A model for predicting real estate prices

Yana Burlachenko

1. Problem definition and Metric

Our goal is to build a model for predicting real estate prices. ML task is supervised learning (because we give both features and target) and regression (prediction a continuous outcome variable based on the value of one or multiple predictor variables).

In this task will be done:

- Choosing the metric;
- Data cleaning and formatting;
- EDA;
- Feature engineering and selection;
- Split Into Training and Testing Sets;
- Baseline;
- Hyperparameter tuning;
- Prediction and writing the result into the file.

It was decided to use Mean Absolute Error (MAE) as a metric, because it simle in usage and not as sensitive as, for example, MSE. It is well-interpretable because it represents the average amount our estimate if off by in the same units as the target value.

2. Data Cleaning and Formating

2.1. Loading in the Data

First of all, let's import all necessary libraries and put some settings in pandas in order to show all rows and columns in data. Then we'll open the data as a Pandas DataFrame:

In [1]:

```
#importing libraries and functions
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.stats import anderson
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import GridSearchCV


#settigs of output
pd.set_option("max_rows", None)
pd.set_option("max_columns", None)

#opening the data
df = pd.DataFrame(pd.read_csv('train.csv'))
df.head()
```

Out[1]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	LotConf
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	Insic
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	FF
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	Insic
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	Corn
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	FF

Also we want to know more about this data. For example, quantity of columns, type of data in it and quantity of non-null rows.

In [2]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 81 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Id             1460 non-null   int64
 1   MSSubClass     1460 non-null   int64
 2   MSZoning       1460 non-null   object
 3   LotFrontage    1201 non-null   float64
 4   LotArea        1460 non-null   int64
 5   Street         1460 non-null   object
 6   Alley          91 non-null     object
 7   LotShape       1460 non-null   object
 8   LandContour    1460 non-null   object
 9   Utilities      1460 non-null   object
 10  LotConfig      1460 non-null   object
 11  LandSlope      1460 non-null   object
 12  Neighborhood   1460 non-null   object
 13  Condition1     1460 non-null   object
 14  Condition2     1460 non-null   object
 15  BldgType       1460 non-null   object
 16  HouseStyle     1460 non-null   object
 17  OverallQual    1460 non-null   int64
 18  OverallCond    1460 non-null   int64
 19  YearBuilt      1460 non-null   int64
 20  YearRemodAdd   1460 non-null   int64
 21  RoofStyle      1460 non-null   object
 22  RoofMatl       1460 non-null   object
 23  Exterior1st    1460 non-null   object
 24  Exterior2nd    1460 non-null   object
 25  MasVnrType     1452 non-null   object
 26  MasVnrArea     1452 non-null   float64
 27  ExterQual      1460 non-null   object
 28  ExterCond      1460 non-null   object
 29  Foundation     1460 non-null   object
 30  BsmtQual       1423 non-null   object
 31  BsmtCond       1423 non-null   object
 32  BsmtExposure   1422 non-null   object
 33  BsmtFinType1   1423 non-null   object
 34  BsmtFinSF1     1460 non-null   int64
 35  BsmtFinType2   1422 non-null   object
 36  BsmtFinSF2     1460 non-null   int64
 37  BsmtUnfSF      1460 non-null   int64
 38  TotalBsmtSF    1460 non-null   int64
 39  Heating        1460 non-null   object
 40  HeatingQC      1460 non-null   object
 41  CentralAir     1460 non-null   object
 42  Electrical     1459 non-null   object
 43  1stFlrSF       1460 non-null   int64
 44  2ndFlrSF       1460 non-null   int64
 45  LowQualFinSF   1460 non-null   int64
 46  GrLivArea      1460 non-null   int64
 47  BsmtFullBath   1460 non-null   int64
 48  BsmtHalfBath   1460 non-null   int64
 49  FullBath       1460 non-null   int64
 50  HalfBath       1460 non-null   int64
 51  BedroomAbvGr   1460 non-null   int64
 52  KitchenAbvGr   1460 non-null   int64
 53  KitchenQual    1460 non-null   object
 54  TotRmsAbvGrd   1460 non-null   int64
 55  Functional     1460 non-null   object
 56  Fireplaces     1460 non-null   int64
 57  FireplaceQu    770 non-null    object
 58  GarageType     1379 non-null   object
 59  GarageYrBlt    1379 non-null   float64
 60  GarageFinish   1379 non-null   object
 61  GarageCars     1460 non-null   int64
 62  GarageArea     1460 non-null   int64
 63  GarageQual     1379 non-null   object
 64  GarageCond     1379 non-null   object
 65  PavedDrive     1460 non-null   object
 66  WoodDeckSF     1460 non-null   int64
 67  OpenPorchSF    1460 non-null   int64
 68  EnclosedPorch  1460 non-null   int64
 69  3SsnPorch      1460 non-null   int64
 70  ScreenPorch    1460 non-null   int64
 71  PoolArea       1460 non-null   int64
 72  PoolQC         7 non-null      object
 73  Fence          281 non-null    object
 74  MiscFeature    54 non-null     object
 75  MiscVal        1460 non-null   int64
 76  MoSold         1460 non-null   int64
 77  YrSold         1460 non-null   int64
 78  SaleType       1460 non-null   object
 79  SaleCondition  1460 non-null   object
 80  SalePrice      1460 non-null   int64
dtypes: float64(3), int64(35), object(43)
memory usage: 924.0+ KB
```

As we can see, there are 81 columns with 1460 row. In some columns there is a huge number of "missing data"(NaN), but it's not quite true. With the view to data description we have, "NaN" in this data means not missing value, but the absence of the thing this column about. For example, we can have in the column "Alley" (means the type of alley access to property) such values as "Grvl" (gravel), "Pave" (paved) an "NA" (means no alley access and displayed in DataFrame as NaN). So cleaning the missing data will drop not only columns with a great number of it, but some columns with non-informative data. All numerical data have type "int64", that's why we can miss the step of changing types.

2.2. Missing Data and Filling the Gaps

Now, we will find out the percentage of missed data in each of the columns and display those with more than 50% missings in it.

In [3]:

```
miss_data_cols = []
for col in df.columns:
    pct_missing = np.mean(df[col].isnull()) #percentage of missed data in column

    if pct_missing > 0.5:
        miss_data_cols.append(col)
        print('{0}: {1:.5f}%'.format(col, pct_missing*100))
```

```
Alley: 93.76712%
PoolQC: 99.52055%
Fence: 80.75342%
MiscFeature: 96.30137%
```

The "leaders" of missing data are "Fence" (81%), "MiscFeature" (96%), "PoolQC"(100%), "Alley" (94%). Last two of them has non-informative data, not gaps. Let's drop all of them, because even if we fill in these gaps, there will be a problem of uninformative data.

In [4]:

```
df = df.drop( miss_data_cols, axis=1)
```

In [5]:

```
df.describe()
```

Out[5]:

	Id	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearBuilt	YearRem
count	1460.000000	1460.000000	1201.000000	1460.000000	1460.000000	1460.000000	1460.000000	1460.
mean	730.500000	56.897260	70.049958	10516.828082	6.099315	5.575342	1971.267808	1984.
std	421.610009	42.300571	24.284752	9981.264932	1.382997	1.112799	30.202904	20.
min	1.000000	20.000000	21.000000	1300.000000	1.000000	1.000000	1872.000000	1950.
25%	365.750000	20.000000	59.000000	7553.500000	5.000000	5.000000	1954.000000	1967.
50%	730.500000	50.000000	69.000000	9478.500000	6.000000	5.000000	1973.000000	1994.
75%	1095.250000	70.000000	80.000000	11601.500000	7.000000	6.000000	2000.000000	2004.
max	1460.000000	190.000000	313.000000	215245.000000	10.000000	9.000000	2010.000000	2010.

2.3. Non-informative features

Let's make a list of features with more than 90% of the same value.

In [6]:

```
num_rows = len(df.index)
low_information_cols = []

for col in df.columns:
    non_inf = df[col].value_counts(dropna=False)
    pct_non_inf = (non_inf/num_rows).iloc[0] #percentage of non-informative data in co

    if pct_non_inf > 0.9:
        low_information_cols.append(col)
        print('{0}: {1:.5f}%'.format(col, pct_non_inf*100))
        print(non_inf)
        print()
```

```
Street: 99.58904%
Pave    1454
Grvl       6
Name: Street, dtype: int64

Utilities: 99.93151%
AllPub    1459
NoSeWa       1
Name: Utilities, dtype: int64

LandSlope: 94.65753%
Gtl    1382
Mod      65
Sev      13
Name: LandSlope, dtype: int64

Condition2: 98.97260%
Norm      1445
Feedr        6
RRNn         2
PosN         2
Artery       2
RRAn         1
PosA         1
RRAe         1
Name: Condition2, dtype: int64

RoofMatl: 98.21918%
CompShg    1434
Tar&Grv      11
WdShngl       6
WdShake       5
ClyTile       1
Membran       1
Metal         1
Roll          1
Name: RoofMatl, dtype: int64

Heating: 97.80822%
GasA     1428
GasW       18
Grav        7
Wall        4
OthW        2
Floor       1
Name: Heating, dtype: int64

CentralAir: 93.49315%
Y    1365
N      95
Name: CentralAir, dtype: int64

Electrical: 91.36986%
SBrkr    1334
FuseA      94
FuseF      27
FuseP       3
Mix         1
NaN         1
Name: Electrical, dtype: int64

LowQualFinSF: 98.21918%
0      1434
80        3
360       2
528       1
53        1
120       1
144       1
156       1
205       1
232       1
234       1
371       1
572       1
390       1
392       1
397       1
420       1
473       1
479       1
481       1
513       1
514       1
515       1
384       1
Name: LowQualFinSF, dtype: int64

BsmtHalfBath: 94.38356%
0    1378
1      80
2       2
Name: BsmtHalfBath, dtype: int64

KitchenAbvGr: 95.34247%
1    1392
2      65
3       2
0       1
Name: KitchenAbvGr, dtype: int64

Functional: 93.15068%
Typ     1360
Min2      34
Min1      31
Mod       15
Maj1      14
Maj2       5
Sev        1
Name: Functional, dtype: int64

GarageCond: 90.82192%
TA     1326
NaN      81
Fa       35
Gd        9
Po        7
Ex        2
Name: GarageCond, dtype: int64

PavedDrive: 91.78082%
Y    1340
N      90
P      30
Name: PavedDrive, dtype: int64

3SsnPorch: 98.35616%
0      1436
168       3
216       2
144       2
180       2
245       1
238       1
290       1
196       1
182       1
407       1
304       1
162       1
153       1
320       1
140       1
130       1
96        1
23        1
508       1
Name: 3SsnPorch, dtype: int64

ScreenPorch: 92.05479%
0      1344
192       6
224       5
120       5
189       4
180       4
160       3
168       3
144       3
126       3
147       3
90        3
200       2
198       2
216       2
184       2
259       2
100       2
176       2
170       2
288       2
142       2
153       1
154       1
152       1
155       1
145       1
156       1
143       1
322       1
140       1
161       1
128       1
122       1
410       1
119       1
116       1
99        1
95        1
80        1
63        1
60        1
53        1
40        1
130       1
175       1
163       1
165       1
291       1
287       1
276       1
273       1
271       1
266       1
265       1
263       1
260       1
252       1
234       1
233       1
225       1
374       1
222       1
220       1
210       1
204       1
197       1
385       1
190       1
396       1
185       1
182       1
440       1
178       1
312       1
480       1
Name: ScreenPorch, dtype: int64

PoolArea: 99.52055%
0      1453
738       1
648       1
576       1
555       1
519       1
512       1
480       1
Name: PoolArea, dtype: int64

MiscVal: 96.43836%
0        1408
400        11
500         8
700         5
450         4
2000        4
600         4
1200        2
480         2
1150        1
800         1
15500       1
620         1
3500        1
560         1
2500        1
1300        1
1400        1
350         1
8300        1
54          1
Name: MiscVal, dtype: int64
```

It turns out, that almost all of houses have pave streets, no alley access, no pool, all neccessary utilities, and other standart characteristics are required. That's why they are not useful for our model and we can drop them.

In [7]:

```
df = df.drop(low_information_cols, axis=1)
```

Let's take a look what we have in the end of the cleaning:

In [8]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 59 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Id             1460 non-null   int64
 1   MSSubClass     1460 non-null   int64
 2   MSZoning       1460 non-null   object
 3   LotFrontage    1201 non-null   float64
 4   LotArea        1460 non-null   int64
 5   LotShape       1460 non-null   object
 6   LandContour    1460 non-null   object
 7   LotConfig      1460 non-null   object
 8   Neighborhood   1460 non-null   object
 9   Condition1     1460 non-null   object
 10  BldgType       1460 non-null   object
 11  HouseStyle     1460 non-null   object
 12  OverallQual    1460 non-null   int64
 13  OverallCond    1460 non-null   int64
 14  YearBuilt      1460 non-null   int64
 15  YearRemodAdd   1460 non-null   int64
 16  RoofStyle      1460 non-null   object
 17  Exterior1st    1460 non-null   object
 18  Exterior2nd    1460 non-null   object
 19  MasVnrType     1452 non-null   object
 20  MasVnrArea     1452 non-null   float64
 21  ExterQual      1460 non-null   object
 22  ExterCond      1460 non-null   object
 23  Foundation     1460 non-null   object
 24  BsmtQual       1423 non-null   object
 25  BsmtCond       1423 non-null   object
 26  BsmtExposure   1422 non-null   object
 27  BsmtFinType1   1423 non-null   object
 28  BsmtFinSF1     1460 non-null   int64
 29  BsmtFinType2   1422 non-null   object
 30  BsmtFinSF2     1460 non-null   int64
 31  BsmtUnfSF      1460 non-null   int64
 32  TotalBsmtSF    1460 non-null   int64
 33  HeatingQC      1460 non-null   object
 34  1stFlrSF       1460 non-null   int64
 35  2ndFlrSF       1460 non-null   int64
 36  GrLivArea      1460 non-null   int64
 37  BsmtFullBath   1460 non-null   int64
 38  FullBath       1460 non-null   int64
 39  HalfBath       1460 non-null   int64
 40  BedroomAbvGr   1460 non-null   int64
 41  KitchenQual    1460 non-null   object
 42  TotRmsAbvGrd   1460 non-null   int64
 43  Fireplaces     1460 non-null   int64
 44  FireplaceQu    770 non-null    object
 45  GarageType     1379 non-null   object
 46  GarageYrBlt    1379 non-null   float64
 47  GarageFinish   1379 non-null   object
 48  GarageCars     1460 non-null   int64
 49  GarageArea     1460 non-null   int64
 50  GarageQual     1379 non-null   object
 51  WoodDeckSF     1460 non-null   int64
 52  OpenPorchSF    1460 non-null   int64
 53  EnclosedPorch  1460 non-null   int64
 54  MoSold         1460 non-null   int64
 55  YrSold         1460 non-null   int64
 56  SaleType       1460 non-null   object
 57  SaleCondition  1460 non-null   object
 58  SalePrice      1460 non-null   int64
dtypes: float64(3), int64(28), object(28)
memory usage: 673.1+ KB
```

There are 59 columns instead 81.

3. EDA

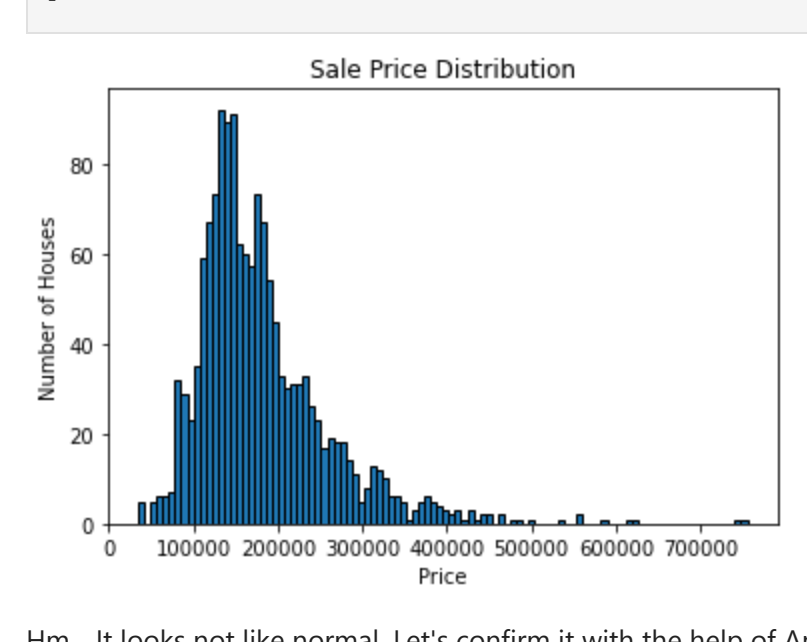
EDA is a very important step in any data science research. It allows you to reveal hidden connections between features and understand what exactly is needed to feed into the model. It also includes searching for unnatural data - outliers that can greatly distort the results.

3.1. Single Variable Plots

Insofar as we'll predicte estate prices, let's investigate the distribution of this value using mathplotlib.

In [9]:

```
plt.hist(df['SalePrice'].dropna(), bins = 100, edgecolor = 'k');
plt.xlabel('Price'); plt.ylabel('Number of Houses');
plt.title('Sale Price Distribution');
```



Hm... It looks not like normal. Let's confirm it with the help of Anderson–Darling test.

In [10]:

```
def anderson_test(massive):
    result = anderson(massive) #testin
    print('Statistics: %.3f' % result.statistic)

    #interpriting
    for i in range(len(result.critical_values)):
        sl, cv = result.significance_level[i], result.critical_values[i]
        if result.statistic < result.critical_values[i]:
            print('%.3f: %.3f, no reasons to reject H0' % (sl, cv))
        else:
            print('%.3f: %.3f, H0 is rejected' % (sl, cv))

anderson_test(df['SalePrice'])
```

```
Statistics: 41.692
15.000: 0.574, H0 is rejected
10.000: 0.654, H0 is rejected
5.000: 0.785, H0 is rejected
2.500: 0.916, H0 is rejected
1.000: 1.089, H0 is rejected
```

Insofar as this is not normal distribution, we'll use Spearman rank correlation in order to find the most influential features.

3.2. Looking for Relationships

Correlation search helps to understand what characteristics of the house significantly affect its price.

In [11]:

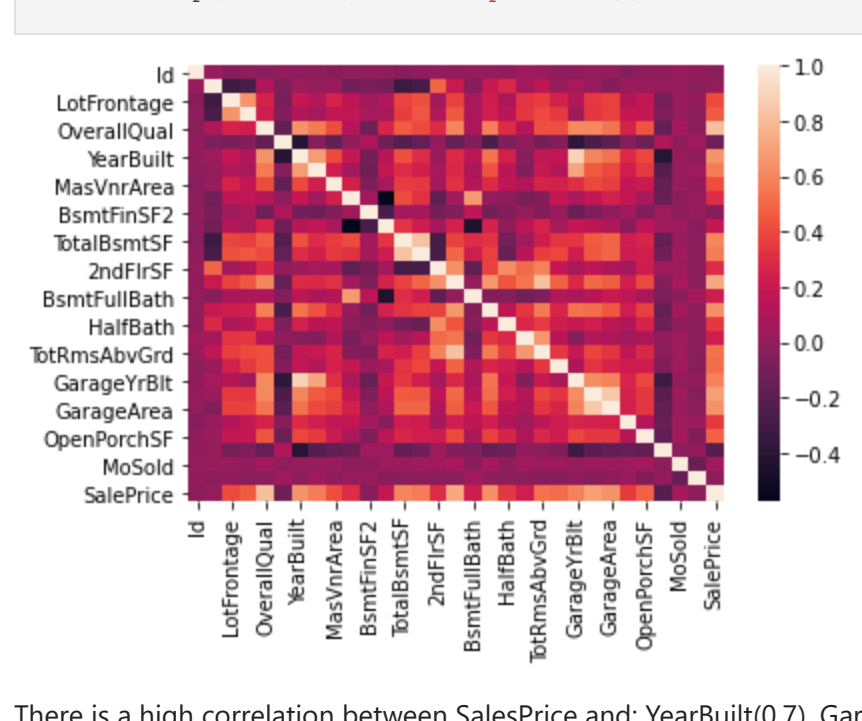
```
correlations_data = df.corr(method='spearman')['SalePrice'].sort_values()
correlations_data
```

Out[11]:

```
EnclosedPorch   -0.218394
OverallCond     -0.129325
BsmtFinSF2      -0.038806
YrSold          -0.029899
Id              -0.018546
MSSubClass       0.007192
MoSold           0.069432
BsmtUnfSF        0.185197
BsmtFullBath     0.225125
BedroomAbvGr     0.234907
2ndFlrSF         0.293598
BsmtFinSF1       0.301871
HalfBath         0.343008
WoodDeckSF       0.353802
LotFrontage      0.409076
MasVnrArea       0.421309
LotArea          0.456461
OpenPorchSF      0.477561
Fireplaces       0.519247
TotRmsAbvGrd     0.532586
YearRemodAdd     0.571159
1stFlrSF         0.575408
GarageYrBlt      0.593788
TotalBsmtSF      0.602725
FullBath         0.635957
GarageArea       0.649379
YearBuilt        0.652682
GarageCars       0.690711
GrLivArea        0.731310
OverallQual      0.809829
SalePrice        1.000000
Name: SalePrice, dtype: float64
```

In [12]:

```
sns.heatmap(df.corr(method='spearman'));
```



There is a high correlation between SalesPrice and: YearBuilt(0.7), GarageCars(0.7), GrLivArea(0.7) and OverallQual(0.8). So the main characteristics are original constructor date, size of garage in car capacity, above grade(ground) living area square feet and rates the overall material and finish of the house.

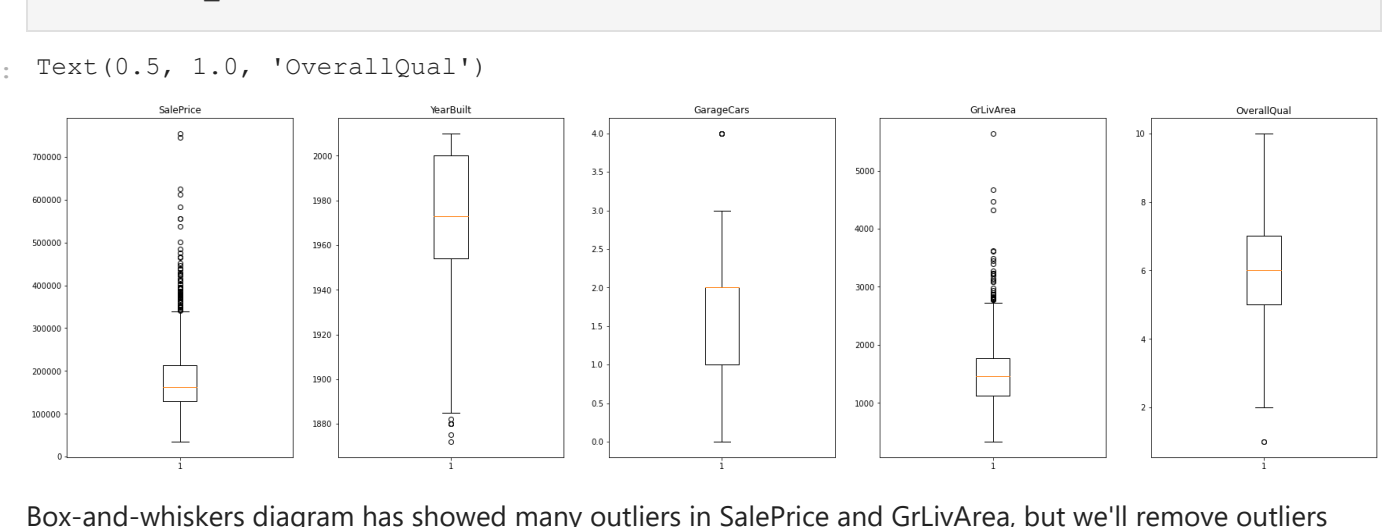
3.3. Outliers

We've figured out which columns are the most important. Now let's look for outliers in them.

In [13]:

```
fig, axs = plt.subplots(1, 5, figsize=(30, 8))

axs[0].boxplot(df['SalePrice'])
axs[0].set_title('SalePrice')

axs[1].boxplot(df['YearBuilt'])
axs[1].set_title('YearBuilt')

axs[2].boxplot(df['GarageCars'])
axs[2].set_title('GarageCars')

axs[3].boxplot(df['GrLivArea'])
axs[3].set_title('GrLivArea')

axs[4].boxplot(df['OverallQual'])
axs[4].set_title('OverallQual')
```

Out[13]: Text(0.5, 1.0, 'OverallQual')



Box-and-whiskers diagram has showed many outliers in SalePrice and GrLivArea, but we'll remove outliers in all of these columns.

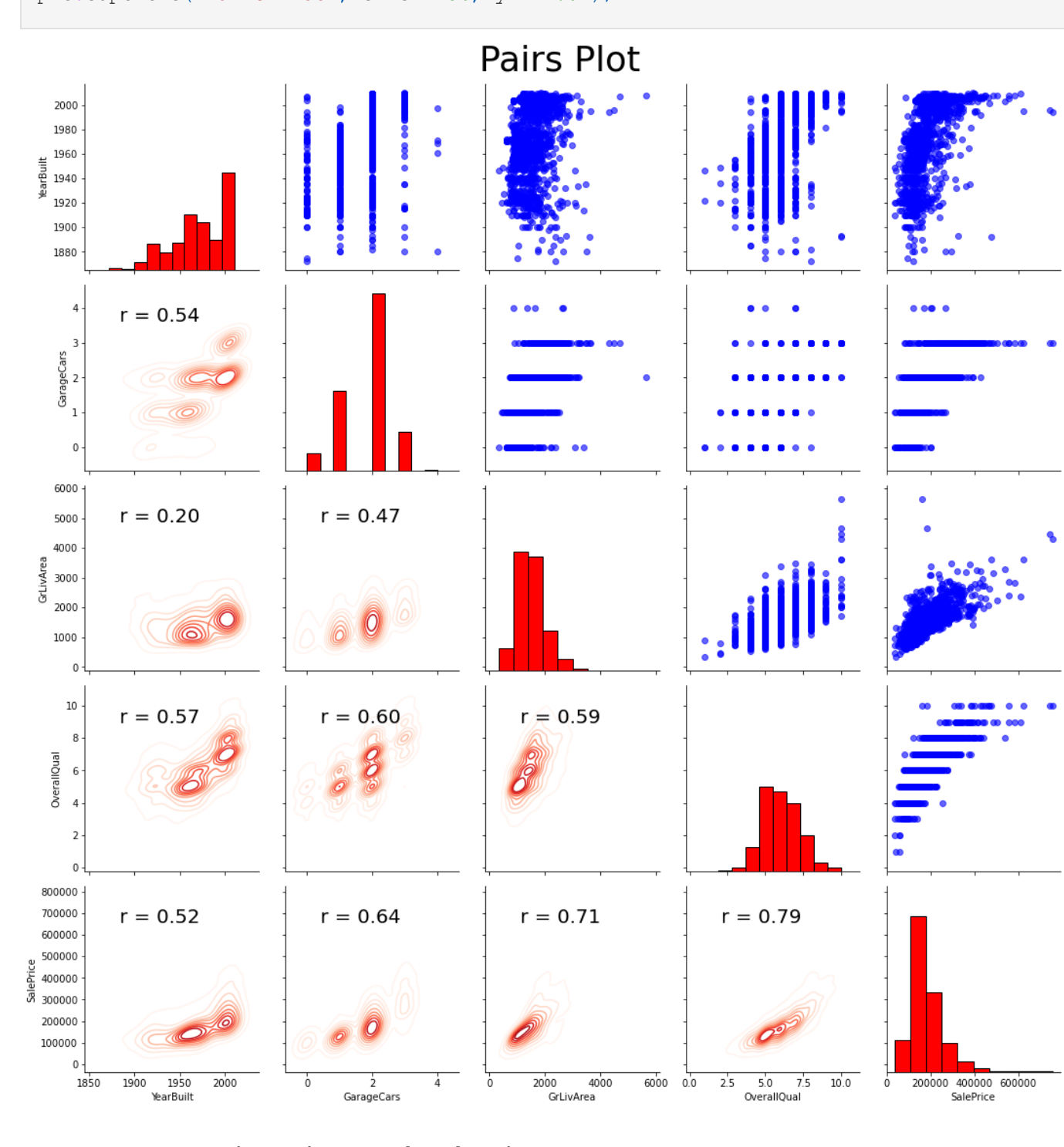
In [14]:

```
def remove_outliers(column, df):
    first_quartile = df[column].describe()['25%']
    third_quartile = df[column].describe()['75%']

    interquartile_range = third_quartile - first_quartile

    df = df[(df[column] > (first_quartile - 3 * interquartile_range)) &
            (df[column] < (third_quartile + 3 * interquartile_range))]

remove_outliers('SalePrice', df)
remove_outliers('YearBuilt', df)
remove_outliers('GarageCars', df)
remove_outliers('GrLivArea', df)
remove_outliers('OverallQual', df)
```

3.4. Pairs Plot

In [15]:

```
# Extract the columns to  plot
plot_data = df[['YearBuilt', 'GarageCars', 'GrLivArea', 'OverallQual', 'SalePrice']]


# Calculation correlation coefficient between two columns
def corr_func(x, y, **kwargs):
    r = np.corrcoef(x, y)[0][1]

    ax = plt.gca()
    ax.annotate("r = {:.2f}".format(r),
                xy=(.2, .8), xycoords=ax.transAxes,
                size = 20)

grid = sns.PairGrid(data = plot_data, height = 3)

grid.map_upper(plt.scatter, color = 'blue', alpha = 0.6)# Upper - scatter plot
grid.map_diag(plt.hist, color = 'red', edgecolor = 'black')# Diagonal is a histogram

grid.map_lower(corr_func);# Bottom - correlation and density plot
grid.map_lower(sns.kdeplot, cmap = plt.cm.Reds)

plt.suptitle('Pairs Plot', size = 36, y = 1.02);
```



4. Features Engineering and Selection

It was decided to choose theese features: YearBuilt, GarageCars, GrLivArea and OverallQual. Insofar as they are numerical, we woun't do one-hot encode the categorical variables.

In [16]:

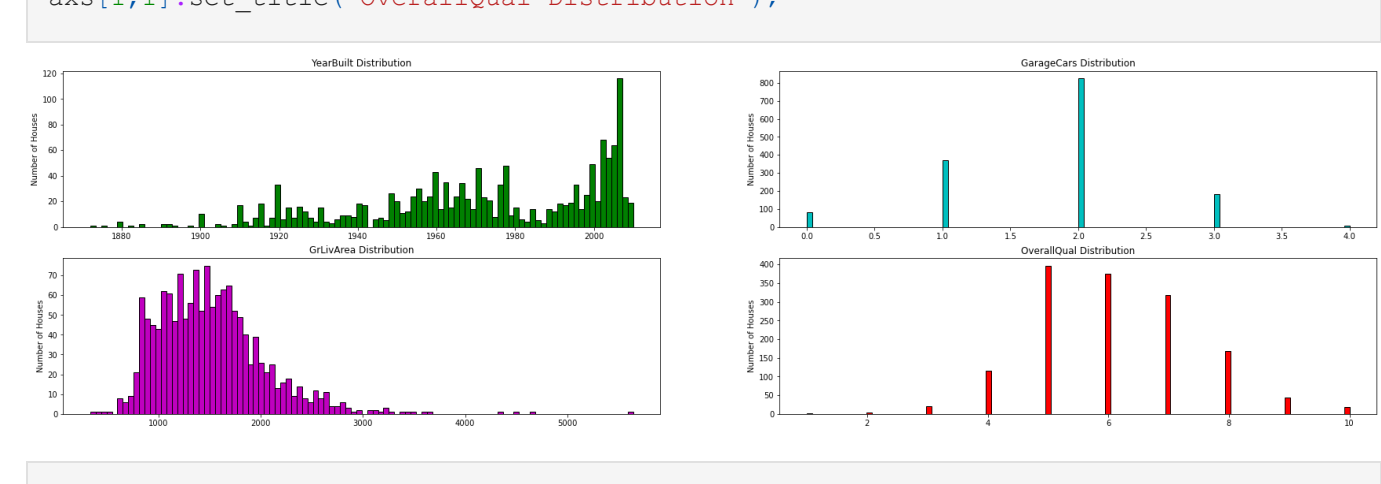
```
data = df[['YearBuilt', 'GarageCars', 'GrLivArea', 'OverallQual', 'SalePrice']]
data.head()
```

Out[16]:

	YearBuilt	GarageCars	GrLivArea	OverallQual	SalePrice
0	2003	2	1710	7	208500
1	1976	2	1262	6	181500
2	2001	2	1786	7	223500
3	1915	3	1717	7	140000
4	2000	3	2198	8	250000

In [17]:

```
fig, axs = plt.subplots(2, 2, figsize=(30, 8))

axs[0,0].hist(data['YearBuilt'].dropna(), bins = 100, color = 'g', edgecolor = 'k');
axs[0,0].set_title('YearBuilt Distribution')
axs[0,0].set(ylabel ='Number of Houses');

axs[0,1].hist(data['GarageCars'].dropna(), bins = 100, color = 'c', edgecolor = 'k');
axs[0,1].set(ylabel ='Number of Houses');
axs[0,1].set_title('GarageCars Distribution');

axs[1,0].hist(data['GrLivArea'].dropna(), bins = 100, color = 'm', edgecolor = 'k');
axs[1,0].set(ylabel ='Number of Houses');
axs[1,0].set_title('GrLivArea Distribution');

axs[1,1].hist(data['OverallQual'].dropna(), bins = 100, color = 'r', edgecolor = 'k');
axs[1,1].set(ylabel ='Number of Houses');
axs[1,1].set_title('OverallQual Distribution');
```



In [18]:

```
data.describe(include='all')
```

Out[18]:

	YearBuilt	GarageCars	GrLivArea	OverallQual	SalePrice
count	1460.000000	1460.000000	1460.000000	1460.000000	1460.000000
mean	1971.267808	1.767123	1515.463699	6.099315	180921.195890
std	30.202904	0.747315	525.480383	1.382997	79442.502883
min	1872.000000	0.000000	334.000000	1.000000	34900.000000
25%	1954.000000	1.000000	1129.500000	5.000000	129975.000000
50%	1973.000000	2.000000	1464.000000	6.000000	163000.000000
75%	2000.000000	2.000000	1776.750000	7.000000	214000.000000
max	2010.000000	4.000000	5642.000000	10.000000	755000.000000

5. Split Into Training and Testing Sets

To check and evaluate the work of the model, we divide the data into training and test data in the ratio of 70% and 30%.

```
In [19]:
target = data.SalePrice
data = data.drop(['SalePrice'], axis=1)

# Split into 70% training and 30% testing set
data_train, data_test, target_train, target_test = train_test_split(data, target, test

print(data_train.shape)
print(data_test.shape)
print(target_train.shape)
print(target_test.shape)
print(data_train.head())
```

```
(1022, 4)
(438, 4)
(1022,)
(438,)
      YearBuilt  GarageCars  GrLivArea  OverallQual
135        1970           2       1682            7
1452       2005           2       1072            5
762        2009           2       1547            7
932        2006           3       1905            9
435        1996           2       1661            7
```

6. Baseline

```
In [20]:
baseline_estimate = np.median(target_train)

print("The baseline estimate: %0.2f" % baseline_estimate)
print("Baseline Performance on the test data: MAE = %0.4f" % np.mean(abs(target_test -
```

```
The baseline estimate: 165000.00
Baseline Performance on the test data: MAE = 57047.0046
```

7. Creating ang Configuring the Model

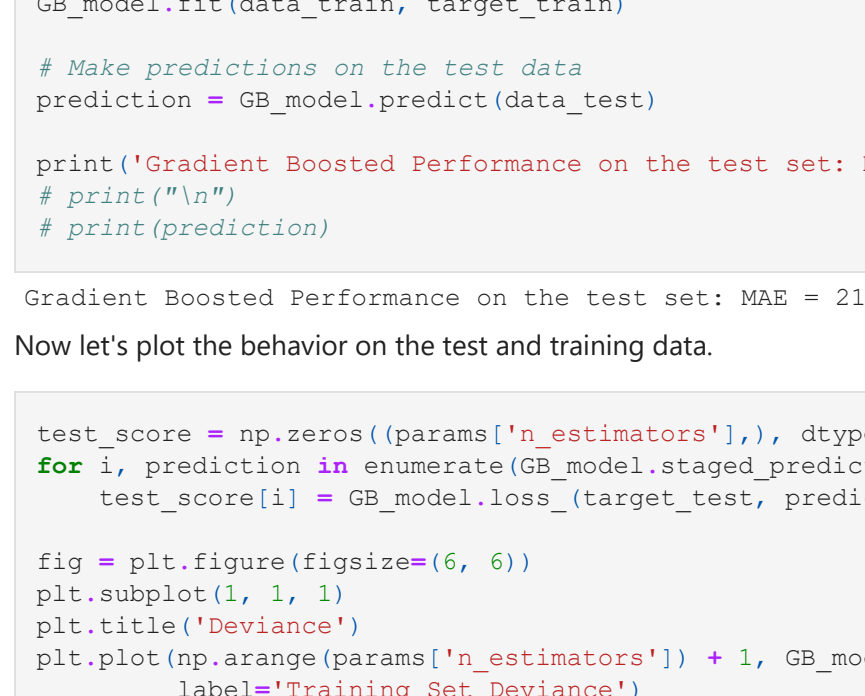
After going through all the important stages, we can finally move on to machine learning. Let's start by creating and training a gradient boosting model and evaluating its performance.

```
In [21]:
params = {'loss': 'ls',
          'learning_rate': 0.01,
          'n_estimators': 300,
          'min_samples_split': 6,
          'max_depth': 4,
          'max_features': None}

# Create the model
GB_model = GradientBoostingRegressor(**params)

# Fit the model on the training data
GB_model.fit(data_train, target_train)

# Make predictions on the test data
prediction = GB_model.predict(data_test)

print('Gradient Boosted Performance on the test set: MAE = %0.4f' % np.mean(abs(predic
# print("\n")
# print(prediction)
```

```
Gradient Boosted Performance on the test set: MAE = 21591.8509
```

Now let's plot the behavior on the test and training data.

```
In [22]:
test_score = np.zeros((params['n_estimators'],), dtype=np.float64)
for i, prediction in enumerate(GB_model.staged_predict(data_test)):
    test_score[i] = GB_model.loss_(target_test, prediction)

fig = plt.figure(figsize=(6, 6))
plt.subplot(1, 1, 1)
plt.title('Deviance')
plt.plot(np.arange(params['n_estimators']) + 1, GB_model.train_score_, 'b-',
         label='Training Set Deviance')
plt.plot(np.arange(params['n_estimators']) + 1, test_score, 'r-',
         label='Test Set Deviance')
plt.legend(loc='upper right')
plt.xlabel('Boosting Iterations')
plt.ylabel('Deviance')
fig.tight_layout()
plt.show()
```



8. Tuning the hyper-parameters

After that, it would be good to break through the hyperparametric tuning of the model to improve its performance. Of course, performance depends most of all on the choice of features, but improvements will not hurt us! To do this, we will write several options for each parameter of the model and run a cross-check with which settings the model works best.

```
In [23]:
loss = ['ls', 'lad', 'huber']
learning_rate = [0.005, 0.01, 0.05, 0.1, 0.2 ]
n_estimators = [250, 300, 500, 600, 700]
min_samples_split = [2, 4, 5, 6, 8]
max_depth = [2, 3, 4, 5, 10]
max_features = ['auto', 'sqrt', 'log2', None]

hyper_grid = {'loss': loss,
              'learning_rate': learning_rate,
              'n_estimators': n_estimators,
              'min_samples_split': min_samples_split,
              'max_depth': max_depth,
              'max_features': max_features}

# Create the model to use for hyperparameter tuning
hyp_tun_model = GradientBoostingRegressor(random_state = 42)

# Set up the random search with 4-fold cross validation
rand_cross_val = RandomizedSearchCV(estimator = hyp_tun_model,
                                    param_distributions = hyper_grid,
                                    cv = 4, n_iter = 25,
                                    scoring = 'neg_mean_absolute_error',
                                    n_jobs = -1, verbose = 1,
                                    return_train_score = True,
                                    random_state = 42)
```

```
In [24]:
rand_cross_val.fit(data_train, target_train)
```

```
Fitting 4 folds for each of 25 candidates, totalling 100 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:   24.0s
[Parallel(n_jobs=-1)]: Done 100 out of 100 | elapsed:  1.9min finished
```

```
Out[24]: RandomizedSearchCV(cv=4, estimator=GradientBoostingRegressor(random_state=42),
                   n_iter=25, n_jobs=-1,
                   param_distributions={'learning_rate': [0.005, 0.01, 0.05,
                                                          0.1, 0.2],
                                        'loss': ['ls', 'lad', 'huber'],
                                        'max_depth': [2, 3, 4, 5, 10],
                                        'max_features': ['auto', 'sqrt', 'log2',
                                                         None],
                                        'min_samples_split': [2, 4, 5, 6, 8],
                                        'n_estimators': [250, 300, 500, 600,
                                                         700]},
                   random_state=42, return_train_score=True,
                   scoring='neg_mean_absolute_error', verbose=1)
```

```
In [25]:
# Get all of the cv results and sort by the test performance
random_results = pd.DataFrame(rand_cross_val.cv_results_).sort_values('mean_test_score

random_results.head()
```

Out[25]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_n_estimators	param_min_samples_split
24	10.303998	0.231196	0.017752	0.003767	600	4
6	0.653240	0.024112	0.006248	0.001299	500	6
3	2.057679	0.111653	0.008246	0.002156	300	2
10	2.978227	0.224596	0.011499	0.004388	700	4
5	1.881749	0.051051	0.008500	0.002063	500	8

The best settings look like this:

```
In [26]:
rand_cross_val.best_estimator_
```

```
Out[26]: GradientBoostingRegressor(learning_rate=0.01, loss='lad', max_depth=10,
                          max_features='sqrt', min_samples_split=4,
                          n_estimators=600, random_state=42)
```

Now let's compare the performance of the tuned model using the function and the performance of the model that I configured.

```
In [27]:
# Select the best model
final_model = rand_cross_val.best_estimator_

GB_prediction = GB_model.predict(data_test)
final_prediction = final_model.predict(data_test)

print('Default model performance on the test set: MAE = %0.4f.' % np.mean(abs(target_t
print('Final model performance on the test set:   MAE = %0.4f.' % np.mean(abs(target_t
```

```
Default model performance on the test set: MAE = 21591.8509.
Final model performance on the test set:   MAE = 21682.5007.
```

Whether it's about variants, or because of an unsuccessful choice of features, a manually tuned model works a little more accurately.

Prediction for Final Data

Now we can use the trained model to predict on test data.

```
In [28]:
test_data = pd.DataFrame(pd.read_csv('test.csv'))
data_for_pred = test_data[['YearBuilt', 'GarageCars', 'GrLivArea', 'OverallQual']]
# data_for_pred.describe()
data_for_pred['GarageCars'] = data_for_pred['GarageCars'].fillna(0)
data_for_pred.describe()
```

```
<ipython-input-28-16e116cca7ec>:4: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/use
r_guide/indexing.html#returning-a-view-versus-a-copy
  data_for_pred['GarageCars'] = data_for_pred['GarageCars'].fillna(0)
```

Out[28]:

	YearBuilt	GarageCars	GrLivArea	OverallQual
count	1459.000000	1459.000000	1459.000000	1459.000000
mean	1971.357779	1.764907	1486.045922	6.078821
std	30.390071	0.777056	485.566099	1.436812
min	1879.000000	0.000000	407.000000	1.000000
25%	1953.000000	1.000000	1117.500000	5.000000
50%	1973.000000	2.000000	1432.000000	6.000000
75%	2001.000000	2.000000	1721.000000	7.000000
max	2010.000000	5.000000	5095.000000	10.000000

```
In [29]:
test_prediction = GB_model.predict(data_for_pred)
test_prediction
```

```
Out[29]: array([125651.53134214, 140092.60928191, 167375.08279449, ...,
       146739.45693515, 128791.82944048, 232667.35227322])
```

```
In [30]:
test_prediction = pd.DataFrame({'SalePrice': np.around(test_prediction, decimals = 2)
# test_prediction

final = pd.concat([test_data['Id'], test_prediction], axis = 1)
# final
```

We have prediction, so we can write it into csv-file.

```
In [31]:
final.to_csv('result.csv', index = False)
```