Data Cleaning and Analysis of Classifieds eBay Used Car Selling Platform 'Kleinanzeigen'

Author : Kenneth Lucas Kusima
Attributor: DataQuest

Project Origin:

This project was made per instructions from:
The DataQuest Course: Pandas and Numpy Fundamentals

Objectives:

The aim is to clean the raw data from the website and analyse the resulting used car listings. Python's Pandas and NumPy libraries will be used to perform the necessary dataset manipulation so that sensible and useful results can be observed. The dataset used contains only the sampled 50,000 data points from the full dataset available. This is so as to minimize the run time and still sucessful implement the fundamental data cleaning techniques necessary for any size datasets.

Link : https://www.kaggle.com/orgesleka/used-cars-database/data (https://www.kaggle.com/orgesleka/used-cars-database/data)

In [249]:

```
import numpy as np
import pandas as pd

#Invoking the raw data
autos = pd.read_csv("autos.csv", encoding = "Latin-1")
```

In [250]:

```
autos #The dataframe that reads/translates csv file (The dataset to be used)
```

Out[250]:

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

In [251]:

```
autos.info() #Informational snapshot of the features behind our dataset
autos.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 20 columns):
dateCrawled             50000 non-null object
name                    50000 non-null object
seller                  50000 non-null object
offerType               50000 non-null object
price                   50000 non-null object
abtest                  50000 non-null object
vehicleType             44905 non-null object
yearOfRegistration      50000 non-null int64
gearbox                 47320 non-null object
powerPS                 50000 non-null int64
model                   47242 non-null object
odometer                50000 non-null object
monthOfRegistration     50000 non-null int64
fuelType                45518 non-null object
brand                   50000 non-null object
notRepairedDamage       40171 non-null object
dateCreated             50000 non-null object
nrOfPictures            50000 non-null int64
postalCode              50000 non-null int64
lastSeen                50000 non-null object
dtypes: int64(5), object(15)
memory usage: 7.6+ MB
```

Out[251]:

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

As seen from the outlines and description above the dataset in questions contains a combination of object and int64 datatypes with varying number of non-null entries. In addition, formating of the 20 columns appears inconsisten with vague nomenclature. In detail we can make the following observations: we can make the following observations:

- The dataset contains 20 columns, most of which are strings.

- Some columns have null values, but none have more than ~20% null values.
- The column names use camelcase instead of Python's preferred snakecase, which means we can't just replace spaces with underscores.

Editing specific column names:

In [252]:

```
autos.columns
```

Out[252]:

```
Index(['dateCrawled', 'name', 'seller', 'offerType', 'price', 'abtes
t',
       'vehicleType', 'yearOfRegistration', 'gearbox', 'powerPS', 'mod
el',
       'odometer', 'monthOfRegistration', 'fuelType', 'brand',
       'notRepairedDamage', 'dateCreated', 'nrOfPictures', 'postalCod
e',
       'lastSeen'],
      dtype='object')
```

In [253]:

```
def clean_column(col):
    #col = col.strip()
    col = col.replace("yearOfRegistration","registration_year")
    col = col.replace("monthOfRegistration", "registration_month")
    col = col.replace("notRepairedDamage", "unrepaired_damage")
    col = col.replace("dateCreated","ad_created" )
    col = col.replace("nrOfPictures","number_of_pictures")

    if col.islower() == False:
        col = col.replace("T","_t")
        col = col.replace("C","_c")
        col = col.replace("PS","_ps")
        col = col.replace("S","_s")
    col = col.lower()
    return col

new_columns = []
for c in autos.columns:
    new = clean_column(c)
    new_columns.append(new)

autos.columns = new_columns
autos.columns
```

Out[253]:

```
Index(['date_crawled', 'name', 'seller', 'offer_type', 'price', 'abtes
t',
       'vehicle_type', 'registration_year', 'gearbox', 'power_ps', 'mo
del',
       'odometer', 'registration_month', 'fuel_type', 'brand',
       'unrepaired_damage', 'ad_created', 'number_of_pictures', 'posta
l_code',
       'last_seen'],
      dtype='object')
```

Further exploration

In [254]:

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

Out[254]:

	date_crawled	name	seller	offer_type	price	abtest	vehicle_type	registration_yea
count	50000	50000	50000	50000	50000	50000	44905	50000.00000
unique	48213	38754	2	2	2357	2	8	Na
top	2016-03-23 18:39:34	Ford_Fiesta	privat	Angebot	$0	test	limousine	Na
freq	3	78	49999	49999	1421	25756	12859	Na
mean	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2005.07328
std	NaN	NaN	NaN	NaN	NaN	NaN	NaN	105.71281
min	NaN	NaN	NaN	NaN	NaN	NaN	NaN	1000.00000
25%	NaN	NaN	NaN	NaN	NaN	NaN	NaN	1999.00000
50%	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2003.00000
75%	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2008.00000
max	NaN	NaN	NaN	NaN	NaN	NaN	NaN	9999.00000

In [255]:

```
print(autos["number_of_pictures"].head())
print(autos["number_of_pictures"].describe())
```

```
0    0
1    0
2    0
3    0
4    0
Name: number_of_pictures, dtype: int64
count    50000.0
mean         0.0
std          0.0
min          0.0
25%          0.0
50%          0.0
75%          0.0
max          0.0
Name: number_of_pictures, dtype: float64
```

Since number_of_pictures contains 5000 empty entries, and the seller and offer_types contain only 2 entries, it is acceptable to remove these two columns

In [256]:

```
autos = autos.drop(["number_of_pictures", "seller", "offer_type"], axis=1)
```

In [257]:

```
print(autos["price"].head())
print(autos["price"].describe())
```

```
0    $5,000
1    $8,500
2    $8,990
3    $4,350
4    $1,350
Name: price, dtype: object
count     50000
unique     2357
top          $0
freq       1421
Name: price, dtype: object
```

In [258]:

```
print(autos["odometer"].head())
print(autos["odometer"].describe())
```

```
0    150,000km
1    150,000km
2     70,000km
3     70,000km
4    150,000km
Name: odometer, dtype: object
count         50000
unique           13
top       150,000km
freq          32424
Name: odometer, dtype: object
```

From the data above, the "price" and "odometer" categories are text entries representing numerical values. It is then neccessary to convert them to numerical entries so as to allow for easier data manipulation. To do so, non-numeric characters like '$' and 'km' and ',' will have to be removed from the corresponding categories. In additon, it was also found that the "number_of_pictures" contains no values. This raises suspicion as it would mean that none of the eBay entries had pictures for the cars published.

In addition the date columns can also be seen to contain incosistent formatting. The "date_crawled", "last_seen", and "ad_created" columnns contain text data thus preventing statistical inferrence.

Removing the non-numeric characters and changing from text to numeric

In [259]:

```
autos["odometer"] = autos["odometer"].str.replace("km","")
autos["odometer"] = autos["odometer"].str.replace(",","")
autos["odometer"] = autos["odometer"].astype(int)

print(autos["odometer"].head())
```

```
0    150000
1    150000
2     70000
3     70000
4    150000
Name: odometer, dtype: int64
```

In [260]:

```
autos["price"] = autos["price"].str.replace("$","")
autos["price"] = autos["price"].str.replace(",","")
autos["price"] = autos["price"].astype(int)

print(autos["price"].head())
```

```
0    5000
1    8500
2    8990
3    4350
4    1350
Name: price, dtype: int64
```

Renaming the columns accordingly:

In [261]:

```
autos.rename({"odometer":"odometer_km", "price":"price_$"}, axis = 1, inplace =True)
```

Analysing price columnn

In [262]:

```
autos["price_$"].value_counts().sort_index(ascending=False).head(20)
```

Out[262]:

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

Since eBay is an auction site, it remains plausible for items to exist at $\$1$ however the data above shows steady increase in price until $\$350000$ is reached of which is followed by a jump in prices. It is therefore necessary to limit the price range from $\$1$ to $\$350000$. To do this, the .between() method will be used.

In [263]:

```
autos = autos[autos["price_$"].between(1,350001)]
autos["price_$"].describe()
```

Out[263]:

```
count     48565.000000
mean       5888.935591
std        9059.854754
min           1.000000
25%        1200.000000
50%        3000.000000
75%        7490.000000
max      350000.000000
Name: price_$, dtype: float64
```

Cleaning the Date Columns

There are 5 columns that should represent date values. Some of these columns were created by the crawler, some came from the website itself. Note that here crawling refers to the systematic visiting a website to create an index of data. By referring to the data dictionary it can be seen that:

- `date_crawled` : added by the crawler

- `last_seen` : added by the crawler
- `ad_created` : from the website
- `registration_month` : from the website
- `registration_year` : from the website

The "date_crawled", "last_seen", and "ad_created" columnns were found to have contain text values and hence need to be converted to numerical values. Before this, an understanding of the format has to be built.

In [264]:

```
(autos[['date_crawled','ad_created','last_seen']][0:5])
```

Out[264]:

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

We can see that the first 10 characters represent the day in the format "YYYY-MM-DD" followed by the 8 character 24 hr time in the format " Hr:Min:Sec". Knowing this, we can create a distribution of the respective day as a percentage:

In [265]:

```
autos["date_crawled"].str[:10].value_counts(normalize=True, dropna=False).sort_index
```

Out[265]:

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

The site appears to have been crawled on a daily basis over a 1 to 2 (March-April) month period. It effectively shows the crawler's systematic cycle period.

In [266]:

```
autos["ad_created"].str[:10].value_counts(normalize=True, dropna=False).sort_index()
```

Out[266]:

```
2015-06-11    0.000021
2015-08-10    0.000021
2015-09-09    0.000021
2015-11-10    0.000021
2015-12-05    0.000021
2015-12-30    0.000021
2016-01-03    0.000021
2016-01-07    0.000021
2016-01-10    0.000041
2016-01-13    0.000021
2016-01-14    0.000021
2016-01-16    0.000021
2016-01-22    0.000021
2016-01-27    0.000062
2016-01-29    0.000021
2016-02-01    0.000021
2016-02-02    0.000041
2016-02-05    0.000041
2016-02-07    0.000021
2016-02-08    0.000021
2016-02-09    0.000021
2016-02-11    0.000021
2016-02-12    0.000041
2016-02-14    0.000041
2016-02-16    0.000021
2016-02-17    0.000021
2016-02-18    0.000041
2016-02-19    0.000062
2016-02-20    0.000041
2016-02-21    0.000062
                ...
2016-03-09    0.033151
2016-03-10    0.031895
2016-03-11    0.032904
2016-03-12    0.036755
2016-03-13    0.017008
2016-03-14    0.035190
2016-03-15    0.034016
2016-03-16    0.030125
2016-03-17    0.031278
2016-03-18    0.013590
2016-03-19    0.033687
2016-03-20    0.037949
2016-03-21    0.037579
2016-03-22    0.032801
2016-03-23    0.032060
2016-03-24    0.029280
2016-03-25    0.031751
2016-03-26    0.032266
2016-03-27    0.030989
2016-03-28    0.034984
2016-03-29    0.034037
2016-03-30    0.033501
2016-03-31    0.031875
2016-04-01    0.033687
```

```
2016-04-02    0.035149
2016-04-03    0.038855
2016-04-04    0.036858
2016-04-05    0.011819
2016-04-06    0.003253
2016-04-07    0.001256
Name: ad_created, Length: 76, dtype: float64
```

The results above expectedly show great in the last days seen. It is observed that most of entires are appearing 1-2 months within the expected range (March-April) and a smaller distribution is still experienced up to 9 months away from the expected period.

In [267]:

```
autos["last_seen"].str[:10].value_counts(normalize=True, dropna=False).sort_index()
```

Out[267]:

```
2016-03-05    0.001071
2016-03-06    0.004324
2016-03-07    0.005395
2016-03-08    0.007413
2016-03-09    0.009595
2016-03-10    0.010666
2016-03-11    0.012375
2016-03-12    0.023783
2016-03-13    0.008895
2016-03-14    0.012602
2016-03-15    0.015876
2016-03-16    0.016452
2016-03-17    0.028086
2016-03-18    0.007351
2016-03-19    0.015834
2016-03-20    0.020653
2016-03-21    0.020632
2016-03-22    0.021373
2016-03-23    0.018532
2016-03-24    0.019767
2016-03-25    0.019211
2016-03-26    0.016802
2016-03-27    0.015649
2016-03-28    0.020859
2016-03-29    0.022341
2016-03-30    0.024771
2016-03-31    0.023783
2016-04-01    0.022794
2016-04-02    0.024915
2016-04-03    0.025203
2016-04-04    0.024483
2016-04-05    0.124761
2016-04-06    0.221806
2016-04-07    0.131947
Name: last_seen, dtype: float64
```

This data shows that the last seen times, of which were recorded by the crawler, had the expectd range of date points. This allows us to further use the information above to predict when the postings were removes and hence, when the cars were sold. Nevertheless, significant jumps in distribution occur at the end dates. This can be attributed as a result of the crawling period ending.

In [268]:

```
autos["registration_year"].describe()
```

Out[268]:

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

The results shows that the average registration year of the cars posted was 2005. This helps in identifying what age of cars to expect from the listings. Moreover, the data shows peculiar results for minimum and maximum car registration years, that is, 1000 and 9999 respectively. These values are erroneous given the fact that cars did not exist in the year 1000 and car registration can be estimated to have become in the first few decades of the 1900s. Furthermore, it remains impossible to have a car registerd in a future year (9999) after the listing was seen i.e 2016. The correctly entered listings are therefore for those cars with registartion year in the range 1900 - 2016.

In [269]:

```
#Removing
year_range = (autos["registration_year"]  > 1900) & (autos["registration_year"]  < 2
autos = autos.loc[year_range]

#Calculating the distribution of listings with cars that fall inside the 1900-2016 i
print(autos["registration_year"].value_counts(normalize=True).head(10))
```

```
2000    0.067608
2005    0.062895
1999    0.062060
2004    0.057904
2003    0.057818
2006    0.057197
2001    0.056468
2002    0.053255
1998    0.050620
2007    0.048778
Name: registration_year, dtype: float64
```

Results show most cars to have been registered within the past 20 years.

Understanding the Brand Column

In [270]:

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

Out[270]:

```
volkswagen        0.211264
bmw               0.110045
opel              0.107581
mercedes_benz     0.096463
audi              0.086566
ford              0.069900
renault           0.047150
peugeot           0.029841
fiat              0.025642
seat              0.018273
skoda             0.016409
nissan            0.015274
mazda             0.015188
smart             0.014160
citroen           0.014010
toyota            0.012703
hyundai           0.010025
sonstige_autos    0.009811
volvo             0.009147
mini              0.008762
mitsubishi        0.008226
honda             0.007840
kia               0.007069
alfa_romeo        0.006641
porsche           0.006127
suzuki            0.005934
chevrolet         0.005698
chrysler          0.003513
dacia             0.002635
daihatsu          0.002506
jeep              0.002271
subaru            0.002142
land_rover        0.002099
saab              0.001649
jaguar            0.001564
daewoo            0.001500
trabant           0.001392
rover             0.001328
lancia            0.001071
lada              0.000578
Name: brand, dtype: float64
```

The most common brand appears to be volkswagen. An expected find given that this is a german manufacturer. It represents approximately 21% of the car brands in the listings.

Analysing most common brands

The criteris to be used for common brands are those with representing at least 5% of the total car listing brand distribution

In [271]:

```
cars = autos["brand"].value_counts(normalize = True)
common_index = cars > 0.05
brands = cars[common_index].index
```

Analysing which of the most common brands has the lowest average price :

In [272]:

```
brand_price = {}

for bnd in brands:
    single = autos[autos["brand"] == bnd] #Creates a set of entries with the brand 
    mean_price = single["price_$"].mean()
    brand_price[bnd] = int(mean_price)

brand_price
```

Out[272]:

```
{'volkswagen': 5402,
 'bmw': 8332,
 'opel': 2975,
 'mercedes_benz': 8628,
 'audi': 9336,
 'ford': 3749}
```

Result above show that on average the cheapest car is the 'opel' and the most expensive car is the 'audi'

Analysing which of the mean registration year per most common brand:

In [273]:

```
brand_reg_year = {}

for bnd in brands:
    single = autos[autos["brand"] == bnd] #Creates a set of entries with the brand 
    mean_price = single["registration_year"].mean()
    brand_reg_year[bnd] = int(mean_price)

brand_reg_year
```

Out[273]:

```
{'volkswagen': 2002,
 'bmw': 2003,
 'opel': 2002,
 'mercedes_benz': 2002,
 'audi': 2004,
 'ford': 2002}
```

Result above show that the audi has on average the most recent cars on the listing and the mercedez benz has the oldest.

Analysing Mileage

In [274]:

```
bmp_series = pd.Series(brand_price)
print(bmp_series)
```

```
volkswagen        5402
bmw               8332
opel              2975
mercedes_benz     8628
audi              9336
ford              3749
dtype: int64
```

In [275]:

```
df = pd.DataFrame(bmp_series, columns=['mean_price'])
df
```

Out[275]:

	mean_price
volkswagen	5402
bmw	8332
opel	2975
mercedes_benz	8628
audi	9336
ford	3749

In [276]:

```
mean_mileage_dict = {}
brand_mean_mileage = {}

for brand in brands:
    brand_only = autos[autos["brand"] == brand]
    mean_mileage = brand_only["odometer_km"].mean()
    mean_mileage_dict[brand] = int(mean_mileage)

mean_mileage = pd.Series(mean_mileage_dict).sort_values(ascending=False)
mean_prices = pd.Series(brand_price).sort_values(ascending=False)
```

In [277]:

```
brand_info = pd.DataFrame(mean_mileage,columns=['mean_mileage'])
brand_info
```

Out[277]:

	mean_mileage
bmw	132572
mercedes_benz	130788
opel	129310
audi	129157
volkswagen	128707
ford	124266

In [278]:

```
brand_info["mean_price"] = mean_prices
brand_info
```

Out[278]:

	mean_mileage	mean_price
bmw	132572	8332
mercedes_benz	130788	8628
opel	129310	2975
audi	129157	9336
volkswagen	128707	5402
ford	124266	3749

Side by side comparison allows us to see that across the most common brands, the mean milage does not vary nearly as much as the mean price. Furthermore, the more expensive cars (like audi, bmw and mercedez benz) are seen to have higher mean mileage