House2

January 2, 2019

1 Stacked Regressions to predict House Prices

1.1 Serigne

July 2017

If you use parts of this notebook in your scripts/notebooks, giving some kind of credit would be very much appreciated :) You can for instance link back to this notebook. Thanks!

This competition is very important to me as it helped me to begin my journey on Kaggle few months ago. I've read some great notebooks here. To name a few:

1. Comprehensive data exploration with Python by **Pedro Marcelino** : Great and very motivational data analysis

2. A study on Regression applied to the Ames dataset by **Julien Cohen-Solal** : Thorough features engeneering and deep dive into linear regression analysis but really easy to follow for beginners.

3. Regularized Linear Models by **Alexandru Papiu** : Great Starter kernel on modelling and Cross-validation

I can't recommend enough every beginner to go carefully through these kernels (and of course through many others great kernels) and get their first insights in data science and kaggle competitions.

After that (and some basic pratices) you should be more confident to go through this great script by **Human Analog** who did an impressive work on features engeneering.

As the dataset is particularly handy, I decided few days ago to get back in this competition and apply things I learnt so far, especially stacking models. For that purpose, we build two stacking classes (the simplest approach and a less simple one).

As these classes are written for general purpose, you can easily adapt them and/or extend them for your regression problems. The overall approach is hopefully concise and easy to follow..

The features engeneering is rather parsimonious (at least compared to some others great scripts) . It is pretty much :

- **Imputing missing values** by proceeding sequentially through the data
- **Transforming** some numerical variables that seem really categorical
- **Label Encoding** some categorical variables that may contain information in their ordering set

- **Box Cox Transformation** of skewed features (instead of log-transformation) : This gave me a **slightly better result** both on leaderboard and cross-validation.
- ** Getting dummy variables** for categorical features.

Then we choose many base models (mostly sklearn based models + sklearn API of DMLC's XGBoost and Microsoft's LightGBM), cross-validate them on the data before stacking/ensembling them. The key here is to make the (linear) models robust to outliers. This improved the result both on LB and cross-validation.

To my surprise, this does well on LB (0.11420 and top 4% the last time I tested it : **July 2, 2017**)

Hope that at the end of this notebook, stacking will be clear for those, like myself, who found the concept not so easy to grasp

```
In [1]: #import some necessary librairies

        import numpy as np # linear algebra
        import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
        %matplotlib inline
        import matplotlib.pyplot as plt  # Matlab-style plotting
        import seaborn as sns
        color = sns.color_palette()
        sns.set_style('darkgrid')
        import warnings
        def ignore_warn(*args, **kwargs):
            pass
        warnings.warn = ignore_warn #ignore annoying warning (from sklearn and seaborn)


        from scipy import stats
        from scipy.stats import norm, skew #for some statistics


        pd.set_option('display.float_format', lambda x: '{:.3f}'.format(x)) #Limiting floats o


        from subprocess import check_output
        print(check_output(["ls"]).decode("utf8")) #check the files available in the directory
```

```
all.zip
cas.ipynb
data_description.txt
House2.ipynb
House.ipynb
notebook.tex
output_16_1.png
output_23_1.png
output_24_1.png
output_30_1.png
```

```
output_33_1.png
output_6_1.png
ridge_sol.csv
sample_submission.csv
submission.csv
test.csv
train.csv
```

In [2]:
```python
#Now let's import and put the train and test datasets in  pandas dataframe

train = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')
```

In [3]:
```python
##display the first five rows of the train dataset.
train.head(5)
```

Out[3]:
```
   Id  MSSubClass MSZoning  LotFrontage  LotArea Street Alley LotShape  \
0   1          60       RL       65.000     8450   Pave   NaN      Reg
1   2          20       RL       80.000     9600   Pave   NaN      Reg
2   3          60       RL       68.000    11250   Pave   NaN      IR1
3   4          70       RL       60.000     9550   Pave   NaN      IR1
4   5          60       RL       84.000    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]
```

In [4]:
```python
##display the first five rows of the test dataset.
test.head(5)
```

Out[4]:
```
     Id  MSSubClass MSZoning  LotFrontage  LotArea Street Alley LotShape  \
0  1461          20       RH       80.000    11622   Pave   NaN      Reg
1  1462          20       RL       81.000    14267   Pave   NaN      IR1
2  1463          60       RL       74.000    13830   Pave   NaN      IR1
3  1464          60       RL       78.000     9978   Pave   NaN      IR1
```

```
4  1465          120        RL       43.000      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]
```

In [5]:
```
#check the numbers of samples and features
print("The train data size before dropping Id feature is : {} ".format(train.shape))
print("The test data size before dropping Id feature is : {} ".format(test.shape))

#Save the 'Id' column
train_ID = train['Id']
test_ID = test['Id']

#Now drop the  'Id' colum since it's unnecessary for  the prediction process.
train.drop("Id", axis = 1, inplace = True)
test.drop("Id", axis = 1, inplace = True)

#check again the data size after dropping the 'Id' variable
print("\nThe train data size after dropping Id feature is : {} ".format(train.shape))
print("The test data size after dropping Id feature is : {} ".format(test.shape))
```

```
The train data size before dropping Id feature is : (1460, 81)
The test data size before dropping Id feature is : (1459, 80)

The train data size after dropping Id feature is : (1460, 80)
The test data size after dropping Id feature is : (1459, 79)
```

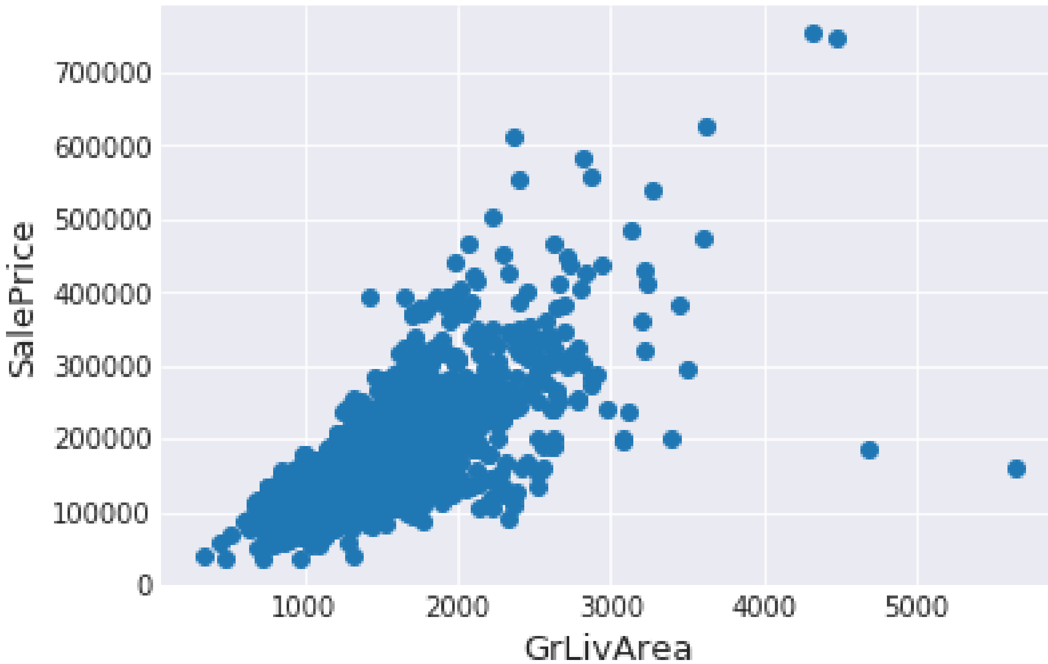
2 Data Processing

2.1 Outliers

[Documentation][1] for the Ames Housing Data indicates that there are outliers present in the training data [1]: http://ww2.amstat.org/publications/jse/v19n3/Decock/DataDocumentation.txt

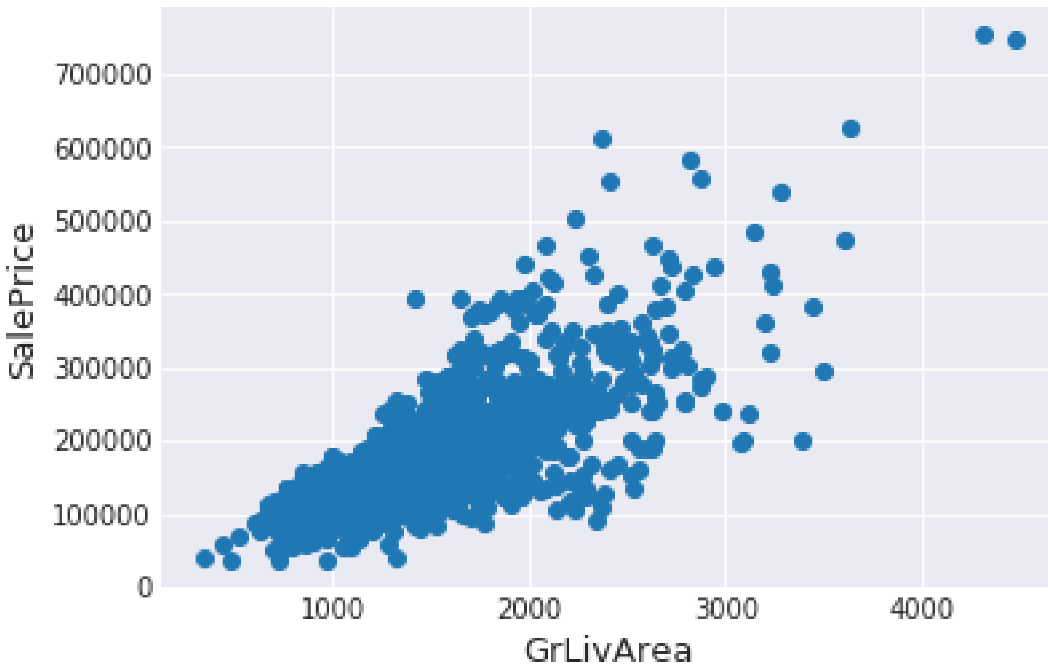
Let's explore these outliers

```
In [6]: fig, ax = plt.subplots()
        ax.scatter(x = train['GrLivArea'], y = train['SalePrice'])
        plt.ylabel('SalePrice', fontsize=13)
        plt.xlabel('GrLivArea', fontsize=13)
        plt.show()
```



We can see at the bottom right two with extremely large GrLivArea that are of a low price. These values are huge oultliers. Therefore, we can safely delete them.

```
In [7]: #Deleting outliers
        train = train.drop(train[(train['GrLivArea']>4000) & (train['SalePrice']<300000)].inde

        #Check the graphic again
        fig, ax = plt.subplots()
        ax.scatter(train['GrLivArea'], train['SalePrice'])
        plt.ylabel('SalePrice', fontsize=13)
        plt.xlabel('GrLivArea', fontsize=13)
        plt.show()
```



2.1.1 Note :

Outliers removal is note always safe. We decided to delete these two as they are very huge and really bad (extremely large areas for very low prices).

There are probably others outliers in the training data. However, removing all them may affect badly our models if ever there were also outliers in the test data. That's why , instead of removing them all, we will just manage to make some of our models robust on them. You can refer to the modelling part of this notebook for that.

2.2 Target Variable

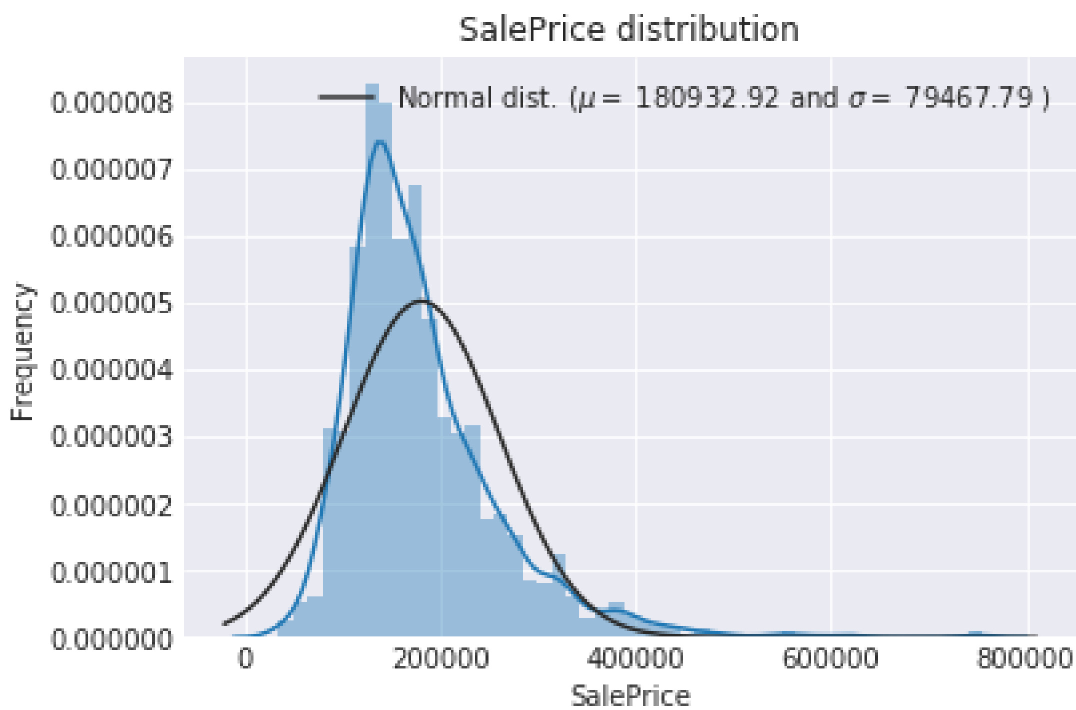
SalePrice is the variable we need to predict. So let's do some analysis on this variable first.

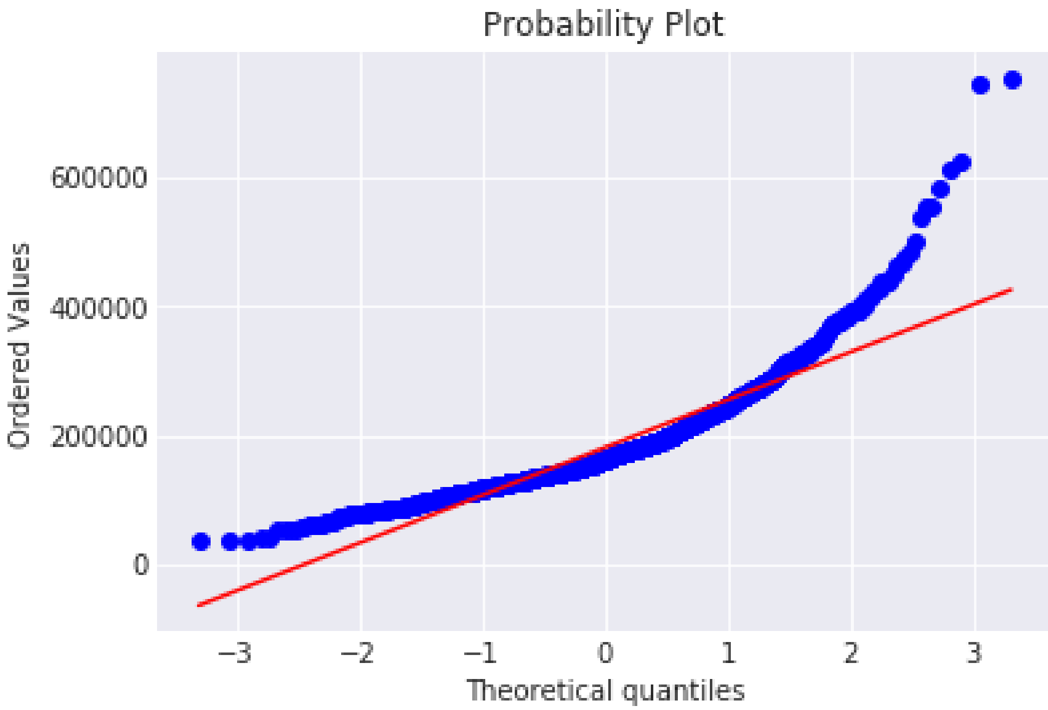
```
In [8]: sns.distplot(train['SalePrice'] , fit=norm);

        # Get the fitted parameters used by the function
        (mu, sigma) = norm.fit(train['SalePrice'])
        print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))

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

```
#Get also the QQ-plot
fig = plt.figure()
res = stats.probplot(train['SalePrice'], plot=plt)
plt.show()
```

```
/home/zy/anaconda3/lib/python3.6/site-packages/scipy/stats/stats.py:1713: FutureWarning: Using
  return np.add.reduce(sorted[indexer] * weights, axis=axis) / sumval
```

```
 mu = 180932.92 and sigma = 79467.79
```



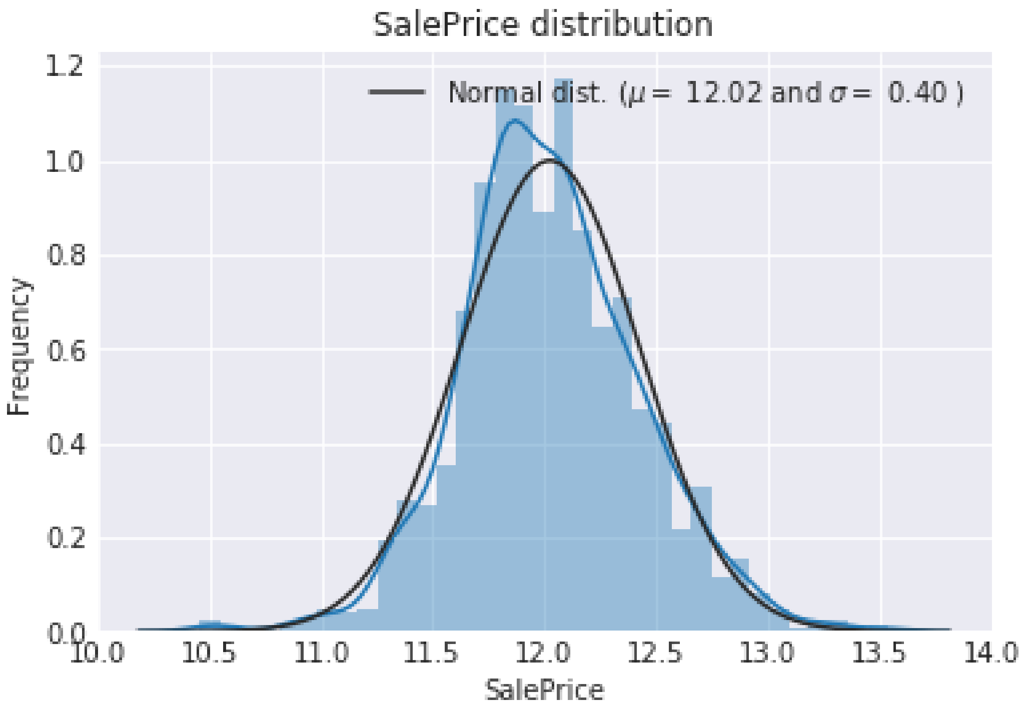


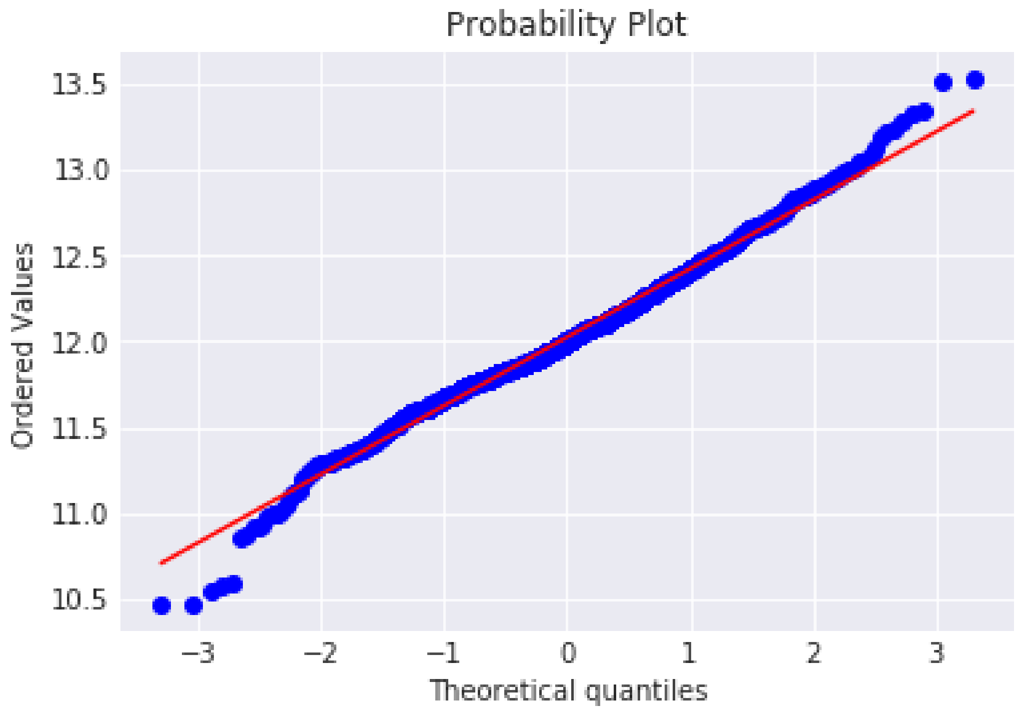
The target variable is right skewed. As (linear) models love normally distributed data , we need to transform this variable and make it more normally distributed.

Log-transformation of the target variable

```
In [9]: #We use the numpy fuction log1p which  applies log(1+x) to all elements of the column
        train["SalePrice"] = np.log1p(train["SalePrice"])

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

        # Get the fitted parameters used by the function
        (mu, sigma) = norm.fit(train['SalePrice'])
        print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))

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

        #Get also the QQ-plot
        fig = plt.figure()
        res = stats.probplot(train['SalePrice'], plot=plt)
        plt.show()
```

```
mu = 12.02 and sigma = 0.40
```





The skew seems now corrected and the data appears more normally distributed.

2.3 Features engineering

let's first concatenate the train and test data in the same dataframe

```
In [10]: ntrain = train.shape[0]
         ntest = test.shape[0]
         y_train = train.SalePrice.values
         all_data = pd.concat((train, test)).reset_index(drop=True)
         all_data.drop(['SalePrice'], axis=1, inplace=True)
         print("all_data size is : {}".format(all_data.shape))
```

```
all_data size is : (2917, 79)
```

2.3.1 Missing Data

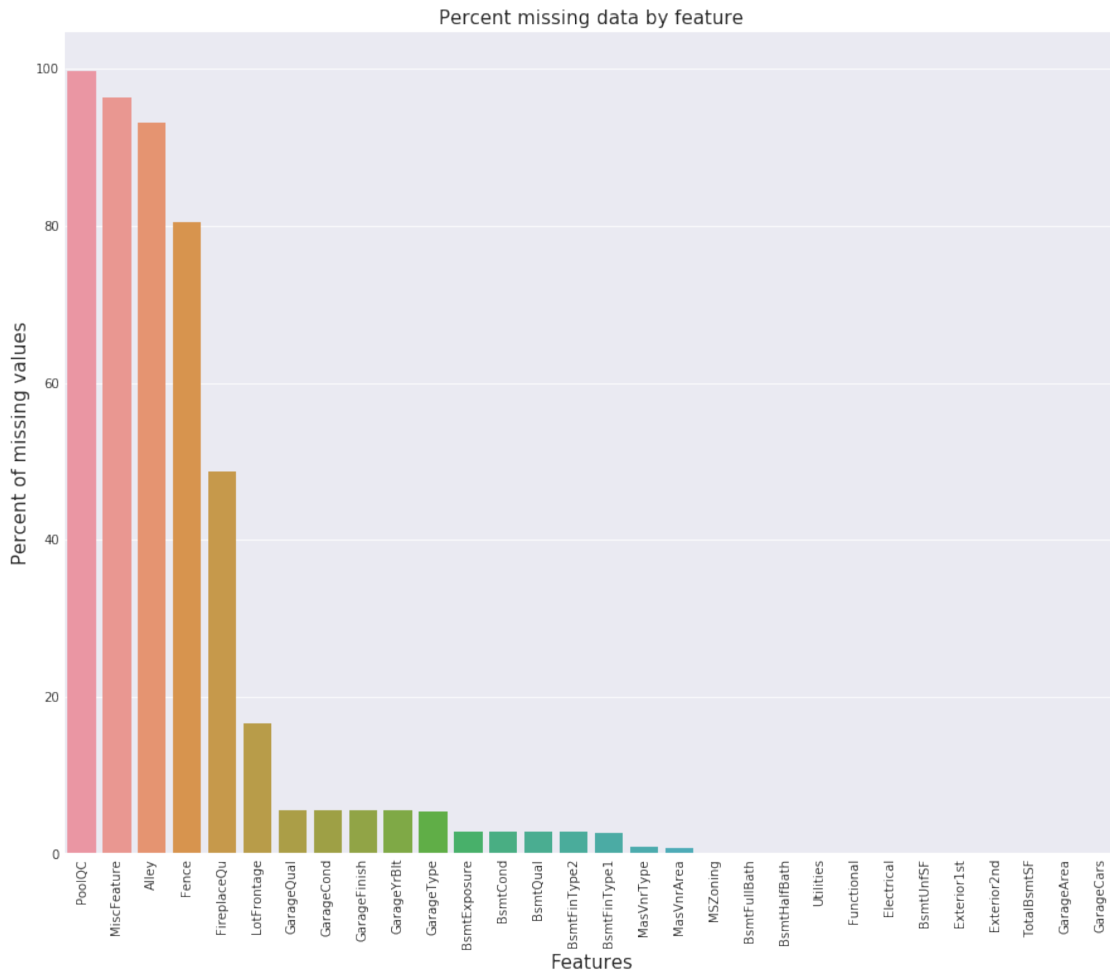
```
In [11]: all_data_na = (all_data.isnull().sum() / len(all_data)) * 100
         all_data_na = all_data_na.drop(all_data_na[all_data_na == 0].index).sort_values(ascend
         missing_data = pd.DataFrame({'Missing Ratio' :all_data_na})
         missing_data.head(20)
```

```
Out[11]:                Missing Ratio
         PoolQC                99.691
```

```
MiscFeature         96.400
Alley               93.212
Fence               80.425
FireplaceQu         48.680
LotFrontage         16.661
GarageQual           5.451
GarageCond           5.451
GarageFinish         5.451
GarageYrBlt          5.451
GarageType           5.382
BsmtExposure         2.811
BsmtCond             2.811
BsmtQual             2.777
BsmtFinType2         2.743
BsmtFinType1         2.708
MasVnrType           0.823
MasVnrArea           0.788
MSZoning             0.137
BsmtFullBath         0.069
```

In [12]:
```
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)
```

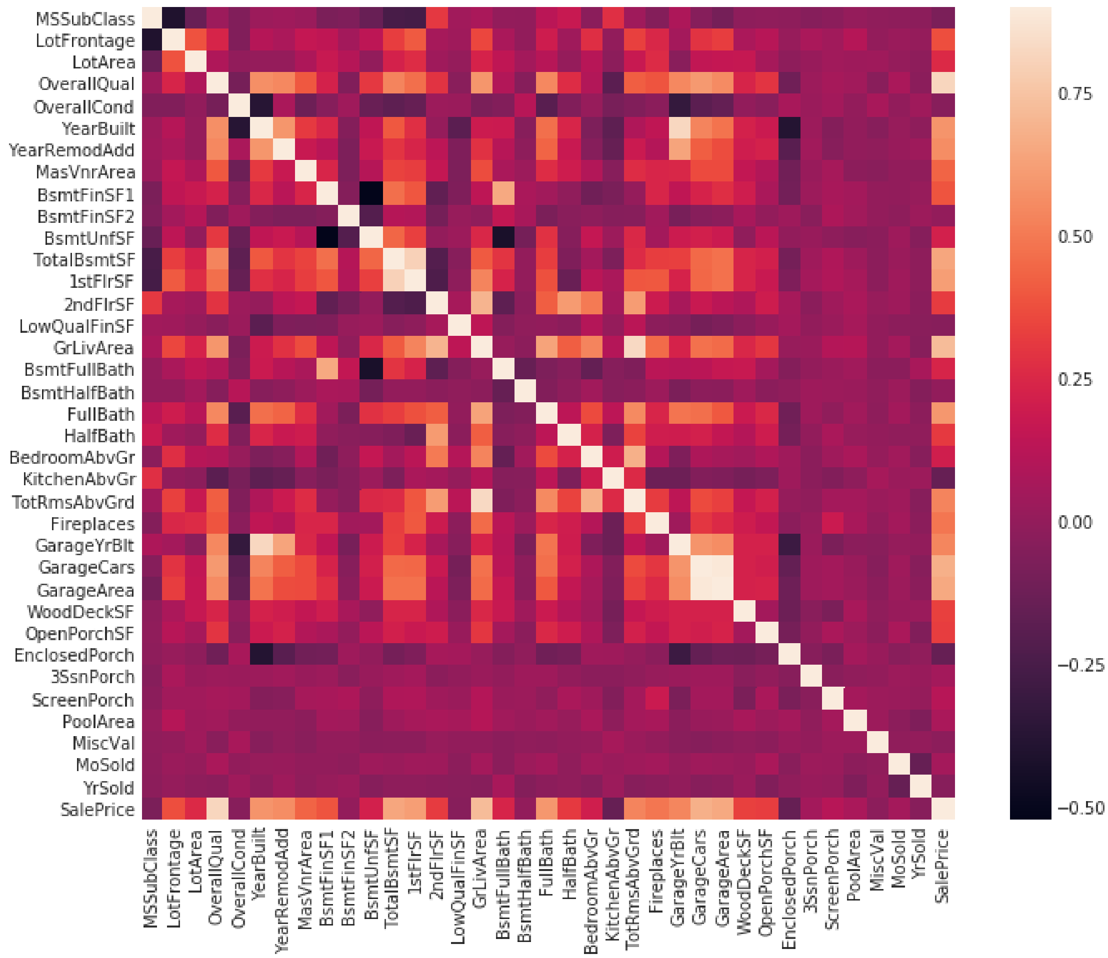
Out[12]:
```
Text(0.5,1,'Percent missing data by feature')
```



Data Correlation

In [13]:
```
#Correlation map to see how feaţtures are correlated with SalePrice
corrmat = train.corr()
plt.subplots(figsize=(12,9))
sns.heatmap(corrmat, vmax=0.9, square=True)
```

Out[13]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fdb8c384780>
```



###Imputing missing values
We impute them by proceeding sequentially through features with missing values

- **PoolQC** : data description says NA means "No Pool". That make sense, given the huge ratio of missing value (+99%) and majority of houses have no Pool at all in general.

```
In [14]: all_data["PoolQC"] = all_data["PoolQC"].fillna("None")
```

- **MiscFeature** : data description says NA means "no misc feature"

```
In [15]: all_data["MiscFeature"] = all_data["MiscFeature"].fillna("None")
```

- **Alley** : data description says NA means "no alley access"

```
In [16]: all_data["Alley"] = all_data["Alley"].fillna("None")
```

- **Fence** : data description says NA means "no fence"

```
In [17]: all_data["Fence"] = all_data["Fence"].fillna("None")
```

- **FireplaceQu** : data description says NA means "no fireplace"

```
In [18]: all_data["FireplaceQu"] = all_data["FireplaceQu"].fillna("None")
```

- **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**.

```
In [19]: #Group by neighborhood and fill in missing value by the median LotFrontage of all the
         all_data["LotFrontage"] = all_data.groupby("Neighborhood")["LotFrontage"].transform(
             lambda x: x.fillna(x.median()))
```

- **GarageType, GarageFinish, GarageQual and GarageCond** : Replacing missing data with None

```
In [20]: for col in ('GarageType', 'GarageFinish', 'GarageQual', 'GarageCond'):
             all_data[col] = all_data[col].fillna('None')
```

- **GarageYrBlt, GarageArea and GarageCars** : Replacing missing data with 0 (Since No garage = no cars in such garage.)

```
In [21]: for col in ('GarageYrBlt', 'GarageArea', 'GarageCars'):
             all_data[col] = all_data[col].fillna(0)
```

- **BsmtFinSF1, BsmtFinSF2, BsmtUnfSF, TotalBsmtSF, BsmtFullBath and BsmtHalfBath** : missing values are likely zero for having no basement

```
In [22]: for col in ('BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF','TotalBsmtSF', 'BsmtFullBath', 'Bs
             all_data[col] = all_data[col].fillna(0)
```

- **BsmtQual, BsmtCond, BsmtExposure, BsmtFinType1 and BsmtFinType2** : For all these categorical basement-related features, NaN means that there is no basement.

```
In [23]: for col in ('BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2'):
             all_data[col] = all_data[col].fillna('None')
```

- **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.

```
In [24]: all_data["MasVnrType"] = all_data["MasVnrType"].fillna("None")
         all_data["MasVnrArea"] = all_data["MasVnrArea"].fillna(0)
```

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

```
In [25]: all_data['MSZoning'] = all_data['MSZoning'].fillna(all_data['MSZoning'].mode()[0])
```

- **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.

```
In [26]: all_data = all_data.drop(['Utilities'], axis=1)
```

- **Functional** : data description says NA means typical

```
In [27]: all_data["Functional"] = all_data["Functional"].fillna("Typ")
```

- **Electrical** : It has one NA value. Since this feature has mostly 'SBrkr', we can set that for the missing value.

```
In [28]: all_data['Electrical'] = all_data['Electrical'].fillna(all_data['Electrical'].mode()[
```

- **KitchenQual**: Only one NA value, and same as Electrical, we set 'TA' (which is the most frequent) for the missing value in KitchenQual.

```
In [29]: all_data['KitchenQual'] = all_data['KitchenQual'].fillna(all_data['KitchenQual'].mode
```

- **Exterior1st and Exterior2nd** : Again Both Exterior 1 & 2 have only one missing value. We will just substitute in the most common string

```
In [30]: all_data['Exterior1st'] = all_data['Exterior1st'].fillna(all_data['Exterior1st'].mode
         all_data['Exterior2nd'] = all_data['Exterior2nd'].fillna(all_data['Exterior2nd'].mode
```

- **SaleType** : Fill in again with most frequent which is "WD"

```
In [31]: all_data['SaleType'] = all_data['SaleType'].fillna(all_data['SaleType'].mode()[0])
```

- **MSSubClass** : Na most likely means No building class. We can replace missing values with None

```
In [32]: all_data['MSSubClass'] = all_data['MSSubClass'].fillna("None")
```

Is there any remaining missing value ?

```
In [33]: #Check remaining missing values if any
         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(ascend
         missing_data = pd.DataFrame({'Missing Ratio' :all_data_na})
         missing_data.head()

Out[33]: Empty DataFrame
         Columns: [Missing Ratio]
         Index: []
```

It remains no missing value.
###More features engeneering
Transforming some numerical variables that are really categorical

```
In [34]: #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)
```

Label Encoding some categorical variables that may contain information in their ordering set

```
In [35]: from sklearn.preprocessing import LabelEncoder
         cols = ('FireplaceQu', 'BsmtQual', 'BsmtCond', 'GarageQual', 'GarageCond',
                 'ExterQual', 'ExterCond','HeatingQC', 'PoolQC', 'KitchenQual', 'BsmtFinType1'
                 'BsmtFinType2', 'Functional', 'Fence', 'BsmtExposure', 'GarageFinish', 'LandS
                 'LotShape', 'PavedDrive', 'Street', 'Alley', 'CentralAir', 'MSSubClass', 'Ove
                 'YrSold', 'MoSold')
         # process columns, 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))

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

Shape all_data: (2917, 78)
```

Adding one more important feature

Since area related features are very important to determine house prices, we add one more feature which is the total area of basement, first and second floor areas of each house

```
In [36]: # Adding total sqfootage feature
         all_data['TotalSF'] = all_data['TotalBsmtSF'] + all_data['1stFlrSF'] + all_data['2ndF
```

Skewed features

```
In [37]: 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())).sort_values(a
         print("\nSkew in numerical features: \n")
         skewness = pd.DataFrame({'Skew' :skewed_feats})
         skewness.head(10)
```

```
Skew in numerical features:
```

```
Out[37]:                  Skew
         MiscVal       21.940
         PoolArea      17.689
         LotArea       13.109
         LowQualFinSF  12.085
         3SsnPorch     11.372
         LandSlope      4.973
         KitchenAbvGr   4.301
         BsmtFinSF2     4.145
         EnclosedPorch  4.002
         ScreenPorch    3.945
```

Box Cox Transformation of (highly) skewed features

We use the scipy function boxcox1p which computes the Box-Cox transformation of $1 + x$.

Note that setting $\lambda = 0$ is equivalent to log1p used above for the target variable.

See [this page][1] for more details on Box Cox Transformation as well as [the scipy function's page][2] [1]: http://onlinestatbook.com/2/transformations/box-cox.html [2]: https://docs.scipy.org/doc/scipy-0.19.0/reference/generated/scipy.special.boxcox1p.html

```
In [38]: skewness = skewness[abs(skewness) > 0.75]
         print("There are {} skewed numerical features to Box Cox transform".format(skewness.sl

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

         #all_data[skewed_features] = np.log1p(all_data[skewed_features])
```

```
There are 59 skewed numerical features to Box Cox transform
```

Getting dummy categorical features

```
In [39]: all_data = pd.get_dummies(all_data)
         print(all_data.shape)
```

```
(2917, 220)
```

Getting the new train and test sets.

In [40]:
```
train = all_data[:ntrain]
test = all_data[ntrain:]
```

#Modelling

Import librairies

In [41]:
```
from sklearn.linear_model import ElasticNet, Lasso,  BayesianRidge, LassoLarsIC
from sklearn.ensemble import RandomForestRegressor,  GradientBoostingRegressor
from sklearn.kernel_ridge import KernelRidge
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import RobustScaler
from sklearn.base import BaseEstimator, TransformerMixin, RegressorMixin, clone
from sklearn.model_selection import KFold, cross_val_score, train_test_split
from sklearn.metrics import mean_squared_error
import xgboost as xgb
import lightgbm as lgb
```

In [42]:
```
!pip install --user lightgbm
```

```
Requirement already satisfied: lightgbm in /home/zy/.local/lib/python3.6/site-packages (2.2.2)
Requirement already satisfied: scikit-learn in /home/zy/.local/lib/python3.6/site-packages (fr
Requirement already satisfied: scipy in /usr/local/lib/python3.6/dist-packages (from lightgbm)
Requirement already satisfied: numpy in /home/zy/.local/lib/python3.6/site-packages (from ligh
```

Define a cross validation strategy

We use the **cross_val_score** function of Sklearn. However this function has not a shuffle attribut, we add then one line of code, in order to shuffle the dataset prior to cross-validation

In [43]:
```
#Validation function
n_folds = 5

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

##Base models

- **LASSO Regression** :

This model may be very sensitive to outliers. So we need to made it more robust on them. For that we use the sklearn's **Robustscaler()** method on pipeline

In [44]:
```
lasso = make_pipeline(RobustScaler(), Lasso(alpha =0.0005, random_state=1))
```

- **Elastic Net Regression** :

again made robust to outliers

```
In [45]: ENet = make_pipeline(RobustScaler(), ElasticNet(alpha=0.0005, l1_ratio=.9, random_sta
```

- **Kernel Ridge Regression** :

```
In [46]: KRR = KernelRidge(alpha=0.6, kernel='polynomial', degree=2, coef0=2.5)
```

- **Gradient Boosting Regression** :

With **huber** loss that makes it robust to outliers

```
In [47]: 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)
```

- **XGBoost** :

```
In [48]: model_xgb = xgb.XGBRegressor(colsample_bytree=0.4603, gamma=0.0468,
                                      learning_rate=0.05, max_depth=3,
                                      min_child_weight=1.7817, n_estimators=2200,
                                      reg_alpha=0.4640, reg_lambda=0.8571,
                                      subsample=0.5213, silent=1,
                                      random_state =7, nthread = -1)
```

- **LightGBM** :

```
In [49]: model_lgb = lgb.LGBMRegressor(objective='regression',num_leaves=5,
                                       learning_rate=0.05, n_estimators=720,
                                       max_bin = 55, bagging_fraction = 0.8,
                                       bagging_freq = 5, feature_fraction = 0.2319,
                                       feature_fraction_seed=9, bagging_seed=9,
                                       min_data_in_leaf =6, min_sum_hessian_in_leaf = 11)
```

###Base models scores
Let's see how these base models perform on the data by evaluating the cross-validation rmsle error

```
In [50]: score = rmsle_cv(lasso)
         print("\nLasso score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
Lasso score: 0.1115 (0.0074)
```

```
In [51]: score = rmsle_cv(ENet)
         print("ElasticNet score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
ElasticNet score: 0.1116 (0.0074)
```

In [52]:
```
score = rmsle_cv(KRR)
print("Kernel Ridge score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
Kernel Ridge score: 0.1153 (0.0075)
```

In [53]:
```
score = rmsle_cv(GBoost)
print("Gradient Boosting score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
Gradient Boosting score: 0.1177 (0.0080)
```

In [54]:
```
score = rmsle_cv(model_xgb)
print("Xgboost score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
```

```
Xgboost score: 0.1162 (0.0078)
```

In [55]:
```
score = rmsle_cv(model_lgb)
print("LGBM score: {:.4f} ({:.4f})\n" .format(score.mean(), score.std()))
```

```
LGBM score: 0.1162 (0.0071)
```

##Stacking models

###Simplest Stacking approach : Averaging base models

We begin with this simple approach of averaging base models. We build a new **class** to extend scikit-learn with our model and also to laverage encapsulation and code reuse (inheritance)

Averaged base models class

In [56]:
```
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
```