House Prices: Advanced Regression Techniques: Kaggle dataset exercice for the subject Research and Innovation from MSc Mathematical Modelling For Science and Engineering at UAB

Marc Fuster Rullan and Andrés Puy Enfedaque

Abstract

On this report we will solve the Kaggle competition "House Prices: Advanced Regression Techniques:" using Python. The first thing to do is to look into the available dataset and try to understand its most important features. Secondly, we will perform some data transformations in order to prepare our data to finally execute some Machine Learning algorithms to fix our model. Finally, we will predict the SalePrice on the test set and submit the prediction into Kaggle. The score we have is 0.12353 and currently (in the 16th of October of 2018) we are the 1265th of 4089 (Top 31%)

This work has been carried out using the guides available on Kaggle. We have used many of the kernels availabels and we have not copied most of the code. It is a mixture of many kernels and our ideas. The main kernels that we have used are: Pedro Marcelino, Serigne, Bsivavenu and the guide of DanB.

1 Data analysis.

The first thing to do is to load the main libraries needed along the script: (pandas as pd, matplotlib.pyplot as plt, seaborn as sns, numpy as np, scipy.stats, sklearn, scipy and time.

Afterwards, we load the datasats:

```
In [2]: train_df = pd.read_csv('train.csv')
        test_df = pd.read_csv('test.csv')
```

1.1 Dataset features and entries

Now we take a look of the features of the dataset. There are 81 features on the train and 80 on the test because test does not have the SalePrice feature:

```
In [3]: train_df.columns
```

```
Out[3]: Index(['Id', 'MSSubClass', 'MSZoning', 'LotFrontage', 'LotArea', 'Street',
               'Alley', 'LotShape', 'LandContour', 'Utilities', 'LotConfig',
               'LandSlope', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType',
               'HouseStyle', 'OverallQual', 'OverallCond', 'YearBuilt', 'YearRemodAdd',
               'RoofStyle', 'RoofMatl', 'Exterior1st', 'Exterior2nd', 'MasVnrType',
               'MasVnrArea', 'ExterQual', 'ExterCond', 'Foundation', 'BsmtQual',
               'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinSF1',
               'BsmtFinType2', 'BsmtFinSF2', 'BsmtUnfSF', 'TotalBsmtSF', 'Heating',
               'HeatingQC', 'CentralAir', 'Electrical', '1stFlrSF', '2ndFlrSF',
               'LowQualFinSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath',
```

```
       'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'KitchenQual',
       'TotRmsAbvGrd', 'Functional', 'Fireplaces', 'FireplaceQu', 'GarageType',
       'GarageYrBlt', 'GarageFinish', 'GarageCars', 'GarageArea', 'GarageQual',
       'GarageCond', 'PavedDrive', 'WoodDeckSF', 'OpenPorchSF',
       'EnclosedPorch', '3SsnPorch', 'ScreenPorch', 'PoolArea', 'PoolQC',
       'Fence', 'MiscFeature', 'MiscVal', 'MoSold', 'YrSold', 'SaleType',
       'SaleCondition', 'SalePrice'],
      dtype='object')
```

Let's also visualize some data of the dataset. All those features are fully described in the datadescription.txt:

```
In [4]: train_df.head(5)
```

Out[4]:

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

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

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

[5 rows x 81 columns]

1.2 Dataset statistics:

```
In [5]: # Describe basic properties of the dataset
        train_df.describe()
```

Out[5]:

	Id	MSSubClass	LotFrontage	LotArea	OverallQual	\
count	1460.000000	1460.000000	1201.000000	1460.000000	1460.000000	
mean	730.500000	56.897260	70.049958	10516.828082	6.099315	
std	421.610009	42.300571	24.284752	9981.264932	1.382997	
min	1.000000	20.000000	21.000000	1300.000000	1.000000	
25%	365.750000	20.000000	59.000000	7553.500000	5.000000	
50%	730.500000	50.000000	69.000000	9478.500000	6.000000	
75%	1095.250000	70.000000	80.000000	11601.500000	7.000000	
max	1460.000000	190.000000	313.000000	215245.000000	10.000000	

```
             ...         WoodDeckSF  OpenPorchSF  EnclosedPorch    3SsnPorch  \
count        ...        1460.000000  1460.000000    1460.000000  1460.000000
mean         ...          94.244521    46.660274      21.954110     3.409589
std          ...         125.338794    66.256028      61.119149    29.317331
min          ...           0.000000     0.000000       0.000000     0.000000
25%          ...           0.000000     0.000000       0.000000     0.000000
50%          ...           0.000000    25.000000       0.000000     0.000000
75%          ...         168.000000    68.000000       0.000000     0.000000
max          ...         857.000000   547.000000     552.000000   508.000000

       ScreenPorch     PoolArea       MiscVal       MoSold       YrSold  \
count  1460.000000  1460.000000   1460.000000  1460.000000  1460.000000
mean     15.060959     2.758904     43.489041     6.321918  2007.815753
std      55.757415    40.177307    496.123024     2.703626     1.328095
min       0.000000     0.000000      0.000000     1.000000  2006.000000
25%       0.000000     0.000000      0.000000     5.000000  2007.000000
50%       0.000000     0.000000      0.000000     6.000000  2008.000000
75%       0.000000     0.000000      0.000000     8.000000  2009.000000
max     480.000000   738.000000  15500.000000    12.000000  2010.000000

[8 rows x 38 columns]
```

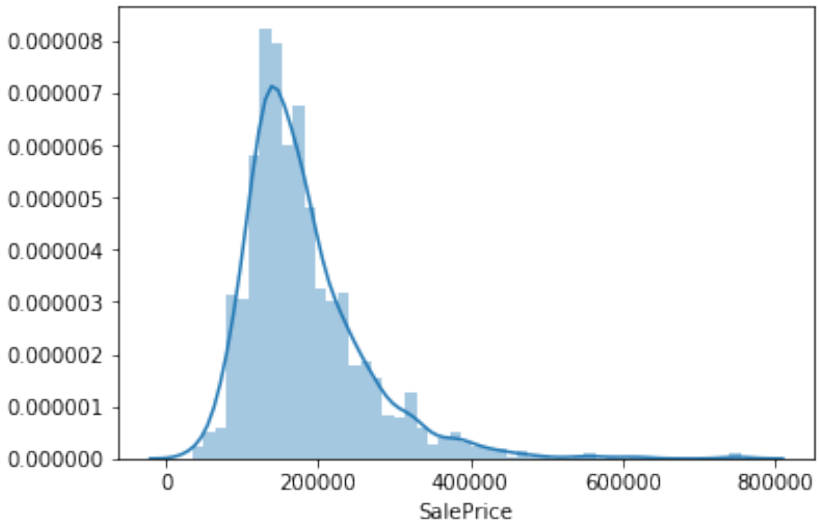
1.3 Let's focus on the score variable: SalePrice

```
In [6]: train_df['SalePrice'].describe()
```

```
Out[6]: count      1460.000000
        mean     180921.195890
        std       79442.502883
        min       34900.000000
        25%      129975.000000
        50%      163000.000000
        75%      214000.000000
        max      755000.000000
        Name: SalePrice, dtype: float64
```

```
In [7]: sns.distplot(train_df['SalePrice'])
```

```
Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd1ff6503c8>
```



As we can see, the score feature 'SalePrice' is morealess a Gaussian with a long tail on the higher prices. This is completely normal since the mean of the distribution is 180921 USD, and there are no houses with negative price (or in fact smaller than 34900 USD), but there are houses with higher price than 2 x 180921 USD = 361842 USD.

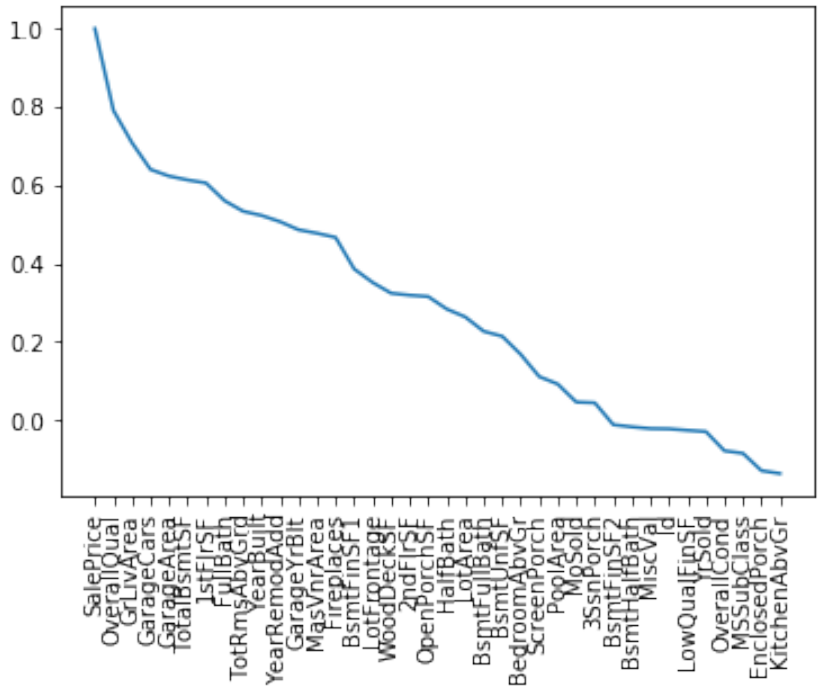
1.4 Correlations among variables

Now we will see which are the relationships among variables, we expect some variables to be very correlated such as Garage cars and Garage Area (because the more area, the more cars you can enter). We can obtain the correlation matrix by simply running:

```
train_df.corr()
```

Plotting this matrix is also very easy and gives a general idea of the correlations among variables. In fact, we can see that garage cars and garage area are very correlated (>0.8). On the script you can find the matrix. However, on this report we will not add the plot of the matrix because we want to focus on the correlations with 'SalePrice'

```
In [10]: plt.plot(corrmat['SalePrice'].sort_values(ascending=False))
         plt.xticks(rotation=90);
```



This correlation number (Pearson's r) gives us an idea of the correlation between two features. Positive means that if one is getting larger so does the other. On the other hand, negative means that if one gets bigger the other decreases. Numbers close to 0 means no correlation. We can see some important positive correlations such as OverallQual (0.790982) and GrLivArea (0.708624) and none important negative correlation.

2 Data treatment

2.1 Deleting ID

Id is completely uncorrelated: so let's copy it and drop it. Moreover, we don't want our model to take mistaken decisions such as if ID>2000 then the prize is higher:

```
In [11]: #Save the 'Id' column
         train_ID = train_df['Id']
         test_ID = test_df['Id']

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

2.2 Missing data and Categorical data

There are variables which have some features incomplete. We could do many things in the missing data case: The easiest solution is to delete all the features which have some missing data. Of course, if there is a low percentile of missing data we could be neglecting important data.

What we will do is to delete the features which have a significant incomplete data and then decide what to do with the not deleted data.

```
In [12]: # First of all we will clean all the dataset: both the test and train.
         # We will join them and then separate them once the data cleaning has finished

         ntrain=len(train_df)

         #Now we need also to drop SalePrice because test df does not have it
         train_y=train_df['SalePrice']
         train_df=train_df.drop(['SalePrice'],axis=1)

         all_data=pd.concat(objs=[train_df, test_df], axis=0).reset_index(drop=True)
```

2.2.1 First of all let's encode the categorical data

As Serigne tutorial indicates, firs let's transform some numerical variables which in fact are categorical

```
In [13]: #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)
```

2.2.2 Now let's use the function LabelEncoder to automatically encode all data which is string type

We have two options: to drop all the features which are categorical or to encode the categorical features. The first one is much simpler but you can be missing information. The second one is more complex but you may gain some accuracy.

We will choose to encode the categorical data we will follow: "Using Categorical Data with One Hot Encoding" from DanB.

```
In [15]: from sklearn.preprocessing import LabelEncoder

         cols=[] #list with the str features

         for i in all_data:
             if type(all_data[i][4])==str: #the number 4 is random. Works any number.
                 cols.append(i)
                 print(i)

         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))
```

```
MSSubClass, MSZoning, Street, LotShape, LandContour, Utilities, LotConfig, LandSlope,
Neighborhood, Condition1, Condition2, BldgTypeHouseStyle, OverallCond,
RoofStyle, RoofMatl, Exterior1st, Exterior2nd, MasVnrType, ExterQual, ExterCond,
Foundation, BsmtQual, BsmtCond, BsmtExposure, BsmtFinType1, BsmtFinType2, Heating, HeatingQC,
CentralAir, Electrical, KitchenQual, Functional, FireplaceQuGarageType, GarageFinish,
GarageQual, GarageCond, PavedDrive, MoSold, YrSold, SaleType, SaleCondition
Shape all_data: (2919, 79)
```

To see how LabelEncoder works let's print the structure of a categorical feature before and after the transformation:

```
In [14]: #let's see an example of how LabelEncoder works
         all_data['Neighborhood'].head(10)
```

```
Out[14]: 0    CollgCr
         1    Veenker
         2    CollgCr
         3    Crawfor
         4    NoRidge
         5    Mitchel
         6    Somerst
         7     NWAmes
         8    OldTown
         9    BrkSide
         Name: Neighborhood, dtype: object
```

After the transformation:

```
In [16]: all_data['Neighborhood'].head(10)
```

```
Out[16]: 0     5
         1    24
         2     5
         3     6
         4    15
         5    11
         6    21
         7    14
         8    17
         9     3
         Name: Neighborhood, dtype: int64
```

As we can see, it automatically assigns a random to each possible value of the feature. This is not the optimal process for all categorical features. For example, imagine we had a feature whose options are Good, Regular and Bad. The encoding should be proportional to the quality (Good-3 Regular-2 Bad-1, or, Good-1 Regular-2 Bad-3) but never in a different order such as Good-2 Regular-1 Bad-3. In this way the decision trees or any other models can identify that they are correlated. For the Neighborhood example we could try to sort the neighborhoods accordingly to their average SalePrice.

2.2.3 Now let's see the missing data ratio

```
In [17]: total = all_data.isnull().sum().sort_values(ascending=False)
         percent = (all_data.isnull().sum()/all_data.isnull().count()).sort_values(ascending=False)
         missing_data = pd.concat([total, percent], axis=1, keys=['Total', 'Percent'])
         #missing_data = pd.concat([total, percent])
         #print(missing_data['Total']>0.1)
         print(missing_data)
```

```
               Total   Percent
PoolQC          2909  0.996574
MiscFeature     2814  0.964029
Alley           2721  0.932169
Fence           2348  0.804385
LotFrontage      486  0.166495
GarageYrBlt      159  0.054471
MasVnrArea        23  0.007879
BsmtHalfBath       2  0.000685
BsmtFullBath       2  0.000685
GarageArea         1  0.000343
BsmtFinSF1         1  0.000343
GarageCars         1  0.000343
TotalBsmtSF        1  0.000343
BsmtUnfSF          1  0.000343
BsmtFinSF2         1  0.000343
Exterior2nd        0  0.000000
Exterior1st        0  0.000000
MasVnrType         0  0.000000
SaleCondition      0  0.000000
...              ...       ...
LowQualFinSF       0  0.000000
2ndFlrSF           0  0.000000
1stFlrSF           0  0.000000
Electrical         0  0.000000
CentralAir         0  0.000000
SaleType           0  0.000000
MSSubClass         0  0.000000

[79 rows x 2 columns]
```

Following the Pmarcelino guide, we will delete the features which have more than 15% of entries missing (until LotFrontage) because filling them could be worse than the information we are loosing.

```
In [18]: # We also must delete it from test

         all_data = all_data.drop((missing_data[missing_data['Percent'] > 0.15]).index,1)
```

2.2.4 Now let's fill the ones that have a lower missing ratio with the average of each feature

```
In [19]: # the data we want to fill is
         filling_features=percent[(percent<0.15) & (percent>0.0001)]
```

```
print(filling_features.index)
```

```
Index(['GarageYrBlt', 'MasVnrArea', 'BsmtHalfBath', 'BsmtFullBath',
       'GarageArea', 'BsmtFinSF1', 'GarageCars', 'TotalBsmtSF', 'BsmtUnfSF',
       'BsmtFinSF2'],
      dtype='object')
```

Those are the features that we will fill with the mean of category.

```
In [20]: for i in filling_features.index:
             all_data[i].fillna((all_data[i].mean()), inplace=True)
```

```
In [21]: #Just to make sure we have not neglected all information about the pool
         all_data['PoolArea'].describe()
```

```
Out[21]: count    2919.000000
         mean        2.251799
         std        35.663946
         min         0.000000
         25%         0.000000
         50%         0.000000
         75%         0.000000
         max       800.000000
         Name: PoolArea, dtype: float64
```

```
In [22]: all_data['FullBath']
```

```
Out[22]: 0       2
         1       2
         2       2
         3       1
         4       2
         5       1
                ..
         2914    1
         2915    1
         2916    1
         2917    1
         2918    2
         Name: FullBath, Length: 2919, dtype: int64
```

Let's do some Feature engineering. We are creating some variables both in the training and test set which may be more correlated to SalePrice

```
In [23]: all_data["TotalBath"] = all_data["FullBath"] + (0.5 * all_data["HalfBath"])
         # Total SF for 1st + 2nd floors
         all_data["AllFlrsSF"] = all_data["1stFlrSF"] + all_data["2ndFlrSF"]
         # Total SF for porch
         all_data["AllPorchSF"] = all_data["OpenPorchSF"] + all_data["EnclosedPorch"] + \
         all_data["3SsnPorch"] + all_data["ScreenPorch"]
```

2.2.5 Separate with Train and Test and Drop SalePrice

```
In [24]: train_df = all_data[:ntrain]
         test_df = all_data[ntrain:]
```

Let's also see if the feature engineering has improved something using the correlation matrix.

```
In [25]: train_df['SalePrice']=train_y

         corrmat = train_df.corr()
         print(corrmat['SalePrice'].sort_values(ascending=False))

         train_df=train_df.drop(['SalePrice'],axis=1)
```

```
SalePrice        1.000000
OverallQual      0.790982
AllFlrsSF        0.716883
GrLivArea        0.708624
GarageCars       0.640409
GarageArea       0.623431
TotalBsmtSF      0.613581
1stFlrSF         0.605852
TotalBath        0.597966
FullBath         0.560664
TotRmsAbvGrd     0.533723
YearBuilt        0.522897
YearRemodAdd     0.507101
MasVnrArea       0.475210
GarageYrBlt      0.471062
Fireplaces       0.466929
BsmtFinSF1       0.386420
Foundation       0.382479
WoodDeckSF       0.324413
2ndFlrSF         0.319334
OpenPorchSF      0.315856
HalfBath         0.284108
LotArea          0.263843
CentralAir       0.251328
Electrical       0.234716
PavedDrive       0.231357
BsmtFullBath     0.227122
RoofStyle        0.222405
BsmtUnfSF        0.214479
SaleCondition    0.213092
                   ...
LandContour      0.015453
BsmtCond         0.015058
BsmtFinType2     0.008041
Condition2       0.007513
GarageQual       0.006861
MoSold           0.005884
```

```
BsmtFinSF2      -0.011378
Utilities       -0.014314
BsmtHalfBath    -0.016844
MiscVal         -0.021190
LowQualFinSF    -0.025606
YrSold          -0.028923
SaleType        -0.054911
LotConfig       -0.067396
OverallCond     -0.077856
BldgType        -0.085591
Heating         -0.098812
BsmtFinType1    -0.103114
EnclosedPorch   -0.128578
KitchenAbvGr    -0.135907
MSZoning        -0.166872
LotShape        -0.255580
BsmtExposure    -0.309043
HeatingQC       -0.400178
GarageType      -0.415283
FireplaceQu     -0.459605
GarageFinish    -0.549247
KitchenQual     -0.589189
BsmtQual        -0.620886
ExterQual       -0.636884
Name: SalePrice, Length: 78, dtype: float64
```

As we can see! The second more correlated is "AllFlrsSF 0.716883" which we have created from "1stFlrSF 0.605852" and "2ndFlrSF 0.319334". The same happens with "TotalBath 0.597966" which is made from "FullBath 0.560664" and "HalfBath 0.284108"

3 Select and train the models

3.1 Import Basic ML algorithms

```
In [26]: 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
```

Let's define a function to score the methods on the TRAINING set. The result is not 100% accurate but it may give us some insights.

```
In [27]: n_folds = 5
def rmsle_cv(model):
```

```
    kf = KFold(n_folds, shuffle=True, random_state=42).get_n_splits(train_df.values)
    rmse= np.sqrt(-cross_val_score(model, train_df.values, train_y, scoring="neg_mean _squared_error",
                return(rmse)
```

Define some models

We will use some linear regression models such as Ridge regularization, Lasso regularization and Elastic net regularization which is a combination of Lasso and Ridge. Moreover, we will use some Gradient Boosting techniques. Gradient Boosting is an ensemble of decision trees. More specifically, Gradient Boosting is a boosting technique, which consists on creating decision trees sequentially so as that each decision tree corrects the previous one. We have 3 gradient boosting models: Gboost is the original algorithm, XGB is newer and improved algorithm from 2014 and LGB is faster (but with slightly worse accuracy) algorithm. Nowadays XGB and LGB are widely used while Gboost isn't. In order to choose among LGB or XGB you have to take into consideration whether you prefer accuracy or computational time.

```
In [28]: GBoost = GradientBoostingRegressor(n_estimators=3000, learning_rate=0.05,
                                            max_depth=4, max_features='sqrt',
                                            min_samples_leaf=15, min_samples_split=10,
                                            loss='huber', random_state =5)

In [29]: 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)

In [30]: 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)

In [31]: lasso = make_pipeline(RobustScaler(), Lasso(alpha =0.0005, random_state=1))

In [32]: ENet = make_pipeline(RobustScaler(), ElasticNet(alpha=0.0005, l1_ratio=.9, random_state=3))

In [33]: KRR = KernelRidge(alpha=0.6, kernel='polynomial', degree=2, coef0=2.5)
```

3.2 Scores on the Training set

Create a function to run the model to know the score and the running time

```
In [34]: def runmodel(model):
             tic=time.time()
             score = rmsle_cv(model)
             print(" score: {:.4f} ({:.4f})\n".format(score.mean(), score.std()))
             toc=time.time()
             elapsedtime=np.abs(toc-tic)
             print('The time taken to test the model is: {:.2f} s'
              .format(elapsedtime))
             return(score.mean(),score.std(),elapsedtime)
```

Let's run them

```
In [35]: print('\n Lasso \t')
         lasso_score=runmodel(lasso)

         print('\n Elastic NET \t')
         Enet_score=runmodel(ENet)

         print('\n Kernel Ridge \t')
         KRR_score=runmodel(KRR)

         print('\n Gboost \t')
         GBoost_score=runmodel(GBoost)

         print('\n XGB \t')
         model_xgb_score=runmodel(model_xgb)

         print('\n LGB \t')
         model_lgb_score=runmodel(model_lgb)
```

```
 Lasso
 score: 33843.6765 (7450.4898)

The time taken to test the model is: 1.52 s

 Elastic NET
 score: 33842.1589 (7449.5913)

The time taken to test the model is: 1.06 s

 Kernel Ridge
 score: 405865.6960 (433773.3798)

The time taken to test the model is: 0.56 s

 Gboost
 score: 27707.3091 (5855.4633)

The time taken to test the model is: 27.19 s

 XGB
 score: 25924.8767 (4426.1173)

The time taken to test the model is: 5.28 s

 LGB
 score: 27513.0227 (4494.7961)

The time taken to test the model is: 0.73 s
```

For now XGB is the best model regarding accuracy. Moreover: a computational time of 5.28 seconds is small enough. If the dataset was larger maybe we would have to use LGB.

3.3 Training on the Train and Predictions on the test

```
In [36]: def rmsle(y, y_pred):
             return np.sqrt(mean_squared_error(y, y_pred))

In [37]: def tryontest(model):
             model.fit(train_df, train_y)
             train_pred = model.predict(train_df)
             test_pred = model.predict(test_df) #np.expm1(model.predict(test_df))
             print("The model applied to the training gives an RMSE of: \t")
             print(rmsle(train_y, train_pred))
             print("\n and predicts on the test set:")
             print(test_pred)
             return test_pred

In [38]: test_result_GBoost=tryontest(GBoost)

The model applied to the training gives an RMSE of:
13069.801538809928

 and predicts on the test set:
[122888.27687152 163393.24012631 189314.20297956 ... 156245.82273464
 117537.98856038 205429.76641072]


In [39]: test_result_lgb=tryontest(model_xgb)

The model applied to the training gives an RMSE of:
3627.4498852240968

 and predicts on the test set:
[129628.234 164592.42  190230.47  ... 157857.19  117884.766 215746.08 ]


In [40]: test_result_xgb=tryontest(model_lgb)

The model applied to the training gives an RMSE of:
13778.118779952258

 and predicts on the test set:
[124719.22330243 158157.68373665 180332.27787661 ... 155017.62646328
 125771.77671429 210738.93159773]
```

Once again, XGB seems to be the best model.

3.3.1 Let's create an Stacking model, an average of different models.

```
In [41]: class AveragingModels(BaseEstimator, RegressorMixin, TransformerMixin):
             def __init__(self, models):
                 self.models = models

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

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

                 return self

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

In [42]: averaged_models = AveragingModels(models = (GBoost, model_lgb, model_xgb))  #(ENet, GBoost,

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

 Averaged base models score: 26316.8728 (5034.2467)



In [43]: #test_result_averaged=tryontest(averaged_models)
         averaged_models.fit(train_df.values, train_y)
         averaged_train_pred = averaged_models.predict(train_df.values)
         averaged_pred = averaged_models.predict(test_df.values)

         print("The model applied to the training gives an RMSE of: \t")
         print(rmsle(train_y, averaged_train_pred))
         print("\n and predicts on the test set:")
         print(averaged_pred)

The model applied to the training gives an RMSE of:
8751.873251248486

 and predicts on the test set:
[125745.24484965 162047.78191265 186625.64986873 ... 156373.54556597
 120398.17696656 210638.25871115]
```

4 Select the model and create the submision

```
In [44]: sub = pd.DataFrame()
         sub['Id'] = test_ID
         sub['SalePrice'] = averaged_pred#test_result_xgb
         sub.to_csv('submission.csv', index=False)
```

5 Results and Benchmarks

As the Kaggle competition we are submitting is a Getting Started, we can submit as many times as we want the results and know our score. If it was a real problem, we will not be able to know our score or we would have to wait until the competition is finished to know our result. If this was a real problem we would probably choose XGB as a solution.

5.1 Kaggle score Statistics

In order to know how good a result is, we need to compare it with the results of other people. This is the leaderboard in the 16th of October of 2018.

Current leaderboard in the 16th of October of 2018.

Position	1	5	20	100	1000
Score	0.066283	0.10641	0.10942	0.10985	0.12083

5.2 Model comparison

We have first tried with linear regressions and we have scores greater than 0.19. If we compare it with the decision tree models (scores <0.13), we can determine that linear regression models should not be used for this problem.

Submission benchmark of different models. Train RMSE is the result of a cross validation within the train dataset. Running time is the computational time, in our computer, to train the model and to predict on the test. Finally, Kaggle score is the output of the validation that Kaggle does with its own unknown dataset.

	GBoost	**XGB**	**LGB**	**Averaging: Gboost+XGB+LGB**
Train RMSE	27707	25925	27513	26117
Running time (s)	5.45	0.97	0.20	1.08
Kaggle Score	0.12855	0.12980	0.12498	0.12353

As we can see, The averaging: GBoost+LGB+XGB is the model which gives the best result. However, what is surprising is that XGB scored better than LGB and GBoosting on the training cross validation. However, on the test benchmark they have done worse. The easiest explanation is that the XGB model has overfitted the data.

5.3 Encoding vs Deleting

Once we have selected The averaging: GBoost+LGB+XGB as the best model, let's compare what would have happened if instead of encoding the categorical data, we would have deleted them.

Now the averaging model scores 0.12353 when encoding categorical data. Deleting the categorical features we score 0.13894. We can see that there encoding has increased a lot the prediction