Pricing prediction

Goal: Develop a machine learning model for used cars market value determination.

Requirements: high accuracy, low execution time

Data Preprocessing ¶

Upload the data and the necessary libraries

In [1]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from math import sqrt
from sklearn.linear_model import Ridge
import lightgbm as lgb
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import GridSearchCV
import warnings
warnings.filterwarnings('ignore')
import sys
!conda install --yes --prefix {sys.prefix} plotly
from sklearn.metrics import make_scorer
import plotly.express as px
import matplotlib.pyplot as plt
```

```
Collecting package metadata (current_repodata.json): ...working... done
Solving environment: ...working... done

# All requested packages already installed.
```

In [2]:
```
#df = pd.read_csv('/datasets/autos.csv')
df = pd.read_csv(r'C:\Users\HP\Downloads\autos.csv')
```

In [3]:
```
df.head()
```

Out[3]:

	DateCrawled	Price	VehicleType	RegistrationYear	Gearbox	Power	Model	Kilometer	Registrat
0	2016-03-24 11:52:17	480	NaN	1993	manual	0	golf	150000	
1	2016-03-24 10:58:45	18300	coupe	2011	manual	190	NaN	125000	
2	2016-03-14 12:52:21	9800	suv	2004	auto	163	grand	125000	
3	2016-03-17 16:54:04	1500	small	2001	manual	75	golf	150000	
4	2016-03-31 17:25:20	3600	small	2008	manual	69	fabia	90000	

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

Out[4]:

	Price	RegistrationYear	Power	Kilometer	RegistrationMonth	NumberO
count	354369.000000	354369.000000	354369.000000	354369.000000	354369.000000	
mean	4416.656776	2004.234448	110.094337	128211.172535	5.714645	
std	4514.158514	90.227958	189.850405	37905.341530	3.726421	
min	0.000000	1000.000000	0.000000	5000.000000	0.000000	
25%	1050.000000	1999.000000	69.000000	125000.000000	3.000000	
50%	2700.000000	2003.000000	105.000000	150000.000000	6.000000	
75%	6400.000000	2008.000000	143.000000	150000.000000	9.000000	
max	20000.000000	9999.000000	20000.000000	150000.000000	12.000000	

Negative values are not found. Check for duplicates:

In [5]:
```
df.duplicated().sum()
```

Out[5]: 4

In [6]:
```
df[df.duplicated(keep=False)]
```

Out[6]:

	DateCrawled	Price	VehicleType	RegistrationYear	Gearbox	Power	Model	Kilometer	Reg
41529	2016-03-18 18:46:15	1999	wagon	2001	manual	131	passat	150000	
88087	2016-03-08 18:42:48	1799	coupe	1999	auto	193	clk	20000	
90964	2016-03-28 00:56:10	1000	small	2002	manual	83	other	150000	
171088	2016-03-08 18:42:48	1799	coupe	1999	auto	193	clk	20000	
187735	2016-04-03 09:01:15	4699	coupe	2003	auto	218	clk	125000	
231258	2016-03-28 00:56:10	1000	small	2002	manual	83	other	150000	
258109	2016-04-03 09:01:15	4699	coupe	2003	auto	218	clk	125000	
325651	2016-03-18 18:46:15	1999	wagon	2001	manual	131	passat	150000	

In [7]:
```
df = df.drop_duplicates()
```

Duplicates were deleted

In [8]:
```
df.isnull().mean()
```

Out[8]:
```
DateCrawled          0.000000
Price                0.000000
VehicleType          0.105795
RegistrationYear     0.000000
Gearbox              0.055968
Power                0.000000
Model                0.055607
Kilometer            0.000000
RegistrationMonth    0.000000
FuelType             0.092828
Brand                0.000000
NotRepaired          0.200793
DateCreated          0.000000
NumberOfPictures     0.000000
PostalCode           0.000000
LastSeen             0.000000
dtype: float64
```

Some features have missing values. Feature 'NotRepaired' has the greatest amount of missing values. Since for missing values there is no information whether a car was repaired or not, missing values are filled in with 'unknown'.

In [9]:
```
df['NotRepaired'].unique()
```

Out[9]:
```
array([nan, 'yes', 'no'], dtype=object)
```

In [10]:
```
df['NotRepaired'] = df['NotRepaired'].fillna(value='unknown')
```

In [11]:
```
df['NotRepaired'].unique()
```

Out[11]:
```
array(['unknown', 'yes', 'no'], dtype=object)
```

In [12]:
```
df.isnull().mean()
```

Out[12]:
```
DateCrawled          0.000000
Price                0.000000
VehicleType          0.105795
RegistrationYear     0.000000
Gearbox              0.055968
Power                0.000000
Model                0.055607
Kilometer            0.000000
RegistrationMonth    0.000000
FuelType             0.092828
Brand                0.000000
NotRepaired          0.000000
DateCreated          0.000000
NumberOfPictures     0.000000
PostalCode           0.000000
LastSeen             0.000000
dtype: float64
```

There are no missing values in feature 'NotRepaired' left. Check rows with missing values in 'Model' to figure out the way to fill in missing values

In [13]:
```
df[df['Model'].isna()]
```

Out[13]:

	DateCrawled	Price	VehicleType	RegistrationYear	Gearbox	Power	Model	Kilometer	Re
1	2016-03-24 10:58:45	18300	coupe	2011	manual	190	NaN	125000	
59	2016-03-29 15:48:15	1	suv	1994	manual	286	NaN	150000	
81	2016-04-03 12:56:45	350	small	1997	manual	54	NaN	150000	
115	2016-03-20 18:53:27	0	small	1999	NaN	0	NaN	5000	
135	2016-03-27 20:51:23	1450	sedan	1992	manual	136	NaN	150000	
...	...	...	...	...	...	...	...	...	
354245	2016-03-07 16:37:42	560	small	2001	auto	170	NaN	90000	
354321	2016-03-15 13:52:34	9400	wagon	2007	manual	200	NaN	150000	
354338	2016-03-31 19:52:33	180	NaN	1995	NaN	0	NaN	125000	
354351	2016-03-11 23:40:32	1900	NaN	2000	manual	110	NaN	150000	
354365	2016-03-14 17:48:27	2200	NaN	2005	NaN	0	NaN	20000	

19705 rows × 16 columns

Missing values in 'Model' can be filled in using 'Price'. For convenience, group values of 'Price' into several groups.

In [14]:
```
df['price_interval'] = pd.cut(df['Price'], 100)
```

In [15]:
```
df['Model'] = df.groupby(['price_interval'])['Model']\
.apply(lambda x: x.fillna(x.mode().iloc[0]))
```

Missing values in VehicleType, FuelType and Gearbox are filled in using 'Model'

In [16]:
```
df['VehicleType'] = df.groupby(['Model'])['VehicleType']\
.apply(lambda x: x.fillna(x.mode().iloc[0]))
```

In [17]:
```
df['FuelType'] = df.groupby(['Model'])['FuelType']\
.apply(lambda x: x.fillna(x.mode().iloc[0]))
```

In [18]:
```
df['Gearbox'] = df.groupby(['Model'])['Gearbox']\
.apply(lambda x: x.fillna(x.mode().iloc[0]))
```

Check results of missing values' processing

In [19]:
```
df.isnull().mean()
```

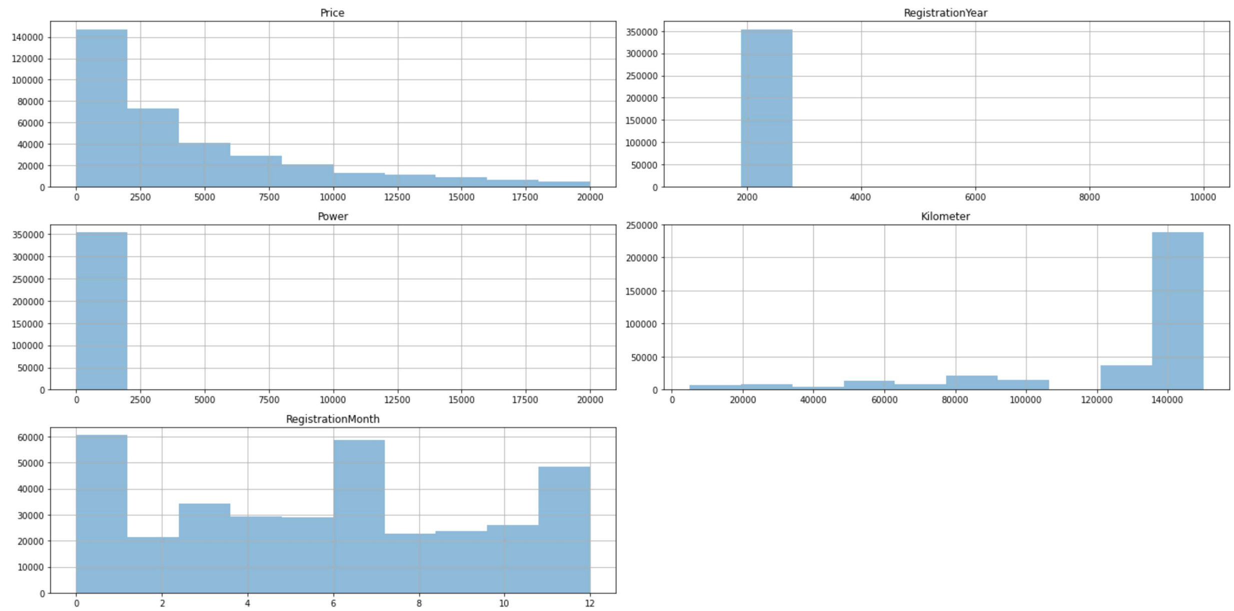
Out[19]:
```
DateCrawled          0.0
Price                0.0
VehicleType          0.0
RegistrationYear     0.0
Gearbox              0.0
Power                0.0
Model                0.0
Kilometer            0.0
RegistrationMonth    0.0
FuelType             0.0
Brand                0.0
NotRepaired          0.0
DateCreated          0.0
NumberOfPictures     0.0
PostalCode           0.0
LastSeen             0.0
price_interval       0.0
dtype: float64
```

Drop features that are not used for model

In [20]:
```
df = df.drop(['DateCrawled', 'DateCreated', 'NumberOfPictures', \
              'PostalCode', 'LastSeen', 'price_interval'], axis=1)
```

Check distribution of values for numeric values

In [21]:
```
df.hist(alpha=0.5, figsize=(20, 10))
plt.tight_layout()
```



Check some of the distributions in more detail

In [22]:
```
fig = px.histogram(df, x="Kilometer", nbins=20)
fig.show()
```

In [23]:
```
fig = px.histogram(df, x="Power")
fig.show()
```

Distribution for 'Power' has a lot of very high values. Set interval for x axis to get more details

In [24]:
```
fig = px.histogram(df, x="Power")
fig.update_xaxes(range=[0, 1000])
fig.show()
```

Most of values for 'Power' are between 40 - 360. There are also some values close to zero. Drop outliers

In [25]:
```
rows = df[df['Power'] < 40]
```

In [26]:
```
rows.shape
```

Out[26]:
```
(41655, 11)
```

In [27]:
```
df.drop(df[df['Power'] < 40].index, inplace = True)
```

In [28]:
```
df.shape
```

Out[28]:
```
(312710, 11)
```

In [29]:
```
df.drop(df[df['Power'] > 360].index, inplace = True)
```

In [30]:
```
df.shape
```

Out[30]:
```
(311643, 11)
```

In [31]:
```
fig = px.histogram(df, x="RegistrationYear")
fig.show()
```

In [32]:
```
fig = px.histogram(df, x="RegistrationYear")
fig.update_xaxes(range=[1900, 2030])
fig.show()
```

Keep objects registered between 1960 and 2018

In [33]:
```
df.drop(df[df['RegistrationYear'] < 1960].index, inplace = True)
df.drop(df[df['RegistrationYear'] > 2018].index, inplace = True)
```

In [34]:
```
fig = px.histogram(df, x="Price")
fig.show()
```

There are some objects with Price very close to zero. They should be deleted.

In [35]:
```
df.drop(df[df['Price'] < 200].index, inplace = True)
```

Transform categorical values to numeric.

In [36]:
```
cat = df[['VehicleType', 'Gearbox', 'Model', 'FuelType', 'Brand', 'NotRepaired']]
```

In [37]:
```
one_hot = pd.get_dummies(cat, drop_first=True)
```

For linear regression, create separate table with no categorical features. Then add features after one hot encoding.

In [38]:
```
df_no_cetegorical = df.drop(cat,axis = 1)
```

In [39]:
```
df_no_cetegorical = df.join(one_hot)
```

In [40]:
```
df_no_cetegorical.head()
```

Out[40]:

	Price	VehicleType	RegistrationYear	Gearbox	Power	Model	Kilometer	RegistrationMonth	Fu
1	18300	coupe	2011	manual	190	golf	125000	5	ga
2	9800	suv	2004	auto	163	grand	125000	8	ga
3	1500	small	2001	manual	75	golf	150000	6	
4	3600	small	2008	manual	69	fabia	90000	7	ga
5	650	sedan	1995	manual	102	3er	150000	10	

5 rows × 314 columns

In [41]:
```
df_no_cetegorical = df_no_cetegorical.drop(cat, axis=1)
```

In [42]:
```
df_no_cetegorical.shape
```

Out[42]:
```
(300849, 308)
```

Remove target feature and split dataset to train and test sets

In [43]:
```
features = df.drop('Price', axis=1)
target = df['Price']
```

Create non-categorical features for linear regression

In [44]:
```
features_no_cetegorical = df_no_cetegorical.drop('Price', axis=1)
```

In [45]:
```
features_train, features_test, target_train, target_test = train_test_split(
    features, target, test_size=0.25, random_state=12345)
```

Create training and test sets for linear regression with no categorical features

In [46]:
```
features_train_no_cetegorical, features_test_no_cetegorical, \
target_train_no_cetegorical, target_test_no_cetegorical = train_test_split(
    features_no_cetegorical, target, test_size=0.25, random_state=12345)
```

Summary

Dataset was uploaded, analysed and prepared for models' training and evaluation

Model Training

Linear Regression

In [47]:
```
model_lin_reg = LinearRegression(normalize=True)
```

In [48]:
```
%%time
model_lin_reg.fit(features_train_no_cetegorical, target_train_no_cetegorical)
```

```
Wall time: 5.57 s
```

Out[48]:
```
LinearRegression(normalize=True)
```

In [49]:
```
%%time
predictions_lin_reg = model_lin_reg.predict(features_test_no_cetegorical)
```

```
Wall time: 209 ms
```

In [50]:
```
predictions_lin_reg
```

Out[50]:
```
array([-314.56737791, 1825.1599859 , 8458.29028859, ..., 3253.69444881,
       3943.85182343, 5548.5112234 ])
```

In [51]:
```
mse_lin_reg = mean_squared_error(target_test_no_cetegorical, predictions_lin_reg)
mse_lin_reg
```

Out[51]:
```
6826715.263278074
```

In [52]:
```
rmse_lin_reg = sqrt(mse_lin_reg)
print("rmse_lin_reg:", rmse_lin_reg)
```

```
rmse_lin_reg: 2612.798358710077
```

In [53]:
```
%%time
predictions_lin_reg_predict = model_lin_reg.predict(features_train_no_cetegorical
```

```
Wall time: 596 ms
```

In [54]:
```
mse_lin_reg_train = mean_squared_error(target_train_no_cetegorical,\
                                       predictions_lin_reg_predict)
mse_lin_reg_train
```

Out[54]:
```
6823043.075354917
```

In [55]:
```
rmse_lin_reg_train = sqrt(mse_lin_reg_train)
print("rmse_lin_reg_train:", rmse_lin_reg_train)
```

```
rmse_lin_reg_train: 2612.095533351511
```

Ridge Regression

In [56]:
```
reg = Ridge(alpha=.5)
```

In [57]:
```
%%time
reg.fit(features_train_no_cetegorical, target_train_no_cetegorical)
```

```
Wall time: 1.34 s
```

Out[57]:
```
Ridge(alpha=0.5)
```

In [58]:
```
%%time
pred_test_reg= reg.predict(features_test_no_cetegorical)
```

```
Wall time: 279 ms
```

In [59]:
```
mse_lin_regularized = mean_squared_error(target_test_no_cetegorical, pred_test_re
mse_lin_regularized
```

Out[59]:
```
6826310.753355717
```

In [60]:
```
rmse_lin_regularized = sqrt(mse_lin_regularized)
print("rmse_Ridge:", rmse_lin_regularized)
```

```
rmse_Ridge: 2612.7209482368603
```

Regularization Coefficient search for Ridge Regression

In [61]:
```
parameters_ridge = {
    'alpha': (0.01, 0.05, 0.1, 0.3, 0.5, 0.7, 0.9),
}
```

In [62]:
```
mse = make_scorer(mean_squared_error,greater_is_better=False)
```

In [63]:
```
ridge_cv = GridSearchCV(Ridge(), parameters_ridge, scoring=mse, n_jobs=-1, verbos
```

In [64]:
```
%%time
ridge_cv.fit(features_train_no_cetegorical, target_train_no_cetegorical)
```

```
Fitting 5 folds for each of 7 candidates, totalling 35 fits
Wall time: 57.6 s
```

Out[64]:
```
GridSearchCV(estimator=Ridge(), n_jobs=-1,
             param_grid={'alpha': (0.01, 0.05, 0.1, 0.3, 0.5, 0.7, 0.9)},
             scoring=make_scorer(mean_squared_error, greater_is_better=False),
             verbose=1)
```

In [65]:
```
ridge_cv.best_params_
```

Out[65]:
```
{'alpha': 0.9}
```

In [66]:
```
%%time
ridge_cv_predict = ridge_cv.predict(features_test_no_cetegorical)
```

```
Wall time: 357 ms
```

In [67]:
```
mean_squared_error(target_test, ridge_cv_predict) ** 0.5
```

Out[67]:
```
2612.6785268578924
```

There is no significant improvement of RMSE for Ridge regression post-cv

Gradient Boosting with LightGBM

Method 1: 'out-of-the-box' model approach

Transform categorical features from type 'object' to type 'category'

In [68]:
```
categorical = ['VehicleType', 'Gearbox', 'Model', 'FuelType', 'Brand', 'NotRepair
```

In [69]:
```
type(categorical)
```

Out[69]:
```
list
```

In [70]:
```
for feat in categorical:
    features_train[feat] = features_train[feat].astype('category')
    features_test[feat] = features_test[feat].astype('category')
```

In [71]:
```
features_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 225636 entries, 319975 to 256208
Data columns (total 10 columns):
 #   Column             Non-Null Count   Dtype   
---  ------             --------------   -----   
 0   VehicleType        225636 non-null  category
 1   RegistrationYear   225636 non-null  int64   
 2   Gearbox            225636 non-null  category
 3   Power              225636 non-null  int64   
 4   Model              225636 non-null  category
 5   Kilometer          225636 non-null  int64   
 6   RegistrationMonth  225636 non-null  int64   
 7   FuelType           225636 non-null  category
 8   Brand              225636 non-null  category
 9   NotRepaired        225636 non-null  category
dtypes: category(6), int64(4)
memory usage: 10.1 MB
```

categorical_feature are added

In [72]:
```
train_data = lgb.Dataset(features_train, label=target_train,\
                        categorical_feature=['VehicleType', 'Gearbox', 'Model', '
test_data = lgb.Dataset(features_test, label=target_test,\
                       categorical_feature=['VehicleType', 'Gearbox', 'Model', 'F
```

In [73]:
```
parameters = {
    'application': 'regression',
    'objective': 'regression',
    'metric': 'rmse',
    'is_unbalance': 'true',
    'boosting': 'gbdt',
    'num_leaves': 31,
    'feature_fraction': 0.5,
    'bagging_fraction': 0.5,
    'bagging_freq': 20,
    'learning_rate': 0.05,
    'verbose': 0,
}
```

In [74]:
```
%%time
model_lightgbm = lgb.train(parameters,
                       train_data,
                       valid_sets=test_data,
                       num_boost_round=500,
                       early_stopping_rounds=20,
                       )
```

```
[LightGBM] [Warning] objective is set=regression, application=regression will be ignored. Current value: objective=regression
[LightGBM] [Warning] objective is set=regression, application=regression will be ignored. Current value: objective=regression
[LightGBM] [Warning] Auto-choosing row-wise multi-threading, the overhead of testing was 0.011625 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Warning] objective is set=regression, application=regression will be ignored. Current value: objective=regression
[1]	valid_0's rmse: 4466.75
Training until validation scores don't improve for 20 rounds
[2]	valid_0's rmse: 4365.18
[3]	valid_0's rmse: 4208.87
[4]	valid_0's rmse: 4059.39
[5]	valid_0's rmse: 3965.58
[6]	valid_0's rmse: 3883.84
[7]	valid_0's rmse: 3754.33
[8]	valid_0's rmse: 3682.6
```

In [75]:
```
%%time
predictions_model_lightgbm = model_lightgbm.predict(features_test)
```

```
Wall time: 1.46 s
```

In [76]:
```
mse_lightgbm = mean_squared_error(target_test, predictions_model_lightgbm)
mse_lightgbm
```

Out[76]:
```
2479947.8787624445
```

In [77]:
```
rmse_lightgbm = sqrt(mse_lightgbm)
print("rmse_lightgbm:", rmse_lightgbm)
```

```
rmse_lightgbm: 1574.7850262059405
```

Method 2: Use improved model

Find better parameters for the model with RandomSearchCV.

In [78]:
```
parameters = {
    'max_depth': (10, 13, 18),
    "n_estimators": (50,100,300),
    "num_leaves": (10,15,30)
}
```

In [79]:
```
%%time
rs_cv = RandomizedSearchCV(estimator=lgb.LGBMRegressor(), param_distributions=par
                           n_iter=100,verbose=1)
rs_cv.fit(features_train, target_train,verbose=1)
```

```
Fitting 2 folds for each of 27 candidates, totalling 54 fits
Wall time: 1min 12s
```

Out[79]:
```
RandomizedSearchCV(cv=2, estimator=LGBMRegressor(), n_iter=100, n_jobs=4,
                   param_distributions={'max_depth': (10, 13, 18),
                                        'n_estimators': (50, 100, 300),
                                        'num_leaves': (10, 15, 30)},
                   verbose=1)
```

In [80]:
```
rs_cv.best_params_
```

Out[80]:
```
{'num_leaves': 30, 'n_estimators': 300, 'max_depth': 13}
```

In [81]:
```
%%time
rs_cv_predict = rs_cv.predict(features_test)
```

```
Wall time: 729 ms
```

In [82]:
```
mean_squared_error(target_test, rs_cv_predict) ** 0.5
```

Out[82]:
```
1546.8504676082032
```

In [86]:
```
data = [["Linear Regression", '5.57 s', '209 ms', 2612.10],
        ["Ridge", '1.34 s', '279 ms', 2612.72],
        ["Ridge with CV", '57.6 s', '357 ms', 2612.68],
        ["LightGBM standard param", '7.78 s', '1.46 s', 1574.79],
        ["LightGBM with CV", '1min 12s', '729 ms', 1546.85]]


col_names = ["model", "train_time", "predict_time", 'rmse']
```

In [87]:
```
pd.DataFrame(data=data, columns=col_names).set_index('model')
```

Out[87]:

model	train_time	predict_time	rmse
Linear Regression	5.57 s	209 ms	2612.10
Ridge	1.34 s	279 ms	2612.72
Ridge with CV	57.6 s	357 ms	2612.68
LightGBM standard param	7.78 s	1.46 s	1574.79
LightGBM with CV	1min 12s	729 ms	1546.85

Summary

RMSE for linear regression is 2612.10 for test set, model training takes 5.57 s.

Ridge Regression with alpha=0.9 shows RMSE = 2612.68 for test set, model training takes 57.6 s.

Gradient Boosting with LightGBM with CV shows the lowest RMSE for the test set (1546.85). Model training takes 1min 12s.

Conclusion

Ridge Regression model takes the least amount of time for model training. Its RMSE = 2612.68 which is quite good for fast predictions.

LightGBM with CV shows the best quality of predictions (RMSE = 1546.85), although it takes longer to train the model.