Projet 3A, House Prices Competition

Mathieu Barré - Skander Jemaa - Kevin Riera

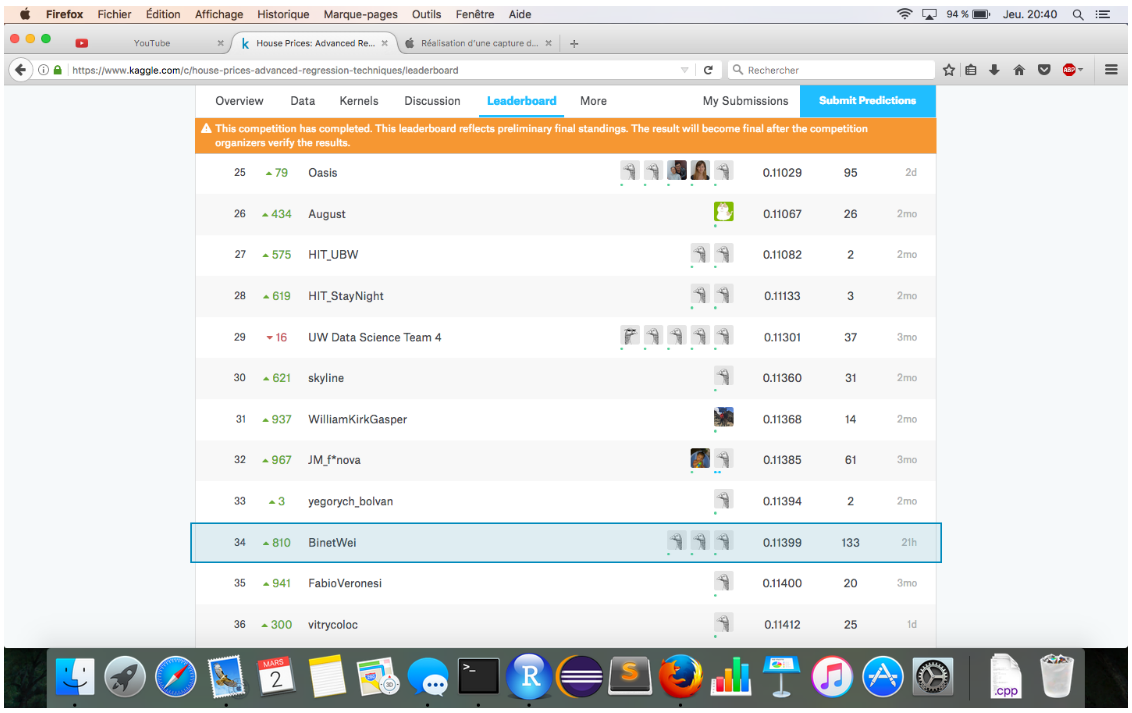
March 10, 2017

1 Introduction

This document is the final report of our Data Science Projet 3A.
We participated in a Kaggle competition named **House Prices: Advanced Regression Techniques**. The competition started in September 2016 and ended in March 2016 in the "Playground" category, it became then a "Getting started" challenge and has been extended for 3 years. The idea is to predict the price of a house using various attributes. Here is Kaggle's description of the competition: *"Ask a home buyer to describe their dream house, and they probably won't begin with the height of the basement ceiling or the proximity to an east-west railroad. But this playground competition's dataset proves that much more influences price negotiations than the number of bedrooms or a white-picket fence. With 79 explanatory variables describing (almost) every aspect of residential homes in Ames, Iowa, this competition challenges you to predict the final price of each home. "* --- Kaggle website
Attending this competition we agreed on the fact that the success of this project would depend on three goals/criterias
Objective 1: Concretly learn and be able to compute machine learning techniques
As a consequence we will try to present lots of techniques, even if they are not used in our final code. We will also try to insist on some theorical particularities and nuances of the different algorithms. Each time we used an new algorithm we tried to implement a simple version of it to really seize the idea behind it. Then we used the ones from optimized library as sklearn of xgboost for our final codes.
Objective 2: Compete
The "challenge" aspect of the project is one of the thing that motivated us the most. Nevertheless, competition is fierce. At the end of the challenge there were almost 5000 teams enrolled. We managed to reach the rank of 34 in the private leaderboard.

```
In [51]: from IPython.display import Image
         Image("private_lb.png")
```

```
Out[51]:
```


Firefox Fichier Édition Affichage Historique Marque-pages Outils Fenêtre Aide
House Prices: Advanced Re...
https://www.kaggle.com/c/house-prices-advanced-regression-techniques/leaderboard
Overview Data Kernels Discussion Leaderboard More My Submissions Submit Predictions

This competition has completed. This leaderboard reflects preliminary final standings. The result will become final after the competition organizers verify the results.

25	▲79	Oasis	0.11029	95	2d
26	▲434	August	0.11067	26	2mo
27	▲575	HIT_UBW	0.11082	2	2mo
28	▲619	HIT_StayNight	0.11133	3	2mo
29	▼16	UW Data Science Team 4	0.11301	37	3mo
30	▲621	skyline	0.11360	31	2mo
31	▲937	WilliamKirkGasper	0.11368	14	2mo
32	▲967	JM_f*nova	0.11385	61	3mo
33	▲3	yegorych_bolvan	0.11394	2	2mo
34	▲810	BinetWei	0.11399	133	21h
35	▲941	FabioVeronesi	0.11400	20	3mo
36	▲300	vitrycoloc	0.11412	25	1d

Objective 3: Show and present the results
We are actually speaking of this report. Being able to present nicely some results is sometimes even more important than the results themselves. This project is also the occasion to train our skills on R or Jupyter, to be able to compute clear graphs and bring out the structure and the ideas of our code in the most visual and intuitive way.

2 Data exploration and analysis

In this first part we will simply try to present the data we are dealing with. It is obviously very important to have an idea of the aspect of the different distributions and to see how some variables may be correlated with our target or not.

```
In [40]: from __future__ import division, print_function
         import os
         import sys
         import glob
         import pandas as pd
         import numpy as np
         import matplotlib.pyplot as plt
         try :
             import seaborn as sns; sns.set()
         except ImportError:
             print("seaborn not found on your computer. "
                   "Install it if you want pretty charts \n"
                   "If you have internet access you can run in a cell : \n"
```

```
              "!pip install -U seaborn ")

        pd.set_option('display.max_columns', None)
        %matplotlib inline
        import warnings;
        warnings.simplefilter('ignore')
```

2.1 Features distributions

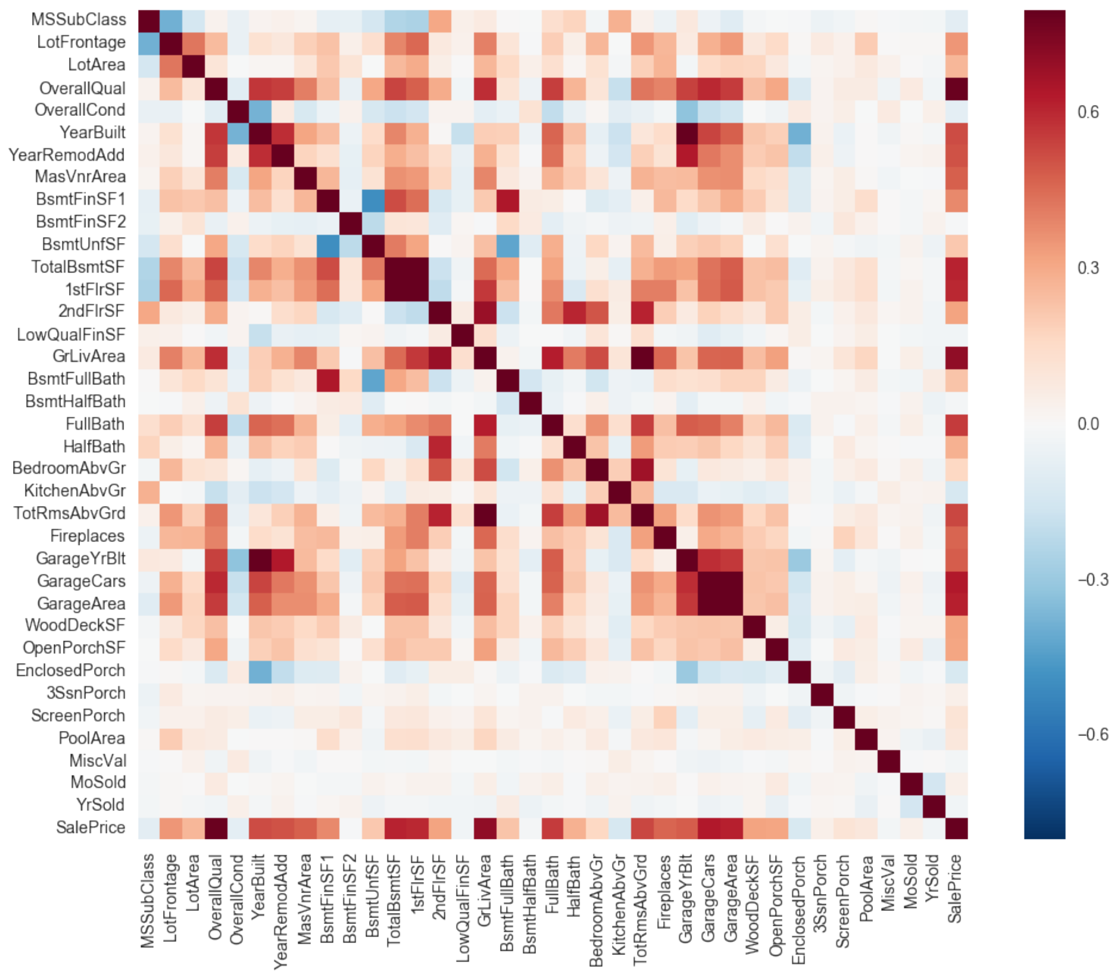
2.1.1 Replacing values

At the beginning, we first looked at the features distributions and their effect on the price. The `data description`indicates that the `train` dataset contains a lot of values referenced as missing while the `NA` state refers to the absence of the considered feature. We replaced these values by `N.a` to avoid their classification as missing values and to create another class in the considered variables.

```
In [41]: train = pd.read_csv("train.csv",index_col = 0)
         train.loc[train['PoolQC'].isnull(),'PoolQC'] = 'N.a'
         train.loc[train['Fence'].isnull(),'Fence'] = 'N.a'
         train.loc[train['MiscFeature'].isnull(),'MiscFeature'] = 'N.a'
         train.loc[train['GarageCond'].isnull(),'GarageCond'] = 'N.a'
         train.loc[train['GarageQual'].isnull(),'GarageQual'] = 'N.a'
         train.loc[train['GarageFinish'].isnull(),'GarageFinish'] = 'N.a'
         train.loc[train['GarageType'].isnull(),'GarageType'] = 'N.a'
         train.loc[train['FireplaceQu'].isnull(),'FireplaceQu'] = 'N.a'
         train.loc[train['BsmtFinType2'].isnull(),'BsmtFinType2'] = 'N.a'
         train.loc[train['BsmtFinType1'].isnull(),'BsmtFinType1'] = 'N.a'
         train.loc[train['BsmtExposure'].isnull(),'BsmtExposure'] = 'N.a'
         train.loc[train['BsmtCond'].isnull(),'BsmtCond'] = 'N.a'
         train.loc[train['BsmtQual'].isnull(),'BsmtQual'] = 'N.a'
         train.loc[train['Alley'].isnull(),'Alley'] = 'N.a'

         y_tr = train[['SalePrice']]
         X_tr = train.drop('SalePrice',axis = 1)
```

Our first goal is to detect correlations between `SalePrice` and the other variables. We have drawn the correlation matrix to do so.

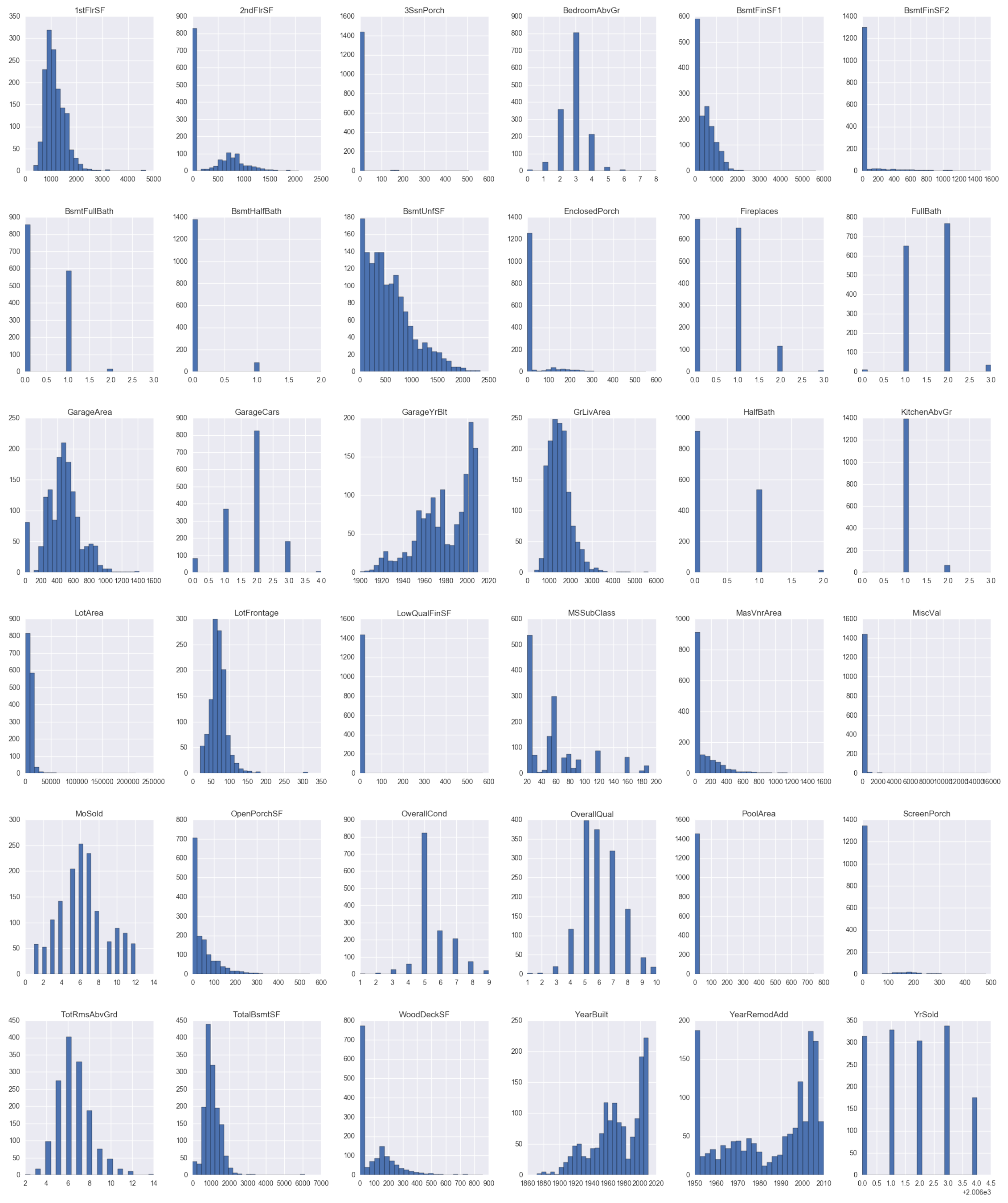
```
In [50]: corrmat = train.corr()
         f, ax = plt.subplots(figsize=(12, 9))
         sns.heatmap(corrmat, vmax=.8, square=True);
```



```
In [44]: names = corrmat.columns
         order = np.argsort(np.abs(corrmat['SalePrice']))
         names[order[26:36]]

Out[44]: Index([u'YearRemodAdd', u'YearBuilt', u'TotRmsAbvGrd', u'FullBath',
                u'1stFlrSF', u'TotalBsmtSF', u'GarageArea', u'GarageCars', u'GrLivArea',
                u'OverallQual'],
               dtype='object')
```

The correlation matrix shows the importances of the overall quality of materials, the space and the novelty of the house. We did not restrain to this analysis as we tried to see whether other variables had an effect on the `SalePrice` mainly categorical ones. Still, our first step was to visualize the quantitative variables distributions.

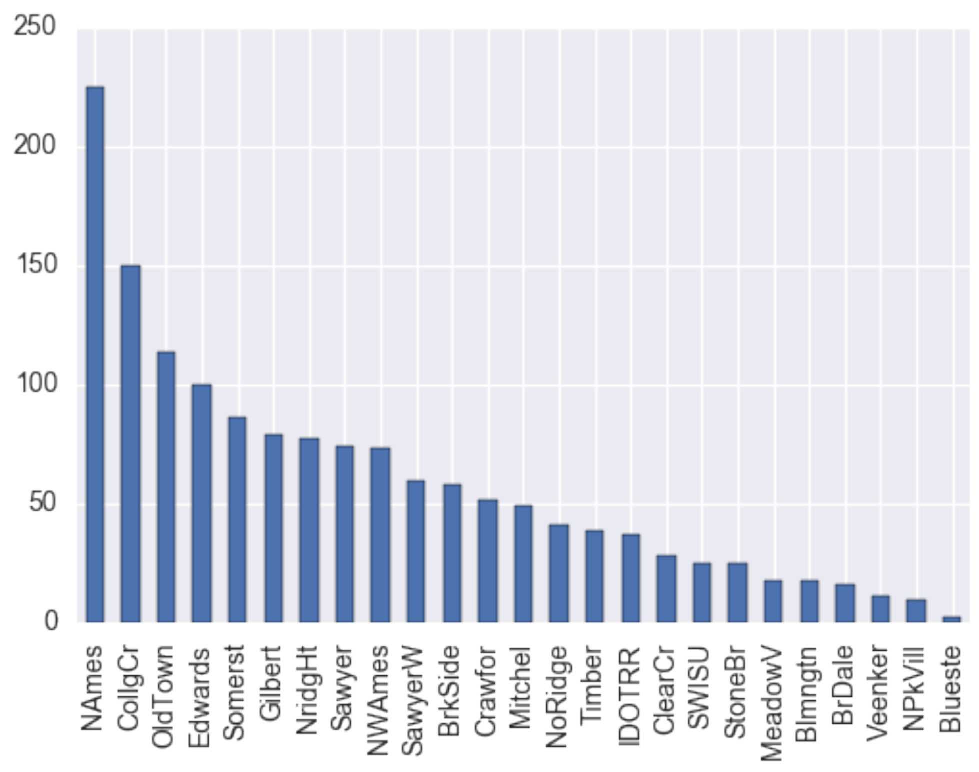
```
In [3]: X_tr.hist(figsize=(25, 30), bins=25);
```



These histograms show us that many variables do not have a centred and symetric distribution. At first glance, we can see for example that `GarageYrBlt` does not have a gaussian distribution. In order to symetrize such skewed distributions we use the `skew` function of `scipy.stat`, if the skewness is larger than 0.75, we take the logarithm of the variable and consider a Linear-log model for these variables.

2.1.2 Neighborhood influence over the SalePrice

```
In [4]: X_tr.Neighborhood.value_counts().plot(kind='bar');
```

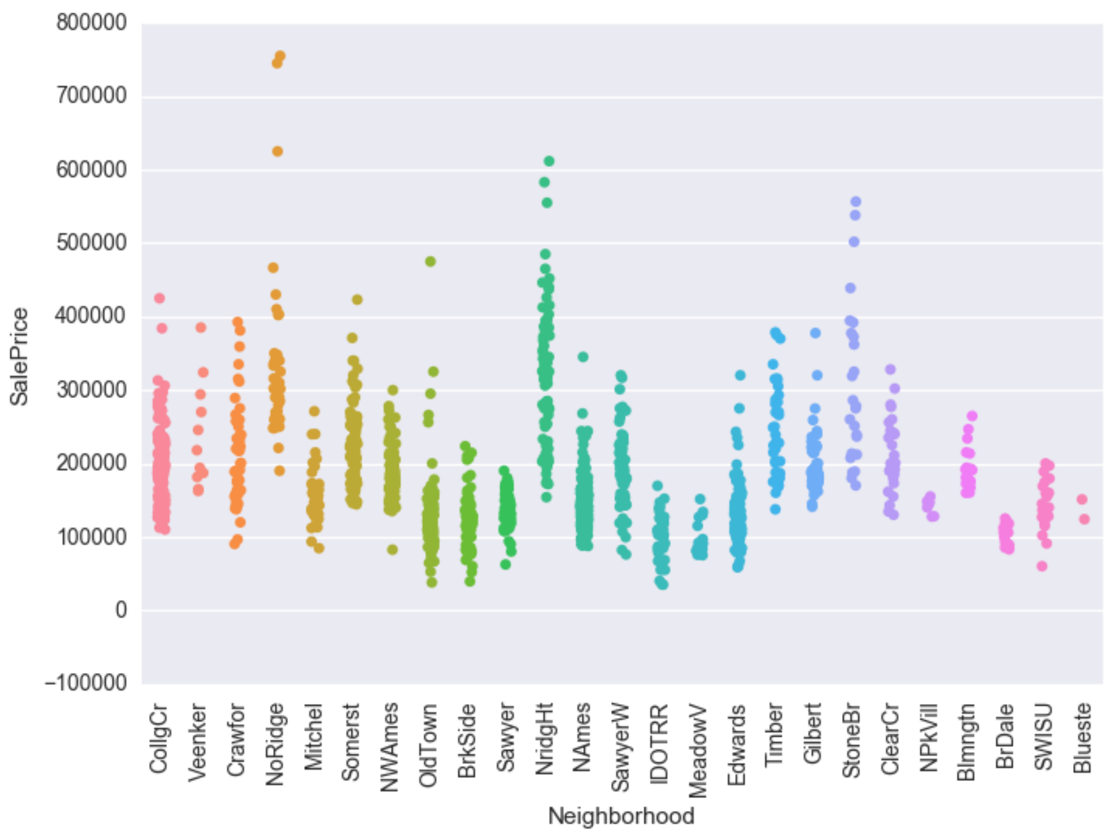


We looked at the distribution of sales by Neighborhood. This histogram showed us two main facts :

- a small number of sales some neighborhoods : this fact made it difficult to use mean variables over the SalePrice by Neighborhood. Indeed we can see that for Blueste the number of sales is so small that it would not be reasonable to take into consideration the mean price for this neighborhood and furthermore a rolling mean over time or a mean adjusted by the Area
- a large difference between the number of samples of each Neighborhood : this difference may engender an obligation to keep the Neighborhood variable if the SalePrice distribution is not the same for every neighborhood which is hard to determine regarding to the large number of neighborhoods in comparison to the number of samples of the training set.

The next step was to look at the prices by Neighborhoods.

```
In [5]: import seaborn as sns
        sns.set(style="darkgrid", color_codes=True)
        sns.stripplot(x="Neighborhood", y="SalePrice",\
                      data=train.loc[:,['Neighborhood','SalePrice']],\
                      jitter = True)
        plt.xticks(rotation=90);
```



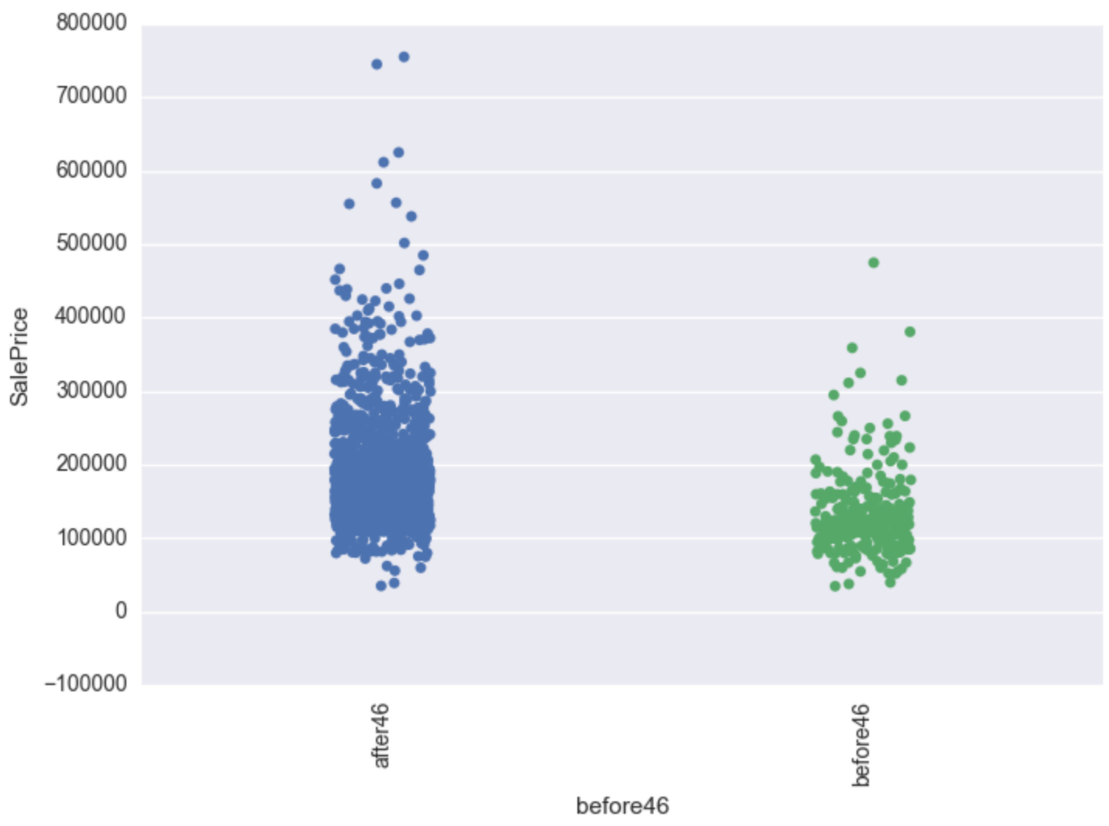
The strip plot shows that the minimum and maximum `Sale prices` vary with neighborhoods. Moreover we can observe a large difference in the prices ranges.

2.1.3 Prices distributions according to the construction date

We have also looked at the differences between old houses and newer ones by creating a categorical variable that indictates wether or not the house has been built before a given year.
After trying multiple values, we decided to take 1946 as a reference as it is already used in the `MSSubClass` variable to identify the type of dwelling and gives the largest price differences.

```
In [7]: train['before46'] = 'after46'
        train.loc[train['YearBuilt']<1946,'before46'] = 'before46'
```

```
In [8]: sns.set(style="darkgrid", color_codes=True)
        sns.stripplot(x="before46", y="SalePrice",\
                      data=train.loc[:,['before46','SalePrice']],\
                      jitter = True)
        plt.xticks(rotation=90);
```

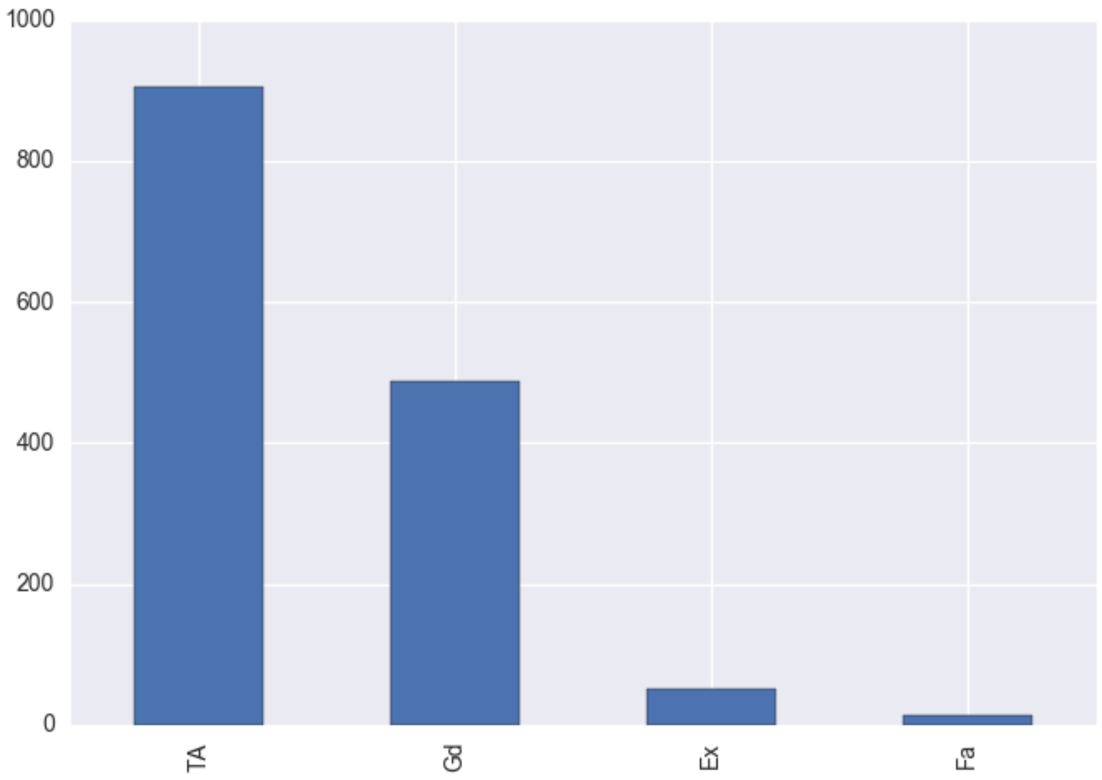


The plot and the statistics show a big difference between the prices of houses built before and after 1946. Therefore adding this variable might increase the regressor's accuracy.

2.1.4 Price distribution conditionally to Exterior features

It seems logical that the exterior condtion and the quality of the materials used on the outter side of the house may have an impact on the house price. Thus, we looked at both the distributions of `ExterQual` and `Exterior1st` and to `Sale Price` conditionally to these two features.

```
In [6]: X_tr.ExterQual.value_counts().plot(kind='bar');
```



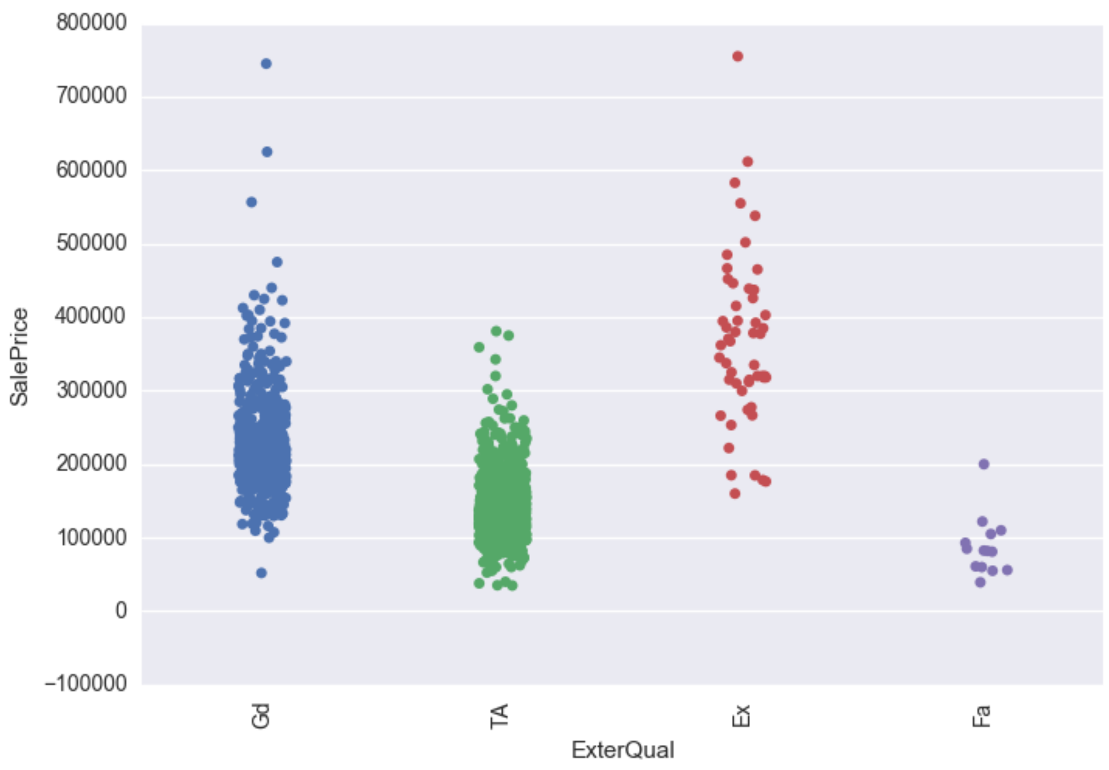
```
In [9]: print("Summary statstics for houses with excellent Exterior Quality",'\n')
        print(train.loc[train['ExterQual']=='Ex','SalePrice'].describe(),'\n')
        print("Summary statstics for houses with good Exterior Quality",'\n')
        print(train.loc[train['ExterQual']=='Gd','SalePrice'].describe(),'\n')
        print("Summary statstics for houses with average Exterior Quality",'\n')
        print(train.loc[train['ExterQual']=='TA','SalePrice'].describe(),'\n')
        print("Summary statstics for houses with fair Exterior Quality",'\n')
        print(train.loc[train['ExterQual']=='Fa','SalePrice'].describe(),'\n')
```

```
Summary statstics for houses with excellent Exterior Quality

count        52.000000
mean     367360.961538
std      116401.264200
min      160000.000000
25%      311404.000000
50%      364606.500000
75%      428788.500000
max      755000.000000
Name: SalePrice, dtype: float64

Summary statstics for houses with good Exterior Quality
```

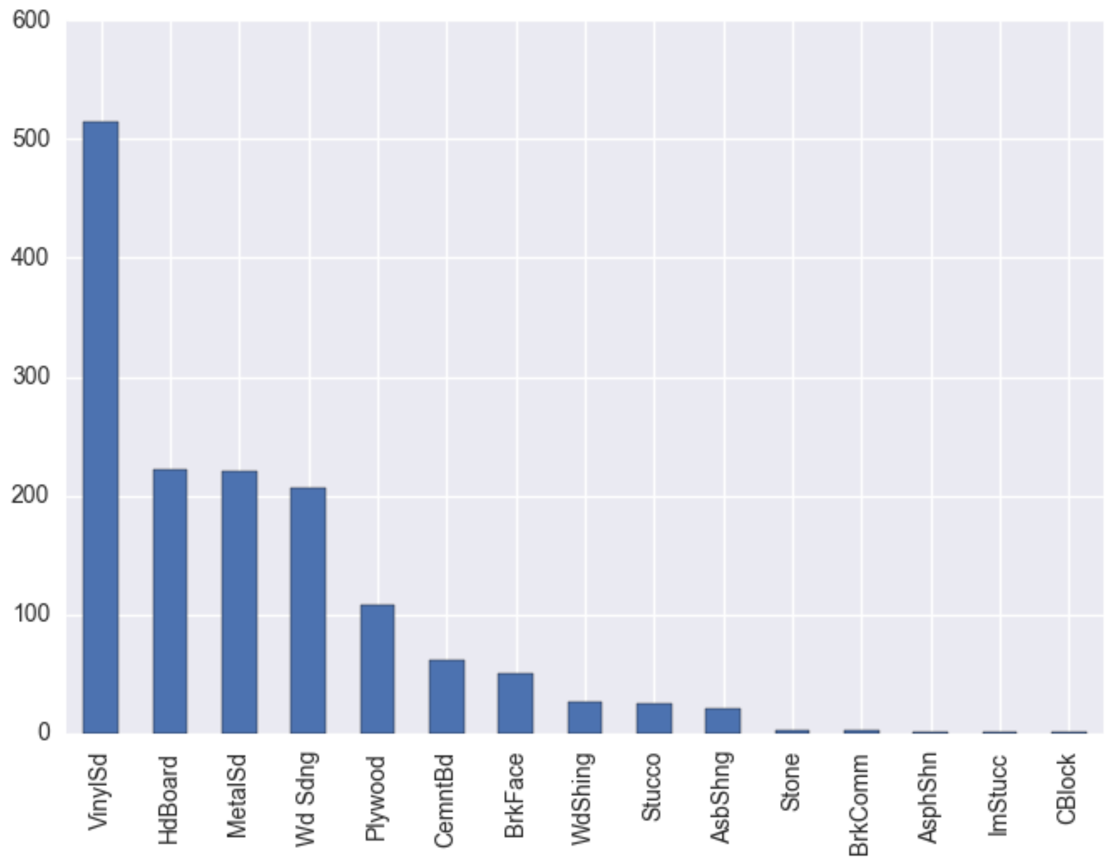
```
count       488.000000
mean     231633.510246
std       71188.873899
min       52000.000000
25%      185000.000000
50%      220000.000000
75%      265984.250000
max      745000.000000
Name: SalePrice, dtype: float64

Summary statstics for houses with average Exterior Quality

count       906.000000
mean     144341.313466
std       42471.815703
min       34900.000000
25%      118589.500000
50%      139450.000000
75%      165500.000000
max      381000.000000
Name: SalePrice, dtype: float64

Summary statstics for houses with fair Exterior Quality

count        14.000000
mean      87985.214286
std       39826.918794
min       39300.000000
25%       60250.000000
50%       82250.000000
75%      102000.000000
max      200000.000000
Name: SalePrice, dtype: float64
```

We can see that the price increases with the Quality of Materials used in the house exterior.

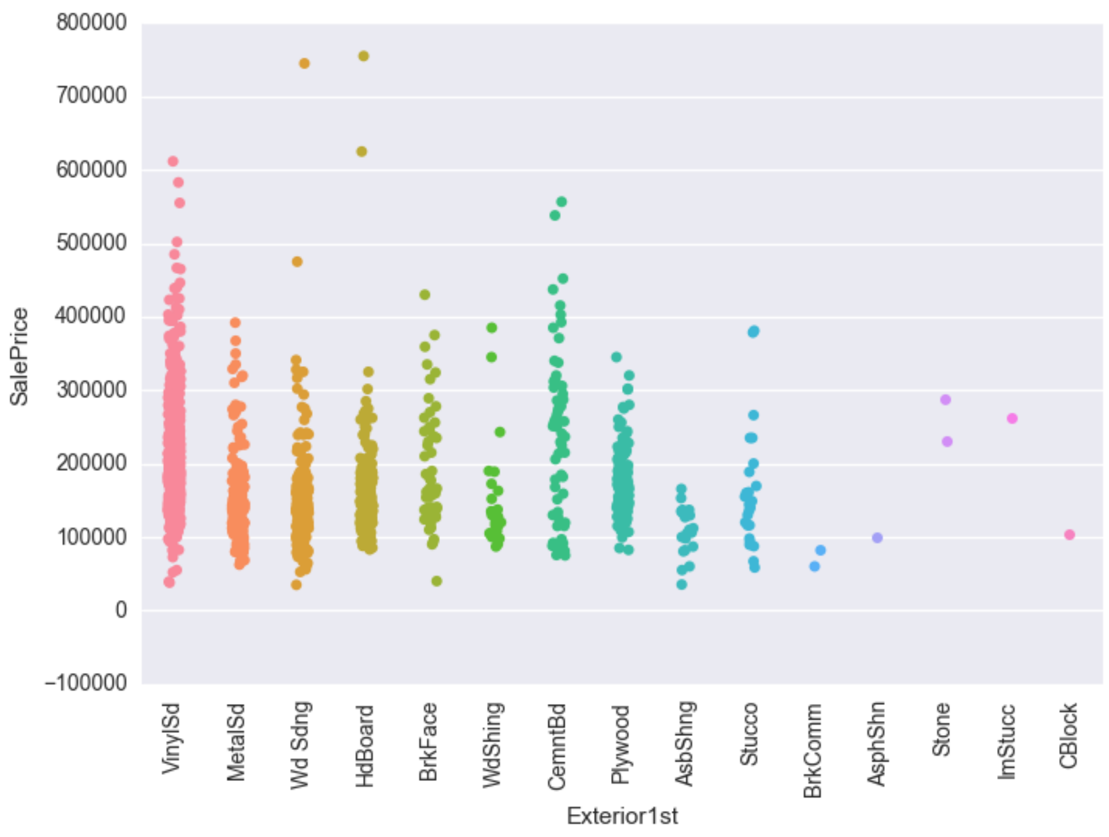
```
In [7]: sns.set(style="darkgrid", color_codes=True)
        sns.stripplot(x="ExterQual", y="SalePrice",\
                      data=train.loc[:,['ExterQual','SalePrice']],\
                      jitter = True)
        plt.xticks(rotation=90);
```



```
In [8]: X_tr.Exterior1st.value_counts().plot(kind='bar');
```



```
In [9]: sns.set(style="darkgrid", color_codes=True)
        sns.stripplot(x="Exterior1st", y="SalePrice", \
                      data=train.loc[:,['Exterior1st','SalePrice']],\
                      jitter = True)
        plt.xticks(rotation=90);
```

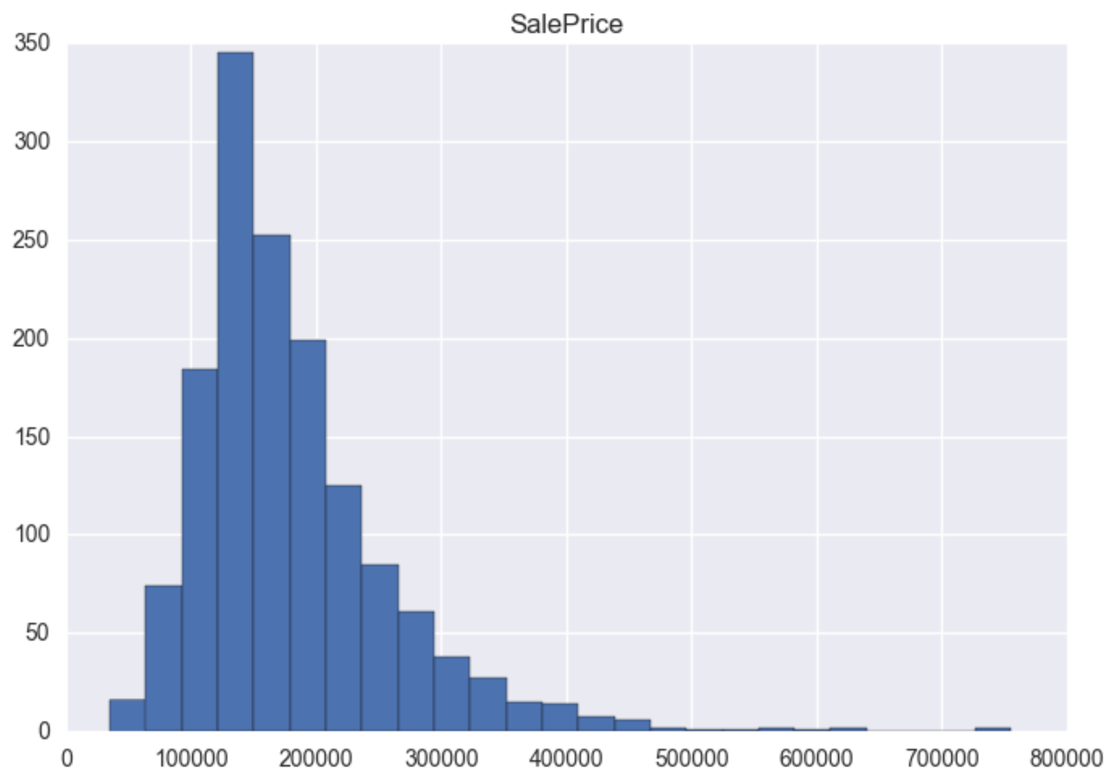


There also seems to be a correlation between the exterior covering of the house and its price. The quality of these materials maybe correlated with the overall quality of the house.

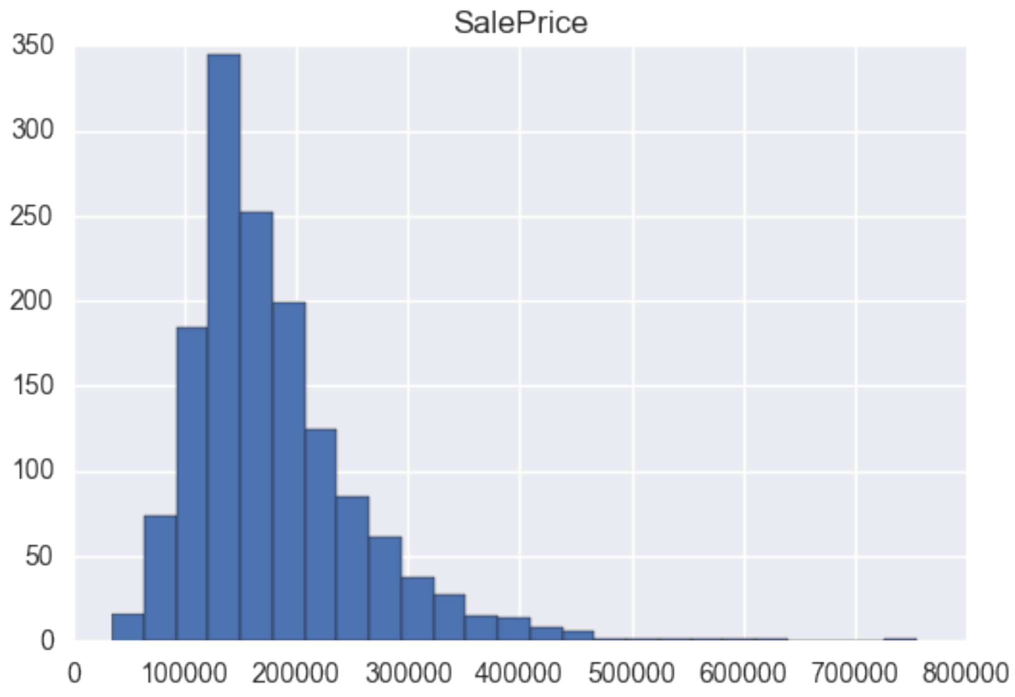
2.1.5 `SalePrice` distribution

We plotted histograms of the `SalePrice` variable. These histograms showed that the prices have a distribution that is close to a Gaussian distribution, hence justifying the use of the Linear-Log model for the skewed features.

```
In [14]: y_tr.hist(bins = 25);
```



```
In [15]: y_tr = np.log(y_tr)

In [52]: y_tr.hist(bins = 25);
```



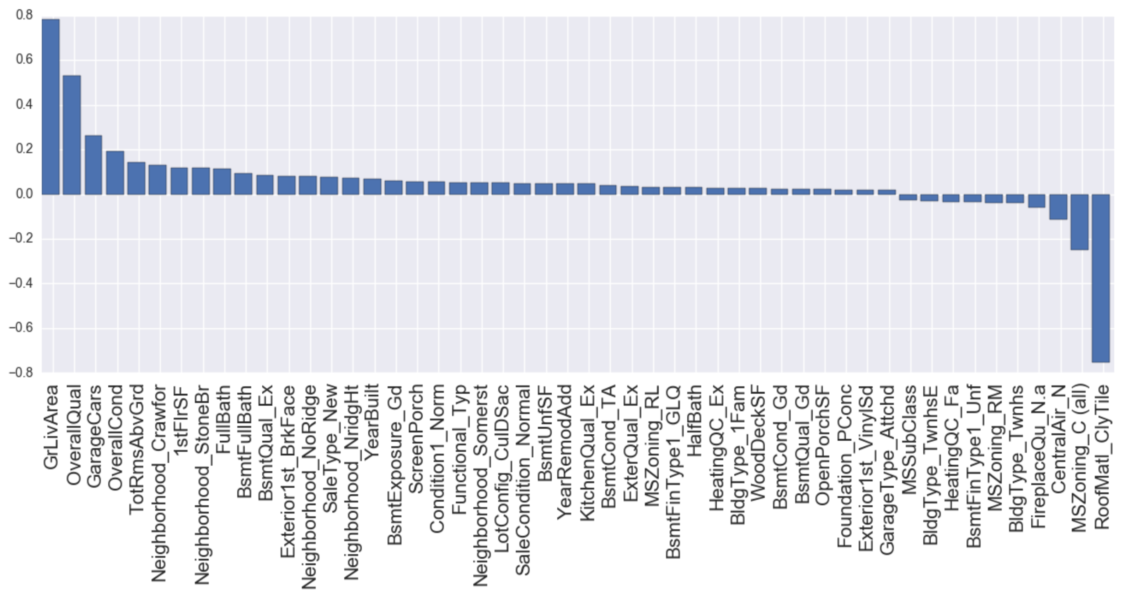
2.2 Features importance

Once we have explored the data, we can interest ourselves in the features importance.
Due to the concrete aspect of the challenge, we could think ; without using any algorithm ; about which features is important or not and how much.
However there are many features, and it's difficult to see if the size of the garage is more important than the quality of the exterior. Thus we are going to apply algorithms to our data in order to get an idea of the features importance.
The first thing we could think of would be to fit a linear regression on our data and look at the coefficient of the regression to see which features counts and which doesn't. To do that we have to assure that the data are normalised, indeed if it is not the case features with high values as the lot area would have very small coefficient in comparison to features as the overall quality which only takes values between 1 and 10.
Here we'll scale the data to have values between 0 and 1. Also we'll use Lasso linear regression instead of a classical linear regression because we've got a lot of features and Lasso helps us to pick the most interesting ones (Lasso procedure will be detailled in the fourth part of this report).

```
In [17]: from sklearn.linear_model import LinearRegression,Lasso,Ridge
         from sklearn.model_selection import KFold
         from sklearn.metrics import mean_squared_error
         from sklearn.preprocessing import MinMaxScaler
         from math import sqrt


         X_df = pd.get_dummies(X_tr)

         X_df.fillna(X_df.mean(),inplace = True)
         plt.figure(figsize=(15, 5))


         X = MinMaxScaler().fit_transform(X_df)
         y = y_tr.values
         skf = KFold(n_splits = 2,shuffle = True,random_state = 7)

         valid_train_is, valid_test_is = list(skf.split(X, y))[0]

         X_valid_train = X[valid_train_is]
         y_valid_train = y[valid_train_is]
         X_valid_test = X[valid_test_is]
         y_valid_test = y[valid_test_is]


         reg = Lasso(0.001)
         reg.fit(X_valid_train, y_valid_train)
```

```python
print(sqrt(mean_squared_error(y_valid_test,reg.predict(X_valid_test))))

l = np.concatenate([np.arange(0,40),np.arange(292,302)])
ordering = np.argsort(reg.coef_)[::-1][l]

importances = reg.coef_[ordering]

feature_names = X_df.columns[ordering]

x = np.arange(len(feature_names))
plt.bar(x, importances)
plt.xticks(x + 0.5, feature_names, rotation=90, fontsize=15);
```

```
0.131683876947
```



We have drawn the values of the 40 higher positive coefficients and the values of the 10 higher negative coefficients. From this plot, we can interpret that features as GrLivArea, OverallQual, GarageCars make the price increase when they increase, and that features like "RoofMatl is ClyTile" or "no Central Air" make the price decrease.
We can use other algorithms to try to understand the features importance in this problem. For instance we are going to apply a random forest algorithm to our data in order to get an idea of what features matter the most. However, unlike the linear model, we don't get directly a quantity which allows us to compare the features.
On way to obtain a quantity that could describe the features importance using random forest regression would be the following :

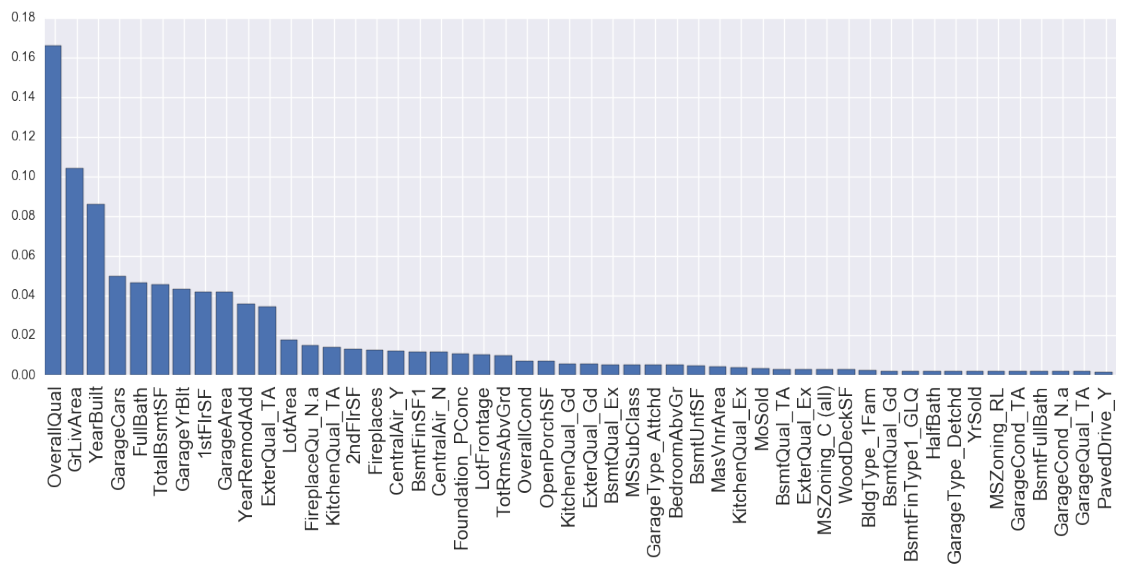
- for each tree in the forest : the importance associated with the features f would be the mean of the error reduction for each node involving f, weighted by the number of sample coming to that node. Where the error reduction of a node would be the sum of the errors of prediction for the node's incoming samples minus the sum of the errors of prediction after the node's split. Here we are working with the mean squared error.

- the quantity obtained for one tree would be averaged for all the tree in the forest

This quantity describe the importance of a feature linked to its power to decrease the training error. In other words, a feature is important if splitting according to this feature allows to reduce a lot the internal error.
In sklearn we have access to the feature importances through the feature_importances attribute of the RandomForestRegressor class.

```
In [31]: from sklearn.ensemble import RandomForestRegressor


         X_df = pd.get_dummies(X_tr)
         X_df.fillna(X_df.mean(),inplace = True)

         plt.figure(figsize=(15, 5))

         X = X_df.values
         y = y_tr.values
         skf = KFold(n_splits = 2,shuffle = True,random_state = 7)

         valid_train_is, valid_test_is = list(skf.split(X, y))[0]

         X_valid_train = X[valid_train_is]
         y_valid_train = y[valid_train_is]
         X_valid_test = X[valid_test_is]
         y_valid_test = y[valid_test_is]


         reg = RandomForestRegressor(n_estimators = 300,max_features = 0.2,\
                                     max_depth = 12,min_samples_leaf = 2)
         reg.fit(X_valid_train, y_valid_train)

         print(sqrt(mean_squared_error(y_valid_test,reg.predict(X_valid_test))))

         ordering = np.argsort(reg.feature_importances_)[::-1][:50]

         importances = reg.feature_importances_[ordering]
         feature_names = X_df.columns[ordering]

         x = np.arange(len(feature_names))
         plt.bar(x, importances)
         plt.xticks(x + 0.5, feature_names, rotation=90, fontsize=15);
```

```
0.145871228379
```



Here we printed the 50 most important features according to the random forest algorithm. We see that the top features are slightly different from those obtain through linear regression, however features as OverallQual, GarageSize or GrLivArea are still in top position, so we are confident they are really important features for our regression problem.

3 Data preprocessing

One of the most important part of our data challenge task is to preprocess the data we got from Kaggle in order to give to the learning algorithms the best representations of our data that can help getting good predictions.

3.1 Symetrize distributions

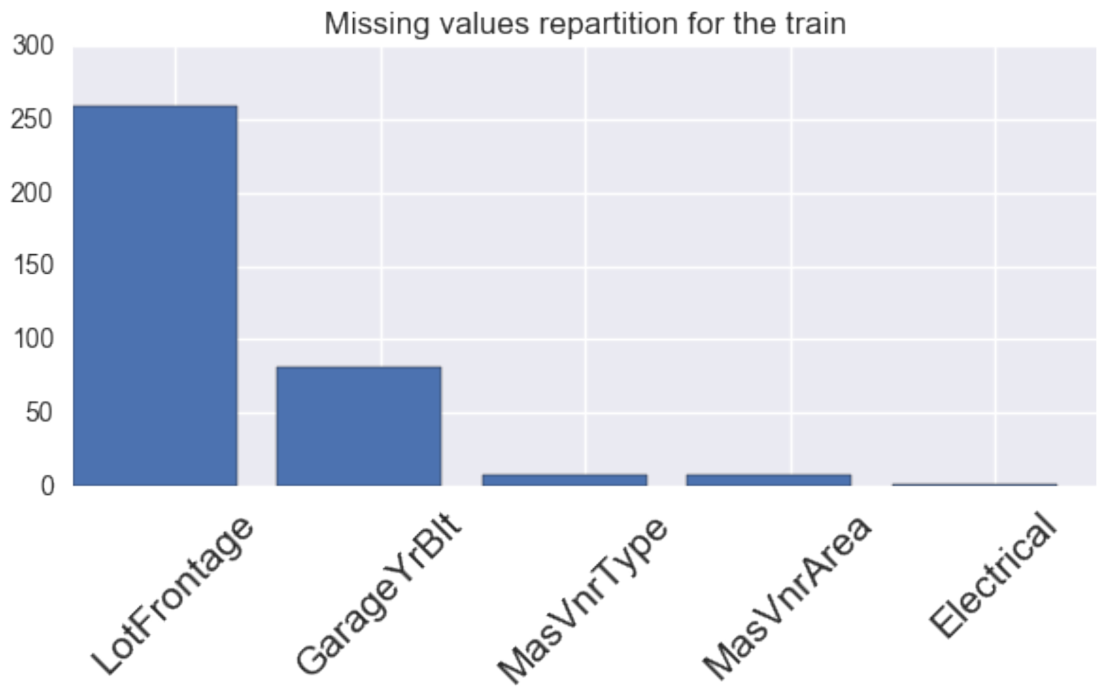
The first task that improve a lot the results in the beginning, was the normalization of the numerical features as we did for the SalePrice feature. To do that we used the *skew* function from the scipy.stats library. *skew*(x) return a postive value that indicates how asymmetric the distribution of x is. *skew*(x) $\approx$ 0 for normally distributed features. We apply the log to each features which have a *skew* values higher than a threshold (that become an hyperparameter).

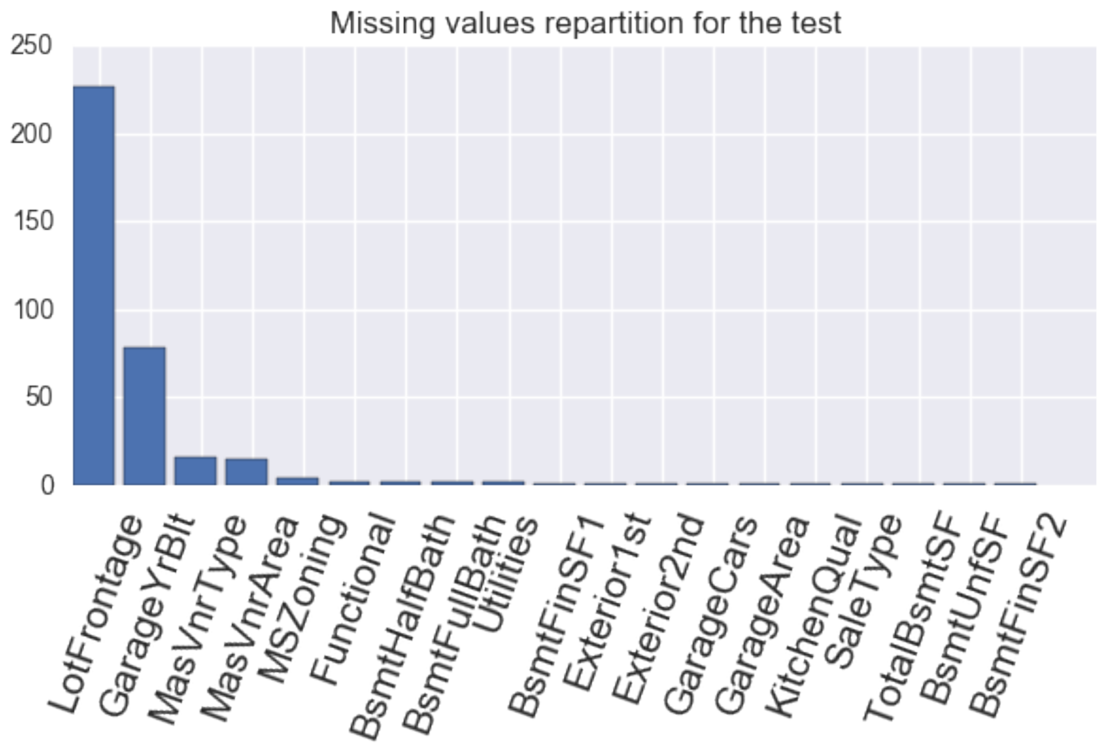
3.2 Missing values

Are there lots of missing values? How do they impact our predictions? How to deal with them?

```
In [32]: Count=X_tr.isnull().sum()
         ordering = np.argsort(Count)[::-1]
         Count=Count[ordering]
         Count=Count[Count>0]

         plt.figure(figsize=(7, 3))
```

```
x = np.arange(len(Count))
plt.bar(x, Count)
plt.title("Missing values repartition for the train")
plt.xticks(x + 0.5, Count.index, rotation=45, fontsize=15)

test = pd.read_csv("test.csv")

test.loc[test['PoolQC'].isnull(),'PoolQC'] = 'N.a'
test.loc[test['Fence'].isnull(),'Fence'] = 'N.a'
test.loc[test['MiscFeature'].isnull(),'MiscFeature'] = 'N.a'
test.loc[test['GarageCond'].isnull(),'GarageCond'] = 'N.a'
test.loc[test['GarageQual'].isnull(),'GarageQual'] = 'N.a'
test.loc[test['GarageFinish'].isnull(),'GarageFinish'] = 'N.a'
test.loc[test['GarageType'].isnull(),'GarageType'] = 'N.a'
test.loc[test['FireplaceQu'].isnull(),'FireplaceQu'] = 'N.a'
test.loc[test['BsmtFinType2'].isnull(),'BsmtFinType2'] = 'N.a'
test.loc[test['BsmtFinType1'].isnull(),'BsmtFinType1'] = 'N.a'
test.loc[test['BsmtExposure'].isnull(),'BsmtExposure'] = 'N.a'
test.loc[test['BsmtCond'].isnull(),'BsmtCond'] = 'N.a'
test.loc[test['BsmtQual'].isnull(),'BsmtQual'] = 'N.a'
test.loc[test['Alley'].isnull(),'Alley'] = 'N.a'

Count2=test.isnull().sum()
ordering = np.argsort(Count2)[::-1]
Count2=Count2[ordering]
Count2=Count2[Count2>0]

plt.figure(figsize=(7, 3))
x = np.arange(len(Count2))
plt.bar(x, Count2)
plt.title("Missing values repartition for the test")
plt.xticks(x + 0.5, Count2.index, rotation=70, fontsize=15);
```

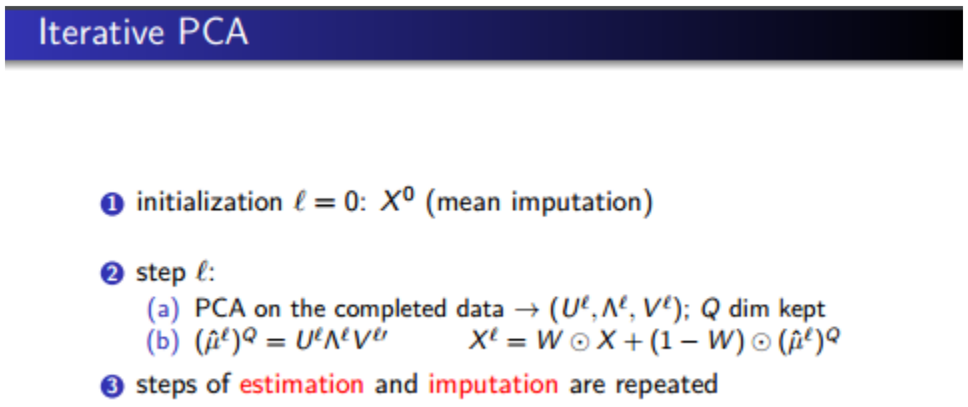

Missing values repartition for the train
300
250
200
150
100
50
0
LotFrontage
GarageYrBlt
MasVnrType
MasVnrArea
Electrical


Missing values repartition for the test
250
200
150
100
50
0
LotFrontage
GarageYrBlt
MasVnrType
MasVnrArea
MSZoning
Functional
BsmtHalfBath
BsmtFullBath
Utilities
BsmtFinSF1
Exterior1st
Exterior2nd
GarageCars
GarageArea
KitchenQual
SaleType
TotalBsmtSF
BsmtUnfSF
BsmtFinSF2

First of all we must mention the ambiguity of some 'NA' values. Infact, when taking categorical variables, sometimes, 'NA' must be considered as real values and not missing ones. When looking at 'GarangeCond' for example, 'NA' just means that there is no garage. This is no big deal since 'NA' will then just have to be considered as another potential value of the variable.
Nevertheless, this phenomena becomes a problem with numerical values. When measuring lot frontage are, does 'NA' mean that there is no lot frontage? Should you then replace 'NA' by 0? By -Infinity? This is not always clear and the question is still open.
For the moment, let's look at the data. The most incomplete column is "LotFrontage" in both cases. We can see that there are not so many missing values but it is still necessary to handle them in an intelligent way. In the competition we simply replaced them with mean values for quantitave attributes and most frequent value for qualitative attribute. However, we will here present another method which we have not used but that is theorically much more interesting. It uses PCA and replace the missing values with what can be seen as a regression of all the other columns. Here below a description of the IterativePCA algorithm from the EA class "R for statistics".

```
In [17]: Image("Capture.png")
```

Out[17]:



Iterative PCA

1. initialization $\ell = 0$: X^0 (mean imputation)
2. step ℓ:
   (a) PCA on the completed data $\rightarrow (U^\ell, \Lambda^\ell, V^\ell)$; Q dim kept
   (b) $(\hat{\mu}^\ell)^Q = U^\ell \Lambda^\ell V^{\ell\prime}$ $\qquad X^\ell = W \odot X + (1 - W) \odot (\hat{\mu}^\ell)^Q$
3. steps of estimation and imputation are repeated

```
In [20]: def IterativePCA(X_tr,n,dimension):

             Numeric_col=X_tr._get_numeric_data().columns
             X=X_tr._get_numeric_data()

             I=Imputer(strategy='mean')
             PCA1=PCA(n_components=dimension)
             S=StandardScaler()


             X_iter=I.fit_transform(X)
             X0=I.fit_transform(X)
             S.fit(X_iter)
```

```
    X_iter=S.transform(X_iter)
    X0=S.transform(X0)
    W=1*np.array(~X.isnull())

    for i in range(n):
        PCA1.fit(X_iter)
        X_iter=PCA1.inverse_transform(PCA1.transform(X_iter))
        X_iter=W*X0+(1-W)*X_iter
    X_iter=S.inverse_transform(X_iter)
    New=pd.DataFrame(X_iter,index=X.index,columns=X.columns)
    X_tr[col]=New
    return X_tr
```

Let's make a few remarks about what has been written:

- When the parameter "dimension" equals the size of the matrix, the function PCA.inverse_transform and PCA.transform are inverses. As a consequence the algorithm is equivalent to a simple mean imputation. In general, the best value of the "dimension" parameter has to be found by cross-validation
- It is important to scale the data before computing the PCA in order to deal with comparable standart deviations
- Here the algorithm has just been used to modify the train dataset. Obviously, the same idea can be applied to the test dataset. However, we will have to be careful about continuing to fit the scaler, the PCA and the Imputer on the train and not on the test dataset.

3.3 Dealing with cathegorical features

The consideration of categorical variables allows us to :

- ** Take more variables into consideration to reduce the error **: by adding more variables we can increase the percentage of explained variance for the output.
- ** Decrease the bias of a variable allready taken into account ** if this variable is correlated with the categorical variable.

3.3.1 Use of the `pandas.get_dummies()`

The `get_dummies` function allows to create indicator variables for each level of the original categorical variable.
This approach also allows us to construct new variables $D_{i,j} = \mathbb{1}_{X_i=j}$
Thus the new regressor :

$$Y = X\beta^* + D\gamma + \epsilon$$

With β and γ minimizing the loss function of the regressor used.
** avoid the dummy trap **
When dealing with dummy variables it is frequently better to drop one of the indicators. Doing so avoids to have correlation between the different indicators variables. This is why we used `drop_first=True`.

3.3.2 Using `cat.codes`

We tried another approach to the dummy variables. This approach consisted to attribute integers to the levels of each categorical variable. In order to do so, we used the `astype('category').cat.codes`. This approach did not gave any result so we tried to change the integers by sorting the mean `SalePrice` for the different levels of the categorical variable considered but it did not improve the results on the Cross-Validation.

3.3.3 Rating type variable

Sometimes, categorical variable take values that can easily be ordered. For example, the factors of "KitchenQuality" variable are just gradual adjectives: Excelent, Good, Typical/Average, Fair and Poor. An simple idea could be to replace them by numbers. Maybe by a linear quantification 4,3,2,1,0. Maybe by a quadratic qualification 16,9,4,1,0. Testing is certainly the best way to have the answer but a good idea could also be to look at the plots and try to see if the impact is more quadratic, linear or logarithmic for example.

```
In [22]: QuallDict={'Ex':10, 'Gd': 7, 'TA': 5,'Av':5, 'Fa': 3, 'Mn': 3,\
                    'Po': 1, 'No':0, 'N.a': 0}

         train_aux = train.copy()

         #Exterior Features
         train_aux['ExterQual']=[QuallDict[w] for w in train['ExterQual']]
         train_aux['ExterCond']=[QuallDict[w] for w in train['ExterCond']]

         #Basement Features
         train_aux['BsmtCond']=[QuallDict[w] for w in train['BsmtCond']]
         train_aux['BsmtExposure']=[QuallDict[w] for w in train['BsmtExposure']]

         #Heating
         train_aux['HeatingQC']=[QuallDict[w] for w in train['HeatingQC']]

         #Kitchen
         train_aux['KitchenQual']=[QuallDict[w] for w in train['KitchenQual']]

         #Fireplace
         train_aux['FireplaceQu']=[QuallDict[w] for w in train['FireplaceQu']]

         #Garage
         train_aux['GarageQual']=[QuallDict[w] for w in train['GarageQual']]
         train_aux['GarageCond']=[QuallDict[w] for w in train['GarageCond']]

         #Pool
         train_aux['PoolQC']=[QuallDict[w] for w in train['PoolQC']]
```

3.4 Features engineering

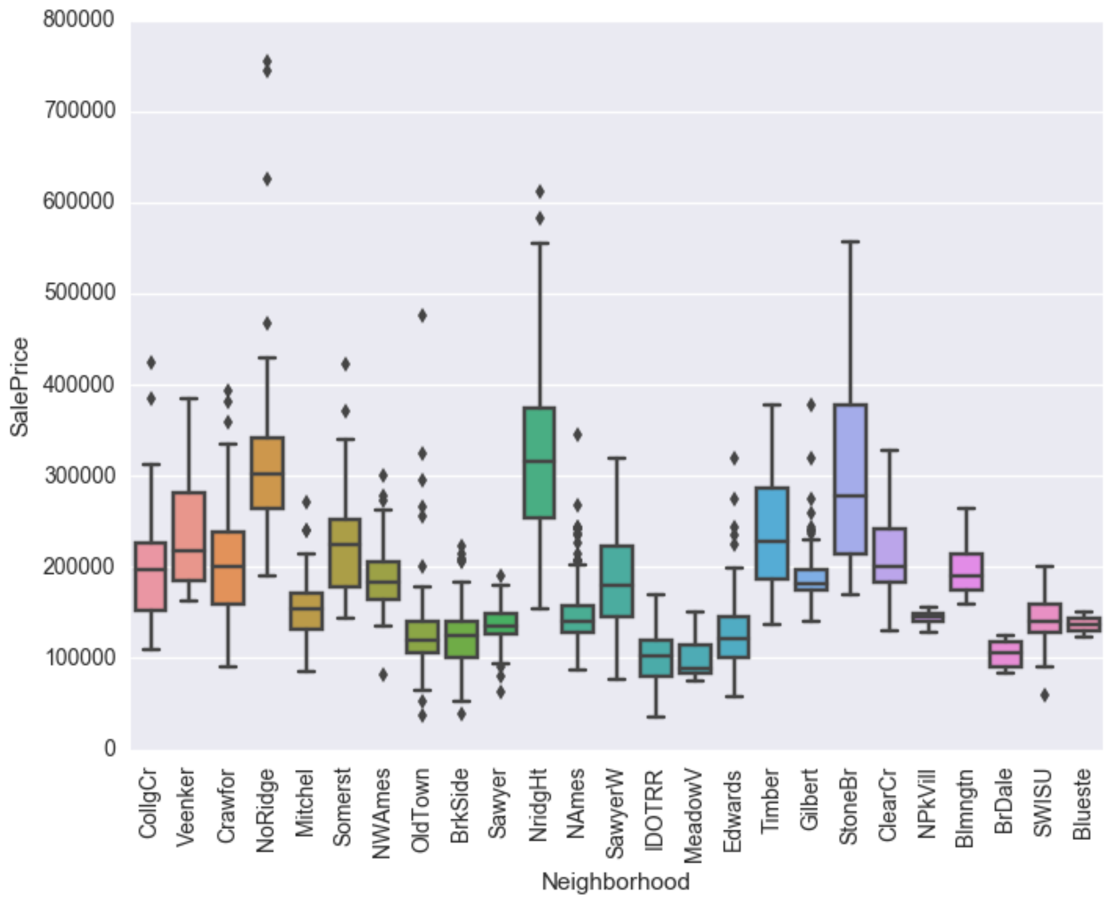
Independetly from the algorithm, is there some special feature ingeenring that can help better exploiting the data? Can we change some of the columns? Are there some columns we can add?
1) Continious variable discretisation This technique is very simple. We juste wrote a function that would enable us to discretize a continous variable into its quantils. The way we used it very empirical since we could not find a lot of accessible theory about this. Basically we are trying to do here is to give away a bit of information hoping, on the other hand, to come up with a more generalisable model.

```
In [19]: class Discretize():




             def __init__(self,quantil):
                 self.quantil = quantil




             def fit(self,y):
                 q=self.quantil

                 s=np.arange(100/q+1)
                 s=s*q
                 self.table=np.percentile(y,s)
                 return(self.table)




             def transform(self,y):
                 table=self.table
                 for i in range(len(y)):
                     for j in range(len(table)):
                         if (y.iloc[i]<table[j]):
                             y.iloc[i]=(table[j]+table[j-1])/2
                             break
                 return(y)
```

** "Target-based" columns**
The idea would be to add columns wich contain information coming from the target. For example we could use a column giving the mean of the Sell Price in each neighborhood or the mean of the square meter price in according to the type of house.
There are two main inconvenients with this method.
Firstly, it is necessary to modifiy the test set too and therefore tamper it with the prices of of the train set.
Secondly, the impact of this technique is difficult to measure. As a matter of fact, the score of the homemade cross validation decreased a lot, but is this significant? It is indeed normal that you get a better result when you add columns containing informations such as means about what

you want to predicit. Probably the right way to measure would be to cut X and Y into two parts, compute the means on the first parts and use the second ones for cross validation. However, this process is quite heavy to compute and moreove our cross-validation will have less and less houses to test which make it less reliable.

```
In [20]: data = pd.concat([train['SalePrice'], train['Neighborhood']], axis=1)
         f, ax = plt.subplots(figsize=(8, 6))
         fig = sns.boxplot(x='Neighborhood', y="SalePrice", data=data)
         fig.axis(ymin=0, ymax=800000)
         plt.xticks(rotation=90);
```



** Other feature engeenering tricks**
Last but not least, here is a small compilation of tricks and intuitive techniques that we discovered or learned from a kernel which seem to improve a bit the prediction. You will also find here below simples lines of code that enable to compute such ideas
1) Manualy try to identify outliers and delete them.
For example, it appears that there are only a few houses where the variable "GrLivArea" is superior to 4000. These are probably luxurious houses which may be not representative at all. Regressiong have the drawback to be very sensible to outliers, deleting these few houses could possibly improve a bit the score

train.drop(train[train["GrLivArea"] > 4000].index, inplace=True)
2) Reducing the number of factors of a categorical value
If we take a variable such as "LotShape", informations about the regularity of the LotShape are give. Shapes can be "Regular", "Slightely irregular", "Moderately Irregular" or just "Irregular". However, in practive, most of the houses are located in "Regular" lots and only a few of them are in Irregular or Moderately Irregular ones. The idea here would be to simplify the distinction and just make a distinction between regular and irregular lots. Once again, we give up information but we hope to come up with a potentially more generalisable model.
Train["IsRegularLotShape"] = (df["Train"] == "Reg")
3) Understand the physical meaning of each feature
This may appear as an evidence but there are sometimes easy but very usefull considerations that can found by just thinking of the type of data we are dealing with. For example, among the various features that are given, there is the year in which the house was build and there is the year in which it was sold. Just adding a column where it is checked if these two years are equal or not is indeed quite smart. If there is equality this mean that the house was brand new when it was sold. This information surely has an important impact on its sell price.
Train["VeryNewHouse"] = (Train["YearBuilt"] == all_df["YrSold"])

4 Prices predictions

In this part we present the algorithm that we studied and using at some point during the challenge.

4.1 Evaluation of our model performances

In Kaggle type competition, we have to predict values on a test set that we don't know the labels. Due to the classical Bias-Variance trade-off we can't content ourselves to minimise the training error, and in particular the bias, which conducts irremediably to over-fitting.
To test the predictive power of our algorithms we have to split our train set into train/test sets in order to have a labelled test set. We use the K-folds method to obtain an estimation of our testing error. Indeed we split our train set into K sets, and test our algorithm on each of this K sets after training it on the four others. We obtained K training errors that we can average. Here we doesn't have any order between the samples, so we can split randomly to obtain the K sets.
In the literature K is often choose as 5 or 10, here we prefer choosing 5 folds because we don't have a large amount of train samples.
We can use this computed test error to compare our different algorithms and in particular to choose the different hyper-parameters of our models.
We preferred recoding a cross validation function using K-fold so we could also deal with the pre-processing in the process. Indeed preprocessing should be done for each set of the split separately, especially when we use features as the mean of prices for each neighbourhoods (means should be calculated using only the train set because in the other case we use information of the test labels in our features, which leads to under estimation of the cross validation error).

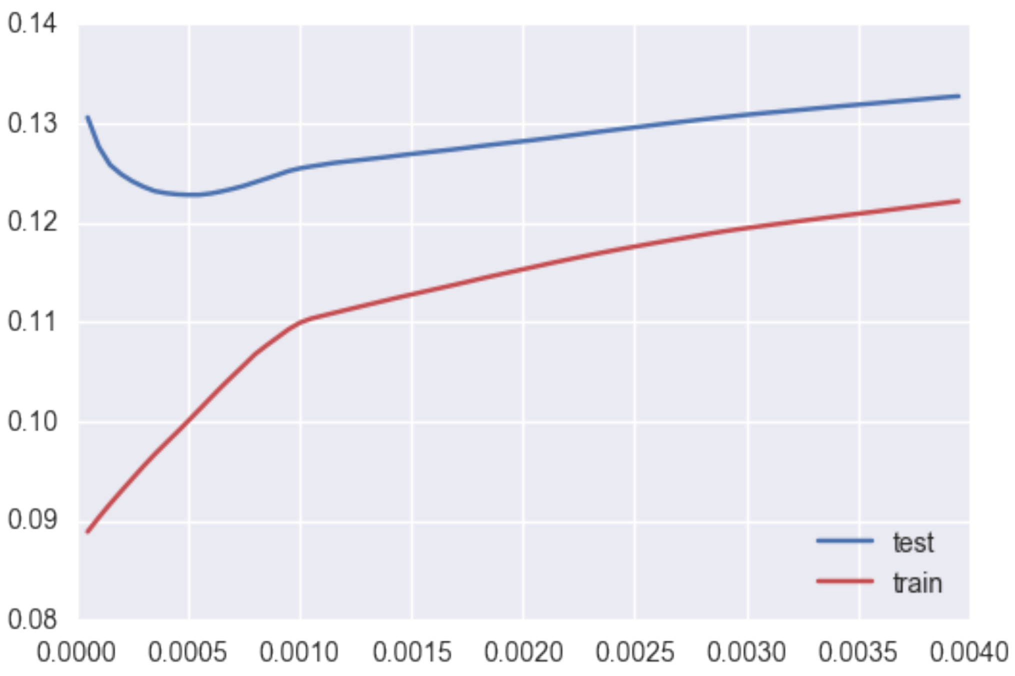
```
In [24]: from sklearn.preprocessing import StandardScaler
         from scipy.stats import skew

         class Preprocess():

             def __init__(self):
```

```
        self.scaler = StandardScaler()
        self.summary_cat = pd.DataFrame()
        pass

    def fit_cat(self,X,y):

        self.mean = dict()
        self.var = dict()
        self.Neigh = ['Blmngtn','Blueste','BrDale','BrkSide','ClearCr','CollgCr',\
         'Crawfor','Edwards','Gilbert','IDOTRR','MeadowV','Mitchel',\
         'NAmes','NoRidge','NPkVill','NridgHt','NWAmes','OldTown',\
         'SWISU','Sawyer','SawyerW','Somerst','StoneBr','Timber','Veenker']
        for name in self.Neigh :
            self.mean[name] = np.nanmean(y.loc[X['Neighborhood'] == name,'SalePrice'])
            self.var[name] = np.nanvar(y.loc[X['Neighborhood'] == name,'SalePrice'])

        self.Zone = ['FV','RH','RL','RM']
        for name in self.Zone :
            self.mean[name] = np.nanmean(y.loc[X['MSZoning'] == name,'SalePrice'])


        self.Func = ['Typ','Min1','Min2','Mod','Maj1','Maj2','Sev']
        for name in self.Func:
            self.mean[name] = np.mean(y.loc[X['Functional'] == name,'SalePrice'])
            self.var[name] = np.var(y.loc[X['Functional'] == name,'SalePrice'])

        return self

    def fit_num(self,X,y):


        for i in np.arange(1,11) :
            self.mean[i] = np.nanmean(y.loc[X['OverallQual'] == i,'SalePrice'])

        return self


    def transform_num(self,train,test = False):
        X = train.copy()

        X['OverallQuallMean'] = 0
        for i in np.arange(1,11) :
            X.loc[X['OverallQual'] == i,'OverallQuallMean'] = self.mean[i]

        X['Yr'] = (X['YearBuilt']*X['YearRemodAdd'])
        X['Overall'] = (X['OverallQual']*(X['OverallCond']))
        X['Tt_Area'] = (X['TotalBsmtSF']/(X['GrLivArea']))
        X.fillna(X.mean(),inplace = True)
```

```
            if not test :
                self.skewed_feats = X.apply(lambda x: skew(x.dropna())) #compute skewness
                self.skewed_feats = self.skewed_feats[self.skewed_feats > 0.75]
                self.skewed_feats = self.skewed_feats.index

            X[self.skewed_feats] = np.log1p(X[self.skewed_feats])
            if(not test):
                self.scaler.fit(X)

            X = pd.DataFrame(self.scaler.transform(X))

            return X

        def transform_cat(self,train):
            X = train.copy()

            X['Neighmean'] = 0
            X['Neighvar'] = 0
            X['Zonemean'] = 0
            X['Funcmean'] = 0
            X['Funcvar'] = 0
            for name in self.Neigh :
                X.loc[X['Neighborhood'] == name,'Neighmean'] = self.mean[name]
                X.loc[X['Neighborhood'] == name,'Neighvar'] = self.var[name]


            for name in self.Zone :
                X.loc[X['MSZoning'] == name,'Zonemean'] = self.mean[name]


            for name in self.Func:
                X.loc[X['Functional'] == name,'Funcmean'] = self.mean[name]
                X.loc[X['Functional'] == name,'Funcvar'] = self.var[name]

            X.fillna(X.mean(),inplace = True)

            return X
```

```
In [25]: from sklearn.model_selection import KFold



         def RMSE(y,y_pred):
             return(np.sqrt(sum((y['SalePrice']-y_pred)**2)/len(y)))

         def Dummies(X_train,X_test):
             tot = pd.concat([X_train,X_test],axis = 0)
```

```
            tot = pd.get_dummies(tot)
            return tot.iloc[:X_train.shape[0],:],tot.iloc[X_train.shape[0]:,:]

        def crossVal(model,X,y,cv = 5):
            Kfold = KFold(n_splits=cv,shuffle=True,random_state=7)
            prep = Preprocess()
            res = []
            tr_error = []
            for train,test in  Kfold.split(X,y):
                X_train, X_test, y_train, y_test = X.iloc[train,:],X.iloc[test,:],\
                y.iloc[train,:],y.iloc[test,:]
                prep.fit_cat(X_train,y_train)
                prep.fit_num(X_train,y_train)
                X_train_num = X_train.select_dtypes(include= ['int','float'])
                X_train_cat = X_train.select_dtypes(include = ['object'])
                X_train_num = prep.transform_num(X_train_num)
                X_train_cat = prep.transform_cat(X_train_cat)
                X_train_num.index = X_train_cat.index
                X_train = pd.concat([X_train_num,X_train_cat],axis = 1)
                X_test_num = X_test.select_dtypes(include= ['int','float'])
                X_test_cat = X_test.select_dtypes(include = ['object'])
                X_test_num = prep.transform_num(X_test_num,test = True)
                X_test_cat = prep.transform_cat(X_test_cat)
                X_test_num.index = X_test_cat.index
                X_test = pd.concat([X_test_num,X_test_cat],axis = 1)
                X_train,X_test = Dummies(X_train,X_test)

                model.fit(X_train,y_train)
                y_pred = model.predict(X_test)
                res.append(RMSE(y_test,y_pred))
                tr_error.append(RMSE(y_train,model.predict(X_train)))
            return np.mean(res),np.mean(tr_error)

In [33]: def plot_errorL(l):
            res = []
            trai = []
            for alpha in l:
                tst,tr = crossVal(Lasso(alpha=alpha),X_tr,y_tr)
                res.append(tst)
                trai.append(tr)
            plt.plot(l,res,label ='test')
            plt.plot(l,trai,label ='train',color = 'r')
            plt.legend(loc = 4)
            plt.show()


        plot_errorL(np.arange(0.00005,0.004,0.00005))
```



Here we see, as we could expect, that test and train errors don't behave the same way, training error increase with the penalisation, whereas test error have a minimum for a $\lambda \neq 0$.
When there are many parameter to optimize, drawing curves is not more a option. We used the library *pysmac* and in particular the *fmin* function from the *optimize* module. It performs a random search in a delimited domain for our hyper-parameters.

```
In [34]: from pysmac.optimize import fmin

         def objective_function(x_int):
             objective_function.n_iterations += 1
             n_estimators,max_depth,max_features =x_int
             n_estimators = int(n_estimators)
             max_depth = int(max_depth)
             max_features = int(max_features)
             reg = RandomForestRegressor(n_estimators=n_estimators,\
                                         max_depth=max_depth,\
                                         max_features = max_features/10.0)
             score,tr = crossVal(reg, X_tr, y_tr)
             print (objective_function.n_iterations, \
                 ":\t n_estimators = ", n_estimators, \
                 "\n\t max_depth = ", max_depth,\
                 "\n\t max_features = ", max_features/10.0, \
                 "\n\t rmse = ", score)
             return score

In [35]: objective_function.n_iterations = 0
         xmin, fval = fmin(objective_function,\
                           x0_int=(10,5,2), xmin_int=(10,5,2), xmax_int=(100,20,10),\
                           max_evaluations=10)

1 :         n_estimators =  10
         max_depth =  5
         max_features =  0.2
         rmse =  0.162751482072
Number of evaluations 1, current fmin: 0.162751
2 :         n_estimators =  17
         max_depth =  15
         max_features =  0.8
         rmse =  0.144158771024
Number of evaluations 2, current fmin: 0.144159
3 :         n_estimators =  71
         max_depth =  8
         max_features =  0.4
         rmse =  0.140157522191
Number of evaluations 3, current fmin: 0.140158
4 :         n_estimators =  19
         max_depth =  11
         max_features =  0.7
         rmse =  0.141415477684
5 :         n_estimators =  91
         max_depth =  8
         max_features =  0.7
         rmse =  0.14078494351
6 :         n_estimators =  100
         max_depth =  10
         max_features =  1.0
         rmse =  0.140689722342
7 :         n_estimators =  54
         max_depth =  7
         max_features =  0.5
         rmse =  0.142605955384
8 :         n_estimators =  65
         max_depth =  6
         max_features =  0.3
         rmse =  0.147198344463
9 :         n_estimators =  97
         max_depth =  14
         max_features =  0.2
         rmse =  0.133482048989
Number of evaluations 9, current fmin: 0.133482
10 :         n_estimators =  38
         max_depth =  17
         max_features =  0.2
         rmse =  0.136268839459
Number of evaluations 10, fmin: 0.133482
```

The Kaggle competitions have the particularity of using a public and a private leaderboard on which our submissions are evaluated on a certain % of the test data. However the public score that allows you to compare what you we did with other can often be a biased indicator. In all data challenge tutorial we can read that cross-validation score should be more trusted than kaggle's public leaderboard. And indeed our cross-validation score was better than our public score and when the private came up we had the good suprised to go up from the 844th rank to the 34th.

4.2 Linear models

Classical linear model :

$$Y = X\beta^* + \epsilon$$

Least-squares minimisation:

$$\hat{\beta} \in argmin_{\beta}||Y - X\beta||^2$$

Ridge minimisation:

$$\hat{\beta} \in argmin_{\beta}||Y - X\beta||^2 + \lambda||\beta||^2$$

Lasso minimisation:

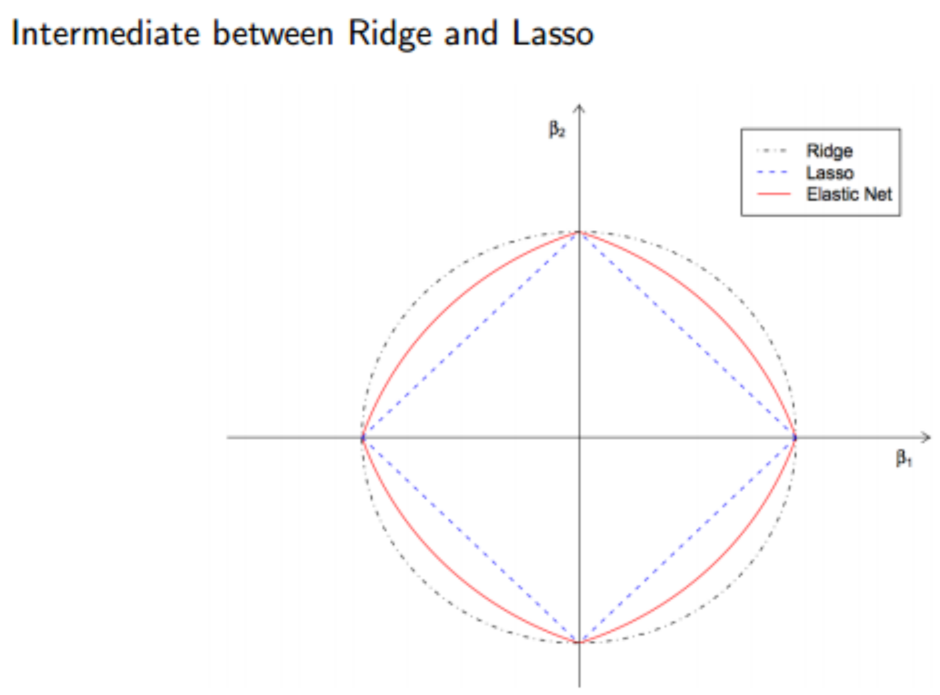
$$\hat{\beta} \in argmin_{\beta}||Y - X\beta||^2 + \lambda|\beta|_{l^1}$$

One possible advantage of Ridge and Lasso compared to the classical Least-squares method is that they penalyse complex models. As a consequence, the results obtained will have less chances to overfitt, problem which is all the plausible when there aren't so many instances like in this dataset. Now the thing is that Ridge and Lasso penalise in different ways. Lasso is much more direct with attributes in the sens that it will set to zero more easily attributes that are not so much correlated to the target and when two attributes are very similar the chances are that one of the gets truncated. The L1 regularisation lead to a sparse solution. On the contrary the risk with Ridge is that it may be too sensible to noise or uncorrelated attirubutes.
A solution is known as elastic Net. This can be seen as a sort of compromise between the two penalisations.
Elastic net:

$$\hat{\beta} \in argmin_{\beta}||Y - X\beta||^2 + \lambda|\beta|_{l^1} + \mu||\beta||^2$$

```
In [46]: from IPython.display import Image
         Image("Lasso.png")
```

```
Out[46]:
```



Adaptative Lasso [1]
We saw that what could be considered as a drawback with Lasso was its tendency to lead to sparse solutions and in particular to reduce the contribution of all the features even the most important. There is an algoithm called "Adaptative Lasso" that tries to adapt Lasso's penalization.
The penalisation $\lambda \sum \beta_j$ gets replaced by $\lambda \sum \omega_j \beta_j$
In other words, coefficients are build to prevent small coordiantes to be shrinked. When β_j get small, the coefficients ω_j will have to be big and vice versa.
Initially we choose $\omega_j = \frac{1}{(\beta_{j,Ridge})^\gamma}$ where β_{Ridge} is the classical Ridge solution and γ a power of our choice.
It is obviously possible to iterate various time this technique and compute everytime new coefficients. Here under, find an implementation of the AdaptativeLasso algorithm.

```
In [47]: from sklearn.linear_model import Lasso
         import numpy as np


         class AdaptativeLasso():




             def __init__(self,alpha,n_iter):
                 self.alpha = alpha
                 self.n_iter = n_iter
```

```
    def get_params(self,deep =True):
        out = dict()
        out['alpha']= self.alpha
        out['n_iter']= self.n_iter
        return out

    def fit(self,X,y):
        gprime = lambda w: 1. / \
        (2. * np.sqrt(np.abs(w)) + np.finfo(float).eps)
        n_samples, n_features = X.shape

        self.weights = np.ones(n_features)
        n_lasso_iterations = self.n_iter

        for k in range(n_lasso_iterations):
            X_w = X / self.weights
            clf = Lasso(alpha=self.alpha)
            clf.fit(X_w, y)
            coef_ = clf.coef_ / self.weights
            self.weights = gprime(coef_)
        self.clf = Lasso(self.alpha);
        self.clf.fit(X / self.weights,y)


    def predict(self,X):
        return self.clf.predict(X / self.weights)
```

** Taking height on regression problem **
This whole part is a summary of the main ideas and results about "Regressions Algorithms" you can find on the piece of work Foundation of Machine Learning[2].
Let's formalise a bit more the problematic.
By X and Y we denote our input and target spaces. D is the unknown distribution over X accordinf to which the points are drawn. $f : X- > Y$ is the target function.
To approximate this target function we have several hypothesis h beloging to a set called H. We also have a so called Loss function $L : YxY- > R_+$
For classic linear regressions: $H = \{x \rightarrow w \cdot x + b\}$
For ridge regressions: $H = \{x \rightarrow w \cdot x : ||w|| < \Gamma\}$
For lasso regressions: $H = \{x \rightarrow w \cdot x : ||w||_{L1} < \Gamma\}$
The ideal h would be an h that minimises the following quantity:

$$R(h) = E_{x\ D}[L(h(x), f(x))]$$

This is difficult when D is unknown so the basic idea of regression is to minimise the following quantity:

$$\hat{R}(h) = \frac{1}{m} \sum L(h(x_i, y_i)$$

This being said, in regressions, two quantities must be controlled: $\hat{R}(h)$ but also $|\hat{R}(h) - R(h)|$

Now, the idea of penalization becomes very clear. Adding contraints in the minmization problem will of course make us less performant with $\hat{R}(h)$ but we are seeking a compromise and with that, we want to diminish the orther quantity which is $|\hat{R}(h) - R(h)|$.
Kernel regressions
Generally, the loss function squared difference: $L(y, y') = ||y - y'||^2$ which can be expressed as a scalar product. Here comes the idea of kernels. Let's consider a mapping $\Phi : X- > H$ where H is an Hilbert called feature space. H is generally much bigger than X. It can contain not only the coordinates of each points but maybe their products, their differences, their power...Working in H would be much more efficient.
A Kernel K is a positive definite symmetric function such that:

$$K(x, x') =< \Phi(x), \Phi(x') >$$

Consequently, it becomes easy to compute the such regression by just replacing the classical scalar product with the kernel K.
Kernel regression are generalisation of classical regressions. Except Lasso regression, classical and ridge regressions can be computed with a kernel.
For Kernel ridge regression: $H = \{x \rightarrow w \cdot \Phi(x) : ||w||_H < \Gamma\}$
** Computation complexity of Kernel regression**
N is dimension of the feature space H, m denotes the number of points and κ is used to denote the complexity of computing a kernel value.
In the primal cas (without the kernel trick):
Computing the solution costs O($mN^2 + N^3$) and predicting one point costs O(N)
In the dual case (already knowing the value of the scalar product in H):
Computing the solution costs O($m^2 + m^3$) and predicting one point costs O(N)
When the feature space is not so big it is generally better to solve the primal problem but as it increases, the dual cas becomes more avantageous.
** Theorical garantees **

A) Kernel ridge regressions:

Under the hypothesis that there exist $r > 0$ such that $K(x, x') \leq r^2$ and $|f(x)| \leq r\Gamma$
For any $\delta > 0$, with probability at least $1 - \delta$ we have for all h:

$$R(h) \leq \hat{R}(h) + \frac{8r^2\Gamma^2}{\sqrt(n)} \left(1 + \frac{1}{2}\sqrt{\frac{log(1/\delta)}{2}}\right)$$

where m is the number of points

B) Lasso regression

Assuming that there exist r such that for all x in X $||x||_\infty \leq r$ and $|f(x)| \leq r\Gamma$
For any $\delta > 0$, with probability at least $1 - \delta$ we have fo all h:

$$R(h) \leq \hat{R}(h) + \frac{8r^2\Gamma^2}{\sqrt(n)} \left(\sqrt(log(2N)) + \frac{1}{2}\sqrt{\frac{log(1/\delta)}{2}}\right)$$

where N is the dimension of X and m the number of points

4.3 Random Forests

Random Forest uses the concept of bagging, still it adds to bagged trees a small twist that decorrelates the data. It builds a number of decision trees on bootstrapped training samples. Each time a split is considered, m predictors are chosen from the p predictors (in practice $m \approx \sqrt{(}p))$.
The algorithm [3] is the following:

```
1) for b=1 to number_of_trees:
        - draw a bootsrap sample from the training data
        - repeat recursively until the minimum node size is reached:
            i) select m variables at random from the p variables
            ii) Pick the best variable among the m
            iii) split the node into 2 nodes
2) Return all the trees
```

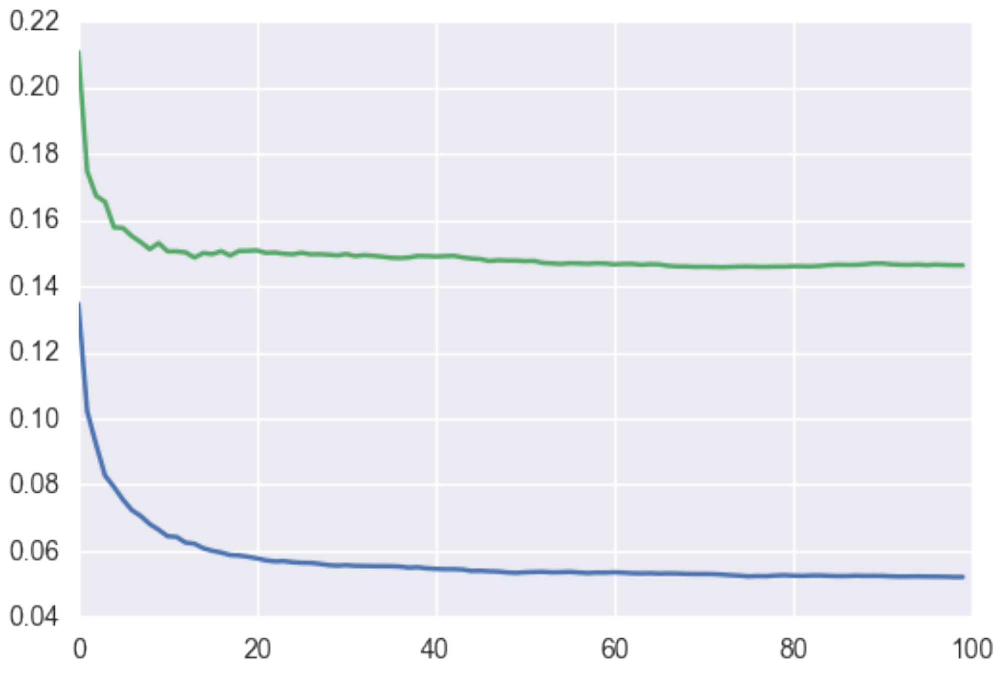
To make a prediction, we take :

$$\hat{f}(x) = \frac{1}{nb\_of\_trees} \sum_{k=1}^{nb\_of\_trees} T_k(x)$$

Hence Random Forest decreases the variance by reducing the odd that a majority of trees use a strong predictor in the top split which decorrelates the predictions. The higher the number of correlated predictors, the more a small number for m will be useful to reduce the variance of the predictions.

```
In [37]: import pylab as pyl
         from time import sleep
         from sklearn.cross_validation import train_test_split
         from sklearn.tree import DecisionTreeRegressor
         from sklearn.metrics import mean_squared_error
         from IPython import display
         from math import sqrt

         X = pd.get_dummies(X_tr)
         X.fillna(X.mean(),inplace = True)

         X=X.values
         Y = y_tr.values

         X_train, X_test, y_train, y_test = train_test_split(
             X, Y, test_size=0.2, random_state=61)



         max_depth = 14 # tree depth
         n_trees = 100 # number of trees
         max_features = 0.4 # number of random features at each cut
         n_samples = X_train.shape[0]
```

```
ts = pyl.arange(n_trees)
training_errors = []
test_errors = []
y_pred_train = np.zeros(len(y_train))
y_pred_test = np.zeros(len(y_test))

for t in range(n_trees):

    train = np.random.choice(range(X_train.shape[0]),X_train.shape[0])
    clf = DecisionTreeRegressor(max_features = max_features,\
                                max_depth = max_depth)
    clf.fit(X_train[train,:],y_train[train])
    pyl.clf()
    y_pred_train = (y_pred_train*t + clf.predict(X_train))/(t+1)
    y_pred_test = (y_pred_test*t + clf.predict(X_test))/(t+1)
    training_error = sqrt(mean_squared_error(y_pred_train,y_train))
    test_error = sqrt(mean_squared_error(y_pred_test,y_test))
    training_errors.append(training_error)
    test_errors.append(test_error)
    pyl.plot(ts[:t+1], training_errors[:t+1])
    pyl.plot(ts[:t+1], test_errors[:t+1])
    display.clear_output(wait=True)
    if(t!= n_trees -1):
        display.display(plt.gcf())
    sleep(.001)
```



The plot shows that :

- Random Forest avoids over fitting by its variance reduction and its random choices
- it converges really fast : 40 trees would be enough to get a precision close to the optimal precision that we could get

Two principles were taken into account when using Random Forest regressors :

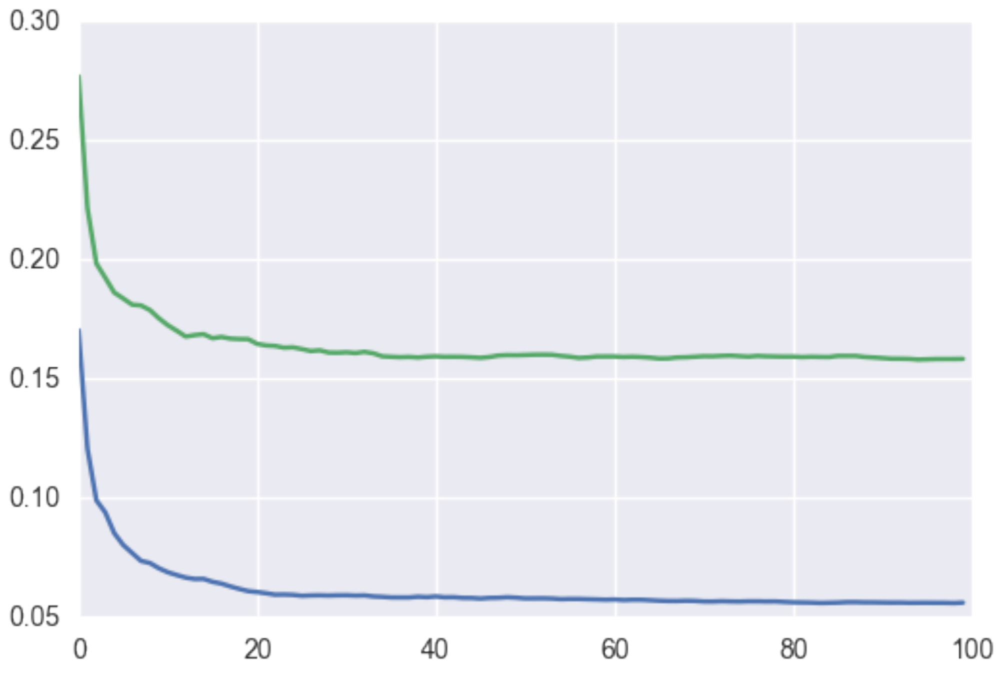
- ** Importance of m the number of predictors picked at random at each step **: when the number of variables is relatively large (which is the case for the considered task), small values of m do not give good results as the the chance to pick the relevant variables at each split can be small. Still, Random Forest is not deeply affected by noise variables even if they are very frequent.

- ** Avoiding overfitting ** : even though Random Forest approximates the expectation, it can overfit.

The first phenomenon is depicted in the following plots. When changing `max_features` from 0.4 to 0.05 we can see that the test error stabilizes on a higher level.

```
In [38]: import pylab as pyl
         from time import sleep
         from sklearn.cross_validation import train_test_split
         from sklearn.tree import DecisionTreeRegressor
         from sklearn.metrics import mean_squared_error
         from IPython import display
         from math import sqrt

         X = pd.get_dummies(X_tr)
         X.fillna(X.mean(),inplace = True)

         X=X.values
         Y = y_tr.values

         X_train, X_test, y_train, y_test = train_test_split(
             X, Y, test_size=0.2, random_state=61)



         max_depth = 14 # tree depth
         n_trees = 100 # number of trees
         max_features = 0.05 # number of random features at each cut
         n_samples = X_train.shape[0]

         ts = pyl.arange(n_trees)
         training_errors = []
         test_errors = []
         y_pred_train = np.zeros(len(y_train))
         y_pred_test = np.zeros(len(y_test))
```

```
for t in range(n_trees):

    train = np.random.choice(range(X_train.shape[0]),X_train.shape[0])
    clf = DecisionTreeRegressor(max_features = max_features)
    clf.fit(X_train[train,:],y_train[train])
    pyl.clf()
    y_pred_train = (y_pred_train*t + clf.predict(X_train))/(t+1)
    y_pred_test = (y_pred_test*t + clf.predict(X_test))/(t+1)
    training_error = sqrt(mean_squared_error(y_pred_train,y_train))
    test_error = sqrt(mean_squared_error(y_pred_test,y_test))
    training_errors.append(training_error)
    test_errors.append(test_error)
    pyl.plot(ts[:t+1], training_errors[:t+1])
    pyl.plot(ts[:t+1], test_errors[:t+1])
    display.clear_output(wait=True)
    if(t!= n_trees -1):
        display.display(plt.gcf())
    sleep(.001)
```



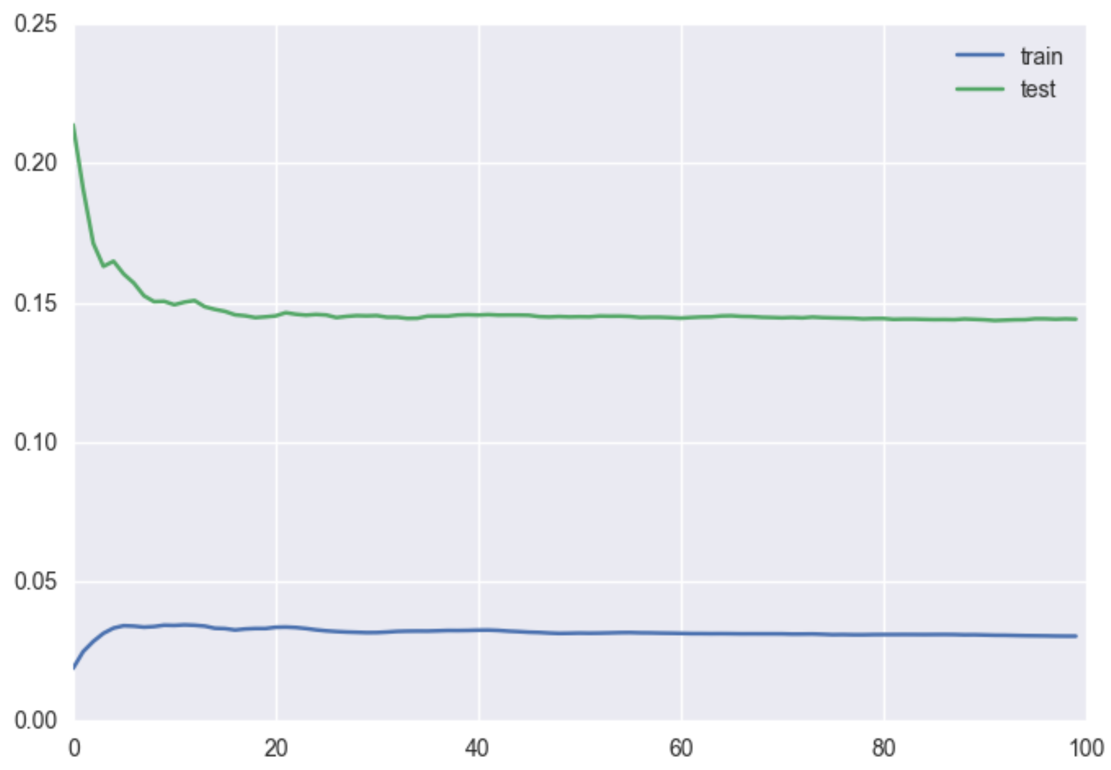
In practice Random Forest gave us good results, still, it appeared that some of the outstanding values (both large and small) were causing some problems. Nonetheless, we used the `RandomForestRegressor` method of `scikit-learn` when combining models in order to reduce the error.

4.4 Boosting methods

Boosting is a very powerful learning technics that is first focused on classifications problems but that can be extend to regressions. The main idea of boosting is to combine several algorithms with a weak power of prediction in order to obtain a much more powerful model. Here we used only tree based boosting algorithms, this means that our weak learners are decision trees. Indeed decision trees have good interpretability but they have a very high variance, which makes them poor predicators. With random forests we have already seen a method to create a powerful predicator by combining decision trees. However boosting methods are really different from the previous ones.
One of the first an most famous boosting algorithm is called AdaBoost. The idea behind the AdaBoost method is to give weights to our training samples and apply our weak learners on updated weighted samples. Indeed, for each boosting round, we apply weak learner on our weighted samples then we update the weights according to the error of prediction of each sample. - If the prediction error is small on this sample, we deacrease his weight for the next boosting round. - If the prediction error is high on this sample, we increase his weight for the next boosting round.
It allows the learners to focused on the worst predicated samples with the boosting iterations. The final result is a linear combination of the predicators obtained at each rounds. The coefficient of this linear combination are chosen to give more credit to the more precise predicators.
We coded the Adaboost.RT version [4] bellow :

```
In [33]: X = pd.get_dummies(X_tr)
         X.fillna(X.mean(),inplace = True)

         X=X.values
         Y = y_tr.values

         X_train, X_test, y_train, y_test = train_test_split(
             X, Y, test_size=0.2, random_state=61)



         max_depth = 14 # tree depth
         n_trees = 100 # number of trees
         max_features = 0.2 # number of random features at each cut
         n_samples = X_train.shape[0]
         phi = 0.005 # error threshold
         weight = np.ones(n_samples)/n_samples
         n = 2


         ts = pyl.arange(n_trees)
         training_errors = []
         test_errors = []
         y_pred_train = np.zeros(len(y_train))
         y_pred_test = np.zeros(len(y_test))
         Sum = 0
```

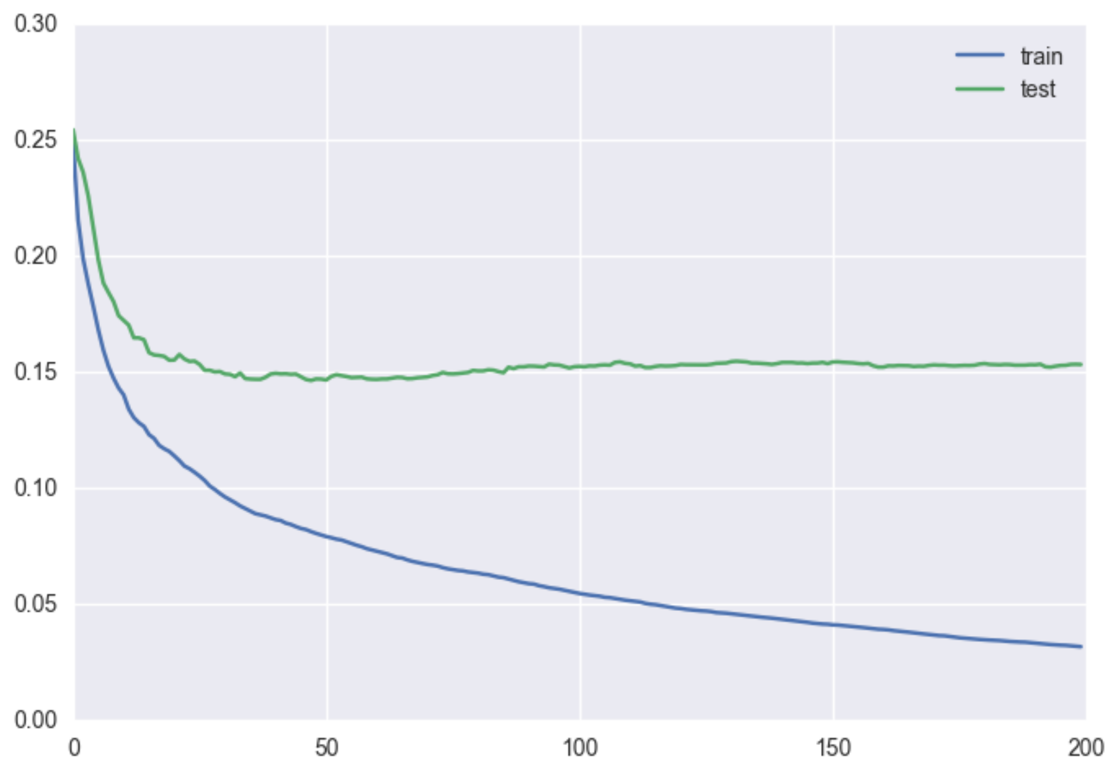
```
for t in range(n_trees):

    reg = DecisionTreeRegressor(max_features = max_features,\
                                max_depth = max_depth)
    #call the weak learner with current weight
    reg.fit(X_train,y_train,sample_weight = weight)
    pyl.clf()
    y_p_train = reg.predict(X_train)
    #determinate the absolute relative error
    #of each training predictions
    ARE = np.abs((y_p_train - y_train[:,0])/y_train[:,0])
    #er is the fraction of weight with ARE higher
    #than the threshold
    er = np.sum((ARE>phi)*weight)
    beta = er**n
    #We update the weights
    #decreasing the weights of the well predicted samples
    #by a factor beta < 1
    weight = weight + ((ARE < phi)*(beta-1))
    #normalise the weights
    weight /= sum(weight)
    #update new predicions values with coefficients depending
    #on beta
    y_pred_test = \
    (y_pred_test*Sum + np.log(1/beta)*reg.predict(X_test))\
    /(Sum + np.log(1/beta))

    y_pred_train = \
    (y_pred_train*Sum + np.log(1/beta)*y_p_train)\
    /(Sum + np.log(1/beta))
    Sum += np.log(1/beta)

    training_error = \
    sqrt(mean_squared_error(y_pred_train,y_train))

    test_error = sqrt(mean_squared_error(y_pred_test,y_test))
    training_errors.append(training_error)
    test_errors.append(test_error)
    pyl.plot(ts[:t+1], training_errors[:t+1],label = 'train')
    pyl.plot(ts[:t+1], test_errors[:t+1],label = 'test')
    pyl.legend()
    display.clear_output(wait=True)
    if(t!= n_trees -1):
        display.display(plt.gcf())
    sleep(.001)
```



We can observe on our data a clear improvement of the test error with the boosting iterations.
An other famous and common used techniques is gradient boosting. Gradient boosting is based on decision trees, at the i-th step the boosting algorithm outputs the predicators : $f_i = f_{i-1} + T_i$ where T_i is a decision tree that targets $-\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}_{|f=f_{i-1}}$ where L is the objective loss function, here $L(x,y) = \frac{1}{2}(y-x)^2$.
We implemented a basic version of gradient boosting algorithm (Algorithm 10.3 from "The Element of Statistical Learning")[3] with the squared loss function. In the case of the squared loss function the decision tree T_i targets $y_i - f_{i-1}$.

```
In [34]: X = pd.get_dummies(X_tr)
         X.fillna(X.mean(),inplace = True)

         X=X.values
         Y = y_tr.values

         X_train, X_test, y_train, y_test = train_test_split(
             X, Y, test_size=0.2, random_state=7)



         max_depth = 2 # tree depth
         n_trees = 200 # number of trees
```

```python
max_features = 0.9 # number of random features at each cut
n_samples = X_train.shape[0]



ts = pyl.arange(n_trees)
training_errors = []
test_errors = []
y_pred_train = np.zeros(len(y_train))
y_pred_test = np.zeros(len(y_test))


for t in range(n_trees):

    reg = DecisionTreeRegressor(max_features = max_features,\
                                max_depth = max_depth)

    reg.fit(X_train,y_train[:,0]-y_pred_train)
    pyl.clf()
    y_pred_test += reg.predict(X_test)

    y_pred_train += reg.predict(X_train)

    training_error = sqrt(mean_squared_error(y_pred_train,y_train))

    test_error = sqrt(mean_squared_error(y_pred_test,y_test))
    training_errors.append(training_error)
    test_errors.append(test_error)
    pyl.plot(ts[:t+1], training_errors[:t+1],label = 'train')
    pyl.plot(ts[:t+1], test_errors[:t+1],label = 'test')
    pyl.legend()
    display.clear_output(wait=True)
    if(t!= n_trees -1):
        display.display(plt.gcf())
    sleep(.001)
```

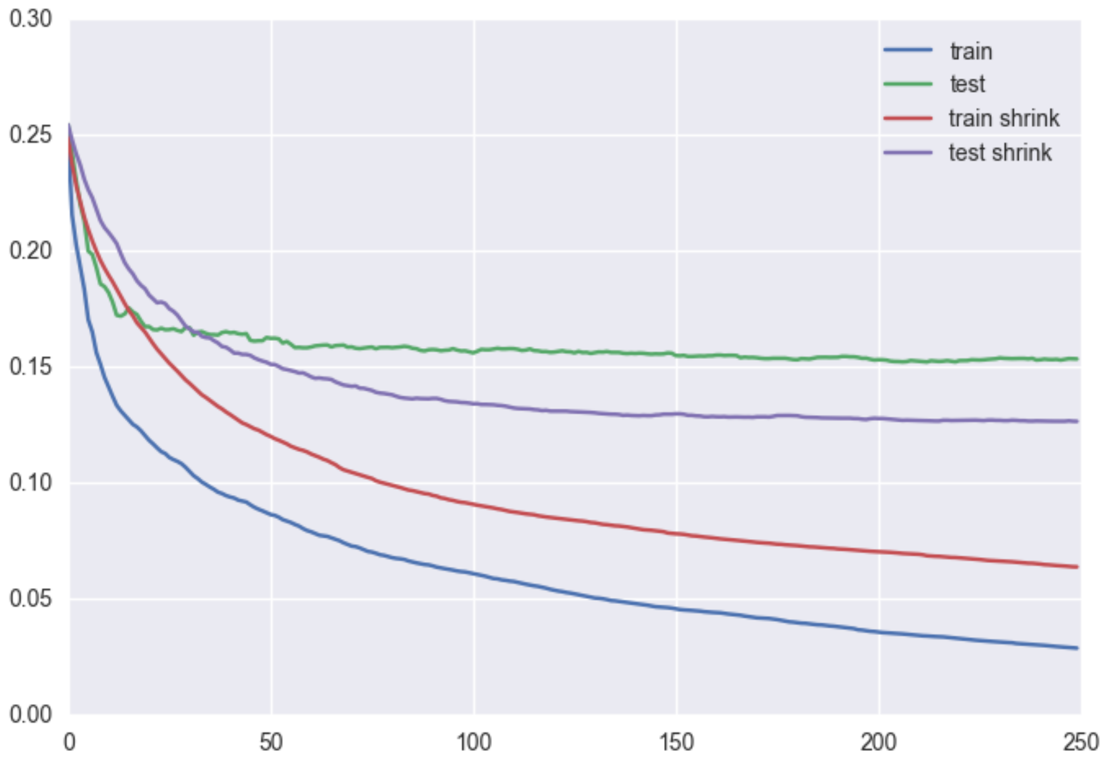


We see that first the test error decrease a lot with the boosting iterations. However it increases when the boosting iterations are too high. Indeed the training error decrease quickly, and fitting the training targets very well often leads to overfitting. To prevent overfitting one method is call shrinkage, when updating the predicator , we put a shrinkage coefficient ν for the new tree. $f_i = f_{i-1} + \nu\, T_i$.

```
In [35]: X = pd.get_dummies(X_tr)
         X.fillna(X.mean(),inplace = True)

         X=X.values
         Y = y_tr.values

         X_train, X_test, y_train, y_test = train_test_split(
             X, Y, test_size=0.2, random_state=7)



         max_depth = 2 # tree depth
         n_trees = 250 # number of trees
         max_features = 0.9 # number of random features at each cut
         n_samples = X_train.shape[0]
         shrink = 0.2
```

```
ts = pyl.arange(n_trees)
training_errors = []
test_errors = []
training_errors_sh = []
test_errors_sh = []
y_pred_train_sh = np.zeros(len(y_train))
y_pred_test_sh = np.zeros(len(y_test))
y_pred_train = np.zeros(len(y_train))
y_pred_test = np.zeros(len(y_test))

for t in range(n_trees):

    reg = DecisionTreeRegressor(max_features = max_features,\
                                max_depth = max_depth)

    reg.fit(X_train,y_train[:,0]-y_pred_train)
    reg_sh = DecisionTreeRegressor(max_features = max_features,\
                                max_depth = max_depth)

    reg_sh.fit(X_train,y_train[:,0]-y_pred_train_sh)
    pyl.clf()
    y_pred_test += reg.predict(X_test)

    y_pred_train += reg.predict(X_train)

    if t == 0 :
        y_pred_test_sh += reg_sh.predict(X_test)

        y_pred_train_sh += reg_sh.predict(X_train)
    else :
        y_pred_test_sh += shrink*reg_sh.predict(X_test)

        y_pred_train_sh += shrink*reg_sh.predict(X_train)

    training_error = sqrt(mean_squared_error(y_pred_train,y_train))

    test_error = sqrt(mean_squared_error(y_pred_test,y_test))
    training_errors.append(training_error)
    test_errors.append(test_error)


    training_error_sh = sqrt(mean_squared_error(y_pred_train_sh,y_train))
    test_error_sh = sqrt(mean_squared_error(y_pred_test_sh,y_test))
    training_errors_sh.append(training_error_sh)
    test_errors_sh.append(test_error_sh)

    pyl.plot(ts[:t+1], training_errors[:t+1],label = 'train')
    pyl.plot(ts[:t+1], test_errors[:t+1],label = 'test')
```

```
        pyl.plot(ts[:t+1], training_errors_sh[:t+1],label = 'train shrink')
        pyl.plot(ts[:t+1], test_errors_sh[:t+1],label = 'test shrink')
        pyl.legend()
        display.clear_output(wait=True)
        if(t!= n_trees -1):
            display.display(plt.gcf())
        sleep(.001)
```

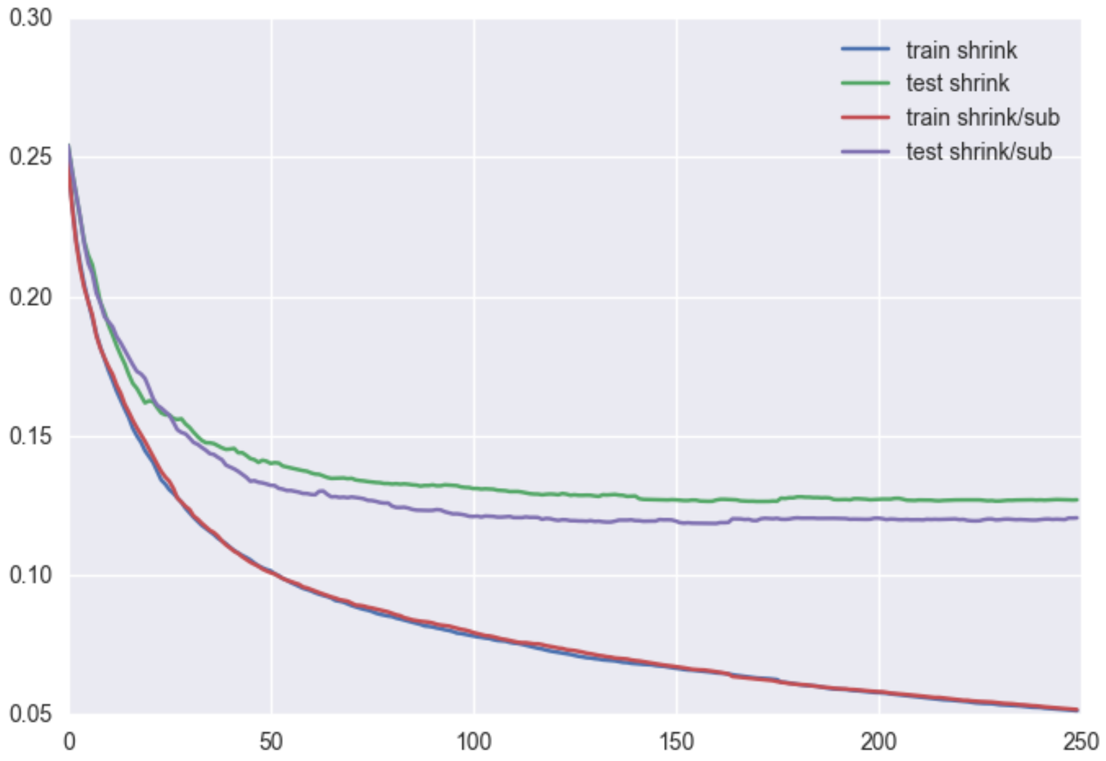


We see a clear improvement of the gradient boosting with shrinkage regularization.
An other method to improve regularization is subsampling. It consists in using only a part of the training data to build new trees at each iterations of the boosting procedure.

```
In [41]: X = pd.get_dummies(X_tr)
         X.fillna(X.mean(),inplace = True)

         X=X.values
         Y = y_tr.values

         X_train, X_test, y_train, y_test = train_test_split(
             X, Y, test_size=0.2, random_state=7)
```

```
max_depth = 2 # tree depth
n_trees = 250 # number of trees
max_features = 0.9 # number of random features at each cut
n_samples = X_train.shape[0]
shrink = 0.3
subsample = 0.9

ts = pyl.arange(n_trees)
training_errors = []
test_errors = []
training_errors_sp = []
test_errors_sp = []
y_pred_train_sp = np.zeros(len(y_train))
y_pred_test_sp = np.zeros(len(y_test))
y_pred_train = np.zeros(len(y_train))
y_pred_test = np.zeros(len(y_test))

for t in range(n_trees):

    reg = DecisionTreeRegressor(max_features = max_features,\
                                max_depth = max_depth)
    sample = np.random.choice(X_train.shape[0],int(subsample*X_train.shape[0]),\
                              replace = False)

    reg.fit(X_train,y_train[:,0]-y_pred_train)

    reg_sp = DecisionTreeRegressor(max_features = max_features,\
                                max_depth = max_depth)

    reg_sp.fit(X_train[sample],y_train[sample,0]-y_pred_train_sp[sample])
    pyl.clf()

    if t == 0 :
        y_pred_test_sp += reg_sp.predict(X_test)

        y_pred_train_sp += reg_sp.predict(X_train)

        y_pred_test += reg.predict(X_test)

        y_pred_train += reg.predict(X_train)
    else :
        y_pred_test_sp += shrink*reg_sp.predict(X_test)

        y_pred_train_sp += shrink*reg_sp.predict(X_train)

        y_pred_test += shrink*reg.predict(X_test)

        y_pred_train += shrink*reg.predict(X_train)
```

```
training_error = sqrt(mean_squared_error(y_pred_train,y_train))
test_error = sqrt(mean_squared_error(y_pred_test,y_test))
training_errors.append(training_error)
test_errors.append(test_error)


training_error_sp = sqrt(mean_squared_error(y_pred_train_sp,y_train))
test_error_sp = sqrt(mean_squared_error(y_pred_test_sp,y_test))
training_errors_sp.append(training_error_sp)
test_errors_sp.append(test_error_sp)

pyl.plot(ts[:t+1], training_errors[:t+1],label = 'train shrink')
pyl.plot(ts[:t+1], test_errors[:t+1],label = 'test shrink')
pyl.plot(ts[:t+1], training_errors_sp[:t+1],label = 'train shrink/sub')
pyl.plot(ts[:t+1], test_errors_sp[:t+1],label = 'test shrink/sub')
pyl.legend()
display.clear_output(wait=True)
if(t!= n_trees -1):
    display.display(plt.gcf())
sleep(.001)
```



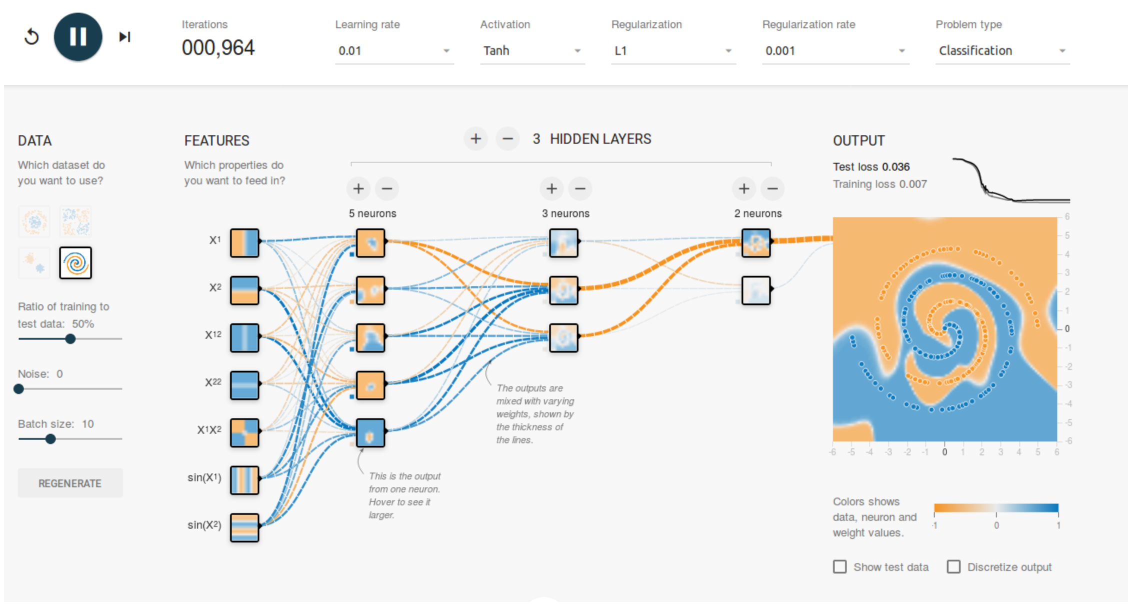
We can observe again an improvement of the test error when combining shrinkage and subsampling. Cross validation should be used to find the optimal hyper-paramters for *shrink* and *subsample*.
In practice we use the *xgboost* library, which provides a very fast implementation of gradient boosting with a lot more parameters.

4.5 Multi-Layer Perceptron

We will brefiely describe what is a multi-layer perceptron and provide an basic implementation on our problem using tensorflow.
First we should describe what a perceptron is. Formally a perceptron is a function that map a vector $x \in \mathbb{R}^d$ to $f(x) = g(W \cdot x + b)$. Where W is call the weight, b the bias and g is an activation function, typically tanh.
In a multi-layer perceptron each layer is constituted by several neurons which are simple perceptrons with their own weights and bias. For the layer i with m neurons, and an input vector $x \in \mathbb{R}^d$ the output is still $f(x) = g(W_i \cdot x + b_i)$ but with W_i the weight matrix in $\mathbb{R}^{m\ x\ d}$ and the bias in $\mathbb{R}^m$.
Each layers are connected by considering the input x equals to the output of the layer before him.
Here is a example created with on http://playground.tensorflow.org/.
Weights and bias are obtained by gradient descent on the error $L(W_i, b_i) = \sum(outputs - label)^2$.

```
In [38]: Image("nnw.png")
```

```
Out[38]:
```


Iterations 000,964
Learning rate 0.01
Activation Tanh
Regularization L1
Regularization rate 0.001
Problem type Classification

Ratio of training to test data: 50%
Noise: 0
Batch size: 10

Test loss 0.036
Training loss 0.007

The outputs are mixed with varying weights, shown by the thickness of the lines.

This is the output from one neuron. Hover to see it larger.

Colors shows data, neuron and weight values.

In order to do regression and not classification, the activation function for the output layer must be $Id : x \rightarrow x$.
Here is a simple implementation of a Multi-layer perceptron with 2 hidden layers of 500 and 10 neurons with tanh activation function for the hidden layers and Id for the output layer.

```
In [44]: import tensorflow as tf

         regression_graph = tf.Graph()

         with regression_graph.as_default():
             # Create a placeholder of dimension `[None, 302]`, containing float32
             # `None` means that the first dimension can be of any length
             # This placeholder will be fed with training data
             train_house =  tf.placeholder(tf.float32, shape=(None, 302))

             # Create a placeholder of dimension `[None, 1]`, containing float32
             # to be fed with the labels
             label =  tf.placeholder(tf.float32, shape=(None,1))

             # Declare model parameters, ie. a tensor containing weights
             # of dimension `[302, 1]` and a tensor containing the bias
             # parameters of dimension `[1]`.
             # Initialise these tensors with random close to 0
             W_hidden =  tf.Variable(tf.random_normal([302, 500], mean=0, stddev=0.1))
             b_hidden =  tf.Variable(tf.zeros([500]))

             W_hidden2 =  tf.Variable(tf.random_normal([500, 10], mean=0, stddev=0.1))
             b_hidden2 =  tf.Variable(tf.zeros([10]))

             W_out =  tf.Variable(tf.random_normal([10, 1], mean=0, stddev=0.1))
             b_out =  tf.Variable(tf.zeros([1]))

             outputs = tf.nn.tanh(tf.matmul(train_house,W_hidden) + b_hidden)
             outputs =  tf.nn.tanh(tf.matmul(outputs,W_hidden2) + b_hidden2)
             outputs = tf.matmul(outputs,W_out) + b_out


             loss = tf.reduce_mean(tf.squared_difference(outputs, label))



             # Optimization step with learning rate = 0.01
             train_step = tf.train.GradientDescentOptimizer(0.01).minimize(loss)

             sqr = tf.sqrt(loss)
```

```
In [45]: # Size of the mini-batch
         mini_batch_size = 200
         # Number of SGD steps
         n_steps = 500
```

```
Y = y_tr.values

X_train, X_test, y_train, y_test = train_test_split(
    X, Y, test_size=0.2, random_state=8)

training_data = {train_house: X_train, label: y_train}
validation_data = {train_house: X_test, label: y_test}

n_tr = X_train.shape[0]

with tf.Session(graph=regression_graph) as sess:
    tf.global_variables_initializer().run()
    # Training loop

    for step in range(n_steps + 1):
        # Get next mini-batch
        offset = (step * mini_batch_size) \
        % (y_train.shape[0] - mini_batch_size)

        batch_house = X_train[offset:(offset + mini_batch_size), :]
        batch_labels = y_train[offset:(offset + mini_batch_size)]
        feed = {train_house: batch_house, label: batch_labels}
        _, current_loss = sess.run([train_step, loss], feed_dict=feed)
        if step % 50== 0:
            print('Step %d' % step)
            print('....Loss:     %f' % current_loss)
            print('....Loss on train: %f' \
                  % sess.run(sqr, feed_dict=training_data))
            print('....Loss on validation: %f' \
                  % sess.run(sqr, feed_dict=validation_data))
```

```
Step 0
...Loss:     147.849808
...Loss on train: 10.836234
...Loss on validation: 10.890841
Step 50
...Loss:     0.160784
...Loss on train: 0.403631
...Loss on validation: 0.382574
Step 100
...Loss:     0.174112
...Loss on train: 0.403365
...Loss on validation: 0.380355
Step 150
...Loss:     0.176428
...Loss on train: 0.403189
```

```
...Loss on validation: 0.379604
Step 200
...Loss:     0.159822
...Loss on train: 0.403046
...Loss on validation: 0.382553
Step 250
...Loss:     0.177834
...Loss on train: 0.402194
...Loss on validation: 0.379541
Step 300
...Loss:     0.172924
...Loss on train: 0.400533
...Loss on validation: 0.377524
Step 350
...Loss:     0.119936
...Loss on train: 0.379161
...Loss on validation: 0.368340
Step 400
...Loss:     0.142665
...Loss on train: 0.359227
...Loss on validation: 0.339513
Step 450
...Loss:     0.124879
...Loss on train: 0.349765
...Loss on validation: 0.315858
Step 500
...Loss:     0.149249
...Loss on train: 0.328206
...Loss on validation: 0.322718
```

On our dataset MLP performs poorly compare to the other algorithms with presented, however it is very different of the previous ones and may be bad globally but it could perform well and samples with which other algorithms have difficulties. This remark will make sense we considering a combination of our models in the next section.

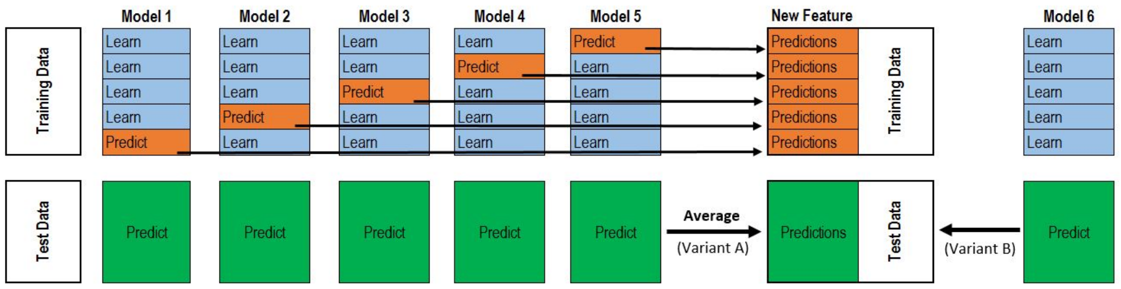
4.6 Combining our models

We've already seen different methods to combine basic models to obtain a more performant one : - **Bagging** : We use random part of the training data (with or without bootstrapping) to train the base predicators and then take the mean of their predictions. That's the strategies of Random Forest or Extra Trees. - **Boosting** : We construct a sequence of base predicators iteratively based on the performances of the previous ones. We saw illustrations with AdaBoost and Gradient Boosting.
Here we are going to introduce **stacking** method. Its principle may remind of the cross-validation procedure. Indeed in stacking, you consider a K-folds partition of your training, and for each base predicator you want to stack in you create the column defined blow-wise with the i th block corresponding to the prediction of the i-th fold of the training data by your current base predicator

trained on the rest of the training data. This gives you a new feature column for the training data. To obtain this new feature column for the test set you can either get K columns corresponding to the prediction on the test set by the base estimator trained on the whole data except one of the fold (for each fold) and then take the average of the K columns, or simply get use the prediction on the test set by the base predicator trained on the whole data.
Here an illustration(citer https://www.kaggle.com/getting-started/18153#post103381) of this method for one base estimator :

```
In [32]: from IPython.display import Image
         Image("stacking.jpeg")
```

Out[32]:



When can repeat this procedure for each estimator we want to use in our stacking method. And we eventually obtain a train and a test matrix of shape *number of samples* * *number of estimators*. We use these new matrix as our new data. And fit a new estimators on the train matrix with the same labels y. And use the new test matrix to get the prediction we wanted.
Here is an implementation of the variante A :

```
In [ ]: import numpy as np
        from sklearn.cross_validation import KFold

        class Ensemble(object):
            def __init__(self, n_folds, stacker, base_models):
                self.n_folds = n_folds
                self.stacker = stacker
                self.base_models = base_models

            def get_params(self,deep =True):
                out = dict()
                out['n_folds']= self.n_folds
                out['stacker'] = self.stacker
                out['base_models'] = self.base_models
                return out

            def fit(self,X,y):
                X = np.array(X)
                y = np.array(y)
                self.folds = list(KFold(len(y), n_folds=self.n_folds,\
                                        shuffle=True, random_state=42))
                S_train = np.zeros((X.shape[0], len(self.base_models)))
                for i, clf in enumerate(self.base_models):
                    for j, (train_idx, test_idx) in enumerate(self.folds):
                        X_train = X[train_idx]
                        y_train = y[train_idx]
                        X_holdout = X[test_idx]
                        # y_holdout = y[test_idx]
                        clf.fit(X_train, y_train)
                        y_pred = clf.predict(X_holdout)[:]
                        S_train[test_idx, i] = y_pred
                        self.stacker.fit(S_train,y)

            def predict(self,X):
                X = np.array(X)
                S_test = np.zeros((X.shape[0], len(self.base_models)))
                for i, clf in enumerate(self.base_models):
                    S_test_i = np.zeros((X.shape[0], len(self.folds)))
                    for j, (train_idx, test_idx) in enumerate(self.folds):
                        S_test_i[:, j] = clf.predict(X)[:]
                    S_test[:, i] = S_test_i.mean(1)
                y_pred = self.stacker.predict(S_test)[:]
                return y_pred
```

The file that gave us the better results on the public and private leaderboards was computed using an ensemble of Random Forest, Extra Trees, Adaboost, Multi layer perceptron, Ridge and Lasso regression, and gradient boosting.
The code we used for the challenge are available here : https://github.com/mathbarre/P3A

References

[1] Hui Zou. The adaptive lasso and its oracle properties. *Journal of the American statistical association*, 101(476):1418–1429, 2006.

[2] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012.

[3] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin, 2001.

[4] Pengbo Zhang and Zhixin Yang. A robust adaboost. rt based ensemble extreme learning machine. *Mathematical Problems in Engineering*, 2015, 2015.