Problem Statement: Houses are one of the necessary need of each and every person around the globe and therefore housing and real estate market is one of the markets which is one of the major contributors in the world’s economy. It is a very large market and there are various companies working in the domain. Data science comes as a very important tool to solve problems in the domain to help the companies increase their overall revenue, profits, improving their marketing strategies and focusing on changing trends in house sales and purchases. Predictive modelling, Market mix modelling, recommendation systems are some of the machine learning techniques used for achieving the business goals for housing companies. Our problem is related to one such housing company. A US-based housing company named Surprise Housing has decided to enter the Australian market. The company uses data analytics to purchase houses at a price below their actual values and flip them at a higher price. For the same purpose, the company has collected a data set from the sale of houses in Australia. The data is provided in the CSV file below. The company is looking at prospective properties to buy houses to enter the market. You are required to build a model using Machine Learning in order to predict the actual value of the prospective properties and decide whether to invest in them or not. For this company wants to know: • Which variables are important to predict the price of variable? • How do these variables describe the price of the house

import libraries

In [1]:

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.stats import norm,skew
import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
df=pd.read_csv(r'C:\Users\polasasuresh\Downloads\Project-Housing--2---1- (2)
df
```

Out[2]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour
0	127	120	RL	NaN	4928	Pave	NaN	IR1	Lvl
1	889	20	RL	95.0	15865	Pave	NaN	IR1	Lvl
2	793	60	RL	92.0	9920	Pave	NaN	IR1	Lvl
3	110	20	RL	105.0	11751	Pave	NaN	IR1	Lvl
4	422	20	RL	NaN	16635	Pave	NaN	IR1	Lvl
...	...	...	...	...	...	...	...	...	...
1163	289	20	RL	NaN	9819	Pave	NaN	IR1	Lvl
1164	554	20	RL	67.0	8777	Pave	NaN	Reg	Lvl
1165	196	160	RL	24.0	2280	Pave	NaN	Reg	Lvl
1166	31	70	C (all)	50.0	8500	Pave	Pave	Reg	Lvl
1167	617	60	RL	NaN	7861	Pave	NaN	IR1	Lvl

1168 rows × 81 columns

In [3]:
```
df_Test=pd.read_csv(r'C:\Users\polasasuresh\Downloads\Project-Housing--2---1
df_Test
```

Out[3]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour
0	337	20	RL	86.0	14157	Pave	NaN	IR1	HLS
1	1018	120	RL	NaN	5814	Pave	NaN	IR1	Lvl
2	929	20	RL	NaN	11838	Pave	NaN	Reg	Lvl
3	1148	70	RL	75.0	12000	Pave	NaN	Reg	Bnk
4	1227	60	RL	86.0	14598	Pave	NaN	IR1	Lvl
...	...	...	...	...	...	...	...	...	...
287	83	20	RL	78.0	10206	Pave	NaN	Reg	Lvl
288	1048	20	RL	57.0	9245	Pave	NaN	IR2	Lvl
289	17	20	RL	NaN	11241	Pave	NaN	IR1	Lvl
290	523	50	RM	50.0	5000	Pave	NaN	Reg	Lvl
291	1379	160	RM	21.0	1953	Pave	NaN	Reg	Lvl

292 rows × 80 columns

In [4]:
```
df.shape
```

Out[4]: (1168, 81)

In [5]:
```
df_Test.shape
```

Out[5]:
```
(292, 80)
```

There are 1168 rows and 81 columns

```
From looking at the both sets, we can see that the only difference in features is "Sale Price". This makes sense because we are trying to predict it!
```

In [6]:
```
df.dtypes
```

Out[6]:
```
Id                int64
MSSubClass        int64
MSZoning         object
LotFrontage     float64
LotArea           int64
                  ...
MoSold            int64
YrSold            int64
SaleType         object
SaleCondition    object
SalePrice         int64
Length: 81, dtype: object
```

In [7]:
```
df.columns
```

Out[7]:
```
Index(['Id', 'MSSubClass', 'MSZoning', 'LotFrontage', 'LotArea', 'Street',
       'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotConfig',
       'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType',
       'HouseStyle', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd',
       'RoofStyle', 'RoofMatl', 'Exterior1st', 'Exterior2nd', 'MasVnrType',
       'MasVnrArea', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual',
       'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinSF1',
       'BsmtFinType2', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', 'Heating',
       'HeatingQC', 'CentralAir', 'Electrical', '1stFlrSF', '2ndFlrSF',
       'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath',
       'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual',
       'TotRmsAbvGrd', 'Functional', 'Fireplaces', 'FireplaceQu', 'GarageType',
       'GarageYrBlt', 'GarageFinish', 'GarageCars', 'GarageArea', 'GarageQual',
       'GarageCond', 'PavedDrive', 'WoodDeckSF', 'OpenPorchSF',
       'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'PoolQC',
       'Fence', 'MiscFeature', 'MiscVal', 'MoSold', 'YrSold', 'SaleType',
       'SaleCondition', 'SalePrice'],
      dtype='object')
```

There are 81 diiferent columns present in the dataset

In [8]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1168 entries, 0 to 1167
Data columns (total 81 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   Id             1168 non-null   int64  
 1   MSSubClass     1168 non-null   int64  
 2   MSZoning       1168 non-null   object 
 3   LotFrontage    954 non-null    float64
 4   LotArea        1168 non-null   int64  
 5   Street         1168 non-null   object 
 6   Alley          77 non-null     object 
 7   LotShape       1168 non-null   object 
 8   LandContour    1168 non-null   object 
 9   Utilities      1168 non-null   object 
 10  LotConfig      1168 non-null   object 
 11  LandSlope      1168 non-null   object 
 12  Neighborhood   1168 non-null   object 
 13  Condition1     1168 non-null   object 
 14  Condition2     1168 non-null   object 
 15  BldgType       1168 non-null   object 
 16  HouseStyle     1168 non-null   object 
 17  OverallQual    1168 non-null   int64  
 18  OverallCond    1168 non-null   int64  
 19  YearBuilt      1168 non-null   int64  
 20  YearRemodAdd   1168 non-null   int64  
 21  RoofStyle      1168 non-null   object 
 22  RoofMatl       1168 non-null   object 
 23  Exterior1st    1168 non-null   object 
 24  Exterior2nd    1168 non-null   object 
 25  MasVnrType     1161 non-null   object 
 26  MasVnrArea     1161 non-null   float64
 27  ExterQual      1168 non-null   object 
 28  ExterCond      1168 non-null   object 
 29  Foundation     1168 non-null   object 
 30  BsmtQual       1138 non-null   object 
 31  BsmtCond       1138 non-null   object 
 32  BsmtExposure   1137 non-null   object 
 33  BsmtFinType1   1138 non-null   object 
 34  BsmtFinSF1     1168 non-null   int64  
 35  BsmtFinType2   1137 non-null   object 
 36  BsmtFinSF2     1168 non-null   int64  
 37  BsmtUnfSF      1168 non-null   int64  
 38  TotalBsmtSF    1168 non-null   int64  
 39  Heating        1168 non-null   object 
 40  HeatingQC      1168 non-null   object 
 41  CentralAir     1168 non-null   object 
 42  Electrical     1168 non-null   object 
 43  1stFlrSF       1168 non-null   int64  
 44  2ndFlrSF       1168 non-null   int64  
 45  LowQualFinSF   1168 non-null   int64  
 46  GrLivArea      1168 non-null   int64  
 47  BsmtFullBath   1168 non-null   int64  
 48  BsmtHalfBath   1168 non-null   int64  
 49  FullBath       1168 non-null   int64  
```

```
 50  HalfBath       1168 non-null   int64
 51  BedroomAbvGr   1168 non-null   int64
 52  KitchenAbvGr   1168 non-null   int64
 53  KitchenQual    1168 non-null   object
 54  TotRmsAbvGrd   1168 non-null   int64
 55  Functional     1168 non-null   object
 56  Fireplaces     1168 non-null   int64
 57  FireplaceQu    617 non-null    object
 58  GarageType     1104 non-null   object
 59  GarageYrBlt    1104 non-null   float64
 60  GarageFinish   1104 non-null   object
 61  GarageCars     1168 non-null   int64
 62  GarageArea     1168 non-null   int64
 63  GarageQual     1104 non-null   object
 64  GarageCond     1104 non-null   object
 65  PavedDrive     1168 non-null   object
 66  WoodDeckSF     1168 non-null   int64
 67  OpenPorchSF    1168 non-null   int64
 68  EnclosedPorch  1168 non-null   int64
 69  3SsnPorch      1168 non-null   int64
 70  ScreenPorch    1168 non-null   int64
 71  PoolArea       1168 non-null   int64
 72  PoolQC         7 non-null      object
 73  Fence          237 non-null    object
 74  MiscFeature    44 non-null     object
 75  MiscVal        1168 non-null   int64
 76  MoSold         1168 non-null   int64
 77  YrSold         1168 non-null   int64
 78  SaleType       1168 non-null   object
 79  SaleCondition  1168 non-null   object
 80  SalePrice      1168 non-null   int64
dtypes: float64(3), int64(35), object(43)
memory usage: 543.0+ KB
```

There are 35 intger type of data and 43 object type of data and 3 float type of data present in the dataset

EDA

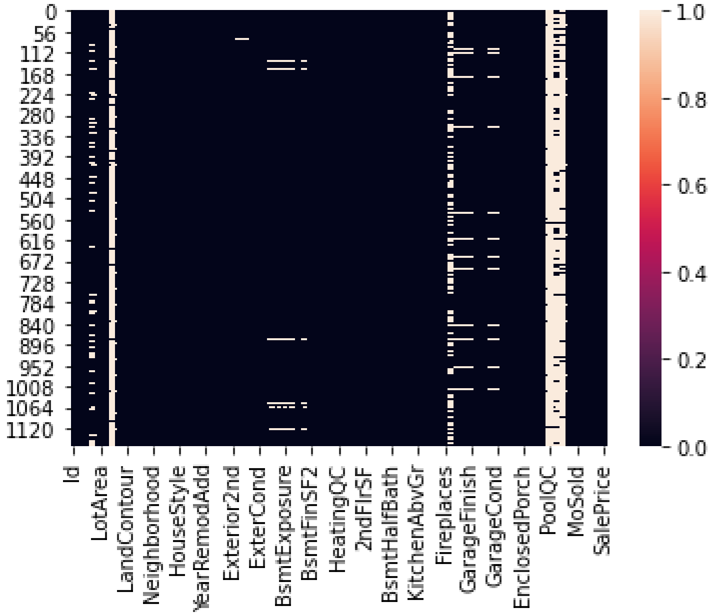
In [9]:
```
df.isnull().sum()
```

Out[9]:
```
Id                 0
MSSubClass         0
MSZoning           0
LotFrontage      214
LotArea            0
                ...
MoSold             0
YrSold             0
SaleType           0
SaleCondition      0
SalePrice          0
Length: 81, dtype: int64
```

There are some null values present in the data

In [10]:
```
sns.heatmap(df.isnull())
```

Out[10]: <AxesSubplot:>



2. Analyzing the Test Variable (Sale Price)

```

Let's check out the most interesting feature in this study: Sale Price. Important Note: This data is from Ames, Iowa. The location is extremely correlated with Sale Price. (I had to take a double-take at a point, since I consider myself a house-browsing enthusiast)
```

In [12]:
```
# Getting Description
df['SalePrice'].describe()
```

Out[12]:
```
count      1168.000000
mean     181477.005993
std       79105.586863
min       34900.000000
25%      130375.000000
50%      163995.000000
75%      215000.000000
max      755000.000000
Name: SalePrice, dtype: float64
```

With an average house price of $180921, it seems like I should relocated to Iowa!

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```

3. Multivariable Analysis

Let's check out all the variables! There are two types of features in housing data, categorical and numerical.

Categorical data is just like it sounds. It is in categories. It isn't necessarily linear, but it follows some kind of pattern. For example, take a feature of "Downtown". The response is either "Near", "Far", "Yes", and "No". Back then, living in downtown usually meant that you couldn't afford to live in uptown. Thus, it could be implied that downtown establishments cost less to live in. However, today, that is not the case. (Thank you, hipsters!) So we can't really establish any particular order of response to be "better" or "worse" than the other.

Numerical data is data in number form. (Who could have thought!) These features are in a linear relationship with each other. For example, a 2,000 square foot place is 2 times "bigger" than a 1,000 square foot place. Plain and simple. Simple and clean.

In [14]:
```
# Checking Categorical Data
df.select_dtypes(include=['object']).columns
```

Out[14]:
```
Index(['MSZoning', 'Street', 'Alley', 'LotShape', 'LandContour', 'Utilities',
       'LotConfig', 'LandSlope', 'Neighborhood', 'Condition1', 'Condition2',
       'BldgType', 'HouseStyle', 'RoofStyle', 'RoofMatl', 'Exterior1st',
       'Exterior2nd', 'MasVnrType', 'ExterQual', 'ExterCond', 'Foundation',
       'BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2',
       'Heating', 'HeatingQC', 'CentralAir', 'Electrical', 'KitchenQual',
       'Functional', 'FireplaceQu', 'GarageType', 'GarageFinish', 'GarageQual',
       'GarageCond', 'PavedDrive', 'PoolQC', 'Fence', 'MiscFeature',
       'SaleType', 'SaleCondition'],
      dtype='object')
```

In [15]:
```
# Checking Numerical Data
df.select_dtypes(include=['int64','float64']).columns
```

Out[15]:
```
Index(['Id', 'MSSubClass', 'LotFrontage', 'LotArea', 'OverallQual',
       'OverallCond', 'YearBuilt', 'YearRemodAdd', 'MasVnrArea', 'BsmtFinSF1',
       'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', '1stFlrSF', '2ndFlrSF',
       'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath',
       'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'TotRmsAbvGrd',
       'Fireplaces', 'GarageYrBlt', 'GarageCars', 'GarageArea', 'WoodDeckSF',
       'OpenPorchSF', 'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea',
       'MiscVal', 'MoSold', 'YrSold', 'SalePrice'],
      dtype='object')
```

In [16]:

```
cat = len(df.select_dtypes(include=['object']).columns)
num = len(df.select_dtypes(include=['int64','float64']).columns)
print('Total Features: ', cat, 'categorical', '+',
      num, 'numerical', '=', cat+num, 'features')
```

```
Total Features:  43 categorical + 38 numerical = 81 features
```

With 81 features, how could we possibly tell which feature is most related to house prices? Good thing we have a correlation matrix. Let's do it!

In [17]:

```
df.corr()
```

Out[17]:

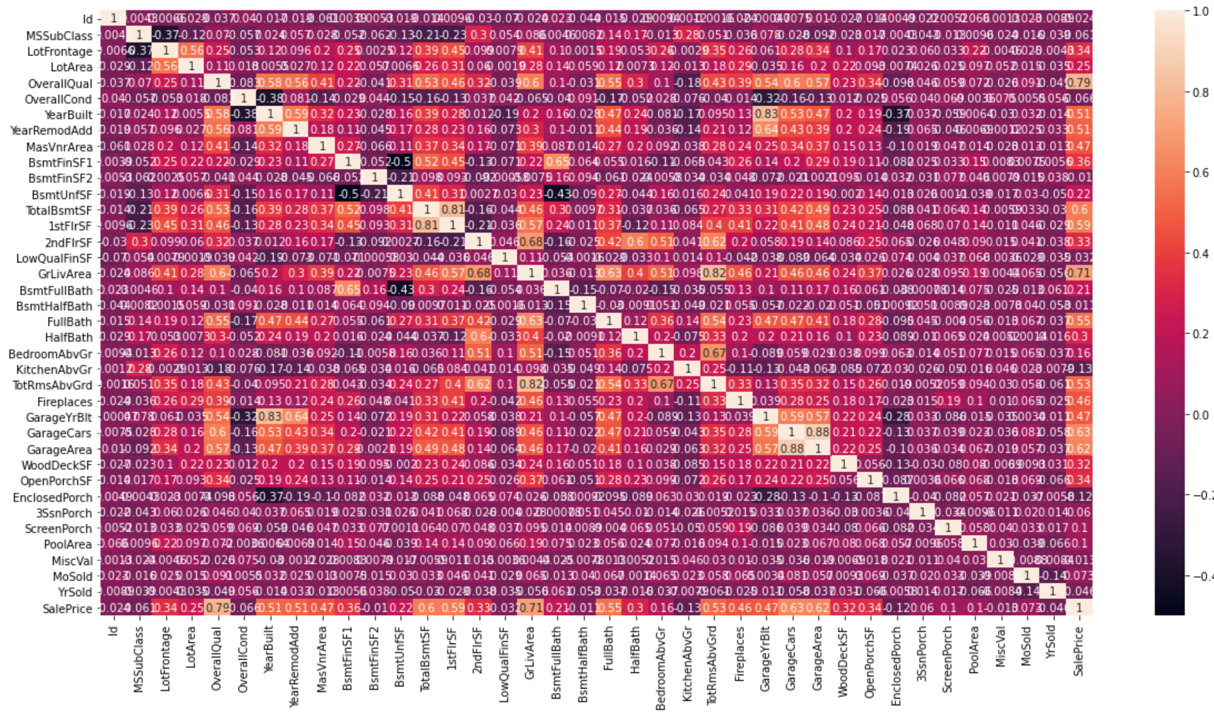
	Id	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	Year
Id	1.000000	0.004259	-0.006629	-0.029212	-0.036965	0.039761	-0.01
MSSubClass	0.004259	1.000000	-0.365220	-0.124151	0.070462	-0.056978	0.02
LotFrontage	-0.006629	-0.365220	1.000000	0.557257	0.247809	-0.053345	0.11
LotArea	-0.029212	-0.124151	0.557257	1.000000	0.107188	0.017513	0.00
OverallQual	-0.036965	0.070462	0.247809	0.107188	1.000000	-0.083167	0.57
OverallCond	0.039761	-0.056978	-0.053345	0.017513	-0.083167	1.000000	-0.37
YearBuilt	-0.016942	0.023988	0.118554	0.005506	0.575800	-0.377731	1.00
YearRemodAdd	-0.018590	0.056618	0.096050	0.027228	0.555945	0.080669	0.59
MasVnrArea	-0.060652	0.027868	0.202225	0.121448	0.409163	-0.137882	0.32
BsmtFinSF1	0.003868	-0.052236	0.247780	0.221851	0.219643	-0.028810	0.22
BsmtFinSF2	0.005269	-0.062403	0.002514	0.056656	-0.040893	0.044336	-0.02
BsmtUnfSF	-0.019494	-0.134170	0.123943	0.006600	0.308676	-0.146384	0.15
TotalBsmtSF	-0.013812	-0.214042	0.386261	0.259733	0.528285	-0.162481	0.38
1stFlrSF	0.009647	-0.227927	0.448186	0.312843	0.458758	-0.134420	0.27
2ndFlrSF	-0.029671	0.300366	0.099250	0.059803	0.316624	0.036668	0.01
LowQualFinSF	-0.070180	0.053737	0.007885	-0.001915	-0.039295	0.041877	-0.18
GrLivArea	-0.024325	0.086448	0.410414	0.281360	0.599700	-0.065006	0.19
BsmtFullBath	0.023027	0.004556	0.104255	0.142387	0.101732	-0.039680	0.16
BsmtHalfBath	-0.043572	0.008207	0.001528	0.059282	-0.030702	0.091016	-0.02
FullBath	-0.015187	0.140807	0.189321	0.123197	0.548824	-0.171931	0.47
HalfBath	-0.028512	0.168423	0.053168	0.007271	0.296134	-0.052125	0.24
BedroomAbvGr	0.009376	-0.013283	0.264010	0.117351	0.099639	0.028393	-0.08
KitchenAbvGr	0.001216	0.283506	-0.002890	-0.013075	-0.178220	-0.076047	-0.16
TotRmsAbvGrd	-0.001613	0.051179	0.351969	0.184546	0.432579	-0.039952	0.09
Fireplaces	-0.024175	-0.035792	0.262076	0.285983	0.390067	-0.013632	0.13
GarageYrBlt	-0.000469	0.077630	0.061101	-0.034981	0.541719	-0.318278	0.82
GarageCars	0.007549	-0.027639	0.276798	0.158313	0.596322	-0.161996	0.52
GarageArea	0.010048	-0.092408	0.344908	0.195162	0.566782	-0.126021	0.47
WoodDeckSF	-0.027498	-0.022609	0.101751	0.216720	0.227137	0.012290	0.20
OpenPorchSF	-0.013642	0.017468	0.167092	0.093080	0.341030	-0.024899	0.19
EnclosedPorch	0.004885	-0.004252	0.023118	-0.007446	-0.098374	0.056074	-0.37
3SsnPorch	-0.021773	-0.043210	0.059508	0.025794	0.045919	0.040476	0.03
ScreenPorch	0.005169	-0.013291	0.033111	0.025256	0.059387	0.069463	-0.05
PoolArea	0.065832	0.009583	0.223429	0.097107	0.072247	-0.003603	0.00

	Id	MSSubClass	LotFrontage	LotArea	OverallQual	OverallCond	YearI
MiscVal	0.001304	-0.023503	-0.004559	0.051679	-0.025786	0.075178	-0.03
MoSold	0.023479	-0.016015	0.025046	0.015141	0.090638	0.005519	0.03
YrSold	-0.008853	-0.038595	-0.004296	-0.035399	-0.048759	0.055517	-0.01
SalePrice	-0.023897	-0.060775	0.341294	0.249499	0.789185	-0.065642	0.51

38 rows × 38 columns

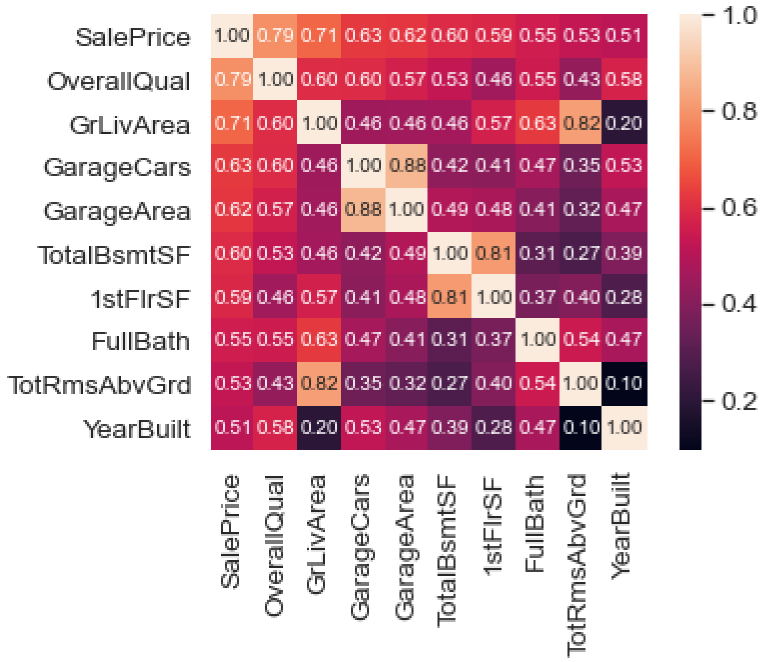
In [18]:
```
plt.figure(figsize=(20,10))
sns.heatmap(df.corr(),annot=True)
```

Out[18]: <AxesSubplot:>



In [20]:

```
# Top 10 Heatmap
k = 10 #number of variables for heatmap
cols = df.corr().nlargest(k, 'SalePrice')['SalePrice'].index
cm = np.corrcoef(df[cols].values.T)
sns.set(font_scale=1.25)
hm = sns.heatmap(cm, cbar=True, annot=True, square=True, fmt='.2f', annot_kw
plt.show()
```



In [21]:
```
most_corr = pd.DataFrame(cols)
most_corr.columns = ['Most Correlated Features']
most_corr
```

Out[21]:

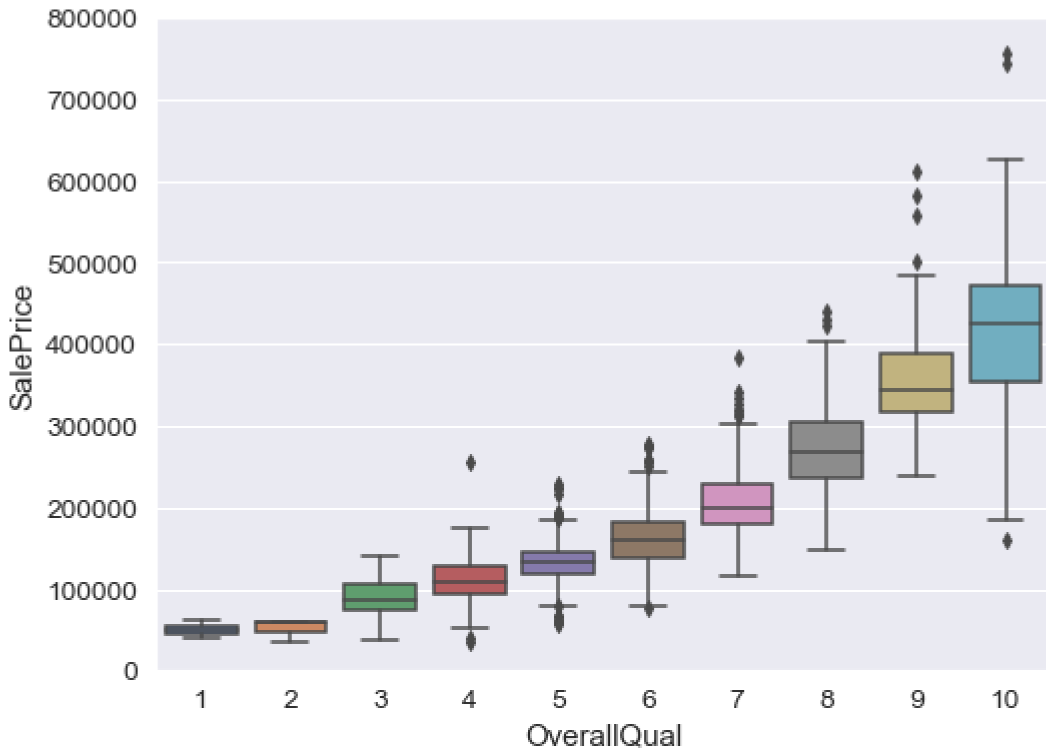
	Most Correlated Features
0	SalePrice
1	OverallQual
2	GrLivArea
3	GarageCars
4	GarageArea
5	TotalBsmtSF
6	1stFlrSF
7	FullBath
8	TotRmsAbvGrd
9	YearBuilt

Well, the most correlated feature to Sale Price is... Sale Price?!? Of course. For the other 9, they are as listed. Here is a short description of each. (Thank you, data_description.txt!)

OverallQual: Rates the overall material and finish of the house (1 = Very Poor, 10 = Very Excellent) GrLivArea: Above grade (ground) living area square feet GarageCars: Size of garage in car capacity GarageArea: Size of garage in square feet TotalBsmtSF: Total square feet of basement area 1stFlrSF: First Floor square feet FullBath: Full bathrooms above grade TotRmsAbvGrd: Total rooms above grade (does not include bathrooms) YearBuilt: Original construction date Let's take a look at how each relates to Sale Price and do some pre-cleaning on each feature if necessary.

In [22]:

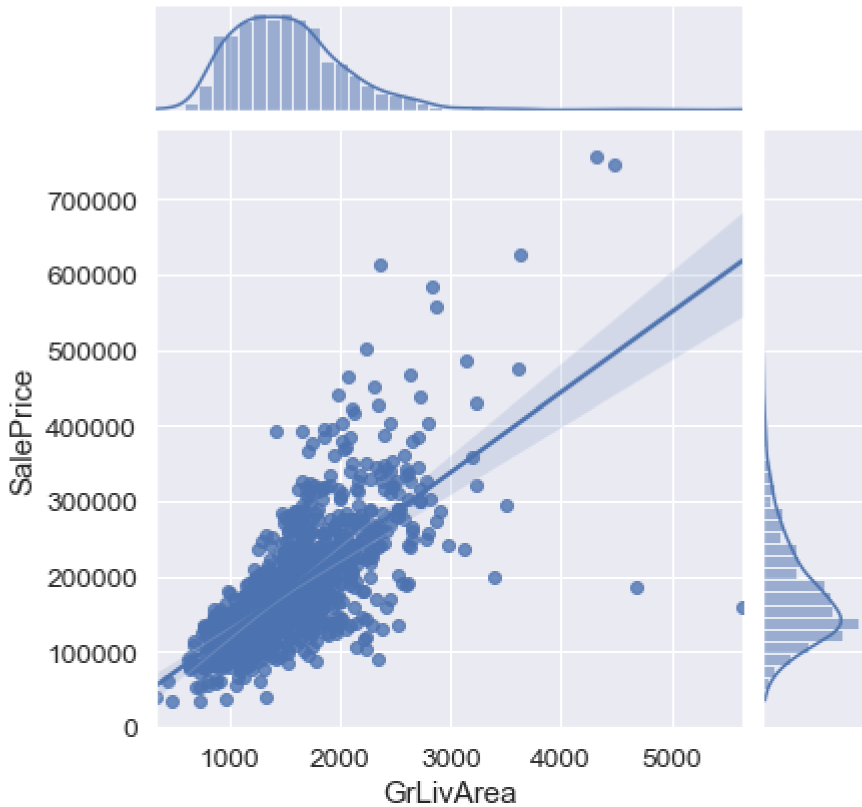
```
# Overall Quality vs Sale Price
var = 'OverallQual'
data = pd.concat([df['SalePrice'], df[var]], axis=1)
f, ax = plt.subplots(figsize=(8, 6))
fig = sns.boxplot(x=var, y="SalePrice", data=data)
fig.axis(ymin=0, ymax=800000);
```



People pay more for better quality? Nothing new here. Let's move on.

In [23]:
```
# Living Area vs Sale Price
sns.jointplot(x=df['GrLivArea'], y=df['SalePrice'], kind='reg')
```

Out[23]: <seaborn.axisgrid.JointGrid at 0xfa045b0>



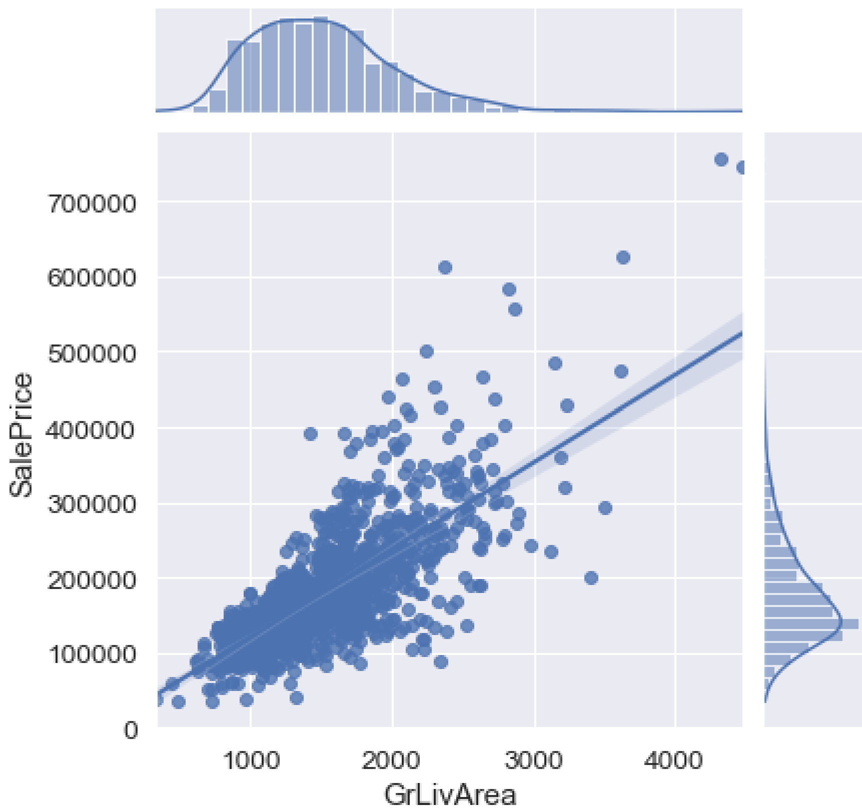
It makes sense that people would pay for the more living area. What doesn't make sense is the two datapoints in the bottom-right of the plot.

We need to take care of this! What we will do is remove these outliers manually.

In [24]:
```
# Removing outliers manually (Two points in the bottom right)
df = df.drop(df[(df['GrLivArea']>4000)
                                    & (df['SalePrice']<300000)].index).reset_index(drop
```

In [25]:
```
# Living Area vs Sale Price
sns.jointplot(x=df['GrLivArea'], y=df['SalePrice'], kind='reg')
```

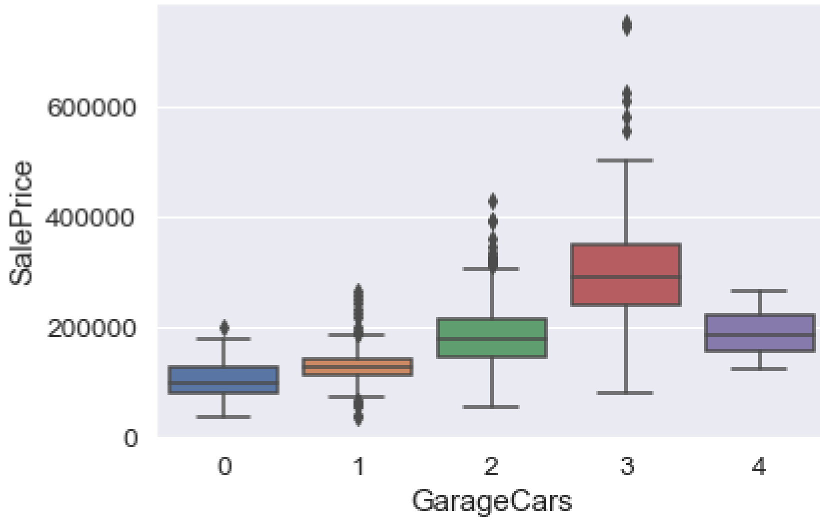
Out[25]: <seaborn.axisgrid.JointGrid at 0x1079d370>



Nice! We got a 0.02 point increase in the Pearson-R Score.

In [26]:
```
# Garage Area vs Sale Price
sns.boxplot(x=df['GarageCars'], y=df['SalePrice'])
```

Out[26]: <AxesSubplot:xlabel='GarageCars', ylabel='SalePrice'>

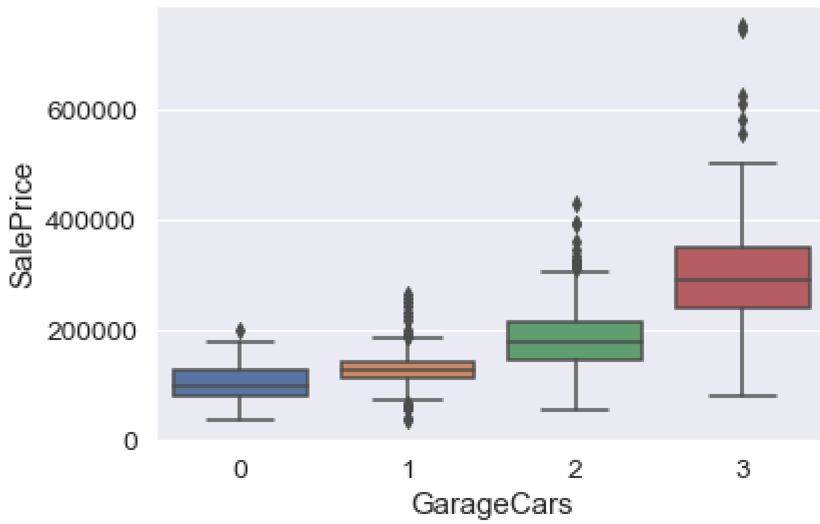


4-car garages result in less Sale Price? That doesn't make much sense. Let's remove those outliers.

In [27]:
```
# Removing outliers manually (More than 4-cars, less than $300k)
df = df.drop(df[(df['GarageCars']>3)
                          & (df['SalePrice']<300000)].index).reset_index(drop
```

In [28]:
```
# Garage Area vs Sale Price
sns.boxplot(x=df['GarageCars'], y=df['SalePrice'])
```

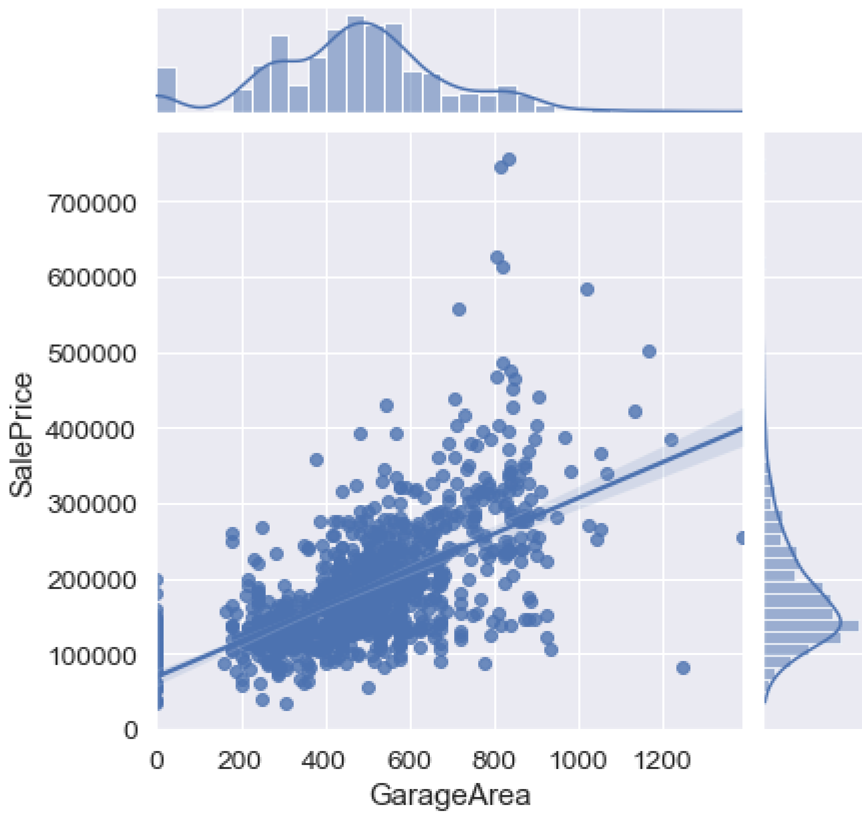
Out[28]: <AxesSubplot:xlabel='GarageCars', ylabel='SalePrice'>



That looks much better. Note: removal of data is totally discretionary and may or may not help in modeling. Use at your own preference.

In [29]:
```
# Garage Area vs Sale Price
sns.jointplot(x=df['GarageArea'], y=df['SalePrice'], kind='reg')
```

Out[29]: <seaborn.axisgrid.JointGrid at 0x10d99838>

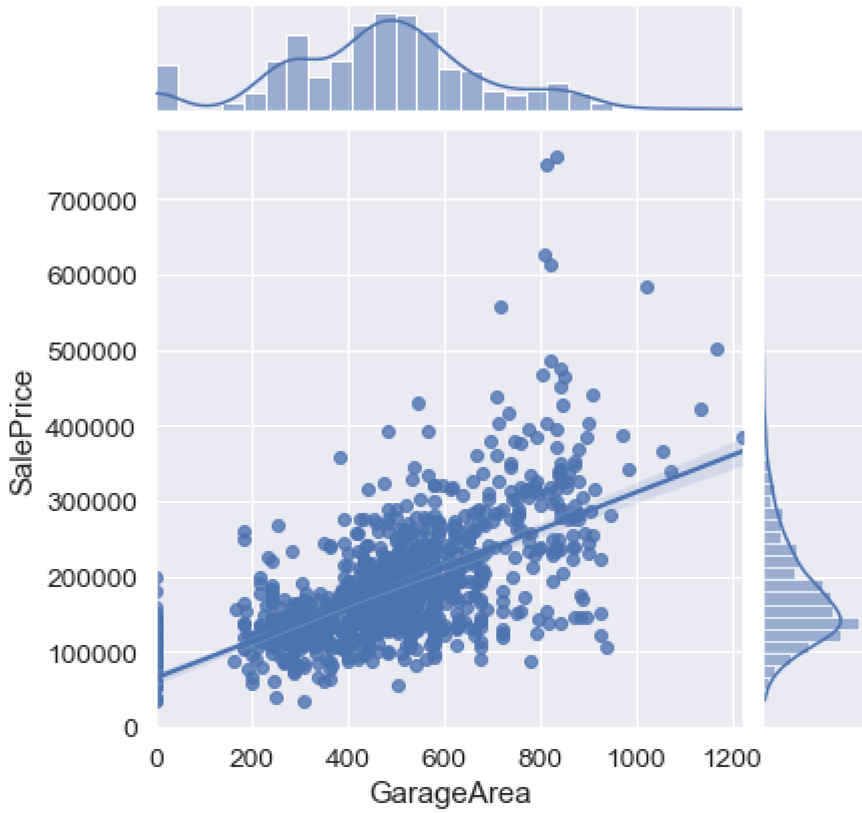


Again with the bottom two data-points. Let's remove those outliers.

In [30]:
```
# Removing outliers manually (More than 1000 sqft, less than $300k)
df= df.drop(df[(df['GarageArea']>1000)
                            & (df['SalePrice']<300000)].index).reset_index(drop
```

In [31]:
```
# Garage Area vs Sale Price
sns.jointplot(x=df['GarageArea'], y=df['SalePrice'], kind='reg')
```

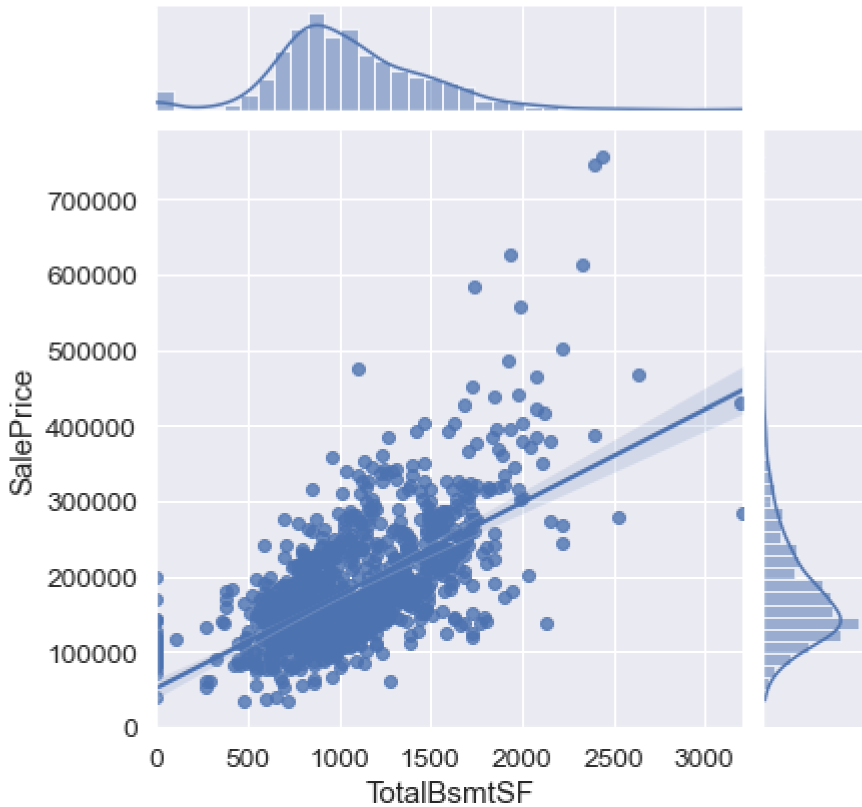
Out[31]: <seaborn.axisgrid.JointGrid at 0x10d4e508>



Only 0.01 point Pearson-R Score increase, but looks much better!

In [33]:
```
# Basement Area vs Sale Price
sns.jointplot(x=df['TotalBsmtSF'], y=df['SalePrice'], kind='reg')
```

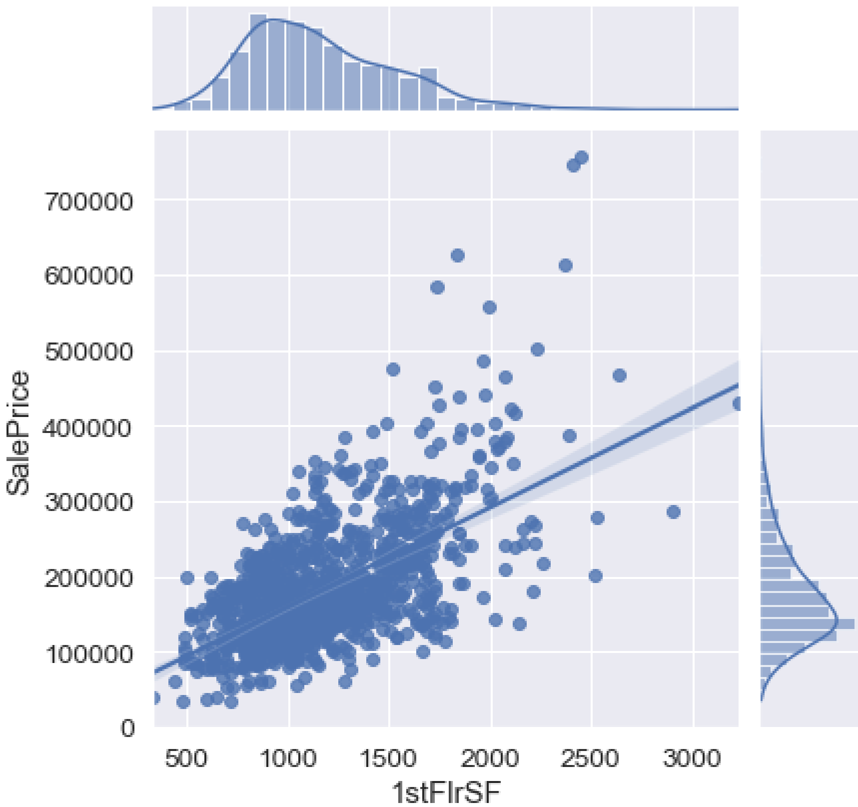
Out[33]: <seaborn.axisgrid.JointGrid at 0x110ab328>



Everything looks fine here.

In [34]:
```
# First Floor Area vs Sale Price
sns.jointplot(x=df['1stFlrSF'], y=df['SalePrice'], kind='reg')
```

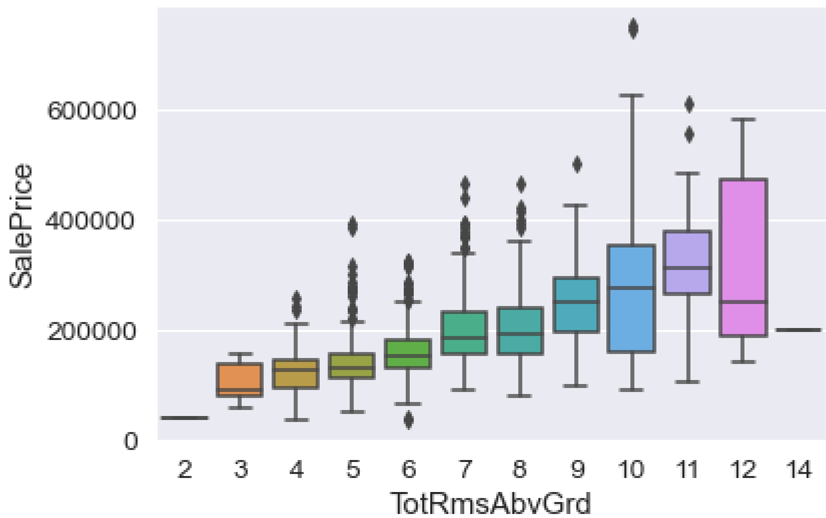
Out[34]: <seaborn.axisgrid.JointGrid at 0x10a79f28>



Looks good.

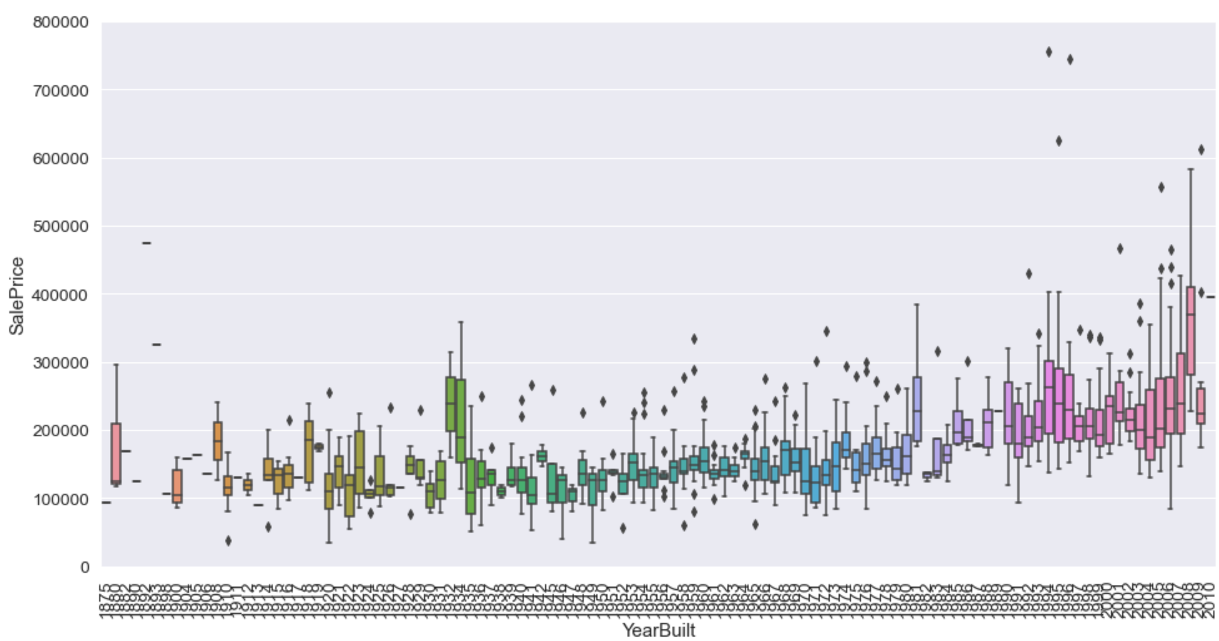
In [36]:
```python
# Total Rooms vs Sale Price
sns.boxplot(x=df['TotRmsAbvGrd'], y=df['SalePrice'])
```

Out[36]: <AxesSubplot:xlabel='TotRmsAbvGrd', ylabel='SalePrice'>



It seems like houses with more than 11 rooms come with a $100k off coupon. It looks like an outlier but I'll let it slide.

In [37]:
```python
# Total Rooms vs Sale Price
var = 'YearBuilt'
data = pd.concat([df['SalePrice'], df[var]], axis=1)
f, ax = plt.subplots(figsize=(16, 8))
fig = sns.boxplot(x=var, y="SalePrice", data=data)
fig.axis(ymin=0, ymax=800000);
plt.xticks(rotation=90);
```



Although it seems like house prices decrease with age, we can't be entirely sure. Is it because of inflation or stock market crashes? Let's leave the years alone.

. Impute Missing Data and Clean Data

Important questions when thinking about missing data:

How prevalent is the missing data? Is missing data random or does it have a pattern? The answer to these questions is important for practical reasons because missing data can imply a reduction of the sample size. This can prevent us from proceeding with the analysis. Moreover, from a substantive perspective, we need to ensure that the missing data process is not biased and hiding an inconvenient truth.

Let's combine both training and test data into one dataset to impute missing values and do some cleaning.

In [40]:

```
# Combining Datasets
ntrain = df.shape[0]
ntest = df_Test.shape[0]
y_train = df.SalePrice.values
all_data = pd.concat((df, df_Test)).reset_index(drop=True)
all_data.drop(['SalePrice'], axis=1, inplace=True)
print("Train data size is : {}".format(df.shape))
print("Test data size is : {}".format(df_Test.shape))
print("Combined dataset size is : {}".format(all_data.shape))
```

```
Train data size is : (1157, 81)
Test data size is : (292, 80)
Combined dataset size is : (1449, 80)
```

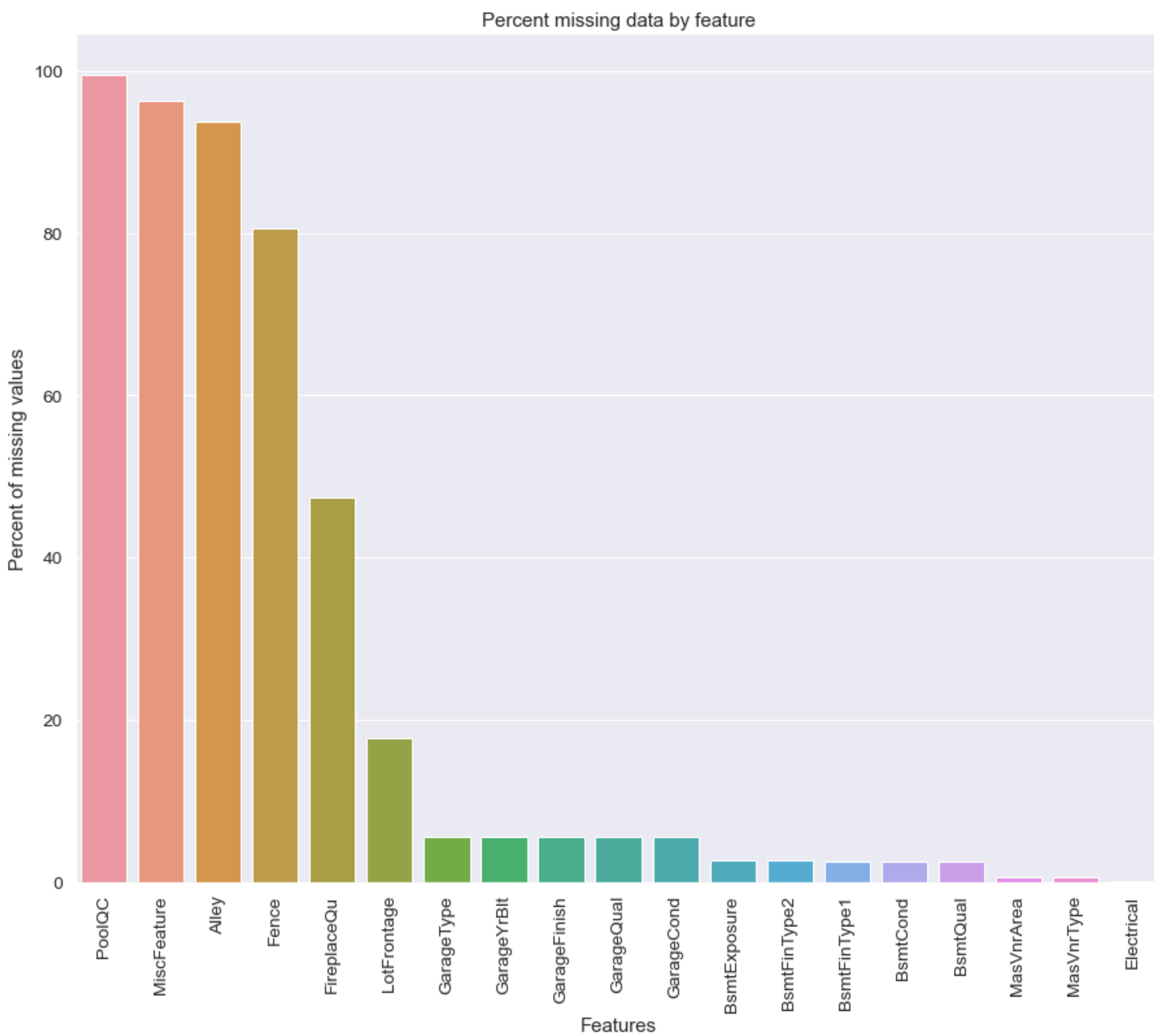
In [41]:
```
# Find Missing Ratio of Dataset
all_data_na = (all_data.isnull().sum() / len(all_data)) * 100
all_data_na = all_data_na.drop(all_data_na[all_data_na == 0].index).sort_val
missing_data = pd.DataFrame({'Missing Ratio' :all_data_na})
missing_data
```

Out[41]:

	Missing Ratio
PoolQC	99.585921
MiscFeature	96.342305
Alley	93.788820
Fence	80.676329
FireplaceQu	47.412008
LotFrontage	17.805383
GarageType	5.590062
GarageYrBlt	5.590062
GarageFinish	5.590062
GarageQual	5.590062
GarageCond	5.590062
BsmtExposure	2.622498
BsmtFinType2	2.622498
BsmtFinType1	2.553485
BsmtCond	2.553485
BsmtQual	2.553485
MasVnrArea	0.552105
MasVnrType	0.552105
Electrical	0.069013

In [42]:
```
# Percent missing data by feature
f, ax = plt.subplots(figsize=(15, 12))
plt.xticks(rotation='90')
sns.barplot(x=all_data_na.index, y=all_data_na)
plt.xlabel('Features', fontsize=15)
plt.ylabel('Percent of missing values', fontsize=15)
plt.title('Percent missing data by feature', fontsize=15)
```

Out[42]: Text(0.5, 1.0, 'Percent missing data by feature')



```
Imputing Missing Values
PoolQC : data description says NA means "No Pool"
MiscFeature : data description says NA means "no misc feature"
Alley : data description says NA means "no alley access"
Fence : data description says NA means "no fence"
FireplaceQu : data description says NA means "no fireplace"
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.
GarageType, GarageFinish, GarageQual and GarageCond : Replacing missing data with "None".
GarageYrBlt, GarageArea and GarageCars : Replacing missing data with 0.
BsmtFinSF1, BsmtFinSF2, BsmtUnfSF, TotalBsmtSF, BsmtFullBath and BsmtHalfBath: Replacing missing data with 0.
BsmtQual, BsmtCond, BsmtExposure, BsmtFinType1 and BsmtFinType2 : For all these categorical basement-related features, NaN means that there isn't a basement.
MasVnrArea and MasVnrType : NA most likely means no masonry veneer for these houses. We can fill 0 for the area and None for the type.
MSZoning (The general zoning classification) : 'RL' is by far the most common value. So we can fill in missing values with 'RL'.
Utilities : For this categorical feature all records are "AllPub", except for one "NoSeWa" and 2 NA . Since the house with 'NoSewa' is in the training set, this feature won't help in predictive modelling. We can then safely remove it.
Functional : data description says NA means typical.
Electrical : It has one NA value. Since this feature has mostly 'SBrkr', we can set that for the missing value.
KitchenQual: Only one NA value, and same as Electrical, we set 'TA' (which is the most frequent) for the missing value in KitchenQual.
Exterior1st and Exterior2nd : Both Exterior 1 & 2 have only one missing value. We will just substitute in the most common string
SaleType : Fill in again with most frequent which is "WD"
MSSubClass : Na most likely means No building class. We can replace missing values with None
```

In [43]:
```
all_data["PoolQC"] = all_data["PoolQC"].fillna("None")
all_data["MiscFeature"] = all_data["MiscFeature"].fillna("None")
all_data["Alley"] = all_data["Alley"].fillna("None")
all_data["Fence"] = all_data["Fence"].fillna("None")
all_data["FireplaceQu"] = all_data["FireplaceQu"].fillna("None")
all_data["LotFrontage"] = all_data.groupby("Neighborhood")["LotFrontage"].tr
for col in ('GarageType', 'GarageFinish', 'GarageQual', 'GarageCond'):
    all_data[col] = all_data[col].fillna('None')
for col in ('GarageYrBlt', 'GarageArea', 'GarageCars'):
    all_data[col] = all_data[col].fillna(0)
for col in ('BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF','TotalBsmtSF', 'BsmtFull
    all_data[col] = all_data[col].fillna(0)
for col in ('BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFin
    all_data[col] = all_data[col].fillna('None')
all_data["MasVnrType"] = all_data["MasVnrType"].fillna("None")
all_data["MasVnrArea"] = all_data["MasVnrArea"].fillna(0)
all_data['MSZoning'] = all_data['MSZoning'].fillna(all_data['MSZoning'].mode
all_data = all_data.drop(['Utilities'], axis=1)
all_data["Functional"] = all_data["Functional"].fillna("Typ")
all_data['Electrical'] = all_data['Electrical'].fillna(all_data['Electrical'
all_data['KitchenQual'] = all_data['KitchenQual'].fillna(all_data['KitchenQu
all_data['Exterior1st'] = all_data['Exterior1st'].fillna(all_data['Exterior1
all_data['Exterior2nd'] = all_data['Exterior2nd'].fillna(all_data['Exterior2
all_data['SaleType'] = all_data['SaleType'].fillna(all_data['SaleType'].mode
all_data['MSSubClass'] = all_data['MSSubClass'].fillna("None")
```

In [44]:
```
# Check if there are any missing values left
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_val
missing_data = pd.DataFrame({'Missing Ratio' :all_data_na})
missing_data.head()
```

Out[44]:

Missing Ratio

5. Feature Transformation/Engineering

In [ ]:
```
Let's take a look at some features that may be misinterpreted to represent s

MSSubClass: Identifies the type of dwelling involved in the sale.

20 1-STORY 1946 & NEWER ALL STYLES
30 1-STORY 1945 & OLDER
40 1-STORY W/FINISHED ATTIC ALL AGES
45 1-1/2 STORY - UNFINISHED ALL AGES
50 1-1/2 STORY FINISHED ALL AGES
60 2-STORY 1946 & NEWER
70 2-STORY 1945 & OLDER
75 2-1/2 STORY ALL AGES
80 SPLIT OR MULTI-LEVEL
85 SPLIT FOYER
90 DUPLEX - ALL STYLES AND AGES
120 1-STORY PUD (Planned Unit Development) - 1946 & NEWER
150 1-1/2 STORY PUD - ALL AGES
160 2-STORY PUD - 1946 & NEWER
180 PUD - MULTILEVEL - INCL SPLIT LEV/FOYER
190 2 FAMILY CONVERSION - ALL STYLES AND AGES
```

In [45]:
```
all_data['MSSubClass'].describe()
```

Out[45]:
```
count    1449.000000
mean       56.832298
std        42.277695
min        20.000000
25%        20.000000
50%        50.000000
75%        70.000000
max       190.000000
Name: MSSubClass, dtype: float64
```

So, the average is a 57 type. What does that mean? Is a 90 type 3 times better than a 30 type? This feature was interpreted as numerical when it is actually categorical. The types listed here are codes, not values. Thus, we need to feature transformation with this and many other features.

In [46]:
```
#MSSubClass =The building class
all_data['MSSubClass'] = all_data['MSSubClass'].apply(str)

#Changing OverallCond into a categorical variable
all_data['OverallCond'] = all_data['OverallCond'].astype(str)

#Year and month sold are transformed into categorical features.
all_data['YrSold'] = all_data['YrSold'].astype(str)
all_data['MoSold'] = all_data['MoSold'].astype(str)
```

```
Let's take a look at "Kitchen Quality".
```

In [47]:
```
all_data['KitchenQual'].unique()
```

Out[47]:
```
array(['TA', 'Gd', 'Ex', 'Fa'], dtype=object)
```

Here, data_description.txt comes to the rescue again!

Kitchen Quality:

Ex: Excellent Gd: Good TA: Typical/Average Fa: Fair Po: Poor Is a score of "Gd" better than "TA" but worse than "Ex"? I think so, let's encode these labels to give meaning to their specific orders.

In [48]:
```
from sklearn.preprocessing import LabelEncoder
cols = ('FireplaceQu', 'BsmtQual', 'BsmtCond', 'GarageQual', 'GarageCond',
        'ExterQual', 'ExterCond','HeatingQC', 'PoolQC', 'KitchenQual', 'Bsmt
        'BsmtFinType2', 'Functional', 'Fence', 'BsmtExposure', 'GarageFinish
        'LotShape', 'PavedDrive', 'Street', 'Alley', 'CentralAir', 'MSSubCla
        'YrSold', 'MoSold')
# Process columns and apply LabelEncoder to categorical features
for c in cols:
    lbl = LabelEncoder()
    lbl.fit(list(all_data[c].values))
    all_data[c] = lbl.transform(list(all_data[c].values))

# Check shape
print('Shape all_data: {}'.format(all_data.shape))
```

```
Shape all_data: (1449, 79)
```

Let's engineer one feature to combine square footage, this may be useful later on.

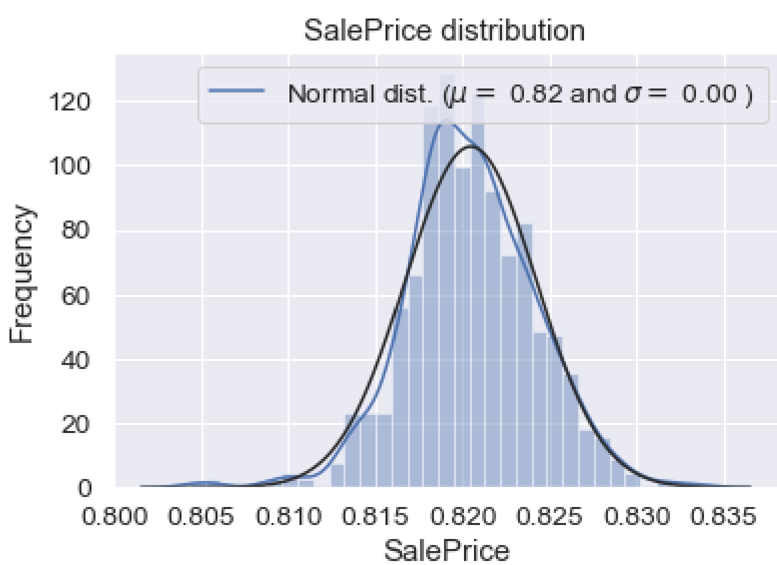
In [49]:
```
# Adding Total Square Feet feature
all_data['TotalSF'] = all_data['TotalBsmtSF'] + all_data['1stFlrSF'] + all_d
```

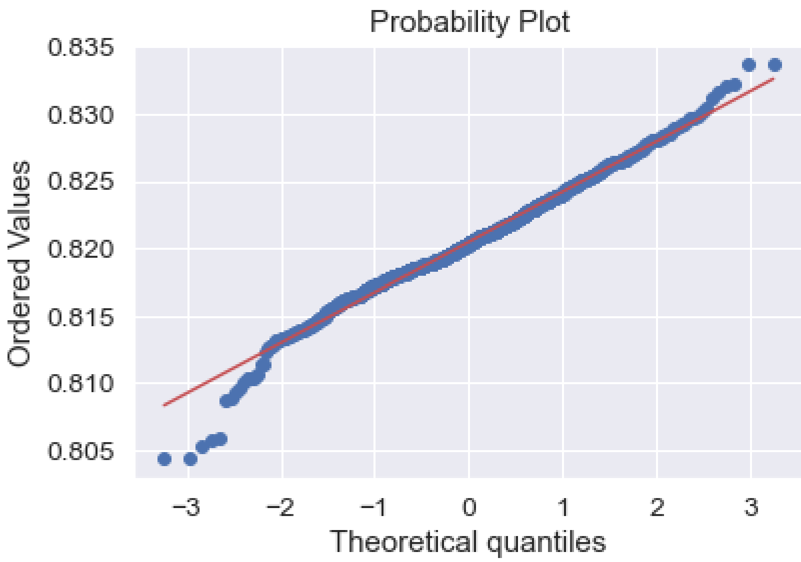
Fixing "skewed" features. Here, we fix all of the skewed data to be more normal so that our models will be more accurate when making predictions.

In [53]:

```python

from scipy import stats
# We use the numpy fuction log1p which  applies log(1+x) to all elements of
df["SalePrice"] = np.log1p(df["SalePrice"])

#Check the new distribution
sns.distplot(df['SalePrice'] , fit=norm);

# Get the fitted parameters used by the function
(mu, sigma) = norm.fit(df['SalePrice'])
print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))
plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu,
            loc='best')
plt.ylabel('Frequency')
plt.title('SalePrice distribution')

fig = plt.figure()
res = stats.probplot(df['SalePrice'], plot=plt)
plt.show()

y_train = df.SalePrice.values

print("Skewness: %f" % df['SalePrice'].skew())
print("Kurtosis: %f" % df['SalePrice'].kurt())
```

```
mu = 0.82 and sigma = 0.00
```






```
Skewness: -0.115359
Kurtosis: 1.199127
```

In [54]:

```python
numeric_feats = all_data.dtypes[all_data.dtypes != "object"].index

# Check the skew of all numerical features
skewed_feats = all_data[numeric_feats].apply(lambda x: skew(x.dropna())).sor
skewness = pd.DataFrame({'Skewed Features' :skewed_feats})
skewness.head()
```

Out[54]:

	Skewed Features
MiscVal	24.388024
PoolArea	15.882700
LotArea	12.595271
3SsnPorch	10.253854
LowQualFinSF	8.966866

In [55]:

```python
skewness = skewness[abs(skewness) > 0.75]
print("There are {} skewed numerical features to Box Cox transform".format(s

from scipy.special import boxcox1p
skewed_features = skewness.index
lam = 0.15
for feat in skewed_features:
    all_data[feat] = boxcox1p(all_data[feat], lam)
    all_data[feat] += 1
```

```
There are 60 skewed numerical features to Box Cox transform
```

In [56]:

```python
all_data = pd.get_dummies(all_data)
print(all_data.shape)
```

```
(1449, 221)
```

In [57]:
```
train = all_data[:ntrain]
test = all_data[ntrain:]
```

6. Modeling and Predictions

In [59]:
```
from sklearn.linear_model import ElasticNet, Lasso,  BayesianRidge, LassoLar
from sklearn.ensemble import RandomForestRegressor,  GradientBoostingRegress
from sklearn.kernel_ridge import KernelRidge
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import RobustScaler
from sklearn.base import BaseEstimator, TransformerMixin, RegressorMixin, cl
from sklearn.model_selection import KFold, cross_val_score, train_test_split
from sklearn.metrics import mean_squared_error

```

In [60]:
```
# Cross-validation with k-folds
n_folds = 5

def rmsle_cv(model):
    kf = KFold(n_folds, shuffle=True, random_state=42).get_n_splits(train.va
    rmse= np.sqrt(-cross_val_score(model, train.values, y_train, scoring="ne
    return(rmse)
```

For our models, we are going to use lasso, elastic net, kernel ridge, gradient boosting

In [61]:
```
lasso = make_pipeline(RobustScaler(), Lasso(alpha =0.0005, random_state=1))
ENet = make_pipeline(RobustScaler(), ElasticNet(alpha=0.0005, l1_ratio=.9, r
KRR = KernelRidge(alpha=0.6, kernel='polynomial', degree=2, coef0=2.5)
GBoost = GradientBoostingRegressor(n_estimators=3000, learning_rate=0.05,
                                   max_depth=4, max_features='sqrt',
                                   min_samples_leaf=15, min_samples_split=10
                                   loss='huber', random_state =5)
```

In [62]:

```
score = rmsle_cv(lasso)
print("\nLasso score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
score = rmsle_cv(ENet)
print("ElasticNet score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()
score = rmsle_cv(KRR)
print("Kernel Ridge score: {:.4f} ({:.4f})\n".format(score.mean(), score.std
score = rmsle_cv(GBoost)
print("Gradient Boosting score: {:.4f} ({:.4f})\n".format(score.mean(), scor

```

```
Lasso score: 0.0017 (0.0003)

ElasticNet score: 0.0017 (0.0003)

Kernel Ridge score: 0.0019 (0.0003)

Gradient Boosting score: 0.0012 (0.0002)
```

Here, we stack the models to average their scores.

In [63]:

```
class AveragingModels(BaseEstimator, RegressorMixin, TransformerMixin):
    def __init__(self, models):
        self.models = models

    # we define clones of the original models to fit the data in
    def fit(self, X, y):
        self.models_ = [clone(x) for x in self.models]

        # Train cloned base models
        for model in self.models_:
            model.fit(X, y)

        return self

    #Now we do the predictions for cloned models and average them
    def predict(self, X):
        predictions = np.column_stack([
            model.predict(X) for model in self.models_
        ])
        return np.mean(predictions, axis=1)
```

Here we average ENet, GBoost, KRR, and lasso

In [64]:

```
averaged_models = AveragingModels(models = (ENet, GBoost, KRR, lasso))

score = rmsle_cv(averaged_models)
print("Averaged base models score: {:.4f} ({:.4f})\n".format(score.mean(), s
```

```
Averaged base models score: 0.0013 (0.0002)
```

In [65]:
```
class StackingAveragedModels(BaseEstimator, RegressorMixin, TransformerMixin
    def __init__(self, base_models, meta_model, n_folds=5):
        self.base_models = base_models
        self.meta_model = meta_model
        self.n_folds = n_folds

    # We again fit the data on clones of the original models
    def fit(self, X, y):
        self.base_models_ = [list() for x in self.base_models]
        self.meta_model_ = clone(self.meta_model)
        kfold = KFold(n_splits=self.n_folds, shuffle=True)

        # Train cloned base models then create out-of-fold predictions
        # that are needed to train the cloned meta-model
        out_of_fold_predictions = np.zeros((X.shape[0], len(self.base_models
        for i, clf in enumerate(self.base_models):
            for train_index, holdout_index in kfold.split(X, y):
                instance = clone(clf)
                self.base_models_[i].append(instance)
                instance.fit(X[train_index], y[train_index])
                y_pred = instance.predict(X[holdout_index])
                out_of_fold_predictions[holdout_index, i] = y_pred

        # Now train the cloned  meta-model using the out-of-fold predictions
        self.meta_model_.fit(out_of_fold_predictions, y)
        return self

    def predict(self, X):
        meta_features = np.column_stack([
            np.column_stack([model.predict(X) for model in base_models]).mea
            for base_models in self.base_models_ ])
        return self.meta_model_.predict(meta_features)
```

Since our lasso model performed the best, we'll use it as a meta-model.

In [66]:
```
stacked_averaged_models = StackingAveragedModels(base_models = (ENet, GBoost
                                                 meta_model = lasso)

score = rmsle_cv(stacked_averaged_models)
print("Stacking Averaged models score: {:.4f} ({:.4f})".format(score.mean(),
```

```
Stacking Averaged models score: 0.0013 (0.0002)
```

In [67]:
```
def rmsle(y, y_pred):
    return np.sqrt(mean_squared_error(y, y_pred))
```

```
Stacked models
```

In [68]:

```
stacked_averaged_models.fit(train.values, y_train)
stacked_train_pred = stacked_averaged_models.predict(train.values)
stacked_pred = np.expm1(stacked_averaged_models.predict(test.values))
print(rmsle(y_train, stacked_train_pred))
```

```
0.0009936753449509559
```

```
Conclusion : Since our Lasso model performs well we accept this model
```

submission

In [71]:

```
sub = pd.DataFrame()
sub['Id'] = df_Test.Id
sub['SalePrice'] = stacked_pred
sub.to_csv('submission.csv',index=False)
```

In [ ]:

```

```