Prediction of BMW used cars price

April 30, 2021

Table of Contents

1. Project Motivation

What determines the price of used cars? The value of a car drops right from the moment it is bought and the depreciation increases with each passing year. In fact, only in the first year, the value of a car can decrease from 10 to 60 percent (Ref. 1). Different factors affect the value of a vehicle such as the mileage, model, fuel type, the condition, location, color of a car, etc.

The goals of this projects are:

- explore what factors affect a sale price of used BMW cars and which characteristics are the most important to determine a car value,
- build a prototype of model which predicts price of used BMW cars.

The results of this project may be useful for businesses that work with reselling of used BWM cars and buyers who are looking to purchase a pre-owned BMW car. **Understanding the factors that determine the price and which of them have the biggest affect can help to improve the process of cars valuation.** On the one hand it can help to ensure the data with all important parameters is collected. On the other hand, it can help to save efforts on the initial steps of valuation, as only the most important car parameters can be collected to make a rough estimation of a car price or to assess how good is a particular deal of a car sale.

The developed sale price predictor can be of help during the process of cars search and valuation as it can be used as a tool that provides a quick rough estimation of a car price based on the given input parameters.

2. The Dataset

The dataset that was chosen for the analysis contains information about BMW Used Car Sales and was obtained from DataCamp GitHub repository for careerhub data (Ref. 2). There is no information about the source of this dataset, but a conclusion was made that most likely the data was collected in the United Kingdom. There was a reference about the `tax` column as information about the road tax, in the same time miles per gallon parameter (mpg) is commonly used only in three big countries: the United States, the United Kingdom and Canada. In the United States and Canada there is no road tax as such, but in the United Kingdom there is a road tax. The values of `tax` column in the data are also correspond to tax rates in UK.

2.1. Dataset characteristics

The dataset consists of 10781 observations and 9 columns:

- 3 categorical variables: `model`, `transmission`, `fuelType`,
- 5 numerical variables: `year`, `engineSize`, `mileage`, `tax`, `mpg`,
- continuous target vatable `price`.

2.2. Context of the dataset

Below are the general statistics of car market in UK, it can be helpful during further analysis of factors that affect cars price.

According to the The Society of Motor Manufacturers and Traders (SMMT) in 2020 used car sales in UK decreased by 14.9% with the 6,753 thousands transactions made during the year. In each quarter of 2020 the most popular BMW car was 3 Series model, which took 6th place by the number of transactions among all models of used cars sales (Ref. 3).

According to Statista data platform the BMW share in new cars market sales in UK was fluctuating from 5% to 9% during 2015-2020 (Ref. 4). In 2020 there were 115.5K new BMW passenger cars sold in UK, which is a decline of about 32% year-on-year amid the outbreak of the coronavirus pandemic (Ref. 5) - this is a significantly bigger decline in sales in comparison to used cars market decline. Another interesting statistic is that as of 2016 the average age of cars on the road in the UK was 7.7 years old (Ref. 6).

3. Analysis Plan

The task to be solved in this project is supervised regression problem. In this project we will follow the steps:

- Define performance metrics based on what is known about expected performance of the model,
- Define machine learning algorithms that may be suitable based on what is known about the task,
- Read the dataset and clean the data,
- Explore independent features relation to target, perform necessary transformations/feature engineering,
- Test regression algorithms,
- Tune a model based on the best performing algorithm,
- Implement BMW used cars predictor,
- Summarize the results and next steps to improve the model.

4. Performance Metrics

The choice of the best metric for regression task depends on specifics of the data and the requirements to the model performance. For instance, what is more important: to predict price as accurate as possible on average or not to exceed certain threshold of acceptable error in predictions?

Based on what is known about the data at current stage and understanding of expectations from the model performance, it is suggested to use R-Squared/RMSE as the main metrics and MAE/RMSLE as the supporting metrics. This decision is based on the assumption that big errors in price predictions should be avoided, but in the same time accuracy of predictions is also should be measured.

R-Squared. R-squared (R2) is a statistical measure, known as coefficient of determination, that represents the proportion of the variance for a dependent variable that is explained by an independent variable. R-squared metric determines goodness of fit and is scaled between 0 and 1. The advantage of this metric is ease of interpretation.

Root Mean Squared Error (RMSE). RMSE is the most widely used metric for regression tasks and is the square root of the averaged squared difference between the target value and the value predicted by the model. It is preferred more in some cases because the errors are first squared before averaging which poses a high penalty on large errors. This implies that RMSE is useful when large errors are undesired.

Mean Absolute Error (MAE). MAE is the absolute difference between the target value and the value predicted by the model. The MAE is more robust to outliers and does not penalize the errors as extremely as RMSE. MAE is a linear score which means all the individual differences are weighted equally.

Root Mean squared logarithmic error (RMSLE). Root mean squared logarithmic error can be interpreted as a measure of the ratio between the true and predicted values. In a simple words we can say that RMSLE only cares about the percentual difference and therefore is robust toward outliers as it scales down big errors nullifying their effect.

5. Machine learning algorithms assumptions

Based on what is known about the data characteristics a linear regression and random forest/gradient boosting algorithms may be suitable for this task. During data preparation and exploration we will take into account assumptions about input data that these algorithms have.

Linear regression may be quite powerful algorithm given the input data comply with following assumptions:

- linearity: assumes that the relationship between predictors and target variable is linear,
- no noise: eg. that there are no outliers in the data,
- no collinearity: if you have highly correlated predictors, it's most likely your model will overfit,
- normal distribution: more reliable predictions are made if the predictors and the target variable are normally distributed,
- scale: it's a distance-based algorithm, so preditors should be scaled (Ref. 7).

Random Forest algorithm on the other hand is less demanding as for the data preprocessing:

- decision trees algorithm is insensitive to the absolute values of predictors, so the data does not need to be rescaled or transformed,
- algorithm is great with high dimensional data since it is working with subsets of data,
- algorithm is robust to outliers and non-linear data,
- problem of overfitting is frequently occur and can be solved with hyperparameters tuning.

6. Read the dataset and clean the data

In [268…

```
# Import necessary libraries
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from patsy import dmatrices
import re
from termcolor import colored

from scipy import stats
from scipy.stats import skew, chi2_contingency, norm
from scipy.special import inv_boxcox
from statsmodels.stats.outliers_influence import variance_inflation_factor
import statsmodels.api as sm
from statsmodels.formula.api import ols

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import LinearRegression, ElasticNetCV, SGDRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import SVR
from sklearn.neighbors import KNeighborsRegressor
from sklearn.ensemble import GradientBoostingRegressor, RandomForestRegressor, AdaBoostRegres
from xgboost import XGBRegressor
import lightgbm as lgb
import catboost as cat

from sklearn.metrics import make_scorer, explained_variance_score, mean_squared_error, mean_s
from yellowbrick.regressor import PredictionError, ResidualsPlot
from sklearn.model_selection import RandomizedSearchCV
from sklearn.feature_selection import RFE
seed=1 #Set variable to set random state for reproducibility
```

6.1 Read the data and check descriptive statistics

In [269…

```
# Read dataset
```

```
data = pd.read_csv('bmw.csv')

# Print dataset statistics
print('There are {} observations and {} columns in the DataFrame.'.format(data.shape[0], data
print(colored('\n First 5 rows of the DataFrame:', attrs=['bold']))
display(data.head())
print(colored('Concise summary of the DataFrame:', attrs=['bold']))
print(data.info(memory_usage=False))
print(' ')
print(colored('Descriptive statistics of the numeric features:', attrs=['bold']))
display(data.describe())
```

There are 10781 observations and 9 columns in the DataFrame.

First 5 rows of the DataFrame:

	model	year	price	transmission	mileage	fuelType	tax	mpg	engineSize
0	5 Series	2014	11200	Automatic	67068	Diesel	125	57.6	2.0
1	6 Series	2018	27000	Automatic	14827	Petrol	145	42.8	2.0
2	5 Series	2016	16000	Automatic	62794	Diesel	160	51.4	3.0
3	1 Series	2017	12750	Automatic	26676	Diesel	145	72.4	1.5
4	7 Series	2014	14500	Automatic	39554	Diesel	160	50.4	3.0

Concise summary of the DataFrame:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10781 entries, 0 to 10780
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   model         10781 non-null  object 
 1   year          10781 non-null  int64  
 2   price         10781 non-null  int64  
 3   transmission  10781 non-null  object 
 4   mileage       10781 non-null  int64  
 5   fuelType      10781 non-null  object 
 6   tax           10781 non-null  int64  
 7   mpg           10781 non-null  float64
 8   engineSize    10781 non-null  float64
dtypes: float64(2), int64(4), object(3)None
```

Descriptive statistics of the numeric features:

	year	price	mileage	tax	mpg	engineSize
count	10781.000000	10781.000000	10781.000000	10781.000000	10781.000000	10781.000000
mean	2017.078935	22733.408867	25496.986550	131.702068	56.399035	2.167767
std	2.349038	11415.528189	25143.192559	61.510755	31.336958	0.552054
min	1996.000000	1200.000000	1.000000	0.000000	5.500000	0.000000
25%	2016.000000	14950.000000	5529.000000	135.000000	45.600000	2.000000
50%	2017.000000	20462.000000	18347.000000	145.000000	53.300000	2.000000
75%	2019.000000	27940.000000	38206.000000	145.000000	62.800000	2.000000
max	2020.000000	123456.000000	214000.000000	580.000000	470.800000	6.600000

6.2 Data cleaning

In [270…

```
# Remove leading whitespaces from values in 'model' column
data['model'] = [i.strip() for i in data.model]

# There are only 3 cars with the value `Electric` for the `fuelType`, so we will merge this g
data.fuelType.replace('Electric', 'Other', inplace=True)
```

From the statistics in previous step we can see that there is no missing values in the dataset and data types are set correctly, but some features contain values that should be checked in more detail as there could be incorrect entries:

1. **`mileage` - min value 1.**

After filtering records with mileage<10, I got 67 observations of cars that were produced in years 2019 and 2020. We will assume that values are correct and these are very new cars that were nor used much

2. **`mpg` – min value: 5.5, max value: 470.**

> *Note*: for Hybrid Electric Vehicles `mpg` may be a lot higher. For this type of vehicles there is a miles per gallon of gasoline-equivalent (MPGe) metric. We can think about it as parameter similar to MPG, but instead of presenting miles per gallon of the vehicle's fuel type, it represents the number of miles the vehicle can go using a quantity of fuel with the same energy content as a gallon of gasoline. (Ref. 9).

- `3 Series` model. The research showed that possible `mpg` values are in the range from 27 to 217 mpg (Ref. 8). So `mpg` value of 8.8 doesn't seem to be valid, let's replace it with average `mpg` for this model for `Hybrid` `fuelType`.
- `M6` model. Official MPG for this model is in the range from 27.4 to 28.5, but there are claims that real MPG is in the range from 15 to 26 (Ref. 10). We will assume that the value of 19.1 is correct and will not make any replacements.
- `X3` model. From the research made for `mpg` values we conclude that `mpg` value of 5.5 is incorrect. Since there is no data for `X3` model with `Hybrid` `fuelType`, we will replace mpg value 5.5 with the average `mpg` for the closest model - `X5` model with `Hybrid` fuelType.
- `i3` model - official `mpg` is 149 (Ref. 11). We will assume that values of `mpg` above 400 are incorrect and will replace them.

3. **`engineSize` – min value 0.** The observations that have zero values for `engineSize` are:

- `Petrol/Diesel` `fuelType` cars of `1 Series`, `2 Series`, `3 Series`, `X5` model – there is an error as `engineSize` can not be equal to zero. We will replace `engineSize` values for these observations with the average `engineSize` for each model.
- `Hybrid` / `Other` `fuelType` cars of `i3` model. This model may not have fuel engine (Ref. 11), so we will assume value 0 is valid.

> Note: model `i8` can have a fuel engine, but it also has an electric engine, so it's not correct to compare engine size of electric cars with engine size of conventual fuel type cars. It was decided to create a new feature `engine_category` with the following values: `small_engine`, `medium_engine`, `big_engine`, `electric_engine`, it will categorize cars by engine size and separate cars of electric line.

6.2.1 Replace incorrect `mpg` values

In [271…
```
# `3 Series` model, save average and replace low 'mpg' values
av_3_Series_2017 = data[(data.model.str.contains('3 Series'))&(data.year==2017)&
                        (data.fuelType=='Hybrid')].groupby('year')['mpg'].mean().iloc[0]
data.loc[(data.model.str.contains('3 Series'))&(data.mpg<10), 'mpg'] = av_3_Series_2017

# `X5` model,  save average and replace low 'mpg' values
avg_mpg_hybrid_X5 = data[(data.fuelType =='Hybrid')&(data.model.str.contains('X5'))]['mpg'].m
data.loc[(data.model.str.contains('X3'))&(data.fuelType=='Hybrid'), 'mpg'] = avg_mpg_hybrid_X

# `i3` model, replace mpg above 400 with value of 149
data.loc[(data.mpg>400), 'mpg'] = 149
```

6.2.2 Replace 0 values for `engineSize` for models `1 Series`, `2 Series`, `3 Series`, `X5`

In [272…
```
# Save average 'engineSize' values by 'model' in variables
avg_mpg_engine_1Series = data[(data.model.str.contains('1 Series'))]['engineSize'].mean()
avg_mpg_engine_2Series = data[(data.model.str.contains('2 Series'))]['engineSize'].mean()
avg_mpg_engine_3Series = data[(data.model.str.contains('3 Series'))]['engineSize'].mean()
avg_mpg_engine_X5 = data[(data.model.str.contains('X5'))]['engineSize'].mean()

# Replace 'engineSize' values
data.loc[(data.model.str.contains('1 Series'))&(data.engineSize<1), 'engineSize'] = avg_mpg_e
```

```
data.loc[(data.model.str.contains('2 Series'))&(data.engineSize<1), 'engineSize'] = avg_mpg_e
data.loc[(data.model.str.contains('3 Series'))&(data.engineSize<1), 'engineSize'] = avg_mpg_e
data.loc[(data.model.str.contains('X5'))&(data.engineSize<1), 'engineSize'] = avg_mpg_engine_
```

6.2.3 Create new feature `engine_category` with the following values: small_engine, medium_engine, big_engine, electric_engine

In [273…
```
def create_engine_category (data=data):
    cut_bins = [-1, 0, 1.9, 2.9, 8]
    data['engine_category'] = pd.cut(data.engineSize, bins=cut_bins,
                                     labels=["electric_line", "small_engine", "medium_engine", "b
    data.loc[(data.model.str.contains('i3'))|(data.model.str.contains('i8')), 'engine_categor
    data['engine_category'] = data.engine_category.astype('object')
create_engine_category()
```

7. Explore independent features relation to target, perform transformations and feature engineering

In this section we will analyze following groups of features one by one:

- 1 target continuous numerical variable: `price` ,
- 4 categorical variables: `model` , `transmission` , `fuelType` , `engine_category` ,
- 8 numerical variables: `year` , `mileage` , `tax` , `mpg` , `engineSize` .

7.1 Analyze target variable `Price`

During modeling stage, we will try the linear regression algorithm, so let's check if target variable contains outliers and is normally distributed (we will calculate skewness for the target price and test the most common transformation methods to remove skewness).

In [274…
```
fig, axs = plt.subplots(ncols=2, figsize=(20,2))
sns.boxplot(data.price, ax=axs[0])
sns.distplot(data.price, kde=True,  ax=axs[1])
axs[0].set_title("Boxplot for target feature 'Price'", fontsize=14)
axs[1].set_title("'Distribution of 'Price' variable", fontsize=14)
plt.show()
print("Skewnesss of 'Price' variable distribution before transformation is {}.".format(round(
```


```
Skewnesss of 'Price' variable distribution before transformation is 1.59.
```

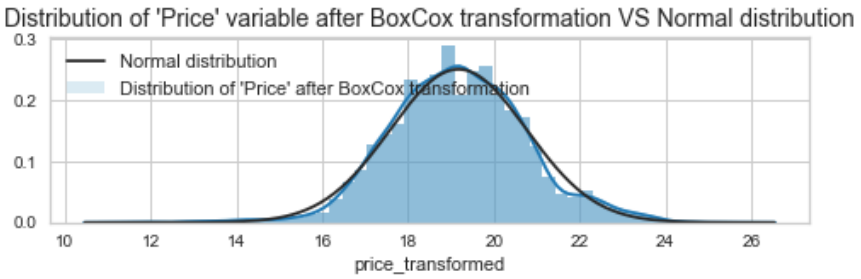
From the figures we can see that the target variable contains many outliers. Few of them, with price over 90K, are especially extreme. Sometimes removing outliers helps to improve the performance of Linear Regression algorithm, we will check this on the modeling stage. Also we can see that target variable is skewed to the right. Let's try the most common transformation methods to remove skewness.

In [275…
```
# Check log transformation for 'Price' variable to remove skewness
transformed = data['price']
data.loc[:, 'price_transformed'] = np.log(transformed)
print("Skewnesss of 'Price' variable after log transformation is {}.".format(round(skew(data.
# Square Root Transform transformation for 'Price' variable to remove skewness
data['price_transformed'] = np.sqrt(transformed)
print("Skewnesss of 'Price' variable after Square Root transformation is {}.".format(round(sk
# BoxCox transformation
transformed = data['price']
data['price_transformed'] = stats.boxcox(transformed)[0]
box_cox_tranform_coef = stats.boxcox(transformed)[1]
print("Skewnesss of 'Price' variable after BoxCox transformation is {}.".format(round(skew(da
```

```
Skewnesss of 'Price' variable after log transformation is -0.24.
```

```
Skewnesss of 'Price' variable after Square Root transformation is 0.7.
Skewnesss of 'Price' variable after BoxCox transformation is 0.01.
```

In [276…
```
# BoxCox transformation gave the best results. Lets plot distribution of 'Price' variable aft
fig = plt.figure(figsize=(8,2))
sns.distplot(data.price_transformed,kde=True)
sns.distplot(data.price_transformed, fit=stats.norm)
plt.title("Distribution of 'Price' variable after BoxCox transformation VS Normal distributio
plt.legend(labels=["Normal distribution","Distribution of 'Price' after BoxCox transformation
plt.show()
```



Good! After transformation the new target variable `price_transformed` is almost normally distributed.

7.2 Analyze correlation of price with categorical variables

7.2.1 Analyze `model` feature

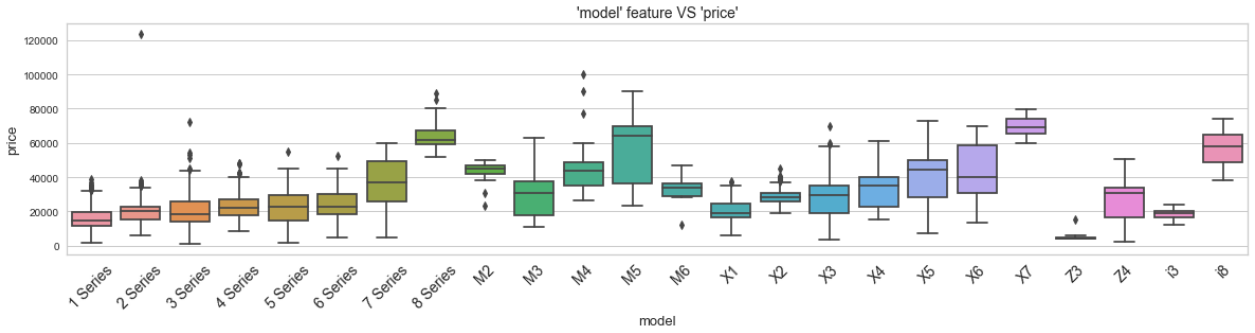
There are following naming conventions for BMW models (Ref. 12):

- Series with even numbers - sportier coupe-style vehicles (2 Series, 4 Series, 6 Series). The lower is a number - the smaller (and less expensive) is a car.
- Series with odd numbers - sedan-style vehicles (3 Series, 5 Series, 7 Series). The lower is a number - the smaller (and less expensive) is a car.
- X Models – SUVs and crossovers (or what BMW refers to as Sports Activity Vehicles and Sports Activity Coupes).
- Z Models – two door roadsters.
- M Models – high-performance version of many BMW models are grouped into the M Models (e.g. the M3 is a high-performance version of the 3 Series sedan). 'M' stands for Motorsport.
- i – electric and plug-in hybrid vehicles.

Let's add a new feature with model groups and look how it correlates with the price.

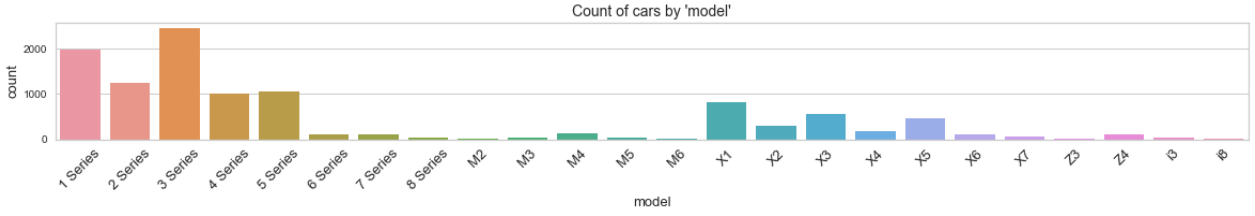
In [277…
```
# Add model_group and model_series features
def create_model_group_and_series (data=data):
    # Extract letter and number from 'model' column
    data['model_group'] = [re.findall('[A-z]+', i)[0] for i in data.model]
    data['model_series'] = [re.findall('[0-9]+', i)[0] for i in data.model]
    data['model_series'] = data.model_series.astype('int')
    # Add 'coupe_style' and 'sedan_style' model groups
    data.loc[(data.model.str.contains('2 Series'))|
             (data.model.str.contains('4 Series')), 'model_group'] = 'coupe_style_small'
    data.loc[(data.model.str.contains('1 Series'))|
         (data.model.str.contains('3 Series')), 'model_group'] = 'sedan_style_small'
    data.loc[(data.model.str.contains('5 Series'))|
         (data.model.str.contains('6 Series')), 'model_group'] = 'sedan_coupe_mid_size'
    data.loc[(data.model.str.contains('8 Series')), 'model_group'] = 'coupe_big_size'
    data.loc[(data.model.str.contains('7 Series')), 'model_group'] = 'sedan_big_size'
create_model_group_and_series()
```

In [278…
```
plt.figure(figsize=(20,4))
ax = sns.boxplot(x='model', y='price', data=data.sort_values(by='model'))
ax.axes.set_title("'model' feature VS 'price'", fontsize=14)
ax.set_xlabel('model', fontsize=13)
ax.set_ylabel('price', fontsize=13)
ax.set_xticklabels(labels=data.sort_values(by='model').model.unique(), size=14, rotation=45)
plt.show()
```



`Model` feature seems to play important role in determining the car price. Almost for all model groups bigger cars seem to be more expensive with the exception of `M` model group. `M2` model on average is more expensive than `M3` and `M4`. Also `M6` model have surprisingly low price on average.

In [279…
```
plt.figure(figsize=(20,2))
ax = sns.countplot('model', data=data.sort_values(by='model'))
ax.axes.set_title("Count of cars by 'model'", fontsize=14)
ax.set_xlabel('model', fontsize=13)
ax.set_ylabel('count', fontsize=13)
ax.set_xticklabels(labels=data.sort_values(by='model').model.unique(), size=13, rotation=45)
plt.show()
```

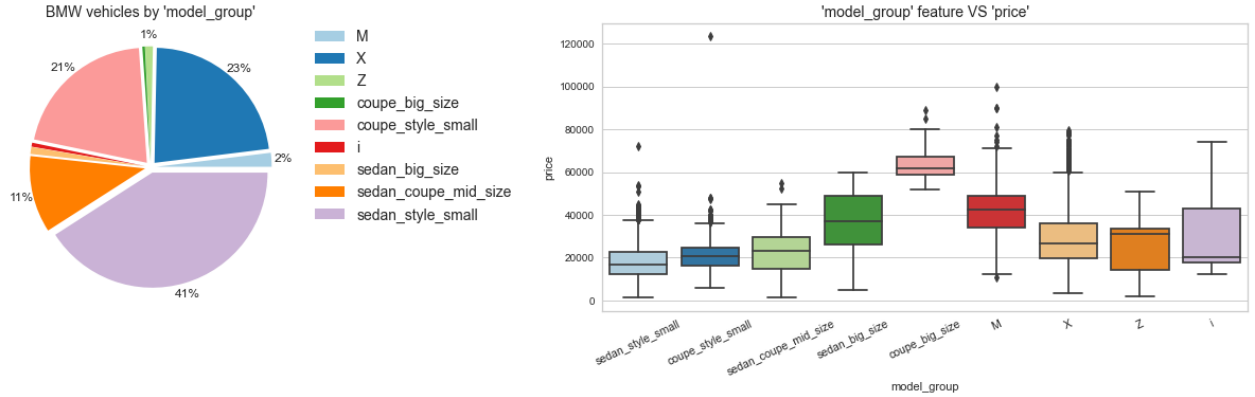


Interestingly, according to the statistics from The Society of Motor Manufacturers and Traders (SMMT) 3 Series model was the most frequently sold BMW car in UK in 2020. In our dataset number of sold `3 Series` models is also the biggest, but difference with count of sold `1 Series` models is not significant.

In [280…
```
def my_autopct(pct):
    return ('%1.0f%%' % pct) if pct > 1 else ''

fig = plt.figure(figsize=(24,5))
ax1 = plt.subplot2grid((1,2),(0,0))
labels, frequencies = np.unique(data.model_group.values, return_counts = True)
plt.pie(frequencies, labels = None, autopct = my_autopct, explode = ([0.05]*9), pctdistance =
plt.title("BMW vehicles by 'model_group'", fontsize=14)
plt.legend(labels, bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.,  fontsize=14)

ax2 = plt.subplot2grid((1,2),(0,1))
ax2 = sns.boxplot(x='model_group', y='price', data=data.sort_values(by='model'))
ax2.axes.set_title("'model_group' feature VS 'price'", fontsize=14)
ax2.set_xticklabels(labels=data.sort_values(by='model').model_group.unique(), size=10.5, rota
plt.show()
```



From the figures we can see that smaller cars from `Series` grouping are the most popular ones (41% are sedan

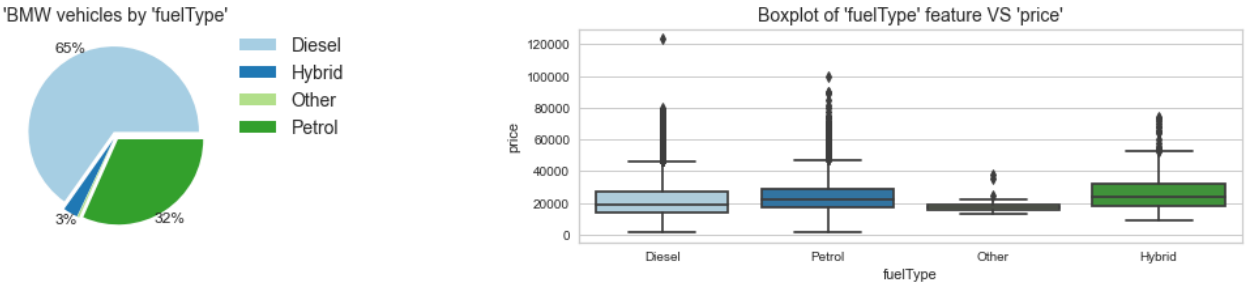
style and 21% of cars arer coupe style, which gives significant 62% in total). These cars also have the lowest average price among other groups. On the second place by popularity are SUVs and crossovers of X series which take share of 23% and on the third place by share are mid-size sedan type cars of 5 Series model.

Electric (i models), performance (M models), roadsters (Z models) and big coupe style (8 Series) vehicles take share of 5% in total. Cars from these groups have different distributions against price. What is interesting is that electric cars from of i series have quite wide range in price (from 1595 to 88980), but comparatively low median 23980, so half of cars in this category have price below 23980.

7.2.2 Analyze `fuelType` feature

In [281…
```
fig = plt.figure(figsize=(20,3))
ax1 = plt.subplot2grid((1,2),(0,0))
labels, frequencies = np.unique(data.fuelType.values, return_counts = True)
plt.pie(frequencies,labels = None, autopct = my_autopct, pctdistance = 1.1,  explode=([0.05]*
plt.legend(labels, bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.,  fontsize=14)
plt.title("'BMW vehicles by 'fuelType'",fontsize=14)

ax2 = plt.subplot2grid((1,2),(0,1))
sns.boxplot(x='fuelType', y='price', data=data)
plt.title("Boxplot of 'fuelType' feature VS 'price'", fontsize=14)
plt.show()
```

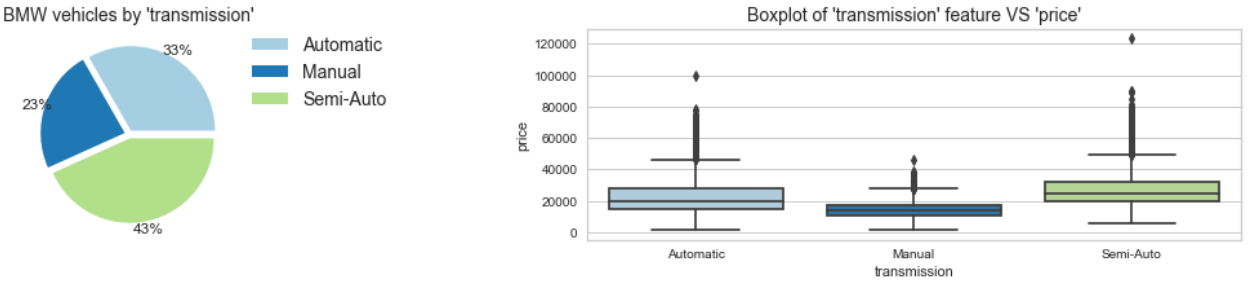


We can see for the figures that range of prices from 25 to 75 percentile for the cars with `Diesel`, `Hybrid` and `Petrol` engines does not differ much. Cars with `Other` fuel type have more narrow range, but it's less then 1% of cars of this category in the data. `fuelType` feature seems to not influence price much. In all plots we don't see specific patterns of price correlation from `fuelType`.

7.2.3 Analyze `transmission` feature

In [282…
```
fig = plt.figure(figsize=(20,3))
ax1 = plt.subplot2grid((1,2),(0,0))
labels, frequencies = np.unique(data.transmission.values, return_counts = True)
plt.pie(frequencies,labels = None, autopct = '%1.0f%%', pctdistance = 1.1, explode=([0.05]*3)
plt.title("BMW vehicles by 'transmission'", fontsize=14)
plt.legend(labels, bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0., fontsize=14)

ax2 = plt.subplot2grid((1,2),(0,1))
sns.boxplot(x='transmission', y='price', data=data)
plt.title("Boxplot of 'transmission' feature VS 'price'", fontsize=14)
plt.show()
```



In [283…
```
fig, axs = plt.subplots(ncols=2, figsize=(17,4))
sns.stripplot(x='transmission', y='price', data=data, hue='engine_category', alpha=0.7, ax=ax
axs[0].set_title("'transmission' feature VS 'price' colored by 'engine_category'", fontsize=1
sns.stripplot(x='transmission', y='price', data=data, hue='model_series', alpha=.5,  ax=axs[1
```

```
axs[1].set_title("'transmission' feature VS 'price' colored by 'model_series'", fontsize=14)
plt.show()
```

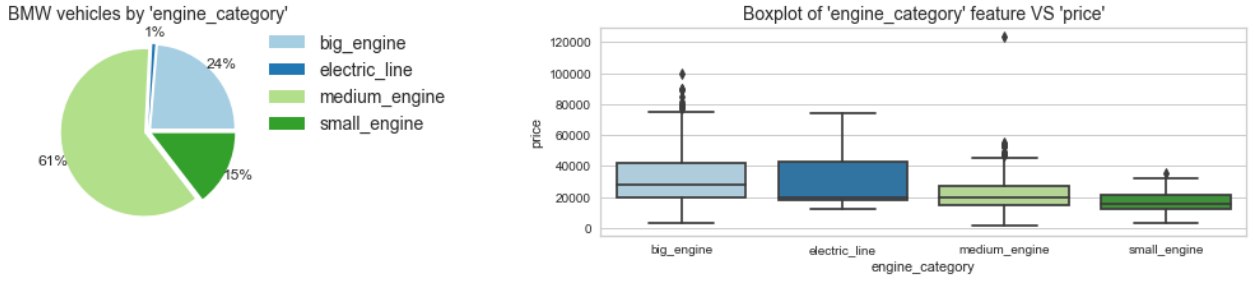
Figures above show that `transmission` feature have an impact on price, cars with `Automatic` and `Semi-Auto` transmission have wider price range and on average tend to be more expensive than cars with `Manual` transmission. Also cars in a higher price segment tend to belong to bigger models, have bigger engines and have `Automatic` / `Semi-Auto` transmission.

In [284…
```
# Let's add boolean feature _big_engine_auto_or_semi_auto, it can be useful on the modeling s
def create__big_engine_auto_semi_auto (data=data):
    data.loc[(data.transmission!='Manual')&(data.engine_category=='big_engine'), '_big_engine
    data._big_engine_auto_semi_auto.fillna(0, inplace=True)
    data._big_engine_auto_semi_auto=data._big_engine_auto_semi_auto.astype('int')
create__big_engine_auto_semi_auto()
```

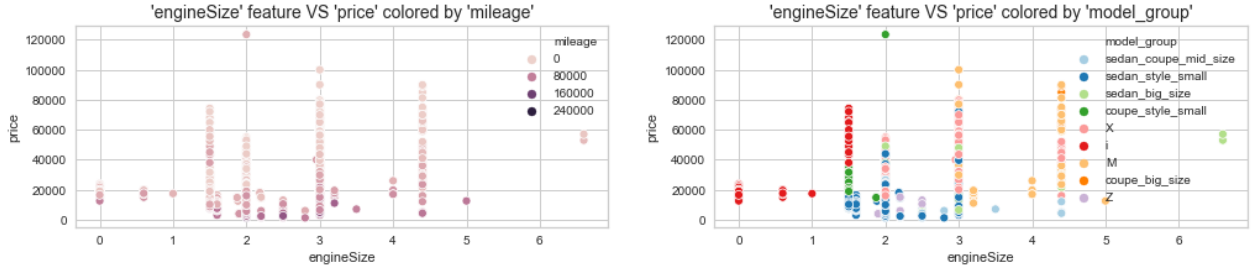
7.2.4 Analyze `engine_category` feature

In [285…
```
fig = plt.figure(figsize=(20,3))
ax1 = plt.subplot2grid((1,2),(0,0))
labels, frequencies = np.unique(data.engine_category.values, return_counts = True)
plt.pie(frequencies,labels = None, autopct = '%1.0f%%', pctdistance = 1.15, explode=([0.05]*4
plt.title("BMW vehicles by 'engine_category'", fontsize=14)
plt.legend(labels, bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.,  fontsize=14)

ax2 = plt.subplot2grid((1,2),(0,1))
sns.boxplot(x='engine_category', y='price', data=data.sort_values(['engine_category']))
plt.title("Boxplot of 'engine_category' feature VS 'price'", fontsize=14)
plt.show()
```



In [286…
```
fig, axs = plt.subplots(ncols=2, figsize=(17,3))
sns.scatterplot(x='engineSize', y='price', data=data, hue='mileage', ax=axs[0])
axs[0].set_title("'engineSize' feature VS 'price' colored by 'mileage'", fontsize=14)
sns.scatterplot(x='engineSize', y='price', data=data, hue='model_group', ax=axs[1])
axs[1].set_title("'engineSize' feature VS 'price' colored by 'model_group'", fontsize=14)
plt.show()
```



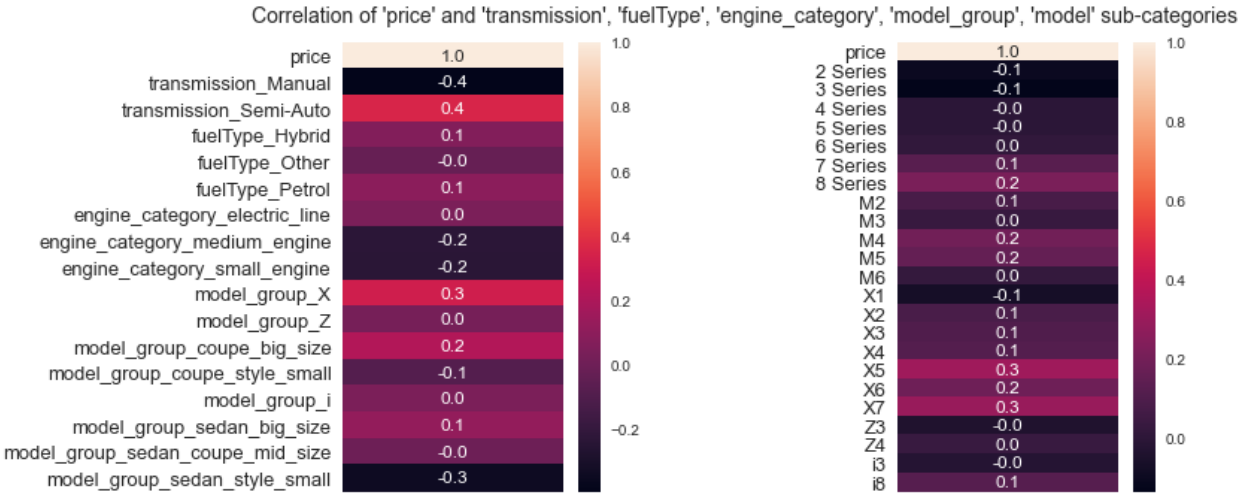
We can see that price range varies depending of `engine_category` of a car, bigger is an engine - higher is an average price and wider a price range becomes. But there are also cars with price below average among the ones with medium/big engines size, they tend to have high `mileage` values and belong to models groups `sedan_coupe_mid_size` and `sedan_style_small`.

7.2.5 Correlation matrix for encoded categorical variables

In [287…

```
fig = plt.figure(figsize=(10,17))
ax1 = plt.subplot2grid((3,2),(0,0))
corr_matrix = pd.concat([data.price, categorical_without_model], axis=1).corr()
sns.heatmap(pd.DataFrame(corr_matrix['price']), annot=True, fmt = '.1f')
plt.xticks([])
plt.yticks(fontsize=13)

ax2 = plt.subplot2grid((3,2),(0,1))
corr_matrix = pd.concat([data.price, categorical_model], axis=1).corr()
sns.heatmap(pd.DataFrame(corr_matrix['price']), annot=True, fmt = '.1f')
plt.xticks([])
plt.yticks(fontsize=13)
plt.subplots_adjust(wspace=1,top=0.96)
plt.suptitle("Correlation of 'price' and 'transmission', 'fuelType', 'engine_category', 'mode
plt.show()
```



From the correlation coefficients we can see that some features do no not have correlation with the price. We need to pay attention to this on the modeling stage, test feature selection and regularization methods during model building.

7.2.6 Run 1-way ANOVA test for categorical variables

Since there is no correlation to price of quite a few encoded categorical variables, let's check if there is a statistically significant difference in price for cars grouped by each categorical variable.

The null hypothesis for this test is that difference in mean price by group is statistically significant (variation in price between different groups can be a random noise due to the sampling effect).

In the results of this test we will compare the p-value to our chosen alpha. As a rule of thumb in our case we will choose alpha=0.05. If our p-value is smaller than 0.05, then we can reject the null hypothesis in favor of the alternative: this means that at least one group price mean is significantly different. If our p-value is larger than 0.05, then we cannot reject our null hypothesis and we cannot accept our alternative.

In [288…

```
categorical_columns=['model', 'transmission', 'fuelType', 'engine_category', 'model_group']
anova_results = {}
for i in categorical_columns:
    lm = ols('price~{}'.format(i), data=data).fit()
    table = sm.stats.anova_lm(lm)
    anova_results[i] = [table['PR(>F)'][0], table['F'][0]]
anova_results = pd.DataFrame.from_dict(anova_results, orient='index', columns = ['P_value', '
display(anova_results)
```

	P_value	F_value
model	0.000000e+00	451.428790
transmission	0.000000e+00	1257.908494
fuelType	8.071119e-37	57.423452
engine_category	0.000000e+00	1040.182002
model_group	0.000000e+00	527.807991

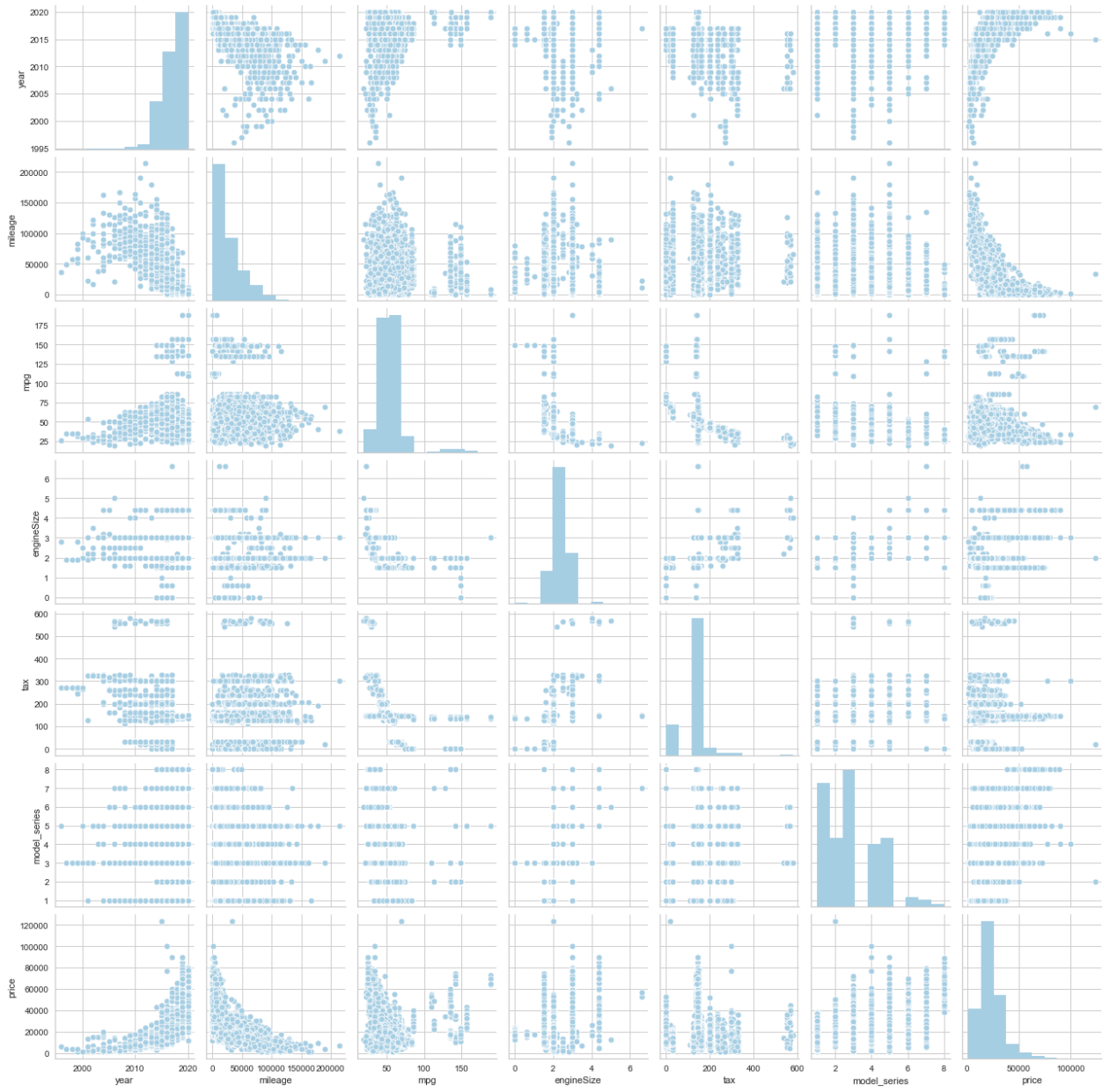
Based on 1-way ANOVA test result we can reject the null hypothesis for all variables and conclude that for each categorical feature at least one group have statistically significant impact on the price. The most important features according to F-value (statistic value for significance of adding model terms) are:

- `transmission` (F=1258),
- `engine_category` (F=1032),
- `model_group` (F=528).

7.3 Analyze correlation of `price` with numerical variables

In [289…

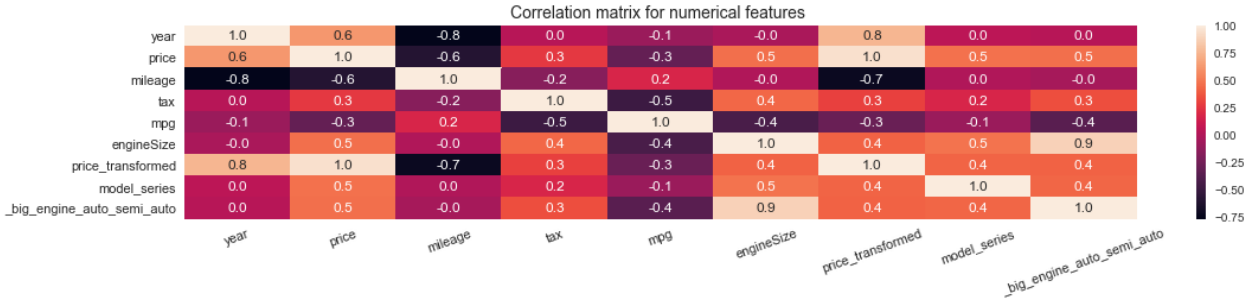
```
# Explore pairplot for numerical variables from the initial data: 'year', 'mileage', 'mpg', '
sns.pairplot(data[['year', 'mileage', 'mpg', 'engineSize', 'tax', 'model_series', 'price']])
plt.show()
```



From the plot we can see that there is:

- a strong correlation between `price` and `year` / `mileage`. In the same time `mileage` has a strong negative correlation with the `year`. Relationship of `year` and `price` variables can be described as an exponential function and between `mileage` and `price` as an inverse square function.
- moderate correlation between the `price` and `mpg` / `engineSize` variables.
- as for `tax` feature - we can't see a pattern on how `tax` affect price. If there is a relationship between road tax and price - it`s non-linear.

In [290…
```
# Check correlation matrix
corr_matrix = data.corr()
plt.figure(figsize=(18,3))
plt.title('Correlation matrix for numerical features ', fontsize=14)
fig = sns.heatmap(corr_matrix, annot=True, fmt = '.1f')
plt.xticks(fontsize=11, rotation=20)
plt.yticks(fontsize=11)
plt.show()
```



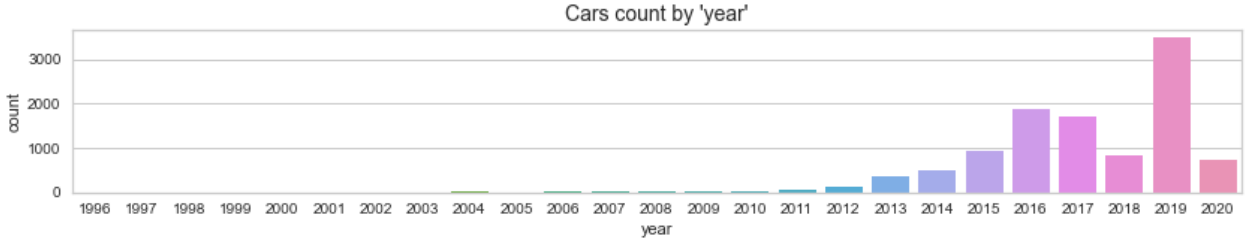
From the matrix above we can see that all numerical features are moderately correlated with the target variable `price`:

- the highest correlation of `price` is with `year`, `mileage` features (0.6 and -0.6 coefficients),
- `engineSize`, `model_series` and `_big_engine_auto_semi_auto` have a a bit lower correlation (0.5 coefficients),
- `mpg` and `tax` features have the lowest correlation with the `price` (-0.3 and 0.3 coefficients) - probably lower correlation coefficient for these features is connected to the fact that relation is not linear.

Independent variables `year` and `mileage`, `_big_engine_auto_semi_auto` and `engineSize` have strong correlation between each other. To check if there is a multicollinearity problem, for all numeric features I calculated correlation between explanatory variables and target using `variance_inflation_factor` metric. The `variance_inflation_factor` is 2.7 for `mileage` and 5.3 for `EngineSize`. A value greater than 5 of indicates potentially severe correlation between a given explanatory variable and other explanatory variables in the model, so potentially there is multicollinearity problem for `EngineSize` feature, especially considering that the model did not include categorical feature `Engine category`.

7.3.1 Analyze `year` feature

In [291…
```
plt.figure(figsize = (14,2))
sns.countplot(data.year)
plt.title("Cars count by 'year'", fontsize=14)
plt.show()
```

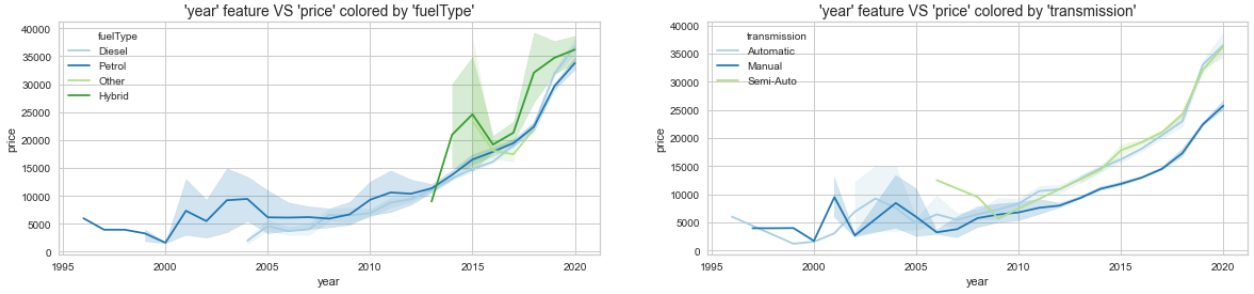


Most cars in our dataset is pretty new, which make sense since average age of cars on the road in the UK was 7.7

years old in 2016.

In [292…
```
fig, axs = plt.subplots(ncols=2, figsize=(20,4))
sns.lineplot(x='year', y='price', data=data, hue='fuelType', ax=axs[0])
axs[0].set_title("'year' feature VS 'price' colored by 'fuelType'", fontsize=14)

sns.lineplot(x='year', y='price', data=data, hue='transmission', ax=axs[1])
axs[1].set_title("'year' feature VS 'price' colored by 'transmission'", fontsize=14)
plt.show()
```



Cars that are 10 years old or newer have tendency to have higher price for newer cars. As for older cars (10 years old and more) - correlation between `year` and `price` doesn't seem to be strong (on average cars from 2000-2005 have even higher prices than cars from 2005-2010).

First figure shows that the `Hybrid` fuel type cars that are presented in the data were all produced in 2013 or later and tend to be more expensive on average in comparison to other fuel types cars.

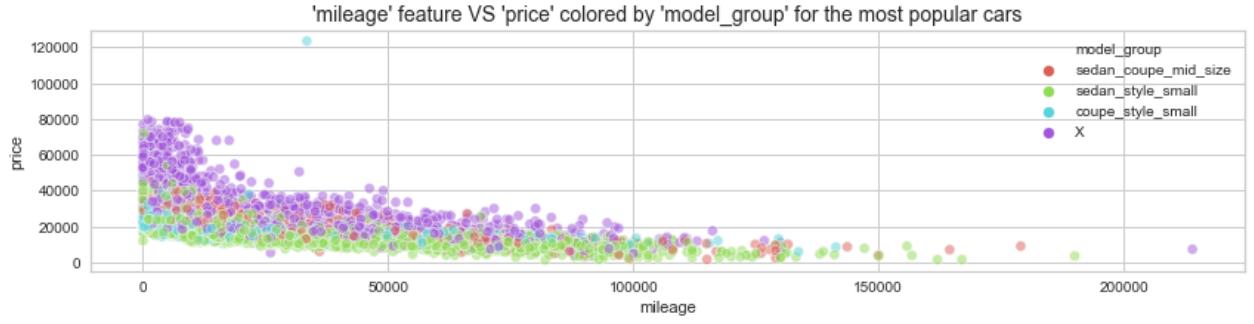
The second figure shows that `transmission` type doesn't have much impact for older cars (from 2010 and older), while newer cars with `Automatic` and `Semi-auto` transmission tend to be more expensive than the ones with `Manual` transmission.

Looks like features affect pricing of cars differently for the older cars (from 2010 and older) and for the newer cars (newer than 2010). Let's use the average age of cars in UK as a threshold to introduce a new boolean feature `_not_old_car` .

In [293…
```
# Add feature '_not_old_car' for cars produced after 2013
def create__not_old_car (data=data):
    data.loc[(data.year>2013), '_not_old_car']='1'
    data._not_old_car.fillna(0, inplace=True)
    data._not_old_car=data._not_old_car.astype('int')
create__not_old_car()
```

7.3.2 Analyze `mileage` feature

In [294…
```
plt.figure(figsize=(14,3))
most_popular_cars = data[data.model_group.isin(['sedan_style_small', 'coupe_style_small', 'se
popular_cars_colors = sns.color_palette("hls", 4)
sns.scatterplot(x='mileage', y='price', data=most_popular_cars, hue='model_group', alpha=0.5,
plt.title("'mileage' feature VS 'price' colored by 'model_group' for the most popular cars",
plt.show()
```



In [295…
```
# The value of price >  100K for `coupe_style_small` car looks not right, let's replace it wi
mean_price_2Series = data[(data.model=='2 Series')&(data.year==2015)&(data.engine_category=='
```

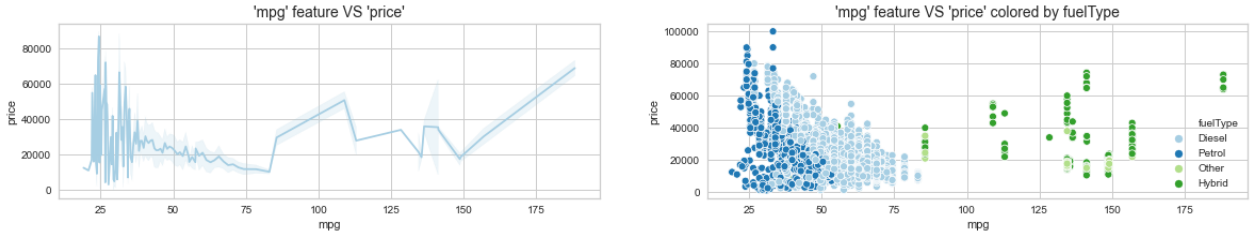
```
    (data.transmission=='Semi-Auto')&(data.price<90000)]['price'].mean()
data.loc[3638, 'price'] = mean_price_2Series
```

To improve readability only data for the most popular model groups were filtered and plotted. Figure shows that selected model groups seem to be in different price segments and there is a visible correlation of `mileage` and `price` variables within the model groups.

7.3.3 Analyze `mpg` feature

In [296…
```
fig, axs = plt.subplots(ncols=2, figsize=(20,3))
sns.lineplot(x='mpg', y='price', data=data, ci='sd', ax=axs[0])
axs[0].set_title("'mpg' feature VS 'price'", fontsize=14)

sns.scatterplot(x='mpg', y='price', data=data, ci='sd', hue='fuelType', ax=axs[1])
axs[1].set_title("'mpg' feature VS 'price' colored by fuelType", fontsize=14)
plt.show()
```



From the plots we can see that there is a non-linear relationship between `mpg` and `price`.

- For `fuelType = Diesel` there seem to be a negative correlation: bigger `mpg` - lower the price,
- For `fuelType = Petrol` negative correlation of `mpg` and `price` are even stronger,
- For `fuelType = Hybrid / Other` there seem to be no strong correlation of `mpg` value and price.

Let's create 2 new features derived from `mpg` feature to reflect found pattern.

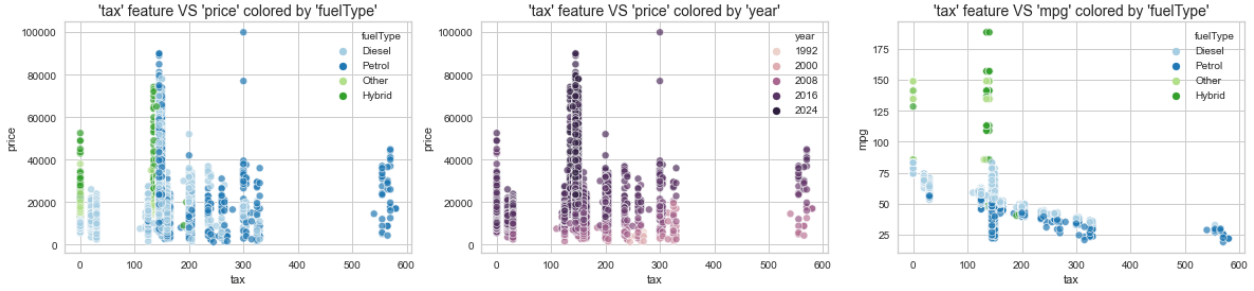
In [297…
```
# Add 2 new features derived from 'mpg' feature
def create_mpg_1_and_mpg_2 (data=data):
    data.loc[data.fuelType=='Petrol', 'mpg_1'] = data.mpg
    data.loc[data.fuelType=='Diesel', 'mpg_2'] = data.mpg
    data.replace(np.nan, 0, inplace=True)
create_mpg_1_and_mpg_2()
```

7.3.4 Analyze `tax` feature

In [298…
```
fig, axs = plt.subplots(ncols=3, figsize=(20,4))
sns.scatterplot(x='tax', y='price', data=data, hue='fuelType', alpha=0.7, ax=axs[0])
axs[0].set_title("'tax' feature VS 'price' colored by 'fuelType'", fontsize=14)

sns.scatterplot(x='tax', y='mpg', data=data, hue='fuelType', alpha=0.7, ax=axs[2])
axs[2].set_title("'tax' feature VS 'mpg' colored by 'fuelType'", fontsize=14)

sns.scatterplot(x='tax', y='price', data=data, hue='year', alpha=0.7, ax=axs[1])
axs[1].set_title("'tax' feature VS 'price' colored by 'year'", fontsize=14)
plt.show()
```



From the figures we can see that there is no strong correlation between `tax` feature and `price`. The first figure shows that cars with `Hybrid` and `Other` fuel types have lower or no tax which was expected.

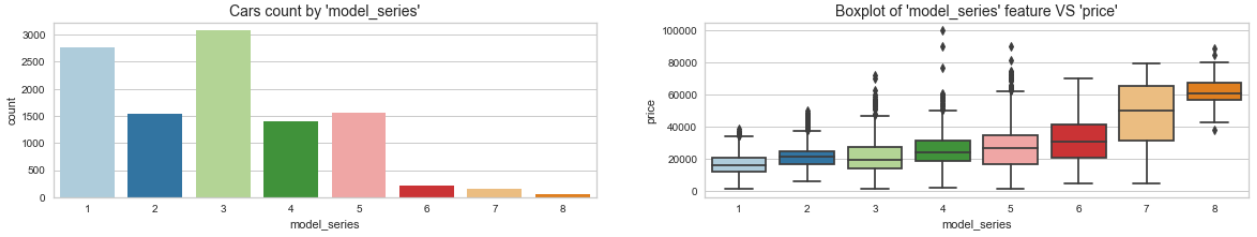
The second figure shows that older cars tend to have higher road taxes, but surprisingly some relativity new cars are in the cluster with the highest tax rate (>500), these cars also have very low `mpg` values.

The last figure shows that there is a linear relation between `tax` and `mpg` variables: the lower the `mpg` - the higher the `tax`, cars with alternative `fuelType` have especially high `mpg`.

7.3.5 Analyze `model_series` feature

In [299…]
```
fig, axs = plt.subplots(ncols=2, figsize=(20,3))
sns.countplot(data.model_series, ax=axs[0])
axs[0].set_title("Cars count by 'model_series'", fontsize=14)

sns.boxplot(x='model_series', y='price', data=data, ax=axs[1])
axs[1].set_title("Boxplot of 'model_series' feature VS 'price'", fontsize=14)
plt.show()
```



From the figures we can see that `model_series` feature has moderate correlation with `price`: bigger models tend to have higher price. What is interesting that among bigger models there are still cheap options.

7.4 Features exploration summary

Target variable `price`.

- `Price` contains a lot of outliers and it's distribution is not normal. Box-Cox transformation was applied and new target `price_transformed` was added.

Categorical variables `model`, `model_group`, `transmission`, `fuelType`, `engine_category`.

- we extracted features `model_group` and `model_series` from `model`,
- `engine_category` feature analysis showed that cars with bigger engines tend to have higher price. But there are also cars with price below average among the ones with medium/big engines size, they tend to have high `mileage` values and belong to models groups `sedan_coupe_mid_size` and `sedan_style_small`.
- for all categorical features at least one group has statistically significant impact on the price. But in correlation matrix we saw that some models/model groups sub-categories don't correlate with the price, so it's necessary to explore if removing them can improve a predictive model.
- the most important features according to F-value are `transmission` (F=1258), `engine_category` (F=1032), `model_group` (F=528).

Numerical variables: `year`, `mileage`, `tax`, `mpg`, `engineSize`, `model_series`.

- `year` and `mileage` features have the strongest correlation with price:
  - newer cars tend to have higher price, especially with `Hybrid` fuel type and `Automatic` / `Semi-Auto` transmission. We have relativly new cars in the dataset (from 2013 and newer). Correlation of `year` to `price` seem to be stronger for newer cars, so new boolean feature `_not_old_car` is added (for cars produced in 2013 or later).
  - different model lines seem to be in different price segments and there is a visible correlation of `mileage` and `price` within model groups.
- `engineSize` and `model_series` features have moderate correlation with price.
- `mpg` and `tax` features have the lowest correlation with the price due to non-linear relation.
  - as for `mpg` feature there is a pattern of it's correlation to price depending on `fuelType`:

- for `Petrol` / `Diesel` fuel type cars there is a moderate negative correlation of `mpg` feature and price,
- for cars with `Hybrid` / `Other` fuel type - there is almost no correlation with `price` .

- There is a weak correlation between price and tax rate. Another finding is that `tax` feature is correlated with `mpg` and also older cars seem to have higher tax rates.

8. Model Development. Test regression algorithms

In [300…
```
# Split data to train and test sets
train_df, test_df = train_test_split(data,
                                     test_size=0.2,
                                     random_state=seed,
                                     stratify=data[["model"]])
```

In [301…
```
# Separate features and target
y_train = train_df['price_transformed']
X_train_df = train_df.drop(['price', 'price_transformed'], axis=1)
X_test_df = test_df.drop(['price', 'price_transformed'], axis=1)
y_test = test_df['price_transformed']
```

In [302…
```
# Standardize the data: StandardScaler for numerical features and OneHotEncoding for categori
scaler=StandardScaler()
encoder = OneHotEncoder(drop='first')
X_train_num =  scaler.fit_transform(X_train_df.select_dtypes(exclude='object'))
X_test_num = scaler.transform(X_test_df.select_dtypes(exclude='object'))
X_train_cat = encoder.fit_transform(X_train_df.select_dtypes(include='object'))
X_test_cat = encoder.transform(X_test_df.select_dtypes(include='object'))

# Combine preprocces features
X_train = np.concatenate((X_train_num, X_train_cat.toarray()), axis=1)
X_test = np.concatenate((X_test_num, X_test_cat.toarray()), axis=1)
```

In [303…
```
# Initiate regression models
KNeighbors = KNeighborsRegressor(n_neighbors=3)
lin_reg = LinearRegression()
SGD = SGDRegressor(random_state = seed)
GBR = GradientBoostingRegressor(random_state = seed)
elasticNet = ElasticNetCV(random_state = seed)
DecisionTree = DecisionTreeRegressor(random_state = seed)
RandomForest = RandomForestRegressor(n_estimators = 10, random_state = seed)
SVR_reg = SVR(kernel = 'rbf')
AdaBoost = AdaBoostRegressor(random_state = seed)
XGBR = XGBRegressor(random_state = seed)
lgb_reg = lgb.LGBMRegressor(random_state = seed)
cat_reg = cat.CatBoostRegressor(random_state = seed, silent=True)

models=[lin_reg, SGD, elasticNet, KNeighbors, SVR_reg, DecisionTree, RandomForest, GBR, AdaBo
```

In [304…
```
# Define a function to calculate metrics
def calculate_metrics(y_pred, model, y=y_test, printing=True):
    explained_variance = round(explained_variance_score(y, y_pred),4)
    MAE = round(mean_absolute_error(y, y_pred),4)
    RMSE = round(np.sqrt(mean_squared_error(y, y_pred)),4)
    RMSLE = round(mean_squared_log_error(y, y_pred),6)
    if printing==True:
        print("Results of {} model: \n - explained_variance: {},".format(model, explained_var
        print(" - mean_absolute_error: {}, \n - root_mean_squared_error: {}, \n - mean_square
                                                                                .format
    else:
        metrics = [model, explained_variance, MAE, RMSE, RMSLE]
        return metrics
```

In [305…
```
def create_dataframe_with_predictions(model, X_test, box_cox_tranform_coef = box_cox_tranform
    """
Parameters expected by the function:
    - model – instance of model to apply to make predictions,
    - name_of_dataframe – defines the name of new dataframe that will be created, dataframe w
    - box_cox_tranform_coef – coefficient for price transformation, default value is variable
Function also calculates and prints metrics of model performance: explained_variance, MAE, RM
    """
    # Inverse BoxCox transformation for the price
```

```
    y_pred_price_by_model = inv_boxcox(model.predict(X_test), box_cox_tranform_coef)
    # Create a table with real price, predicted price and features
    dataframe = pd.concat([pd.DataFrame(y_test_price).reset_index(drop=True),
                           pd.DataFrame(y_pred_price_by_model),
                           X_test_df.reset_index(drop=True)], axis=1)
    dataframe.columns=['Real_price', 'Predicted_price']+[i for i in X_test_df.columns]
    dataframe['%error'] = abs(dataframe.Real_price - dataframe.Predicted_price)/dataframe.Rea
    # Calculate  metrics
    calculate_metrics(y_pred=y_pred_price_by_model, model=model, y=y_test_price)
    return(dataframe)
```

In [306…

```
# Fit Regressors and print performance results
results={}
for i in models:
    i.fit(X_train, y_train)
    y_pred_reg = i.predict(X_test)
    result = calculate_metrics(y_pred=y_pred_reg, model=i, y=y_test, printing=False)
    results[str(result[0]).split('(')[0]] = result[1:]
display(pd.DataFrame.from_dict(results, orient='index', columns = ['R2', 'MAE', 'RMSE', 'RMSL
```

	R2	MAE	RMSE	RMSLE
LinearRegression	0.9367	0.2935	0.3937	0.000388
SGDRegressor	0.9005	0.3445	0.4938	0.000607
ElasticNetCV	0.9316	0.3018	0.4093	0.000418
KNeighborsRegressor	0.9450	0.2538	0.3672	0.000348
SVR	0.9494	0.2443	0.3523	0.000315
DecisionTreeRegressor	0.9289	0.2855	0.4173	0.000449
RandomForestRegressor	0.9514	0.2371	0.3449	0.000311
GradientBoostingRegressor	0.9454	0.2726	0.3657	0.000340
AdaBoostRegressor	0.8442	0.5164	0.6552	0.001074
XGBRegressor	0.9604	0.2228	0.3116	0.000253
LGBMRegressor	0.9584	0.2285	0.3194	0.000267
<catboost.core.CatBoostRegressor object at 0x000001BAD6D99970>	0.9630	0.2153	0.3011	0.000238

From the results we can see that CatBoost and XGBoost algorithms have the best performance. Since CatBoostRegressor has the lowest loss, we will explore it in more detail. CatBoost is a relatively new open-source machine learning algorithm that was built upon the theory of decision trees and gradient boosting. What is interesting about it is that it also accepts input data with minimal preprocessing: categorical features can be not encoded (it's possible to pass text as input) and scaling for numerical features is also not required.

Let's transform price to pounds and look at results in absolute values for the best performing algorithm CatBoost.

In [307…

```
# Calculate  metrics on training set
predicted_train = cat_reg.predict(X_train)
calculate_metrics(y_pred=predicted_train, model=cat_reg, y=y_train)
```

```
Results of <catboost.core.CatBoostRegressor object at 0x000001BAD6D99970> model:
 - explained_variance: 0.9761,
 - mean_absolute_error: 0.1817,
 - root_mean_squared_error: 0.247,
 - mean_squared_log_error: 0.000149.
```

Errors calculated on the training set are slightly smaller than the ones that we got on the test set. The difference is not significant, so looks like the model does not overfit.

In [308…

```
y_test_price = inv_boxcox(y_test, box_cox_tranform_coef)
predicted_cat_reg=create_dataframe_with_predictions(model=cat_reg, X_test=X_test)
```

```
Results of <catboost.core.CatBoostRegressor object at 0x000001BAD6D99970> model:
 - explained_variance: 0.9613,
 - mean_absolute_error: 1452.784,
 - root_mean_squared_error: 2257.0741,
 - mean_squared_log_error: 0.008612.
```

Seems like we got pretty good results, features describe 96% of variance in price variable. Standard error of predictions in absolute values is about 1453 pounds. Also, I checked generated dataframe and `%error` column descriptive statistics: on average the model is wrong about a car price by 6.6% and standard deviation of the mean error is 9.5%.

Let's look at predictions where the model have the biggest loss.

In [309…
```
# Rows with error in predicted prices > 40%
predicted_cat_reg[predicted_cat_reg['%error']>0.4]
```

Out[309…

	Real_price	Predicted_price	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	engine_categ
915	3500.0	4981.598541	1 Series	2009	Automatic	103735	Petrol	205	44.1	1.6	small_eng
1197	3076.0	13810.490483	Z4	2014	Manual	31074	Petrol	205	41.5	2.0	medium_eng
1253	2995.0	4467.530637	3 Series	2010	Manual	101104	Petrol	165	44.8	2.0	medium_eng
1977	7500.0	10884.271786	1 Series	2013	Manual	29000	Diesel	120	58.9	2.0	medium_eng

I filtered out cars with the similar characteristics to the ones that are in the table above (same model/year/transmission), analyzed their characteristics and found that prices for these cars differ significantly in comparison to other similar cars, maybe we miss some information that can explain these variations.

9. Tune model based on CatBoostRegression algorithm

9.1 Train CatBoostRegressor using target without transformation

Since decision trees algorithm is insensitive to the specific values of predictors, we don't have to worry about 'non-normality'"and skewed distribution of the target variable. Here we will try to use real, not transformed, values of `price` .

In [310…
```
# Redefine target dataset
y_train = train_df['price']
y_test = test_df['price']

cat_reg_without_transformation = cat.CatBoostRegressor(random_state = seed, loss_function='RM
cat_reg_without_transformation.fit(X_train, y_train)
predicted_cat_reg_without_transformation = create_dataframe_with_predictions(model=cat_reg_wi
                                                                             X_test=X_test, box_cox
```

```
Results of <catboost.core.CatBoostRegressor object at 0x000001BAD8482A60> model:
 - explained_variance: 0.9627,
 - mean_absolute_error: 1464.8996,
 - root_mean_squared_error: 2216.6259,
 - mean_squared_log_error: 0.008831.
```

The results show that the main metrics (explained variance and root mean squared error) are improved, but MAE/RMSLE became worse. That means that the new version of the model makes less big mistakes, but on average is less accurate with prediction in relative terms. Since we defined that big errors should be avoided, we will keep the changes and from now on will use this version of model.

9.2 Fine-tune hyperparameters for the CatBoostRegressor

Let's see if we can improve model performance with hyper-parameter tuning. In this project we will use RandomizedSearchCV, as it less computationally expensive than GridSearchCV and allows to check wider range of parameters. Disadvantage of this method is that search of best parameters is not exhaustive and the best combination of hyperparameters can be skipped.

We will explore the most common hyperparameters:

- number of iterations (iterations) - the maximum number of trees that can be built, defines how complex a model can be, a higher number of trees gives better performance, but model training takes more time.
- learning rate (learning_rate) - used for reducing the gradient step. It affects the overall time of training: the smaller the value, the more iterations are required for training. Increasing the learning rate can decrease overfitting, but also can cause a model to converge too quickly to a suboptimal solution.
- tree depth (depth) – setting depth of trees.
- random strength - the amount of randomness to use for scoring splits when a tree structure is selected, can be used to avoid overfitting a model.

In [311…
```
# Create custom scorer for tracking by RandomizedSearchCV - mean_squared_error
scorer = make_scorer(mean_squared_error, greater_is_better = False)

# Set parameter grid
parameters = {'depth'         : [6, 7, 8, 9, 10, 11],
              'learning_rate' : [0.03, 0.04, 0.5, 0.06, 0.07],
              'iterations'    : [900, 1000, 1100, 1200, 1300, 1400, 1500],
               'random_strength': [0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2]
                 }

# Search for optimal parameters
grid_search = RandomizedSearchCV(estimator = cat_reg_without_transformation,
                           param_distributions = parameters,
                           scoring = scorer,
                           cv = 5,
                           n_iter=30,
                           verbose=1,random_state=seed)
grid_search.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 30 candidates, totalling 150 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 150 out of 150 | elapsed: 28.0min finished
```

Out[311…
```
RandomizedSearchCV(cv=5,
                   estimator=<catboost.core.CatBoostRegressor object at 0x000001BAD8482A60>,
                   n_iter=30,
                   param_distributions={'depth': [6, 7, 8, 9, 10, 11],
                                        'iterations': [900, 1000, 1100, 1200,
                                                       1300, 1400, 1500],
                                        'learning_rate': [0.03, 0.04, 0.5, 0.06,
                                                          0.07],
                                        'random_strength': [0.25, 0.5, 0.75, 1,
                                                            1.25, 1.5, 1.75,
                                                            2]},
                   random_state=1,
                   scoring=make_scorer(mean_squared_error, greater_is_better=False),
                   verbose=1)
```

In [312…
```
best_params = grid_search.best_params_
print(best_params)
cat_reg_tuned = cat.CatBoostRegressor(**best_params, random_state = seed, loss_function='RMSE
cat_reg_tuned.fit(X_train, y_train)
# Check model performance
predicted_cat_reg_tuned = create_dataframe_with_predictions(model=cat_reg_tuned,
                                                            X_test=X_test, box_cox
```

```
{'random_strength': 0.25, 'learning_rate': 0.06, 'iterations': 1300, 'depth': 7}
Results of <catboost.core.CatBoostRegressor object at 0x000001BAD6063730> model:
 - explained_variance: 0.9631,
 - mean_absolute_error: 1435.5959,
 - root_mean_squared_error: 2202.7952,
 - mean_squared_log_error: 0.008453.
```

As a result of fine-tuning the model's hyperparameters explained variance and root mean squared error have slightly improved, in the same there is also an improvement in MAE and MSLE metrics.

9.3 Feature selection

On the features exploration step we saw from correlation coefficients that some features have very little or no correlation with the price. Cutting off irrelevant features reduces computational cost, can allow algorithms to run more efficiently and even improve performance as some algorithms can be misled by irrelevant input features.

Chosen algorithm CatBoost is based on decision trees theory and should be robust towards excessive input features due to process of random sampling of features subsets during training, but it's worth exploring if we can reduce number of features.

9.3.1 Search for optimal number of features using recursive feature elimination (RFE)

To define which features we can drop and which features are the most relevant it is suggested to use Recursive Feature Elimination. RFE is a wrapper-type feature that works by fitting the given machine learning algorithm, ranking features by importance, discarding the least important features and re-fitting the model. This process is repeated until a specified number of features remains. Advantage of such approach is that features importance can change after removing certain feature, so recursive refitting allows to define the least important feature on each step of the process.

By conducting exhaustive search for the best number of features with RFE I found that reducing number of features doesn't improve RMSE much (some very insignificant improvements may be random). Results also showed that a number of features can be reduced to up to 8 without losing much in model performance (loss increases from 2203 to 2292).

9.3.2 Explore 8 the most important features selected by RFE

In [313…

```
# Check results in absolute values for model trained on 8 features
rfe_8 = RFE(cat_reg_tuned, n_features_to_select = 8)
X_train_selected_8 = rfe_8.fit_transform(X_train, y_train)
X_test_selected_8 = rfe_8.transform(X_test)
cat_reg_selected_8=cat.CatBoostRegressor(**best_params, random_state = seed,  loss_function =
cat_reg_selected_8.fit(X_train_selected_8, y_train)
predicted_cat_reg_reduced_features_8 = create_dataframe_with_predictions(cat_reg_selected_8,
                                                                         X_test=X_test_select
```

```
Results of <catboost.core.CatBoostRegressor object at 0x000001BAD7108610> model:
 - explained_variance: 0.9601,
 - mean_absolute_error: 1525.5789,
 - root_mean_squared_error: 2292.0882,
 - mean_squared_log_error: 0.0099.
```

The results for the model trained only on 8 features are worse, but are still pretty good, 96% of price variance is explained by only 8 features!

The great feature of Cat Boost model is the extensive possibilities to interpret the results:

- one of the attributes of CatBoostRegressor is `feature_importances_` allows to get the individual importance values for each of the input features. As in our project we have non-ranking loss function, `PredictionValuesChange` method will be used for the scores calculation. Importance scores will be normalized (all the importances will add up to 100). One of disadvantages of this metric is that it may give misleading results for ranking objectives (it might put groupwise features into the top, even though they have a little influence on the resulting loss value). (Ref. 13)
- we can plot `PredictionValuesChange` for each feature. It will calculate how much on average the prediction changes if the feature value changes. The bigger the value of the importance the bigger on average is the change to the prediction value if this feature is changed. X-axis of the plot will contain values of the feature divided into buckets. Y-axis will contain following values for each bucket: average target value, average prediction, average prediction with substituted feature.
- we can excess features contributions values and see how each feature contributes to the prediction on the examples.

Let's see what 8 features were selected and explore contribution of each of them on the example of train set.

In [314…

```
# Get features_names
object_features = [i for i in X_train_df.select_dtypes(exclude='object').columns]
numeric_features = [i for i in encoder.get_feature_names(X_train_df.select_dtypes(include='ob
features_names = object_features + numeric_features

# Print selected features names
selected_features = pd.DataFrame(X_train, columns=features_names).iloc[:, rfe_8.support_]
features_importance = pd.DataFrame(zip(selected_features.columns,
                                       cat_reg_selected_8.get_feature_importance()),
                                       columns=['param_name','importance_score']).sort_values
```

```
print('Selected features and their importance score:')
display(features_importance)
```

```
Selected features and their importance score:
```

	param_name	importance_score
0	year	29.691007
4	model_series	17.770697
1	mileage	15.780130
2	mpg	13.157151
5	_big_engine_auto_semi_auto	7.885853
7	model_group_X	7.347829
3	engineSize	4.513381
6	mpg_2	3.853952

In [315…
```
# Define DataFrame with features contributions
features_contribution = cat_reg_selected_8.get_feature_importance(data=cat.Pool(X_train_selec
                                                                  fstr_type='ShapValues',
                                                                  prettified=True).iloc[:, :-1]
features_contribution.columns=[selected_features.columns]

print('Features contribution values:')
display(features_contribution.head(3))
```

```
Features contribution values:
```

	year	mileage	mpg	engineSize	model_series	_big_engine_auto_semi_auto	mpg_2	mo
0	-2195.852447	-869.390787	-1682.071670	449.043804	-343.642141	1441.966750	-132.383960	-
1	3401.443620	2702.252951	350.918453	-2545.511446	-3411.523956	-470.568343	175.039810	
2	3284.858551	2775.161897	6970.351832	-966.980189	996.966485	-3634.498375	-26.026595	

In [316…
```
print('Trainset data:')
display(train_df.head(3))
```

```
Trainset data:
```

	model	year	price	transmission	mileage	fuelType	tax	mpg	engineSize	engine_category	price_transform
566	3 Series	2016	17498.0	Semi-Auto	27146	Diesel	150	56.5	3.0	big_engine	18.6412
2681	X1	2019	25450.0	Semi-Auto	5021	Petrol	145	40.9	1.5	small_engine	19.8857
5686	X3	2019	34882.0	Semi-Auto	6000	Petrol	145	30.4	2.0	medium_engine	20.9773

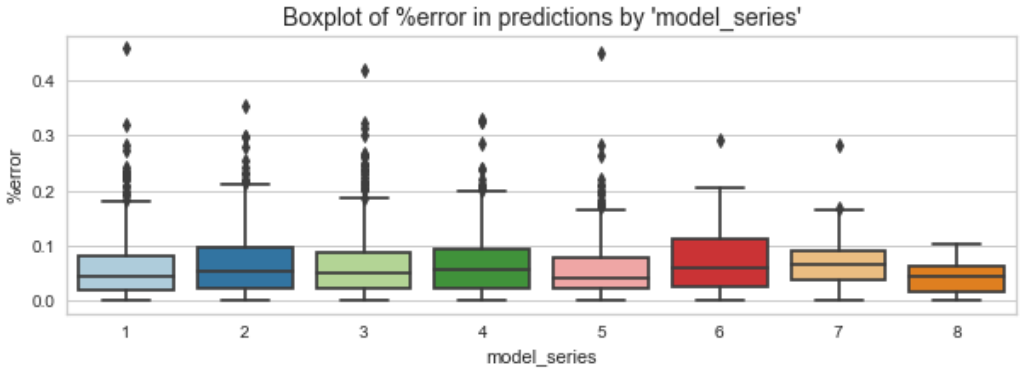
By comparing how different features impact the price on the example of first two observations we can see that:

- `year` value 2016 negatively affect the base price, while value 2019 adds to price.
- `mileage` value for the second car is very low, it has significant positive impact on price. Higher mileage for the first car affect the price negatively
- `mpg` for the first car is slightly higher and this value has significant negative impact on price. In the same time `mpg` for the second car does not differ much, but has positive affect. il's possible that 'good' and 'bad' `mpg` for different models are different.
- Small engine in the second observation has a great negative contribution to the price, while relatively big engine for the first cat doesn't affect the price much.

- Both cars relate to small model series line, so it affects price negatively. Impact from the smallest model line for the second car is much more significant.

9.4 Explore predictions of the best performing model based on CatBoostRegressor algorithm

In [317…
```
pred_without_outlier = predicted_cat_reg_tuned[predicted_cat_reg_tuned['%error']<1]
plt.figure(figsize=(10,3))
sns.boxplot(x='model_series', y = '%error', data = pred_without_outlier)
plt.title("Boxplot of %error in predictions by 'model_series'", fontsize=14)
plt.show()
```

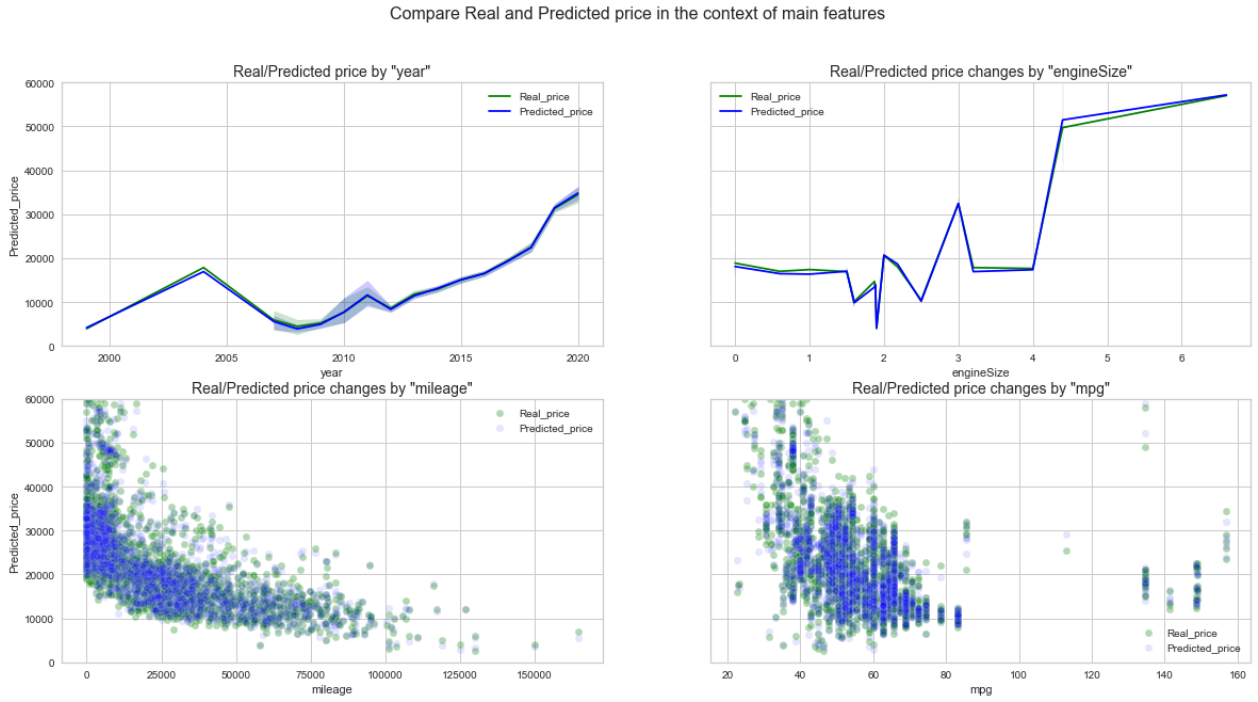


From the boxplot we can see that for all model series 75% of predictions have error about 10% or less, but there are many outliers in `%error` column. The widest interquartile range of `%error` values is for 6 model series. Median of errors are the highest for 6 and 7 model series.

In [318…
```
print('{}% of predictions have error above 10% and {}% have error above 15%.'.format(
    round(predicted_cat_reg_tuned[predicted_cat_reg_tuned['%error']>0.10].Real_price.count()/
          predicted_cat_reg_tuned.Real_price.count(), 3)*100,
    round(predicted_cat_reg_tuned[predicted_cat_reg_tuned['%error']>0.15].Real_price.count()/
          predicted_cat_reg_tuned.Real_price.count(), 3)*100))
```

```
19.8% of predictions have error above 10% and 7.8% have error above 15%.
```

In [319…
```
# Plot new '%error'  and predictions in context with other most important features
fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(20,10), sharey=True)
plt.suptitle('Compare Real and Predicted price in the context of main features',  fontsize=16
sns.lineplot(x='year', y='Real_price', data = pred_without_outlier, color='green', ax=axs[0,
             label="Real_price")
sns.lineplot(x='year', y='Predicted_price', data = pred_without_outlier, color='blue', ax=axs
             label="Predicted_price")
sns.lineplot(x='engineSize', y='Real_price', data = pred_without_outlier, color='green',  ax=
             label="Real_price")
sns.lineplot(x='engineSize', y='Predicted_price', data = pred_without_outlier, color='blue',
             label="Predicted_price")
sns.scatterplot(x='mileage', y='Real_price', data = pred_without_outlier, color='green', alph
                label="Real_price")
sns.scatterplot(x='mileage', y='Predicted_price', data = pred_without_outlier, color='blue',
                label="Predicted_price")
sns.scatterplot(x='mpg', y='Real_price', data = pred_without_outlier, color='green', alpha=0.
                label="Real_price")
sns.scatterplot(x='mpg', y='Predicted_price', data = pred_without_outlier, color='blue', alph
                label="Predicted_price")
axs[0, 0].set_ylim((0,60000))
axs[0, 1].set_ylim((0,60000))
axs[0, 0].set_title('Real/Predicted price by "year"', fontsize=14)
axs[0, 1].set_title('Real/Predicted price changes by "engineSize"', fontsize=14)
axs[1, 0].set_title('Real/Predicted price changes by "mileage"', fontsize=14)
axs[1, 1].set_title('Real/Predicted price changes by "mpg"', fontsize=14)
plt.show()
```



Figures show that there is a very slight deviation of average predicted price from the real price for the older cars (approximately older than 2007). Also for cars that were produced in about 2007-2012 years the interval of confidence in predictions is wider. This can be connected to the fact that there is not much data for the older cars.

As for figures with `mileage` and `mpg` features – seems like predictions are more conservative in comparison to real observations and algorithm does not catch some of the most extreme cases as predicted price differ from real for observations with very low or high values for `mileage` / `mpg` .

10. Implement BMW used cars predictor using CatBoostRegressor algorithm

In this section we will define a function which predicts car price and will test it on some real world examples from the web site https://www.autotrader.co.uk.

In [320…
```
# Define function to predict cars prices
def predict_car_price (model: object, year: int, transmission: object, mileage, fuelType: obj
                       engineSize: float, prediction_model=cat_reg_tuned):

    car = pd.DataFrame({'model': [model], 'year': int(year), 'transmission': [transmission],
                        'fuelType': [fuelType], 'tax': int(tax), 'mpg': float(mpg), 'engineSi
    # Add features
    create_engine_category (data=car)
    create_model_group_and_series (data=car)
    create__big_engine_auto_semi_auto (data=car)
    create__not_old_car (data=car)
    create_mpg_1_and_mpg_2 (data=car)
    # Scale and encode fetures, combine preprocces features
    car_num = scaler.transform(car.select_dtypes(exclude='object'))
    cart_cat = encoder.transform(car.select_dtypes(include='object'))
    car_processed = np.concatenate((car_num, cart_cat.toarray()), axis=1)
    predcition = prediction_model.predict(car_processed)
    return car, predcition
```

In [321…
```
# Prepare input data
cars = [{'model': '2 Series',  'year': 2017,  'transmission':  'Automatic',  'mileage': 31000
         'mpg': 48.7, 'engineSize': 2},
        {'model': 'X3',  'year': 2016,  'transmission':  'Automatic',  'mileage': 40500, 'fue
         'mpg': 49.6, 'engineSize': 2},
        {'model': 'M4',  'year': 2017,  'transmission':  'Automatic',  'mileage': 45884, 'fue
         'mpg': 46.3, 'engineSize': 2},
        {'model': '3 Series',  'year': 2016,  'transmission':  'Manual',  'mileage': 41351, '
         'mpg': 45.6, 'engineSize': 1.5}]

real_samples = pd.DataFrame.from_dict({'https://www.autotrader.co.uk/car-details/202104281944
              'https://www.autotrader.co.uk/car-details/202104281946523': [18990],
```

```
                'https://www.autotrader.co.uk/car-details/202104291960876': [21000],
                'https://www.autotrader.co.uk/car-details/202104291961070': [11500]}, orient='i
```

In [322…
```
# Make a prediction of prices with tuned version of CatBoostRegressor
real_samples['predicted_price'] = np.arange(0, 4)
for num, i in enumerate(cars):
    _, value = predict_car_price(**i, prediction_model=cat_reg_without_transformation)
    real_samples.iloc[num, 1] = np.round(value)
```

In [323…
```
display(real_samples)
calculate_metrics(y_pred=real_samples.predicted_price, model=cat_reg_without_transformation,
```

	real_price	predicted_price
https://www.autotrader.co.uk/car-details/202104281944760	17000	16750.0
https://www.autotrader.co.uk/car-details/202104281946523	18990	18745.0
https://www.autotrader.co.uk/car-details/202104291960876	21000	20867.0
https://www.autotrader.co.uk/car-details/202104291961070	11500	13387.0

```
Results of <catboost.core.CatBoostRegressor object at 0x000001BAD8482A60> model:
 - explained_variance: 0.9341,
 - mean_absolute_error: 628.75,
 - root_mean_squared_error: 961.897,
 - mean_squared_log_error: 0.005877.
```

The sample for this test is not large enough to make any conclusions, but surprisingly RMSE and MSE for these 4 examples are even smaller than on the testing set. It's recommended to test the model on the bigger sample of real data.

11. Conclusion and Recommendations

The aim of this project was to explore what factors affect used cars price the most and to implement a prototype of used BMW cars price predictor which can provide an assistance for the potential buyers and sellers to define a fair price of a car with minimal time efforts.

Exploration of factors that affect price of used BMW cars

The results on the test set show that only 8 parameters can explain almost 96% of variance in price:

- Year when car was produced (importance_score – 29.7),
- Number in model name (importance_score – 17.7),
- Mileage (importance_score – 15.8),
- Mileage per gallon parameter (importance_score – 13.2),
- Boolean parameter `_big_engine_auto_semi_auto` which is True when a car has automatic or semi-automatic transmission and engine size above 2.9 liters (importance_score – 7.9),
- Boolean parameter which is True when a car belongs to `X` model line (importance_score – 7.3),
- Size of engine in liters (importance_score – 4.5),
- Added feature `mpg_2` which is filled if `fuelType` is `Diesel` (importance_score – 3.9).

In other words we can conclude that all the features except `tax`, that were in the initial dataset proved are important for price prediction: `year`, `model`, `mileage`, `mpg`, `transmission`, `engineSize` and `fuelType`.

Prototype of price predictor for used BMW cars

In this project several machine learning algorithms were tested and different data preprocessing approaches were explored. New features were extracted from the initial data. Data was spit into training (80%) and testing (20%) set.

Predictive model based on CatBoostRegressor algorithm had the best results on the test set:

- explained_variance: 0.9631,

- mean_absolute_error: 1435.5959,
- root_mean_squared_error: 2202.7952,
- mean_squared_log_error: 0.008453.

To check the model performance on the real data, 4 inputs of samples from autotrader.co.uk web-site were manually added and price were predicted for them. The results we got are: RMSE of £962 and MAE of £629 which is better than on training or testing set. The sample size for this test is not large enough to make any conclusions, so it's recommended to check model performance on the bigger sample and analyze the results.

Limitations of the price prediction model:

1. Results of the best performing model on the test set are: mean absolute error of £1436 and mean squared error of £2203. There are about 20% of predictions that have error above 10% and almost 8% of predictions have error above 15%. It's important to define what are minimal requirements for the model performance, because the current model's predictions might not be reliable enough.
2. CatBoostRegressor algorithm provided the best results on the test set, but is more computationally expensive than other simpler algorithms with slightly bigger loss. Amount of data to be processed and the expected execution time should be taken into consideration when applying price prediction model.
3. Cars sales data quickly gets irrelevant with time: demand on the market changes, new models are released and with each time period cars loose in value because their age increases, etc. The price predictor should be updated regularly to provide relevant predictions.

Possible ways to improve performance of the model:

1. Explore in greater depth hyperparameter tuning for CatBoostRegressor.
2. Some values of price are not explained by the parameters that we have in the dataset. It's possible that more information about cars can be retrieved that can explain unclear variance in price, for example:
   - assessment of car condition,
   - more information about important car characteristics (engine model, color, if navigation is installed, leather/textile interior, if car is convertible, number of doors, if a car was in accident and what was a severity of damage, if a car had more than one owner, etc),
   - information about additional packages/additions installed in a car or assessment of value of additions/packages,
   - location, etc.
3. Collect more data (for example in the dataset there is not much data for the older cars).
4. Explore model performance on the real unseen data and tune it in accordance to received results.

12. References

(1) Car depreciation
(2) DataCamp GitHub repository for careerhub data
(3) The Society of Motor Manufacturers and Traders.Used car sales
(4) Monthly BMW car market share in the United Kingdom (UK) from January 2015 to December 2020
(5) Monthly sales volume of BMW passenger cars in the United Kingdom (UK) from January 2019 to December 2020
(6) The Ultimate List of UK Car Stats 2020
(7) Linear regression assumptions
(8) BMW 3-Series running costs and mpg
(9) Calculation of MPG for Hybrid Electric Vehicles
(10) Honest John. Detailed reviews of new and used cars. Real MPG of BMW M6 model
(11) Wikipedia. BMW i3 model
(12) Understanding BMW naming conventions
(13) Deep Dive into Catboost Functionalities for Model Interpretation