HousePricesPart1

October 14, 2021

1 House Prices: Advanced Regression Techniques (Part 1)

In this notebook we apply linear regression to some data from a Kaggle. The notebook is divided into two sections. First we perform some in-depth data exploration and pre-processing, next we build the actual models.

https://www.kaggle.com/c/house-prices-advanced-regression-techniques

This notebook is derived from other existing notebooks by other authors, like,

- https://www.kaggle.com/serigne/stacked-regressions-top-4-on-leaderboard

First we import some of the libraries we will need:

[ ]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib

import matplotlib.pyplot as plt
from scipy import stats
from scipy.stats import skew
from scipy.stats import norm
from scipy.stats.stats import pearsonr

%config InlineBackend.figure_format = 'retina' #set 'png' here when working on␣
↪notebook
%matplotlib inline
```

[ ]:
```
train = pd.read_csv("HousePricesTrain.csv")
test = pd.read_csv("HousePricesTest.csv")
```

[ ]:
```
train.head()
```

[ ]:
```
   Id  MSSubClass MSZoning  LotFrontage  LotArea Street Alley LotShape  \
0   1          60       RL         65.0     8450   Pave   NaN      Reg
1   2          20       RL         80.0     9600   Pave   NaN      Reg
2   3          60       RL         68.0    11250   Pave   NaN      IR1
3   4          70       RL         60.0     9550   Pave   NaN      IR1
4   5          60       RL         84.0    14260   Pave   NaN      IR1
```

```
  LandContour Utilities    ...     PoolArea PoolQC Fence MiscFeature MiscVal  \
0         Lvl    AllPub    ...            0    NaN   NaN         NaN       0
1         Lvl    AllPub    ...            0    NaN   NaN         NaN       0
2         Lvl    AllPub    ...            0    NaN   NaN         NaN       0
3         Lvl    AllPub    ...            0    NaN   NaN         NaN       0
4         Lvl    AllPub    ...            0    NaN   NaN         NaN       0

  MoSold YrSold  SaleType  SaleCondition  SalePrice
0      2   2008        WD         Normal     208500
1      5   2007        WD         Normal     181500
2      9   2008        WD         Normal     223500
3      2   2006        WD        Abnorml     140000
4     12   2008        WD         Normal     250000

[5 rows x 81 columns]
```

[ ]:
```
test.head()
```

[ ]:
```
     Id  MSSubClass MSZoning  LotFrontage  LotArea Street Alley LotShape  \
0  1461          20       RH         80.0    11622   Pave   NaN      Reg
1  1462          20       RL         81.0    14267   Pave   NaN      IR1
2  1463          60       RL         74.0    13830   Pave   NaN      IR1
3  1464          60       RL         78.0     9978   Pave   NaN      IR1
4  1465         120       RL         43.0     5005   Pave   NaN      IR1

  LandContour Utilities      ...       ScreenPorch PoolArea PoolQC  Fence  \
0         Lvl    AllPub      ...               120        0    NaN  MnPrv
1         Lvl    AllPub      ...                 0        0    NaN    NaN
2         Lvl    AllPub      ...                 0        0    NaN  MnPrv
3         Lvl    AllPub      ...                 0        0    NaN    NaN
4         HLS    AllPub      ...               144        0    NaN    NaN

  MiscFeature MiscVal MoSold  YrSold  SaleType  SaleCondition
0         NaN       0      6    2010        WD         Normal
1        Gar2   12500      6    2010        WD         Normal
2         NaN       0      3    2010        WD         Normal
3         NaN       0      6    2010        WD         Normal
4         NaN       0      1    2010        WD         Normal

[5 rows x 80 columns]
```

1.1 Note: There is No Class Label in the Test Set!

Note that the data set provided in a competition (and typically by a client) consist of a single batch of label data which we should use to build the model and possibly a set of “test” data without the target label. Your goal is to use the training set to develop a model to generate the target label for the data in the test set. You cannot use the test set for evaluating your model. In this case, we can only use the training dataset.

We generate a data set containing all the data except the class label for preprocessing purposes.

[ ]:
```
all_data = pd.concat((train.loc[:,'MSSubClass':'SaleCondition'],
                      test.loc[:,'MSSubClass':'SaleCondition']))
```

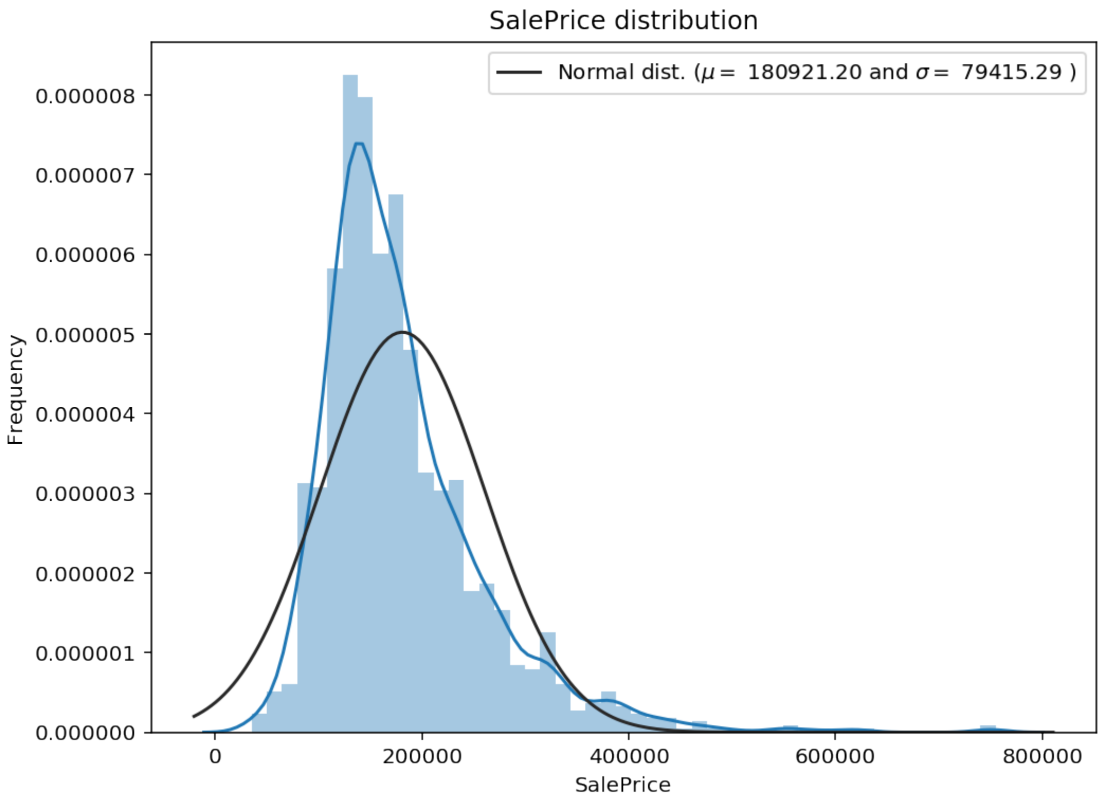
1.2 Data Exploration

First, we should explore the data. We start from the target variable **SalePrice**, the variable we need to predict. We check its distribution.

[ ]:
```
matplotlib.rcParams['figure.figsize'] = (8.0, 6.0)
# fit the data with a normal distribution and
sns.distplot(train['SalePrice'] , fit=norm)

# and check the fitted mu and sigma
(mu, sigma) = norm.fit(train['SalePrice'])
print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))
print("Skewness: %f" % train['SalePrice'].skew())
print("Kurtosis: %f" % train['SalePrice'].kurt())

plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu,␣
 ↪sigma)],
            loc='best')
plt.ylabel('Frequency')
plt.title('SalePrice distribution')
```

```
 mu = 180921.20 and sigma = 79415.29

Skewness: 1.882876
Kurtosis: 6.536282
```

[ ]:
```
Text(0.5,1,'SalePrice distribution')
```



SalePrice has a skewed distribution. We can make **SalePrice** more normal by by taking log(**SalePrice** + 1) and this is generally a good practise.

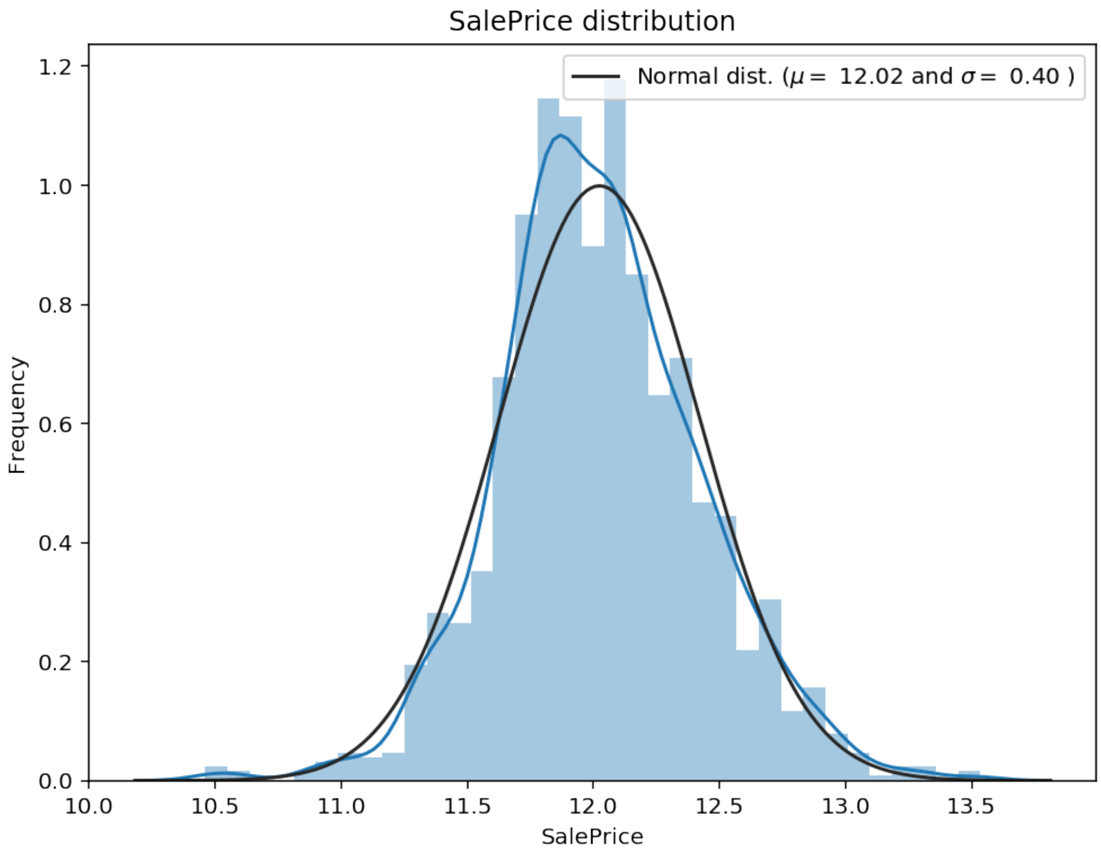
```
[ ]: #We use the numpy fuction log1p which  applies log(1+x) to all elements of the␣
     ↪column
     train["SalePrice"] = np.log1p(train["SalePrice"])
     (mu, sigma) = norm.fit(train['SalePrice'])
     print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))
     print("Skewness: %f" % train['SalePrice'].skew())
     print("Kurtosis: %f" % train['SalePrice'].kurt())
```

```
 mu = 12.02 and sigma = 0.40

Skewness: 0.121347
Kurtosis: 0.809519
```

```
[ ]: matplotlib.rcParams['figure.figsize'] = (8.0, 6.0)
     sns.distplot(train['SalePrice'] , fit=norm)
     plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu,␣
     ↪sigma)],loc='best')
     plt.ylabel('Frequency')
```

```
plt.title('SalePrice distribution')
```

[ ]: Text(0.5,1,'SalePrice distribution')



1.3 Missing Values

Let's check how many missing values are in the data set and how can we deal with them.

[ ]:
```
all_data_na = (all_data.isnull().sum() / len(all_data)) * 100
all_data_na = all_data_na.drop(all_data_na[all_data_na == 0].index).
 ↪sort_values(ascending=False)[:30]
missing_data = pd.DataFrame({'Missing Ratio' :all_data_na})
missing_data.head(10)
```

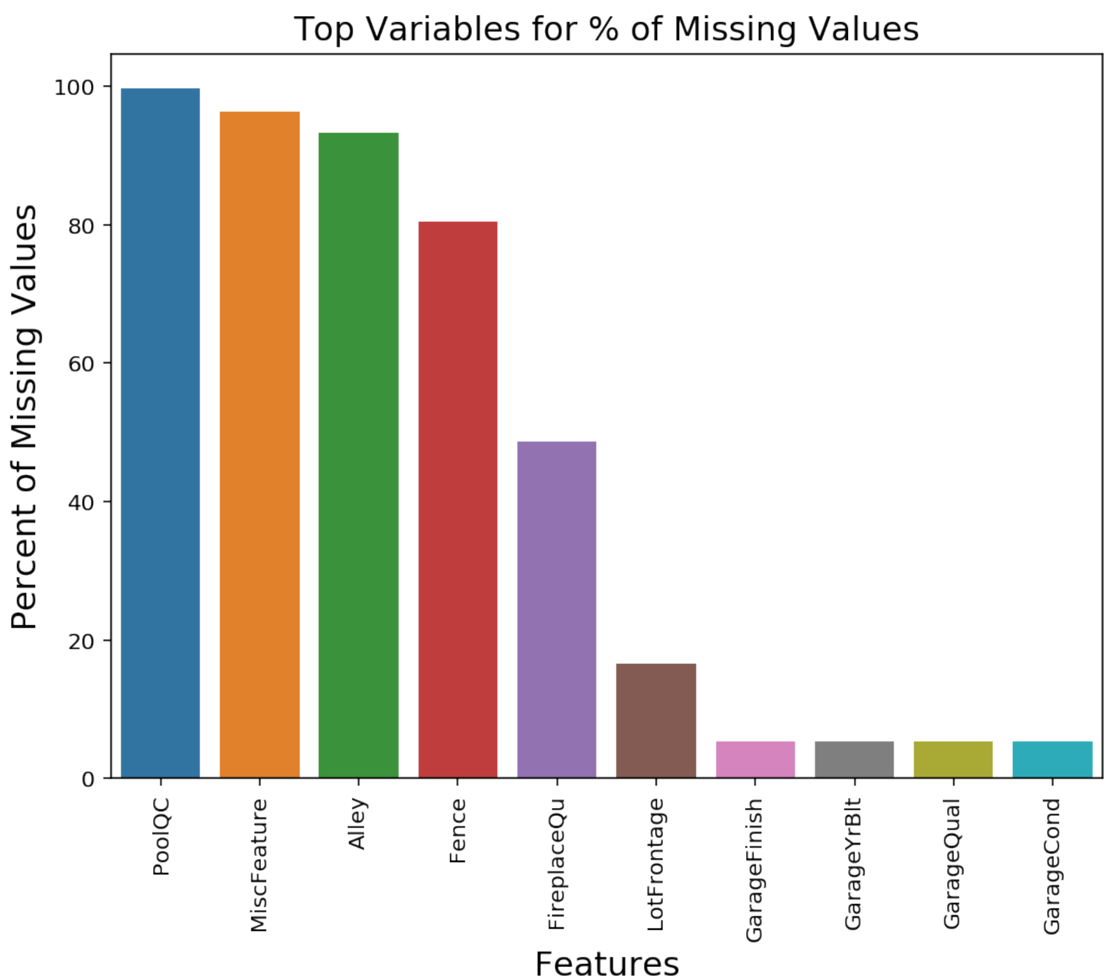
[ ]:

	Missing Ratio
PoolQC	99.657417
MiscFeature	96.402878
Alley	93.216855
Fence	80.438506
FireplaceQu	48.646797

```
LotFrontage       16.649538
GarageFinish       5.447071
GarageYrBlt        5.447071
GarageQual         5.447071
GarageCond         5.447071
```

[ ]:
```
f, ax = plt.subplots(figsize=(8,6))
plt.xticks(rotation='90')
sns.barplot(x=all_data_na.index[:10], y=all_data_na[:10])
plt.xlabel('Features', fontsize=15)
plt.ylabel('Percent of Missing Values', fontsize=15)
plt.title('Top Variables for % of Missing Values', fontsize=15)
```

[ ]:
```
Text(0.5,1,'Top Variables for % of Missing Values')
```



We note that some of the attributes have the vast majority of the values set to unknown. If we apply imputation without any knowledge about the meaning of what a missing value represent, we would end up with attributes set almost completely to the same value. So they would be almost useless.

How can we deal with all these missing values? First, we need to ask the domain expert whether some missing values have a special meaning. We actually don't have a domain expert but we have the data description in which we find out that

- Alley: Type of alley access to property, NA means "No alley access"
- BsmtCond: Evaluates the general condition of the basement, NA means "No Basement"
- BsmtExposure: Refers to walkout or garden level walls, NA means "No Basement"
- BsmtFinType1: Rating of basement finished area, NA means "No Basement"
- BsmtFinType2: Rating of basement finished area (if multiple types), NA means "No Basement"
- FireplaceQu: Fireplace quality, NA means "No Fireplace"
- Functional: data description says NA means typical
- GarageType: Garage location, NA means "No Garage"
- GarageFinish: Interior finish of the garage, NA means "No Garage"
- GarageQual: Garage quality, NA means "No Garage"
- GarageCond: Garage condition, NA means "No Garage"
- PoolQC: Pool quality, NA means "No Pool"
- Fence: Fence quality, NA means "No Fence"
- MiscFeature: Miscellaneous feature not covered in other categories, NA means "None"

Accordingly, we need to keep this information into account when dealing with the missing values of these variables.

1.4 Impute Categorical Values with Known Meaning

Let's impute the missing values for these attributes

[ ]:
```python
all_data["Alley"] = all_data["Alley"].fillna("None")

# for all basement features a missing value means that there is no basement
for col in ('BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1',␣
↪'BsmtFinType2'):
    all_data[col] = all_data[col].fillna('None')

all_data["FireplaceQu"] = all_data["FireplaceQu"].fillna("None")

all_data["Functional"] = all_data["Functional"].fillna("Typ")

for col in ('GarageType', 'GarageFinish', 'GarageQual', 'GarageCond'):
    all_data[col] = all_data[col].fillna('None')

all_data["PoolQC"] = all_data["PoolQC"].fillna("None")

all_data["Fence"] = all_data["Fence"].fillna("None")

all_data["MiscFeature"] = all_data["MiscFeature"].fillna("None")
```

We can also deal with other numerical variables and use some heuristic to impute them. For instance, for **LotFrontage**, since the area of each street connected to the house property most likely have a similar area to other houses in its neighborhood , we can fill in missing values by the median LotFrontage of the neighborhood.

[ ]:
```
all_data["LotFrontage"] = all_data.groupby("Neighborhood")["LotFrontage"].
↪transform(
    lambda x: x.fillna(x.median()))
```

We can also set the garage year, area and number of cars to zero for missing values since this means that there is no garage.

[ ]:
```
for col in ('GarageYrBlt', 'GarageArea', 'GarageCars'):
    all_data[col] = all_data[col].fillna(0)
```

And we can do the same for basement measures.

[ ]:
```
for col in ('BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF','TotalBsmtSF',␣
↪'BsmtFullBath', 'BsmtHalfBath'):
    all_data[col] = all_data[col].fillna(0)
```

When **MasVnrArea** and **MasVnrType** are missing, it most likely means no masonry veneer for these houses. We can fill 0 for the area and None for the type.

[ ]:
```
all_data["MasVnrType"] = all_data["MasVnrType"].fillna("None")
all_data["MasVnrArea"] = all_data["MasVnrArea"].fillna(0)
```

For **MSZoning** (the general zoning classification), 'RL' is by far the most common value. So we can fill in missing values with 'RL'

[ ]:
```
all_data['MSZoning'] = all_data['MSZoning'].fillna(all_data['MSZoning'].
↪mode()[0])
```

Utilities has all records are "AllPub", except for one "NoSeWa" and 2 missing values. Since the house with 'NoSewa' is in the training set, this feature won't help to predict labels in the test set. We can then safely remove it.

[ ]:
```
all_data = all_data.drop(['Utilities'], axis=1)
```

KitchenQual has only one missing value, and same as Electrical, we set it to the most frequent values (that is 'TA')

[ ]:
```
all_data['KitchenQual'] = all_data['KitchenQual'].
↪fillna(all_data['KitchenQual'].mode()[0])
```

Exterior1st and **Exterior2nd** have only one missing value. We will use the mode (the most common value)

[ ]:
```
all_data['Exterior1st'] = all_data['Exterior1st'].
↪fillna(all_data['Exterior1st'].mode()[0])
```

```
all_data['Exterior2nd'] = all_data['Exterior2nd'].
 ↪fillna(all_data['Exterior2nd'].mode()[0])
```

For **SaleType** we can use the most frequent value (the mode) which correspond to “WD”

[ ]:
```
all_data['SaleType'] = all_data['SaleType'].fillna(all_data['SaleType'].
 ↪mode()[0])
```

For **MSSubClass** a missing value most likely means No building class. We can replace missing values with None

[ ]:
```
all_data['MSSubClass'] = all_data['MSSubClass'].fillna("None")
```

Electrical has one missing value. Since this feature has mostly ‘SBrkr’, we can set that for the missing value.

[ ]:
```
all_data['Electrical'] = all_data['Electrical'].fillna(all_data['Electrical'].
 ↪mode()[0])
```

Anymore missing?

[ ]:
```
all_data_na = (all_data.isnull().sum() / len(all_data)) * 100
all_data_na = all_data_na.drop(all_data_na[all_data_na == 0].index).
 ↪sort_values(ascending=False)[:30]
missing_data = pd.DataFrame({'Missing Ratio' :all_data_na})
missing_data.head(10)
```

[ ]:
```
Empty DataFrame
Columns: [Missing Ratio]
Index: []
```

No more missing values! What would have happened if we did not use or did not have the data description?

1.5 Distribution of Numerical Variables

We now explore the distribution of numerical variables. As we did for the class, we will apply the log1p function to all the skewed numerical variables.

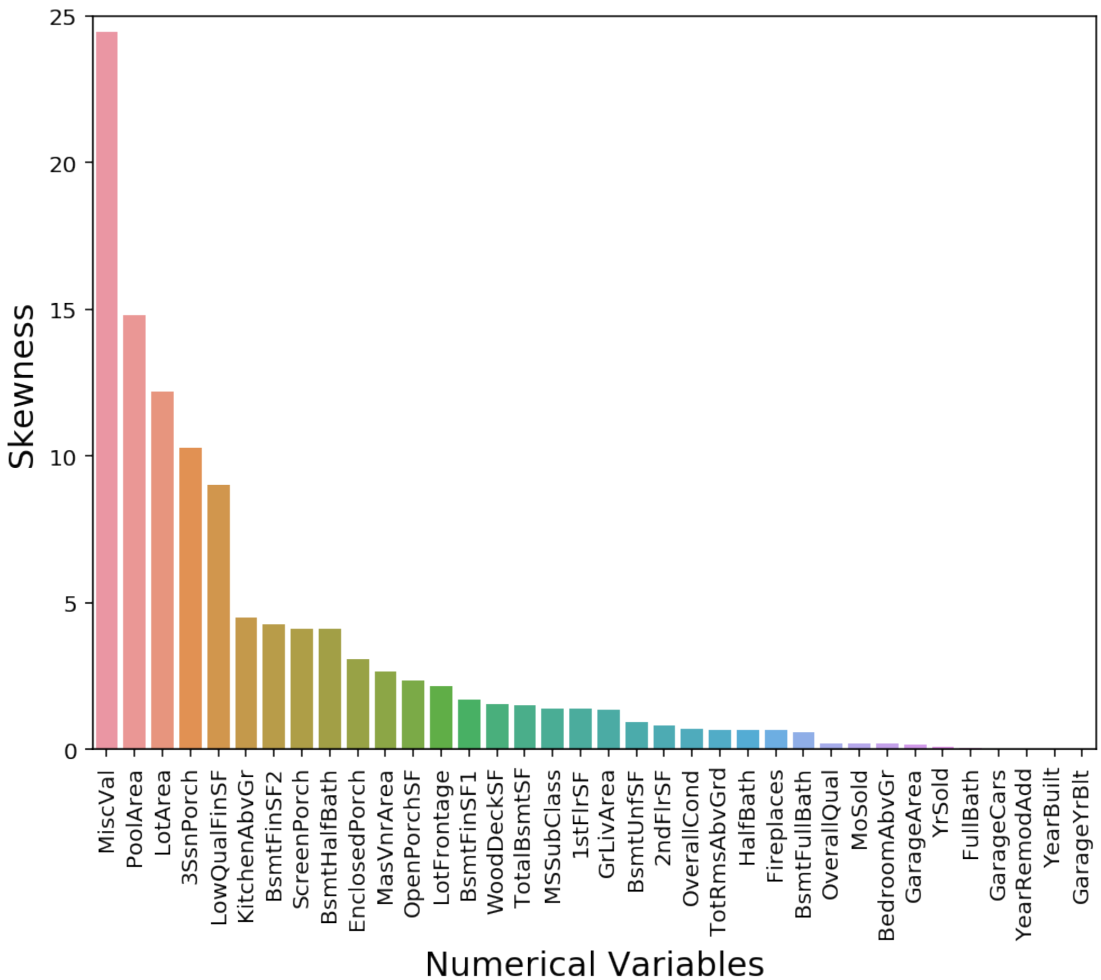
[ ]:
```
# take the numerical features
numeric_feats = all_data.dtypes[all_data.dtypes != "object"].index

# compute the skewness but only for non missing variables (we already imputed␣
 ↪them but just in case ...)
skewed_feats = train[numeric_feats].apply(lambda x: skew(x.dropna()))

skewness = pd.DataFrame({"Variable":skewed_feats.index, "Skewness":skewed_feats.
 ↪data})
# select the variables with a skewness above a certain threshold
```

[ ]:

```
skewness = skewness.sort_values('Skewness', ascending=[0])

f, ax = plt.subplots(figsize=(8,6))
plt.xticks(rotation='90')
sns.barplot(x=skewness['Variable'], y=skewness['Skewness'])
plt.ylim(0,25)
plt.xlabel('Numerical Variables', fontsize=15)
plt.ylabel('Skewness', fontsize=15)
plt.title('', fontsize=15)
```

[ ]:

```
Text(0.5,1,'')
```

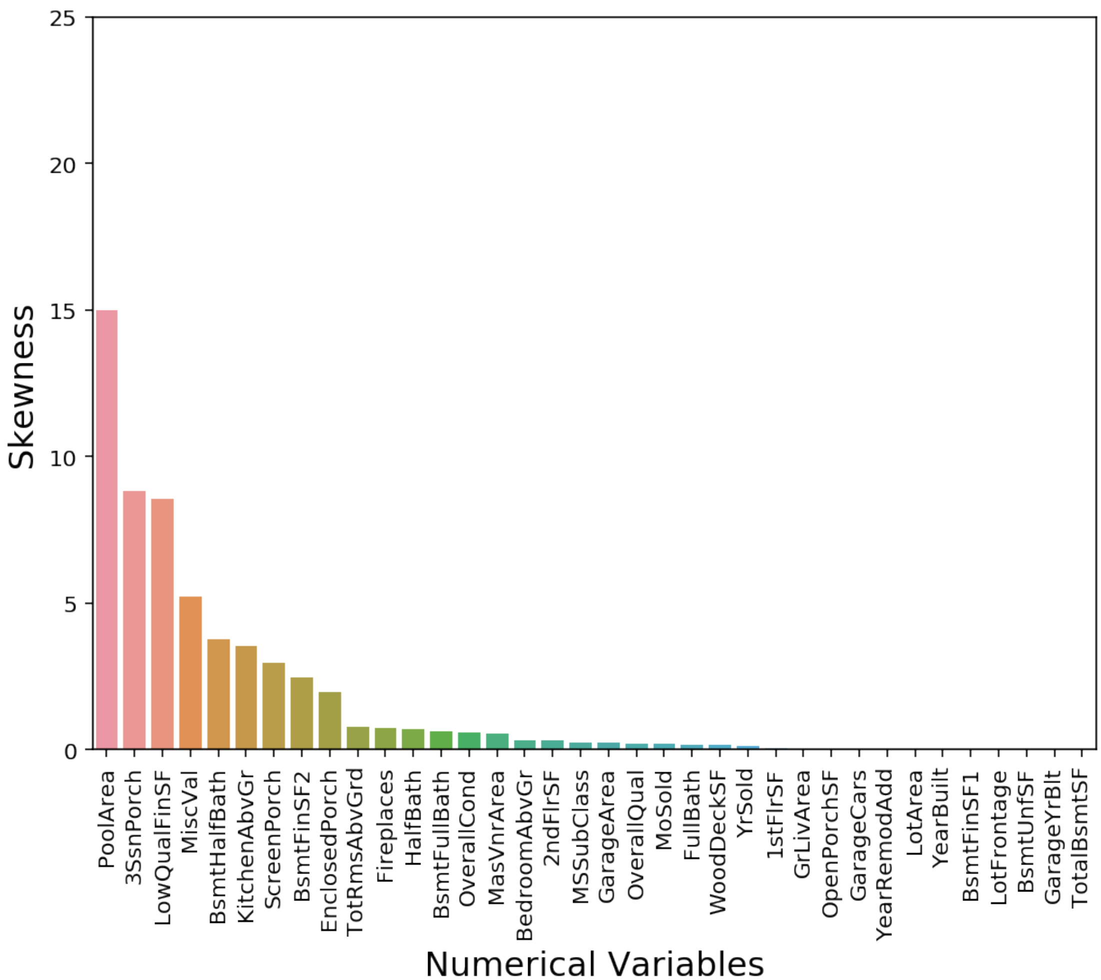


Let’s apply the logarithmic transformation to all the variables with a skewness above a certain threshold (0.75). Then, replot the skewness of attributes. Note that to have a fair comparison the two plots should have the same scale.

[ ]:
```
skewed_feats = skewed_feats[skewed_feats > 0.75]
all_data[skewed_feats.index] = np.log1p(all_data[skewed_feats.index])
```

[ ]:
```
# compute the skewness but only for non missing variables (we already imputed␣
↪them but just in case ...)
skewed_feats = all_data[numeric_feats].apply(lambda x: skew(x.dropna()))
skewness_new = pd.DataFrame({"Variable":skewed_feats.index, "Skewness":
↪skewed_feats.data})
# select the variables with a skewness above a certain threshold

skewness_new = skewness_new.sort_values('Skewness', ascending=[0])

f, ax = plt.subplots(figsize=(8,6))
plt.xticks(rotation='90')
sns.barplot(x=skewness_new['Variable'], y=skewness_new['Skewness'])
plt.ylim(0,25)
plt.xlabel('Numerical Variables', fontsize=15)
plt.ylabel('Skewness', fontsize=15)
plt.title('', fontsize=15)
```

[ ]:
```
Text(0.5,1,'')
```

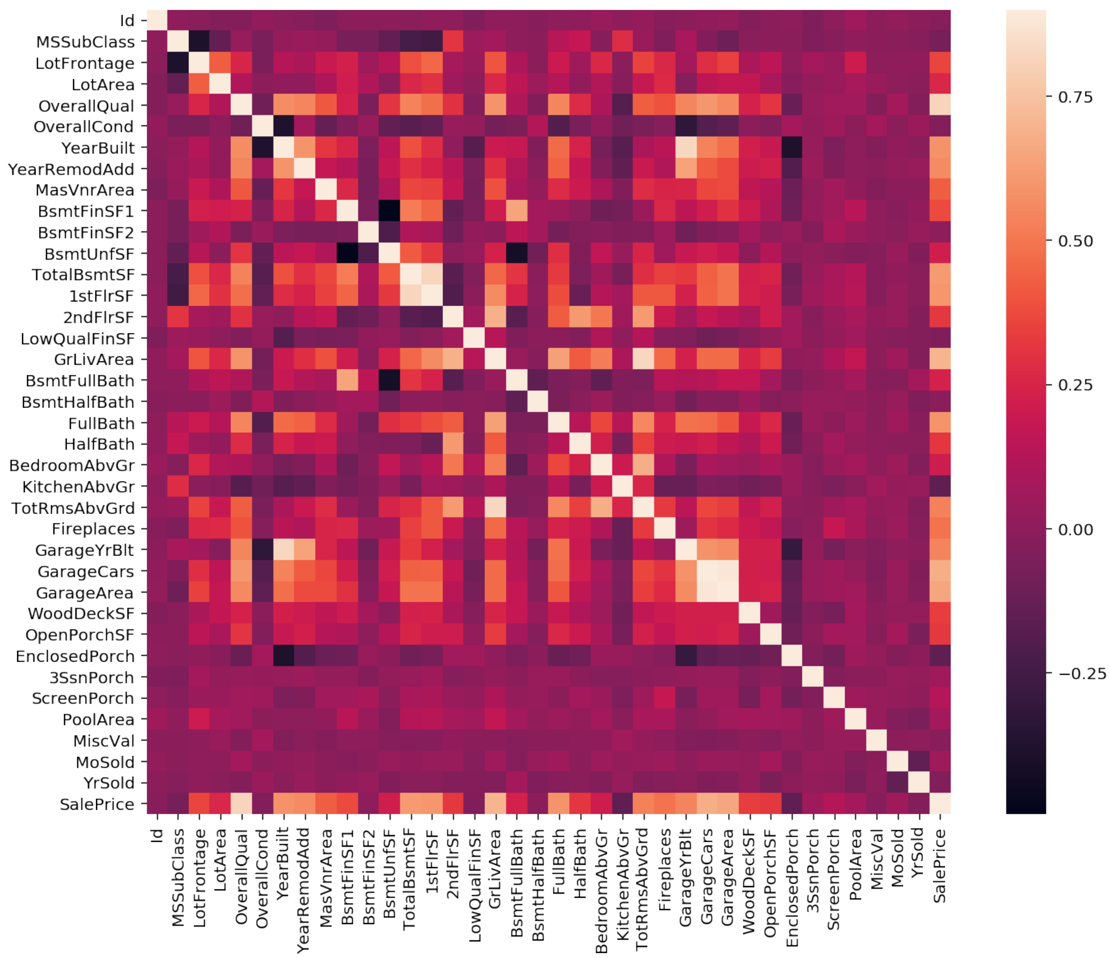


1.6 Correlation Analysis

We can finally perform some correlation analysis.

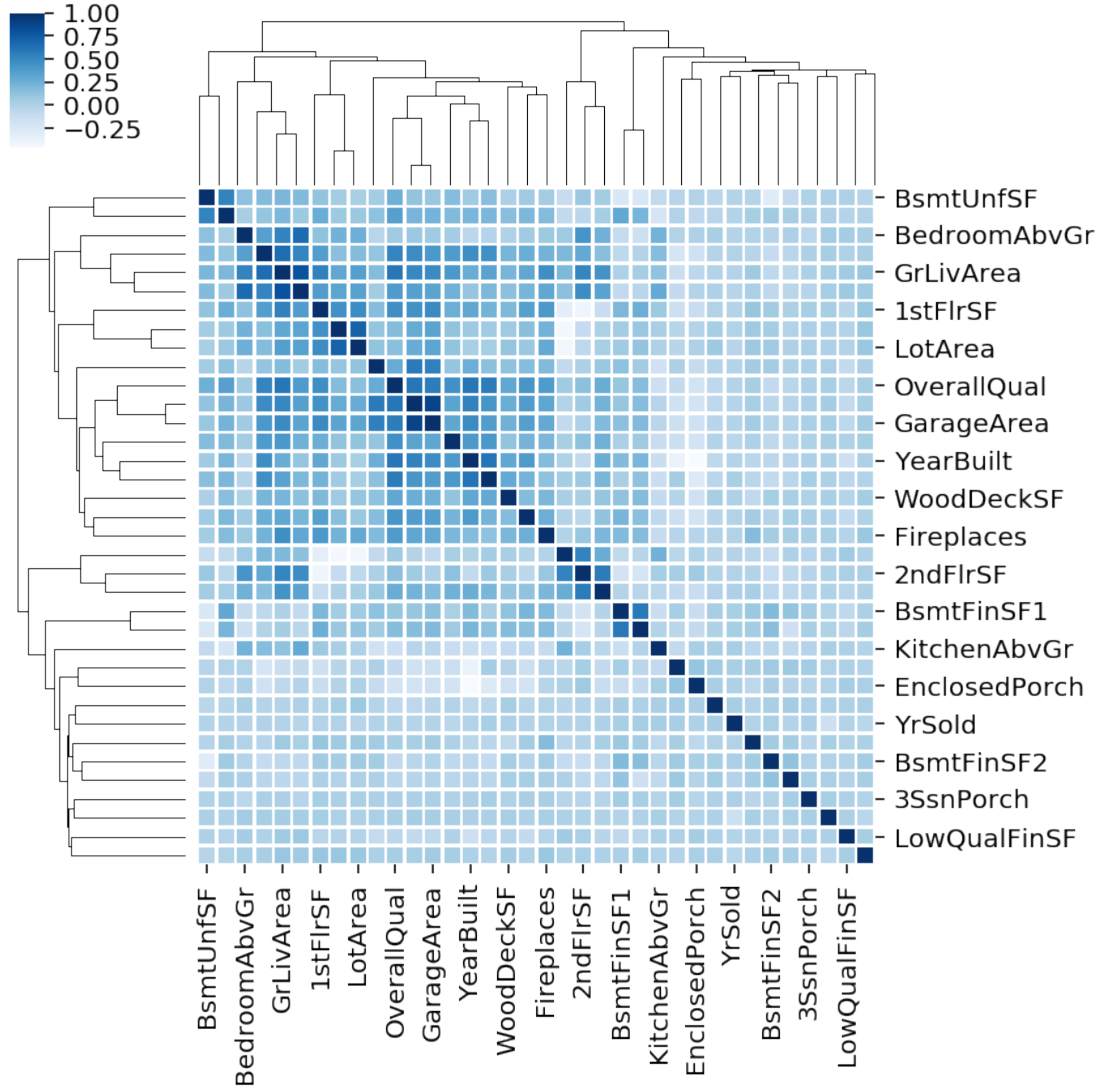
[ ]:
```
corrmat = train.corr()
plt.subplots(figsize=(12,9))
sns.heatmap(corrmat, vmax=0.9, square=True)
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x1a243d4ef0>



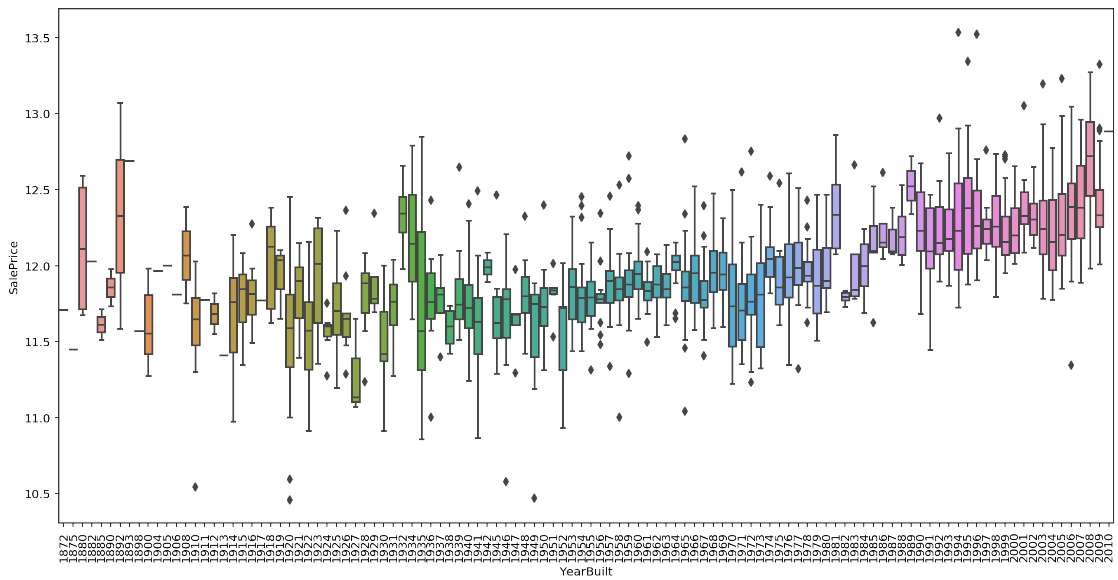
[ ]:
```
import seaborn as sns
sns.clustermap(all_data.corr(), square=True, annot=False, cmap="Blues",
               linewidths=.75, figsize=(6, 6))
```

[ ]: <seaborn.matrix.ClusterGrid at 0x1a2134f1d0>



Or we can further analyze the distribution of some variables.

[ ]:
```
var = 'YearBuilt'
data = pd.concat([train['SalePrice'], train[var]], axis=1)
f, ax = plt.subplots(figsize=(16, 8))
fig = sns.boxplot(x=var, y="SalePrice", data=data)
plt.xticks(rotation=90);
```



1.7 One Hot Encoding

We now generate the one hot encoding for all the categorical variables. Pandas has the function **get_dummies** that generates the binary variables for all the categorical variables

[ ]:
```
print("Number of Variables before OHE: "+str(all_data.shape[1]))
```

```
Number of Variables before OHE: 78
```

[ ]:
```
all_data = pd.get_dummies(all_data)
```

[ ]:
```
print("Number of Variables after OHE: "+str(all_data.shape[1]))
```

```
Number of Variables after OHE: 300
```

1.8 Saving the Preprocessed Data

We create the matrices to be used for computing the models and also save the cleaned data so that we can avoid repeating the process.

[ ]:
```
X_full = all_data[:train.shape[0]].copy()
X_full['SalePrice'] = train.SalePrice
X_full.to_csv("HousePricesTrainClean.csv")

# extract the test examples (we don't have the class value for this)
X_test = all_data[train.shape[0]:]

# save the test data to file
X_test.to_csv("HousePricesTestClean.csv")
```