Predicting Used Car Prices with Machine Learning

A complete data science project from data collection to model evaluation

I'm planning to sell my car which is a 4-year-old Volkswagen polo. Used cars are usually sold on a website called "sahibinden" in Turkey. "Sahibinden" means "from the owner" although there are many dealers using this website to sell or buy used cars. The most critical part of selling a used car is to determine the optimal price. There are many websites that give you a price for used cars but you still want to search the market before setting the price. Moreover, there are other factors which affect the price such as location, how fast you want to sell the car, smoking in the car and so on. Before you post your ad on the website, it is best to look through the price of similar cars. However, this process might be exhausting because there are too many ads online. Therefore, I decided to take advantage of the convenience offered by machine learning to create a model that predicts used car prices based on the data available on "sahibinden". It will not only help solve my problem of determining a price for my car but also help me learn and practice many topics related to data science.

This project is divided into 5 subsections as follows:

- Data collection
- Data cleaning
- Exploratory Data Analysis
- Regression Model and Evaluation
- Further improvement

All the data and codes are available on a github repository. Feel free to use or distribute.

1. Data Collection

There are more than six thousand Volkswagen polo for sale on "sahibinden" website. I had to do web scraping to collect data from the website. I'm not an expert on web scraping but I've learned enough to get what I need. I think it is very important to learn web scraping to a certain level if you want to work or are working in data science domain because data is not usually served on a plate to us. We have to get what we need.

I used beautiful soup which is a python library for pulling data out of HTML and XML files. The syntax is pretty simple and easy to learn. There are a few important details that you need to pay attention especially if the data is listed on several pages.

Always import the dependencies first:

```
import pandas as pd
import numpy as np
import requests
from bs4 import BeautifulSoup as bs
```

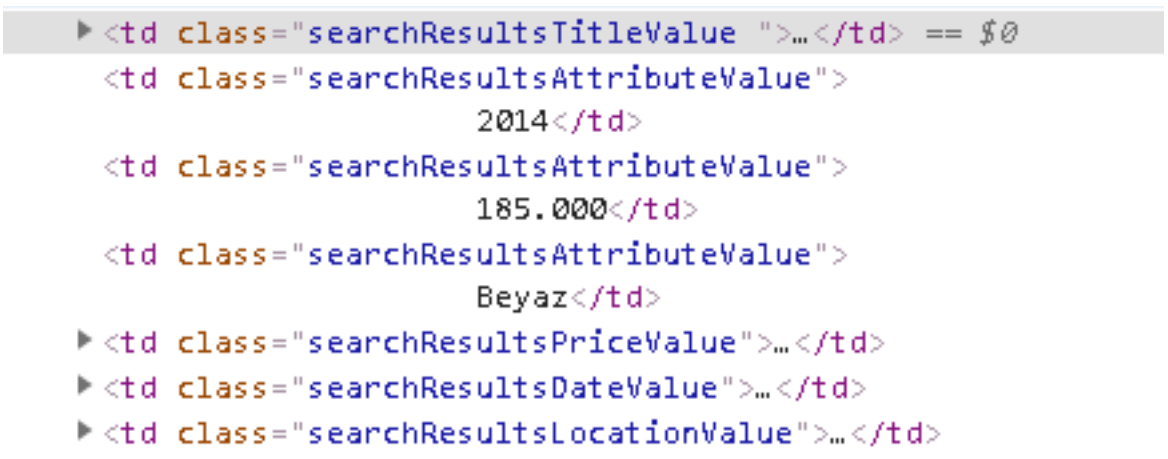
I used the **get()** method of python's **requests** library to retrieve data from the source and store it in a variable. Then I used beautiful soup to extract and organize the content of this variable. Since the data is on several pages, I had to create list to help parse through different pages and also initiate empty lists to save the data.

```
#initiate empty lists to save data
model_info = []
ad_title = []
year_km_color = []
price = []
ad_date = []
location = []
#create lists to parse through pages
page_offset = list(np.arange(0,1000,50))
min_km = [0, 50000, 85000, 119000, 153000, 190000, 230000]
max_km = [50000, 85000, 119000, 153000, 190000, 230000, 500000]
```

The maximum number of ads displayed on a page is 50. In order to scrape data for about six thousand cars, I needed to iterate over 120 pages. First, I organized the code in a for loop to extract data from 120 pages. However, after the process was done, I found out that data was repeated after first 1000 entries. Then, I decided to group data into smaller sections which would not exceed 1000 entries per group so I used 'km' criteria to differentiate groups. I created nested for loops to extract data for about six thousands cars as below:

```
for i, j in zip(min_km, max_km):
    for page in page_offset:
        r = requests.get(f'https://www.sahibinden.com/volkswagen-
polo?pagingOffset={page}&pagingSize=50&a4_max={j}&sorting=date_asc&a4_min=
{i}', headers=headers)
        soup = bs(r.content,'lxml')
        model_info +=
soup.find_all("td",{"class":"searchResultsTagAttributeValue"})
        ad_title +=
soup.find_all("td",{"class":"searchResultsTitleValue"})
        year_km_color +=
soup.find_all("td",{"class":"searchResultsAttributeValue"})
        price += soup.find_all("td",{"class":"searchResultsPriceValue"})
        ad_date += soup.find_all("td",{"class":"searchResultsDateValue"})
        location +=
soup.find_all("td",{"class":"searchResultsLocationValue"})
```

At each iteration, the base url is modified using the values in page_offset, max_km and min_km lists to go to next page. Then the content of website is decomposed into pre-defined lists based on the tag and class. The classes and tags in html can be displayed by inspecting the website on the browser.



```
▶ <td class="searchResultsTitleValue ">…</td> == $0
  <td class="searchResultsAttributeValue">
                      2014</td>
  <td class="searchResultsAttributeValue">
                      185.000</td>
  <td class="searchResultsAttributeValue">
                      Beyaz</td>
▶ <td class="searchResultsPriceValue">…</td>
▶ <td class="searchResultsDateValue">…</td>
▶ <td class="searchResultsLocationValue">…</td>
```

HTML of “sahibinden” website

After getting the content of html, I extracted the text part:

```
model_info_text = []
for i in range(0,6731):
    model_info_text.append(model_info[i].text)
```

This process was done for each list and then I combined the lists to build a pandas DataFrame:

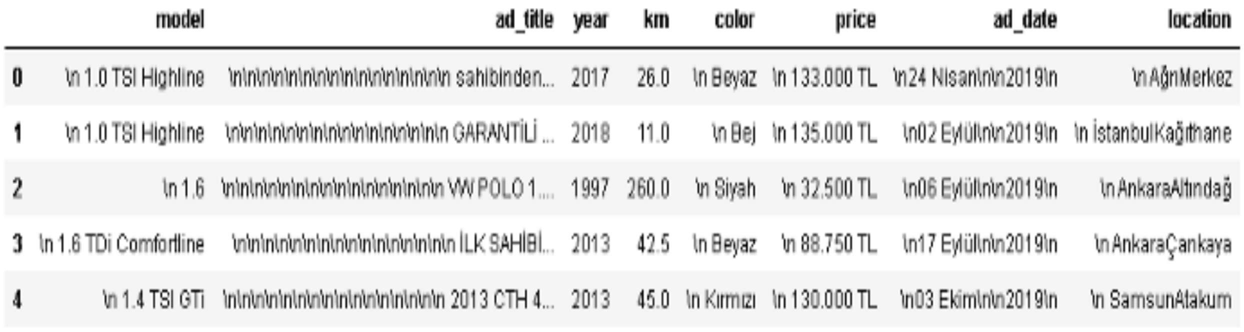
```
df = pd.DataFrame({"model":model_info_text,
"ad_title":ad_title_text,"year":year_text, "km":km_text,
"color":color_text,"price":price_text, "ad_date":ad_date_text,
"location":location_text})
print(df.shape)
print(df['ad_title'].nunique())
(6731, 8)
6293
```

Dataframe includes 6731 entries but 6293 of them seem to be unique according to the title of the ad which I think is the best option to distinguish ads. Some users might re-post the same ad or titles of some ads might be exactly the same.

2. Data Cleaning

I saved the data scraped from the website as a csv file.

```
df = pd.read_csv('polo_data.csv')
df.head()
```



	model	ad_title	year	km	color	price	ad_date	location
0	\n 1.0 TSI Highline	\n\n\n\n\n\n\n\n\n\n\n\n\n\n sahibinden...	2017	26.0	\n Beyaz	\n 133.000 TL	\n24 Nisan\n\n2019\n	\n AğrıMerkez
1	\n 1.0 TSI Highline	\n\n\n\n\n\n\n\n\n\n\n\n\n\n GARANTİLİ ...	2018	11.0	\n Bej	\n 135.000 TL	\n02 Eylül\n\n2019\n	\n İstanbulKağıthane
2	\n 1.6	\n\n\n\n\n\n\n\n\n\n\n\n\n\n VW POLO 1....	1997	260.0	\n Siyah	\n 32.500 TL	\n06 Eylül\n\n2019\n	\n AnkaraAltındağ
3	\n 1.6 TDi Comfortline	\n\n\n\n\n\n\n\n\n\n\n\n\n\n İLK SAHİBİ...	2013	42.5	\n Beyaz	\n 88.750 TL	\n17 Eylül\n\n2019\n	\n AnkaraÇankaya
4	\n 1.4 TSI GTi	\n\n\n\n\n\n\n\n\n\n\n\n\n\n 2013 CTH 4...	2013	45.0	\n Kırmızı	\n 130.000 TL	\n03 Ekim\n\n2019\n	\n SamsunAtakum

New line indicators (\n) had to be removed. I used pandas **remove()** function with **regex** parameter set True. Similarly TL representing Turkish currency in price cell had to be removed to make numerical analysis.

```
df = df.replace('\n','',regex=True)
df.price = df.price.replace('TL','',regex=True)
```

We always need to look for missing values and check data types before trying to do any analysis:

```
df.isna().any()
model       False
ad_title    False
year        False
km          False
color       False
price       False
ad_date     False
location    False
dtype: bool
df.dtypes
model        object
ad_title     object
year          int64
km          float64
color        object
price        object
ad_date      object
location     object
dtype: object
```

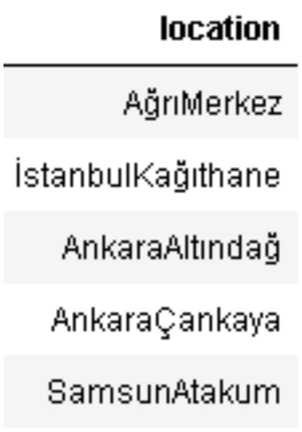
The data type of date was object. To be able to use the dates properly, I converted data dype to **datetime**. The data is in Turkish so I changed the name of months to English before using **astpye()** function. I used a dictionary to change the names of the months.

```
months = {"Ocak":"January", "Şubat":"February", "Mart":"March",
"Nisan":"April","Mayıs":"May","Haziran":"June","Temmuz":"July","Ağustos":"
August","Eylül":"September", "Ekim":"October", "Kasım":"November",
"Aralık":"December"}
df.ad_date = df.ad_date.replace(months, regex=True)
#change the datatype
df.ad_date = pd.to_datetime(df.ad_date)
```

The “km” colums which shows how many kilometres the car has made so for was truncated while reading the csv file. It is because of ‘dot’ used in thousands. For example, 25.000 which is twenty five thousands detected as 25.0. To fix this issue, I multiplied ‘km’ column with 1000. To be able to change the datatype of “km” column to numeric (int or float), I also removed “.” and “,” characters.

```
df.km = df.km * 1000
df.iloc[:,5] = df.iloc[:,5].str.replace(r'.','')
df.iloc[:,5] = df.iloc[:,5].str.replace(r',','')
#change the datatype
df.price = df.price.astype('float64')
```

In Turkey, location might be a factor in determining the price of a used car due to uneven population distribution. Location data in our dataframe includes city and district. I don’t think price changes in different districts of the same city. Therefore, I modified location data to include only the name of the city.



location
AğrıMerkez
İstanbulKağıthane
AnkaraAltındağ
AnkaraÇankaya
SamsunAtakum

Location information is formatted as CityDistrict (no space in between). The name of the district starts with a capital letter which can be used to separate city and district. I used the **sub()** function of **re** module of python.

```
import re
s = df['location']
city_district = []
for i in range(0,6731):
    city_district.append(re.sub( r"([A-Z, 'Ç', 'İ', 'Ö', 'Ş', 'Ü'])", r"
\1", s[i]).split())
city_district[:5]
[['Ağrı', 'Merkez'],
 ['İstanbul', 'Kağıthane'],
 ['Ankara', 'Altındağ'],
 ['Ankara', 'Çankaya'],
 ['Samsun', 'Atakum']]
```

This for loop splits the strings in each cell of location column at capital letters. Turkish alphabet has letters that are not in the [A-Z] range of English alphabet. I added these letters in sub function as well. The output is a list of two-item lists. I created another column named “city” using the first items of this list.

```
city = []
```

```
for i in range(0,6731):
    city.append(city_district[i][0])
city[:5]
['Ağrı', 'İstanbul', 'Ankara', 'Ankara', 'Samsun']
df['city'] = city
```

nunique() function counts the unique values which can be useful for both exploratory data analysis and confirming the results.

```
df.city.nunique()
81
```

There are 81 cities in Turkey so the dataset includes at least one car in each city.

3. Exploratory Data Analysis

Price

It's always good to get some insight about the target variable. The target or dependent variable is price in our case.

```
print(df.price.mean())
print(df.price.median())
83153.7379289853
64250.0
```

Mean is much higher than median which indicates there are outliers or extreme values. Let's also check maximum and minimum values:

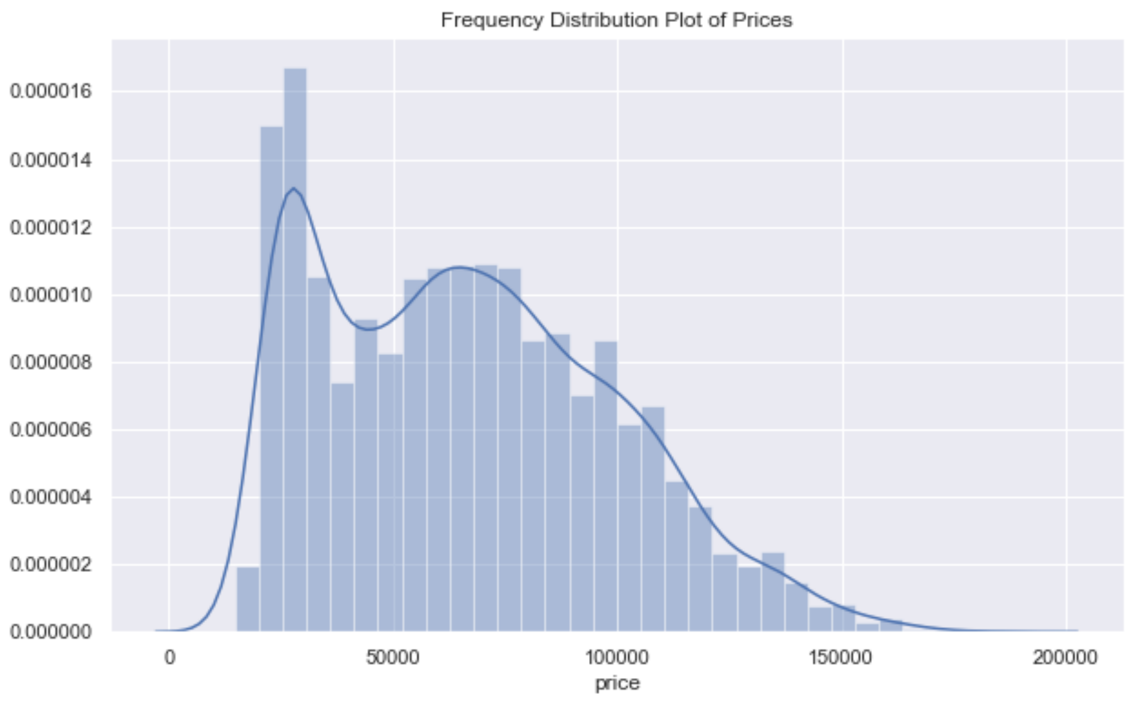
```
print(df.price.max())
print(df.price.min())
111111111.0
24.0
```

This values are obviously wrong. There is no Volkswagen polo for over 100 million unless it is gold coated. Similarly, the value of 24 Turkish Liras is not possible. After sorting values in price column by using **sort_values()** function, I detected a few more outliers and dropped them using pandas **drop()** function by passing indexes of the values to be dropped. Let's check new mean and median values:

```
print(df.price.mean())
print(df.price.median())
print(df.price.median())
66694.66636931311
64275.0
25000.0
```

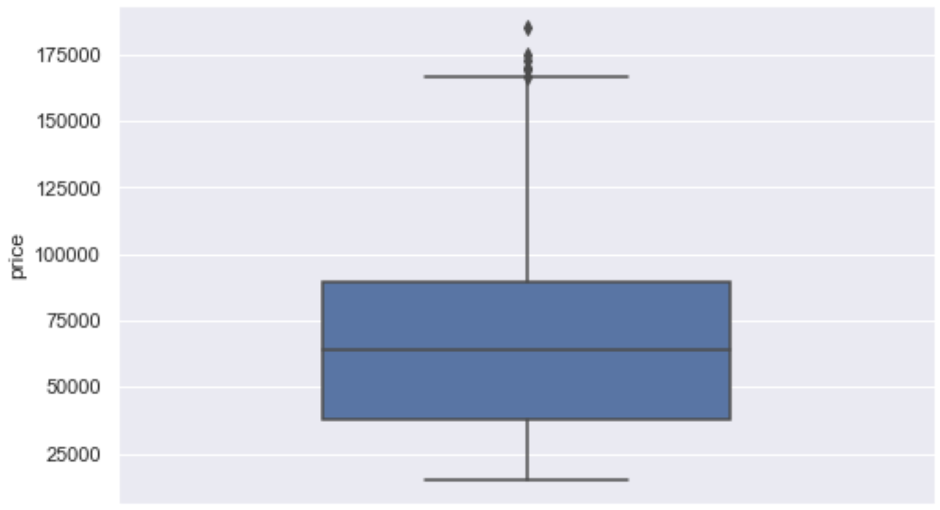
Mean is still higher than median but the difference is not extreme. I also checked mode which is the value that occurs most often. Mean being higher than median indicates that the data is right or positive skewed which means we have more of lower prices and some outliers with higher values. Measures of central tendency being sorted as mean > median > mode is an indication of positive (right) skewness. We can double check with distribution plot:

```
x = df.price
plt.figure(figsize=(10,6))
sns.distplot(x).set_title('Frequency Distribution Plot of Prices')
```



It can be seen from the graph that the data is right skewed and the peak around 25000 shows us the mode. Another way of checking the distribution and outliers is **boxplot**:

```
plt.figure(figsize=(8,5))
sns.boxplot(y='price', data=df, width=0.5)
```

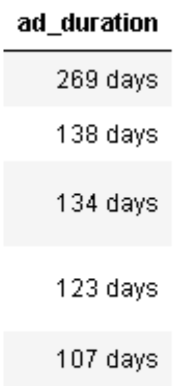


The bottom and top of the blue box represent first quartile (25%) and third quartile (75%), respectively. First quartile means 25% of data points are below this point. The line in the middle is the median (50%). The outliers are shown with dots above the maximum line.

Date

I don’t think date by itself has an effect on the price but waiting period of the ad on website is a factor to be considered. Longer waiting time might motivate owner to reduce the price. If an ad stays on the website for a long time, it might be because the price is not set properly. So I will add a column indicating the number of days ad has been on the website. Data was scraped on 18.01.2020.

```
df['ad_duration'] = pd.to_datetime('2020-01-18') - df['ad_date']
```

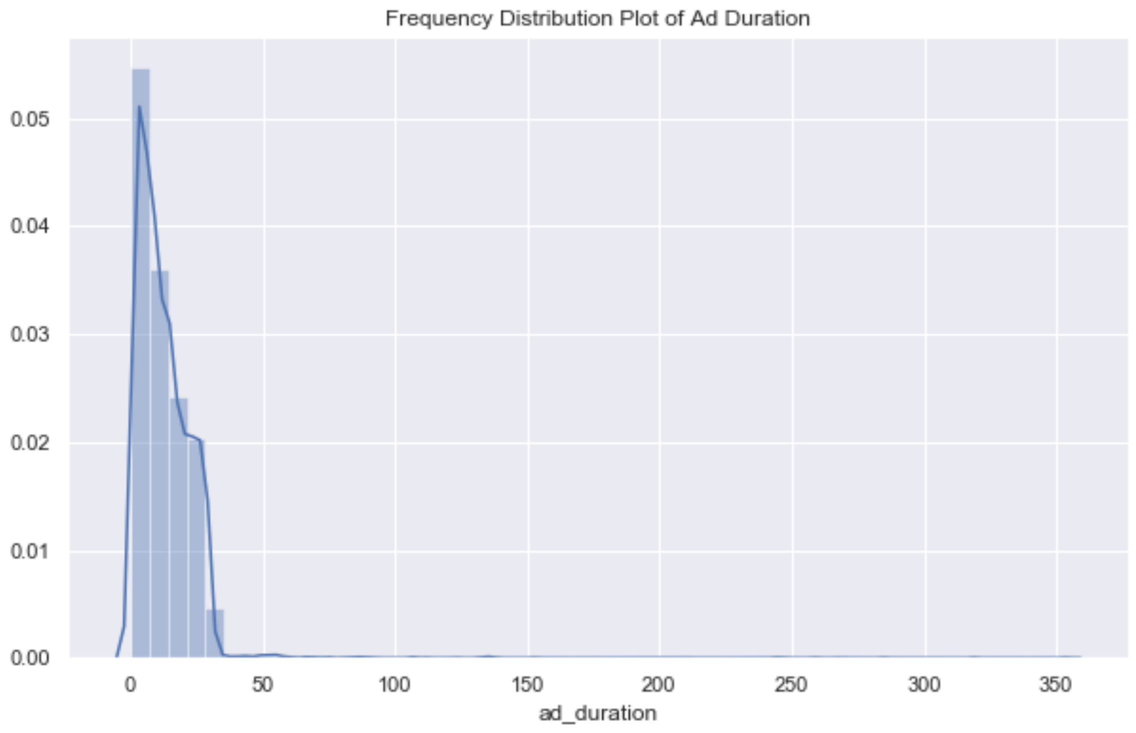


ad_duration
269 days
138 days
134 days
123 days
107 days

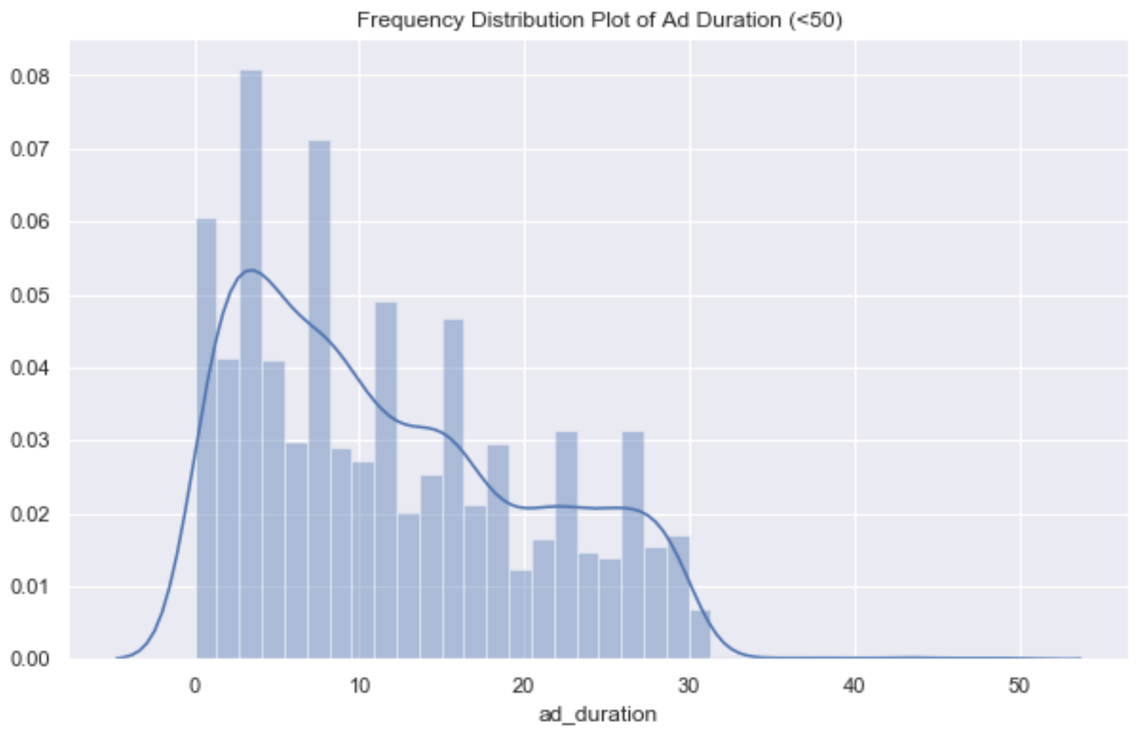
Ad_duration must be a numerical data so ‘days’ next to numbers need to be removed. I used pandas **replace()** function to remove ‘days’.

Let’s check the distribution of ad duration data:

```
print(df.ad_duration.mean())
print(df.ad_duration.median())
12.641540291406482
10.0
```



Mean is higher than the median and there are many outliers. Data is right skewed. To get a better understanding, I also plotted data points less than 50:

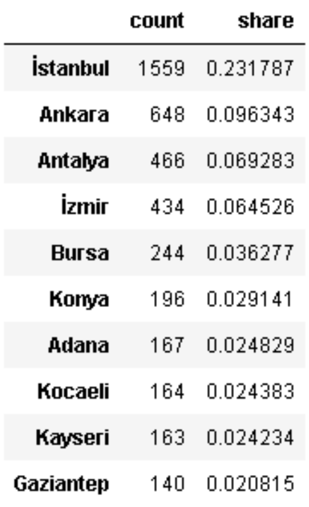


Location

There are 81 different cities but 62% of all ads are listed in top 10 cities with Istanbul having 23% of all ads.

```
a = df.city.value_counts()[:10]
```

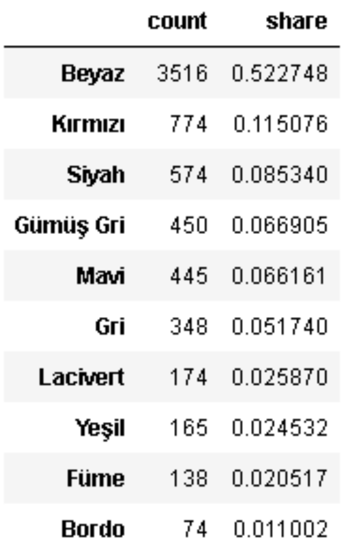
```
df_location = pd.DataFrame({"city": a , "share": a/6726})
df_location.share.sum()
0.6216176033303599
```



	count	share
İstanbul	1559	0.231787
Ankara	648	0.096343
Antalya	466	0.069283
İzmir	434	0.064526
Bursa	244	0.036277
Konya	196	0.029141
Adana	167	0.024829
Kocaeli	164	0.024383
Kayseri	163	0.024234
Gaziantep	140	0.020815

Color

It seems like the optimal choice of color is white for Volkswagen polo. More than half of the cars are white followed by red and black. Top 3 colors cover 72% of all cars.

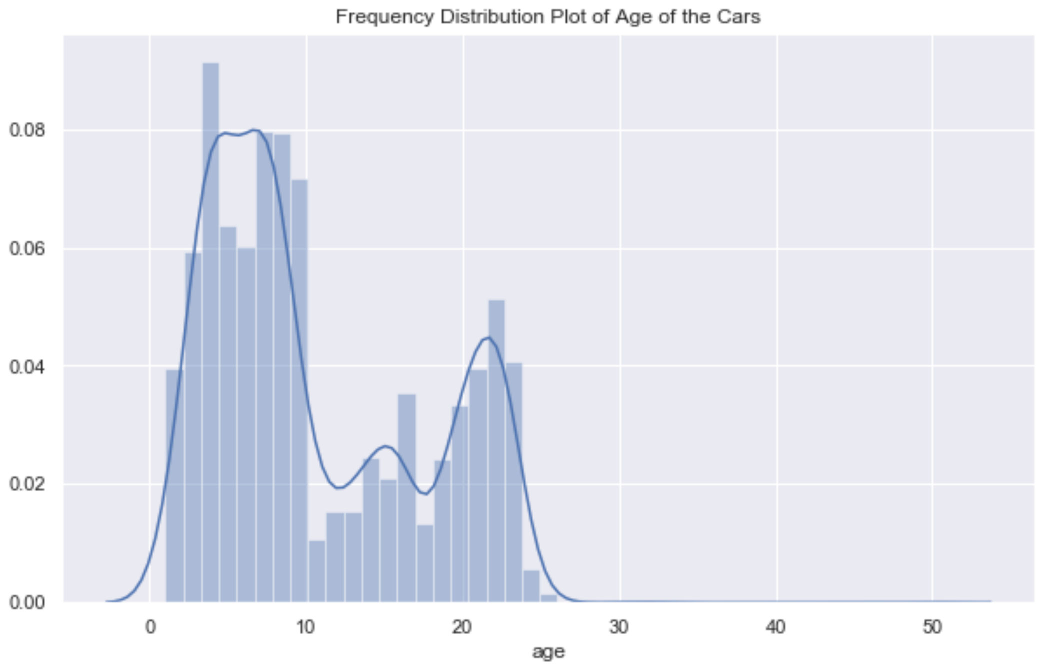


	count	share
Beyaz	3516	0.522748
Kırmızı	774	0.115076
Siyah	574	0.085340
Gümüş Gri	450	0.066905
Mavi	445	0.066161
Gri	348	0.051740
Lacivert	174	0.025870
Yeşil	165	0.024532
Füme	138	0.020517
Bordo	74	0.011002

Year

The age of the car definitely effects the prices. However, instead of the model year of the car, it makes more sense to use is as age. So I substituted ‘year’ column from current year.

```
df['age'] = 2020 - df['year']
```

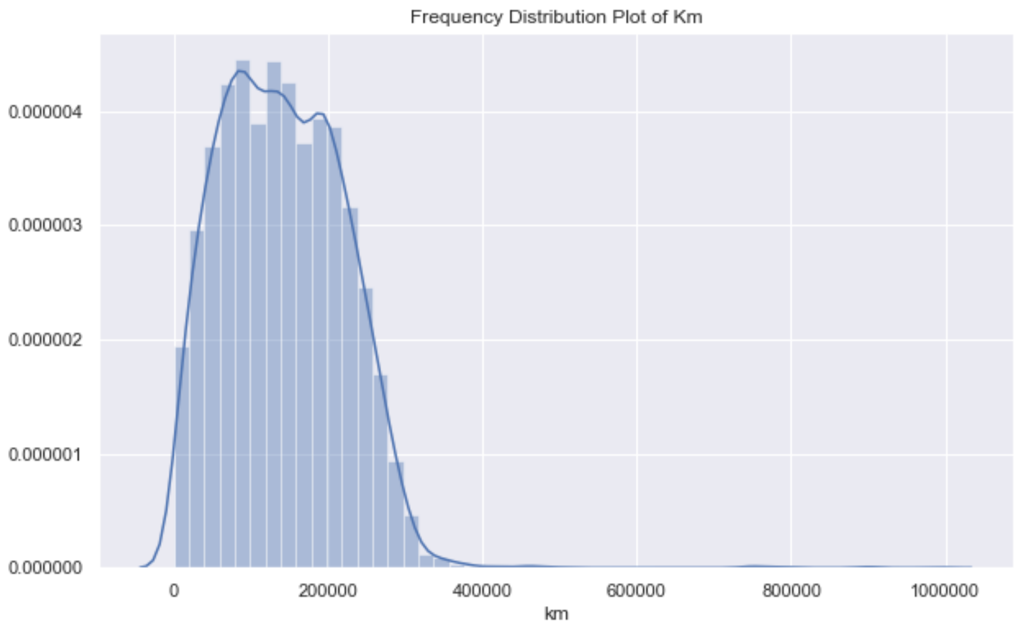


According to the distribution, most of the cars are less than 10 years old. There is a huge drop at 10 followed by an increasing trend.

Km

Km value shows how much the car has ben driven so it is definitely an important factor determining the price. Km data has approximately a normal distribution.

```
print(df.km.mean())
print(df.km.median())
141011.5676479334
137000.0
```



Ad title

Ad title is kind of a caption of the ad. Sellers try to attract possible buyers with a limited number of characters. Once an ad is clicked on, another page with pictures and more detailed information opens up. However, the first step is to get people to click on your ad so ad title plays a critical role in selling process.

Let's check what people usually write in the title. I used **wordcloud** for this task.

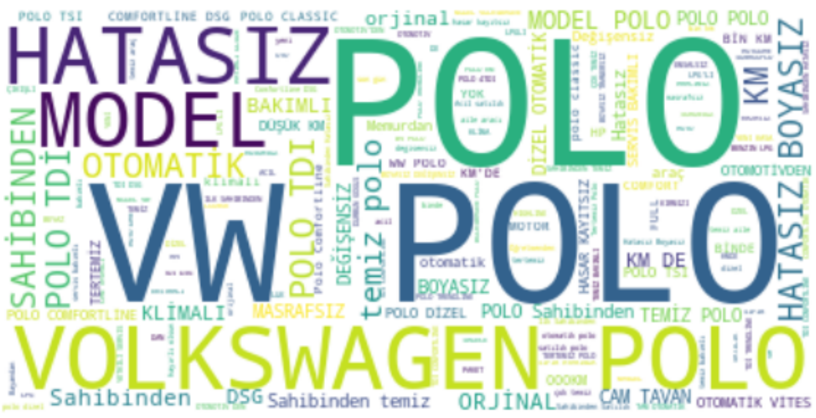
```
#import dependencies
from wordcloud import WordCloud, STOPWORDS
```

The only required parameter for WordCloud is a text. You can check the docstring by typing "?WordCloud" for other optional parameters. We cannot input a list to wordcloud so I created a text by concatenating all the titles in ad_title column:

```
text_list = list(df.ad_title)
text = '-'.join(text_list)
```

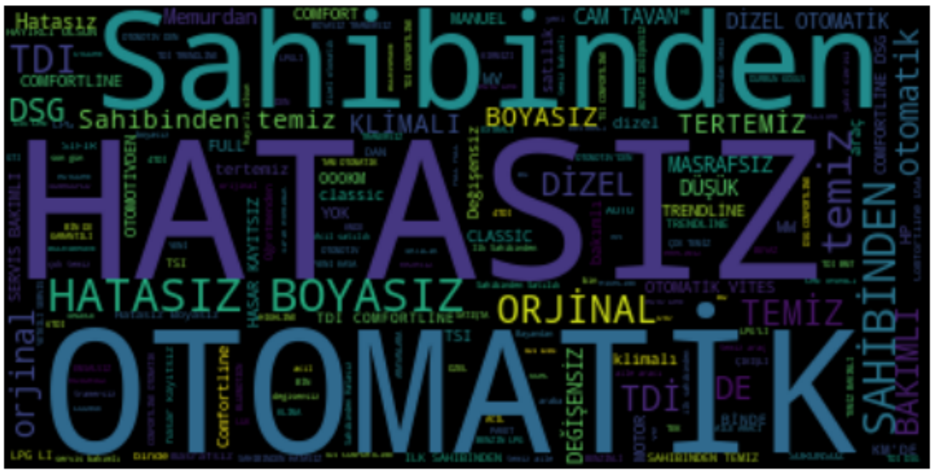
Then used this text to generate a wordcloud:

```
#generate wordcloud
wordcloud = WordCloud(background_color='white').generate(text)
#plot wordcloud
plt.figure(figsize=(10,6))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()
```



The idea of a wordcloud is pretty simple. The more frequent words are shown bigger. It is an informative and easy-to-understand tool for text analysis. However, the wordcloud above does not tell us much because the words "vw", "Volkswagen" and "polo" are not what we are looking for. They show the brand we are analyzing. In this case, we should use **stopwords** parameter of wordcloud to list the words that need to be excluded.

```
stopwords = ['VW', 'VOLKSWAGEN', 'POLO', 'MODEL', 'KM']
wordcloud = WordCloud(stopwords=stopwords).generate(text)
plt.figure(figsize=(10,6))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
plt.show()
```



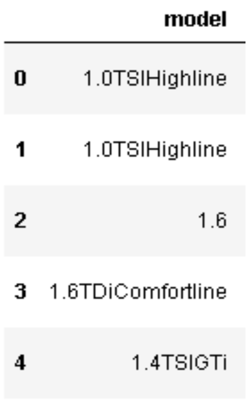
I did not use **background_color** parameter this time just to show the difference. The words are in Turkish so I will give a brief explanation:

> "Hatasız" : Without any problem/issue
> "Sahibinden": From the owner (this is important because people tend to buy from the owner rather than a dealer).
> "Otomatik": Automatic transmission
> "Boyasız": No paint (no part of the car painted due to a crack, scratch or a repair)

The other words are mainly about being clean, not having any previous repairs.

Model

Model column includes three different kinds of information: engine size, fuel type and variant. After checking the values, I found out that only engine size information is complete for all cells. Fuel type and variant are missing for most of the cells so I created a separate column for engine size.



	model
0	1.0TSIHighline
1	1.0TSIHighline
2	1.6
3	1.6TDiComfortline
4	1.4TSIGTi

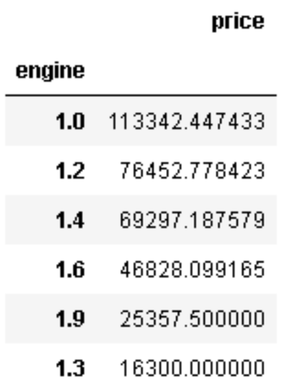
The first three characters after spaces represent engine size. I first removed spaces and extracted the first three characters from model column:

```
#remove spaces
```

```
df.model = df.model.replace(' ','',regex=True)
engine = [x[:3] for x in df.model]
df['engine'] = engine
```

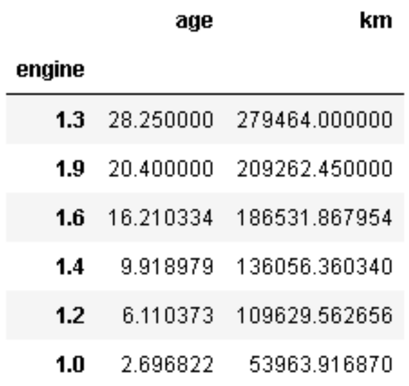
Let’s check how price changes with different engine sizes:

```
df.engine.value_counts()
1.4    3172
1.6    1916
1.2    1205
1.0     409
1.9      20
1.3       4
Name: engine, dtype: int64
df[['engine','price']].groupby(['engine']).mean().sort_values(by='price',
ascending=False)
```



	price
engine	
1.0	113342.447433
1.2	76452.778423
1.4	69297.187579
1.6	46828.099165
1.9	25357.500000
1.3	16300.000000

It seems like average price decreases with increasing engine size. 1.3 can be ignored since there are only 4 cars with 1.3 engine. There is a big gap with 1.0 and the other engine sizes because 1.0 is a newer model. As you can see on the chart below, cars with 1.0 engine size have both lowest age and km on average which shows that they are newer models.



	age	km
engine		
1.3	28.250000	279464.000000
1.9	20.400000	209262.450000
1.6	16.210334	186531.867954
1.4	9.918979	136056.360340
1.2	6.110373	109629.562656
1.0	2.696822	53963.916870

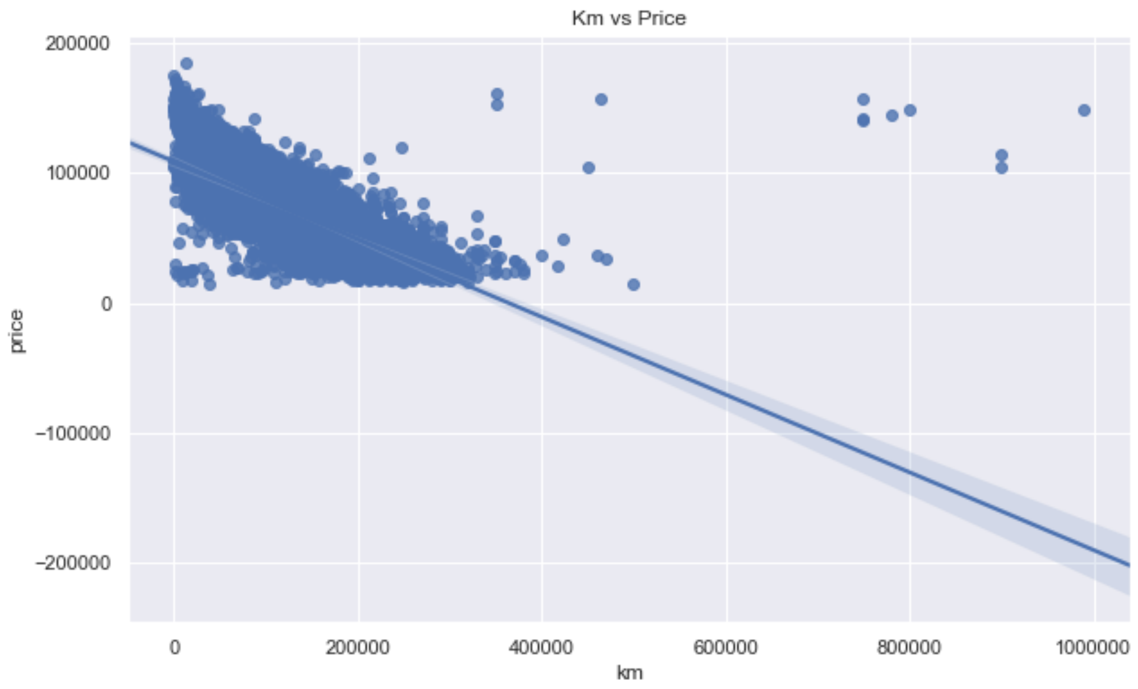
4. Regression Model

Linear regression is a widely-used supervised learning algorithm to predict a continuous dependent (or target) variable. Depending on the number of independent variables, it could be in the form of simple or multiple linear regression. I created a multiple linear regression model because I used many independent variables to predict the dependent variable which is the price of a used car.

We should not just use all of the independent variables without any pre-processing or prior judgment. **Feature selection** is the process to decide which features (independent variables) to use in the model. Feature selection is a very critical step because using unnecessary features has a negative effect on the performance and eliminating important features prevent us from getting a high accuracy.

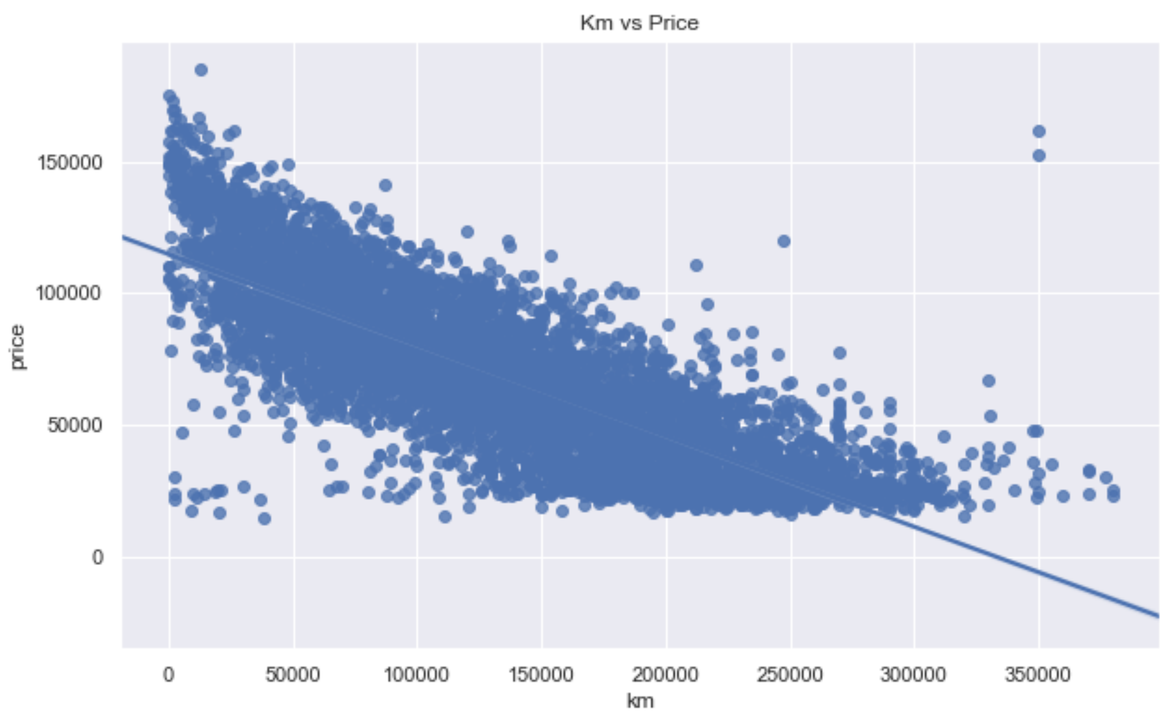
We can use **regression plots** to check the relation between dependent variable and independent variables. I checked the relationship between km and price which I think is highly correlated.

```
plt.figure(figsize=(10,6))
sns.regplot(x='km', y='price', data=df).set_title('Km vs Price')
```

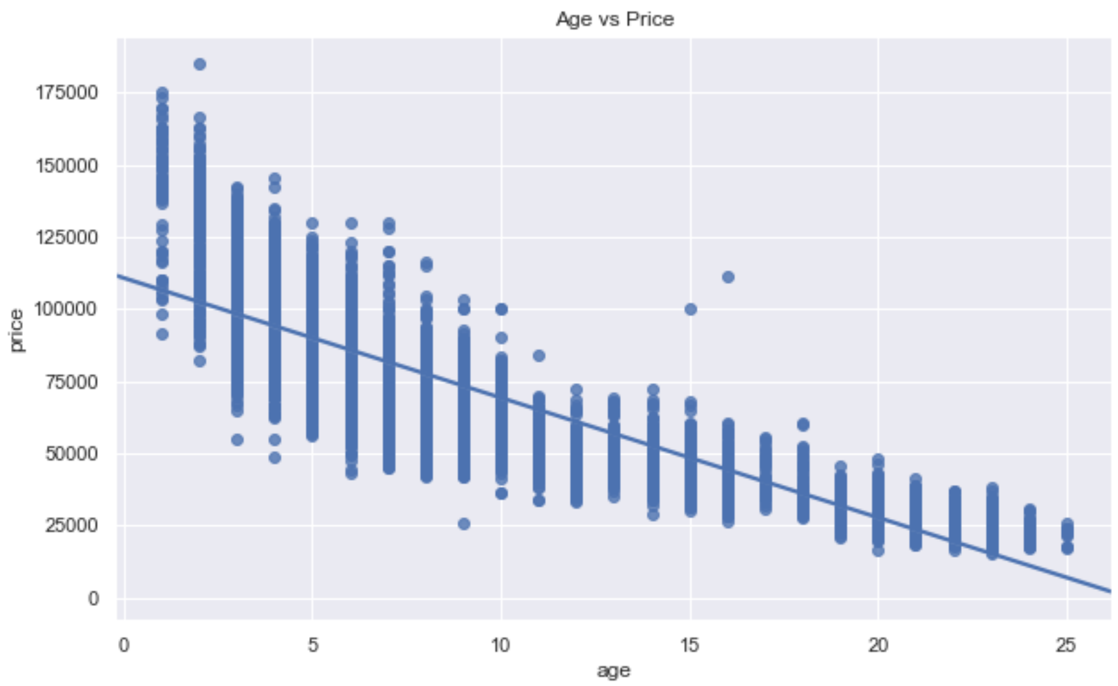


It is clearly seen that as the km goes up, price goes down. However, there are outliers. According to the regression plot above, cars with km higher than 400000 can be marked as outliers. I removed these outliers in order to increase the accuracy of the model. Outliers tend to make the model over fitting.

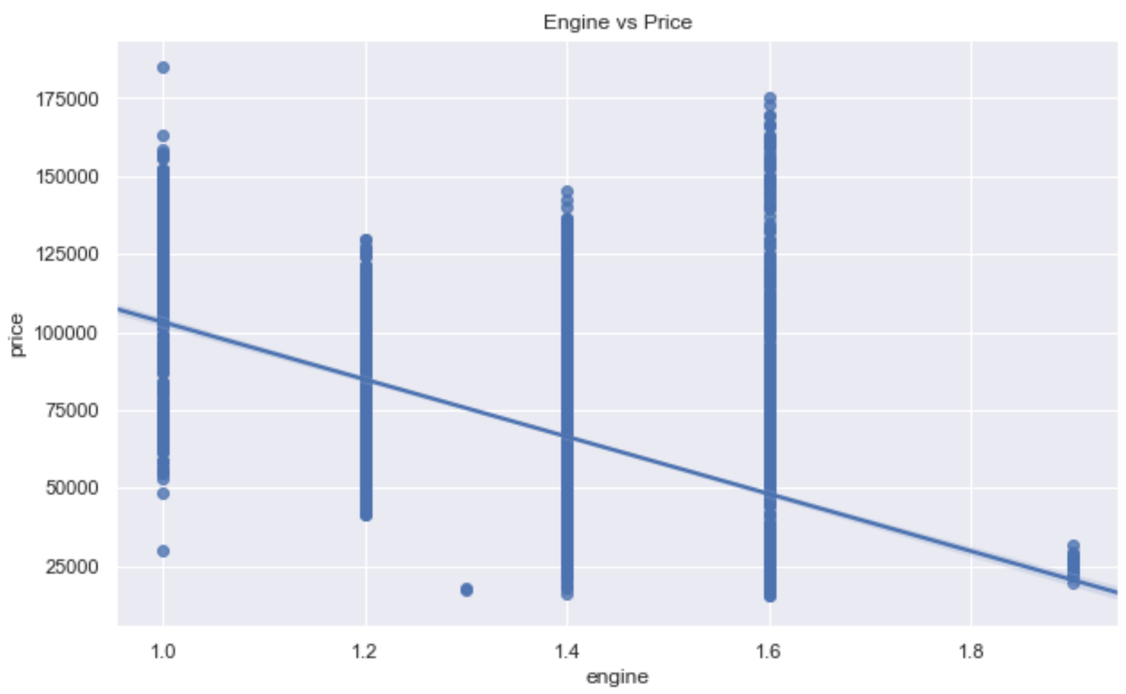
```
df = df[df.km < 400000]
plt.figure(figsize=(10,6))
sns.regplot(x='km', y='price', data=df).set_title('Km vs Price')
```

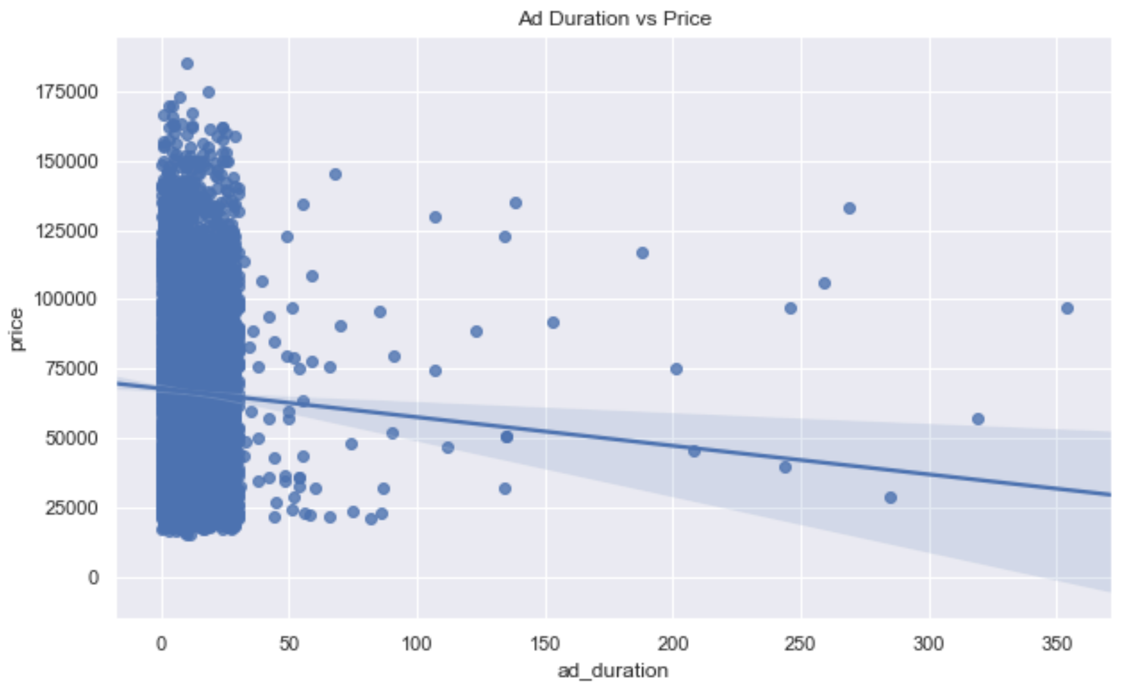


Much better now!
After applying same steps with age and price, a similar relationship was observed:



I also checked the relationship between ad duration and engine size with price. Average price decreases as the engine size gets bigger. However, ad duration seems to have little to no effect on price.





Another way to check relationship between variables is **correlation matrix**. Pandas **corr()** function calculates correlation between numerical variables.

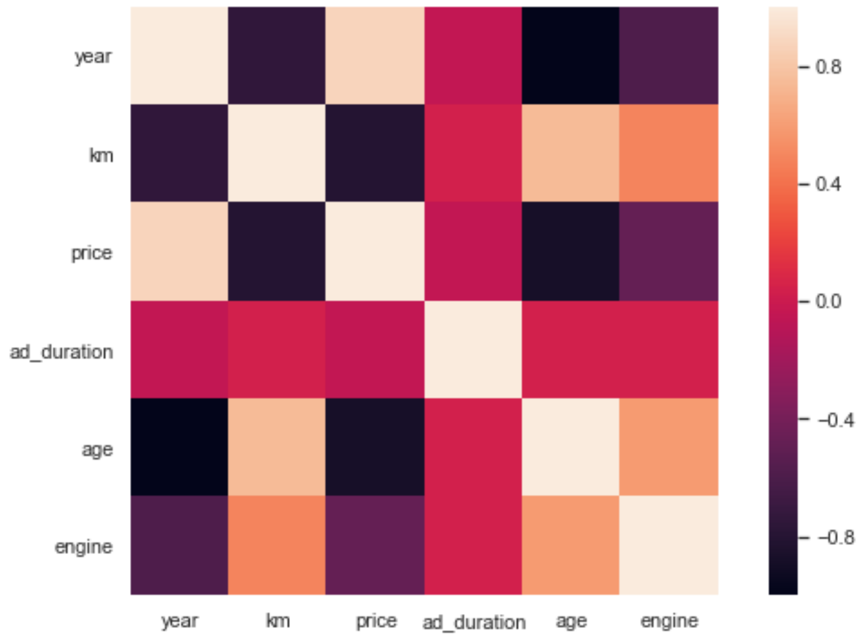
```
print(df.corr())
```

```
                  year        km     price  ad_duration       age    engine
year          1.000000 -0.745587  0.882633    -0.041150 -1.000000 -0.588472
km           -0.745587  1.000000 -0.811924     0.044801  0.745587  0.484709
price         0.882633 -0.811924  1.000000    -0.045212 -0.882633 -0.484242
ad_duration  -0.041150  0.044801 -0.045212     1.000000  0.041150  0.044776
age          -1.000000  0.745587 -0.882633     0.041150  1.000000  0.588472
engine       -0.588472  0.484709 -0.484242     0.044776  0.588472  1.000000
```

The closer the value is to 1, the higher the correlation. '-' sign indicates negative correlation. These values are inline with the regression plots above.

We can also visualize the correlation matrix using seaborn **heatmap**:

```
corr = df.corr()
plt.figure(figsize=(10,6))
sns.heatmap(corr, vmax=1, square=True)
```



The color of box at the intersection of two variables shows the correlation value according to the color chart at the right.

Linear Regression Model

After checking the correlation and distribution of variables, I decided to use age, km, engine size and ad duration to predict the price of a used car.

I used scikit-learn which provides simple and effective machine learning tools.

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
```

I extracted the features (columns) to be used:

```
X = df[['age','km','engine','ad_duration']] #independent variables
y = df['price'] #dependent (target) variable
```

Then using **train_test_split** function of scikit-learn, I divided the data into train and test subsets. To separate train and test set is a very important step for every machine learning algorithm. Otherwise, if we both train and test on the same dataset, we would be asking the model to predict something it already knows.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

Then I created a LinearRegression() object, trained it with train dataset.

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

It's time to measure the accuracy of the model. I measured the accuracy of model on both train and test dataset. If accuracy on train dataset is much higher than the accuracy on test dataset, we have a serious problem: **overfitting**. I will not go in detail about overfitting. It might be a topic of another post but I just want to give a brief explanation. Overfitting means the model is too specific and not generalized well. An overfit model tries to capture noise and extreme values on training dataset.

```
linreg.score(X_train, y_train)
0.8351901442035045
linreg.score(X_test, y_test)
0.8394139260643358
```

The scores are very close which is good. The score here is **R-squared** score which is a measure to determine how close the actual data points are to the fitted regression line. The closer R-squared score is to 1, the more accurate our model is. R-squared measures how much of the variation of target variable is explained by our model.

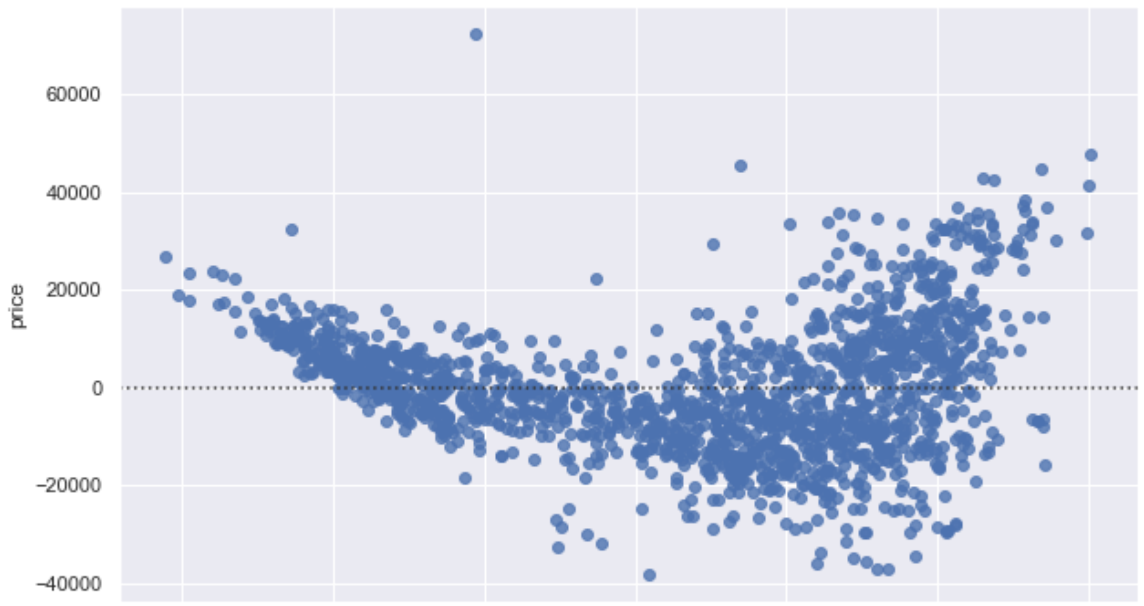
Residual plots are used to check the error between actual values and predicted values. If a linear model is appropriate, we expect to see the errors to be randomly spread and have a zero mean.

```
plt.figure(figsize=(10,6))
sns.residplot(x=y_pred, y=y_test)
```

The mean of the points might be close to zero but obviously they are not randomly spread. The spread is close to a U-shape which indicates a linear model might not be the best option for this task.

In this case, I wanted to try another model using **RandomForestRegressor()** of scikit-learn. Random Forest is an ensemble method built on **decision trees**. This post by Will Koehrsen gives a comprehensive explanation about decision trees and random forests. Random forest are generally used for classification task but work well on regression too.

```
from sklearn.ensemble import RandomForestRegressor
regr = RandomForestRegressor(max_depth=5, random_state=0, n_estimators=10)
```



Residual Plots figure

Unlike linear regression, there are critical hyperparametes to optimize for random forests. **max_depth** is maximum depth of a tree (quite self-explanatory) which controls how deep a tree is or how many splits you want. **n_estimator** is the number of trees in a forest. Decision trees are prone to overfitting which means you can easily make it too specific. If max_depth is too high, you will likely end up with an overfit model. I manually changed max_depth in order to check accuracy and overfitting. However, scikit-learn provides very good tools for hyperparameter tuning: **RandomizedSearchCV and GridSearchCV.** For more complex tasks and model, I highly recommend to use it.

```
print('R-squared score (training): {:.3f}'
     .format(regr.score(X_train, y_train)))
R-squared score (training): 0.902
print('R-squared score (training): {:.3f}'
     .format(regr.score(X_test, y_test)))
R-squared score (training): 0.899
```

R-squared score on test set is 0.899 which indicates a significant improvement compared to linear regression. I also tried with max_depth parameter set to 20 and the result is on overfit model as below. Model is very accurate on training set but accuracy on test set becomes lower.

```
regr = RandomForestRegressor(max_depth=20, random_state=0,
n_estimators=10)
regr.fit(X_train, y_train)
print('R-squared score (training): {:.3f}'
     .format(regr.score(X_train, y_train)))
R-squared score (training): 0.979
print('R-squared score (training): {:.3f}'
     .format(regr.score(X_test, y_test)))
R-squared score (training): 0.884
```

Finally, I checked the price of my car according to the model:

```
regr.predict([[4,75000,1.2,1]])
array([99743.84587199])
```

The model suggested me to sell my car for almost 100 thousand which is higher than the price in my mind. However, my car has been in an accident and repaired which lowers the price. This information was not taken into account as an independent variable which brings us to the last section of this post: Further improvement.

5. Further improvement

There are many ways to improve a machine learning model. I think the most fundamental and effective one is to gather more data. In our case, we can (1) collect data for more cars or (2) more information of the cars in the current dataset or both. For the first one, there are other websites to sell used cars so we can increase the size of our dataset by adding new cars. For the second one, we can scrape more data about the cars from “sahibinden” website. If we click on an ad, another page with detailed information and pictures opens up. In this page, people write about the problems of the car, any previous accident or repairs and so on. This kind information is definitely valuable.

Another way to improve is to adjust model hyper-parameters. We can use RandomizedSearchCV to find optimum hyperparameter values.

6. Conclusion

I tried to give you an overview of how a machine learning project builds up. Although this is not a very complicated task, most of the data science projects follow a similar pattern.

- Define the problem or question
- Collect and clean the data
- Do exploratory data analysis to get some insight about data
- Build a model
- Evaluate the model
  - Go back to any of the previous steps unless the result is sufficient.