885320>



3.1 Feature Engineering & Selection

After dealing with duplicates, outliers and transformation, we moved to the next part; Feature Engineering.

3.1.1 daysBeforeSold

[13]:
```
dfl.loc[:,("dateCrawled", "dateCreated", "postalCode", "lastSeen")]
```

[13]:
```
                dateCrawled          dateCreated  postalCode  \
1       2016-03-24 10:58:45  2016-03-24 00:00:00       66954
2       2016-03-14 12:52:21  2016-03-14 00:00:00       90480
3       2016-03-17 16:54:04  2016-03-17 00:00:00       91074
4       2016-03-31 17:25:20  2016-03-31 00:00:00       60437
5       2016-04-04 17:36:23  2016-04-04 00:00:00       33775
…                        …                    …          …
371520  2016-03-19 19:53:49  2016-03-19 00:00:00       96465
371524  2016-03-05 19:56:21  2016-03-05 00:00:00       26135
371525  2016-03-19 18:57:12  2016-03-19 00:00:00       87439
371526  2016-03-20 19:41:08  2016-03-20 00:00:00       40764
371527  2016-03-07 19:39:19  2016-03-07 00:00:00       73326

                   lastSeen
1       2016-04-07 01:46:50
2       2016-04-05 12:47:46
3       2016-03-17 17:40:17
4       2016-04-06 10:17:21
5       2016-04-06 19:17:07
…                        …
371520  2016-03-19 20:44:43
371524  2016-03-11 18:17:12
371525  2016-04-07 07:15:26
371526  2016-03-24 12:45:21
371527  2016-03-22 03:17:10

[297125 rows x 4 columns]
```

We don't think "dateCrawled" and "postalCode" would be useful for price prediction so we decided to remove them.
Although "dateCreated" and "lastSeen" could be useful if can be modified as how many days advert was on the site before it sold.(or removed)

[14]:
```
dfl = dfl.drop(["dateCrawled", "postalCode"], axis='columns')
```

In the dataset dateCreated and lastSeen is in string format. In order get the days we choosed to change the format to timestamp.

Since we can use substraction in timestamp, we substract dateCreated from lastSeen.

And finally take just take the day since we don't need hours minutes and seconds in our models.

[15]:
```
# creating funtion to change string to timestamp
def changeformat(date):
    datefr = datetime.strptime(date, '%Y-%m-%d %H:%M:%S')
    return datefr
```

[16]:
```
dfl["dateCreated"] = dfl["dateCreated"].apply(changeformat)
dfl["lastSeen"] = dfl["lastSeen"].apply(changeformat)
dfl["diff"] = dfl["lastSeen"]-dfl["dateCreated"]
dfl["daysBeforeSold"] = dfl["diff"].apply((lambda x: x.days))
```

[17]:
```
dfl.loc[:,("dateCreated", "lastSeen", "diff", "daysBeforeSold")]
```

[17]:
```
        dateCreated            lastSeen            diff  daysBeforeSold
1        2016-03-24 2016-04-07 01:46:50 14 days 01:46:50              14
2        2016-03-14 2016-04-05 12:47:46 22 days 12:47:46              22
3        2016-03-17 2016-03-17 17:40:17  0 days 17:40:17               0
4        2016-03-31 2016-04-06 10:17:21  6 days 10:17:21               6
5        2016-04-04 2016-04-06 19:17:07  2 days 19:17:07               2
…            …                  …                  …             …
371520   2016-03-19 2016-03-19 20:44:43  0 days 20:44:43               0
371524   2016-03-05 2016-03-11 18:17:12  6 days 18:17:12               6
371525   2016-03-19 2016-04-07 07:15:26 19 days 07:15:26              19
371526   2016-03-20 2016-03-24 12:45:21  4 days 12:45:21               4
371527   2016-03-07 2016-03-22 03:17:10 15 days 03:17:10              15

[297125 rows x 4 columns]
```

After succesfully getting the day difference in integer format we removed "dateCreated", "lastSeen" and "diff" from the data.

[18]:
```
dfl = dfl.drop(["dateCreated", "lastSeen", "diff"], axis='columns')
```

3.1.2 namelen

[19]:
```
dfl["name"]
```

[19]:
```
1                                       A5_Sportback_2.7_Tdi
2                                 Jeep_Grand_Cherokee_"Overland"
3                                          GOLF_4_1_4__3TÜRER
4                                 Skoda_Fabia_1.4_TDI_PD_Classic
5          BMW_316i___e36_Limousine___Bastlerfahrzeug__Ex…
                                    …
371520                                          turbo_defekt
```

```
371524                Smart_smart_leistungssteigerung_100ps
371525                   Volkswagen_Multivan_T4_TDI_7DC_UY2
371526                               VW_Golf_Kombi_1_9l_TDI
371527         BMW_M135i_vollausgestattet_NP_52.720____Euro
Name: name, Length: 297125, dtype: object
```

Names of adverts are not usefull how they are right now we want to try to get something from this.

Taking their length and checking if it would be helpful in our prediction sounded like it worths to give it a try.

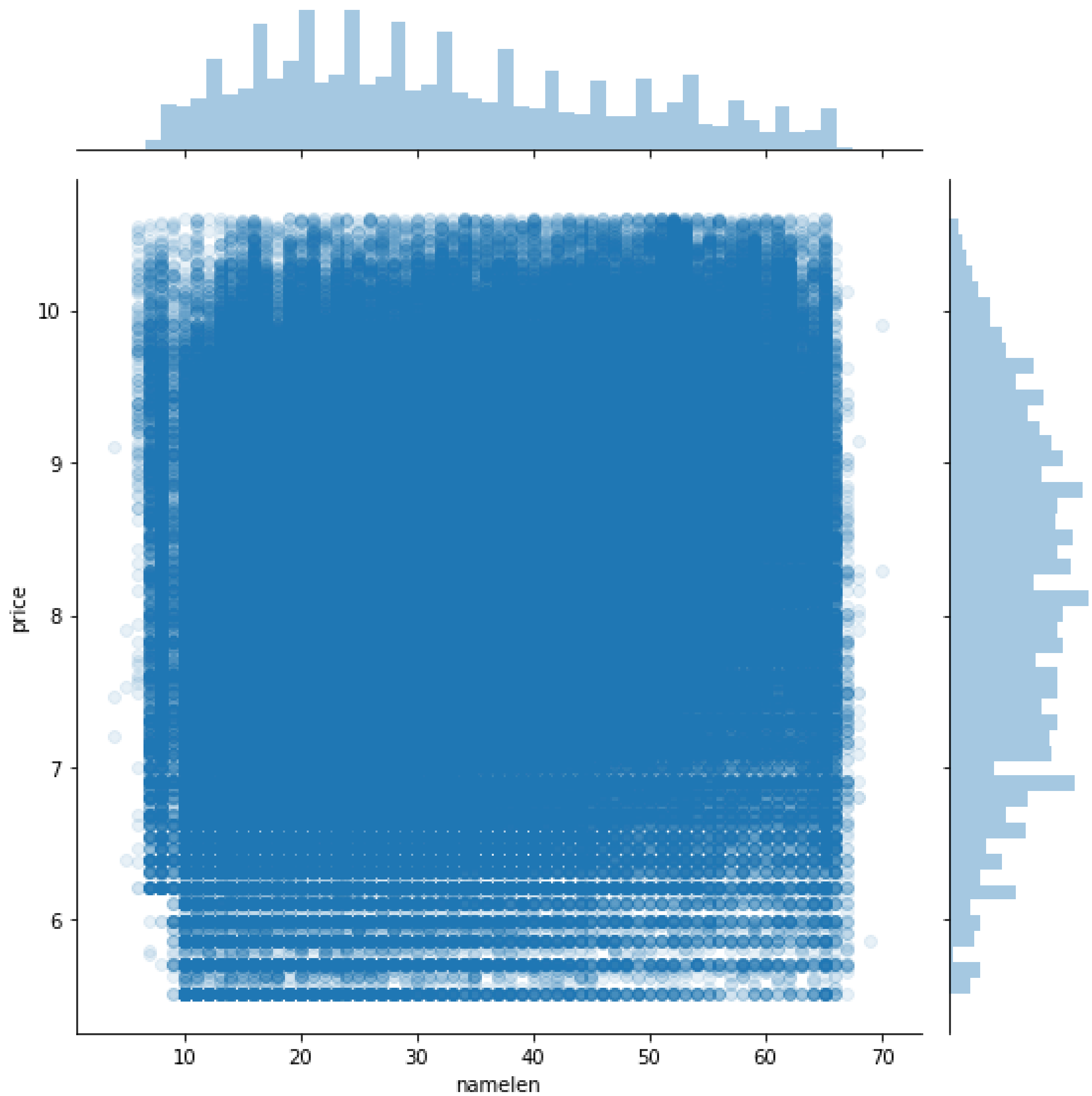
[20]:
```
dfl['namelen'] = [min(70, len(n)) for n in dfl['name']]
```

[21]:
```
dfl.loc[:,'namelen']
```

[21]:
```
1         20
2         30
3         18
4         30
5         50
          ..
371520    12
371524    37
371525    34
371526    22
371527    44
Name: namelen, Length: 297125, dtype: int64
```

[22]:
```
sns.jointplot(x='namelen',
                   y='price',
                   data=dfl[['namelen','price']],
                    alpha=0.1,
                    height=8)
```

[22]:
```
<seaborn.axisgrid.JointGrid at 0x1ba1ae40b38>
```

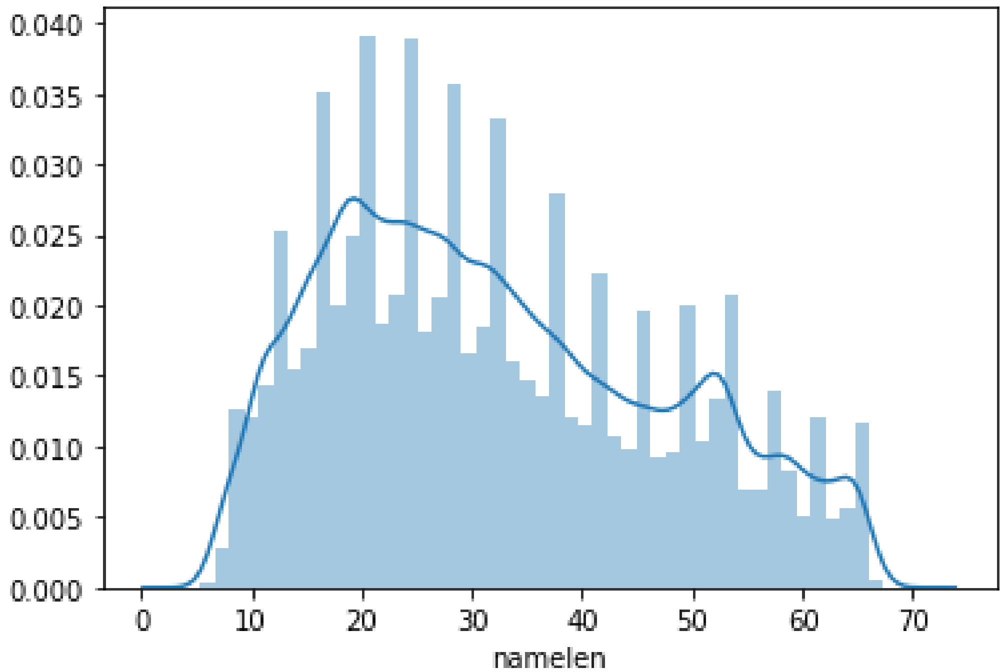


It seems that a name length can help us to predict the price since it looks like there is some connection between them.

Longer the name is more expensive the car, it can be because the cars with additional features would take longer to describe them. From other side shorter the explanation can lead us to think that car doesn't have much to write about that's why it's cheaper.

[23]:
```
sns.distplot(dfl.namelen)
```

[23]: <matplotlib.axes._subplots.AxesSubplot at 0x1ba1ae406d8>



We droped the name column since we took length of it.

[24]:
```
dfl = dfl.drop(["name"], axis='columns')
```

3.1.3 Age

Instead of dealing with years we create “age” of the car by using yearOfRegistration and simply substacting it from 2017(the year data collected)

[25]:
```
dfl["age"] = dfl["yearOfRegistration"].apply((lambda x: max(0, 2017-x)))
```

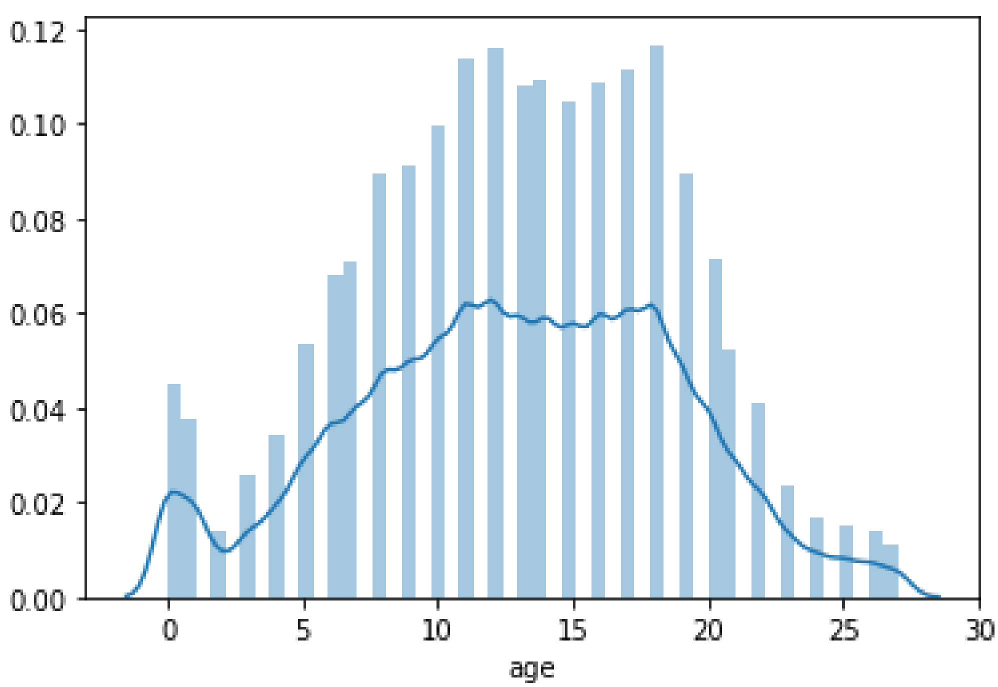
[26]:
```
dfl.describe()
```

[26]:
```
                price  yearOfRegistration        powerPS      kilometer  \
count  297125.000000       297125.000000  297125.000000  297125.000000
mean        8.107775         2004.062714       4.733331  126377.063525
std         1.110681            5.932930       0.433482   38409.786388
min         5.521461         1990.000000       2.302585    5000.000000
25%         7.279319         2000.000000       4.406719  125000.000000
50%         8.160232         2004.000000       4.753590  150000.000000
75%         8.961879         2008.000000       5.010635  150000.000000
max        10.596635         2017.000000       5.991465  150000.000000

       monthOfRegistration  nrOfPictures    priceNormal  daysBeforeSold  \
count        297125.000000      297125.0  297125.000000   297125.000000
```

```
mean             6.016505           0.0    5805.787638         9.059598
std              3.577724           0.0    6364.402007         8.629988
min              0.000000           0.0     250.000000         0.000000
25%              3.000000           0.0    1450.000000         2.000000
50%              6.000000           0.0    3499.000000         6.000000
75%              9.000000           0.0    7800.000000        14.000000
max             12.000000           0.0   40000.000000       384.000000

             namelen            age
count  297125.000000  297125.000000
mean       32.280074      12.937286
std        15.048183       5.932930
min         4.000000       0.000000
25%        20.000000       9.000000
50%        30.000000      13.000000
75%        43.000000      17.000000
max        70.000000      27.000000
```

[27]:
```
sns.distplot(dfl["age"])
```

[27]: <matplotlib.axes._subplots.AxesSubplot at 0x1ba23b14da0>



Since we have age of the car we drop yearOfRegistration and monthOfRegistration.

[28]:
```
dfl = dfl.drop(["yearOfRegistration", "monthOfRegistration"], axis='columns')
```

3.1.4 gearbox abtest notRepairedDamage

[29]:
```
print(dfl.gearbox.value_counts())
print(dfl.abtest.value_counts())
print(dfl.notRepairedDamage.value_counts())
```

```
manuell      227724
automatik     64153
Name: gearbox, dtype: int64
test       153806
control    143319
Name: abtest, dtype: int64
nein    228640
ja       26457
Name: notRepairedDamage, dtype: int64
```

We decided to keep gearbox and notRepairedDamage since they differ among the dataset and could be useful for model
Although abtest is A/B testing(user experience research) of eBay, so we decided to drop it.

[30]:
```
dfl = dfl.drop(["abtest"], axis='columns')
```

3.1.5 offerType nrOfPictures seller

[31]:
```
print(dfl.offerType.value_counts())
print(dfl.nrOfPictures.value_counts())
print(dfl.seller.value_counts())
```

```
Angebot    297123
Gesuch          2
Name: offerType, dtype: int64
0    297125
Name: nrOfPictures, dtype: int64
privat        297124
gewerblich         1
Name: seller, dtype: int64
```

Since there are just 2 unique values and 1-2 value in second ones in seller and offerType we decided to drop them.

If it comes to nrOfPictures looks like there was a problem with scrawler and we have only zeros. That's why we have to drop them since it will not help to us how it is right now and there is no way to predict it at the moment.

[32]:
```
dfl = dfl.drop(["offerType", "nrOfPictures", "seller"], axis='columns')
```

[33]:
```
dfl.head(10)
```

[33]:
```
       price vehicleType    gearbox   powerPS    model  kilometer fuelType  \
1   9.814656       coupe    manuell  5.247024      NaN     125000   diesel
2   9.190138         suv  automatik  5.093750    grand     125000   diesel
3   7.313220  kleinwagen    manuell  4.317488     golf     150000   benzin
4   8.188689  kleinwagen    manuell  4.234107    fabia      90000   diesel
5   6.476972   limousine    manuell  4.624973      3er     150000   benzin
6   7.696213      cabrio    manuell  4.691348  2_reihe     150000   benzin
8   9.581904         bus    manuell  4.828314    c_max      30000   benzin
9   6.906755  kleinwagen    manuell  4.615121     golf     150000      NaN
10  7.600902   limousine    manuell  4.653960  3_reihe     150000   benzin
11  7.937017       kombi    manuell  4.941642   passat     150000   diesel

         brand notRepairedDamage  priceNormal  daysBeforeSold  namelen  age
1         audi                ja        18300              14       20    6
2         jeep               NaN         9800              22       30   13
3   volkswagen              nein         1500               0       18   16
4        skoda              nein         3600               6       30    9
5          bmw                ja          650               2       50   22
6      peugeot              nein         2200               4       27   13
8         ford               NaN        14500               0       36    3
9   volkswagen               NaN          999              14       53   19
10       mazda              nein         2000              11       17   13
11  volkswagen                ja         2799               0       45   12
```

3.1.6 Translating data from German to English

Since we finished feature engineering and selection before moving to dealing with missing values we translate data from German to English.

[34]:
```
## translate data from german to english
dfl['gearbox'] = dfl['gearbox'].replace(to_replace=['automatik', 'manuell'],␣
 ↪value=['automatic', 'manual'], inplace=False, limit=None)
dfl['fuelType'] = dfl['fuelType'].replace(to_replace=['andere', 'benzin',␣
 ↪'elektro'], value=['others', 'petrol', 'electric'], inplace=False,␣
 ↪limit=None)
dfl['vehicleType'] = dfl['vehicleType'].replace(to_replace=['andere', 'kombi',␣
 ↪'kleinwagen'], value=['others', 'station wagon', 'small car'],␣
 ↪inplace=False, limit=None)
dfl['notRepairedDamage'] = dfl['notRepairedDamage'].replace(to_replace=['ja',␣
 ↪'nein'], value=['yes', 'no'], inplace=False, limit=None)
dfl['brand'] = dfl['brand'].replace(to_replace = ["sonstige_autos"],␣
 ↪value=["other"], inplace=False, limit=None)
```

3.2 Missing values

We have removed problematic values like duplicates and outliers, after that we have chosed the features we are going to work with and drop the rest. Now we are going to deal with missing values in the dataset.

[35]:
```
dfdpna = dfl.copy()
dfdpna = dfdpna.dropna()

missing = 100 - 100 * dfdpna['price'].count() / dfl['price'].count()
print("Missing values:", missing, "%")
```

```
Missing values: 21.833235170382835 %
```

It looks like around 22% of the data has missing values.

3.3 Imputation

We decided to investigate every column with missing values one by one and see how we can deal with them.

[36]:
```
dfim = dfl.copy()
```

[37]:
```
dfim.isnull().sum()
```

[37]:
```
price                    0
vehicleType          17107
gearbox               5248
powerPS                  0
model                10686
kilometer                0
fuelType             16998
brand                    0
notRepairedDamage    42028
priceNormal              0
daysBeforeSold           0
namelen                  0
age                      0
dtype: int64
```

3.3.1 vehicleType

vehicleType has 17107 missing values.

This column there are 8 categorical values and their distribution looks like as it follows.

[38]:
```
dfim['vehicleType'].value_counts()
```

[38]:
```
limousine        80872
small car        65554
station wagon    58854
bus              26371
cabrio           19283
coupe            14652
```

```
suv               12267
others             2165
Name: vehicleType, dtype: int64
```

We checked the mean of every group in order to do some meaningful imputation.

[39]:
```
dfim.groupby(['vehicleType']).priceNormal.mean()
```

[39]:
```
vehicleType
bus                6713.635547
cabrio             8792.936213
coupe              8559.776754
limousine          5987.652265
others             4219.128868
small car          3047.271867
station wagon      5968.634859
suv               12458.418032
Name: priceNormal, dtype: float64
```

After seeing difference in the average prices we decided to create a function for filling missing values in vehicleType by using means

[40]:
```
def fillnavt(x):
    if type(x) == str:
        return x
    if x >= 12000:
        return "suv"
    if x < 12000 and x >= 10500:
        return "coupe"
    if x < 10500 and x >= 9000:
        return "cabrio"
    if x < 9000 and x >= 6700:
        return "bus"
    if x < 6700 and x >= 6100:
        return "limousine"
    if x < 6100 and x >= 4500:
        return "station wagon"
    if x < 4500:
        return "small car"
```

[41]:
```
dfim['Col2'] = dfim.priceNormal.apply(lambda x: fillnavt(x))
```

[42]:
```
dfim['vehicleType'] = dfim['vehicleType'].combine_first(dfim['Col2'])
```

[43]:
```
dfim['vehicleType'].value_counts()
```

[43]:
```
limousine        81167
small car        78797
station wagon    60130
bus              27313
cabrio           19600
coupe            14913
suv              13040
others            2165
Name: vehicleType, dtype: int64
```

3.3.2 gearbox

We decided to drop gearbox since it's hard to predict the car's gear.

[44]:
```python
dfim = dfim[-dfim['gearbox'].isnull()]
```

[45]:
```python
dfim['model'].unique().size
```

[45]:
```
248
```

3.3.3 Model

[46]:
```python
dfim.groupby(['brand','model']).priceNormal.mean()
```

[46]:
```
brand       model
alfa_romeo  145            1103.093750
            147            2366.404472
            156            1661.785855
            159            7318.851282
            andere         6192.350120
                              …
volvo       v40            1912.220257
            v50            6066.584746
            v60           16028.181818
            v70            4626.147887
            xc_reihe      15487.252595
Name: priceNormal, Length: 294, dtype: float64
```

Since there is 251 model and price differ a lot between models, it is very hard to replace the missings in a sensible way.
To not cause bias we decided to drop them.

[47]:
```python
dfim = dfim[-dfim['model'].isnull()]
```

[48]:
```python
dfim.isnull().sum()
```

[48]:
```
price                     0
vehicleType               0
gearbox                   0
powerPS                   0
model                     0
kilometer                 0
fuelType              13217
brand                     0
notRepairedDamage     36471
priceNormal               0
daysBeforeSold            0
namelen                   0
age                       0
Col2                      0
dtype: int64
```

3.3.4 fuelType

There are 13217 missing values in the fuelType. We want to fill those missing values in a sensible way and in order to do so we check avarage prices and distributions of it.

[49]:
```
dfim['fuelType'].value_counts()
```

[49]:
```
petrol      173508
diesel       90085
lpg           4279
cng            467
hybrid         201
electric        53
others          43
Name: fuelType, dtype: int64
```

[50]:
```
dfim.groupby(['fuelType']).priceNormal.mean()
```

[50]:
```
fuelType
cng          4971.732334
diesel       8708.634967
electric     9604.188679
hybrid      11885.194030
lpg          4301.146997
others       4184.534884
petrol       4697.426032
Name: priceNormal, dtype: float64
```

Looks like fuelType is mostly petrol and after that diesel. Also diesel cars are more expensive than cars with petrol.

We decided to write the following function in order to guess the fuelType according to price.

[51]:
```python
def fillft(x):
    if x >= 6000:
        return "diesel"
    if x < 6000:
        return "petrol"
```

[52]:
```python
dfim['Col3'] = dfim.priceNormal.apply(lambda x: fillft(x))

dfim['fuelType'] = dfim['fuelType'].combine_first(dfim['Col3'])

dfim['fuelType'].value_counts()
```

[52]:
```
petrol      184759
diesel       92051
lpg           4279
cng            467
hybrid         201
electric        53
others          43
Name: fuelType, dtype: int64
```

[53]:
```python
dfim.groupby(['fuelType']).priceNormal.mean()
```

[53]:
```
fuelType
cng          4971.732334
diesel       8758.919338
electric     9604.188679
hybrid      11885.194030
lpg          4301.146997
others       4184.534884
petrol       4522.545733
Name: priceNormal, dtype: float64
```

3.3.5 notRepairedDamage

Missing value in notRepairedDamage highly possible means that, car doesn't have a damage which needs to be repaired

So we decided to fill them with "No"

[54]:
```python
dfim['notRepairedDamage'].value_counts()
```

[54]:
```
no     220650
yes     24732
Name: notRepairedDamage, dtype: int64
```

[55]:
```python
dfim['notRepairedDamage'].fillna(value='no', inplace=True)
```

[56]:
```
dfim['notRepairedDamage'].value_counts()
```

[56]:
```
no     257121
yes     24732
Name: notRepairedDamage, dtype: int64
```

[57]:
```
dfim = dfim.drop(["Col2", "Col3"], axis='columns')
```

[58]:
```
dfim.sample(10)
```

[58]:
```
           price vehicleType    gearbox   powerPS    model  kilometer  \
241901  7.313220   limousine     manual  4.836282  6_reihe     150000
292850  9.433484   limousine     manual  4.465908     golf      40000
61419   7.801391       coupe     manual  5.262690      clk     150000
107362  8.902456   limousine     manual  5.318120      3er     150000
172310  6.318968   small car     manual  4.094345    corsa     150000
294588  8.174703   limousine     manual  4.290459       c3     125000
345536  8.006368   limousine     manual  4.744932      3er     150000
162853  7.762171   small car     manual  4.912655  3_reihe     150000
298023  6.801283   limousine     manual  4.499810  1_reihe     150000
281755  9.200290   small car  automatic  4.653960     golf      30000

       fuelType          brand notRepairedDamage  priceNormal  daysBeforeSold  \
241901   petrol          mazda               yes         1500              15
292850   petrol     volkswagen                no        12500               3
61419    petrol  mercedes_benz                no         2444              21
107362   diesel            bmw                no         7350               9
172310   petrol           opel                no          555               4
294588   petrol        citroen                no         3550              31
345536   petrol            bmw                no         3000               0
162853   petrol        peugeot                no         2350              20
298023   petrol          mazda                no          899               3
281755   petrol     volkswagen                no         9900               4

        namelen  age
241901       26   16
292850       48    4
61419        23   19
107362       16   14
172310       16   21
294588       19   14
345536       12   17
162853       35    0
298023       27   20
281755       18    6
```

[59]:
```
dfim.isnull().sum()
```

[59]:
```
price                0
vehicleType          0
gearbox              0
powerPS              0
model                0
kilometer            0
fuelType             0
brand                0
notRepairedDamage    0
priceNormal          0
daysBeforeSold       0
namelen              0
age                  0
dtype: int64
```

Since we impute or remove all the missing values successfully we don't need priceNormal in our dataset anymore.

[60]:
```python
priceNormal = dfim.priceNormal
dfim = dfim.drop(["priceNormal"], axis='columns')
```

Original dataset with ouitliers with nas

[61]:
```python
df.price.size
```

[61]: 371528

Dataset no ouitliers with nas

[62]:
```python
dfl.price.size
```

[62]: 297125

Original dataset no ouitliers with imputation of nas

[63]:
```python
dfim.price.size
```

[63]: 281853

Original dataset no ouitliers no nas (drop)

[64]:
```python
dfdpna.price.size
```

[64]: 232253

Looks like thanks to imputation we saved 49600 rows(13% of the data)

We will see how it will effect our models

[65]:
```
print("Outliers and duplicates(dropped):", 100 - left, "%")
missingdp = 100 - 100 * dfim['price'].count() / dfl['price'].count()
print("Missing values(dropped):", missingdp, "%")

print("Missing values(imputed):", missing - missingdp, "%")

leftraw2 = 100 * dfim['price'].count() / df['price'].count()
print("The amount of data left(from raw data):", leftraw2, "%")
```

```
Outliers and duplicates(dropped): 20.026216059085726 %
Missing values(dropped): 5.139924274295325 %
Missing values(imputed): 16.69331089608751 %
The amount of data left(from raw data): 75.86319200706272 %
```

3.3.6 Converting string into integers

We need to: * create ordered numerical labels for notRepairedDamage * do one-hot encoding (binarization) for vehicleType gearbox model fuelType and brand

[66]:
```
dfim.dtypes
```

[66]:
```
price                float64
vehicleType           object
gearbox               object
powerPS              float64
model                 object
kilometer              int64
fuelType              object
brand                 object
notRepairedDamage     object
daysBeforeSold         int64
namelen                int64
age                    int64
dtype: object
```

[67]:
```
dfim.head(10)
```

[67]:
```
       price    vehicleType    gearbox   powerPS    model  kilometer fuelType  \
2   9.190138            suv  automatic  5.093750    grand     125000   diesel
3   7.313220      small car     manual  4.317488     golf     150000   petrol
4   8.188689      small car     manual  4.234107    fabia      90000   diesel
5   6.476972      limousine     manual  4.624973      3er     150000   petrol
6   7.696213         cabrio     manual  4.691348  2_reihe     150000   petrol
8   9.581904            bus     manual  4.828314    c_max      30000   petrol
9   6.906755      small car     manual  4.615121     golf     150000   petrol
10  7.600902      limousine     manual  4.653960  3_reihe     150000   petrol
11  7.937017  station wagon     manual  4.941642   passat     150000   diesel
12  6.906755  station wagon     manual  4.744932   passat     150000   petrol
```

```
         brand notRepairedDamage  daysBeforeSold  namelen  age
2         jeep                no              22       30   13
3   volkswagen                no               0       18   16
4        skoda                no               6       30    9
5          bmw               yes               2       50   22
6      peugeot                no               4       27   13
8         ford                no               0       36    3
9   volkswagen                no              14       53   19
10       mazda                no              11       17   13
11  volkswagen               yes               0       45   12
12  volkswagen                no              17       33   22
```

We create ordered numerical labels for notRepairedDamage.

[68]:
```python
nrdDict = {'no':0, "yes":1}
dfim["notRepairedDamage"]=dfim.notRepairedDamage.map(nrdDict)
dfdpna["notRepairedDamage"]=dfdpna.notRepairedDamage.map(nrdDict)
```

We did one-hot encoding for all variables with nominal levels to binary form.

[69]:
```python
levCols = []
numCols = []
for col in dfim.columns:
    if dfim[col].dtype==object:
        levCols.append(col)
    else:
        numCols.append(col)
```

[70]:
```python
dummimint = pd.get_dummies(dfim[levCols])
a = dfim.drop(['vehicleType', 'gearbox', 'model', 'fuelType', 'brand'],␣
 ↪axis='columns')
dfimdm = pd.concat([a, dummimint], axis=1, sort=False)

dummdpnaint = pd.get_dummies(dfdpna[levCols])
b = dfdpna.drop(['vehicleType', 'gearbox', 'model', 'fuelType', 'brand'],␣
 ↪axis='columns')
dfdpnadm = pd.concat([b, dummdpnaint], axis=1, sort=False)
```

Since model has 251 unique variable and it is not possible order them, after doing the one-hot encoding we end up with 310 columns.

[71]:
```python
dfimdm.head()
```

[71]:
```
      price   powerPS  kilometer  notRepairedDamage  daysBeforeSold  namelen  \
2  9.190138  5.093750     125000                  0              22       30
3  7.313220  4.317488     150000                  0               0       18
4  8.188689  4.234107      90000                  0               6       30
```

```
5  6.476972  4.624973     150000                  1              2       50
6  7.696213  4.691348     150000                  0              4       27

   age  vehicleType_bus  vehicleType_cabrio  vehicleType_coupe  …  \
2   13                0                   0                  0  …
3   16                0                   0                  0  …
4    9                0                   0                  0  …
5   22                0                   0                  0  …
6   13                0                   1                  0  …

   brand_saab  brand_seat  brand_skoda  brand_smart  brand_subaru  \
2           0           0            0            0             0
3           0           0            0            0             0
4           0           0            1            0             0
5           0           0            0            0             0
6           0           0            0            0             0

   brand_suzuki  brand_toyota  brand_trabant  brand_volkswagen  brand_volvo
2             0             0              0                 0            0
3             0             0              0                 1            0
4             0             0              0                 0            0
5             0             0              0                 0            0
6             0             0              0                 0            0

[5 rows x 310 columns]
```

Same steps but without model in order to see the effects.

[72]:
```python
dfimNoModel = dfim.copy()
dfimNoModel = dfimNoModel.drop(['model'], axis = 'columns')

dfimNoModeldm = pd.get_dummies(dfim[['vehicleType', 'gearbox', 'fuelType',␣
 ↪'brand']])

a = dfimNoModel.drop(['vehicleType', 'gearbox', 'fuelType', 'brand'],␣
 ↪axis='columns')

dmNoModel = pd.concat([a, dfimNoModeldm], axis=1, sort=False)
```

After dropping 'model' from our data and doing one-hot encoding we end up with 63 columns. We are thinking that it might be easier to work with especially with KNN method.

[73]:
```python
dmNoModel.head()
```

[73]:
```
      price   powerPS  kilometer  notRepairedDamage  daysBeforeSold  namelen  \
2  9.190138  5.093750     125000                  0              22       30
3  7.313220  4.317488     150000                  0               0       18
```

```
4  8.188689  4.234107     90000                 0               6       30
5  6.476972  4.624973    150000                 1               2       50
6  7.696213  4.691348    150000                 0               4       27

   age  vehicleType_bus  vehicleType_cabrio  vehicleType_coupe  …  \
2   13                0                   0                  0  …
3   16                0                   0                  0  …
4    9                0                   0                  0  …
5   22                0                   0                  0  …
6   13                0                   1                  0  …

   brand_saab  brand_seat  brand_skoda  brand_smart  brand_subaru  \
2           0           0            0            0             0
3           0           0            0            0             0
4           0           0            1            0             0
5           0           0            0            0             0
6           0           0            0            0             0

   brand_suzuki  brand_toyota  brand_trabant  brand_volkswagen  brand_volvo
2             0             0              0                 0            0
3             0             0              0                 1            0
4             0             0              0                 0            0
5             0             0              0                 0            0
6             0             0              0                 0            0

[5 rows x 63 columns]
```

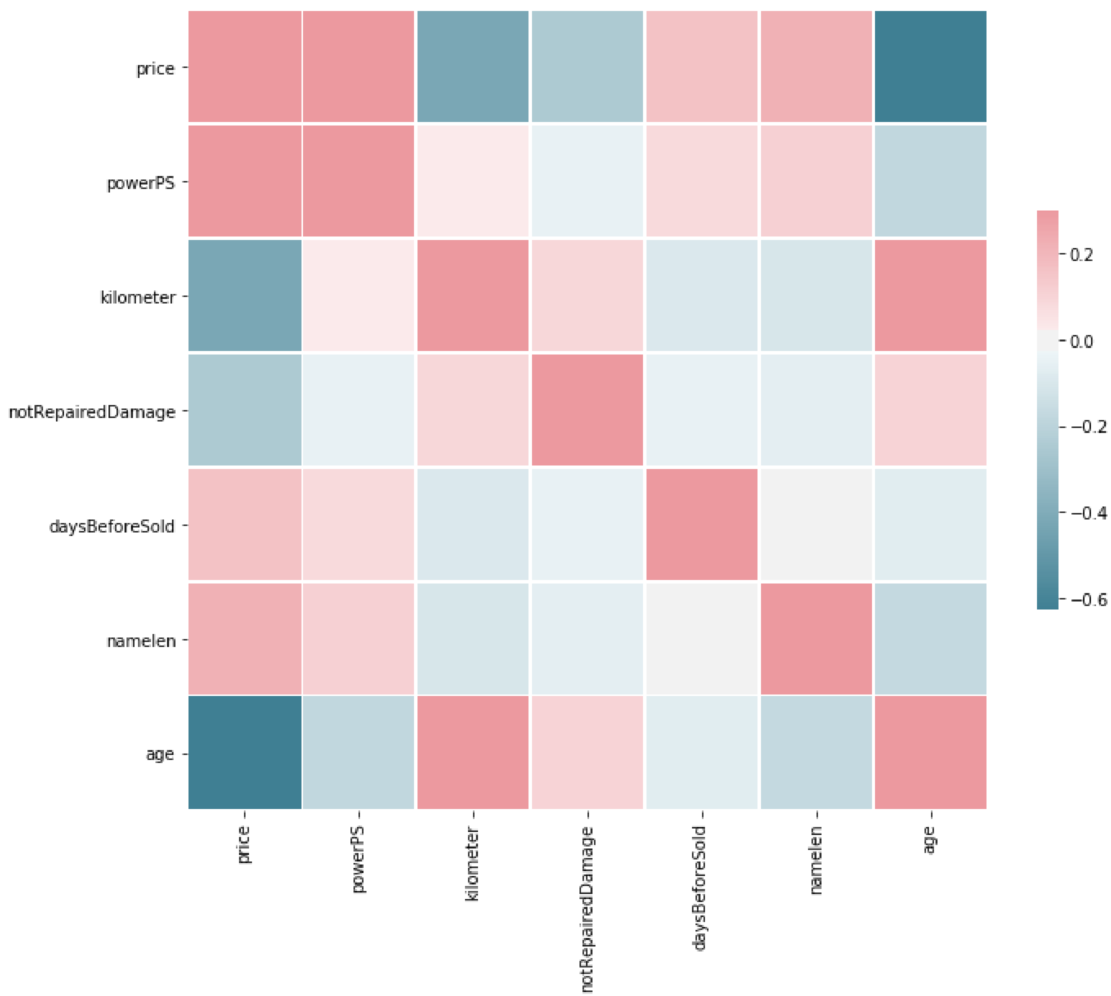
3.4 Correlation

As we can see in the correlation plot while powerPS, daysBeforeSold and namelen has positive effect on price, kilometer and age has negative effect as we suspected.

Kilometer and age has also positive correlation but that's not a surprise.

[74]:
```
# Compute the correlation matrix
corr = dfim.corr()

# Set up the matplotlib figure
f, ax = plt.subplots(figsize=(11, 9))

# Generate a custom diverging colormap
cmap = sns.diverging_palette(220, 10, as_cmap=True)

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(corr, cmap=cmap, vmax=.3, center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5})
```

[74]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1ba23c752e8>
```



3.5 Linear regression

We decided to start with linear regression in order to have a base to compare other methods.

First we wanted to check OLS Regression Results to see how every independent variable effecting price.

[75]:
```
mod = smf.ols(formula='price ~ vehicleType + gearbox + powerPS + kilometer +␣
 ↪fuelType + model + brand + notRepairedDamage', data=dfim)
res = mod.fit()
res.summary()
```

[75]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.700
```

```
Model:                            OLS   Adj. R-squared:                  0.700
Method:                 Least Squares   F-statistic:                     2188.
Date:                Wed, 10 Jun 2020   Prob (F-statistic):               0.00
Time:                        20:16:15   Log-Likelihood:            -2.5839e+05
No. Observations:              281853   AIC:                         5.174e+05
Df Residuals:                  281551   BIC:                         5.206e+05
Df Model:                         301
Covariance Type:            nonrobust
================================================================================
================
                                  coef    std err          t      P>|t|
[0.025      0.975]
--------------------------------------------------------------------------------
----------------
Intercept                       1.8468      0.064     28.686      0.000
1.721       1.973
vehicleType[T.cabrio]           0.1488      0.009     16.918      0.000
0.132       0.166
vehicleType[T.coupe]           -0.1174      0.009    -12.969      0.000
-0.135      -0.100
vehicleType[T.limousine]       -0.0822      0.007    -11.250      0.000
-0.097      -0.068
vehicleType[T.others]          -0.2043      0.015    -13.857      0.000
-0.233      -0.175
vehicleType[T.small car]       -0.2414      0.008    -31.538      0.000
-0.256      -0.226
vehicleType[T.station wagon]   -0.0899      0.007    -12.197      0.000
-0.104      -0.075
vehicleType[T.suv]              0.0374      0.012      3.194      0.001
0.014       0.060
gearbox[T.manual]              -0.0569      0.003    -16.577      0.000
-0.064      -0.050
fuelType[T.diesel]              0.1762      0.029      6.178      0.000
0.120       0.232
fuelType[T.electric]            0.9381      0.089     10.568      0.000
0.764       1.112
fuelType[T.hybrid]              0.3212      0.053      6.071      0.000
0.218       0.425
fuelType[T.lpg]                -0.2961      0.030     -9.885      0.000
-0.355      -0.237
fuelType[T.others]             -0.4119      0.097     -4.235      0.000
-0.602      -0.221
fuelType[T.petrol]             -0.3833      0.029    -13.440      0.000
-0.439      -0.327
model[T.145]                    0.1046      0.119      0.881      0.379
-0.128       0.337
model[T.147]                    0.6549      0.058     11.245      0.000
```

```
0.541       0.769
model[T.156]                     -0.0202      0.058     -0.348      0.728
-0.134       0.094
model[T.159]                      0.9781      0.067     14.514      0.000
0.846       1.110
model[T.1_reihe]                  1.2092      0.049     24.612      0.000
1.113       1.306
model[T.1er]                      1.3812      0.064     21.681      0.000
1.256       1.506
model[T.200]                      1.1303      0.352      3.214      0.001
0.441       1.820
model[T.2_reihe]                  1.1847      0.046     25.553      0.000
1.094       1.276
model[T.300c]                     1.6323      0.076     21.471      0.000
1.483       1.781
model[T.3_reihe]                  0.8219      0.046     17.835      0.000
0.732       0.912
model[T.3er]                      0.8472      0.063     13.431      0.000
0.724       0.971
model[T.4_reihe]                  0.5593      0.052     10.670      0.000
0.457       0.662
model[T.500]                      1.8519      0.050     36.696      0.000
1.753       1.951
model[T.5_reihe]                  1.2407      0.054     22.870      0.000
1.134       1.347
model[T.5er]                      0.7099      0.063     11.203      0.000
0.586       0.834
model[T.601]                      1.4441      0.276      5.238      0.000
0.904       1.984
model[T.6_reihe]                  0.7601      0.049     15.447      0.000
0.664       0.857
model[T.6er]                      1.1729      0.078     14.959      0.000
1.019       1.327
model[T.7er]                      0.4934      0.066      7.439      0.000
0.363       0.623
model[T.80]                       0.2014      0.042      4.759      0.000
0.118       0.284
model[T.850]                      0.3803      0.070      5.452      0.000
0.244       0.517
model[T.90]                       0.2780      0.100      2.785      0.005
0.082       0.474
model[T.900]                      0.8701      0.086     10.065      0.000
0.701       1.040
model[T.9000]                     0.8645      0.137      6.301      0.000
0.596       1.133
model[T.911]                      1.0419      0.079     13.223      0.000
0.887       1.196
```

```
model[T.a1]                        1.7708      0.046     38.665      0.000
1.681       1.861
model[T.a2]                        1.8264      0.051     35.721      0.000
1.726       1.927
model[T.a3]                        1.0951      0.039     28.225      0.000
1.019       1.171
model[T.a4]                        0.8239      0.039     21.399      0.000
0.748       0.899
model[T.a5]                        1.3844      0.043     32.027      0.000
1.300       1.469
model[T.a6]                        0.6483      0.039     16.658      0.000
0.572       0.725
model[T.a8]                        0.4554      0.047      9.697      0.000
0.363       0.547
model[T.a_klasse]                  0.9897      0.045     21.776      0.000
0.901       1.079
model[T.accord]                    0.9295      0.067     13.921      0.000
0.799       1.060
model[T.agila]                     1.3170      0.058     22.775      0.000
1.204       1.430
model[T.alhambra]                  0.9981      0.094     10.635      0.000
0.814       1.182
model[T.almera]                    0.6720      0.057     11.888      0.000
0.561       0.783
model[T.altea]                     1.2975      0.094     13.809      0.000
1.113       1.482
model[T.amarok]                    1.0028      0.113      8.841      0.000
0.781       1.225
model[T.andere]                    0.9266      0.042     22.016      0.000
0.844       1.009
model[T.antara]                    0.9805      0.079     12.425      0.000
0.826       1.135
model[T.arosa]                     1.5088      0.091     16.625      0.000
1.331       1.687
model[T.astra]                     0.8834      0.048     18.353      0.000
0.789       0.978
model[T.auris]                     1.1629      0.064     18.202      0.000
1.038       1.288
model[T.avensis]                   0.9226      0.055     16.753      0.000
0.815       1.031
model[T.aveo]                      1.6050      0.078     20.639      0.000
1.453       1.757
model[T.aygo]                      1.6855      0.057     29.339      0.000
1.573       1.798
model[T.b_klasse]                  1.4349      0.049     29.164      0.000
1.338       1.531
model[T.b_max]                     2.0223      0.130     15.580      0.000
```

```
1.768       2.277
model[T.beetle]                       1.2838      0.055     23.227      0.000
1.175       1.392
model[T.berlingo]                     1.2449      0.051     24.439      0.000
1.145       1.345
model[T.bora]                         0.7539      0.055     13.635      0.000
0.646       0.862
model[T.boxster]                      0.5675      0.075      7.553      0.000
0.420       0.715
model[T.bravo]                        0.8557      0.060     14.224      0.000
0.738       0.974
model[T.c1]                           1.9147      0.056     34.154      0.000
1.805       2.025
model[T.c2]                           1.7804      0.056     31.705      0.000
1.670       1.890
model[T.c3]                           1.7084      0.054     31.775      0.000
1.603       1.814
model[T.c4]                           1.3733      0.055     25.159      0.000
1.266       1.480
model[T.c5]                           0.7872      0.054     14.533      0.000
0.681       0.893
model[T.c_klasse]                     0.7589      0.045     16.919      0.000
0.671       0.847
model[T.c_max]                        1.6816      0.052     32.082      0.000
1.579       1.784
model[T.c_reihe]                      1.5307      0.068     22.500      0.000
1.397       1.664
model[T.caddy]                        1.3847      0.053     25.932      0.000
1.280       1.489
model[T.calibra]                      0.4456      0.068      6.560      0.000
0.312       0.579
model[T.captiva]                      1.2458      0.068     18.228      0.000
1.112       1.380
model[T.carisma]                      0.4578      0.059      7.724      0.000
0.342       0.574
model[T.carnival]                     0.3182      0.067      4.760      0.000
0.187       0.449
model[T.cayenne]                      0.5196      0.078      6.689      0.000
0.367       0.672
model[T.cc]                           1.4227      0.068     20.819      0.000
1.289       1.557
model[T.ceed]                         1.7403      0.065     26.798      0.000
1.613       1.868
model[T.charade]                      0.0631      0.159      0.396      0.692
-0.249       0.375
model[T.cherokee]                     0.8410      0.078     10.800      0.000
0.688       0.994
```

```
model[T.citigo]                    1.9904      0.090     22.070      0.000
1.814       2.167
model[T.civic]                     1.0204      0.060     17.131      0.000
0.904       1.137
model[T.cl]                        0.9466      0.059     15.976      0.000
0.831       1.063
model[T.clio]                      1.1741      0.049     23.768      0.000
1.077       1.271
model[T.clk]                       0.8045      0.047     17.094      0.000
0.712       0.897
model[T.clubman]                   0.5661      0.096      5.903      0.000
0.378       0.754
model[T.colt]                      1.3028      0.054     24.238      0.000
1.197       1.408
model[T.combo]                     1.1012      0.058     18.928      0.000
0.987       1.215
model[T.cooper]                    0.4562      0.086      5.287      0.000
0.287       0.625
model[T.cordoba]                   0.7100      0.094      7.516      0.000
0.525       0.895
model[T.corolla]                   0.9233      0.054     17.013      0.000
0.817       1.030
model[T.corsa]                     1.3150      0.048     27.319      0.000
1.221       1.409
model[T.cr_reihe]                  1.2148      0.067     18.266      0.000
1.084       1.345
model[T.croma]                     0.5627      0.111      5.081      0.000
0.346       0.780
model[T.crossfire]                 1.6196      0.095     17.042      0.000
1.433       1.806
model[T.cuore]                     0.8917      0.087     10.241      0.000
0.721       1.062
model[T.cx_reihe]                  1.1878      0.074     16.001      0.000
1.042       1.333
model[T.defender]                  2.2918      0.242      9.463      0.000
1.817       2.766
model[T.delta]                     1.7674      0.142     12.483      0.000
1.490       2.045
model[T.discovery]                 1.4359      0.237      6.047      0.000
0.971       1.901
model[T.doblo]                     1.1523      0.059     19.672      0.000
1.038       1.267
model[T.ducato]                    0.8436      0.057     14.777      0.000
0.732       0.955
model[T.duster]                    0.8548      0.152      5.623      0.000
0.557       1.153
model[T.e_klasse]                  0.6944      0.045     15.420      0.000
```

```
0.606       0.783
model[T.elefantino]                 1.5818      0.261      6.063      0.000
1.070       2.093
model[T.eos]                        1.1719      0.058     20.236      0.000
1.058       1.285
model[T.escort]                     0.5166      0.053      9.663      0.000
0.412       0.621
model[T.espace]                     0.1571      0.055      2.834      0.005
0.048       0.266
model[T.exeo]                       1.3274      0.107     12.426      0.000
1.118       1.537
model[T.fabia]                      1.7701      0.064     27.633      0.000
1.645       1.896
model[T.fiesta]                     1.7904      0.047     37.822      0.000
1.698       1.883
model[T.focus]                      1.3199      0.047     28.011      0.000
1.228       1.412
model[T.forester]                   0.7604      0.107      7.103      0.000
0.551       0.970
model[T.forfour]                    0.9733      0.115      8.495      0.000
0.749       1.198
model[T.fortwo]                     1.1146      0.110     10.143      0.000
0.899       1.330
model[T.fox]                        1.8658      0.056     33.162      0.000
1.756       1.976
model[T.freelander]                 1.3894      0.239      5.812      0.000
0.921       1.858
model[T.fusion]                     1.9567      0.060     32.849      0.000
1.840       2.073
model[T.g_klasse]                   1.7848      0.077     23.283      0.000
1.635       1.935
model[T.galant]                     0.1469      0.066      2.241      0.025
0.018       0.275
model[T.galaxy]                     1.1274      0.051     22.300      0.000
1.028       1.227
model[T.getz]                       1.6072      0.057     28.108      0.000
1.495       1.719
model[T.gl]                         1.3157      0.102     12.896      0.000
1.116       1.516
model[T.glk]                        1.2267      0.062     19.743      0.000
1.105       1.348
model[T.golf]                       1.0616      0.051     20.853      0.000
0.962       1.161
model[T.grand]                      1.0174      0.055     18.586      0.000
0.910       1.125
model[T.i3]                         0.7359      0.255      2.881      0.004
0.235       1.237
```

```
model[T.i_reihe]                  1.7304      0.050     34.802      0.000
1.633       1.828
model[T.ibiza]                    1.5529      0.089     17.495      0.000
1.379       1.727
model[T.impreza]                  0.4156      0.101      4.105      0.000
0.217       0.614
model[T.insignia]                 1.2794      0.054     23.816      0.000
1.174       1.385
model[T.jazz]                     1.7915      0.068     26.506      0.000
1.659       1.924
model[T.jetta]                    1.1531      0.064     18.092      0.000
1.028       1.278
model[T.jimny]                    1.6136      0.065     24.867      0.000
1.486       1.741
model[T.juke]                     1.5374      0.079     19.400      0.000
1.382       1.693
model[T.justy]                    0.5270      0.109      4.833      0.000
0.313       0.741
model[T.ka]                       1.4858      0.049     30.596      0.000
1.391       1.581
model[T.kadett]                   0.5534      0.075      7.369      0.000
0.406       0.701
model[T.kaefer]                   2.2031      0.116     19.054      0.000
1.976       2.430
model[T.kalina]                   0.7649      0.334      2.288      0.022
0.110       1.420
model[T.kalos]                    1.5361      0.097     15.854      0.000
1.346       1.726
model[T.kangoo]                   1.0455      0.053     19.828      0.000
0.942       1.149
model[T.kappa]                    0.8499      0.188      4.532      0.000
0.482       1.217
model[T.kuga]                     1.7540      0.057     30.574      0.000
1.642       1.866
model[T.laguna]                   0.2409      0.051      4.681      0.000
0.140       0.342
model[T.lancer]                   1.2222      0.066     18.587      0.000
1.093       1.351
model[T.lanos]                    0.9721      0.102      9.486      0.000
0.771       1.173
model[T.legacy]                   0.1467      0.105      1.390      0.164
-0.060       0.353
model[T.leon]                     1.2703      0.090     14.181      0.000
1.095       1.446
model[T.lodgy]                    0.7458      0.184      4.061      0.000
0.386       1.106
model[T.logan]                    0.8080      0.150      5.399      0.000
```

```
0.515       1.101
model[T.lupo]                         1.4643      0.053     27.844      0.000
1.361       1.567
model[T.lybra]                        0.1121      0.130      0.863      0.388
-0.142       0.367
model[T.m_klasse]                     0.8297      0.049     16.765      0.000
0.733       0.927
model[T.m_reihe]                      0.9758      0.080     12.205      0.000
0.819       1.133
model[T.materia]                      1.3600      0.166      8.176      0.000
1.034       1.686
model[T.matiz]                        1.6560      0.059     27.896      0.000
1.540       1.772
model[T.megane]                       0.6688      0.050     13.482      0.000
0.572       0.766
model[T.meriva]                       1.1897      0.052     22.974      0.000
1.088       1.291
model[T.micra]                        1.3602      0.052     26.120      0.000
1.258       1.462
model[T.mii]                          2.0504      0.113     18.160      0.000
1.829       2.272
model[T.modus]                        1.2621      0.061     20.853      0.000
1.143       1.381
model[T.mondeo]                       0.8577      0.048     17.955      0.000
0.764       0.951
model[T.move]                         0.7553      0.131      5.759      0.000
0.498       1.012
model[T.musa]                         1.9493      0.162     12.035      0.000
1.632       2.267
model[T.mustang]                      1.4954      0.071     21.089      0.000
1.356       1.634
model[T.mx_reihe]                     1.1171      0.050     22.205      0.000
1.018       1.216
model[T.navara]                       1.2215      0.078     15.694      0.000
1.069       1.374
model[T.niva]                         1.4055      0.154      9.104      0.000
1.103       1.708
model[T.note]                         1.4948      0.074     20.137      0.000
1.349       1.640
model[T.nubira]                       0.6658      0.109      6.084      0.000
0.451       0.880
model[T.octavia]                      1.3846      0.064     21.601      0.000
1.259       1.510
model[T.omega]                       -0.2049      0.051     -4.005      0.000
-0.305      -0.105
model[T.one]                          0.8781      0.088     10.009      0.000
0.706       1.050
```

```
model[T.outlander]               1.5699      0.075     21.049      0.000
1.424       1.716
model[T.pajero]                  1.1979      0.066     18.098      0.000
1.068       1.328
model[T.panda]                   1.7906      0.052     34.277      0.000
1.688       1.893
model[T.passat]                  0.7819      0.051     15.248      0.000
0.681       0.882
model[T.phaeton]                 0.6811      0.068     10.089      0.000
0.549       0.813
model[T.picanto]                 2.0694      0.062     33.310      0.000
1.948       2.191
model[T.polo]                    1.4358      0.051     28.033      0.000
1.335       1.536
model[T.primera]                 0.4019      0.056      7.137      0.000
0.292       0.512
model[T.ptcruiser]               1.2115      0.065     18.619      0.000
1.084       1.339
model[T.punto]                   1.2169      0.047     25.919      0.000
1.125       1.309
model[T.q3]                      1.2182      0.056     21.886      0.000
1.109       1.327
model[T.q5]                      1.3957      0.050     27.979      0.000
1.298       1.493
model[T.q7]                      1.1500      0.053     21.586      0.000
1.046       1.254
model[T.qashqai]                 1.5911      0.056     28.415      0.000
1.481       1.701
model[T.r19]                     0.2552      0.082      3.100      0.002
0.094       0.417
model[T.range_rover]             1.1596      0.246      4.709      0.000
0.677       1.642
model[T.range_rover_evoque]      1.6421      0.249      6.584      0.000
1.153       2.131
model[T.range_rover_sport]       1.5029      0.245      6.123      0.000
1.022       1.984
model[T.rangerover]              1.8623      0.276      6.738      0.000
1.321       2.404
model[T.rav]                     0.9549      0.059     16.133      0.000
0.839       1.071
model[T.rio]                     1.5225      0.062     24.392      0.000
1.400       1.645
model[T.roadster]                1.1239      0.120      9.351      0.000
0.888       1.359
model[T.roomster]                1.8212      0.073     24.846      0.000
1.678       1.965
model[T.rx_reihe]                0.6012      0.073      8.282      0.000
```

```
0.459       0.743
model[T.s60]                    1.2958      0.072     18.016      0.000
1.155       1.437
model[T.s_klasse]               0.6658      0.050     13.317      0.000
0.568       0.764
model[T.s_max]                  1.7983      0.055     32.472      0.000
1.690       1.907
model[T.s_type]                 0.2808      0.083      3.380      0.001
0.118       0.444
model[T.samara]                 0.7614      0.306      2.491      0.013
0.162       1.360
model[T.sandero]                1.0600      0.150      7.072      0.000
0.766       1.354
model[T.santa]                  1.1583      0.059     19.643      0.000
1.043       1.274
model[T.scenic]                 0.5441      0.051     10.657      0.000
0.444       0.644
model[T.scirocco]               1.4196      0.058     24.664      0.000
1.307       1.532
model[T.seicento]               1.0837      0.054     20.242      0.000
0.979       1.189
model[T.sharan]                 0.8614      0.054     16.003      0.000
0.756       0.967
model[T.signum]                 0.6907      0.056     12.363      0.000
0.581       0.800
model[T.sirion]                 1.1011      0.096     11.441      0.000
0.912       1.290
model[T.sl]                     1.1197      0.054     20.546      0.000
1.013       1.227
model[T.slk]                    0.9618      0.048     20.033      0.000
0.868       1.056
model[T.sorento]                1.2642      0.063     20.115      0.000
1.141       1.387
model[T.spark]                  1.6810      0.074     22.653      0.000
1.536       1.826
model[T.spider]                 0.7522      0.075     10.085      0.000
0.606       0.898
model[T.sportage]               1.5523      0.064     24.225      0.000
1.427       1.678
model[T.sprinter]               1.1359      0.051     22.319      0.000
1.036       1.236
model[T.stilo]                  0.6264      0.053     11.786      0.000
0.522       0.731
model[T.superb]                 1.3854      0.070     19.760      0.000
1.248       1.523
model[T.swift]                  1.2914      0.053     24.223      0.000
1.187       1.396
```

```
model[T.terios]                 1.1534      0.137      8.427      0.000
0.885       1.422
model[T.tigra]                  0.8146      0.054     15.219      0.000
0.710       0.920
model[T.tiguan]                 1.3716      0.056     24.545      0.000
1.262       1.481
model[T.toledo]                 0.5895      0.096      6.171      0.000
0.402       0.777
model[T.touareg]                0.8677      0.058     15.084      0.000
0.755       0.980
model[T.touran]                 1.2259      0.052     23.530      0.000
1.124       1.328
model[T.transit]                1.5779      0.053     29.979      0.000
1.475       1.681
model[T.transporter]            1.4610      0.052     28.247      0.000
1.360       1.562
model[T.tt]                     1.1289      0.042     26.875      0.000
1.047       1.211
model[T.tucson]                 1.3483      0.063     21.474      0.000
1.225       1.471
model[T.twingo]                 1.1578      0.049     23.563      0.000
1.061       1.254
model[T.up]                     1.9380      0.061     31.829      0.000
1.819       2.057
model[T.v40]                    0.6088      0.057     10.645      0.000
0.497       0.721
model[T.v50]                    1.5457      0.065     23.734      0.000
1.418       1.673
model[T.v60]                    1.7313      0.105     16.488      0.000
1.525       1.937
model[T.v70]                    0.9641      0.058     16.699      0.000
0.851       1.077
model[T.v_klasse]               0.7042      0.071      9.981      0.000
0.566       0.842
model[T.vectra]                 0.3085      0.049      6.313      0.000
0.213       0.404
model[T.verso]                  1.0629      0.060     17.717      0.000
0.945       1.181
model[T.viano]                  1.3865      0.056     24.837      0.000
1.277       1.496
model[T.vito]                   1.0376      0.050     20.853      0.000
0.940       1.135
model[T.vivaro]                 1.3636      0.061     22.338      0.000
1.244       1.483
model[T.voyager]                0.8044      0.066     12.246      0.000
0.676       0.933
model[T.wrangler]               1.5290      0.079     19.313      0.000
```

```
1.374       1.684
model[T.x_reihe]                    1.1355      0.065     17.581      0.000
1.009       1.262
model[T.x_trail]                    1.0725      0.067     16.107      0.000
0.942       1.203
model[T.x_type]                     0.7191      0.079      9.113      0.000
0.564       0.874
model[T.xc_reihe]                   1.6091      0.063     25.350      0.000
1.485       1.734
model[T.yaris]                      1.4819      0.052     28.448      0.000
1.380       1.584
model[T.yeti]                       1.5807      0.077     20.581      0.000
1.430       1.731
model[T.ypsilon]                    1.7062      0.096     17.748      0.000
1.518       1.895
model[T.z_reihe]                    1.2700      0.067     19.058      0.000
1.139       1.401
model[T.zafira]                     0.8742      0.049     17.706      0.000
0.777       0.971
brand[T.audi]                       0.1514      0.035      4.347      0.000
0.083       0.220
brand[T.bmw]                        0.0831      0.055      1.499      0.134
-0.026       0.192
brand[T.chevrolet]                 -0.1161      0.038     -3.084      0.002
-0.190      -0.042
brand[T.chrysler]                  -0.7256      0.044    -16.625      0.000
-0.811      -0.640
brand[T.citroen]                   -0.1802      0.034     -5.261      0.000
-0.247      -0.113
brand[T.dacia]                      0.5313      0.142      3.738      0.000
0.253       0.810
brand[T.daewoo]                    -0.6205      0.060    -10.332      0.000
-0.738      -0.503
brand[T.daihatsu]                   0.2599      0.072      3.598      0.000
0.118       0.402
brand[T.fiat]                      -0.0164      0.034     -0.478      0.633
-0.084       0.051
brand[T.ford]                      -0.4807      0.036    -13.535      0.000
-0.550      -0.411
brand[T.honda]                     -0.1289      0.048     -2.667      0.008
-0.224      -0.034
brand[T.hyundai]                   -0.2242      0.036     -6.286      0.000
-0.294      -0.154
brand[T.jaguar]                     0.0285      0.053      0.543      0.587
-0.074       0.131
brand[T.jeep]                      -0.1692      0.051     -3.312      0.001
-0.269      -0.069
```

brand[T.kia]	-0.3766	0.042	-8.868	0.000	-0.460	-0.293
brand[T.lada]	-0.4098	0.139	-2.955	0.003	-0.682	-0.138
brand[T.lancia]	-0.5988	0.078	-7.687	0.000	-0.751	-0.446
brand[T.land_rover]	-0.2261	0.234	-0.966	0.334	-0.685	0.233
brand[T.mazda]	-0.0614	0.035	-1.777	0.076	-0.129	0.006
brand[T.mercedes_benz]	0.1136	0.033	3.460	0.001	0.049	0.178
brand[T.mini]	1.1033	0.080	13.828	0.000	0.947	1.260
brand[T.mitsubishi]	-0.3105	0.038	-8.086	0.000	-0.386	-0.235
brand[T.nissan]	-0.0224	0.039	-0.571	0.568	-0.099	0.054
brand[T.opel]	0.0124	0.037	0.331	0.740	-0.061	0.085
brand[T.peugeot]	-0.0273	0.034	-0.797	0.425	-0.094	0.040
brand[T.porsche]	0.6413	0.060	10.702	0.000	0.524	0.759
brand[T.renault]	0.0563	0.038	1.497	0.134	-0.017	0.130
brand[T.rover]	-0.5899	0.046	-12.789	0.000	-0.680	-0.499
brand[T.saab]	-0.4309	0.045	-9.641	0.000	-0.518	-0.343
brand[T.seat]	-0.0635	0.083	-0.768	0.442	-0.226	0.099
brand[T.skoda]	-0.1085	0.055	-1.972	0.049	-0.216	-0.001
brand[T.smart]	0.5569	0.105	5.283	0.000	0.350	0.763
brand[T.subaru]	0.2625	0.086	3.043	0.002	0.093	0.432
brand[T.suzuki]	0.1446	0.036	4.001	0.000	0.074	0.215
brand[T.toyota]	0.1394	0.038	3.704	0.000	0.066	0.213
brand[T.trabant]	0.8057	0.231	3.489	0.000	0.353	1.258
brand[T.volkswagen]	0.0958	0.041	2.328	0.020	0.015	0.176
brand[T.volvo]	-0.3695	0.042	-8.727	0.000		

```
-0.452      -0.286
powerPS                          1.4694      0.005    293.154      0.000
1.460       1.479
kilometer                    -1.069e-05   3.31e-08   -322.660      0.000
-1.08e-05   -1.06e-05
notRepairedDamage               -0.6162      0.004   -151.312      0.000
-0.624      -0.608
==============================================================================
Omnibus:                    28567.517   Durbin-Watson:                   2.000
Prob(Omnibus):                  0.000   Jarque-Bera (JB):            61264.757
Skew:                          -0.643   Prob(JB):                         0.00
Kurtosis:                       4.887   Cond. No.                     8.01e+07
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 8.01e+07. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

First step for linear regression prediction is preparing dependent and independent variables

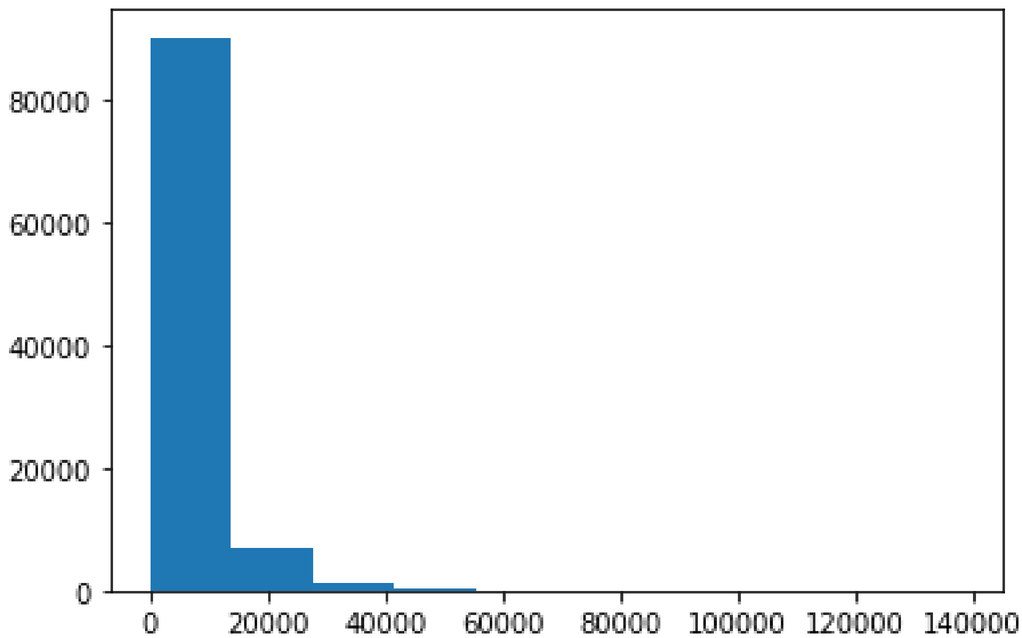
[76]:
```
X = dfimdm.drop(['price'], axis='columns')
y = dfimdm.price
```

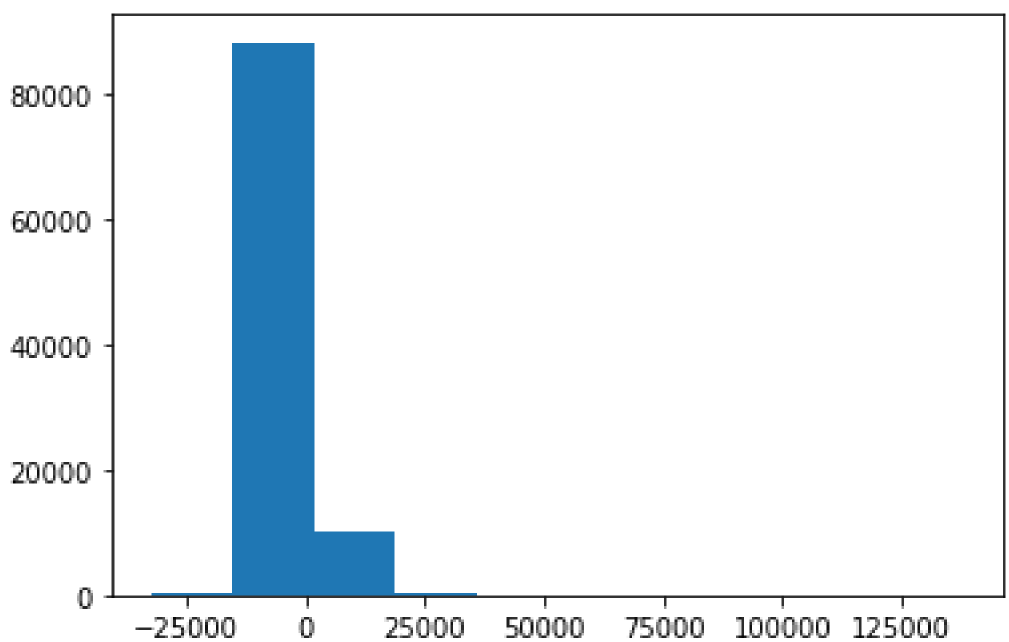
We decided to calculate predicted values and check the distribution of errors(differences between real and predicted values).

[77]:
```
X_train, X_test, y_train, y_testLR = train_test_split(X, y,test_size=0.35)

linreg = LinearRegression()
linreg.fit(X_train, y_train)
y_predLR = linreg.predict(X_test)
scoreLR = linreg.score(X_test, y_testLR)
print("Mean squared error:", np.sqrt(metrics.mean_squared_error(y_testLR,␣
 ↪y_predLR)))
print("R-squared score:", scoreLR*100,"%")
```

```
Mean squared error: 0.5129529221824182
R-squared score: 78.58930520058853 %
```

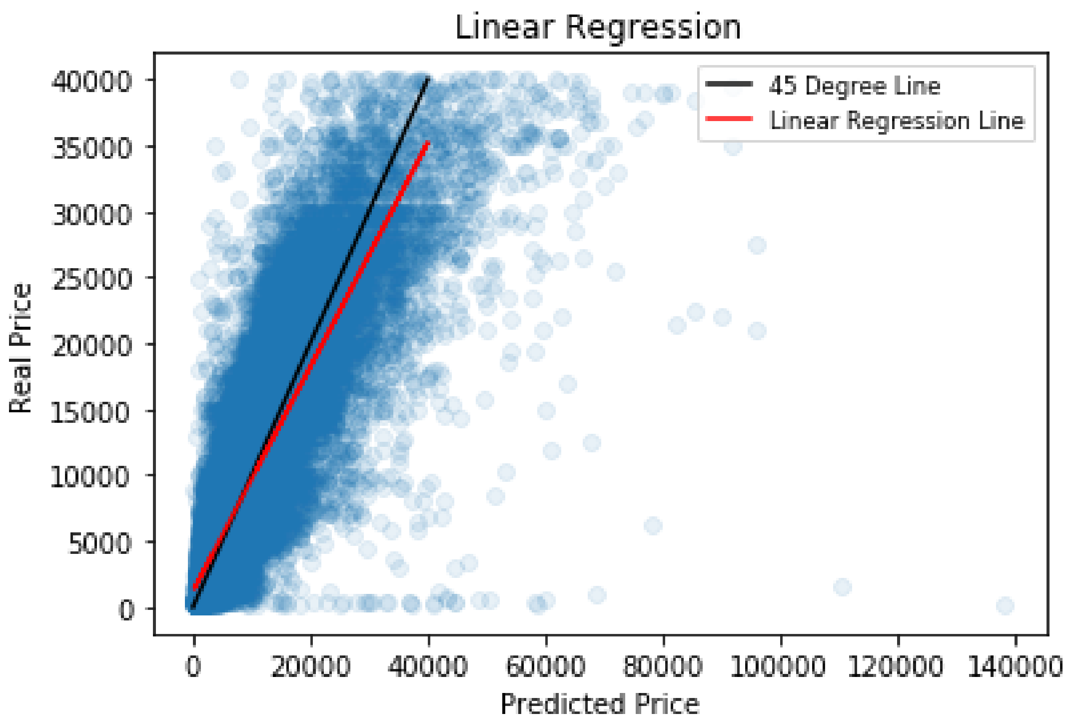
[78]:
```
plt.hist(np.exp(y_predLR))
plt.show()
plt.hist(np.exp(y_predLR) - np.exp(y_testLR.values))
plt.show()
```





In the first plot we can see how Linear Regression prediction looks like and in the second plot we can see that the difference between predicted and real prices.

[79]:
```
plt.scatter(np.exp(y_predLR), np.exp(y_testLR.values), alpha=0.1)
plt.xlabel('Predicted Price')
plt.ylabel('Real Price')
m, b = np.polyfit(np.exp(y_predLR), np.exp(y_testLR.values), 1)
plt.plot([0, 40000], [0, 40000], color = 'black')
plt.plot(np.exp(y_testLR.values), m*np.exp(y_testLR.values) + b, 'red')
plt.legend(["45 Degree Line","Linear Regression Line"], fontsize = "small")
plt.title("Linear Regression")
plt.show()
```



Looks like with linear regression our estimates was correct 78% but it overshoot for some cars and cause outliers. Since we set 40000 as max in our dataset it should have stay there, but we can clearly see from plot that there are even predictions for 160000. Beside that the line is almost visible.

We decided to try Linear Regression with the data which we removed "model" from it.

[80]:
```
X = dmNoModel.drop(['price'], axis='columns')
y = dmNoModel.price

X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.35)

linregNM = LinearRegression()
linregNM.fit(X_train, y_train)
```

```
y_predLRnm = linregNM.predict(X_test)
scoreLRnm = linregNM.score(X_test, y_test)
print("Mean squared error:", np.sqrt(metrics.mean_squared_error(y_test,␣
 ↪y_predLRnm)))
print("R-squared score:", scoreLRnm*100,"%")
```

```
Mean squared error: 0.5461403062141418
R-squared score: 75.69693562079023 %
```

It looks like R-squared score decrease 3% we can clearly say that model is very important feature for our models. Having it in our models increases need to computational power and time it takes to run the code. But looks like it worths it.

3.6 K-Nearest Neighbors

Next method we want to use is KNN. It is pretty hard to deal large dataset with KNN but we are thinking it might be successfull.

[81]:
```
dfimdm.head()
```

[81]:

	price	powerPS	kilometer	notRepairedDamage	daysBeforeSold	namelen	\
2	9.190138	5.093750	125000	0	22	30	
3	7.313220	4.317488	150000	0	0	18	
4	8.188689	4.234107	90000	0	6	30	
5	6.476972	4.624973	150000	1	2	50	
6	7.696213	4.691348	150000	0	4	27	

	age	vehicleType_bus	vehicleType_cabrio	vehicleType_coupe	…	\
2	13	0	0	0	…	
3	16	0	0	0	…	
4	9	0	0	0	…	
5	22	0	0	0	…	
6	13	0	1	0	…	

	brand_saab	brand_seat	brand_skoda	brand_smart	brand_subaru	\
2	0	0	0	0	0	
3	0	0	0	0	0	
4	0	0	1	0	0	
5	0	0	0	0	0	
6	0	0	0	0	0	

	brand_suzuki	brand_toyota	brand_trabant	brand_volkswagen	brand_volvo
2	0	0	0	0	0
3	0	0	0	1	0
4	0	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0

```
[5 rows x 310 columns]
```

Firstly we wanted to work on the sample to see how KNN is dealing with our data. If it's worth to trouble.

After taking 50000 rows from our data we split them as 0.7 train and 0.3 test

[82]:
```
dfimdmsm = dfimdm.sample(50000)
# Create train and test set

train , test = train_test_split(dfimdmsm, test_size = 0.3)

x_train = train.drop('price', axis=1)
y_train = train['price']

x_test = test.drop('price', axis = 1)
y_test = test['price']
```

Since KNN is a Distance-Based algorithm we need to scale all variables otherwise KNN will not perform optimally.

We decided to scale them between 0 and 1.

[83]:
```
# Preprocessing - Scaling the features

scaler = MinMaxScaler(feature_range=(0, 1))

x_train_scaled = scaler.fit_transform(x_train)
x_train = pd.DataFrame(x_train_scaled)

x_test_scaled = scaler.fit_transform(x_test)
x_test = pd.DataFrame(x_test_scaled)
```

In order to decide how many neighbors we should have take we decided to check one by one what are the mean squared errors and what are the R-squared score.

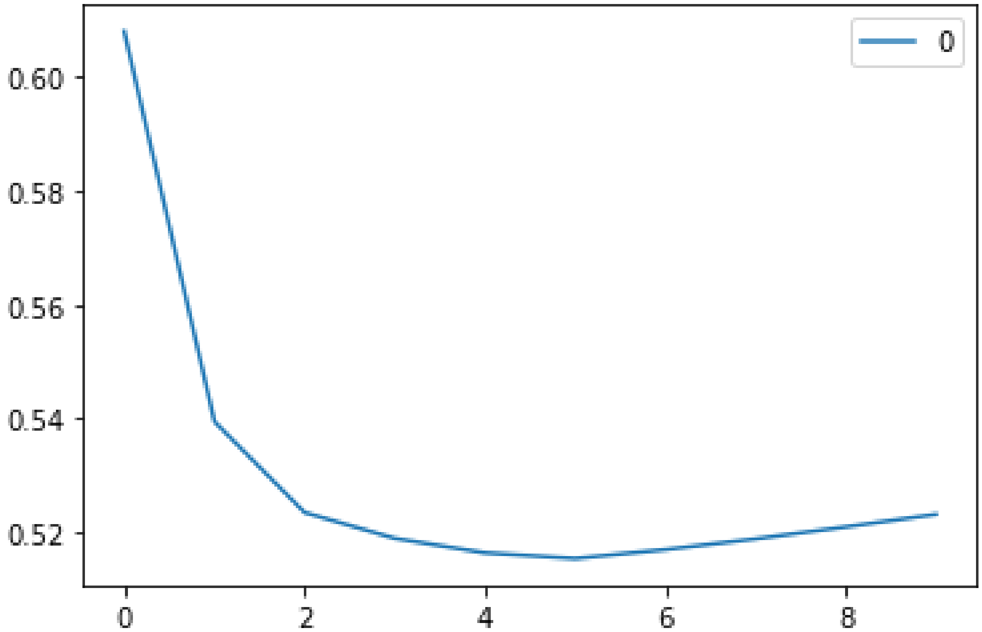
[84]:
```
# 50000 sample
# checking the error rate for different n_neighbors

rmse_val = [] #to store rmse values for different k
for K in range(10):
    K = K+1
    model = neighbors.KNeighborsRegressor(n_neighbors = K)

    model.fit(x_train, y_train)  #fit the model
    pred=model.predict(x_test) #make prediction on test set
    error = sqrt(mean_squared_error(y_test,pred)) #calculate rmse
    rmse_val.append(error) #store rmse values
```

```
    accur = metrics.r2_score(y_test.values, pred)
    print('RMSE value for k= ' , K , 'is:', error)
    print('R-squared score:', accur)
```

```
RMSE value for k=  1 is: 0.6080560421523825
R-squared score: 0.6973588626075502
RMSE value for k=  2 is: 0.5394626768055544
R-squared score: 0.761788046954268
RMSE value for k=  3 is: 0.5234642572165564
R-squared score: 0.7757074701359661
RMSE value for k=  4 is: 0.5189137986664447
R-squared score: 0.7795900568722829
RMSE value for k=  5 is: 0.5163611492951381
R-squared score: 0.7817532117411445
RMSE value for k=  6 is: 0.5153717265257459
R-squared score: 0.7825887953530072
RMSE value for k=  7 is: 0.5169386244209656
R-squared score: 0.7812647839576071
RMSE value for k=  8 is: 0.5188240400932973
R-squared score: 0.7796663006377123
RMSE value for k=  9 is: 0.5209363471358256
R-squared score: 0.7778685433480225
RMSE value for k=  10 is: 0.5231404769646025
R-squared score: 0.7759848492747023
```

[85]:
```
# 50000
# k = 6
#plotting the rmse values against k values
curve = pd.DataFrame(rmse_val) #elbow curve
curve.plot()
```

[85]: <matplotlib.axes._subplots.AxesSubplot at 0x1ba3ee8c5c0>



[86]:
```
kf = KFold(n_splits=5, shuffle=True, random_state=random.randint(0,10000))
errors = []
accurs = []
for train, test in kf.split(dfimdmsm.index.values):
    x = dfimdmsm.drop('price', axis=1)
    y = dfimdmsm['price']
    x_train = x.iloc[train]
    y_train = y.iloc[train]

    x_test = x.iloc[test]
    y_test = y.iloc[test]

    scaler = MinMaxScaler(feature_range=(0, 1))

    x_train_scaled = scaler.fit_transform(x_train)
    x_train = pd.DataFrame(x_train_scaled)

    x_test_scaled = scaler.fit_transform(x_test)
    x_test = pd.DataFrame(x_test_scaled)

    model = neighbors.KNeighborsRegressor(n_neighbors = 6)

    model.fit(x_train, y_train)  #fit the model
    pred=model.predict(x_test) #make prediction on test set
    error = sqrt(mean_squared_error(y_test,pred)) #calculate rmse
    errors.append(error) #store rmse values
```

```
    accur = metrics.r2_score(y_test.values, pred)
    accurs.append(accur)
    print('Mean squared error:', error)
    print('R-squared score:', accur)
print('Average Mean squared error:', np.mean(error))
print('Average R-squared score:', np.mean(accur))
```

```
Mean squared error: 0.5097667245894799
R-squared score: 0.7887567176356063
Mean squared error: 0.4972572716109484
R-squared score: 0.7967629504564305
Mean squared error: 0.5130122267239671
R-squared score: 0.7806008658495368
Mean squared error: 0.5073440351737304
R-squared score: 0.788694163689195
Mean squared error: 0.5311165244331162
R-squared score: 0.7682000380565733
Average Mean squared error: 0.5311165244331162
Average R-squared score: 0.7682000380565733
```

It looks like it worth the trouble. Even for 50000 sample it gave same result like linear regression(78%). So we decided to go for it with the all data.

[87]:
```
x = dfimdm.drop(['price'], axis='columns')
y = dfimdm.price
```

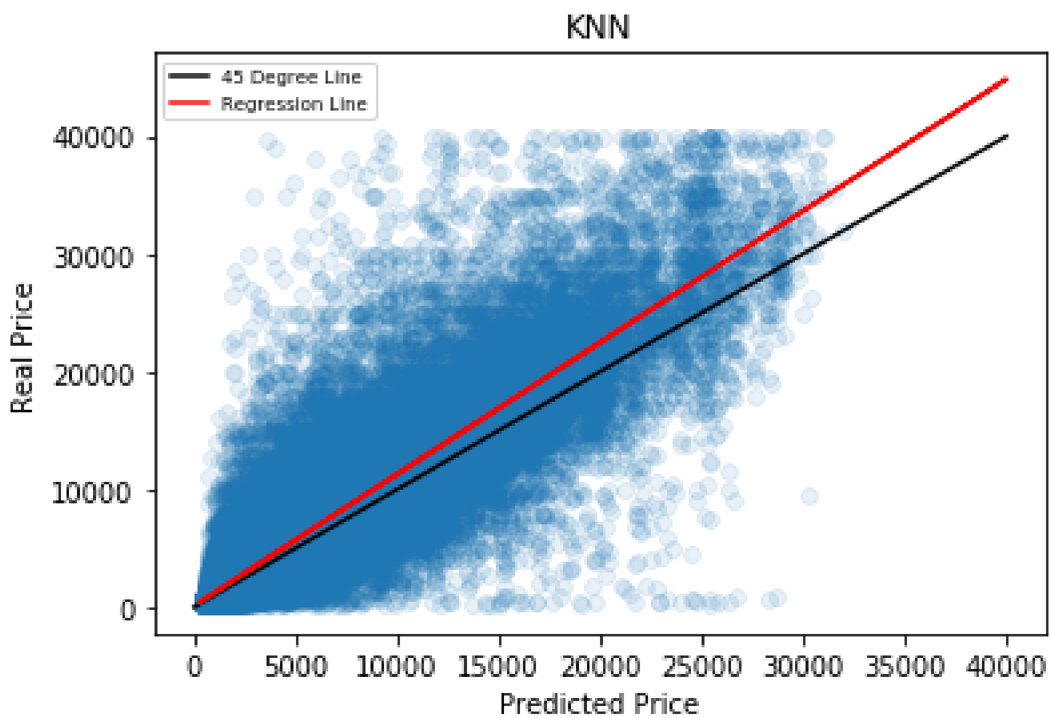
Since we are to fit and test with all data this time we were not be able to use Kfold or check what is the best n_neighbhors but we are guessing that 50 should be enough.

[88]:
```
x_train, x_test, y_train, y_test = train_test_split(x, y,test_size=0.35)



# Preprocessing - Scaling the features

scaler = MinMaxScaler(feature_range=(0, 1))

x_train_scaled = scaler.fit_transform(x_train)
x_train = pd.DataFrame(x_train_scaled)

x_test_scaled = scaler.fit_transform(x_test)
x_test = pd.DataFrame(x_test_scaled)

# classifier
model = neighbors.KNeighborsRegressor(n_neighbors = 50)

model.fit(x_train, y_train)  #fit the model
pred=model.predict(x_test) #make prediction on test set
error = sqrt(mean_squared_error(y_test,pred)) #calculate rmse
```

```python
errors.append(error) #store rmse values
accur = metrics.r2_score(y_test.values, pred)
accurs.append(accur)
print('Mean squared error:', error)
print('R-squared score:', accur)
```

```
Mean squared error: 0.4991365616561806
R-squared score: 0.7962880626504695
```

[89]:
```python
plt.scatter(np.exp(pred), np.exp(y_test.values), alpha=0.1)
plt.xlabel('Predicted Price')
plt.ylabel('Real Price')
m, b = np.polyfit(np.exp(pred), np.exp(y_test.values), 1)
plt.plot([0, 40000], [0, 40000], color = 'black')
plt.plot(np.exp(y_test.values), m*np.exp(y_test.values) + b, 'red')
plt.legend(["45 Degree Line","Regression Line"], fontsize = "x-small")
plt.title("KNN")
plt.show()
```



We see that KNN with 50 neighbors couldn't predict expensive cars accurately. It did more or less good job until 25k but after that start to not be accurate. Especially after 30k it did not work well at all.

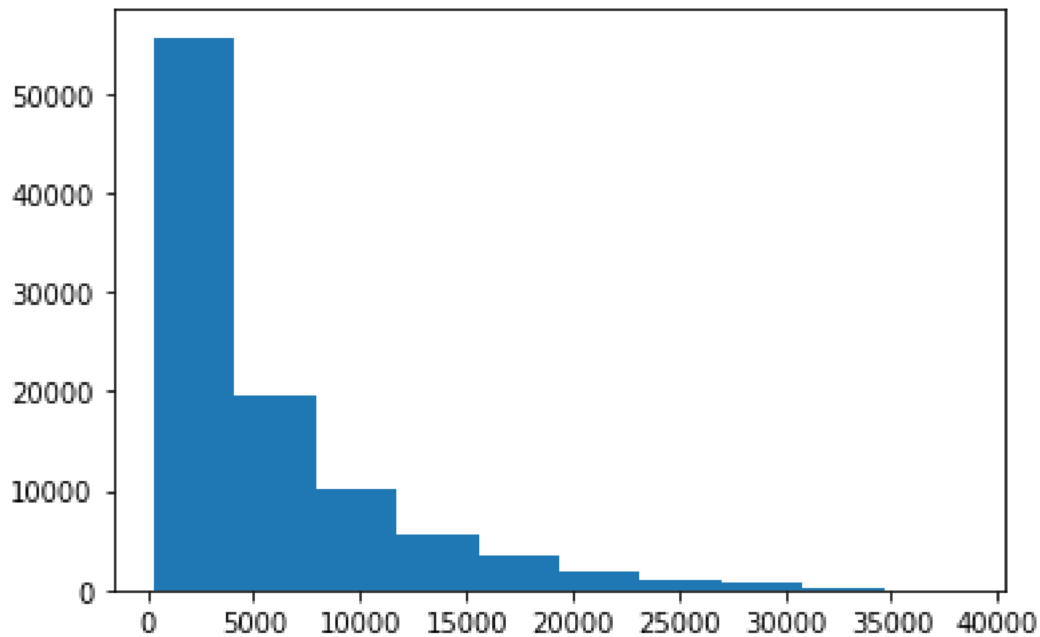
After getting 79% of R-squared score with KNN with 50 neighbors we suspect that maybe same neighbors number with 50000 sample would work better for all data.

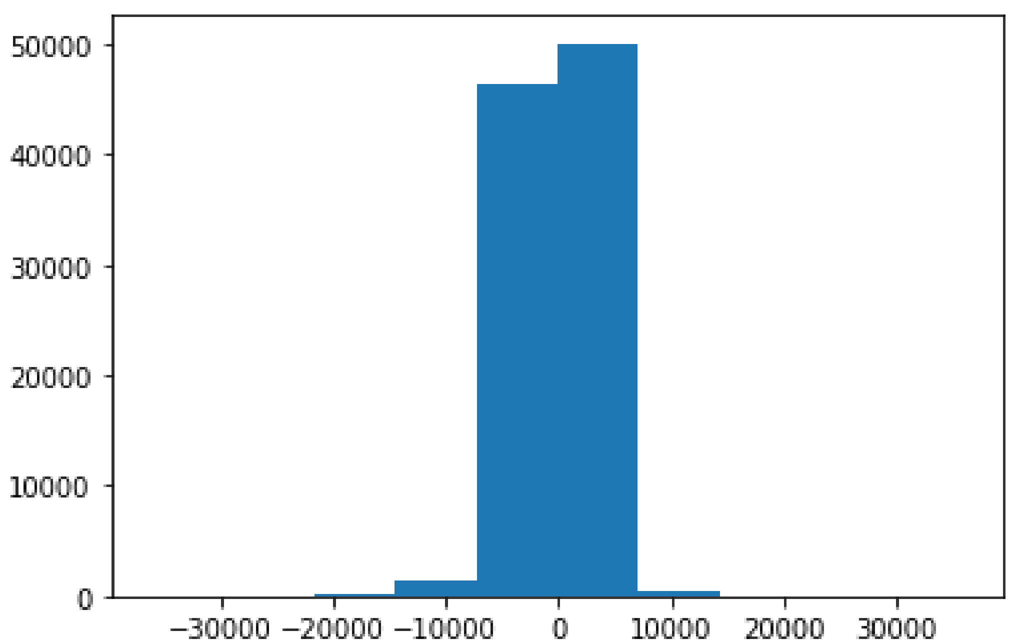
[90]:
```
x = dfimdm.drop(['price'], axis='columns')
y = dfimdm.price

x_train, x_test, y_train, y_testknn = train_test_split(x, y,test_size=0.35)


# Preprocessing - Scaling the features

scaler = MinMaxScaler(feature_range=(0, 1))

x_train_scaled = scaler.fit_transform(x_train)
x_train = pd.DataFrame(x_train_scaled)

x_test_scaled = scaler.fit_transform(x_test)
x_test = pd.DataFrame(x_test_scaled)

# classifier
model = neighbors.KNeighborsRegressor(n_neighbors = 6)

model.fit(x_train, y_train)  #fit the model
pred=model.predict(x_test) #make prediction on test set
error = sqrt(mean_squared_error(y_testknn,pred)) #calculate rmse
accur = metrics.r2_score(y_testknn.values, pred)
print('Mean squared error:', error)
print('R-squared score:', accur*100,"%")
```

```
Mean squared error: 0.44738439622778964
R-squared score: 83.6361413985903 %
```

Decreasing numbers of neighbors worked well and it increased R-squared score from 79% to 83%. We wanted to plot the predictions and check how KNN worked.

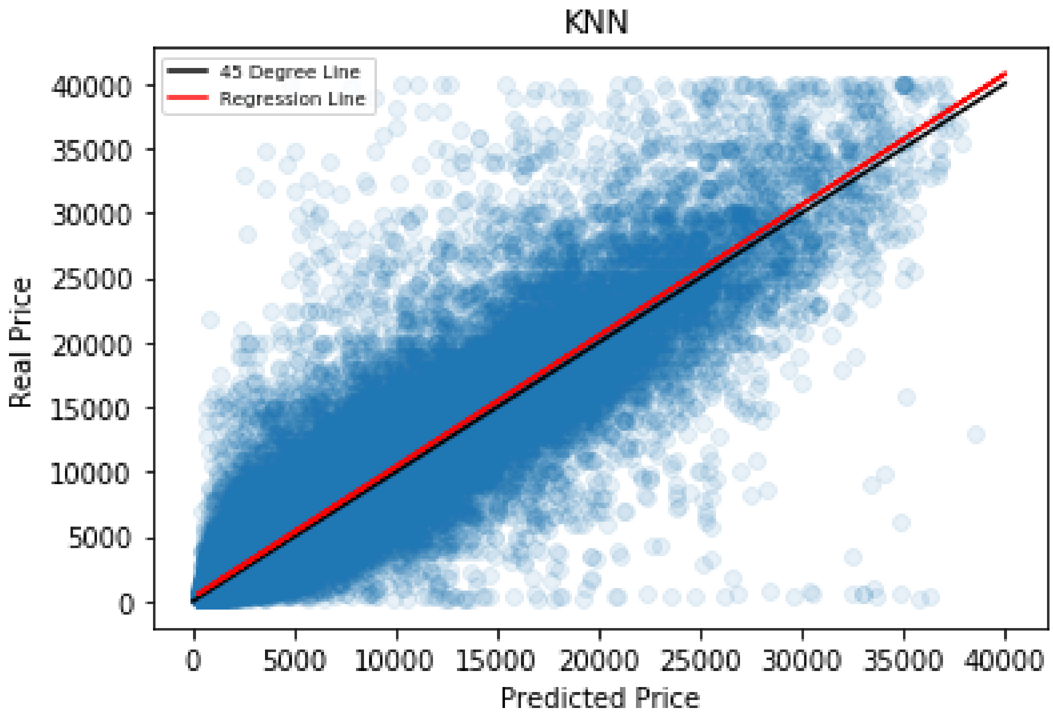
[91]:
```
plt.hist(np.exp(pred))
plt.show()
plt.hist(np.exp(pred) - np.exp(y_testknn))
plt.show()
```





In the first plot we can see how KNN prediction looks like and in the second plot we can see that the difference between predicted and real prices.

[92]:
```
plt.scatter(np.exp(pred), np.exp(y_testknn.values), alpha=0.1)
plt.xlabel('Predicted Price')
plt.ylabel('Real Price')
m, b = np.polyfit(np.exp(pred), np.exp(y_testknn.values), 1)
plt.plot([0, 40000], [0, 40000], color = 'black')
plt.plot(np.exp(y_testknn.values), m*np.exp(y_testknn.values) + b, 'red')
plt.legend(["45 Degree Line","Regression Line"], fontsize = "x-small")
plt.title("KNN")
plt.show()
```



This time looks like predictions became more accurate. Although we can see in general KNN under-price cars. Despite that it's very clear that it made better job than Linear Regression.

Before moving to the next method we want to check again how data without "model" would work with KNN.

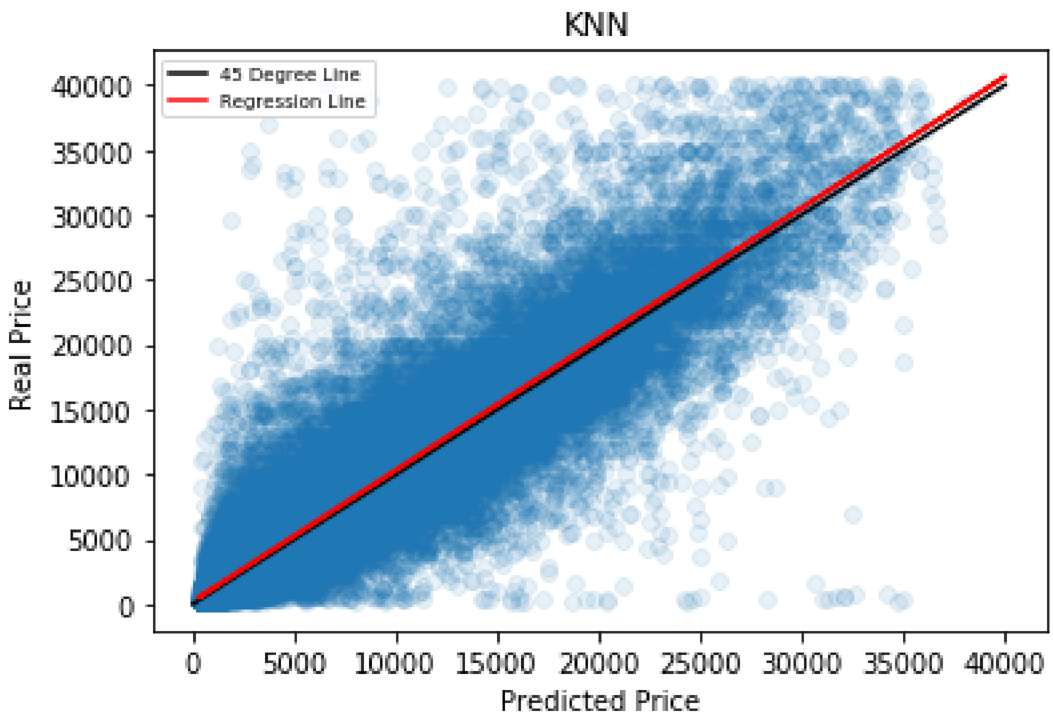
[93]:
```
x = dmNoModel.drop(['price'], axis='columns')
y = dmNoModel.price
x_train, x_test, y_train, y_test = train_test_split(x, y,test_size=0.35)


# Preprocessing - Scaling the features

scaler = MinMaxScaler(feature_range=(0, 1))
```

```
x_train_scaled = scaler.fit_transform(x_train)
x_train = pd.DataFrame(x_train_scaled)

x_test_scaled = scaler.fit_transform(x_test)
x_test = pd.DataFrame(x_test_scaled)

# classifier
modelNM = neighbors.KNeighborsRegressor(n_neighbors = 6)

modelNM.fit(x_train, y_train)  #fit the model
predNM=modelNM.predict(x_test) #make prediction on test set
errorNM = sqrt(mean_squared_error(y_test,predNM)) #calculate rmse
accurNM = metrics.r2_score(y_test.values, predNM)
print('Mean squared error:', errorNM)
print('R-squared score:', accurNM)
```

```
Mean squared error: 0.44659729097938095
R-squared score: 0.8360531864475668
```

[94]:

```
plt.scatter(np.exp(predNM), np.exp(y_test.values), alpha=0.1)
plt.xlabel('Predicted Price')
plt.ylabel('Real Price')
m, b = np.polyfit(np.exp(predNM), np.exp(y_test.values), 1)
plt.plot([0, 40000], [0, 40000], color = 'black')
plt.plot(np.exp(y_test.values), m*np.exp(y_test.values) + b, 'red')
plt.legend(["45 Degree Line","Regression Line"], fontsize = "x-small")
plt.title("KNN")
plt.show()
```



Unlike Linear Regression in KNN data without "model" worked similiar. Since KNN working with distance we are thinking "model" was causing overfitting.

4 Random Forest

Next method we used is random forest. Since we have plenty categories especially under brand and model we are thinking that random forest method will be great fit to our data.

Cross Validation (K-Fold) We split our data into 5 parts, with KFold. And iterate it 5 times, every time test and train data was changing as follows:

test-train-train-train-train
train-**test**-train-train-train
train-train-**test**-train-train
train-train-train-**test**-train
train-train-train-train-**test**

We used this method to be able get more from our data and see if R-squared score changes depending on the which part of data we are training and testing our model.

Since random forest is working with tree-based model we wanted to change the way we encode our data and try to make depth shorter.

[95]:
```
dfimrf = dfim.copy()
```

[96]:
```python
for col in ['vehicleType', 'gearbox', 'model', 'fuelType', 'brand']:
    le = preprocessing.LabelEncoder()
    le.fit(dfimrf[col].unique())
    dfimrf[col] = le.transform(dfimrf[col])
```

[97]:
```python
dfimrf.head()
```

[97]:
```
      price  vehicleType  gearbox   powerPS  model  kilometer  fuelType  \
2  9.190138            7        0  5.093750    118     125000         1
3  7.313220            5        1  4.317488    117     150000         6
4  8.188689            5        1  4.234107    102      90000         1
5  6.476972            3        1  4.624973     11     150000         6
6  7.696213            1        1  4.691348      8     150000         6

   brand  notRepairedDamage  daysBeforeSold  namelen  age
2     14                  0              22       30   13
3     37                  0               0       18   16
4     31                  0               6       30    9
5      2                  1               2       50   22
6     25                  0               4       27   13
```

[98]:
```python
# Create train and test set

kf = KFold(n_splits=5, shuffle=True, random_state=random.randint(0,10000))
ttscores = []
tnscores = []
errorsRF = []
k = 0
for train, test in kf.split(dfimrf.index.values):
    k += 1
    x = dfimrf.drop('price', axis=1)
    y = dfimrf['price']
    x_train = x.iloc[train]
    y_train = y.iloc[train]

    x_test = x.iloc[test]
    y_testrf = y.iloc[test]


    # classifier
    rfr = RandomForestRegressor()
    rfr.fit(x_train, y_train)
    tnscore = rfr.score(x_train, y_train)
    tnscores.append(tnscore)
    predRF = rfr.predict(x_test)
    ttscore = rfr.score(x_test, y_testrf)
    ttscores.append(ttscore)
```

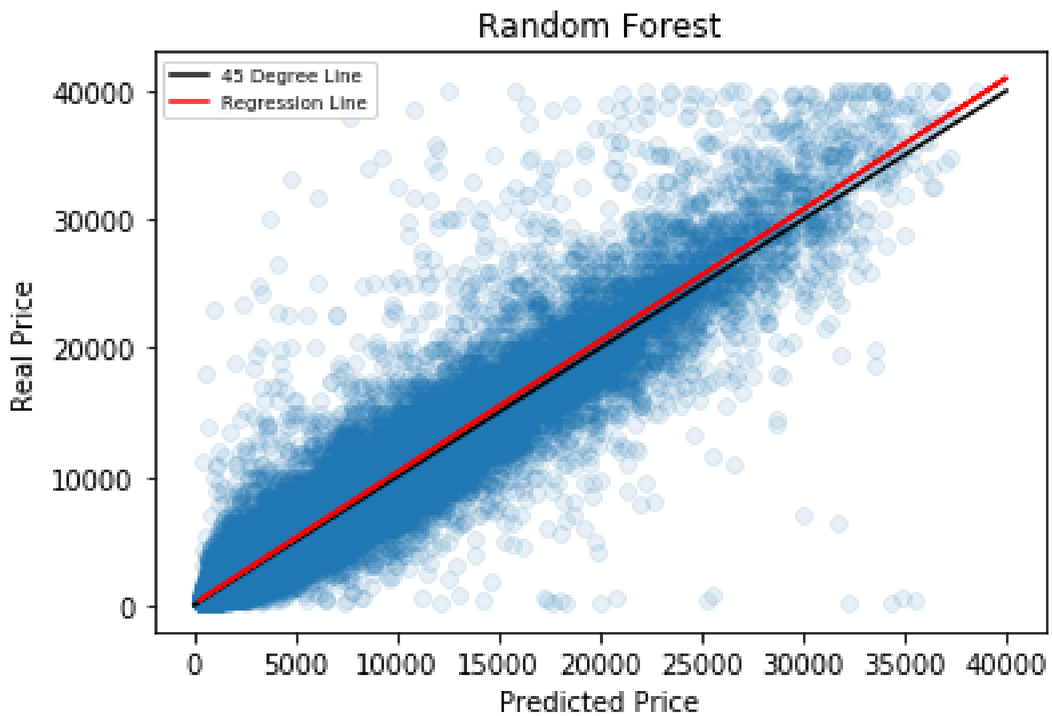
```
    errorRF = sqrt(mean_squared_error(y_testrf,predRF)) #calculate rmse
    errorsRF.append(errorRF)

    print("Iteration:", k)
    print('Mean squared error:', errorRF)
    print("Score on train dataset:", tnscore*100,"%")
    print("Score on test dataset:", ttscore*100,"%")
print("-------------------------------------------------------------")
print("Average score on train dataset:", np.mean(tnscores)*100,"%")
print("Average score on test dataset:", np.mean(ttscores)*100,"%")
print("Average mean squared error:", np.mean(errorsRF))
scoreRF = np.mean(ttscores)
predRF = rfr.predict(x_test)
```

```
Iteration: 1
Mean squared error: 0.38130632863129355
Score on train dataset: 98.17644246192957 %
Score on test dataset: 88.09088949726018 %
Iteration: 2
Mean squared error: 0.3828974997674672
Score on train dataset: 98.18472038271105 %
Score on test dataset: 88.07246176960331 %
Iteration: 3
Mean squared error: 0.3849012253238724
Score on train dataset: 98.18044549733833 %
Score on test dataset: 87.79987633126763 %
Iteration: 4
Mean squared error: 0.38119757851744873
Score on train dataset: 98.17228778364473 %
Score on test dataset: 88.12875074459188 %
Iteration: 5
Mean squared error: 0.38310438233143723
Score on train dataset: 98.16512373246707 %
Score on test dataset: 88.02976192026522 %
------------------------------------------------------------
Average score on train dataset: 98.17580397161817 %
Average score on test dataset: 88.02434805259765 %
Average mean squared error: 0.38268140291430386
```

The R-squared score didn't differ much among the data in 5 iteration. Although 88% R-squared score is impressive compare to Linear Regression and KNN methods. Let's see in the plots how predictions of the random forest model looks like.

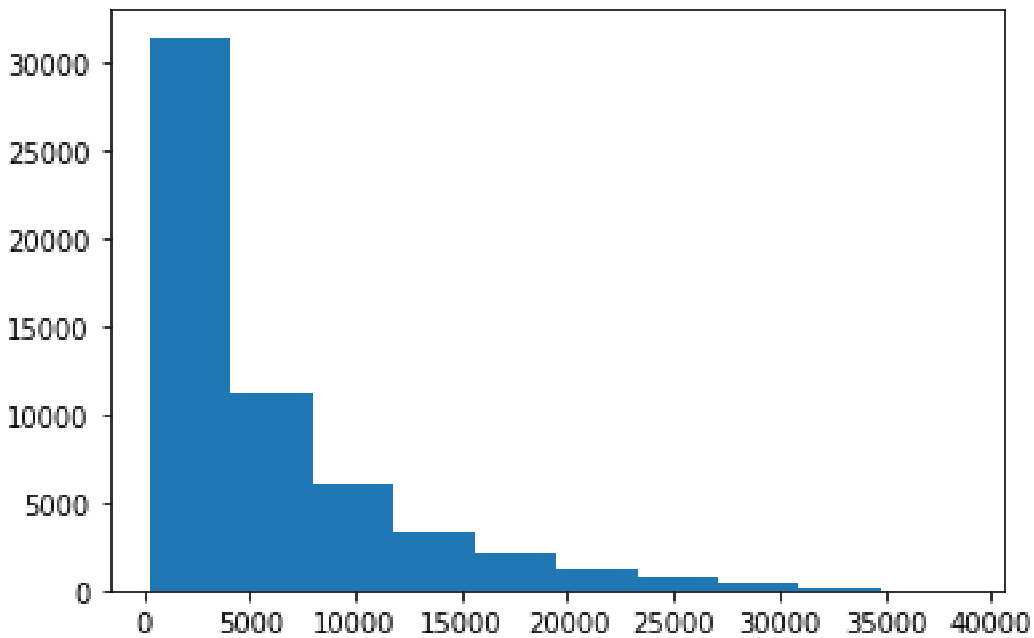
[99]:
```
plt.scatter(np.exp(predRF) ,np.exp(y_testrf), alpha=0.1)
plt.xlabel('Predicted Price')
plt.ylabel('Real Price')
m, b = np.polyfit(np.exp(predRF), np.exp(y_testrf), 1)
plt.plot([0, 40000], [0, 40000], color = 'black')
```

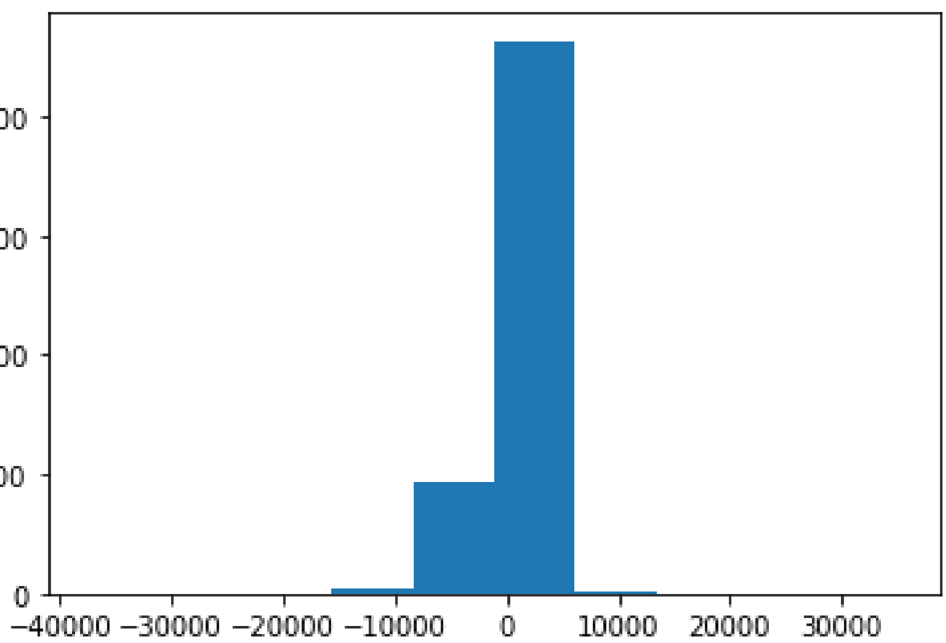
```
plt.plot(np.exp(y_testrf.values), m*np.exp(y_testrf.values) + b, 'red')
plt.legend(["45 Degree Line","Regression Line"], fontsize = "x-small")
plt.title("Random Forest")
plt.show()
```



It's clearly visible that this time model predict better. Of course there are some outliers like over and under pricing but in general the line became thiner and more visible.

[100]:
```
plt.hist(np.exp(predRF))
plt.show()
plt.hist(np.exp(predRF) - np.exp(y_testrf.values))
plt.show()
```





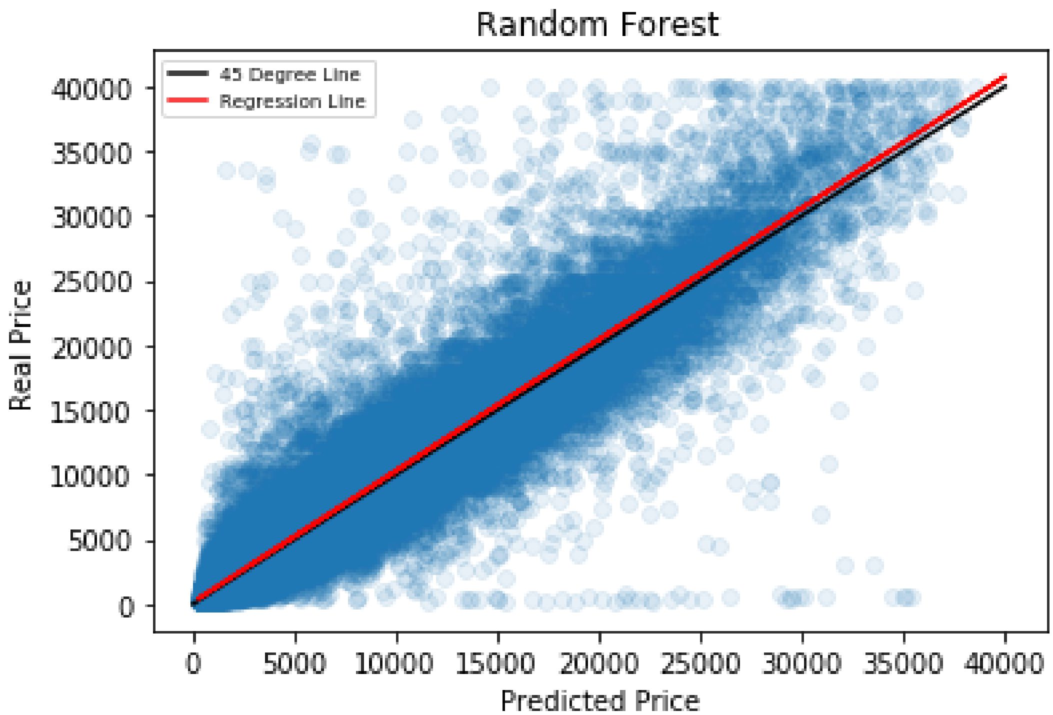
In the first plot we can see how Random Forest prediction looks like and in the second plot we can see that the difference between predicted and real prices.

[101]:
```
dfimNoModelrf = dfimNoModel.copy()
for col in ['vehicleType', 'gearbox', 'fuelType', 'brand']:
    le = preprocessing.LabelEncoder()
    le.fit(dfimNoModelrf[col].unique())
    dfimNoModelrf[col] = le.transform(dfimNoModelrf[col])

xNM = dfimNoModelrf.drop(['price'], axis='columns')
yNM = dfimNoModelrf.price
```

[102]:
```
x_train, x_test, y_train, y_test = train_test_split(xNM, yNM,test_size=0.35)

rfrNM = RandomForestRegressor()
rfrNM.fit(x_train, y_train)
predRFNM = rfrNM.predict(x_test)
tnscoreNM = rfrNM.score(x_train, y_train)
ttscoreNM = rfrNM.score(x_test, y_test)

errorRFNM = sqrt(mean_squared_error(y_test,predRFNM))

print('Mean squared error:', errorRFNM)
print("Score on train dataset:", tnscoreNM*100,"%")
print("Score on test dataset:", ttscoreNM*100,"%")
```

```
Mean squared error: 0.39518677236334177
Score on train dataset: 98.05167813414398 %
Score on test dataset: 87.26288219023658 %
```

[103]:
```
plt.scatter(np.exp(predRFNM), np.exp(y_test.values), alpha=0.1)
plt.xlabel('Predicted Price')
plt.ylabel('Real Price')
m, b = np.polyfit(np.exp(predRFNM), np.exp(y_test.values), 1)
plt.plot([0, 40000], [0, 40000], color = 'black')
plt.plot(np.exp(y_test.values), m*np.exp(y_test.values) + b, 'red')
plt.legend(["45 Degree Line","Regression Line"], fontsize = "x-small")
plt.title("Random Forest")
plt.show()
```



Despite R-squared score results looks same with and without “model” from plot above we can see that the line was thiner if it comes to predictions with “model”. So we can say that thanks to model in the data our errors are smaller.

4.1 Comparison of Methods and Conclusions

In this project we aimed to find the best regression model for used cars dataset to be able to predict used cars price.

Dataset was scraped with Scrapy from German eBay and there were some mistakes, duplications and outliers. As first step we removed them from the dataset and stayed with 79.9% of the original data.

There were 20 variables in the original dataset and we had to remove “dateCrawled”, “postalCode”, “abtest”, “offerType”, “nrOfPictures”, and “seller” because they were not usefull for our purpose. We created new features: * “daysBeforeSold” by using “dateCreated” and “lastSeen” * “namelen” by using “name” * “age” by using “yearOfRegistration”

After that we worked on missing values columnwise and impute what we could impute meaningfully and drop the rest. At the end we had 75.8% of the original data ready to use for price prediction.

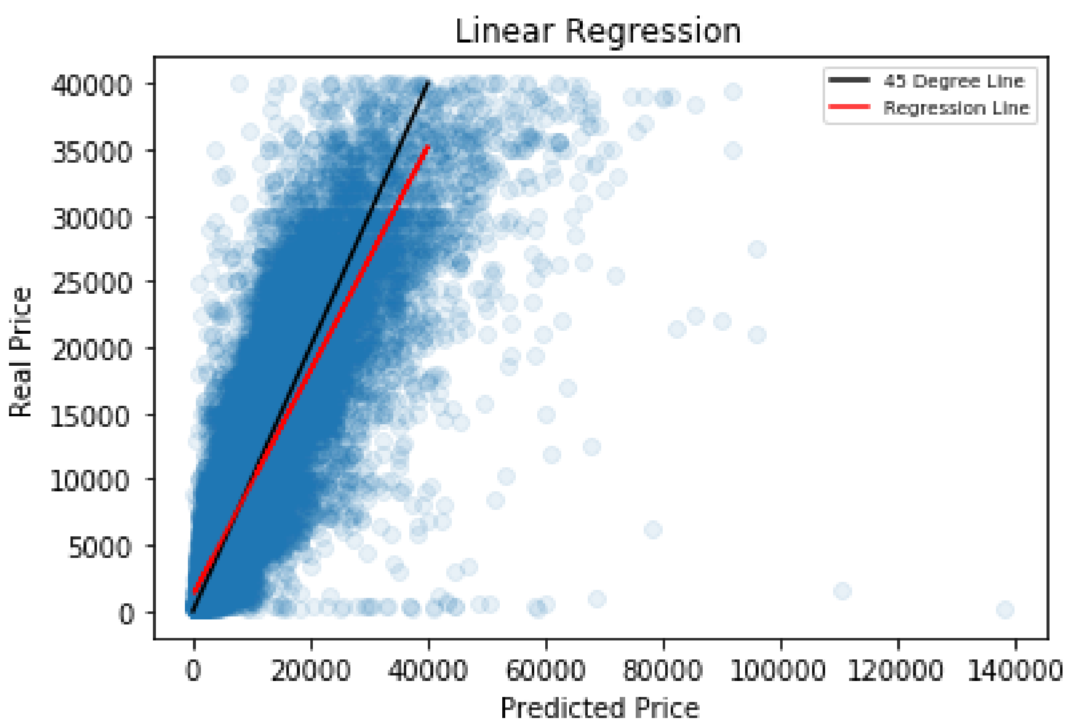
We used log-transformation in price and powerPs and one hot encoding for categorical variables to be able to perform machine learning algorithms.

We used Linear Model, K Nearest Neighbor and Random Forest. The most successul one was Random Forest with 88.01% R-squared score. Followed by KNN with 83.70% and the worst one was as expected Linear Regression with 78.67%.

4.1.1 Linear Regression

Linear regression R-squared score was 78% but it overshoot for some cars and cause outliers also underpricing was very common. Despite 40k is maximum price in our dataset Linear Regression model made some predictions even 160k. Beside that the line was very thick so those predictions were not good.
We also tried to run Linear Regression without "model" variable to see the effects of it. It gave 3% less accuarcy on the dataset. For Linear Regression full data gave better results.

[104]:
```
plt.scatter(np.exp(y_predLR), np.exp(y_testLR.values), alpha=0.1)
plt.xlabel('Predicted Price')
plt.ylabel('Real Price')
m, b = np.polyfit(np.exp(y_predLR), np.exp(y_testLR.values), 1)
plt.plot([0, 40000], [0, 40000], color = 'black')
plt.plot(np.exp(y_testLR.values), m*np.exp(y_testLR.values) + b, 'red')
plt.legend(["45 Degree Line","Regression Line"], fontsize = "x-small")
plt.title("Linear Regression")
plt.show()
```



[105]:
```
print("Linear Regression R-squared score(with model):", scoreLR*100, "%")
print("Linear Regression R-squared score(without model):", scoreLRnm*100, "%")
```

```
Linear Regression R-squared score(with model): 78.58930520058853 %
Linear Regression R-squared score(without model): 75.69693562079023 %
```

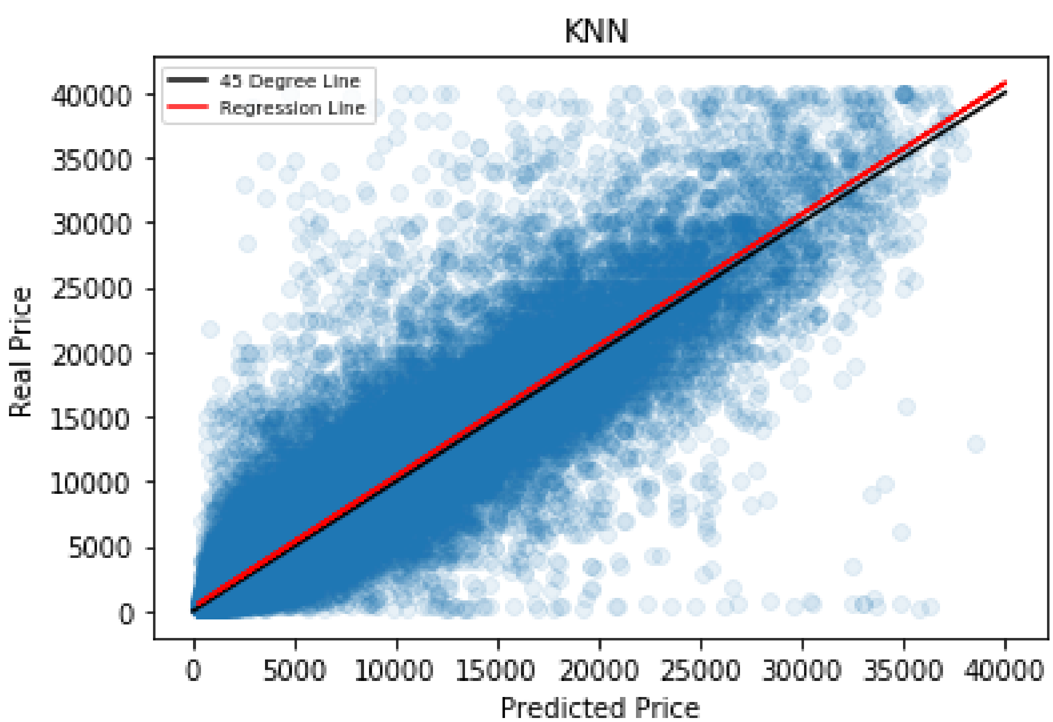
4.1.2 KNN

Since KNN is a Distance-Based algorithm we needed to start with scaling all variables. After data was ready next step was chosing how many neighbors to use. We decided to check one by one what are the mean squared errors and the R-squared score in the sample we took from data.

After running test on the sample we run the KNN with 6 neighbors for all data, results were relatively better. And as usual we fit the KNN with data without “model” and this time results were similar.

Since predictions were same with the data with and without “model”, below plot is with “model”.

[106]:
```
plt.scatter(np.exp(pred), np.exp(y_testknn.values), alpha=0.1)
plt.xlabel('Predicted Price')
plt.ylabel('Real Price')
m, b = np.polyfit(np.exp(pred), np.exp(y_testknn.values), 1)
plt.plot([0, 40000], [0, 40000], color = 'black')
plt.plot(np.exp(y_testknn.values), m*np.exp(y_testknn.values) + b, 'red')
plt.legend(["45 Degree Line","Regression Line"], fontsize = "x-small")
plt.title("KNN")
plt.show()
```



[107]:
```
print("KNN R-squared score(with model):", accur*100, "%")
print("KNN R-squared score(without model):", accurNM*100, "%")
```

```
KNN R-squared score(with model): 83.6361413985903 %
```

```
KNN R-squared score(without model): 83.60531864475668 %
```

4.1.3 Random Forest

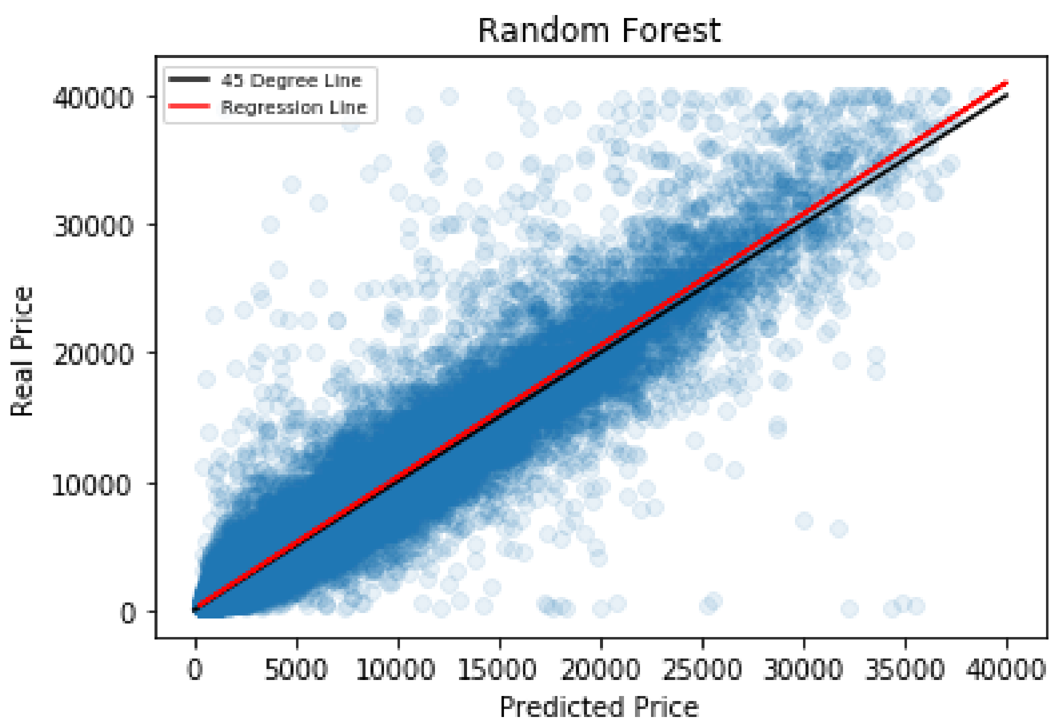
Last method we used was Random forest. We changed the way of encoding the data for this method in order to decrease depth of trees shorter.

We used KFold method to split our data to be able get more from it and see if R-squared score changes depending on the which part of data we are training and testing our model.

Results were better than Linear Regression and KNN.

And as usual we fit the KNN with data without "model" and this time R-squared score results were same. Although in without "model" prediction plot we realized that the line was thicker than predictions with "model". So we can say that thanks to "model" in the data errors were smaller. For Random Forest data with "model" was better that's why as a final plot belongs to that.

[108]:
```
plt.scatter(np.exp(predRF) ,np.exp(y_testrf), alpha=0.1)
plt.xlabel('Predicted Price')
plt.ylabel('Real Price')
m, b = np.polyfit(np.exp(predRF), np.exp(y_testrf), 1)
plt.plot([0, 40000], [0, 40000], color = 'black')
plt.plot(np.exp(y_testrf.values), m*np.exp(y_testrf.values) + b, 'red')
plt.legend(["45 Degree Line","Regression Line"], fontsize = "x-small")
plt.title("Random Forest")
plt.show()
```



[109]:
```python
print("Random Forest R-squared score(with model):", scoreRF*100, "%")
print("Random Forest R-squared score(without model):", ttscoreNM*100, "%")
```

```
Random Forest R-squared score(with model): 88.02434805259765 %
Random Forest R-squared score(without model): 87.26288219023658 %
```

Random Forest(with model) gave the best result 88%. For this dataset Random Forest perform in a best way in prediction and also computational power and time. We are thinking that it is the best fit among all 3 methods we used.

Despite the quality of the dataset we believe that results were pretty outstanding.