House Price Prediction with Linear Regression and Random Forest

The aim of this project is to predict real-estate prices using the machine learning algorithm, Linear Regression, Random Forest. Both will show different results for the accuracy. Also, I will use regression with regularization - Ridge and Lasso to try to improve the prediction accuracy.

Imports

```
from bs4 import BeautifulSoup as bs4
from requests import get
import json
import pandas as pd
import requests
import matplotlib.pyplot as plt
import seaborn as sns
import mpl_toolkits
import numpy as np
%matplotlib inline
#from fake_useragent import UserAgent
```

Data preparation (Web scraping)

Scraping data from the first website - 'FriendlyHousing'

```
url_1 = 'https://www.friendlyhousing.nl/nl/aanbod/kamer'
url_2 = 'https://www.friendlyhousing.nl/nl/aanbod/studio'
url_3 = 'https://www.friendlyhousing.nl/nl/aanbod/appartement'
urls= [url_1, url_2, url_3]
```

Scraping data from the second website - 'Pararius'

```
url_1p = 'https://www.pararius.com/apartments/eindhoven'
url_2p = 'https://www.pararius.com/apartments/eindhoven/page-2'
url_3p = 'https://www.pararius.com/apartments/eindhoven/page-3'
urls_p= [url_1p, url_2p, url_3p]
```

'FriendlyHousing'

```
#user_agent = UserAgent()
#headers={"user-agent": user_agent.chrome}
soup_array=[]
for url in urls:
    ## getting the reponse from the page using get method of requests module
    page = get(url)

    ## storing the content of the page in a variable
    html = page.content

    ## creating BeautifulSoup object
    soup = bs4(html, "html.parser")
    soup_array.append(soup)
```

'Pararius'

```
soup_array_p=[]
for url in urls_p:
    ## getting the reponse from the page using get method of requests module
    page = get(url)

    ## storing the content of the page in a variable
    html = page.content

    ## creating BeautifulSoup object
    soup = bs4(html, "html.parser")
    soup_array_p.append(soup)
```

'FriendlyHousing' - finding the elements from the html file

```
houses=[]
for s in soup_array:
    allHouses = s.find("ul", {"class": "list list-unstyled row equal-row"})
    #print(len(allHouses))
    for h in allHouses.find_all("li", {"class": "col-xs-12 col-sm-6 col-md-4 equal-col"}):
     # print(h)

      houses.append(h)
     # print(h.findAll("li", {"class": "search-list__item search-list__item--listing"}))
```

```
catalog=[]
for h in houses:
```

```
  #data['houses'].append({
      type__= h.find('div', class_= 'specs').text
      t = type__.split()
      type_=t[0]
      street_ = h.find('h3').text
      s = street_.split()
      street = s[0]
      address = h.find('p').text
      a = address.split()
      postcode = a[0]
      #city = a[2]
      price = h.find('div', class_= 'price').text
      vars = type_,street, postcode, price
      catalog.append(vars)
      #print(city)
```

'Pararius' - finding the elements from the html file

```
houses_p=[]
for s in soup_array_p:
    allHouses = s.find("ul", {"class": "search-list"})
    #print(len(allHouses))
    for h in allHouses.find_all("li", {"class": "search-list__item search-list__item--listing
     # print(h)

      houses_p.append(h)
     # print(h.findAll("li", {"class": "search-list__item search-list__item--listing"}))
```

```
catalog_p=[]
for h in houses_p:
  #data['houses'].append({
        name = h.find('a',class_='listing-search-item__link listing-search-item__link--title'
        _name = name.split()
        house_type = _name[0]
        street = _name[1]
        _address= h.findAll('div', class_='listing-search-item__location')[0].text
        #String manipulation to remove the unwanted signs from the address
        __address = _address.replace("\nnew\n ", "")
        address = __address.replace("\n ", "")   #actual address after string manipulation -
        new_address = address.split()
        postcode = new_address[0]
        price_ = h.findAll('span', class_='listing-search-item__price')[0].text
        #splitting the string to find the price
        p=price_.split()
        _price = p[0] #actual price before string manipulation
        __price = _price.replace("€", "")  #actual price before full string manipulation
        price = __price.replace(",", "")   #actual price after string manipulation - ready to

        #finding the whole element from the web page
```

```
        #finding the whole element from the web page
        ylr= h.findAll('section', class_= 'illustrated-features illustrated-features--vertica

        #splitting the string to find the living are, rooms and year
        lry= ylr.split()

        #living_area after taking the indexes that define it
        living_area = lry[0]

        #rooms after taking the index that defines the variable
        rooms = lry[4]

        vars = house_type, street, postcode,price,living_area,rooms
        catalog_p.append(vars)

print(catalog_p)
```

```
[('Apartment', 'St', '5645', '1225', '71', '3'), ('Apartment', 'Limburglaan', '5616', '
```

'FriendlyHousing' - creating the dataframe

```
dataframe = pd.DataFrame(catalog)
dataframe.columns=['TYPE', 'STREET NAME', 'POSTCODE', 'PRICE']
dataframe
```

	TYPE	STREET NAME	POSTCODE	PRICE
0	Kamer	Heezerweg	5614	390
1	Kamer	Willem	5611	320
2	Kamer	Willem	5611	310
3	Kamer	Julianastraat	5611	375
4	Kamer	Bennekelstraat	5654	430
...	...	...	...	...
114	Appartement	Frankrijkstraat	5622	925
115	Appartement	Kerkakkerstraat	5616	950
116	Appartement	Leenderweg	5614	800
117	Appartement	Leostraat	5615	775
118	Appartement	Stratumsedijk	5614	1075

119 rows × 4 columns

'Pararius'- creating the dataframe

```
df_ = pd.DataFrame(catalog_p)
df_.columns=['TYPE', 'STREET NAME', 'POSTCODE', 'PRICE','LIVING_AREA', 'ROOMS']
df_
```

	TYPE	STREET NAME	POSTCODE	PRICE	LIVING_AREA	ROOMS
0	Apartment	St	5645	1225	71	3
1	Apartment	Limburglaan	5616	995	52	2
2	Apartment	Limburglaan	5616	1050	51	2
3	Apartment	Welschapsedijk	5652	1025	75	3
4	Apartment	De	5611	1099	73	2
...	...	...	...	...	...	...
85	Apartment	Cornelis	5654	720	25	1
86	Apartment	Philitelaan	5617	1900	135	5
87	House	Frans	5613	1295	85	3
88	House	Vrijkensven	5646	1090	121	5
89	House	Vrijkensven	5646	1200	130	5

90 rows × 6 columns

Data integration

> Using concat to create a `Union` between the two datasets and then, integrate them into one dataset.

```
frames = [dataframe, df_]
```

```
df = pd.concat(frames)
df
```

	TYPE	STREET NAME	POSTCODE	PRICE	LIVING_AREA	ROOMS
0	Kamer	Heezerweg	5614	390	NaN	NaN
1	Kamer	Willem	5611	320	NaN	NaN
2	Kamer	Willem	5611	310	NaN	NaN
3	Kamer	Julianastraat	5611	375	NaN	NaN
4	Kamer	Bennekelstraat	5654	430	NaN	NaN
...	...	...	...	...	...	...
88	House	Vrijkensven	5646	1090	121	5

Saving into csv file.

```
df.to_csv('data.csv')
```

Data analysis

Checking the dimension of the dataset and the features.

```
# Check the dimension of the dataset
df.shape
```

```
(209, 6)
```

> The dataset has 219 observations and 6 features, but the observations(rows) will change with time because the data is scraped and this means it is up to date. Whenever there is a change on the websites, there is a change in the dataset.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 209 entries, 0 to 89
Data columns (total 6 columns):
 #   Column       Non-Null Count  Dtype 
---  ------       --------------  ----- 
 0   TYPE         209 non-null    object
 1   STREET NAME  209 non-null    object
 2   POSTCODE     209 non-null    object
 3   PRICE        209 non-null    object
 4   LIVING_AREA  90 non-null     object
 5   ROOMS        90 non-null     object
dtypes: object(6)
memory usage: 11.4+ KB
```

> It can be seen that none features are numeric, but objects. Later, they will have to be converted into either float or int in order to be plotted and then used for the trainig of the models. There are also missing values in the dataset.

There are missing values in the dataset, which appeared after the data integration of the two datasets. This will be fixed later before the training of the models.

```
df.isnull().sum()
```

```
TYPE             0
STREET NAME      0
POSTCODE         0
PRICE            0
LIVING_AREA    119
ROOMS          119
dtype: int64
```

```
# Find columns with missing values and their percent missing
df.isnull().sum()
miss_val = df.isnull().sum().sort_values(ascending=False)
miss_val = pd.DataFrame(data=df.isnull().sum().sort_values(ascending=False), columns=['Missva

# Add a new column to the dataframe and fill it with the percentage of missing values
miss_val['Percent'] = miss_val.MissvalCount.apply(lambda x : '{:.2f}'.format(float(x)/df.shap
miss_val = miss_val[miss_val.MissvalCount > 0].style.background_gradient(cmap='Reds')
miss_val
```

	MissvalCount	Percent
ROOMS	119	56.94
LIVING_AREA	119	56.94

> The light red color shows the small amount of NaN values. If the features were with a high percent of missing values, they would have to be removed. Yet, in this case, they have relatively low percentage so they can be used in future. Then, the NaN values will be replaced.

```
#Description of the dataset
df.describe()
```

	TYPE	STREET NAME	POSTCODE	PRICE	LIVING_AREA	ROOMS
count	209	209	209	209	90	90

```
#First 5 rows of our dataset
df.head()
```

	TYPE	STREET NAME	POSTCODE	PRICE	LIVING_AREA	ROOMS
0	Kamer	Heezerweg	5614	390	NaN	NaN
1	Kamer	Willem	5611	320	NaN	NaN
2	Kamer	Willem	5611	310	NaN	NaN
3	Kamer	Julianastraat	5611	375	NaN	NaN
4	Kamer	Bennekelstraat	5654	430	NaN	NaN

```
#Last 5 rows of our dataset
df.tail()
```

	TYPE	STREET NAME	POSTCODE	PRICE	LIVING_AREA	ROOMS
85	Apartment	Cornelis	5654	720	25	1
86	Apartment	Philitelaan	5617	1900	135	5
87	House	Frans	5613	1295	85	3
88	House	Vrijkensven	5646	1090	121	5
89	House	Vrijkensven	5646	1200	130	5

```
df['TYPE'].value_counts()
```

```
Apartment      74
Kamer          47
Appartement    36
Studio         36
House          10
Room            6
Name: TYPE, dtype: int64
```

```
df.iloc[0]
```

```
TYPE               Kamer
STREET NAME    Heezerweg
POSTCODE            5614
PRICE                390
LIVING_AREA          NaN
```

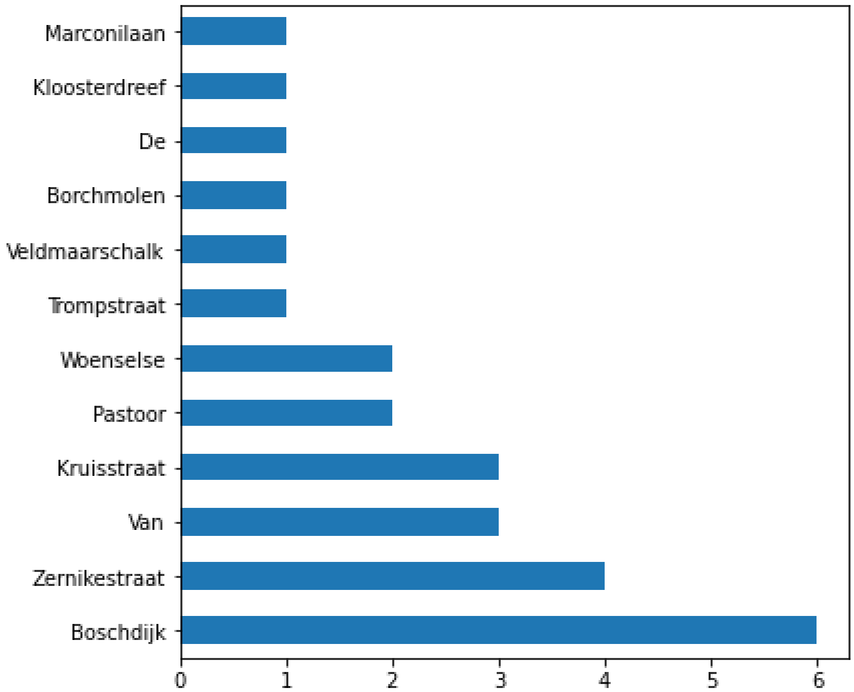
```
ROOMS                NaN
```

```
df.groupby('POSTCODE').count()
```

	TYPE	STREET NAME	PRICE	LIVING_AREA	ROOMS
POSTCODE					
5503	1	1	1	0	0
5611	48	48	48	31	31
5612	26	26	26	10	10
5613	9	9	9	4	4
5614	13	13	13	2	2
5615	14	14	14	6	6
5616	10	10	10	8	8
5617	1	1	1	1	1
5621	8	8	8	1	1
5622	7	7	7	2	2
5623	10	10	10	1	1
5624	1	1	1	0	0
5625	4	4	4	3	3
5629	1	1	1	1	1
5631	3	3	3	0	0
5642	7	7	7	2	2
5643	13	13	13	2	2
5644	6	6	6	3	3
5645	1	1	1	1	1
5646	2	2	2	2	2
5651	4	4	4	0	0
5652	2	2	2	2	2
5653	5	5	5	1	1
5654	12	12	12	6	6
5658	1	1	1	1	1

```
df[(df['POSTCODE'] == '5612')]['STREET NAME'].value_counts().plot(kind='barh', figsize=(6, 6)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8d66e8aa90>
```



Sorting the data by `Type`.

```
df.sort_values('TYPE', ascending = True)
```

	TYPE	STREET NAME	POSTCODE	PRICE	LIVING_AREA	ROOMS
26	Apartment	Jeroen	5613	750	40	2
22	Apartment	Aalsterweg	5615	795	45	2
25	Apartment	Hertogstraat	5611	1200	100	1
27	Apartment	De	5611	1349	91	3
29	Apartment	Margrietstraat	5643	685	20	1
...	...	...	...	...	...	...
60	Studio	Woenselsestraat	5623	520	NaN	NaN
62	Studio	Dr.	5623	640	NaN	NaN
63	Studio	Van	5612	602	NaN	NaN
55	Studio	Koenraadlaan	5651	665	NaN	NaN
82	Studio	Kleine	5611	705	NaN	NaN

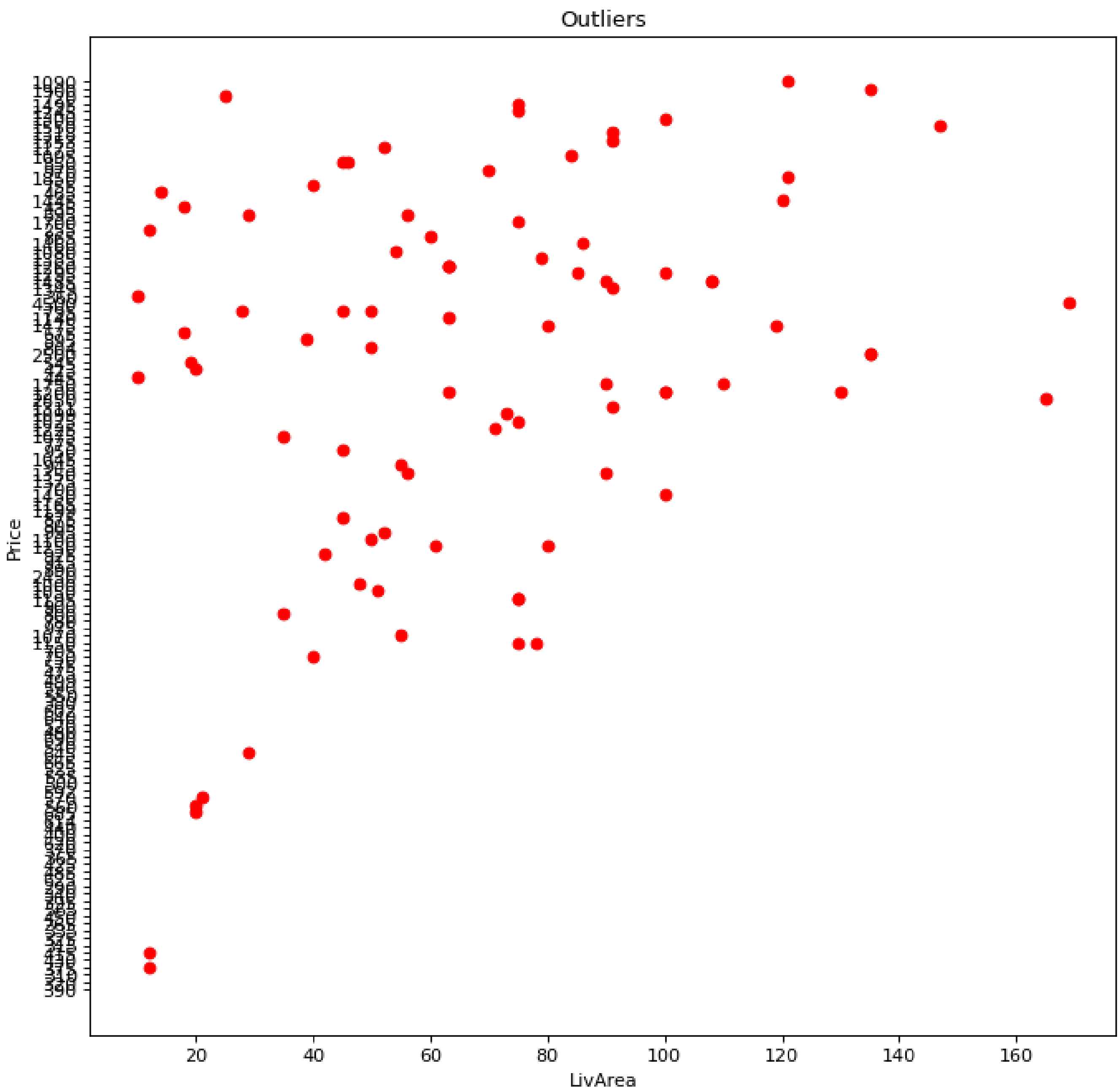
209 rows × 6 columns

Pre Processing

Handling Outlier

> An **outlier** is a data point in a data set that is distant from all other observations (a data point that lies outside the overall distribution of the dataset.)

```
plt.figure(figsize=(10, 10), dpi=80)
plt.scatter(df.LIVING_AREA, df.PRICE, c= 'red')
plt.title("Outliers")
plt.xlabel("LivArea")
plt.ylabel("Price")
plt.show()
```




```
df['PRICE'] =df['PRICE'].astype(float)
```

```
df['PRICE'] =df['PRICE'].astype(float)
df['POSTCODE'] =df['POSTCODE'].astype(int)
df['LIVING_AREA'] =df['LIVING_AREA'].astype(float)
df['ROOMS'] =df['ROOMS'].astype(float)
code_numeric = {'Kamer': 5,'Apartment': 1, 'Appartement': 1, 'Room': 2, 'Studio': 4, 'House':
df ['TYPE'] = df['TYPE'].map(code_numeric)
df['TYPE'] =df['TYPE'].astype(float)


df['PRICE'] =df['PRICE'].astype(float)


df.boxplot(column=['PRICE'])
plt.show
```

```
<function matplotlib.pyplot.show>
```


```
#Check the mean values
df['LIVING_AREA'].mean()
```

```
67.85555555555555
```

```
#Check the median
df['LIVING_AREA'].median()
```

```
63.0
```

```
Q1 = df.quantile(0.25)
Q3 = df.quantile(0.75)
IQR = Q3 - Q1
print(IQR)
```

```
TYPE             3.0
POSTCODE        30.0
PRICE          685.0
LIVING_AREA     48.0
ROOMS            1.0
dtype: float64
```

```
print(df['PRICE'].skew())
df['PRICE'].describe()
```

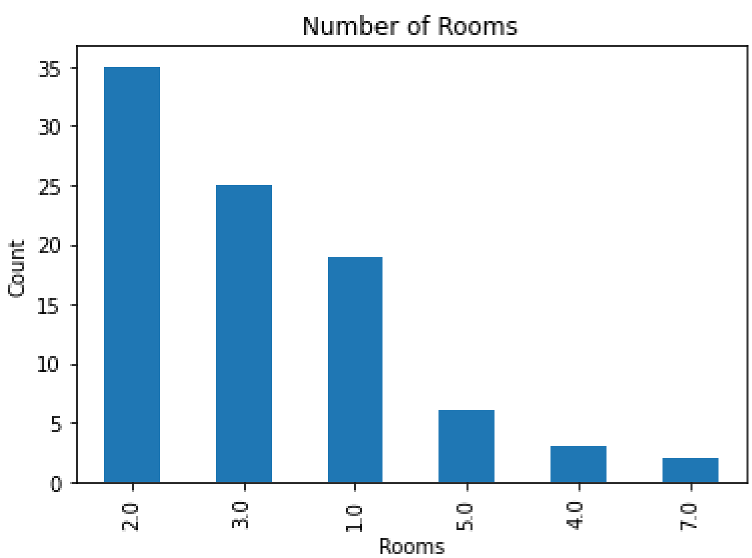
```
2.470034807914771
count     209.000000
mean      868.674641
std       516.119861
min       255.000000
25%       465.000000
50%       755.000000
75%      1150.000000
max      4500.000000
Name: PRICE, dtype: float64
```

```
print(df['PRICE'].quantile(0.10))
print(df['PRICE'].quantile(0.90))
```

```
399.0
1397.000000000001
```

```
df['ROOMS'].value_counts().plot(kind='bar')
plt.title('Number of Rooms')
plt.xlabel('Rooms')
plt.ylabel('Count')
sns.despine
```

```
<function seaborn.utils.despine>
```




```
print(df['PRICE'])
```

```
0      390.0
1      320.0
2      310.0
3      375.0
4      430.0
```

```
        ...
85     720.0
86    1900.0
87    1295.0
88    1090.0
89    1200.0
Name: PRICE, Length: 209, dtype: float64
```

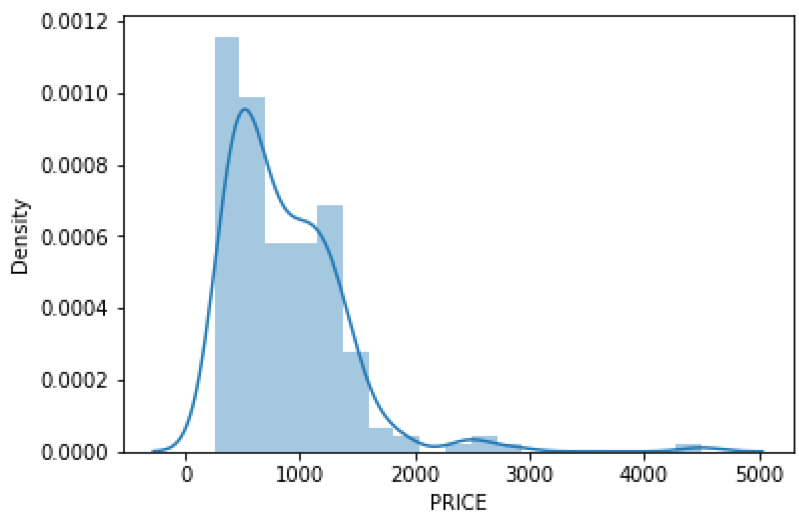
We will analyze the features in their descending of correlation with sales price

Examining the data distributions of the features. We will start with the target variable, `PRICE`, to make sure it's normally distributed.

This is important because most machine learning algorithms make the assumption that the data is normally distributed. When data fits a normal distribution, statements about the price using analytical techniques will be made.

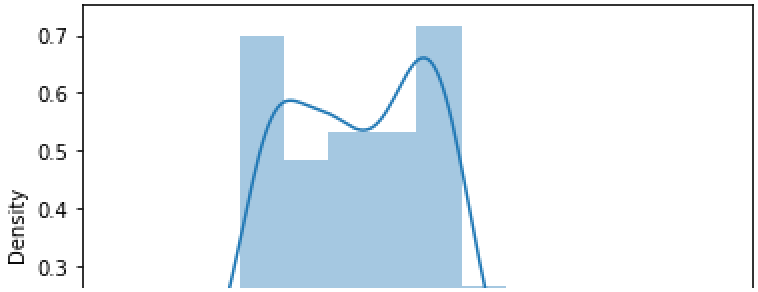
```
sns.distplot(df['PRICE'])
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2557: FutureWarning: `di
  warnings.warn(msg, FutureWarning)
<matplotlib.axes._subplots.AxesSubplot at 0x7f8d6637bd50>
```




```
# Transform the target variable
sns.distplot(np.log(df.PRICE))
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2557: FutureWarning: `d:
  warnings.warn(msg, FutureWarning)
<matplotlib.axes._subplots.AxesSubplot at 0x7f8d663a8b50>
```



> We can see that the `PRICE` distribution is not skewed after the transformation, but normally distributed. The transformed data will be used in in the dataframe and remove the skewed distribution:

> **Normally distributed** means that the data is symmetric about the mean, showing that data near the mean are more frequent in occurrence than data far from the mean.

```
df['LogOfPrice'] = np.log(df.PRICE)
df.drop(["PRICE"], axis=1, inplace=True)
```

Reviewing the skewness of each feature

```
df.skew().sort_values(ascending=False)
```

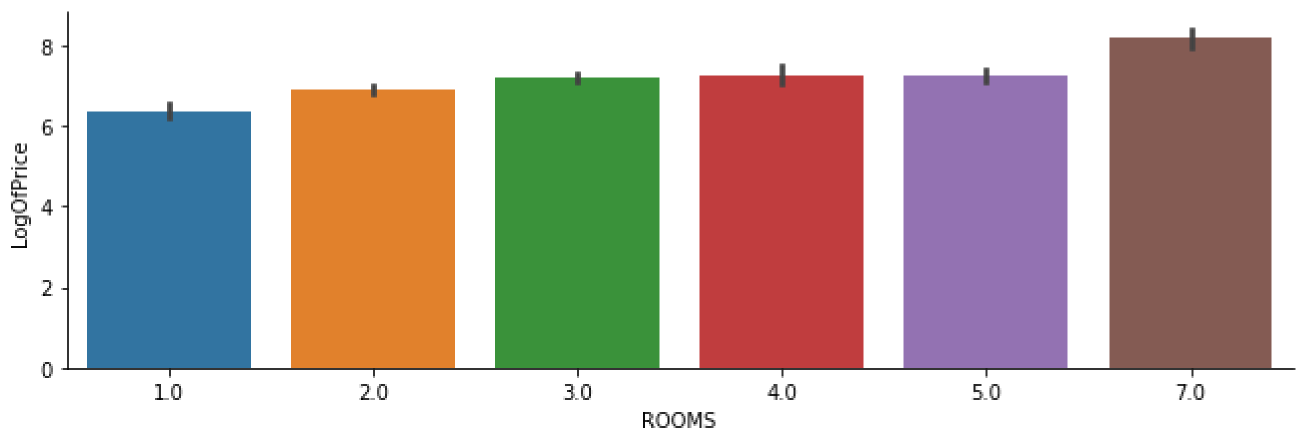
```
ROOMS          1.390682
LIVING_AREA    0.508258
TYPE           0.380100
LogOfPrice     0.207423
POSTCODE      -0.862667
dtype: float64
```

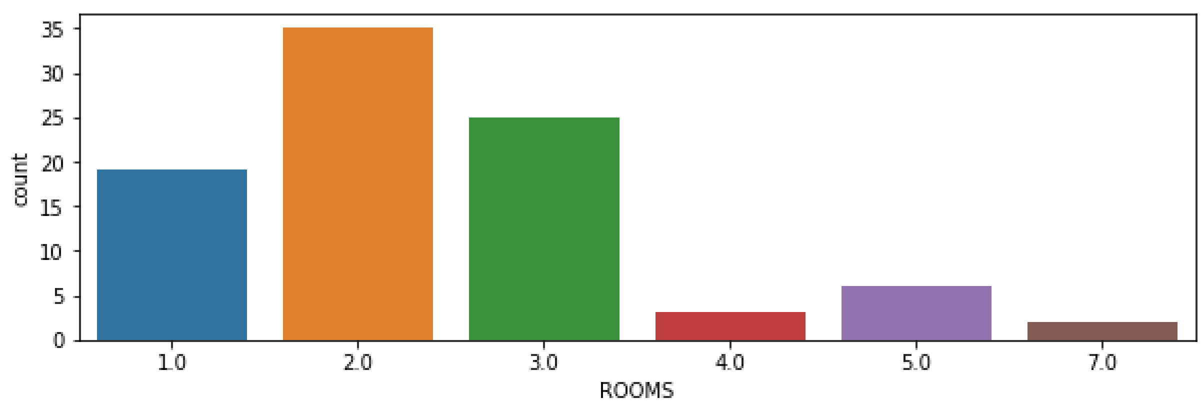
> Values closer to zero are less skewed. The results show some features having a positive (right-tailed) or negative (left-tailed) skew.

Factor plot is informative when we have multiple groups to compare.

```
sns.factorplot('ROOMS', 'LogOfPrice', data=df,kind='bar',size=3,aspect=3)
fig, (axis1) = plt.subplots(1,1,figsize=(10,3))
sns.countplot('ROOMS', data=df)
df['LogOfPrice'].value_counts()
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/categorical.py:3714: UserWarning: The `fa
  warnings.warn(msg)
/usr/local/lib/python3.7/dist-packages/seaborn/categorical.py:3720: UserWarning: The `si
  warnings.warn(msg, UserWarning)
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass th
  FutureWarning
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass th
  FutureWarning
6.028279    14
6.109248     5
7.090077     4
6.345636     4
6.380123     4
            ..
5.783825     1
7.110696     1
8.411833     1
6.791221     1
7.210080     1
Name: LogOfPrice, Length: 124, dtype: int64
```



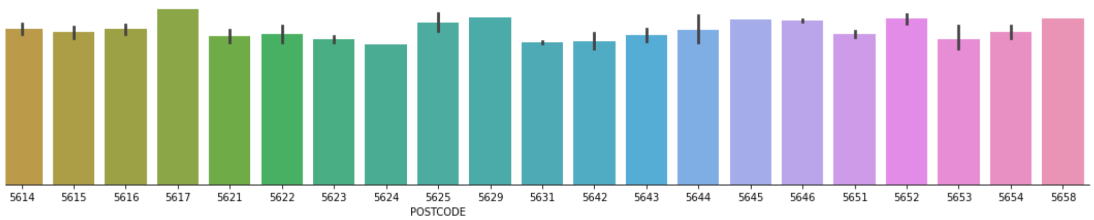


> Real estate with 5 rooms has the highest `Price` while the sales of others with rooms of 2 is the most sold ones.

```
#g = sns.factorplot(x='POSTCODE', y='Skewed_SP', col='PRICE', data=df, kind='bar', col_wrap=4
sns.factorplot('POSTCODE', 'LogOfPrice', data=df,kind='bar',size=3,aspect=6)
```

```
/thon3.7/dist-packages/seaborn/categorical.py:3714: UserWarning: The `factorplot` functic
nsg)
/thon3.7/dist-packages/seaborn/categorical.py:3720: UserWarning: The `size` parameter has
nsg, UserWarning)
/thon3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass the following varia

d.FacetGrid at 0x7f8d661c3910>
```



> The diagram represents the `price` of a rpoperty, depending on its `postcode`.

Filling up the null values in order to train the model.

```
df.fillna(0)
```

	TYPE	STREET NAME	POSTCODE	LIVING_AREA	ROOMS	LogOfPrice
0	5.0	Heezerweg	5614	0.0	0.0	5.966147
1	5.0	Willem	5611	0.0	0.0	5.768321
2	5.0	Willem	5611	0.0	0.0	5.736572
3	5.0	Julianastraat	5611	0.0	0.0	5.926926
4	5.0	Bennekelstraat	5654	0.0	0.0	6.063785
...	...	...	...	...	...	...
85	1.0	Cornelis	5654	25.0	1.0	6.579251
86	1.0	Philitelaan	5617	135.0	5.0	7.549609
87	3.0	Frans	5613	85.0	3.0	7.166266
88	3.0	Vrijkensven	5646	121.0	5.0	6.993933
89	3.0	Vrijkensven	5646	130.0	5.0	7.090077

209 rows × 6 columns

```
df.dropna(inplace=True)
```

```
df.isnull()
```

	TYPE	STREET NAME	POSTCODE	LIVING_AREA	ROOMS	LogOfPrice
0	False	False	False	False	False	False
1	False	False	False	False	False	False
2	False	False	False	False	False	False
3	False	False	False	False	False	False
4	False	False	False	False	False	False
...	...	...	...	...	...	...
85	False	False	False	False	False	False
86	False	False	False	False	False	False
87	False	False	False	False	False	False
88	False	False	False	False	False	False
89	False	False	False	False	False	False

90 rows × 6 columns

Conclusion

Data collection:

For the data collection part, I decided to use `web scraping` as e technique because it gives the opportunity to work with a data set that is up to date and therefore, makes more accurate summaries.

Data preprocessing:

I tried different types of data transforms to expose the data structure better, so we may be able to improve model accuracy later.

- `Standardizing` was made to the data set so as to reduce the effects of differing distributions.
- `The skewness` of the features was checked in order to see how distorted a data sample is from the normal distribution.
- `Rescaling (normalizing)` the dataset was also included to reduce the effects of differing scales