Data Science en pratique et insertion professionelle

Arthur Llau, data scientist chez Safety Line :
arthur.llau@safety-line.fr

Prochain cours les mercredi 7 et 21 Février, ainsi que le 7 et le 21 Mars. Salle habituelle.

Cours à venir :

- Cours 6: Algorithmes et techniques avancés
- Cours 7: Méthode d'amélioration des prédictions (Agrégation, Stacking, Super Learner)
- Cours 8: Introduction au NLP/Text Engineering
- Cours 9: Introduction à la Classification d'images
- Cours 10: Challenge par équipe

Cours 6 : Algorithmes et techniques avancés

Objectif du cours:

- Présentation de quelques algorithmes et implémentations avancés.
- Présenter rapidemment l'importance des features
- TP de mise en pratique (as usual)

1 - Algorithmes et implémentations avancés

1.1 - Extremely Randomized Trees

Original paper (P. Gueurts et al., 2005) : https://link.springer.com/article/10.1007/s10994-006-6226-1 (https://link.springer.com/article/10.1007/s10994-006-6226-1)

Très similaire à l'algorithme des forêts aléatoire, cette algorithme possède toutefois deux différences majeurs:

- L'algorithme des ETR n'utilise pas de procédure de bagging pour construire l'ensemble d'entrainement de chaque arbre. Les arbres apprennent tous sur le jeu de données tout entier.
- Là où dans l'algorithme des RF la séparation de chaque noeud est choisie de manière heuristique : on prend le meilleur noeud possible dans un ensemble aléatoire de variables. L'algorithme des ETR choisit de manière aléatoire la séparation.

Les hyperparamètres des ETR sont similaires, aux différences algorithmique près, à ceux des RF. On trouve une implémentation de cet algorithme dans Sklearn sous le nom de ExtraTrees[Reg/Clf].

1.2 Boosting & XtremBoosting

Les trois algorithmes décrits ci-dessous sont des méthodes de gradient boosting. LightGBM et Xgboost sont des implémentations dites d'extrême boosting, c'est-à-dire qu'elles sont construites de manière à utiliser du mieu possible les ressources computationnelles. Néanmoins, quelques points diffères dans l'algorithme lui même.

Il est important de noté que ces méthodes sont implémentées de manière propre à chacune. Mais elles possèdent également un wrapper scikit-learn.

Un autre atout majeur de ces implémentations est l'utilisation de jeux de validation pour obtenir le nombre d'itérations optimal. À chaque itération on regarde si les performances sur le jeu de validation sont améliorés, sinon on arrête. On prend alors la dernière itération la plus performante. Évidemment, il est possible de choisir un nombre d'itération pour lequel il n'y a d'améliorations. (voir Tp)

1.2.1 XGBoost

Original paper (T.Chen, 2016) :
https://arxiv.org/abs/1603.02754
(https://arxiv.org/abs/1603.02754)

Il existe plusieurs diffèrences notable avec l'implémentation du GBT classique :

- Cette implémentation gère les valeurs manquantes
- Le modèle est régularisé,i.e. on contrôle mieux l'overfitting, ce qui explique ces bonnes performances
- On trouve des paramètres de pénalisation L^1 et L^2, qui n'existe pas dans la version originale
- D'autres methodes que les arbres CART peuvent être utilisés: des régréssions linéaire (Linear Boosting), ou des arbres DART (Dropouts Additive Regression Trees, http://proceedings.mlr.press/v38/korlakaivinayak15.pdf (http://proceedings.mlr.press/v38/korlakaivinayak15.pd
- On peut résoudre la grande majorité des problématiques industrielles avec cette implémentation, de la régréssion au ranking, en passant par de la classification multi-classes.
- Très customisable

On retrouve les même paramètres que pour l'implémentation du GBT dans sklearn. Attention, le nom n'est pas forcément le même, je vous invite à lire la documentation. Cependant, il en existe quelques autres qui sont importants:

- **booster** : type de méthode à booster
- **objective** : l'objectif du modèle (régéssion, etc.)
- **tree_method** : méthodes de construction des arbres (exact, approx, histogramme)
- **eval_metric** : choix de la métrique d'évaluation

Pour utiliser la version sklearn : "from xgboost import XGBRegressor" (ou XGBClassifier).

/!\ Tous les paramètres ne sont pas disponibles dans le wrapper sklearn.

In [1]:
```
#Sans wrapper cela ressemble à cela.
import xgboost as xgb
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

train = load_boston()['data']
target = load_boston()['target']
x_train,x_test,y_train,y_test = train_test_split(trai
n,target)

#Definition des objets d'apprentissage et test
dtrain = xgb.DMatrix(x_train,y_train)
dtest = xgb.DMatrix(x_test,y_test)

#Apprentissage
param = {'boost':'linear',
         'learnin_rate':0.1,
         'max_depth': 5,
         'objective': 'reg:linear',
          'eval_metric':'rmse'}
num_round = 100
bst = xgb.train(param, dtrain, num_round)


#Prediction
preds = bst.predict(dtest)
print 'Xgboost scoring : {}'.format(mean_squared_erro
r(y_test,preds))
```

```
Xgboost scoring : 10.9779122244
```

1.2.2 Lightgbm

Original paper (Microsoft Research, 2017) : https://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree (https://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree)

Introduit par Microsoft Research cet algorithme est très performant, mais difficile à calibrer. Des différences majeures avec les anciennes implémentations sont également présente:

- Construction des arbres verticale et non horizontale, i.e, l'algorithme choisi la feuille avec la meilleure loss pour grandir.
- Très efficace et rapide sur les données sparse et, les gros volumes de données.
- Comme pour xgboost, d'autres booster sont disponible comme les random forest.
- Il consomme très peu de ressources mémoire.
- Résolution de n'importe quel type de problématique.
- Très customisable

On retrouve les même paramètres que pour l'implémentation du GBT dans sklearn. Attention, le nom n'est pas forcément le même, je vous invite à lire la doc. Cependant, il en existe quelques autres qui sont importants - les noms varient par rapport à xgb:

- **boosting_type** : type de méthode à booster
- **task** : l'objectif du modèle (régéssion, etc.)
- **num_leaves** : contrôle la complexité du modèle, en relation avec le GBT num_leaves = $2^{\text{max\_depth}}$
- **device** : CPU/GPU
- **metric** : choix de la métrique d'évaluation

Pour utiliser la version sklearn : "from lightgbm import LGBMRegressor" (ou LGBMClassifier).

/!\ Tous les paramètres ne sont pas disponibles dans le wrapper sklearn.

In [2]:
```
import lightgbm as lgb

lgb_train = lgb.Dataset(x_train, y_train)
lgb_eval = lgb.Dataset(x_test, y_test, reference=lgb_
train)

params = {

    'boosting_type': 'rf',
    'objective': 'regression',
    'metric': {'l2', 'rmse'},
    'num_leaves': 31,
    'learning_rate': 0.05,
    'feature_fraction': 0.9,
```

```
    'bagging_fraction': 0.8,
    'bagging_freq': 5,
    'verbose_eval':5
}

print('Start training...')
gbm = lgb.train(params,
                lgb_train,
                num_boost_round=50,
                valid_sets=lgb_eval,
                early_stopping_rounds=5) # Arret si 5
iterations sans gain de performance


print('Start predicting...')
preds = gbm.predict(x_test, num_iteration=gbm.best_it
eration)
print('LGBM scoring :', mean_squared_error(y_test, pr
eds))
```

```
Start training...
[1]     valid_0's rmse: 6.4842  valid_0's l2: 42.
0449
Training until validation scores don't improve fo
r 5 rounds.
[2]     valid_0's rmse: 6.4842  valid_0's l2: 42.
0449
[3]     valid_0's rmse: 6.45817 valid_0's l2: 41.
7079
[4]     valid_0's rmse: 6.46152 valid_0's l2: 41.
7512
[5]     valid_0's rmse: 6.54774 valid_0's l2: 42.
8729
[6]     valid_0's rmse: 6.516   valid_0's l2: 42.
4583
[7]     valid_0's rmse: 6.49382 valid_0's l2: 42.
```

```
1697
[8]     valid_0's rmse: 6.24507 valid_0's l2: 39.
001
[9]     valid_0's rmse: 6.24156 valid_0's l2: 38.
9571
[10]    valid_0's rmse: 6.25525 valid_0's l2: 39.
1282
[11]    valid_0's rmse: 6.2966  valid_0's l2: 39.
6472
[12]    valid_0's rmse: 6.33072 valid_0's l2: 40.
078
[13]    valid_0's rmse: 6.22891 valid_0's l2: 38.
7993
[14]    valid_0's rmse: 6.15115 valid_0's l2: 37.
8366
[15]    valid_0's rmse: 6.09114 valid_0's l2: 37.
102
[16]    valid_0's rmse: 6.17311 valid_0's l2: 38.
1073
[17]    valid_0's rmse: 6.10094 valid_0's l2: 37.
2215
[18]    valid_0's rmse: 6.10263 valid_0's l2: 37.
2421
[19]    valid_0's rmse: 6.10793 valid_0's l2: 37.
3068
[20]    valid_0's rmse: 6.03273 valid_0's l2: 36.
3939
[21]    valid_0's rmse: 5.98    valid_0's l2: 35.
7604
[22]    valid_0's rmse: 5.93069 valid_0's l2: 35.
1731
[23]    valid_0's rmse: 5.9716  valid_0's l2: 35.
6601
[24]    valid_0's rmse: 5.93222 valid_0's l2: 35.
1912
[25]    valid_0's rmse: 5.89791 valid_0's l2: 34.
```

```
7853
[26]    valid_0's rmse: 5.86187 valid_0's l2: 34.
3615
[27]    valid_0's rmse: 5.84091 valid_0's l2: 34.
1162
[28]    valid_0's rmse: 5.82116 valid_0's l2: 33.
8858
[29]    valid_0's rmse: 5.80514 valid_0's l2: 33.
6996
[30]    valid_0's rmse: 5.78587 valid_0's l2: 33.
4762
[31]    valid_0's rmse: 5.77161 valid_0's l2: 33.
3114
[32]    valid_0's rmse: 5.75622 valid_0's l2: 33.
1341
[33]    valid_0's rmse: 5.74516 valid_0's l2: 33.
0069
[34]    valid_0's rmse: 5.73333 valid_0's l2: 32.
8711
[35]    valid_0's rmse: 5.72642 valid_0's l2: 32.
7918
[36]    valid_0's rmse: 5.71813 valid_0's l2: 32.
697
[37]    valid_0's rmse: 5.70963 valid_0's l2: 32.
5999
[38]    valid_0's rmse: 5.73011 valid_0's l2: 32.
8341
[39]    valid_0's rmse: 5.72223 valid_0's l2: 32.
7439
[40]    valid_0's rmse: 5.74146 valid_0's l2: 32.
9644
[41]    valid_0's rmse: 5.71764 valid_0's l2: 32.
6914
[42]    valid_0's rmse: 5.69546 valid_0's l2: 32.
4383
[43]    valid_0's rmse: 5.67574 valid_0's l2: 32.
```

```
214
[44]    valid_0's rmse: 5.66334 valid_0's l2: 32.
0734
[45]    valid_0's rmse: 5.6452  valid_0's l2: 31.
8683
[46]    valid_0's rmse: 5.64804 valid_0's l2: 31.
9003
[47]    valid_0's rmse: 5.65154 valid_0's l2: 31.
9399
[48]    valid_0's rmse: 5.66964 valid_0's l2: 32.
1448
[49]    valid_0's rmse: 5.65941 valid_0's l2: 32.
0289
[50]    valid_0's rmse: 5.66476 valid_0's l2: 32.
0896
Early stopping, best iteration is:
[45]    valid_0's rmse: 5.6452  valid_0's l2: 31.
8683
Start predicting...
('LGBM scoring :', 31.868316460801914)
```

1.2.3 CatBoost

Original paper (Yandex, 2017) : https://arxiv.org/abs/1706.09516 (https://arxiv.org/abs/1706.09516)

Dernier né de la firme russe Yandex, CatBoost est une implémentation similaire à celle de xgboost et de lightgbm mais qui a la particularité de tenir compte des variables catégorielles pour l'apprentissage. L'algorithme va construire à partir des variables catégorielles diverses statistiques* et, tenir compte de celles-ci pour l'apprentissage. On trouve quelque différences notables avec les deux autres algorithmes présentés ci-dessous:

- Très performant mais très lent.
- Pas d'autre booster que les arbres de disponibles
- Deux taches : régréssion ou classification
- Plusieurs paramètres pour l'apprentissage des variables catégorielles.
- Paramètres de détéction de l'overfitting
- Il y a une interface graphique super sexy...

*Voir https://tech.yandex.com/catboost/doc/dg/concepts/algorit
main-stages_cat-to-numberic-docpage/ (https://tech.yandex.com/catboost/doc/dg/concepts/algorit
main-stages_cat-to-numberic-docpage/)

Quelques paramètres importants :

- **iterations** : nombre d'arbres
- **eval_metric** : la métrique d'évaluation
- **ctr** : paramètre de transformation des variables catégorielles
- **fold_permutation_block_size** : nombre de permutations des catégorielles

Pour utiliser la version sklearn : "from catboost import CatBoostRegressor" (ou CatBoostClassifier). Et pour utiliser l'apprentissage sur les variables catégorielles, ajouter dans *fit* cat_features = [index des features catégorielles].

/!\ Tous les paramètres ne sont pas disponibles dans le wrapper sklearn.

In [3]:
```
import catboost as cat
train_pool = cat.Pool(x_train,y_train,cat_features = [3,8])
test_pool = cat.Pool(x_test,cat_features = [3,8])

param = {'logging_level':'Silent'}
model = cat.CatBoost(param)
model.fit(train_pool)

preds = model.predict(test_pool)
print('Cat scoring :', mean_squared_error(y_test, preds))
```

```
('Cat scoring :', 12.603604896736309)
```

In [4]:
```
w = cat.CatboostIpythonWidget('')
w.update_widget()
```

```
Widget Javascript not detected.  It may not be in
stalled or enabled properly.
```

Notez que les trois implémentations ci-dessus sont également portable sur GPU.

Chaque algorithme aura des performances différentes selon le jeu de données et le problème à traiter, à vous de choisir (et de tester) le meilleur modèle.

1.3 - Regularized Greedy Forest

Original paper (R. Johnson et al., 2014) : https://arxiv.org/pdf/1109.0887.pdf (https://arxiv.org/pdf/1109.0887.pdf)

Popularisé il y a peu, les RGF sont une sorte de mélange entre des forêts aléatoire et du boosting. Une RGF est un boosting de forêts d'arbres décisionnels construit de manière détérministe, et non aléatoirement.

C'est un algorithme puissant mais complexe, je vous invite si cela vous intéresse à lire le papier original.

La liste des paramètres est disponible ici : https://github.com/fukatani/rgf_python (https://github.com/fukatani/rgf_python)

Deux modèles sont implémentés : RGFRegressor et RGFClassifier du package rgf_python.

1.4 Autres algorithmes plus ou moins célèbres

Vous pouvez également vous penchez sur les algorithmes suivant :

- Rotation Forest : http://ieeexplore.ieee.org/document/1677518/ (http://ieeexplore.ieee.org/document/1677518/) PCA sur des K-Folds pour chaque arbre de la forêt
- Adaptative Hybrid Extreme Rotation Forest (AdaHERF) : https://www.ncbi.nlm.nih.gov/pubmed/24480062 (https://www.ncbi.nlm.nih.gov/pubmed/24480062) Rotation Forest + Extreme Learning
- NeuralNet et dérivés ...

Et le non supervisé ?

Il existe également d'autres algorithmes avancés pour la réduction de dimension et le clustering :

- T-SNE & variantes (Réduction de dimensions, https://lvdmaaten.github.io/tsne/ (https://lvdmaaten.github.io/tsne/)) : Minimiser la divergence de KL
- TruncatedSVD (Réduction de dimensions, voir cours de C.Boyer)
- SparsePCA (Réduction de dimensions, https://arxiv.org/abs/1211.1309 (https://arxiv.org/abs/1211.1309)) : PCA sur des données sparses
- HDDBScan (Clustering, https://link.springer.com/chapter/10.1007%2F978-3-642-37456-2_14 (https://link.springer.com/chapter/10.1007%2F978-3-642-37456-2_14)) : Clustering par densité

Pour découvrir de nouveaux algorithmes je vous invite à consulter les divers forums de machine learning indiqués au premier cours, ainsi que les papiers des conférences NIPS, ICML & MLConf...

2 - Feature importance & Feature Selection

2.1 - Feature importance

On parle de feature importance presque uniquement dans le cas des algorithmes de machine learning basés sur les arbres CART. L'importance d'une variable, n'est autre que le poids - selon une mesure donnée - qu'apporte cette variable dans l'apprentissage. Pour les arbres CART on retrouve la définition suivante :

Les arbres aléatoires sont une règle de décision interprétable pour la prédiction des classes.En effet, les variables de découpe sont choisies selon leur capacité à classifier les données. De ce fait, une variable permettant de découper les premiers nœuds a un pouvoir discriminant plus important qu'une variable apparaissant dans les dernières découpes (ou n'apparaissant pas dans l'arbre). B.Gregorutti- https://tel.archives-ouvertes.fr/tel-01146830/document (https://tel.archives-ouvertes.fr/tel-01146830/document)

Dans les méthodes d'ensemble, l'importance des features est calculée généralement de la manière suivante: On compte le nombre de fois où la variable est séléctionnée pour séparer un noeud, pondérée par l'amélioration du modèle à chaque séparation. Puis on moyenne les résultats.

Pour résumer : variable très utilisée qui sépare bien les données = variable importante.

2.2 - Feature Selection

Il est parfois nécéssaire d'effectuer une séléction des variables utilisées dans le modèle pour plusieurs raisons:

- Trop de corrélations entre certaines variables
- Trop de variables ce qui entraine un temps de calcul excessif
- Expertise métier
- Problème de dimensions
- Variable à la distribution étrange
- Et enfin, importance faible

Il existe alors plusieurs technique pour séléctionner les variables, autre que les critères d'Akkaike etc.

Voici la liste de ceux que j'utilise en pratique:

- VarianceThreshold : On supprime les features avec une variance dépassant un certain seuil.
- SelectKBest : On ne sélectionne que les k features les plus important
- RFECV : On supprime récursivement des features et on regarde les performance du modèle

Tous ces modèles de séléction sont disponible dans sklearn.

TP de mise en pratique

Les données proviennent du Kaggle suivant : https://www.kaggle.com/c/house-prices-advanced-regression-techniques (https://www.kaggle.com/c/house-prices-advanced-regression-techniques)

La variable à prédire est la "SalePrice".

In [5]:
```
import pandas as pd, numpy as np, matplotlib.pyplot a
s plt, seaborn as sns,time

from sklearn.ensemble import ExtraTreesRegressor
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import LabelEncoder

from xgboost import XGBRegressor
from lightgbm import LGBMRegressor
from catboost import CatBoostRegressor, Pool
from rgf.sklearn import *
```

1 Importer les données.

In [6]:
```
train = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')
train.head() #Tout est bon
```

Out[6]:

	Id	MSSubClass	MSZoning	LotFrontage	LotAr
0	831	20	RL	80.0	11900
1	1012	90	RL	75.0	9825
2	1165	80	RL	NaN	16157
3	1247	60	FV	65.0	8125
4	487	20	RL	79.0	10289

5 rows × 81 columns

2 Faites une rapide étude statistique et quelques graphiques sur l'importance des features par rapport à la variable cible. Les valeurs manquantes ?

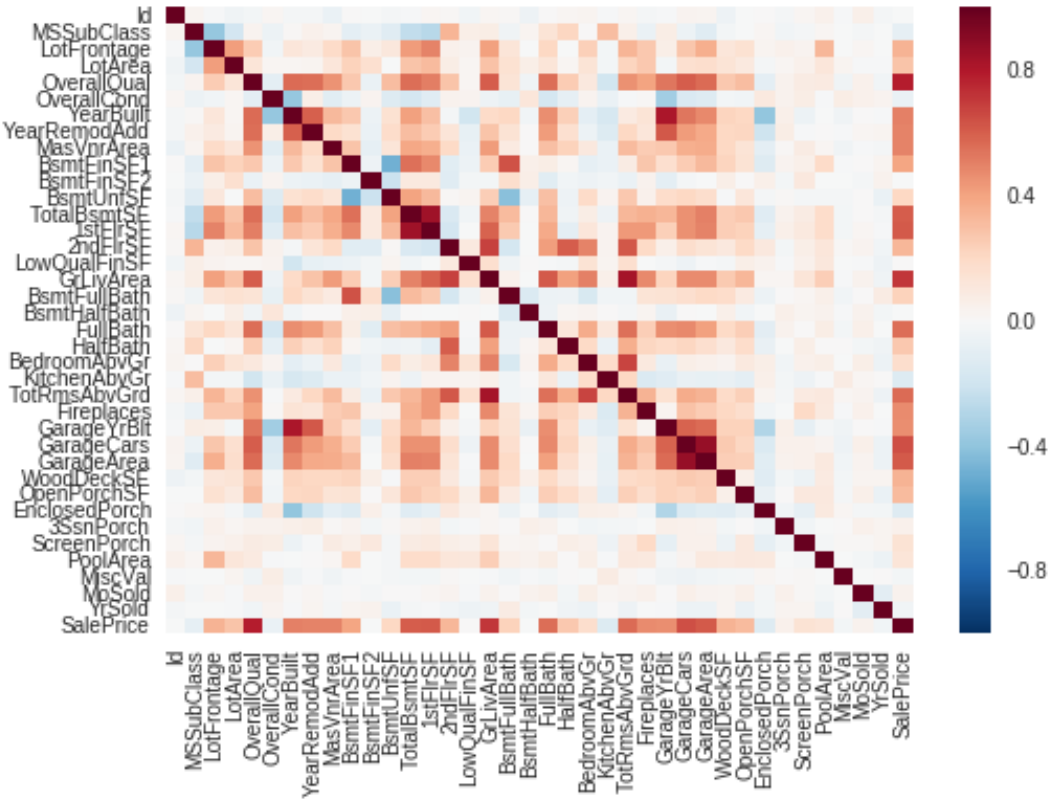
In [7]:
```
train.describe()
```

Out[7]:

	Id	MSSubClass	LotFrontage	
count	1022.000000	1022.000000	837.000000	1022
mean	730.187867	56.409002	70.425329	1056
std	418.632789	41.195064	25.416685	9805
min	1.000000	20.000000	21.000000	1300
25%	368.250000	20.000000	60.000000	7633
50%	734.500000	50.000000	70.000000	9580
75%	1092.500000	70.000000	80.000000	1170
max	1459.000000	190.000000	313.000000	2152

8 rows × 38 columns

In [8]:
```
sns.heatmap(train.corr())
plt.show()
```



In [9]:
```
print train.isnull().sum()[train.isnull().sum()>0], "
\n"
print test.isnull().sum()[test.isnull().sum()>0]
```

```
LotFrontage     185
Alley           963
MasVnrType        5
MasVnrArea        5
BsmtQual         23
BsmtCond         23
BsmtExposure     24
BsmtFinType1     23
BsmtFinType2     23
Electrical        1
FireplaceQu     483
GarageType       55
```

```
GarageYrBlt       55
GarageFinish      55
GarageQual        55
GarageCond        55
PoolQC          1018
Fence            830
MiscFeature      982
dtype: int64

LotFrontage       74
Alley            406
MasVnrType         3
MasVnrArea         3
BsmtQual          14
BsmtCond          14
BsmtExposure      14
BsmtFinType1      14
BsmtFinType2      15
FireplaceQu      207
GarageType        26
GarageYrBlt       26
GarageFinish      26
GarageQual        26
GarageCond        26
PoolQC           435
Fence            349
MiscFeature      424
dtype: int64
```

3 Effectuez un peu de feature engineering, pour rendre les jeux de données utilisables. Commencez par supprimer les id et récupérer les targets.

Effectuer trois modifications des dataframes (faites des copies !):

- Traitement des NA et des catégorielles
- Uniquement traitement des catégorielles
- Uniquement traitement des valeurs manquantes

In [10]:
```
y_train = train['SalePrice']
y_test = test['SalePrice']

train = train.drop(['Id','SalePrice'], axis = 1)
test = test.drop(['Id','SalePrice'], axis = 1)
```

In [11]:
```
#Copies des dataframes
train_full = train.copy()
test_full = test.copy()

train_cat = train.copy()
test_cat = test.copy()

train_na = train.copy()
test_na = test.copy()
```

In [12]:
```
#Full engineering

# Commençons par les variables catégorielles.

for c in train_full.columns:
    if train_full[c].dtype == object:
        encoder = LabelEncoder()
        encoder.fit(list(train_full[c])+list(test_ful
l[c]))
        train_full[c] = encoder.transform(list(train_
full[c]))
        test_full[c] = encoder.transform(list(test_fu
ll[c]))

#Dummy filling
train_full.fillna(0,inplace = True)
test_full.fillna(0,inplace = True)
```

In [13]:
```
#Only cat
for c in train_cat.columns:
    if train_cat[c].dtype == object:
        encoder = LabelEncoder()
        encoder.fit(list(train_cat[c])+list(test_cat[
c]))
        train_cat[c] = encoder.transform(list(train_c
at[c]))
        test_cat[c] = encoder.transform(list(test_cat
[c]))
```

In [14]:
```
#Only NA
train_na = train_na.fillna(0)
test_na = test_na.fillna(0)
```

4 Quelle mesure de performance devrions nous utiliser ? Implémentez la.

La rmse semble pertinente, implémentons-la !

In [15]:
```
def rmse(preds, targets):
    return np.sqrt(((preds - targets) ** 2).mean())
```

5 Comparez xgboost, lightgbm, catboost, extratrees, Regularized Greedy Forest et une régression au choix sur le jeu de données adéquat à chaque algorithme, sans chercher à optimiser les paramètres des modèles. Donnez également le temps de calcul nécessaire à l'apprentissage. Une fois cela effectué, recommencer sur uniquement le jeu "full".

In [16]:
```
regs = {}
regs['xgb'] = {'reg':XGBRegressor(),'name':'XGBRegressor'}
regs['lgb'] = {'reg':LGBMRegressor(),'name':'LGBMRegressor'}
regs['cat'] = {'reg':CatBoostRegressor(logging_level='Silent'),'name':'CatBoostRegressor'}
regs['ext'] = {'reg':ExtraTreesRegressor(),'name':'ExtraTreesRegressor'}
regs['lr'] = {'reg':LinearRegression(),'name':'LinearRegression'}
regs['rgf'] = {'reg':RGFRegressor(),'name':'RGFRegressor'}

def training(regs,key,train,y_train,test,y_test):
    if key != 'cat':
        t0 = time.time()
        regs[key]['reg'].fit(train,y_train)
        t1 = time.time()
    else:
        cat_features_idx = [c for c,_ in enumerate(train.columns) if train[_].dtype == object]
        t0 = time.time()
        regs[key]['reg'].fit(train,y_train,cat_features = cat_features_idx)
        t1 = time.time()
    print '{} apprend en {}s et a une performance de {}'.format(regs[key]['name'],t1-t0,rmse(y_test,regs[key]['reg'].predict(test)))
```

In [17]:
```
training(regs,'lr',train_full,y_train,test_full,y_tes
t)
training(regs,'lgb',train_full,y_train,test_full,y_te
st)
training(regs,'ext',train_full,y_train,test_full,y_te
st)
training(regs,'xgb',train_cat,y_train,test_cat,y_test
)
training(regs,'cat',train_na,y_train,test_full,y_test
)
training(regs,'rgf',train_full,y_train,test_full,y_te
st)
```

```
LinearRegression apprend en 0.00884795188904s et
a une performance de 28252.6046359
LGBMRegressor apprend en 0.274455070496s et a une
performance de 65675.8824534
ExtraTreesRegressor apprend en 0.14523601532s et
a une performance de 26652.4283558
XGBRegressor apprend en 0.263610124588s et a une
performance de 22495.4974998
CatBoostRegressor apprend en 45.4231359959s et a
une performance de 56658.9931549
RGFRegressor apprend en 0.528362989426s et a une
performance de 22885.8662082
```

In [18]:
```
#Que constate-t-on?
for reg in regs:
    training(regs,reg,train_full,y_train,test_full,y_test)
```

```
LGBMRegressor apprend en 0.259593963623s et a une performance de 65675.8824534
XGBRegressor apprend en 0.209308862686s et a une performance de 22629.7132304
CatBoostRegressor apprend en 11.1938290596s et a une performance de 22273.9658911
ExtraTreesRegressor apprend en 0.139853000641s et a une performance de 25738.5544057
LinearRegression apprend en 0.00773882865906s et a une performance de 28252.6046359
RGFRegressor apprend en 0.525736093521s et a une performance de 22885.8662082
```

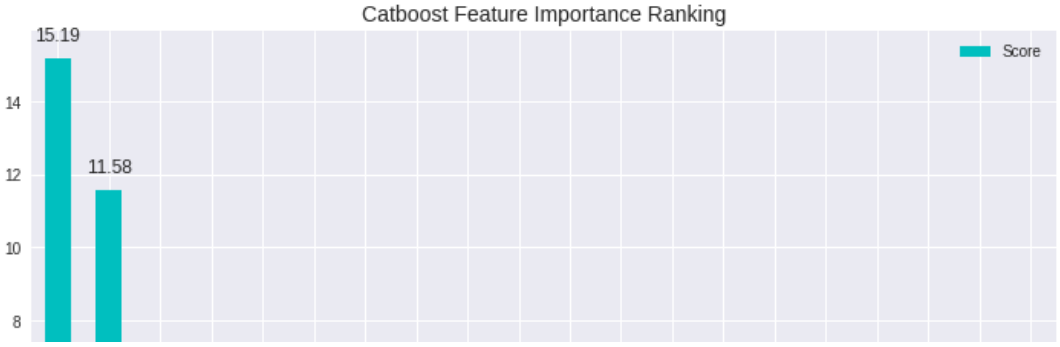
(bonus) Fittez une dernière fois catboost sans tenir compte des variables catégorielles, que se passe-t-il?

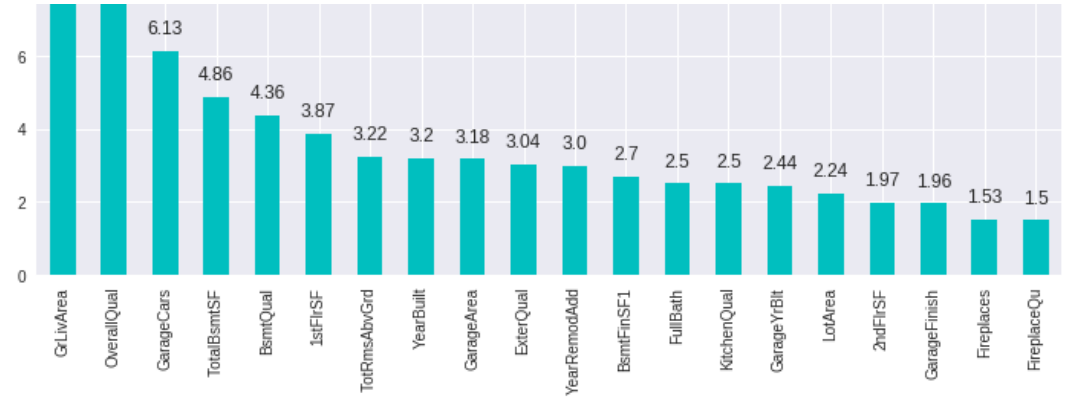
In [19]:
```
regs['cat'] = {'reg':CatBoostRegressor(logging_level='Silent'),'name':'CatBoostRegressor'}
key = 'cat'
t0 = time.time()
regs[key]['reg'].fit(train_full,y_train)
t1 = time.time()
print '{} apprend en {}s et a une performance de {}'.format(regs[key]['name'],t1-t0,rmse(y_test,regs[key]['reg'].predict(test_full)))
```

```
CatBoostRegressor apprend en 11.1554839611s et a une performance de 20963.5562934
```

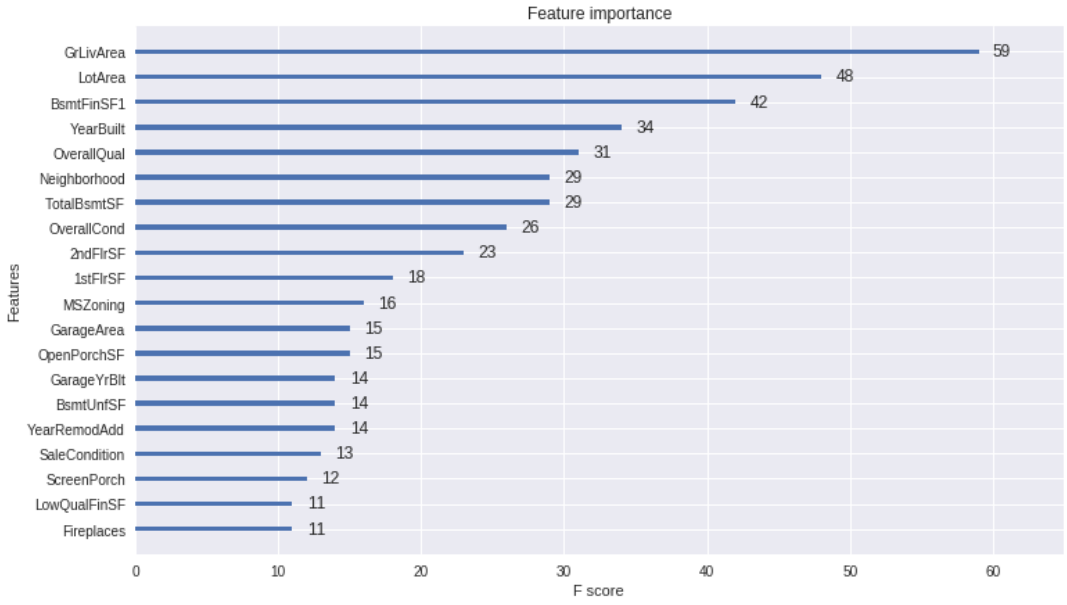
6 Affichez l'importance des 20 features les plus importants pour vos deux modèles les plus performants.

In [20]:
```
# Pas si simple pour catboos
feature_score = pd.DataFrame(list(zip(train_full.dtyp
es.index, regs['cat']['reg'].get_feature_importance(P
ool(train_full,label=y_train) ))),columns=['Feature',
'Score'])
feature_score = feature_score.sort_values(by='Score',
ascending=False, inplace=False, kind='quicksort', na_
position='last')
feature_score = feature_score[:20]
plt.rcParams["figure.figsize"] = (12,7)
ax = feature_score.plot('Feature', 'Score', kind='bar
', color='c')
ax.set_title("Catboost Feature Importance Ranking", f
ontsize = 14)
ax.set_xlabel('')

rects = ax.patches

labels = feature_score['Score'].round(2)
for rect, label in zip(rects, labels):
    height = rect.get_height()
    ax.text(rect.get_x() + rect.get_width()/2, height
+ 0.35, label, ha='center', va='bottom')

plt.show()
```





In [21]:
```
from xgboost import plot_importance
plot_importance(regs['xgb']['reg'],max_num_features=20)
plt.show()
```



7 Entrainez à nouveau les modèles mais seulement sur les 20 features les plus importants d'un des deux modèles. Donnez également le temps de calcul nécessaire à l'apprentissage et les performances sur le jeu de test. Qu'observez-vous ?

In [22]:
```
#Recuper les 20 features les plus important de catboo
st
best_features = feature_score['Feature'].values
train_importance = train_full[best_features]
test_importance = test_full[best_features]
```

In [23]:
```
training(regs,'lr',train_importance,y_train,test_impo
rtance,y_test)
training(regs,'lgb',train_importance,y_train,test_imp
ortance,y_test)
training(regs,'ext',train_importance,y_train,test_imp
ortance,y_test)
training(regs,'xgb',train_importance,y_train,test_imp
ortance,y_test)
training(regs,'cat',train_importance,y_train,test_imp
ortance,y_test)
training(regs,'rgf',train_importance,y_train,test_imp
ortance,y_test)
```

```
LinearRegression apprend en 0.0020010471344s et a
une performance de 28526.3432143
LGBMRegressor apprend en 0.139203071594s et a une
performance de 65637.4741292
ExtraTreesRegressor apprend en 0.0636188983917s e
t a une performance de 27455.2991159
XGBRegressor apprend en 0.0840659141541s et a une
performance de 25028.6800559
CatBoostRegressor apprend en 6.59738016129s et a
une performance de 23176.0795378
RGFRegressor apprend en 0.235453128815s et a une
performance de 26592.6444897
```

Ne prendre qu'un certain nombre des features importante dégrade les performances de nos modèles, mais cependant le temps de calcul est grandement amélioré. Il faut donc trouver un compromis entre performance et temps de calcul, selon le problème donné.

8 Jouez avec les paramètres des modèles, et déterminer le plus performant sur le jeu de test.

In [24]:
```
regs = {}
regs['xgb'] = {'reg':XGBRegressor(n_estimators=400, m
ax_depth=4),'name':'XGBRegressor'}
regs['lgb'] = {'reg':LGBMRegressor(n_estimators=2200,
max_bin=40,num_leaves=20,learning_rate=0.2),'name':'L
GBMRegressor'}
regs['cat'] = {'reg':CatBoostRegressor(iterations=80,
learning_rate=0.1,logging_level='Silent'),'name':'Cat
BoostRegressor'}
regs['ext'] = {'reg':ExtraTreesRegressor(n_estimators
=500, max_depth=10),'name':'ExtraTreesRegressor'}
regs['lr'] = {'reg':LinearRegression(),'name':'Linear
Regression'}
regs['rgf'] = {'reg':RGFRegressor(n_iter=80),'name':'
RGFRegressor'}

training(regs,'lr',train_full,y_train,test_full,y_tes
t)
training(regs,'lgb',train_full,y_train,test_full,y_te
st)
training(regs,'ext',train_full,y_train,test_full,y_te
st)
training(regs,'xgb',train_full,y_train,test_full,y_te
st)
```

```
training(regs,'cat',train_full,y_train,test_full,y_te
st)
training(regs,'rgf',train_full,y_train,test_full,y_te
st)
```

```
LinearRegression apprend en 0.0151281356812s et a
une performance de 28252.6046359
LGBMRegressor apprend en 2.68547320366s et a une
performance de 24642.4156611
ExtraTreesRegressor apprend en 4.72896003723s et
a une performance de 25095.2156858
XGBRegressor apprend en 1.0796251297s et a une pe
rformance de 22366.4664178
CatBoostRegressor apprend en 1.1181871891s et a u
ne performance de 24130.6660472
RGFRegressor apprend en 0.651011943817s et a une
performance de 22660.5929693
```

9 Utilisez un ensemble de validation pour connaitre le nombre d'itérations optimal de boosting avec Xgboost avec ou sans wrapper sklearn.

In [25]:
```
from sklearn.model_selection import train_test_split
X_tr,X_va,Y_tr,Y_va = train_test_split(train_full,y_train)

model = XGBRegressor(n_estimators=1000, max_depth=3, learning_rate=0.1)
model.fit(X_tr,Y_tr,eval_set=[(X_va,Y_va)],early_stopping_rounds=100,verbose=50)
print rmse(y_test,model.predict(test_full))
```

```
[0]     validation_0-rmse:188852
Will train until validation_0-rmse hasn't improved in 100 rounds.
[50]    validation_0-rmse:30167.3
[100]   validation_0-rmse:29077.2
[150]   validation_0-rmse:28733.2
[200]   validation_0-rmse:28499
[250]   validation_0-rmse:28358.3
[300]   validation_0-rmse:28470.3
[350]   validation_0-rmse:28432.1
Stopping. Best iteration:
[254]   validation_0-rmse:28349.2

23979.912141299963
```

10 Comment améliorer les résultats ?

- Utiliser des statistiques autour des prix pour construire de nouvelles variables
- Remplir les NA de manière plus intelligente
- Importer des données externes, comme des positions géographiques
- Utiliser des méthodes d'agrégation
- Stacker plusieurs modèles