Numerical Methods Project

Car price prediction using gradient boosting

Table of Contents

- 1 Goal
- 2 Data description
- 3 Imports
- 4 Input data
- 5 Descriptive statistics
- 6 Data preprocessing
    - 6.1 Column names
    - 6.2 Data type change
    - 6.3 Missing values
        - 6.3.1 Not_repaired
        - 6.3.2 Model, vehicle_type, fuel_type
        - 6.3.3 Gearbox
    - 6.4 Duplicates
- 7 EDA
    - 7.1 Outliers
        - 7.1.1 Registration_year
        - 7.1.2 Price
        - 7.1.3 Mileage
        - 7.1.4 Power
    - 7.2 Car age calculation
        - 7.2.1 Postal_code
    - 7.3 Drop columns
- 8 Encoding of categorical variables
- 9 Splitting data into train, validation and test sets
- 10 Standard scaling
- 11 Model selection
    - 11.1 Linear regression
    - 11.2 Random Forest
        - 11.2.1 Base model
        - 11.2.2 Hyperparameters tuning
    - 11.3 XGBoost
        - 11.3.1 Hyperparameters tuning
    - 11.4 LightGBM
        - 11.4.1 Hyperparameters tuning
    - 11.5 CatBoost
        - 11.5.1 Hyperparameters tuning
    - 11.6 Results

- 12 Retrain the best tuned model
- 13 Sanity check

Goal

Develop a model for Rusty Bargain used car sales service to determine the market value of a car based on historical data (technical specifications, trim versions, and prices).

Key metrics:

- the quality of the prediction
- the speed of the prediction
- the time required for training

Data description

Features

- *DateCrawled* — date profile was downloaded from the database
- *VehicleType* — vehicle body type
- *RegistrationYear* — vehicle registration year
- *Gearbox* — gearbox type
- *Power* — power (hp)
- *Model* — vehicle model
- Mileage — mileage (measured in km due to dataset's regional specifics)
- *RegistrationMonth* — vehicle registration month
- *FuelType* — fuel type
- *Brand* — vehicle brand
- *NotRepaired* — vehicle repaired or not
- *DateCreated* — date of profile creation
- *NumberOfPictures* — number of vehicle pictures
- *PostalCode* — postal code of profile owner (user)
- *LastSeen* — date of the last activity of the user

Target

Price — price (Euro)

Imports

In [6]:

```
import pandas as pd
import matplotlib
import numpy as np
from numpy import *
import re
from time import time

from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from catboost import CatBoostRegressor
from lightgbm import LGBMRegressor
from xgboost import XGBRegressor
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler as ss
from sklearn.metrics import mean_squared_error

import matplotlib.pyplot as plt
%matplotlib inline

import sys
import warnings
if not sys.warnoptions:
       warnings.simplefilter("ignore")

pd.set_option('display.max_rows', None, 'display.max_columns', None)

print("Setup Complete")
```

```
Setup Complete
```

Input data

In [7]:
```
try:
    df = pd.read_csv('car_data.csv')

except:
    df = pd.read_csv('/datasets/car_data.csv')
```

Descriptive statistics

In [8]:
```
df.head()
```

Out[8]:

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

In [9]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 354369 entries, 0 to 354368
Data columns (total 16 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   DateCrawled        354369 non-null  object
 1   Price              354369 non-null  int64
 2   VehicleType        316879 non-null  object
 3   RegistrationYear   354369 non-null  int64
 4   Gearbox            334536 non-null  object
 5   Power              354369 non-null  int64
 6   Model              334664 non-null  object
 7   Mileage            354369 non-null  int64
 8   RegistrationMonth  354369 non-null  int64
 9   FuelType           321474 non-null  object
```

```
 10  Brand              354369 non-null  object
 11  NotRepaired        283215 non-null  object
 12  DateCreated        354369 non-null  object
 13  NumberOfPictures   354369 non-null  int64
 14  PostalCode         354369 non-null  int64
 15  LastSeen           354369 non-null  object
dtypes: int64(7), object(9)
memory usage: 43.3+ MB
```

Notes for preprocessing:

- There are more than 35k observations with 15 features (6 categorical, 9 numeric) and 1 target variables;
- Convert column names to lower case;
- Fill in missing values;
- Check for duplicates;
- Convert 3 date features to datetime format;
- Encode 6 categorical features (needed for some models);
- `DateCrawled` and `LastSeen` can be removed as not related to price;
- Calculate the age of a car at the moment of price checking as a difference between `RegistrationYear` and `DateCreated` year;
- The target is numeric, it's a regression task.

In [10]:

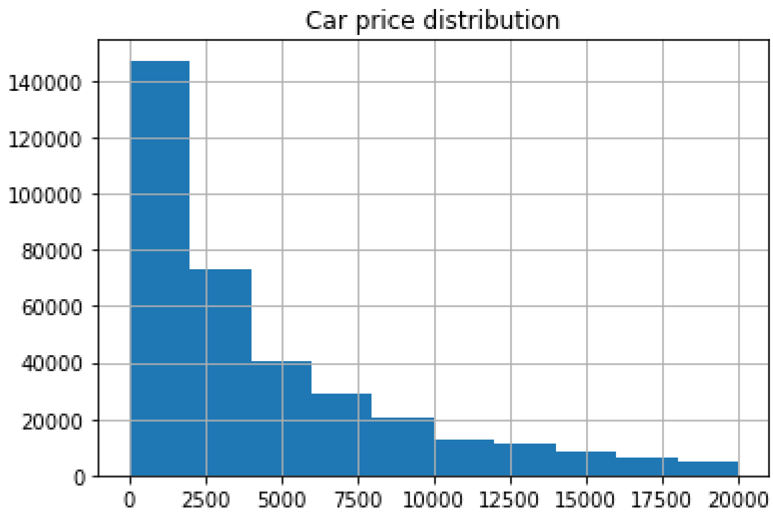
```
df.describe()
```

Out[10]:

	Price	RegistrationYear	Power	Mileage	RegistrationMonth
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

In [11]:

```
df['Price'].hist()
plt.title('Car price distribution');
```



Notes for preprocessing:

- Check `RegistrationYear` as the min value is 1000 and max is 9999 -> outliers;
- Check 0 price observations. The target distribution is positively skewed;
- Check `Mileage` column for outliers;
- Check `Power` feature as max value is too big and min is 0 -> outliers;
- `NumberOfPictures` feature has only 0 values -> can be removed as non-informative.

Data preprocessing

Column names

In [12]:
```
columns = []
for name in df.columns.values:
    name = re.sub('([A-Z])', r' \1', name).lower().replace(' ', '_')[1:]
    columns.append(name)
```

In [13]:
```
df.columns = columns
```

In [14]:
```
df.head(3)
```

Out[14]:

	date_crawled	price	vehicle_type	registration_year	gearbox	power	model	mileage	regi
0	24/03/2016 11:52	480	NaN	1993	manual	0	golf	150000	
1	24/03/2016 10:58	18300	coupe	2011	manual	190	NaN	125000	
2	14/03/2016 12:52	9800	suv	2004	auto	163	grand	125000	

Data type change

As mentioned above, let's convert all the date columns to the datetime type.

In [15]:
```
df['date_created'] = pd.to_datetime(df['date_created'])
```

Missing values

In [16]:
```
df.isnull().sum()/df.shape[0]
```

Out[16]:
```
date_crawled          0.000000
price                 0.000000
vehicle_type          0.105794
registration_year     0.000000
gearbox               0.055967
power                 0.000000
model                 0.055606
mileage               0.000000
registration_month    0.000000
fuel_type             0.092827
brand                 0.000000
not_repaired          0.200791
date_created          0.000000
number_of_pictures    0.000000
postal_code           0.000000
last_seen             0.000000
dtype: float64
```

5 features have missing values, let's start with the one that has the most significant number of them - over 20%.

Not_repaired

In [17]:
```
df['not_repaired'].value_counts()
```

Out[17]:
```
no     247161
yes     36054
Name: not_repaired, dtype: int64
```

Let's assume that a missing value in this column means not repaired (a person probably just forgot to fill this column). Besides, it's the majority group, so this way we will not influence the ratio much.

In [18]:
```
df['not_repaired'].fillna('no', inplace=True)
```

Model, vehicle_type, fuel_type

We will replace all the missing values in these columns with the 'n/a' string.

In [19]:
```
for col in ['model','vehicle_type','fuel_type']:
    df[col].fillna('n/a', inplace=True)
```

Gearbox

As there are only 2 possible values in this column, let's fill the missing values with the majority group.

In [20]:
```
df['gearbox'].value_counts()
```

Out[20]:
```
manual    268251
auto       66285
Name: gearbox, dtype: int64
```

In [21]:
```
df['gearbox'].fillna('manual', inplace=True)
```

Duplicates

Let's check if any rows are duplicated.

In [22]:
```
df.duplicated().sum()
```

Out[22]:
```
292
```

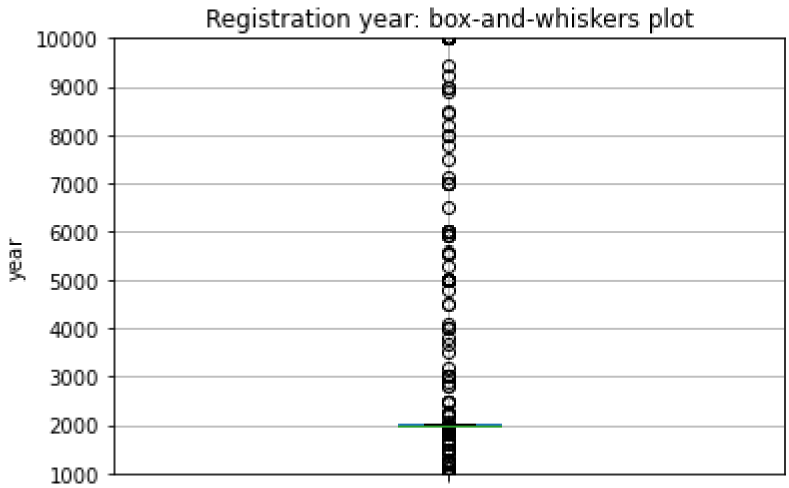
In [23]:
```
df = df.drop_duplicates(ignore_index=True)
```

EDA

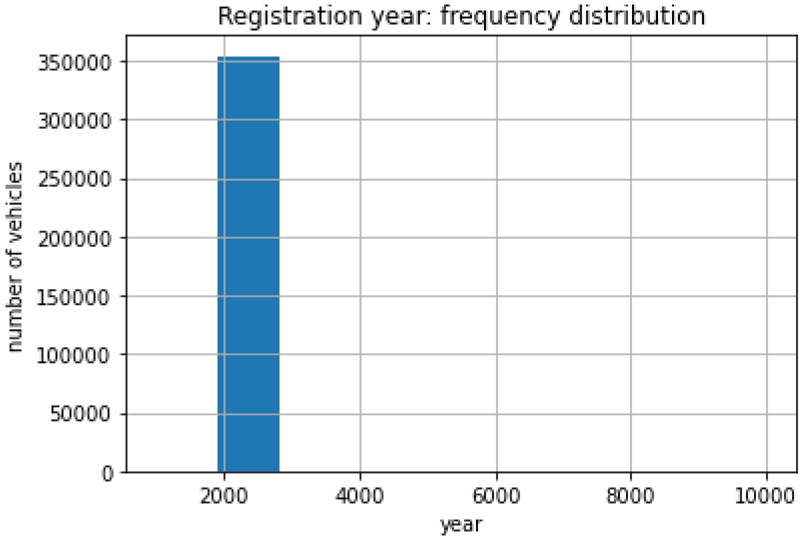
Outliers

Registration_year

In [24]:
```
df.boxplot('registration_year')
plt.ylim(1000, 10000)
plt.title('Registration year: box-and-whiskers plot')
plt.xticks([1], [''])
plt.ylabel('year');
```




```
In [25]: df.hist('registration_year')
plt.title('Registration year: frequency distribution')
plt.xlabel('year')
plt.ylabel('number of vehicles');
```



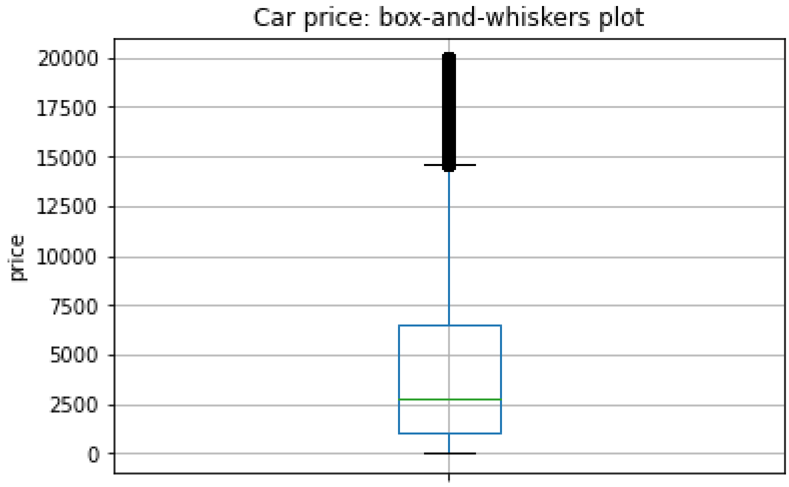
We see a lot of outliers but most observations are still around 2000. Since data is extracted in 2016, there shouldn't be any `registration_year` more than that. Let's set the lower border to be 1900.

```
In [26]: df.loc[(df['registration_year'] < 1900) | (df['registration_year'] > 2016), ':
```

```
In [27]: df = df.dropna(subset=['registration_year'], axis=0)
df.reset_index(drop=True, inplace=True)
```

Price

```
In [28]: df.boxplot('price')
plt.title('Car price: box-and-whiskers plot')
plt.xticks([1], [''])
plt.ylabel('price');
```



Most prices are in range from around 1000 to 6000 euro. There are some higher values but the maximum price is 20000 euro and it seems reasonable. Let's check for other artifacts.

In [29]:
```
len(df[df['price'] == 12345])
```

Out[29]: 7

In [30]:
```
len(df[df['price'] == 1])
```

Out[30]: 1118

In [31]:
```
len(df[df['price'] == 0])
```

Out[31]: 10006

Let's simply remove them, as the `price` column is our target variable and the above values are non-informative for our purpose.

In [32]:
```
df.loc[(df['price'] == 12345) | (df['price'] == 1) | (df['price'] == 0), 'pri
```

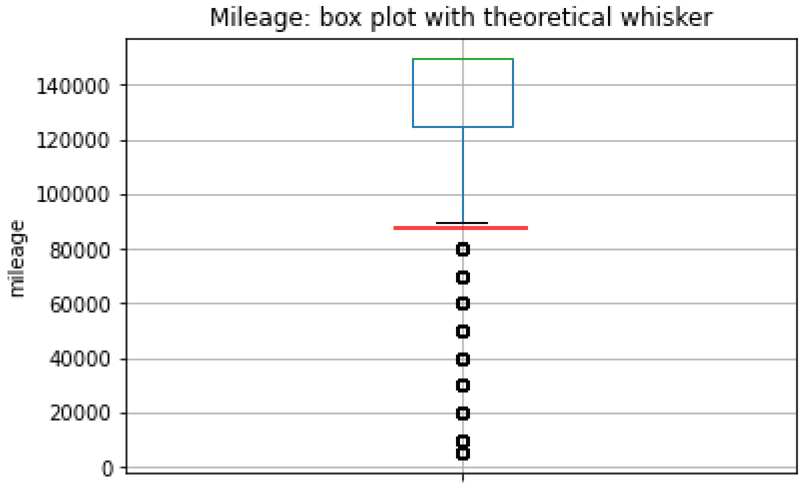
In [33]:
```
df = df.dropna(subset=['price'], axis=0)
df.reset_index(drop=True, inplace=True)
df.shape
```

Out[33]: (328351, 16)

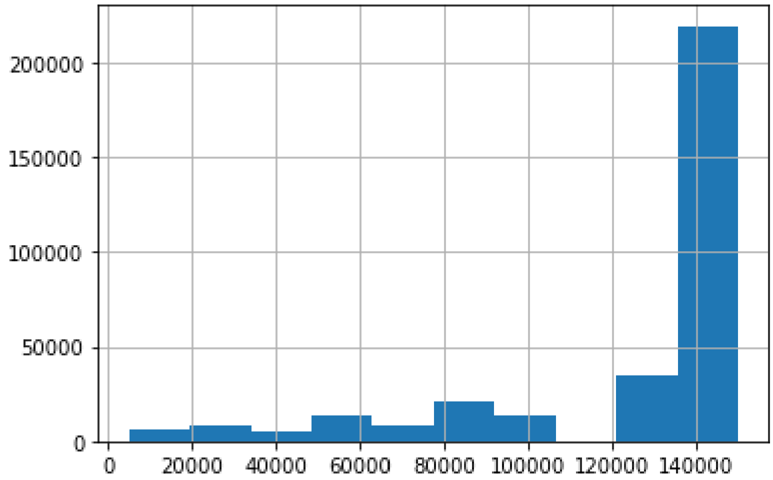
Mileage

In [34]:
```
Q1 = df['mileage'].quantile(0.25)
Q3 = df['mileage'].quantile(0.75)
IQR = Q3 - Q1
lower_whisker = Q1 - 1.5 * IQR
df.boxplot('mileage')
plt.hlines(y=lower_whisker, xmin=0.9, xmax=1.1, color='red')

plt.title('Mileage: box plot with theoretical whisker')
plt.xticks([1], [''])
plt.ylabel('mileage');
```



In [35]:
```
df['mileage'].hist();
```

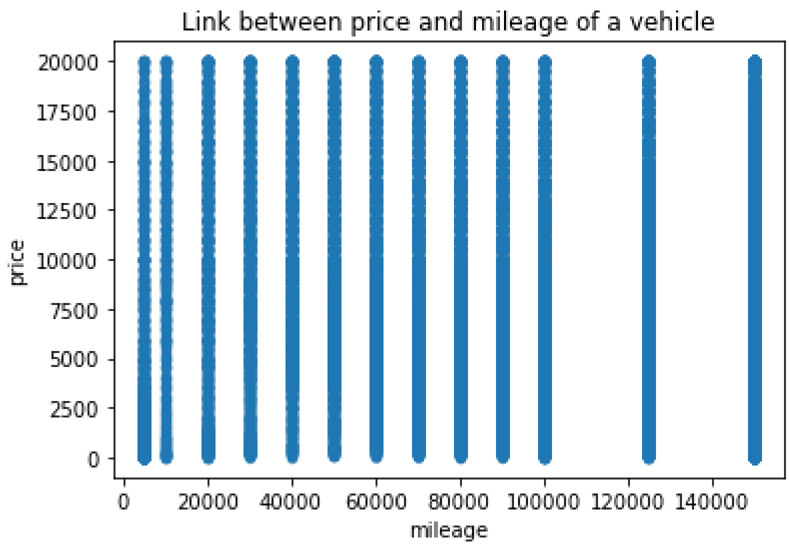


In [36]:
```
len(df[df['mileage'] < lower_whisker])
```

Out[36]: 49516

Most observations have a high mileage value, there are some values lower than the lower whisker but we will keep them as they are not numerous and these values could possibly exist.

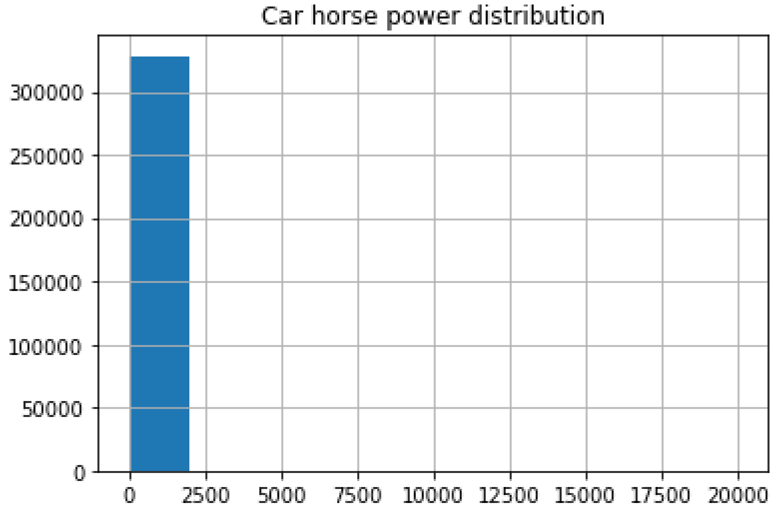
In [37]:
```
df.plot.scatter(x='mileage', y='price', alpha=.25)
plt.title('Link between price and mileage of a vehicle');
```



No clear linear connection between car price and mileage detected.

Power

In [38]:
```
df['power'].hist()
plt.title('Car horse power distribution');
```



In [39]:
```
len(df[df['power'] > 2500])
```

Out[39]: 85

In [40]:
```
len(df[df['power'] == 0])
```

Out[40]: 32395

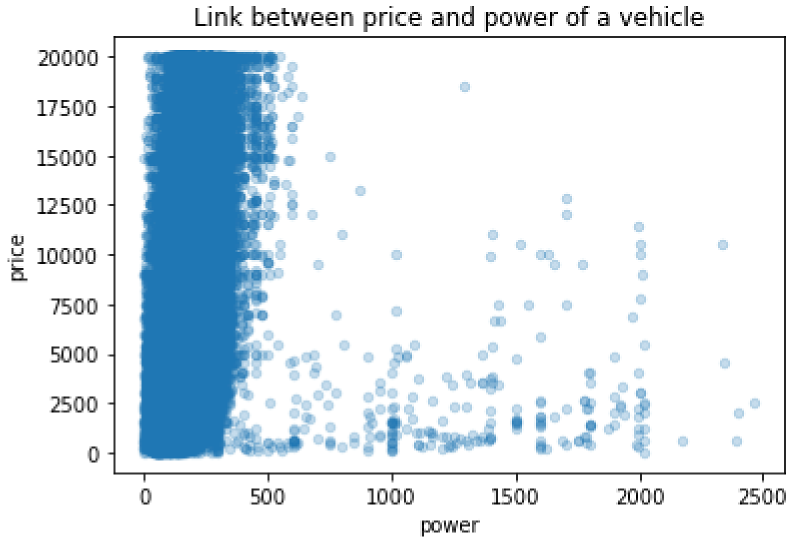
Let's remove observations with car power higher than 2500 and also those with 0 horse power as not valid values.

In [41]:
```
df.loc[(df['power'] > 2500) | (df['power'] == 0), 'power'] = np.nan

df = df.dropna(subset=['power'], axis=0)
df.reset_index(drop=True, inplace=True)
df.shape
```

Out[41]: (295871, 16)

In [42]:
```
df.plot.scatter(x='power', y='price', alpha=.25)
plt.title('Link between price and power of a vehicle');
```



No clear linear connection between car price and power detected.

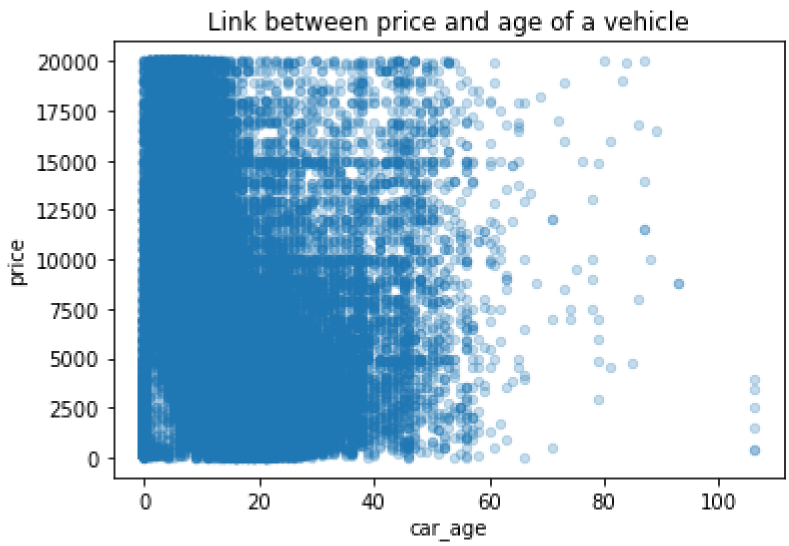
Car age calculation

Let's calculate car age at the moment of price inquiry. We'll extract the year of profile creation and subtract the registration year.

In [43]:
```
df['year_created'] = df['date_created'].dt.year
```

In [44]:
```
df['car_age'] = df['year_created'] - df['registration_year']
```

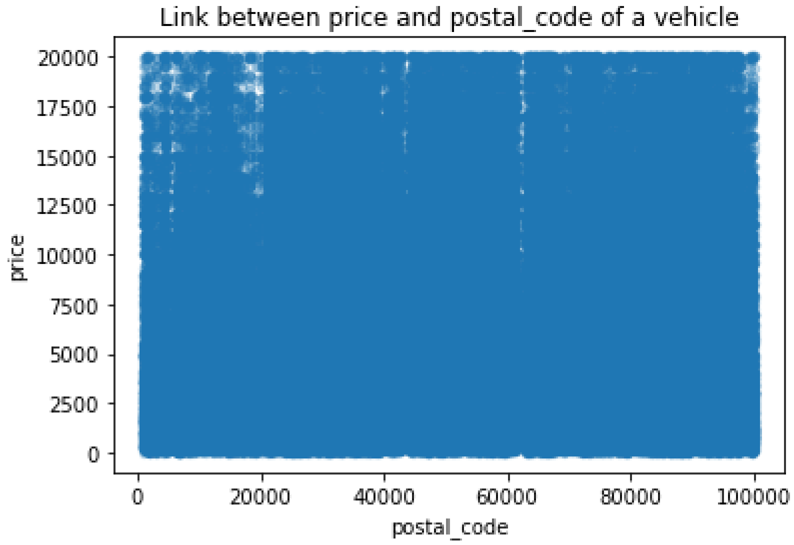
In [45]:
```
df.plot.scatter(x='car_age', y='price', alpha=.25)
plt.title('Link between price and age of a vehicle');
```



There is a slight linear negative connection between car price and car age - the lower the car age, the lower the price, although it does not stand for all observations (this could be due to other features).

Postal_code

In [46]:
```
df.plot.scatter(x='postal_code', y='price', alpha=.25)
plt.title('Link between price and postal_code of a vehicle');
```



No linear dependency between these 2 variables, consider to test a model without `postal_code` feature.

Drop columns

In [47]:
```
df = df.drop(['date_crawled','last_seen', 'number_of_pictures', 'registration_
         'date_created', 'registration_year', 'year_created'], axis=1)
```

In [48]:
```
df.head()
```

Out[48]:

	price	vehicle_type	gearbox	power	model	mileage	fuel_type	brand	not_repaire
0	18300.0	coupe	manual	190.0	n/a	125000	gasoline	audi	ye
1	9800.0	suv	auto	163.0	grand	125000	gasoline	jeep	n
2	1500.0	small	manual	75.0	golf	150000	petrol	volkswagen	n
3	3600.0	small	manual	69.0	fabia	90000	gasoline	skoda	n
4	650.0	sedan	manual	102.0	3er	150000	petrol	bmw	ye

Encoding of categorical variables

In [49]:
```
df = pd.get_dummies(df, drop_first=True)
df.head()
```

Out[49]:

	price	power	mileage	postal_code	car_age	vehicle_type_convertible	vehicle_type_coup
0	18300.0	190.0	125000	66954	5.0	0	
1	9800.0	163.0	125000	90480	12.0	0	(
2	1500.0	75.0	150000	91074	15.0	0	(
3	3600.0	69.0	90000	60437	8.0	0	(
4	650.0	102.0	150000	33775	21.0	0	(

Splitting data into train, validation and test sets

First, let's split data into train and test sets with the 80/20 proportion, respectively.

In [50]:
```
X = df.drop('price', axis=1)
y = df['price']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, ra
```

In [51]:
```
X_train, X_valid, y_train, y_valid = train_test_split(X_train, y_train,
                                                       test_si
```

Standard scaling

Let's scale the features before modeling to be able to compare their coefficients in the later sections.

In [52]:
```
sc = ss()
X_train = sc.fit_transform(X_train)
X_valid = sc.transform(X_valid)
X_test = sc.transform(X_test)
```

Model selection

We will be using the RMSE metric for best model selection.

Linear regression

In [53]:
```
LR = LinearRegression()
start = time()
LR.fit(X_train, y_train)
end = time()
fit_time_LR = end - start

start = time()
y_pred = LR.predict(X_valid)
end = time()
predict_time_LR = end - start

LR_score = mean_squared_error(y_valid, y_pred) ** 0.5
LR_score
```

Out[53]:
```
2719.450324154307
```

Random Forest

Base model

In [54]:
```
RF = RandomForestRegressor()

start = time()
RF.fit(X_train, y_train)
end = time()
fit_time_RF = end - start

start = time()
y_pred = RF.predict(X_valid)
end = time()
predict_time_RF = end - start

RF_score_base = mean_squared_error(y_valid, y_pred) ** 0.5
RF_score_base
```

```
1572.6741109836544
```

Out[54]:

Hyperparameters tuning

In [55]:

```
d = []
for estim in [100, 500]:
    for depth in [5,10]:
        RF = RandomForestRegressor(random_state=12345, n_estimators=estim, ma

        start = time()
        RF.fit(X_train, y_train)
        end = time()
        fit_time_RF_tuned = end - start

        start = time()
        y_pred = RF.predict(X_valid)
        end = time()
        predict_time_RF_tuned = end - start

        RF_score_tuned = mean_squared_error(y_valid, y_pred) ** 0.5
        d.append(
            {
                'n_estimators': estim,
                'max_depth': depth,
                'RF_score_tuned':  RF_score_tuned,
                'fit_time_RF_tuned': fit_time_RF_tuned,
                'predict_time_RF_tuned': predict_time_RF_tuned
            }
        )
best_param = pd.DataFrame(d).nsmallest(1, ['RF_score_tuned'], keep='first')
RF_score_tuned = best_param['RF_score_tuned'].values
fit_time_RF_tuned = best_param['fit_time_RF_tuned'].values
predict_time_RF_tuned = best_param['predict_time_RF_tuned'].values

best_param
```

Out[55]:

	n_estimators	max_depth	RF_score_tuned	fit_time_RF_tuned	predict_time_RF_tuned
3	500	10	1895.311581	1357.000438	2.313169

XGBoost

Advice from Machinelearningmastery blog:

- Decision trees are added to the model sequentially in an effort to correct and improve upon the predictions made by prior trees. As such, more trees is often better.
- The tree depth controls how specialized each tree is to the training dataset: how general or overfit it might be. Trees are preferred that are not too shallow and general (like AdaBoost) and not too deep and specialized (like bootstrap aggregation). Gradient boosting generally performs well with trees that have a modest depth, finding a balance between skill and generality.
- Learning rate controls the amount of contribution that each model has on the ensemble prediction. Smaller rates may require more decision trees in the ensemble.
- The number of samples used to fit each tree can be varied. This means that each tree is fit on a randomly selected subset of the training dataset. Using fewer samples

introduces more variance for each tree, although it can improve the overall performance of the model.

- Changing the number of features introduces additional variance into the model, which may improve performance, although it might require an increase in the number of trees.

In [56]:
```
XGB = XGBRegressor(n_jobs=-1)

start = time()
XGB.fit(X_train, y_train)
end = time()
fit_time_XGB = end - start

start = time()
y_pred = XGB.predict(X_valid)
end = time()
predict_time_XGB = end - start

XGB_score_base = mean_squared_error(y_valid, y_pred) ** 0.5
XGB_score_base
```

Out[56]: 1644.948134881074

Hyperparameters tuning

In [57]:
```
d = []
for estim in [100, 500]:
    for depth in [5, 10]:
        XGB = XGBRegressor(random_state=12345, n_estimators=estim,
                                                max_depth=depth, n_jobs=-1)
        start = time()
        XGB.fit(X_train, y_train)
        end = time()
        fit_time_XGB_tuned = end - start

        start = time()
        y_pred = XGB.predict(X_valid)
        end = time()
        predict_time_XGB_tuned = end - start

        XGB_score_tuned = mean_squared_error(y_valid, y_pred) ** 0.5
        d.append(
                        {
                            'n_estimators': estim,
                            'max_depth': depth,
                            'XGB_score_tuned':  XGB_score_tuned,
                            'fit_time_XGB_tuned': fit_time_XGB_tuned,
                            'predict_time_XGB_tuned': predict_time_XGB_tuned
                        }
                    )

best_param = pd.DataFrame(d).nsmallest(1, ['XGB_score_tuned'], keep='first')
XGB_score_tuned = best_param['XGB_score_tuned'].values
fit_time_XGB_tuned = best_param['fit_time_XGB_tuned'].values
predict_time_XGB_tuned = best_param['predict_time_XGB_tuned'].values

best_param
```

Out[57]:

	n_estimators	max_depth	XGB_score_tuned	fit_time_XGB_tuned	predict_time_XGB_tuned
3	500	10	1550.721324	1287.395914	0.96534

LightGBM

In [58]:
```
LGB = LGBMRegressor()

start = time()
LGB.fit(X_train, y_train)
end = time()
fit_time_LGB = end - start

start = time()
y_pred = LGB.predict(X_valid)
end = time()
predict_time_LGB = end - start

LGB_score_base = mean_squared_error(y_valid, y_pred) ** 0.5
LGB_score_base
```

Out[58]:
```
1685.5284808924687
```

Hyperparameters tuning

In [59]:
```
d = []
for estim in [100, 500]:
    for depth in [5, 10]:
        LGB = LGBMRegressor(random_state=12345, n_estimators=estim,
                                               max_depth=depth)
        start = time()
        LGB.fit(X_train, y_train)
        end = time()
        fit_time_LGB_tuned = end - start

        start = time()
        y_pred = LGB.predict(X_valid)
        end = time()
        predict_time_LGB_tuned = end - start

        LGB_score_tuned = mean_squared_error(y_valid, y_pred) ** 0.5
        d.append(
            {
                 'n_estimators': estim,
                  'max_depth': depth,
                  'LGB_score_tuned':  LGB_score_tuned,
                  'fit_time_LGB_tuned': fit_time_LGB_tuned,
                  'predict_time_LGB_tuned': predict_time_LGB_tuned
                        }
                    )

best_param = pd.DataFrame(d).nsmallest(1, ['LGB_score_tuned'], keep='first')
LGB_score_tuned = best_param['LGB_score_tuned'].values
fit_time_LGB_tuned = best_param['fit_time_LGB_tuned'].values
predict_time_LGB_tuned = best_param['predict_time_LGB_tuned'].values

best_param
```

```
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_l
eaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_l
eaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_l
eaves OR 2^max_depth > num_leaves. (num_leaves=31).
[LightGBM] [Warning] Accuracy may be bad since you didn't explicitly set num_l
eaves OR 2^max_depth > num_leaves. (num_leaves=31).
```

Out[59]:

	n_estimators	max_depth	LGB_score_tuned	fit_time_LGB_tuned	predict_time_LGB_tuned

	n_estimators	max_depth	LGB_score_tuned	fit_time_LGB_tuned	predict_time_LGB_tuned
3	500	10	1596.001587	8.077975	1.028998

CatBoost

In [60]:

```
CB = CatBoostRegressor(verbose=0)

start = time()
CB.fit(X_train, y_train)
end = time()
fit_time_CB = end - start

start = time()
y_pred = CB.predict(X_valid)
end = time()
predict_time_CB = end - start

CB_score_base = mean_squared_error(y_valid, y_pred) ** 0.5
CB_score_base
```

Out[60]: 1596.4549062403007

Hyperparameters tuning

In [61]:

```
d = []
for iterations in [100, 500]:
    for depth in [5, 10]:
        CB = CatBoostRegressor(random_state=12345, iterations=iterations,
                               depth=depth, verbose=0)
        start = time()
        CB.fit(X_train, y_train)
        end = time()
        fit_time_CB_tuned = end - start

        start = time()
        y_pred = CB.predict(X_valid)
        end = time()
        predict_time_CB_tuned = end - start

        CB_score_tuned = mean_squared_error(y_valid, y_pred) ** 0.5
        d.append(
            {
                              'n_estimators': estim,
                              'depth': depth,
                              'CB_score_tuned':  CB_score_tuned,
                              'fit_time_CB_tuned': fit_time_CB_tuned,
                              'predict_time_CB_tuned': predict_time_CB_tuned
                        }
                    )

best_param = pd.DataFrame(d).nsmallest(1, ['CB_score_tuned'], keep='first')
CB_score_tuned = best_param['CB_score_tuned'].values
fit_time_CB_tuned = best_param['fit_time_CB_tuned'].values
predict_time_CB_tuned = best_param['predict_time_CB_tuned'].values

best_param
```

Out[61]:

	n_estimators	depth	CB_score_tuned	fit_time_CB_tuned	predict_time_CB_tuned
3	500	10	1555.803504	24.208392	0.089928

Results

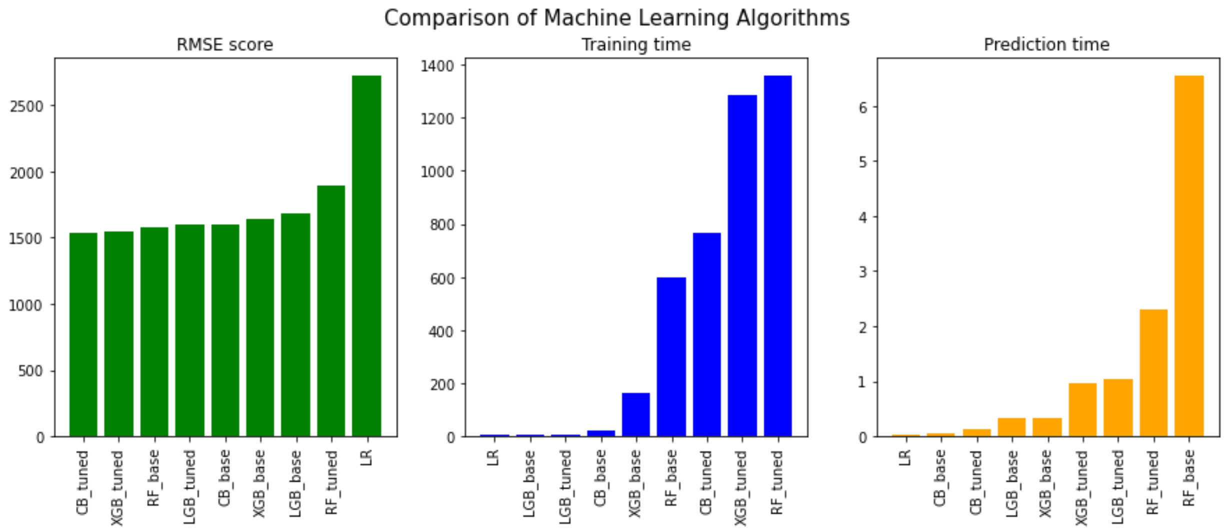
In [79]:
```
models = pd.DataFrame({
    'Model': ['LR', 'RF_base', 'XGB_base', 'LGB_base', 'CB_base',
              'RF_tuned', 'XGB_tuned', 'LGB_tuned', 'CB_tuned'],
    'Score': [LR_score, RF_score_base, XGB_score_base, LGB_score_base, CB_sco
              RF_score_tuned, XGB_score_tuned, LGB_score_tuned, CB_score_tuned
    'Training_time': [fit_time_LR, fit_time_RF, fit_time_XGB, fit_time_LGB, f
              fit_time_XGB_tuned, fit_time_LGB_tuned, fit_time_CB_tuned],
    'Prediction_time': [predict_time_LR, predict_time_RF, predict_time_XGB, p
              predict_time_XGB_tuned, predict_time_LGB_tuned, predict_time_CB_
```

In [80]:
```
fig, axs = plt.subplots(1,3,figsize=(15,5))
fig.suptitle('Comparison of Machine Learning Algorithms', fontsize=15)

labels = models.sort_values(by='Score')['Model']
values = models.sort_values(by='Score')['Score']
axs[0].bar(labels, values, color = 'g')
axs[0].set_xticklabels(labels, rotation='vertical')
axs[0].set_title('RMSE score')

labels = models.sort_values(by='Training_time')['Model']
values = models.sort_values(by='Training_time')['Training_time']
axs[1].bar(labels, values, color = 'b')
axs[1].set_xticklabels(labels, rotation='vertical')
axs[1].set_title('Training time')

labels = models.sort_values(by='Prediction_time')['Model']
values = models.sort_values(by='Prediction_time')['Prediction_time']
axs[2].bar(labels, values, color = 'orange')
axs[2].set_xticklabels(labels, rotation='vertical')
axs[2].set_title('Prediction time');
```



As we can see, **XGB_tuned** has the best score (the smallest error). **RF_base** score is close but its training and prediction time is very high. Tuning did not improve much the score for the CB model, probably some other parameters should be changed. Even though the difference between XGB and CB scores is not very significant, the CB model is much faster than the XGB. Based on the combination of all 3 factors, we are going to recommend the **CB_tuned as the final model**.

Let's try to tune the learning rate as well for the chosen CB model.

In [64]:
```
d = []
for iterations in [100, 500]:
    for depth in [10, 16]:
```

```
        for learning_rate in [0.01, 0.05, 0.1]:
            CB = CatBoostRegressor(random_state=12345, iterations=iterations,
                                   depth=depth, learning_rate=learnin
            start = time()
            CB.fit(X_train, y_train)
            end = time()
            fit_time_CB_tuned = end - start

            start = time()
            y_pred = CB.predict(X_valid)
            end = time()
            predict_time_CB_tuned = end - start

            CB_score_tuned = mean_squared_error(y_valid, y_pred) ** 0.5
            d.append(
                {
                            'n_estimators': estim,
                            'depth': depth,
                            'CB_score_tuned':  CB_score_tuned,
                            'learning_rate': learning_rate,
                            'fit_time_CB_tuned': fit_time_CB_tuned,
                            'predict_time_CB_tuned': predict_time_CB_tuned
                        }
                    )

best_param = pd.DataFrame(d).nsmallest(1, ['CB_score_tuned'], keep='first')
CB_score_tuned = best_param['CB_score_tuned'].values
fit_time_CB_tuned = best_param['fit_time_CB_tuned'].values
predict_time_CB_tuned = best_param['predict_time_CB_tuned'].values
learning_rate = best_param['learning_rate'].values

best_param
```

Out[64]:

	n_estimators	depth	CB_score_tuned	learning_rate	fit_time_CB_tuned	predict_time_CB_tun
11	500	16	1539.363113	0.1	764.321055	0.1425

The score improved slightly while the training time has increased several folds. Since the speed is important in this task, let's keep the model with the default learning rate. Finally, we will test our model on the test set.

In [65]:

```
CB = CatBoostRegressor(random_state=12345, iterations=500,
                       depth=10, verbose=0)
CB.fit(X_train, y_train)
y_pred = CB.predict(X_test)
CB_score_test = mean_squared_error(y_test, y_pred) ** 0.5
CB_score_test
```

Out[65]: 1531.2740290195325

Both validation and test scores are close to each other, we do not observe overfitting.

Retrain the best tuned model

Now, let's retrain our final model on the whole training set and test it on the test set.

In [66]:

```
X = df.drop('price', axis=1)
y = df['price']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, r

X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

In [77]:
```
CB_final = CatBoostRegressor(random_state=12345, iterations=500,
                                       depth=10, verbose=0)
start = time()
CB_final.fit(X_train, y_train)
end = time()
fit_time_final = end - start

start = time()
y_pred = CB_final.predict(X_test)
end = time()
predict_time_final = end - start

CB_final_score_test = mean_squared_error(y_test, y_pred) ** 0.5
print('test RMSE score, eur:', round(CB_final_score_test,0))
print('Training time, sec:', round(fit_time_final,0))
print('Prediction time, sec:', round(predict_time_final,2))
```

```
test RMSE score, eur: 1524.0
Training time, sec: 27.0
Prediction time, sec: 0.11
```

The score slightly improved due to the bigger train set.

Sanity check

In [68]:
```
LR.fit(X_train, y_train)
y_pred = LR.predict(X_test)
LR_score_test = mean_squared_error(y_test, y_pred) ** 0.5
LR_score_test
```

Out[68]:
```
2698.7226521074763
```

In [71]:
```
round((2698.7226521074763-1523.6258860003888)/2698.7226521074763 * 100, 2)
```

Out[71]:
```
43.54
```

The test score of the Linear regression model, that we use as a baseline to analyze the model quality, is 43.54% higher than our final chosen model. It means that the modeling was useful.

Conclusion

The goal of this project was to develop a model to determine the market value of a car based on historical data. The model had to have a good quality and high offline (training) and online (prediction) speed.

We have completed the following steps in this project:

1. Descriptive statistics. We found missing values, wrong features format, 6 categorical features, possible ouliers in several variables.
2. Data preprocessing. We have converted column names to lower case, changed data type, filled in missing values and removed duplicates.
3. EDA. We have checked for outliers 5 variables, created a new featured (`car_age`) and dropped non-informative columns.
4. Encoding of categorical variables. All categorical variables were one-hot encoded.

5. Splitting data into train, validation and test sets. Data was split into 3 sets to perform best model selection.
6. Standard scaling
7. Model selection. We have compared Linear Regression, Random Forest, XGBoost, LightGBM and CatBoost models. We have also tuned a few hyperparameters for these algorithms. We have chosen the **tuned CatBoost model** based on RMSE score and training and prediction speed.
8. Retrain the best tuned model. The best model was retrained on the whole training data set in order to increase the volume of data it can learn from. The test score has slightly improved thanks to this step.
9. Sanity check. The test score of the Linear regression model, that we use as a baseline to analyze the model quality, is 43.54% higher than our final chosen model. It means that the modeling was useful.

The CatBoost model with the tuned hyperparameters has shown the best results (**test RMSE of 1524, time of training 27 seconds, time of prediction 0.11 seconds**) in terms of both quality and speed of training and prediction. Due to time limit we couldn't test more hyperparameters. This can be done in the future.