1 Objectif

Etude du package H2O sous Python. Efficacité des calculs sur une machine avec un processeur multicœur. Etude des algorithmes d'analyse prédictive.

« H2O » est une plate-forme JAVA de machine learning. Elle propose des outils pour la manipulation et la préparation de données, des algorithmes de modélisation, supervisées, non-supervisées ou de réduction de dimensionnalité. Nous pouvons accéder à ses fonctionnalités en mode client-serveur via différents langages de programmation avec le mécanisme des API (application programming interface). Nous nous appuierons sur Python dans ce tutoriel, mais nous aurions pu réaliser entièrement la même trame sous R.

« H2O » m'a intrigué lors de l'étude des librairies R et Python pour le deep learning. Elle semble savoir tirer parti des capacités multicœurs des processeurs des machines modernes. J'avais exploré ce sujet par le passé, soit en étudiant les possibilités de parallélisation des calculs sous R, soit en programmant moi-même nativement des variantes multithreads des arbres de décision ou de l'analyse discriminante. Je sais que la chose n'est pas facile. Le fait que H2O propose des solutions efficaces est un véritable atout parce que, même aujourd'hui (janvier 2019), elles ne sont pas très présentes encore dans les outils de data science.

Ce tutoriel comporte trois grandes parties : (section 2) nous évaluerons son aptitude à paralleliser ses algorithmes d'analyse prédictive ; (section 3) nous étudierons ensuite dans le détail ces approches supervisées, en regardant de près les (une partie des) paramètres et les sorties ; (section 4) enfin, nous jetterons un œil sur quelques outils additionnels de H2O, toujours pour le supervisé.

Bien sûr, nous ne couvrons pas tout dans ce document. D'autres thèmes tout aussi intéressants peuvent faire l'objet d'autres études. Je pense notamment à la capacité de H2O à traiter des données dans le cloud ou sur des systèmes de fichiers distribués tels que HDFS, sous-jacent à Hadoop.

2 Exploitation d'un processeur multicœur avec H2O

2.1 Données « waveform »

Dans cette section, nous utilisons une variante des données « waveform » (UCI, version 1) avec 100.000 observations et 22 variables, dont la cible « onde » à 3 modalités {A, B, C}. Après avoir affiché la version de Python, nous chargeons le fichier « wave100k.txt » et nous en listons les propriétés.

```
#version de Python
import sys
print(sys.version)
```

```
3.6.7 |Anaconda, Inc.| (default, Dec 10 2018, 20:35:02) [MSC v.1915 64 bit (AMD64)]
```

```
#changer le répertoire courant
import os
os.chdir("… votre dossier …")

#charger les données
import pandas
wave = pandas.read_table("wave100k.txt",sep="\t",decimal=".",header=0)

#vérification
print(wave.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 22 columns):
onde    100000 non-null object
v01     100000 non-null float64
v02     100000 non-null float64
v03     100000 non-null float64
v04     100000 non-null float64
v05     100000 non-null float64
v06     100000 non-null float64
v07     100000 non-null float64
v08     100000 non-null float64
v09     100000 non-null float64
v10     100000 non-null float64
v11     100000 non-null float64
v12     100000 non-null float64
v13     100000 non-null float64
v14     100000 non-null float64
v15     100000 non-null float64
v16     100000 non-null float64
v17     100000 non-null float64
v18     100000 non-null float64
v19     100000 non-null float64
v20     100000 non-null float64
v21     100000 non-null float64
dtypes: float64(21), object(1)
```

Nous traiterons cet ensemble de données d'un seul tenant dans cette partie dédiée à l'étude des capacités de parallélisation de H2O.

2.2 Procédure d'évaluation des algorithmes

Nous utilisons le module « timeit » pour mesure les temps d'exécution.

```
#outil pour mesurer le temps
import timeit
print(dir(timeit))
```

```
['Timer', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', '_globals', 'default_number', 'default_repeat',
'default_timer', 'dummy_src_name', 'gc', 'itertools', 'main', 'reindent', 'repeat', 'sys',
'template', 'time', 'timeit']
```

La fonction default_timer() permet de mesurer les écarts. Pour ne pas interférer avec les résultats, il faudra faire attention à ne pas solliciter la machine lorsque nous lançons les calculs.

Pour appréhender les différentes étapes dans un premier temps, nous réalisons un processus simple de modélisation à l'aide de la régression logistique multinomiale (voir Packages Python pour le Deep Learning, section 3.4). Tout d'abord, nous chargeons le package et nous vérifions la version utilisée.

```
#importation de la librairie H2O
import h2o

#vérification de version
print(h2o.__version__)
```

```
3.22.0.3
```

Ensuite, nous démarrons le serveur H2O en spécifiant le nombre de threads (nthreads = 1) c.-à-d. nous sollicitions un seul thread pour les calculs (concrètement 1 cœur pour un processeur multicœur).

```
#démarrage de la machine "h2o"
h2o.init(nthreads = 1)
```

```
Checking whether there is an H2O instance running at http://localhost:54321..... not found.
Attempting to start a local H2O server...
; OpenJDK 64-Bit Server VM (build 25.152-b12, mixed mode)56-b12)
  Starting server from D:\Logiciels\Anaconda3\lib\site-packages\h2o\backend\bin\h2o.jar
  Ice root: C:\Users\Zatovo\AppData\Local\Temp\tmpgg_fd2ra
  JVM stdout: C:\Users\Zatovo\AppData\Local\Temp\tmpgg_fd2ra\h2o_Zatovo_started_from_python.out
  JVM stderr: C:\Users\Zatovo\AppData\Local\Temp\tmpgg_fd2ra\h2o_Zatovo_started_from_python.err
  Server is running at http://127.0.0.1:54321

Connecting to H2O server at http://127.0.0.1:54321... successful.
--------------------------  --------------------------------
H2O cluster uptime:         03 secs
H2O cluster timezone:       Europe/Paris
H2O data parsing timezone:  UTC
H2O cluster version:        3.22.0.3
H2O cluster version age:    6 days
H2O cluster name:           H2O_from_python_Zatovo_6nuryz
H2O cluster total nodes:    1
H2O cluster free memory:    1.759 Gb
H2O cluster total cores:    8
H2O cluster allowed cores:  1
H2O cluster status:         accepting new members, healthy
H2O connection url:         http://127.0.0.1:54321
H2O connection proxy:
H2O internal security:      False
H2O API Extensions:         Algos, AutoML, Core V3, Core V4
Python version:             3.6.7 final
--------------------------  --------------------------------
```

Notre version de H2O est récente (6 jours). Nous sollicitions 1 cœur, parmi les 8 disponibles. En effet, ma machine est un quad-core (4 cœurs physiques). Avec l'hyperthreading, je dispose bien de 8 cœurs logiques.

Nous typons notre data frame dans un format spécifique à H2O.

```
#typage du data frame au format h2o
h2owave = h2o.H2OFrame(wave)
```

Puis nous importons la classe de modélisation H2OGeneralizedLinearEstimator que nous instancions.

```
#importation de la régression généralisée
from h2o.estimators import H2OGeneralizedLinearEstimator

#instanciation - régression logistique multinomiale
modelH2o = H2OGeneralizedLinearEstimator(family="multinomial")
```

Nous demandons une régression logistique multinomiale parce que la variable cible « onde » possède 3 modalités. « modelH2o » est l'identifiant de l'objet généré.

Il ne reste plus qu'à réaliser les calculs, en encadrant l'appel de la fonction train() par les instructions permettant de mesurer la durée d'exécution.

```
#apprentissage -- mesure du temps de calcul
depart = timeit.default_timer()
modelH2o.train(h2owave.columns[1:],"onde",training_frame=h2owave)
duree = timeit.default_timer()- depart
print(str(duree)+" sec.")
```

```
glm Model Build progress:
|████████████████████████████████████████████████████████████████| 100%
7.249490621273253 sec.
```

L'opération – l'estimation des paramètres du modèle à partir des données disponibles – a duré 7.24 secondes. Nous affichons la matrice de confusion en resubstitution (calculé sur les données ayant servi à la construction du modèle) pour nous assurer que le processus a été mené à bon port.

```
#affichage matrice de confusion
modelH2o.confusion_matrix(h2owave)
```

```
Confusion Matrix: Row labels: Actual class; Column labels: Predicted class

A      B      C      Error     Rate
-----  -----  -----  --------  -----------------
28078  2770   2737   0.163972  5 507 / 33 585
2295   29423  1628   0.117645  3 923 / 33 346
2226   1657   29186  0.117421  3 883 / 33 069
32599  33850  33551  0.13313   13 313 / 100 000
```

C'est le cas. L'apprentissage a fonctionné, nous classons mieux que le hasard.

Nous stoppons le serveur pour clôturer la session d'analyse.

```
#éteindre le moteur "h2o"
h2o.cluster().shutdown()
```

```
H2O session _sid_9614 closed.
```

2.3 Expérimentation et résultats

Tout est en place pour mener l'expérimentation. Nous croisons le nombre de threads sollicités (1, 4 et 8) avec 5 méthodes supervisées de machine learning (régression logistique multinomiale, random forest, gradient boosting, naive bayes et perceptron multicouche). Chaque combinaison a été réitérée 10 fois pour disposer d'une mesure stable du temps d'exécution. Enfin, autant que faire se peut, nous avons conservé les paramètres par défaut des algorithmes.

```
#démarrage du moteur h2o
#spécification du nombre de threads à utiliser
h2o.init(nthreads=1)
#h2o.init(nthreads=4)
#h2o.init(nthreads=8)

#typage du data frame au format h2o
h2owave = h2o.H2OFrame(wave)

#régression logistique
from h2o.estimators import H2OGeneralizedLinearEstimator

#random forest
#from h2o.estimators import H2ORandomForestEstimator

#gradient boosting
#from h2o.estimators import H2OGradientBoostingEstimator

#naive bayes
#from h2o.estimators import H2ONaiveBayesEstimator

#perceptron multicouche
#from h2o.estimators import H2ODeepLearningEstimator

#comptabilisation temps moyen
temps = 0.0

#répétition - 10 fois
for i in range(10):
    #instanciation
    modelH2o = H2OGeneralizedLinearEstimator(family="multinomial")
    #modelH2o = H2ORandomForestEstimator()
    #modelH2o = H2OGradientBoostingEstimator()
    #modelH2o = H2ONaiveBayesEstimator()
    #modelH2o = H2ODeepLearningEstimator(hidden=[2],activation="Tanh",distribution = "multinomial")
    #apprentissage
    depart = timeit.default_timer()
    modelH2o.train(h2owave.columns[1:],"onde",training_frame=h2owave)
```

```
    duree = timeit.default_timer()- depart
    print(str(duree)+" sec.")
    #addition du temps
    temps = temps + duree

#temps moyen
temps = temps / 10.0
print(temps)

#éteindre le moteur "h2o" (stopper le serveur)
h2o.cluster().shutdown()
```

Nous récapitulons les résultats dans le tableau suivant :

nthreads	1	4	8
H2OGeneralizedLinearEstimator	7.04	2.52	1.59
H2ORandomForestEstimator (50 trees, max depth = 20)	64.09	20.89	14.74
H2OGradientBoostingEstimator (50 trees, max depth = 5)	27.04	9.38	6.37
H2ONaiveBayesEstimator	0.84	0.46	0.47
H2ODeepLearningEstimator (1 hidden, 2 neurons, epochs = 10)	2.36	1.24	1.16

Figure 1 - Durée moyenne d'exécution sur le fichier "waveform" (100.000 obs.) en secondes

Plusieurs commentaires viennent à la lecture de ce récapitulatif :

- Dans la limite des cœurs physiques, H2O sait exploiter les capacités multicœurs du processeur (passage de 1 à 4).
- Mais le gain n'est pas proportionnel à l'augmentation du nombre de threads, parce des composantes des calculs restent séquentielles et nécessitent des synchronisations contraignantes.
- Pour certains algorithmes, le passage aux cœurs logiques avec l'hyperthreading introduit quand-même une amélioration (passage de 4 à 8 threads, lorsque nous sollicitons les cœurs logiques), mais dans une proportion moindre cependant.
- Globalement, le gain dépend de l'algorithme de machine learning.

Ces résultats sont remarquables. Pour certains algorithmes, la parallélisation semble évidente, le random forest par exemple (on peut construire les arbres en parallèle, indépendamment les uns des autres). Mais lorsque l'on constate que le gradient boosting, basé également sur les arbres de décision, mais qui est séquentiel par nature (l'arbre à l'étape t a besoin des résultats de celui à l'étape $t-1$), progresse également, on comprend que les améliorations apportées aux implémentations sont plus sophistiquées qu'il n'y paraît au premier abord.

3 Etude des méthodes supervisées de H2O

Dans cette section, nous examinons les propriétés des méthodes supervisées disponibles sous H2O. Nous essayons de mettre en évidence les fonctionnalités que l'on doit retrouver dans ce type de librairie, et celles qui seraient un peu plus particulières, moins présentes dans les autres outils.

3.1 Données « pima »

Nous traitons la base « PIMA » cette fois-ci (Pima Indians Diabete Database). On souhaite prédire l'occurrence du diabète (positive, negative) chez des personnes à partir de leurs caractéristiques (âge, indice de masse corporelle, etc.). Elle est de taille réduite (768 observations, 9 variables, variable cible binaire), plus facile à manipuler dans ce nouveau contexte. Elle est surtout très connue. Pour l'avoir triturée dans tous les sens, je sais à peu près à l'avance ce que l'on pourra en tirer. Par exemple, les variables PLASMA et BODYMASS sont les plus pertinentes en prédiction pour la majorité des méthodes supervisées, le taux d'erreur tourne souvent autour de 24%, le F1-Score autour de 0.64. Si nos résultats s'écartent sensiblement de ces références communément admises, il faudra essayer d'en comprendre les raisons.

Démarrage et configuration du serveur. Nous créons un nouveau projet Python, nous importons le package puis nous démarrons le serveur H2O en demandant toute la puissance disponible (nthreads = -1), soit 8 cœurs sur ma machine.

```
#importation du package
import h2o

#démarrage H2O
h2o.init(nthreads=-1)
```

```
Checking whether there is an H2O instance running at http://localhost:54321..... not found.
Attempting to start a local H2O server...
; OpenJDK 64-Bit Server VM (build 25.152-b12, mixed mode)56-b12)
  Starting server from D:\Logiciels\Anaconda3\lib\site-packages\h2o\backend\bin\h2o.jar
  Ice root: C:\Users\Zatovo\AppData\Local\Temp\tmpra5py4pr
  JVM stdout: C:\Users\Zatovo\AppData\Local\Temp\tmpra5py4pr\h2o_Zatovo_started_from_python.out
  JVM stderr: C:\Users\Zatovo\AppData\Local\Temp\tmpra5py4pr\h2o_Zatovo_started_from_python.err
  Server is running at http://127.0.0.1:54321

Connecting to H2O server at http://127.0.0.1:54321... successful.
--------------------------  ------------------------------
H2O cluster uptime:         01 secs
H2O cluster timezone:       Europe/Paris
H2O data parsing timezone:  UTC
H2O cluster version:        3.22.0.3
H2O cluster version age:    8 days
H2O cluster name:           H2O_from_python_Zatovo_vccjds
H2O cluster total nodes:    1
H2O cluster free memory:    1.759 Gb
H2O cluster total cores:    8
H2O cluster allowed cores:  8
H2O cluster status:         accepting new members, healthy
```

```
H2O connection url:         http://127.0.0.1:54321
H2O connection proxy:
H2O internal security:      False
H2O API Extensions:         Algos, AutoML, Core V3, Core V4
Python version:             3.6.7 final
--------------------------  ------------------------------
```

Importation des données. Nous importons le fichier de données au format CSV (séparateur '' ; '') et nous affichons les 10 premières lignes.

```
#changer le répertoire courant
import os
os.chdir(" ... votre dossier de travail ...")

#chargement des données
pima = h2o.import_file("pima.csv")

#affichage des premières valeurs
print(pima.head().as_data_frame())
```

```
   pregnant  diastolic  triceps    ...     plasma  serum   diabete
0         6         72       35    ...        148      0  positive
1         1         66       29    ...         85      0  negative
2         8         64        0    ...        183      0  positive
3         1         66       23    ...         89     94  negative
4         0         40       35    ...        137    168  positive
5         5         74        0    ...        116      0  negative
6         3         50       32    ...         78     88  positive
7        10          0        0    ...        115      0  negative
8         2         70       45    ...        197    543  positive
9         8         96        0    ...        125      0  positive
```

Précision importante, en utilisant la commande import_file() de H2O, notre ensemble de données « pima » est directement typé au format Data Frame spécifique à H2O (h2o.frame.H2OFrame). Il est par conséquent directement reconnu par les différents outils de la librairie.

```
#affichage du type
print(type(pima))
```

```
<class 'h2o.frame.H2OFrame'>
```

Nous disposons de 768 observations et 9 variables.

```
#dimension
print(pima.shape)
```

```
(768, 9)
```

Nous affichons la liste des variables.

```
#affichage de la liste des colonnes
print(pima.col_names)
```

```
['pregnant', 'diastolic', 'triceps', 'bodymass', 'pedigree', 'age', 'plasma', 'serum',
'diabete']
```

Nous nous assurons que la variable cible « diabète » est bien reconnue comme type factor, avec les modalités, dans l'ordre alphabétique, {negative, positive}.

```
#diabete est bien un type facteur
pima['diabete'].isfactor()  # [True]

#nombre de niveaux (modalités) de "diabete"
pima['diabete'].levels()
```

```
[['negative', 'positive']]
```

Subdivision en échantillons d'apprentissage et de test. L'étape suivante consiste à scinder les données en ensembles d'apprentissage et de test. La fonction split_frame() fait l'affaire. Nous indiquons la proportion des données (ratios) dévolue à l'apprentissage. L'option seed assure la reproductibilité à l'identique de l'opération. Nous vérifions ensuite les dimensions des sous-échantillons.

```
#subdivision
pimaTrain,pimaTest = pima.split_frame(ratios=[0.67],seed=1)

#vérification train
pimaTrain.shape    #(509, 9)
#vérification test
pimaTest.shape     #(259, 9)
```

3.2 Régression logistique binaire

Initialisation. Nous souhaitons réaliser une régression logistique. Après avoir importé la classe de calcul, nous l'instancions en précisant les paramètres de l'algorithme.

```
#régression généralisée
from h2o.estimators import H2OGeneralizedLinearEstimator

#régression logistique
reg = H2OGeneralizedLinearEstimator(seed=100,family="binomial",standardize=True,lambda_=0,compute_p_values=True)
```

H2OGeneralizedLinearEstimator implémente la régression généralisée, dans notre initialisation :

- family = ''binomial'' précise la nature binaire de la variable cible, nous réalisons bien une régression logistique « binaire » ;
- Les variables sont automatiquement centrées et réduites (standardize = True) ;
- Nous désactivons la régularisation des paramètres estimés en mettant à zéro le coefficient de pénalité (lambda_ = 0) ;
- Nous demandons le calcul des probabilités critiques (compute_p_values = True) du test de significativité individuel des coefficients. De fait, nous obligeons l'outil à utiliser un algorithme (IRLS) capable de produire les écarts-type estimés des coefficients estimés.

Modélisation. Nous lançons les calculs avec la méthode train() en précisant l'ensemble de données à traiter (training_frame), la liste des explicatives (x est un vecteur contenant le nom des variables explicatives), et la variable expliquée (y).

```
#modélisation
reg.train(x=pimaTrain.columns[:-1],y="diabete",training_frame=pimaTrain)
```

Affichage des résultats. Nous disposons de plusieurs affichages.

```
#affichage par défaut
reg.show()
```

show() correspond à l'affichage par défaut. Il fournit moult informations.

```
Model Details
=============
H2OGeneralizedLinearEstimator :  Generalized Linear Modeling
Model Key:  GLM_model_python_1546242989411_1


ModelMetricsBinomialGLM: glm
** Reported on train data. **

MSE: 0.1476906026290413
RMSE: 0.38430535076816363
LogLoss: 0.44947869914882604
Null degrees of freedom: 508
Residual degrees of freedom: 500
Null deviance: 652.4944896569244
Residual deviance: 457.56931573350494
AIC: 475.56931573350494
AUC: 0.8519044178364987
pr_auc: 0.7263703195250606
Gini: 0.7038088356729975
```

Ci-dessus, nous disposons des différents indicateurs que qualité de modélisation, calculés sur les données d'apprentissage. Il est possible d'effectuer un post-traitement à partir de ces informations (ex. déviance, degrés de liberté), nous verrons cela plus loin (ex. Test de significativité globale, page 13).

Ensuite, nous avons la matrice de confusion en resubstitution (calculée sur les données d'apprentissage), mais avec une particularité : le seuil d'affectation, qui est normalement de 0.5 dans un problème binaire, a été choisi (0.2798…) de manière à maximiser le F1-Score (F-Measure). Avec pour deux conséquences : nous aurions obtenu une matrice différente s'il avait utilisé le seuil usuel ; ce choix impactera également les prédictions à venir sur l'échantillon test (page 15).

H2O enfonce le clou en produisant, pour chaque indicateur de performance (F1, F2, F0.5, taux de reconnaissance [accuracy], précision [precision], rappel / sensibilité [recall], etc.), le seuil optimal d'affectation (threshold) et la valeur associée (value).

```
Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.27983856706746923:
```

```
          negative    positive    Error    Rate
--------  ----------  ----------  -------  -------------
negative  243         93          0.2768   (93.0/336.0)
positive  33          140         0.1908   (33.0/173.0)
Total     276         233         0.2475   (126.0/509.0)

Maximum Metrics: Maximum metrics at their respective thresholds
metric                       threshold    value     idx
---------------------------  -----------  --------  -----
max f1                       0.279839     0.689655  205
max f2                       0.181572     0.810811  258
max f0point5                 0.655162     0.710572  91
max accuracy                 0.551291     0.787819  114
max precision                0.997423     1         0
max recall                   0.0878787    1         322
max specificity              0.997423     1         0
max absolute_mcc             0.379549     0.517518  160
max min_per_class_accuracy   0.309139     0.761905  190
max mean_per_class_accuracy  0.279839     0.766231  205
```

Le tableau ci-dessous détaille le calcul de la courbe de gain (gain chart ou cumulative lift chart selon les logiciels). Son intérêt n'est pas mirobolant et alourdit l'affichage je trouve.

```
Gains/Lift Table: Avg response rate: 33,99 %, avg score: 33,99 %
```

group	cumulative_data_fraction	lower_threshold	lift	cumulative_lift	response_rate	score	cumulative_response_rate	cumulative_score	capture_rate	cumulative_capture_rate	gain	cumulative_gain
1	0.0117878	0.959246	2.45183	2.45183	0.833333	0.974978	0.833333	0.974978	0.0289017	0.0289017	145.183	145.183
2	0.021611	0.936059	2.35376	2.40725	0.8	0.94901	0.818182	0.963174	0.0231214	0.0520231	135.376	140.725
3	0.0314342	0.915047	2.9422	2.57442	1	0.925198	0.875	0.951307	0.0289017	0.0809249	194.22	157.442
4	0.0412574	0.89374	2.9422	2.66199	1	0.90365	0.904762	0.93996	0.0289017	0.109827	194.22	166.199
5	0.0510806	0.878179	2.9422	2.71587	1	0.886609	0.923077	0.9297	0.0289017	0.138728	194.22	171.587
6	0.100196	0.789567	2.23607	2.48068	0.76	0.840051	0.843137	0.885754	0.109827	0.248555	123.607	148.068
7	0.151277	0.728646	2.15007	2.36904	0.730769	0.751606	0.805195	0.840457	0.109827	0.358382	115.007	136.904
8	0.200393	0.65323	2.35376	2.3653	0.8	0.68908	0.803922	0.803355	0.115607	0.473988	135.376	136.53
9	0.300589	0.453488	1.38456	2.03838	0.470588	0.554686	0.69281	0.720465	0.138728	0.612717	38.4563	103.838
10	0.400786	0.327567	1.26918	1.84608	0.431373	0.389769	0.627451	0.637791	0.127168	0.739884	26.9183	84.6084
11	0.500982	0.255229	0.980732	1.67301	0.333333	0.284955	0.568627	0.567224	0.0982659	0.83815	-1.92678	67.3014
12	0.599214	0.184006	0.941503	1.55309	0.32	0.218943	0.527869	0.510129	0.0924855	0.930636	-5.84971	55.3094
13	0.699411	0.122432	0.461521	1.39672	0.156863	0.15144	0.474719	0.458744	0.0462428	0.976879	-53.8479	39.6717
14	0.799607	0.0842905	0.230761	1.25061	0.0784314	0.0992038	0.425061	0.413691	0.0231214	1	-76.9239	25.0614
15	0.899804	0.0466038	0	1.11135	0	0.0648993	0.377729	0.374852	0	1	-100	11.1354
16	1	0.00105129	0	1	0	0.0258428	0.339882	0.339882	0	1	-100	0

Enfin, nous disposons de l'historique de l'optimisation. L'algorithme a convergé au bout de 5 itérations, avec une log-vraisemblance LL de -228.785. La déviance est bien (-2 x LL) = 457.57 vu plus haut.

```
Scoring History:
    timestamp            duration    iterations    negative_log_likelihood    objective
--  -------------------  ----------  ------------  -------------------------  -----------
    2018-12-31 09:20:11  0.000 sec   0             326.247                    0.640957
    2018-12-31 09:20:11  0.038 sec   1             236.889                    0.4654
    2018-12-31 09:20:11  0.040 sec   2             229.16                     0.450216
    2018-12-31 09:20:11  0.042 sec   3             228.786                    0.449481
    2018-12-31 09:20:11  0.043 sec   4             228.785                    0.449479
    2018-12-31 09:20:11  0.044 sec   5             228.785                    0.449479
```

coef() affiche les coefficients du modèle.

```
#affichage des coefficients
reg.coef()
```

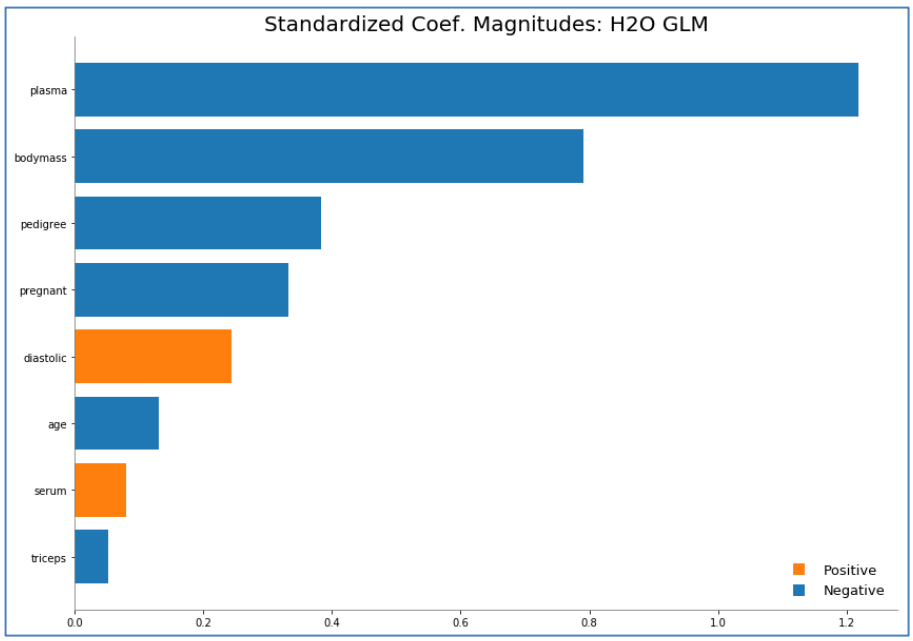
```
{'Intercept': -9.3109910740745,
 'pregnant': 0.10251290143745925,
 'diastolic': -0.012356490310693354,
 'triceps': 0.0033802134565292232,
 'bodymass': 0.09998583311516211,
 'pedigree': 1.1812519689247938,
 'age': 0.011012464469597862,
 'plasma': 0.038648352910967694,
 'serum': -0.0006694305221616449}
```

coef_norm() produit les coefficients standardisés, intéressants pour comparer l'impact des variables dans le modèle.

```
{'Intercept': -0.9708101593539284,
 'pregnant': 0.3324537162601467,
 'diastolic': -0.24385620434917502,
 'triceps': 0.052771090087344416,
 'bodymass': 0.7907131236643259,
 'pedigree': 0.38328833929699774,
 'age': 0.13186352722894912,
 'plasma': 1.2181757375986073,
 'serum': -0.08054404570340926}
```

Les 4 variables les plus influentes sont (dans l'ordre de la valeur absolue des coefficients) : plasma, bodymass, pedigree, pregnant. On peut obtenir un affichage graphique avec le signe de l'impact. Ca fait joli avec des couleurs… on remarque surtout que « plasma » et « bodymass » ont une contribution un peu plus marquée par rapport aux autres variables.

```
#affichage graphique des coefficients normalisées
reg.std_coef_plot()
```



Enfin, nous disposons du tableau complet des coefficients avec les tests de significativité avec pprint_coef(), un peu à la manière de summary() de la procédure glm() de R.

```
#affichage complet avec les tests de significativité
reg.pprint_coef()
```

```
Coefficients: glm coefficients

names      coefficients    std_error    z_value    p_value      standardized_coefficients
---------  --------------  -----------  ---------  -----------  --------------------------
Intercept  -9.31099        0.941792     -9.88646   0            -0.97081
pregnant   0.102513        0.0419376    2.44442    0.0145087    0.332454
diastolic  -0.0123565      0.00658693   -1.87591   0.0606677    -0.243856
triceps    0.00338021      0.00902742   0.374438   0.708078     0.0527711
bodymass   0.0999858       0.0196895    5.07813    3.81171e-07  0.790713
pedigree   1.18125         0.399051     2.96015    0.00307485   0.383288
age        0.0110125       0.0118915    0.92608    0.354404     0.131864
plasma     0.0386484       0.00484544   7.97622    1.55431e-15  1.21818
serum      -0.000669431    0.00108466   -0.61718   0.537116     -0.080544
```

On note la cohérence des résultats (ouf !). Les variables les plus significatives, avec les p-values les plus faibles, sont celles qui présentent les coefficients standardisés les plus élevés en valeur absolue.

Nous pouvons accéder nommément aux vecteurs des valeurs en manipulant les champs internes de l'objet GLM. Pour les écarts-type estimés par exemple.

```
#accéder à un champ en particlier, ex. ecarts-type
reg._model_json['output']['coefficients_table']['std_error']
```

```
[0.9417923763702468,
 0.04193758264822159,
 0.006586932717509827,
 0.00902742267534447,
 0.019689504657106317,
 0.39905080347872535,
 0.011891476650705484,
 0.004845444755115984,
 0.0010846604491290056]
```

Test de significativité globale du modèle. Il est possible de réaliser des calculs supplémentaires à partir des résultats intermédiaires fournis par l'objet GLM. Voyons ce qu'il en par exemple du test de rapport de vraisemblance (Pratique de la régression logistique, section 3.2.4).

Nous produisons tout d'abord la statistique de test en effectuant la différence entre les déviances « null » et « residual ».

```
#stat. test du rapport de vraisemblance
LR = reg.null_deviance() - reg.residual_deviance()
print(LR)
```

```
194.9251739234195
```

Ensuite les degrés de liberté :

```
#degrés de liberté
ddl = reg.null_degrees_of_freedom() - reg.residual_degrees_of_freedom()
print(ddl)
```

```
8
```

Nous calculons enfin la p-value en passant par la librairie Scipy pour l'utilisation de la fonction de distribution cumulée (cdf) de la loi du KHI-2 (cf. « Probabilités et quantiles sous Excel, R et Python »).

```
#librairie scipy
import scipy.stats as stats

#p-value de significativité globale
pvalue = 1.0 - stats.chi2.cdf(x=LR,df=ddl)
print(pvalue)
```

```
0.0
```

Le modèle est globalement très significatif.

Performances en test. H2O propose un outil qui permet de reproduire sur l'échantillon test les évaluations précédemment réalisées lors de l'apprentissage (show).

```
#evaluation des performances sur l'échantillon test
reg.model_performance(pimaTest)
```

```
ModelMetricsBinomialGLM: glm
** Reported on test data. **

MSE: 0.16413828925910307
RMSE: 0.4051398391408861
LogLoss: 0.5262065467958137
Null degrees of freedom: 258
Residual degrees of freedom: 250
Null deviance: 341.27020915066095
Residual deviance: 272.5749912402315
AIC: 290.5749912402315
AUC: 0.8087291399229781
pr_auc: 0.7100942425162344
Gini: 0.6174582798459562
```

Bien sûr, les performances mesurées sont moindres (la plupart du temps) sur l'échantillon test. Nous apprenons entres autres que la « vraie » (guillemets parce que mesurée sur un échantillon, et de ce fait soumise à une certaine variabilité) valeur de l'AUC (l'aire sous la courbe ROC) est de 0.8087, et non pas de 0.8519 comme on pouvait le croire lorsqu'elle était calculée sur l'ensemble d'apprentissage (page 10).

Ici également, nous avons une matrice de confusion avec un seuil d'affectation (0.31883...) optimisé pour le F1-Score. La même opération est réitérée pour les autres indicateurs (couple seuil – valeur de la mesure). Remarque : Autant cette approche d'optimisation des seuils me semblait intéressante et originale sur l'échantillon d'apprentissage, autant je suis assez réservé ici. En procédant ainsi, nous

utilisons l'échantillon test pour l'estimation des paramètres (les seuils d'affectation) permettant d'optimiser les mesures d'évaluation. Ce n'est pas vraiment son rôle. Il doit intervenir comme un arbitre impartial. En revanche, utiliser un échantillon spécifique à cet effet, on parle alors d'ensemble de validation (« validation set », je préfère le terme « tuning set »), peut être une piste possible. Nous reviendrons sur ce thème du troisième échantillon plus loin car H2O propose des outils originaux dans cette optique (section 4.1).

```
Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.3183236641504005:
          negative    positive    Error    Rate
--------  ----------  ----------  -------  -------------
negative  130         34          0.2073   (34.0/164.0)
positive  25          70          0.2632   (25.0/95.0)
Total     155         104         0.2278   (59.0/259.0)

Maximum Metrics: Maximum metrics at their respective thresholds
metric                        threshold    value     idx
----------------------------  -----------  --------  -----
max f1                        0.318324     0.703518  103
max f2                        0.137252     0.782374  175
max f0point5                  0.619581     0.723473  53
max accuracy                  0.468013     0.779923  73
max precision                 0.943157     0.875     7
max recall                    0.00846608   1         256
max specificity               0.985847     0.993902  0
max absolute_mcc              0.318324     0.52058   103
max min_per_class_accuracy    0.318324     0.736842  103
max mean_per_class_accuracy   0.318324     0.764763  103
Gains/Lift Table: Avg response rate: 36,68 %, avg score: 35,69 %
```

De nouveau, nous avons la table de construction de la courbe de gain.

group	cumulative_data_fraction	lower_threshold	lift	cumulative_lift	response_rate	score	cumulative_response_rate	cumulative_score	capture_rate	cumulative_capture_rate	gain	cumulative_gain
1	0.011583	0.756597	1.81754	1.81754	0.666667	0.898794	0.666667	0.898794	0.0210526	0.0210526	81.7544	81.7544
2	0.023166	0.717013	0.908772	1.36316	0.333333	0.729667	0.5	0.814231	0.0105263	0.0315789	-9.12281	36.3158
3	0.030888	0.677128	1.36316	1.36316	0.5	0.699708	0.5	0.7856	0.0105263	0.0421053	36.3158	36.3158
4	0.042471	0.635533	1.81754	1.48708	0.666667	0.651377	0.545455	0.748994	0.0210526	0.0631579	81.7544	48.7081
5	0.0501931	0.617627	2.72632	1.67773	1	0.630196	0.615385	0.730717	0.0210526	0.0842105	172.632	67.7733
6	0.100386	0.508364	1.67773	1.67773	0.615385	0.544803	0.615385	0.63776	0.0842105	0.168421	67.7733	67.7733
7	0.150579	0.444023	1.46802	1.60783	0.538462	0.47734	0.589744	0.584287	0.0736842	0.242105	46.8016	60.7827
8	0.200772	0.412099	2.09717	1.73016	0.769231	0.4306	0.634615	0.545865	0.105263	0.347368	109.717	73.0162
9	0.301158	0.375876	1.36316	1.60783	0.5	0.394803	0.589744	0.495511	0.136842	0.484211	36.3158	60.7827
10	0.401544	0.350976	0.838866	1.41559	0.307692	0.362626	0.519231	0.46229	0.0842105	0.568421	-16.1134	41.5587
11	0.501931	0.332842	0.943725	1.32121	0.346154	0.341249	0.484615	0.438082	0.0947368	0.663158	-5.62753	32.1215
12	0.598456	0.310453	0.872421	1.24883	0.32	0.318866	0.458065	0.418853	0.0842105	0.747368	-12.7579	24.8829
13	0.698842	0.294305	0.524291	1.14475	0.192308	0.301547	0.41989	0.402003	0.0526316	0.8	-47.5709	14.4751
14	0.799228	0.26633	0.943725	1.1195	0.346154	0.279129	0.410628	0.386569	0.0947368	0.894737	-5.62753	11.9502
15	0.899614	0.245264	0.419433	1.04138	0.153846	0.255004	0.381974	0.371888	0.0421053	0.936842	-58.0567	4.13824
16	1	0.188054	0.62915	1	0.230769	0.222138	0.366795	0.356855	0.0631579	1	-37.085	0

Performances en test (bis) – Prédiction + Confrontation observé-prédiction. Nous revenons sur une procédure d'évaluation un peu plus classique ici. Nous réalisons une prédiction en aveugle sur l'échantillon test dans un premier temps, puis nous confronterons les classes prédites avec les observées. La prédiction est relativement facile à réaliser avec la commande predict().

```
#prediction - conversion en format data frame
#attention au seuil utilisé pour l'affectation
predReg = reg.predict(pimaTest).as_data_frame()
predReg.head(20)
```

```
    predict  negative  positive    StdErr
0   positive  0.219803  0.780197  0.443473
1   positive  0.053610  0.946390  0.780507
2   negative  0.946779  0.053221  0.316262
3   negative  0.981609  0.018391  0.672891
4   negative  0.813203  0.186797  0.339791
5   positive  0.242826  0.757174  0.520822
6   positive  0.545245  0.454755  0.237925
7   positive  0.055552  0.944448  0.440860
8   negative  0.755804  0.244196  0.350869
9   positive  0.638720  0.361280  0.310989
10  positive  0.663433  0.336567  0.312859
11  positive  0.380419  0.619581  0.508836
12  negative  0.923914  0.076086  0.328067
13  negative  0.928833  0.071167  0.272955
14  positive  0.158008  0.841992  0.358008
15  negative  0.984439  0.015561  0.374865
16  positive  0.095961  0.904039  0.369348
17  negative  0.994878  0.005122  0.718305
18  positive  0.671361  0.328639  0.294836
19  negative  0.722359  0.277641  0.234975
```

Plusieurs éléments attirent notre attention :

- predict() fournit les classes prédites, les probabilités d'appartenance aux classes et les écarts-types associées ;
- A partir de ces probabilités et de leurs écarts-type, il est possible de produire les intervalles de confiance, importantes dans certaines études (Pratique de la régression logistique, section 4.2).
- Habituellement, on prend le maximum des probabilités conditionnelles pour l'attribution des classes, ce qui correspond à un seul d'affectation de 0.5. Nous constatons que ce n'est pas le cas pour nos prédictions en affichant le détail pour les 20 premiers individus de l'échantillon test. Voyons cela plus précisément.

Pour les individus n°6, 9, 10, 18 (et il en en a d'autres sûrement encore dans le fichier), malgré que P(Y = + / X) < P(Y = - / X), la classe « positive » leur est attribuée. Pourquoi ? La réponse est dans la matrice de confusion proposée par la commande show() vue plus haut (page 10). Le seuil utilisé est 0.27983856706746923 pour optimiser le F1-Score. De fait, tous les individus pour lesquels l'estimation de P(Y = + / X) est supérieur à ce seuil se voient attribuer la classe « positive ». A contrario, lorsque P(Y = + / X) < 0.27983856706746923, la prédiction est « negative ».

Calculons la matrice de confusion sur l'échantillon test avec ce procédé de prédiction.

```
#matrice de confusion en test -- avec le seuil de H20
import pandas
pandas.crosstab(pimaTest.as_data_frame()["diabete"],predReg.predict)
```

```
predict   negative  positive
diabete
negative       117        47
positive        24        71
```

Et le taux d'erreur est :

```
#scikit-learn
from sklearn import metrics

#taux d'erreur
print(1-metrics.accuracy_score(pimaTest.as_data_frame()["diabete"],predReg.predict))
```

```
0.27413127413127414
```

Tandis que le F1-Score :

```
#F1-Score
print(metrics.f1_score(pimaTest.as_data_frame()["diabete"],predReg.predict,pos_label="positive"))
```

```
0.6666666666666667
```

Performances en test (ter) – Utilisation du seuil usuel de 0.5. On comprend la démarche de H20 d'optimiser les seuils d'affectation en fonction des mesures de performances à privilégier. Mais est-ce vraiment à bon escient ? Voyons ce qu'il en est sur notre échantillon test si nous utilisons le seuil usuel de 0.5. Deux questions sont sous-jacentes à l'opération : le taux d'erreur sera-t-il meilleur (on ne sait pas) ? et le F1-Score moins bon (normalement oui) ?

Calculons les prédictions avec le seuil 0.5.

```
#prédiction avec le seuil = 0.5
import numpy
predReg2 = numpy.repeat("negative",pimaTest.shape[0])
predReg2[predReg.positive > 0.5] = "positive"
```

La nouvelle version de la matrice de confusion est :

```
#matrice de confusion en test -- seuil = 0.5
pandas.crosstab(pimaTest.as_data_frame()["diabete"],predReg2)
```

```
col_0     negative  positive
diabete
negative       147        17
positive        43        52
```

Avec un taux d'erreur :

```
#taux d'erreur
print(1-metrics.accuracy_score(pimaTest.as_data_frame()["diabete"],predReg2))
```

```
0.23166023166023164
```

Plus faible, donc meilleur. Avoir un avis préétabli sur les valeurs qu'il devait prendre n'était pas évident parce que le F1-Score et le taux d'erreur ne sont pas structurellement concomitants ou opposés.

Si le F1-Score est aussi meilleur (plus élevé), là il y aurait un problème pour H2O. Voyons voir :

```
#F1-Score
print(metrics.f1_score(pimaTest.as_data_frame()["diabete"],predReg2,pos_label="positive"))
```

```
0.6341463414634145
```

Il est moins bon (il était de 0.6666… avec le seuil proposé par H2O). L'opération d'optimisation des seuils d'affectation de H2O est pertinente, au moins en ce qui concerne le F1-Score sur nos données.

3.3 Random forest

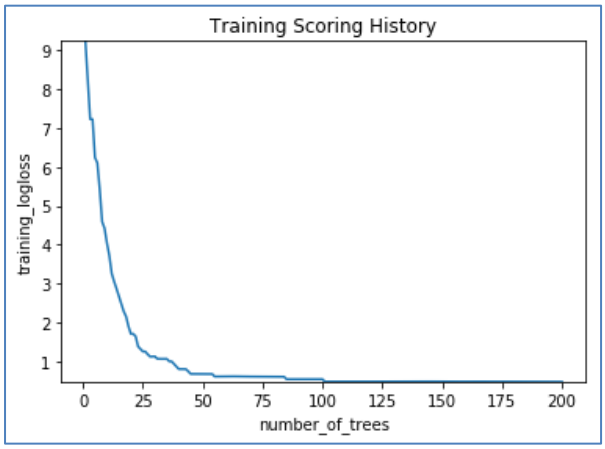
La méthode Random Forest est basée sur l'agrégation d'arbres de décision élaborés de façon à maximiser leur décorrélation. Ainsi, nous maximisons leur efficacité lorsque nous les faisons coopérer.

Initialisation et apprentissage. Sous H2O, nous utilisons la classe H2ORandomForestEstimator. Nous l'instancions avec (ntrees = 200) arbres, de profondeur maximum (max_depth = 50) et avec des feuilles comportant au moins (min_rows = 1) observation. Pour rappel, plus les arbres sont grands, plus Random Forest est performant.

```
#random forest
from h2o.estimators import H2ORandomForestEstimator

#instanciation
rf = H2ORandomForestEstimator(seed=100,ntrees=200,max_depth=50,min_rows=1)

#apprentissage
rf.train(x=pimaTrain.columns[:-1],y="diabete",training_frame=pimaTrain)
```

Nous pouvons suivre l'évolution de la fonction de perte avec plot().

```
#evolution de l'apprentissage
rf.plot()
```



Une centaine d'arbre aurait suffi apparemment. Mais Random Forest possède la propriété très avantageuse d'être robuste au surapprentissage. L'empilement des arbres ne lui porte pas préjudice. Ce n'est pas le cas des autres méthodes ensemblistes (cf. Gradient Boosting, section 3.4).

Un résumé du modèle est disponible.

```
#résumé
rf.summary()
```

```
Model Summary:
  number_of_trees  number_of_internal_trees  model_size_in_bytes  min_depth  max_depth  mean_depth  min_leaves  max_leaves  mean_leaves
--  ---------------  -------------------------  -------------------  ---------  ---------  ----------  ----------  ----------  -------------
  200              200                     234396         11         23       14.27          69         102         88.405
```

200 arbres ont bien été générés, leur profondeur moyenne est de 14.27, avec un nombre de feuilles moyen de 88.405 (nombre de règles par arbre). Globalement, les arbres individuels sont assez complexes compte tenu du faible effectif de l'échantillon d'apprentissage et du faible nombre de variables. Ce qui convient parfaitement à Random Forest.

Nous affichons les résultats de l'apprentissage.

```
#affichage
rf.show()
```

Une grande partie des sorties sont les mêmes que pour la régression logistique.

```
Model Details
=============
H2ORandomForestEstimator :  Distributed Random Forest
Model Key:  DRF_model_python_1546242989411_5


ModelMetricsBinomial: drf
** Reported on train data. **

MSE: 0.15919059185360485
RMSE: 0.3989869569968483
LogLoss: 0.47971422895094906
Mean Per-Class Error: 0.22727429121937792
AUC: 0.826641205615194
pr_auc: 0.6756652021771573
Gini: 0.6532824112303881

Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.29032258064516125:
          negative    positive    Error    Rate
--------  ----------  ----------  -------  -------------
negative  226         110         0.3274   (110.0/336.0)
positive  22          151         0.1272   (22.0/173.0)
Total     248         261         0.2593   (132.0/509.0)

Maximum Metrics: Maximum metrics at their respective thresholds
metric                       threshold    value     idx
---------------------------  -----------  --------  -----
max f1                       0.290323     0.695853  224
max f2                       0.208861     0.813253  261
max f0point5                 0.394366     0.639594  170
max accuracy                 0.394366     0.756385  170
max precision                0.941176     1         0
max recall                   0.0132749    1         393
```

```
max specificity                 0.941176     1          0
max absolute_mcc                0.285714     0.516924   227
max min_per_class_accuracy      0.36875      0.751445   179
max mean_per_class_accuracy     0.290323     0.772726   224
Gains/Lift Table: Avg response rate: 33,99 %, avg score: 34,13 %
```

group	cumulative_data_fraction	lower_threshold	lift	cumulative_lift	response_rate	score	cumulative_response_rate	cumulative_score	capture_rate	cumulative_capture_rate	gain	cumulative_gain
1	0.0117878	0.892591	2.9422	2.9422	1	0.916807	1	0.916807	0.0346821	0.0346821	194.22	194.22
2	0.021611	0.867202	2.35376	2.67472	0.8	0.875826	0.909091	0.89818	0.0231214	0.0578035	135.376	167.472
3	0.0314342	0.842836	2.35376	2.57442	0.8	0.856135	0.875	0.885041	0.0231214	0.0809249	135.376	157.442
4	0.0412574	0.813067	2.35376	2.52188	0.8	0.822496	0.857143	0.870149	0.0231214	0.104046	135.376	152.188
5	0.0530452	0.794521	1.96146	2.39735	0.666667	0.803867	0.814815	0.85542	0.0231214	0.127168	96.1464	139.735
6	0.102161	0.733333	2.11838	2.26323	0.72	0.758314	0.769231	0.808734	0.104046	0.231214	111.838	126.323
7	0.151277	0.676674	1.76532	2.10157	0.6	0.708553	0.714286	0.776208	0.0867052	0.317919	76.5318	110.157
8	0.200393	0.606474	2.11838	2.10569	0.72	0.646168	0.715686	0.744335	0.104046	0.421965	111.838	110.569
9	0.300589	0.484984	1.38456	1.86531	0.470588	0.540574	0.633987	0.676415	0.138728	0.560694	38.4563	86.5314
10	0.400786	0.390142	1.67301	1.81724	0.568627	0.440202	0.617647	0.617362	0.16763	0.728324	67.3014	81.7239
11	0.500982	0.296296	1.21149	1.69609	0.411765	0.337485	0.576471	0.561386	0.121387	0.849711	21.1493	69.609
12	0.601179	0.208333	0.865352	1.55763	0.294118	0.253178	0.529412	0.510018	0.0867052	0.936416	-13.4648	55.7633
13	0.699411	0.135627	0.176532	1.36366	0.06	0.169319	0.463483	0.462167	0.017341	0.953757	-82.3468	36.3659
14	0.799607	0.0806486	0.288451	1.22893	0.0980392	0.109744	0.41769	0.418006	0.0289017	0.982659	-71.1549	22.8927
15	0.899804	0.0297629	0.0576901	1.09851	0.0196078	0.0565035	0.373362	0.377752	0.00578035	0.988439	-94.231	9.85057
16	1	0	0.11538	1	0.0392157	0.0135423	0.339882	0.341259	0.0115607	1	-88.462	0

Scoring History:

timestamp	duration	number_of_trees	training_rmse	training_logloss	training_auc	training_pr_auc	training_lift	training_classification_error
2018-12-31 12:39:39	0.029 sec	0.0	nan	nan	nan	nan	nan	nan
2018-12-31 12:39:39	0.114 sec	1.0	0.5170876899950192	9.234966950510872	0.6970729751403368	0.17342724833891493	1.6609173969793027	0.26737967914438504
2018-12-31 12:39:39	0.133 sec	2.0	0.5051304442186866	8.255273898433419	0.7041988512576748	0.2125237336467729	1.805438780872307	0.28052805280528054
2018-12-31 12:39:39	0.146 sec	3.0	0.48392778047812063	7.232395326595819	0.7341575091575091	0.21399181341316037	1.8819455293443732	0.27202072538860106
2018-12-31 12:39:39	0.158 sec	4.0	0.49606216837924594	7.227187858208416	0.7176542010684799	0.23723890187268026	1.8105824811027122	0.28935185185185186
2018-12-31 12:39:41	1.425 sec	196.0	0.39912316768859674	0.480235029632076	0.8258240434902284	0.6740859606817491	2.9421965317919074	0.2455795677799607
2018-12-31 12:39:41	1.431 sec	197.0	0.3991662336412445	0.4799761162728152	0.826082094687586	0.6750827420270649	2.9421965317919074	0.2475442043222004
2018-12-31 12:39:41	1.437 sec	198.0	0.3990263791624162	0.4797266614059679	0.8263315441783651	0.6756624017222859	2.9421965317919074	0.25147347740667975
2018-12-31 12:39:41	1.443 sec	199.0	0.39907275002363923	0.4798499647080816	0.8264261629507295	0.675139195411401	2.9421965317919074	0.2593320235756385
2018-12-31 12:39:41	1.449 sec	200.0	0.3989869569968483	0.47971422895094906	0.826641205615194	0.6756652021771573	2.9421965317919074	0.2593320235756385

```
See the whole table with table.as_data_frame()

Variable Importances:
```

variable	relative_importance	scaled_importance	percentage
plasma	4136.33	1	0.270722
bodymass	2709.15	0.654965	0.177313
pedigree	1823.28	0.440797	0.119333
age	1695.64	0.409937	0.110979
serum	1337.48	0.32335	0.0875377
diastolic	1249.47	0.302072	0.0817775
pregnant	1224.19	0.29596	0.0801229

Cette dernière partie attire notre attention. Random Forest sait évaluer la contribution des variables dans le modèle. PLASMA et BODYMASS se démarquent à nouveau. Une partie des variables seulement sont affichées lorsqu'elles sont nombreuses (les meilleurs et les pires). Nous utilisons varimp() pour obtenir la liste exhaustive.

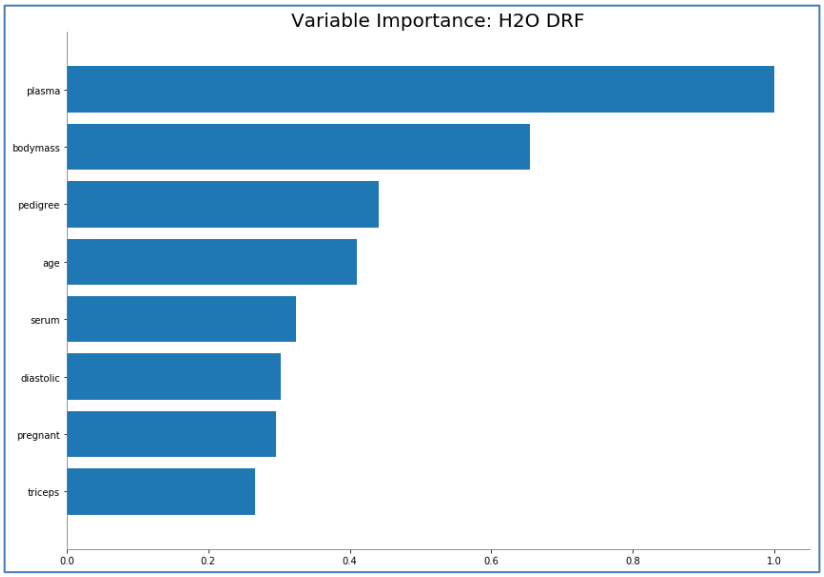
```
#ou tabulaire
```

```
pandas.DataFrame(rf.varimp())
```

```
         0            1         2         3
0     plasma  4136.327148  1.000000  0.270722
1   bodymass  2709.148926  0.654965  0.177313
2   pedigree  1823.282227  0.440797  0.119333
3        age  1695.635498  0.409937  0.110979
4      serum  1337.479370  0.323350  0.087538
5  diastolic  1249.468872  0.302072  0.081777
6   pregnant  1224.188965  0.295960  0.080123
7    triceps  1103.356201  0.266748  0.072214
```

Un graphique peut faire l'affaire également.

```
#importance - graphique
rf.varimp_plot()
```



Contrairement à GLM, nous ne disposons pas du sens (signe) de la relation.

Evaluation en test. De nouveau, model_performance() propose une vue globale des performances sur l'échantillon passé en paramètre.

```
#evaluation
rf.model_performance(pimaTest)
```

```
ModelMetricsBinomial: drf
** Reported on test data. **

MSE: 0.1677196390090528
RMSE: 0.40953588244383765
LogLoss: 0.5026371627225671
Mean Per-Class Error: 0.24881258023106545
AUC: 0.812772785622593
pr_auc: 0.7209664279934564
Gini: 0.6255455712451861
```

```
Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.39:
          negative    positive    Error    Rate
--------  ----------  ----------  -------  -------------
negative  129         35          0.2134   (35.0/164.0)
positive  27          68          0.2842   (27.0/95.0)
Total     156         103         0.2394   (62.0/259.0)

Maximum Metrics: Maximum metrics at their respective thresholds
metric                        threshold    value     idx
----------------------------  -----------  --------  -----
max f1                        0.39         0.686869  80
max f2                        0.13         0.791815  144
max f0point5                  0.464712     0.687361  69
max accuracy                  0.464712     0.76834   69
max precision                 0.94         1         0
max recall                    0.055        1         167
max specificity               0.94         1         0
max absolute_mcc              0.464712     0.495201  69
max min_per_class_accuracy    0.36         0.726316  89
max mean_per_class_accuracy   0.39         0.751187  80

Gains/Lift Table: Avg response rate: 36,68 %, avg score: 34,29 %
```

group	cumulative_data_fraction	lower_threshold	lift	cumulative_lift	response_rate	score	cumulative_response_rate	cumulative_score	capture_rate	cumulative_capture_rate	gain	cumulative_gain
1	0.011583	0.8971	2.72632	2.72632	1	0.921667	1	0.921667	0.0315789	0.0315789	172.632	172.632
2	0.030888	0.855	2.72632	2.72632	1	0.866333	1	0.887083	0.0526316	0.0842105	172.632	172.632
3	0.030888	0.8439	0	2.72632	0	0	1	0.887083	0	0.0842105	-100	172.632
4	0.042471	0.8368	2.72632	2.72632	1	0.84	1	0.874242	0.0315789	0.115789	172.632	172.632
5	0.0501931	0.8015	1.36316	2.5166	0.5	0.8225	0.923077	0.866282	0.0105263	0.126316	36.3158	151.66
6	0.108108	0.72	2.36281	2.43421	0.866667	0.757006	0.892857	0.807741	0.136842	0.263158	136.281	143.421
7	0.150579	0.676917	1.48708	2.16707	0.545455	0.695295	0.794872	0.776026	0.0631579	0.326316	48.7081	116.707
8	0.200772	0.6045	1.67773	2.04474	0.615385	0.64642	0.75	0.743624	0.0842105	0.410526	67.7733	104.474
9	0.301158	0.498	1.46802	1.8525	0.538462	0.553041	0.679487	0.680097	0.147368	0.557895	46.8016	85.2497
10	0.405405	0.3825	1.51462	1.76561	0.555556	0.447352	0.647619	0.620248	0.157895	0.715789	51.462	76.5614
11	0.505792	0.29	0.734008	1.56087	0.269231	0.337994	0.572519	0.564228	0.0736842	0.789474	-26.5992	56.0868
12	0.598456	0.201769	0.795175	1.44231	0.291667	0.246002	0.529032	0.514955	0.0736842	0.863158	-20.4825	44.2309
13	0.702703	0.13	0.706823	1.3332	0.259259	0.168283	0.489011	0.463525	0.0736842	0.936842	-29.3177	33.3198
14	0.799228	0.0732917	0.218105	1.19853	0.08	0.0999526	0.439614	0.419616	0.0210526	0.957895	-78.1895	19.8525
15	0.899614	0.038	0.419433	1.11159	0.153846	0.0548552	0.407725	0.378913	0.0421053	1	-58.0567	11.1588
16	1	0	0	1	0	0.0205995	0.366795	0.342943	0	1	-100	0

Encore une fois ici, la modulation des seuils d'affectation ne donne pas une image objective des performances du modèle. Nous préférons réaliser manuellement la séquence prédiction et confrontation sur l'échantillon test.

```
#prediction - de nouveau voir le seuil d'affectation
predRf = rf.predict(pimaTest).as_data_frame()
print(predRf.head(10))
```

```
    predict  negative  positive
0  positive  0.527500  0.472500
1  positive  0.267500  0.732500
2  negative  0.945000  0.055000
3  negative  0.935000  0.065000
4  negative  0.871538  0.128462
5  positive  0.670000  0.330000
```

```
6  positive  0.617500  0.382500
7  positive  0.293167  0.706833
8  positive  0.697500  0.302500
9  positive  0.322500  0.677500
```

```
#F1-score
print(metrics.f1_score(pimaTest.as_data_frame()["diabete"],predRf.predict,pos_label="positive"))
```

```
0.6696428571428572
```

3.4 Gradient boosting

Gradient boosting est une variante du boosting où l'on exploite les écarts entre les probabilités d'affectation estimées et les classes d'appartenance pour corriger les modèles (arbres) successifs (en schématisant parce que les présentations usuelles ne sont pas très explicites souvent). Par rapport à Random Forest, elle est plus efficace car nécessite peu d'arbres, mais elle est autrement plus sujette au surapprentissage. Le paramétrage joue un rôle essentiel pour cette méthode.

Initialisation et apprentissage. Justement, dans cette section, nous allons sciemment pénaliser la méthode en créant trop d'arbres (ntrees = 1500) dimensionnés raisonnablement (max_depth = 5).

```
#gradient boosting
from h2o.estimators import H2OGradientBoostingEstimator

#instanciation
gb = H2OGradientBoostingEstimator(seed=100,ntrees=1500,max_depth=5)

#apprentissage
gb.train(x=pimaTrain.columns[:-1],y="diabete",training_frame=pimaTrain)

#évolution
gb.plot()
```

Manifestement, nous avons trop d'arbres. Pour un Random Forest, l'information était anecdotique. Pour Gradient Boosting, c'est un problème potentiel.

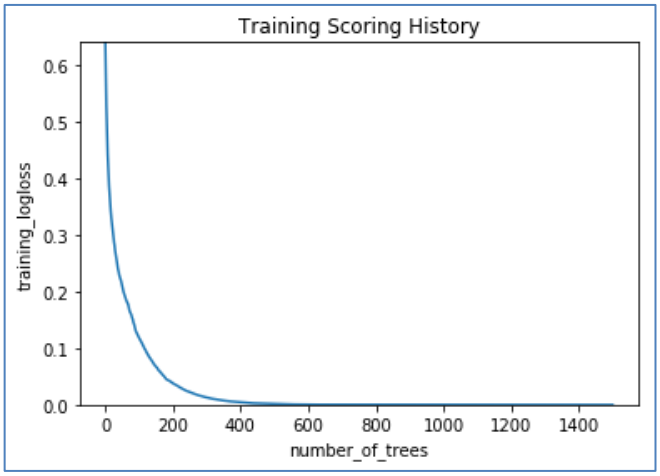


Figure 2 : Gradient Boosting - Evolution de la perte en fonction du nombre d'arbres

L'affichage des résultats (show) n'apporte rien de particulier par rapport aux autres approches. On notera quand même la valeur invraisemblable du seuil d'affectation proposé (0.999999652004392) pour maximiser le F1-Score. Forcément les prédictions seront faussées par la suite.

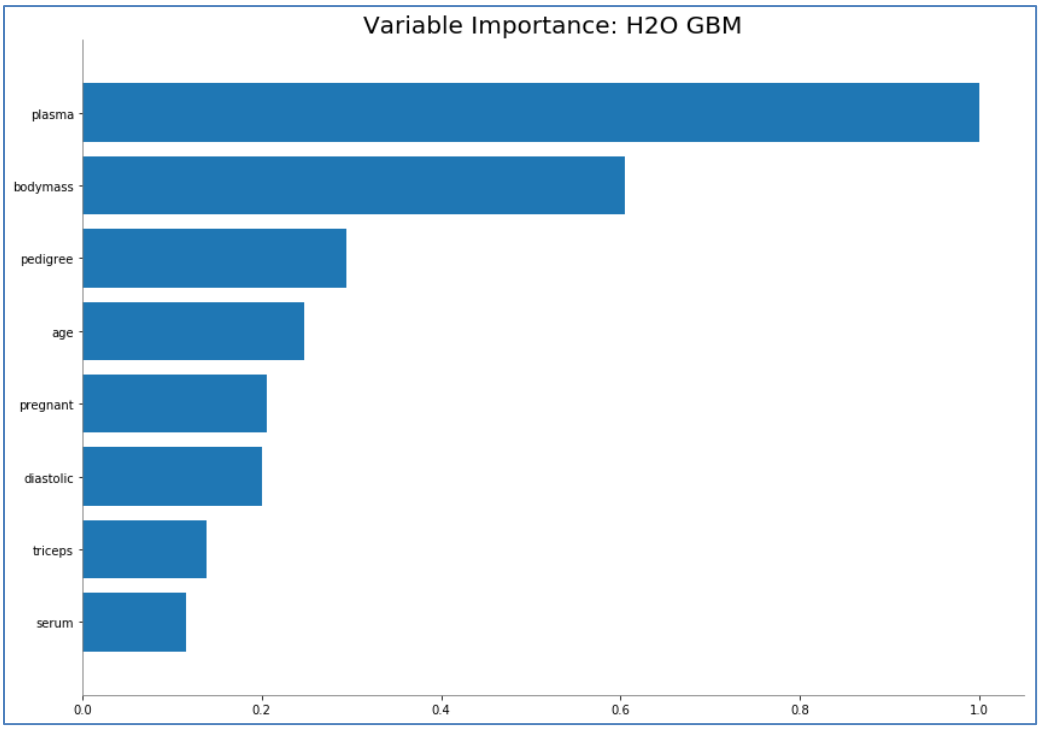
```
#affichage
gb.show()
```

```
Model Details
=============
H2OGradientBoostingEstimator :  Gradient Boosting Machine
Model Key:  GBM_model_python_1546336908592_208


ModelMetricsBinomial: gbm
** Reported on train data. **

MSE: 7.54883516641043e-15
RMSE: 8.688403286226089e-08
LogLoss: 5.6788466738592787e-08
Mean Per-Class Error: 0.0
AUC: 1.0
pr_auc: 0.9942196531791908
Gini: 1.0

Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.999999652004392:
          negative    positive    Error    Rate
--------  ----------  ----------  -------  -----------
negative  336         0           0        (0.0/336.0)
positive  0           173         0        (0.0/173.0)
Total     336         173         0        (0.0/509.0)

Maximum Metrics: Maximum metrics at their respective thresholds
metric                        threshold    value    idx
----------------------------  -----------  -------  -----
max f1                        1            1        63
max f2                        1            1        63
max f0point5                  1            1        63
max accuracy                  1            1        63
max precision                 1            1        0
max recall                    1            1        63
max specificity               1            1        0
max absolute_mcc              1            1        63
max min_per_class_accuracy    1            1        63
max mean_per_class_accuracy   1            1        63
Gains/Lift Table: Avg response rate: 33,99 %, avg score: 33,99 %

  group  cumulative_data_fraction  lower_threshold  lift  cumulative_lift  response_rate  score  cumulative_response_rate  cumulative_score  capture_rate  cumulative_capture_rate  gain  cumulative_gain
-- -------  --------------------------  ----------------  ------  ---------------  -------------  -----------  --------------------------  ----------------  -------------  -----------------------  ------  -----------------
  1  0.0117878  1  2.9422  2.9422  1  1  1  1  0.0346821  0.0346821  194.22  194.22
  2  0.021611  1  2.9422  2.9422  1  1  1  1  0.0289017  0.0635838  194.22  194.22
  3  0.0314342  1  2.9422  2.9422  1  1  1  1  0.0289017  0.0924855  194.22  194.22
  4  0.0412574  1  2.9422  2.9422  1  1  1  1  0.0289017  0.121387  194.22  194.22
  5  0.0510806  1  2.9422  2.9422  1  1  1  1  0.0289017  0.150289  194.22  194.22
  6  0.100196  1  2.9422  2.9422  1  1  1  1  0.144509  0.294798  194.22  194.22
  7  0.151277  1  2.9422  2.9422  1  1  1  1  0.150289  0.445087  194.22  194.22
  8  0.200393  1  2.9422  2.9422  1  1  1  1  0.144509  0.589595  194.22  194.22
```

```
9     0.300589         1             2.9422  2.9422       1          1       1                1          0.294798     0.884393           194.22  194.22
10    0.400786         1.38381e-07   1.1538  2.4951       0.392157   0.392157   0.848039        0.848039     0.115607     1             15.3803  149.51
11    0.500982         7.03745e-08   0       1.99608      0          1.00323e-07  0.678431      0.678431     0            1             -100    99.6078
12    0.599214         3.41984e-08   0       1.66885      0          5.08716e-08  0.567213      0.567213     0            1             -100    66.8852
13    0.699411         7.94929e-09   0       1.42978      0          1.94661e-08  0.485955      0.485955     0            1             -100    42.9775
14    0.799607         2.53533e-10   0       1.25061      0          2.96146e-09  0.425061      0.425061     0            1             -100    25.0614
15    0.899804         1.12535e-12   0       1.11135      0          4.20929e-11  0.377729      0.377729     0            1             -100    11.1354
16    1                1.16158e-18   0       1            0          1.15916e-13  0.339882      0.339882     0            1             -100    0


Scoring History:
    timestamp            duration    number_of_trees    training_rmse           training_logloss        training_auc          training_pr_auc       training_lift         training_classification_error
--- -------------------  ----------  -----------------  ----------------------  ----------------------  --------------------  --------------------  --------------------  -------------------------------
    2019-01-01 11:02:07  0.010 sec   0.0                0.473668940382543       0.6409572589950142      0.5                   0.0                   1.0                   0.6601178781925344
    2019-01-01 11:02:07  0.040 sec   1.0                0.4517980998285188      0.596355572478795       0.8998073217726398    0.7001784702001936    2.9421965317919074    0.2043222003929273
    2019-01-01 11:02:07  0.043 sec   2.0                0.4330048116415299      0.5599178388428999      0.9117206853839802    0.7868339539117197    2.9421965317919074    0.16110019646365423
    2019-01-01 11:02:07  0.052 sec   3.0                0.4171966186238149      0.5301366831053512      0.914025942747041     0.7850080092669086    2.9421965317919074    0.16306483300589392
    2019-01-01 11:02:07  0.062 sec   4.0                0.403747939569155       0.5052469271103167      0.9172601844205891    0.8574007952513998    2.9421965317919074    0.16895874263261296
--- ---                  ---         ---                ---                     ---                     ---                   ---                   ---                   ---
    2019-01-01 11:02:11  3.980 sec   661.0              0.00044695281260692546  0.00029251490509713945  1.0                   0.9942196531791907    2.9421965317919074    0.0
    2019-01-01 11:02:11  3.990 sec   662.0              0.00044404224226484926  0.0002907596983448853   1.0                   0.9942196531791907    2.9421965317919074    0.0
    2019-01-01 11:02:11  4.000 sec   663.0              0.00043846655356657825  0.0002868500012156345   1.0                   0.9942196531791907    2.9421965317919074    0.0
    2019-01-01 11:02:11  4.013 sec   664.0              0.0004329750699803277   0.0002834112408768481   1.0                   0.9942196531791907    2.9421965317919074    0.0
    2019-01-01 11:02:12  4.730 sec   1500.0             8.688403286226089e-08   5.6788466738592787e-08  1.0                   0.9942196531791908    2.9421965317919074    0.0

See the whole table with table.as_data_frame()

Variable Importances:
variable    relative_importance    scaled_importance    percentage
----------  ---------------------  -------------------  ------------
plasma      187.014                1                    0.35623
bodymass    113.052                0.604511             0.215345
pedigree    55.0631                0.294433             0.104886
age         46.2234                0.247165             0.0880478
pregnant    38.5402                0.206082             0.0734126
diastolic   37.5789                0.200941             0.0715814
triceps     25.8928                0.138453             0.0493213
serum       21.6165                0.115588             0.0411758
```

Pour cette méthode également, nous pouvons obtenir l'importance relative des variables. PLASMA et BODYMASS se démarquent encore une fois.

```python
#plotting var importance
gb.varimp_plot()
```



Performances en test. Nous réalisons la séquence prédiction – confrontation sur l'échantillon test pour évaluer la pertinence du modèle.

```
#prediction - de nouveau voir le seuil d'affectation
predGb = gb.predict(pimaTest).as_data_frame()
print(predGb.head(10))
```

```
   predict  negative      positive
0  negative  0.999970  2.969765e-05
1  negative  0.000022  9.999776e-01
2  negative  1.000000  6.229725e-14
3  negative  1.000000  5.406599e-11
4  negative  1.000000  9.093690e-11
5  negative  0.999998  2.022516e-06
6  negative  0.999983  1.749077e-05
7  negative  0.000040  9.999597e-01
8  negative  0.985897  1.410296e-02
9  negative  0.006775  9.932246e-01
```

```
#F1-score
print(metrics.f1_score(pimaTest.as_data_frame()["diabete"],predGb.predict,pos_label="positive"))
```

```
0.3333333333333333
```

Le F1-Score est catastrophique (deux fois moins élevé par rapport aux autres approches). On le sait pourquoi, le seuil d'affectation déterminé par la méthode est inadapté. Nous avons quasi-exclusivement des prédictions négatives (234 prédictions négatives, 25 positives pour être précis). Nous pouvons résoudre ce problème en choisissant un paramétrage adapté de l'algorithme (combinaison nombre d'arbres et leur complexité individuelle). Nous verrons comment nous pouvons

nous faire aider pour cela, sans connaissances approfondies concernant les méthodes de machines learning manipulées, toujours avec les outils de H2O (sections 4.1, 4.2 et 4.3).

3.5 Naive Bayes

Naive Bayes est un algorithme s'appuyant sur l'indépendance des descripteurs conditionnellement aux valeurs de la cible. Sous H2O, les calculs internes se limitent aux estimations des moyennes et écarts-type conditionnels pour l'ensemble des descripteurs. Tu m'étonnes que la méthode soit si rapide, y compris sur des très grandes bases de données. La complexité est linéaire tant en nombre de variables qu'en nombre d'observations, tous les paramètres peuvent être estimés en une seule passe sur l'ensemble d'apprentissage.

Initialisation et apprentissage. Pas de paramètres particuliers à spécifier pour l'instanciation.

```
#naive bayes
from h2o.estimators import H2ONaiveBayesEstimator

#instanciation
nb = H2ONaiveBayesEstimator(seed=100)

#apprentissage
nb.train(x=pimaTrain.columns[:-1],y="diabete",training_frame=pimaTrain)

#affichage
nb.show()
```

```
Model Details
=============
H2ONaiveBayesEstimator :  Naive Bayes
Model Key:  NaiveBayes_model_python_1546242989411_845


ModelMetricsBinomial: naivebayes
** Reported on train data. **

MSE: 0.1677877334900519
RMSE: 0.40961901016682795
LogLoss: 0.5865637648213471
Mean Per-Class Error: 0.24149291219377922
AUC: 0.8309936691439581
pr_auc: 0.6469071186980442
Gini: 0.6619873382879162

Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.2807268051582214:
          negative    positive    Error    Rate
--------  ----------  ----------  -------  -------------
negative  265         71          0.2113   (71.0/336.0)
positive  47          126         0.2717   (47.0/173.0)
Total     312         197         0.2318   (118.0/509.0)

Maximum Metrics: Maximum metrics at their respective thresholds
```

```
metric                          threshold    value     idx
------------------------------  -----------  --------  -----
max f1                          0.280727     0.681081  177
max f2                          0.050077     0.807365  318
max f0point5                    0.507319     0.676741  128
max accuracy                    0.507319     0.776031  128
max precision                   0.999746     0.846154  1
max recall                      0.0152082    1         370
max specificity                 0.99998      0.994048  0
max absolute_mcc                0.319103     0.506304  167
max min_per_class_accuracy      0.227937     0.745665  194
max mean_per_class_accuracy     0.280727     0.758507  177
Gains/Lift Table: Avg response rate: 33,99 %, avg score: 32,60 %
```

group	cumulative_data_fraction	lower_threshold	lift	cumulative_lift	response_rate	score	cumulative_response_rate	cumulative_score	capture_rate	cumulative_capture_rate	gain	cumulative_gain
1	0.0117878	0.999983	2.45183	2.45183	0.833333	0.999996	0.833333	0.999996	0.0289017	0.0289017	145.183	145.183
2	0.021611	0.999788	2.35376	2.40725	0.8	0.999896	0.818182	0.999951	0.0231214	0.0520231	135.376	140.725
3	0.0314342	0.998874	2.35376	2.39053	0.8	0.99938	0.8125	0.999772	0.0231214	0.0751445	135.376	139.053
4	0.0412574	0.997778	1.76532	2.24167	0.6	0.998073	0.761905	0.999368	0.017341	0.0924855	76.5318	124.167
5	0.0510806	0.994071	1.76532	2.15007	0.6	0.996478	0.730769	0.998812	0.017341	0.109827	76.5318	115.007
6	0.100196	0.914514	2.35376	2.24991	0.8	0.964193	0.764706	0.981842	0.115607	0.225434	135.376	124.991
7	0.151277	0.831458	2.37639	2.29262	0.807692	0.879365	0.779221	0.947239	0.121387	0.346821	137.639	129.262
8	0.200393	0.729915	1.88301	2.19222	0.64	0.778218	0.745098	0.905813	0.0924855	0.439306	88.3006	119.222
9	0.300589	0.45651	1.67301	2.01915	0.568627	0.6062	0.686275	0.805942	0.16763	0.606936	67.3014	101.915
10	0.400786	0.259268	1.26918	1.83166	0.431373	0.351128	0.622549	0.692238	0.127168	0.734104	26.9183	83.1662
11	0.500982	0.170935	0.865352	1.6384	0.294118	0.209169	0.556863	0.595624	0.0867052	0.820809	-13.4648	63.84
12	0.599214	0.104874	0.764971	1.49521	0.26	0.134645	0.508197	0.520054	0.0751445	0.895954	-23.5029	49.5215
13	0.699411	0.058481	0.692282	1.38019	0.235294	0.0767439	0.469101	0.456546	0.0693642	0.965318	-30.7718	38.0188
14	0.799607	0.0276404	0.288451	1.24339	0.0980392	0.0408129	0.422604	0.404452	0.0289017	0.99422	-71.1549	24.3385
15	0.899804	0.0114572	0.0576901	1.11135	0.0196078	0.0197003	0.377729	0.361608	0.00578035	1	-94.231	11.1354
16	1	1.28955e-05	0	1	0	0.00575432	0.339882	0.325953	0	1	-100	0

L'objet est peu disert quand même. Nous n'avons pas d'information sur l'importance des variables, encore moins sur les coefficients estimés. Or, contrairement à ce que l'on peut croire, avec l'estimation paramétrique des distributions conditionnelles (hypothèse gaussienne), il est possible de produire un modèle explicite (Bayésien naïf pour prédicteurs continus, sections 2.2 et 2.3).

Prédiction et évaluation en test. Voyons ce que donne le modèle sur l'échantillon test.

```
#prediction - de nouveau voir le seuil d'affectation
predNb = nb.predict(pimaTest).as_data_frame()
print(predNb.head(10))
```

```
    predict  negative  positive
0  positive  0.353805  0.646195
1  positive  0.000014  0.999986
2  negative  0.983700  0.016300
3  negative  0.999268  0.000732
4  negative  0.884007  0.115993
5  positive  0.024626  0.975374
6  positive  0.269759  0.730241
7  positive  0.040998  0.959002
8  negative  0.799211  0.200789
9  positive  0.646737  0.353263
```

```
#F1-score
print(metrics.f1_score(pimaTest.as_data_frame()["diabete"],predNb.predict,pos_label="positive"))
```

```
0.6497461928934011
```

Malgré sa rusticité (indépendance conditionnelle, distribution gaussienne, autant d'hypothèses assez réductrices a priori), Naive Bayes tient la route par rapport aux autres approches étudiées jusqu'à présent.

3.6 Perceptron

Nous étudions le perceptron en deux temps dans cette section, avec la même classe de calcul H2ODeepLearningEstimator mais avec un paramétrage différent pour définir un perceptron simple tout d'abord (section 3.6.1), puis un perceptron multicouche avec une couche cachée par la suite (section 3.6.2). Dans les deux cas, les variables sont standardisées (standardize=True) ; nous passons 1250 fois sur la base (epochs = 1250) ; la variable cible étant binaire, nous choisissons (distribution = bernoulli) ; (export_weights_and_biases = True) permet de produire les coefficients estimés.

3.6.1 Perceptron simple

Le paramètre `hidden = []` défini un perceptron sans couche cachée.

Initialisation et apprentissage. Après instanciation, nous lançons l'apprentissage. Pas de remarques particulières à ce stade.

```
#deep learning
from h2o.estimators import H2ODeepLearningEstimator

#instanciation
ps = H2ODeepLearningEstimator(seed=100,epochs=1250,standardize=True,hidden=[],distribution="bernoulli")

#apprentissage
ps.train(x=pimaTrain.columns[:-1],y="diabete",training_frame=pimaTrain)
```

Structure du réseau. La méthode summary() permet d'afficher l'architecture du réseau.

```
#structure du réseau
ps.summary()
```

```
Status of Neuron Layers: predicting diabete, 2-class classification, bernoulli distribution,
CrossEntropy loss, 18 weights/biases, 3,0 KB, 636Â 250 training samples, mini-batch size 1

  layer  units  type    dropout  l1  l2  mean_rate           rate_rms            momentum  mean_weight          weight_rms          mean_bias           bias_rms
--- ------- ------- ------- --------- ---- ---- ------------------- ------------------- --------- -------------------- ------------------- ------------------- -------------------
  1      8  Input   0.0
  2      2  Softmax         0.0  0.0  0.0010115477998624556 0.0003913163673132658 0.0       -0.38231324683874846 1.1285338401794434 0.0304955363126663  0.4281446933746338
```

Nous observons 8 neurones dans la couche d'entrée (layer = 1) parce que nous avons 8 variables explicatives ; 2 neurones dans la couche de sortie (layer = 2), parce que la variable cible est binaire. Au total, 18 coefficients ont été estimés en prenant en compte du biais : (8 + 1) x 2 = 18.

Poids synaptiques. Où sont ces fameux coefficients estimés justement ? L'option (export_weights_and_biases = True) lors de l'initialisation de l'objet était importante pour pouvoir en disposer avec les commandes weights() and biaises().

```
#poids synaptiques - coefficients
ps.weights().as_data_frame()
```

```
   pregnant  diastolic   triceps    ...          age    plasma     serum
0  1.224154  -1.544131 -0.785932    ...    -0.685517 -2.809473  0.211043
1  1.621310  -1.803169 -0.730538    ...    -0.558874 -1.582023  0.110001

[2 rows x 8 columns]
```

```
#poids synatpiques- biais
ps.biases().as_data_frame()
```

```
         C1
0  0.322395
1 -0.505916
```

Importance des variables (prim). Puisqu'un perceptron simple est un classifieur linéaire, nous pouvons mesurer l'influence des variables dans le modèle à partir des valeurs absolues des coefficients. Nous devons pour cela dans un premier temps déduire une fonction de classement unique en opposant terme à terme les poids synaptiques des deux sorties du réseau. En effet, dans un problème binaire, nous aurions pu utiliser un réseau avec une seule sortie, où la variable cible serait codée 0/1.

```
#récupération des poids
score = ps.weights().as_data_frame().values

#différences termes à termes - coefficients de la fonction score
fcnScore = score[0,:] - score[1,:]
print(fcnScore)
```

```
[-0.397156    0.25903749 -0.05539423 -0.78685617 -0.44376697 -0.1266433
 -1.22744966  0.10104156]
```

Il ne reste plus qu'à trier les variables selon la valeur absolue de ces coefficients.

```
#index selon la valeur absolue du coefs
index = numpy.argsort(numpy.abs(fcnScore))

#inversion
index = index[::-1]

#affichage
temp = {'variable':ps.weights().as_data_frame().columns[index],'influence':fcnScore[index]}
print(pandas.DataFrame(temp))
```

```
    variable  influence
0     plasma  -1.227450
```

```
1   bodymass  -0.786856
2   pedigree  -0.443767
3   pregnant  -0.397156
4  diastolic   0.259037
5        age  -0.126643
6      serum   0.101042
7    triceps  -0.055394
```

Ces valeurs ne sont pas sans rappeler les coefficients standardisés de la régression logistique (page 12). Ce n'est absolument pas étonnant. Perceptron simple et régression logistique s'appuient sur le même système de représentation, une combinaison linéaire des variables explicatives. Seuls les algorithmes d'optimisation de la fonction de perte utilisés différent (et les valeurs par défaut des innombrables paramètres des méthodes, impossible à contrôler complètement). On note encore une fois que PLASMA et BODYMASS se démarquent par rapport aux autres variables.

Importance des variables (bis). La sortie standard des modèles show() propose des sorties additionnelles qui ont attiré mon attention.

```
#affichage
ps.show()
```

```
Model Details
=============
H2ODeepLearningEstimator :  Deep Learning
Model Key:  DeepLearning_model_python_1546242989411_857


ModelMetricsBinomial: deeplearning
** Reported on train data. **

MSE: 0.14823787798141633
RMSE: 0.3850167242879409
LogLoss: 0.45140669181837106
Mean Per-Class Error: 0.23588459950454177
AUC: 0.8506313652628682
pr_auc: 0.7267311610099244
Gini: 0.7012627305257364

Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.35670682573600015:
          negative    positive    Error    Rate
--------  ----------  ----------  -------  -------------
negative  261         75          0.2232   (75.0/336.0)
positive  43          130         0.2486   (43.0/173.0)
Total     304         205         0.2318   (118.0/509.0)

Maximum Metrics: Maximum metrics at their respective thresholds

metric                       threshold    value     idx
---------------------------  -----------  --------  -----
max f1                       0.356707     0.687831  181
max f2                       0.199578     0.811623  255
max f0point5                 0.649068     0.712     98
max accuracy                 0.570424     0.791749  116
```

```
max precision                  0.998523     1          0
max recall                     0.0945328    1          320
max specificity                0.998523     1          0
max absolute_mcc               0.417104     0.518975   160
max min_per_class_accuracy     0.340778     0.761905   186
max mean_per_class_accuracy    0.356707     0.764115   181
Gains/Lift Table: Avg response rate: 33,99 %, avg score: 35,81 %
```

group	cumulative_data_fraction	lower_threshold	lift	cumulative_lift	response_rate	score	cumulative_response_rate	cumulative_score	capture_rate	cumulative_capture_rate	gain	cumulative_gain
1	0.0117878	0.972058	2.45183	2.45183	0.833333	0.980969	0.833333	0.980969	0.0289017	0.0289017	145.183	145.183
2	0.021611	0.951928	2.35376	2.40725	0.8	0.960065	0.818182	0.971467	0.0231214	0.0520231	135.376	140.725
3	0.0314342	0.935568	2.9422	2.57442	1	0.94201	0.875	0.962262	0.0289017	0.0809249	194.22	157.442
4	0.0412574	0.910796	2.9422	2.66199	1	0.925141	0.904762	0.953423	0.0289017	0.109827	194.22	166.199
5	0.0510806	0.89398	2.9422	2.71587	1	0.900299	0.923077	0.943207	0.0289017	0.138728	194.22	171.587
6	0.100196	0.824041	2.23607	2.48068	0.76	0.865195	0.843137	0.904966	0.109827	0.248555	123.607	148.068
7	0.151277	0.754328	2.03691	2.33083	0.692308	0.787745	0.792208	0.865385	0.104046	0.352601	103.691	133.083
8	0.200393	0.681399	2.23607	2.30761	0.76	0.715583	0.784314	0.828669	0.109827	0.462428	123.607	130.761
9	0.300589	0.498231	1.49994	2.03838	0.509804	0.590083	0.69281	0.74914	0.150289	0.612717	49.9943	103.838
10	0.400786	0.358482	1.32687	1.86051	0.45098	0.423434	0.632353	0.667714	0.132948	0.745665	32.6873	86.0507
11	0.500982	0.271645	0.807662	1.64994	0.27451	0.306941	0.560784	0.595559	0.0809249	0.82659	-19.2338	64.9938
12	0.599214	0.199812	1.05919	1.55309	0.36	0.237435	0.527869	0.53685	0.104046	0.930636	5.91908	55.3094
13	0.699411	0.129233	0.403831	1.38845	0.137255	0.165007	0.47191	0.48358	0.0404624	0.971098	-59.6169	38.8452
14	0.799607	0.0872242	0.288451	1.25061	0.0980392	0.104162	0.425061	0.436037	0.0289017	1	-71.1549	25.0614
15	0.899804	0.0476657	0	1.11135	0	0.0675077	0.377729	0.395	0	1	-100	11.1354
16	1	0.00104718	0	1	0	0.0265055	0.339882	0.358078	0	1	-100	0

Scoring History:

timestamp	duration	training_speed	epochs	iterations	samples	training_rmse	training_logloss	training_r2	training_auc	training_pr_auc	training_lift	training_classification_error
2018-12-31 15:36:46	0.000 sec		0	0	0	nan	nan	nan	nan	nan	nan	nan
2018-12-31 15:36:46	0.010 sec	462727 obs/sec	10	1	5090	0.523841	1.0333	-0.223065	0.650083	0.533649	2.9422	0.349705
2018-12-31 15:36:46	0.290 sec	2248233 obs/sec	1250	125	636250	0.385017	0.451407	0.339292	0.850631	0.726731	2.45183	0.231827

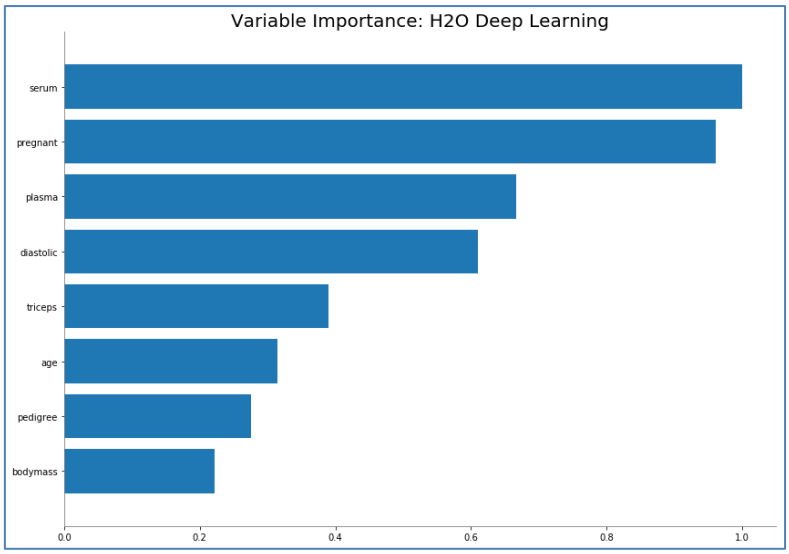
Variable Importances:

variable	relative_importance	scaled_importance	percentage
plasma	1	1	0.265971
diastolic	0.766143	0.766143	0.203772
pregnant	0.711991	0.711991	0.189369
triceps	0.38459	0.38459	0.10229
bodymass	0.368926	0.368926	0.0981238
age	0.267089	0.267089	0.0710379
pedigree	0.188259	0.188259	0.0500715
serum	0.0728074	0.0728074	0.0193647

L'outil présente une mesure de l'impact des variables et, patatras, nous n'avons pas le même classement qu'avec l'analyse des coefficients. Par exemple, ''diastolic'' devient bien placé, a contrario, ''bodymass'' est reléguée en 5ème position…

Nous pouvons de nouveau produire un affichage graphique, sans le sens (signe) des influences :

```
#importance des variables - graphique
ps.varimp_plot()
```



Nous trouvons la description suivante sur le site de H2O : « Variable importances for Neural Network models are notoriously difficult to compute, and there are many pitfalls. H2O Deep Learning has implemented the method of Gedeon, and returns relative variable importances in descending order of importance. ». L'article cité (T. Gédéon, ''Data Mining of Inputs : analysing magnitude and functional measures'', Int. J. of Neural Syst., 8(2):209-208, 1997), décrit le calcul d'une mesure (functional measure) de l'influence des entrées via les caractéristiques du réseau. L'idée est de pouvoir écarter les descripteurs peu pertinents, c'est la moindre des choses, mais surtout de mettre en lumière les descripteurs qui agissent, même faiblement sur la prédiction, mais en étant peu corrélée aux autres, au détriment de ceux qui sont redondants. Pourquoi pas. Il n'en reste pas moins que les résultats proposés ne sont pas complètement (le rôle primordial de PLASMA a bien été détecté) cohérent avec l'analyse des coefficients qui fait référence dans un modèle linéaire. L'énorme avantage de cette approche en revanche est de pouvoir s'appliquer pour un réseau multicouche, nous verrons cela dans la section suivante (section 3.6.2).

Performances en test. Voyons ce que donne le perceptron simple sur l'échantillon test.

```
#prediction - de nouveau voir le seuil d'affectation
predPs = ps.predict(pimaTest).as_data_frame()
print(predPs.head(10))
```

```
    predict  negative  positive
0  positive  0.210280  0.789720
1  positive  0.040390  0.959610
2  negative  0.931060  0.068940
3  negative  0.981996  0.018004
4  negative  0.809858  0.190142
5  positive  0.227948  0.772052
6  positive  0.445003  0.554997
7  positive  0.050892  0.949108
8  negative  0.672606  0.327394
9  positive  0.547592  0.452408
```

```
#F1-score
print(metrics.f1_score(pimaTest.as_data_frame()["diabete"],predPs.predict,pos_label="positive"))
```

```
0.681081081081081
```

3.6.2 Perceptron multicouche

Le paramètre hidden joue un rôle clé pour la définition d'un perceptron multicouche. Dans la liste, nous avons autant de valeurs qu'il y a de couches cachées, et elles (les valeurs) représentent le nombre de neurones. Dans notre cas, une couche cachée à deux neurones, nous indiquons (hidden = [2]). Nous devons aussi indiquer la fonction d'activation à utiliser dans les couches cachées, nous choisissons la tangente hyperbolique (activation = ''Tanh''). Pour le reste, nous adoptons les mêmes spécifications que pour le perceptron simple.

Initialisation et apprentissage. Comme pour les autres méthodes, nous initialisons l'algorithme puis nous lançons l'apprentissage.

```
#instanciation
pmc   =   H2ODeepLearningEstimator(seed=100,epochs=1250,standardize=True,hidden=[2],
activation="Tanh",distribution="bernoulli", export_weights_and_biases = True)

#apprentissage
pmc.train(x=pimaTrain.columns[:-1],y="diabete",training_frame=pimaTrain)

#structure du réseau
pmc.summary()
```

La structure du réseau correspond bien à nos spécifications : 8 neurones en entrée, 2 neurones dans l'unique couche cachée, 2 neurones dans la couche de sortie.

```
Status of Neuron Layers: predicting diabete, 2-class classification, bernoulli distribution, CrossEntropy loss, 24 weights/biases, 3,7 KB, 636Â 250 training samples, mini-batch size 1

  layer  units  type     dropout  l1   l2   mean_rate            rate_rms             momentum  mean_weight          weight_rms           mean_bias            bias_rms
--- ------- ------- -------- ---- ---- ---- -------------------- -------------------- --------- -------------------- -------------------- -------------------- --------------------
    1      8 Input    0.0
    2      2 Tanh     0.0      0.0  0.0  0.0008600479477536283 0.0005428462754935026 0.0       -0.07937054010108113 0.6695523262023926  0.19239773511699743  0.039906978607177734
    3      2 Softmax           0.0  0.0  0.0031578594935126603 3.0307630368042737e-05 0.0       -0.6641974356025457  1.406151294708252   -0.12795481419197946 0.8296313285827637
```

Nous avons une structure à deux niveaux maintenant pour les poids synaptiques. Entre la couche d'entrée et la couche cachée…

```
#poids - entrée -> cachée
pmc.weights(0).as_data_frame()
```

```
   pregnant  diastolic   triceps    ...          age    plasma     serum
0  0.780549   0.701436  0.347671    ...    -1.908585 -0.117882  0.197698
1  0.232197   0.050922  0.076213    ...    -0.264321  0.449429  0.016090
```

… et entre la couche cachée et la sortie.

```
#poids - cachée -> sortie
pmc.weights(1).as_data_frame()
```

```
         C1        C2
0 -0.065575 -2.357026
1 -1.469590  1.235401
```

L'affichage inclut de nouveau l'importance des variables.

```
#affichage
pmc.show()
```

```
Model Details
=============
H2ODeepLearningEstimator :  Deep Learning
Model Key:  DeepLearning_model_python_1546420368286_5


ModelMetricsBinomial: deeplearning
** Reported on train data. **

MSE: 0.14109382046809732
RMSE: 0.375624573834164
LogLoss: 0.43132812699612166
Mean Per-Class Error: 0.21256537296999722
AUC: 0.8649704101293697
pr_auc: 0.7638876695853001
Gini: 0.7299408202587394

Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.25353058727013555:
          negative    positive    Error    Rate
--------  ----------  ----------  -------  -------------
negative  253         83          0.247    (83.0/336.0)
positive  31          142         0.1792   (31.0/173.0)
Total     284         225         0.224    (114.0/509.0)

Maximum Metrics: Maximum metrics at their respective thresholds
metric                        threshold    value     idx
----------------------------  -----------  --------  -----
max f1                        0.253531     0.713568  200
max f2                        0.193144     0.807128  231
max f0point5                  0.563769     0.729323  106
max accuracy                  0.445304     0.803536  137
max precision                 0.951798     1         0
max recall                    0.0436701    1         343
max specificity               0.951798     1         0
max absolute_mcc              0.445304     0.553549  137
max min_per_class_accuracy    0.28871      0.776786  186
max mean_per_class_accuracy   0.198426     0.787435  226

Gains/Lift Table: Avg response rate: 33,99 %, avg score: 31,83 %

    group    cumulative_data_fraction    lower_threshold    lift    cumulative_lift    response_rate    score    cumulative_response_rate    cumulative_score    capture_rate    cumulative_capture_rate    gain    cumulative_gain
--  -------  --------------------------  -----------------  ------  -----------------  ---------------  -------  --------------------------  ------------------  --------------  -------------------------  ------  -----------------
 1  0.0117878                 0.934653   2.9422  2.9422             1         0.941733  1                 0.941733            0.0346821       0.0346821                194.22  194.22
 2  0.021611                  0.923141   2.9422  2.9422             1         0.926388  1                 0.934758            0.0289017       0.0635838                194.22  194.22
 3  0.0314342                 0.905096   2.35376  2.75831           0.8       0.91373   0.9375            0.928187            0.0231214       0.0867052                135.376  175.831
 4  0.0412574                 0.900285   2.35376  2.66199           0.8       0.902776  0.904762          0.922137            0.0231214       0.109827                 135.376  166.199
 5  0.0510806                 0.886337   2.9422  2.71587            1         0.890434  0.923077          0.91604             0.0289017       0.138728                 194.22  171.587
 6  0.100196                  0.805161   2.58913  2.65375           0.88      0.843539  0.901961          0.8805              0.127168        0.265896                 158.913  165.375
```

```
  7    0.151277          0.720334      2.15007  2.48367      0.730769    0.771841  0.844156          0.84381      0.109827    0.375723          115.007  148.367
  8    0.200393          0.617319      2.00069  2.3653       0.68        0.661774  0.803922          0.799193     0.0982659   0.473988          100.069  136.53
  9    0.300589          0.471378      1.67301  2.13453      0.568627    0.547639  0.72549           0.715342     0.16763     0.641618          67.3014  113.453
  10   0.400786          0.302726      1.1538   1.88935      0.392157    0.373303  0.642157          0.629832     0.115607    0.757225          15.3803  88.9352
  11   0.500982          0.202012      1.1538   1.74224      0.392157    0.244424  0.592157          0.552751     0.115607    0.872832          15.3803  74.2242
  12   0.599214          0.144092      0.529595 1.54345      0.18        0.170525  0.52459           0.490091     0.0520231   0.924855          -47.0405 54.3447
  13   0.699411          0.0946819     0.461521 1.38845      0.156863    0.119345  0.47191           0.436978     0.0462428   0.971098          -53.8479 38.8452
  14   0.799607          0.0547137     0.11538  1.22893      0.0392157   0.0731048 0.41769           0.391382     0.0115607   0.982659          -88.462  22.8927
  15   0.899804          0.0255627     0.17307  1.11135      0.0588235   0.0371009 0.377729          0.351932     0.017341    1                 -82.693  11.1354
  16   1                 0.00176812    0        1           0           0.0158513 0.339882          0.318258     0           1                 -100     0

Scoring History:
    timestamp          duration   training_speed   epochs   iterations   samples   training_rmse   training_logloss   training_r2   training_auc   training_pr_auc   training_lift   training_classification_error
--  -------------------  ----------  ---------------  --------  ------------  ---------  ----------------  ------------------  ------------  -------------  ----------------  --------------  -------------------------------
    2019-01-02 10:51:04  0.000 sec                 0         0             0        nan               nan                 nan           nan            nan               nan             nan
    2019-01-02 10:51:04  0.049 sec  169666 obs/sec    10         1          5090   0.400604          0.488604            0.284714      0.831278       0.669643          1.96146         0.231827
    2019-01-02 10:51:05  0.489 sec  1374190 obs/sec   1250       125        636250   0.375625          0.431328            0.371134      0.86497        0.763888          2.9422          0.223969

Variable Importances:
variable    relative_importance    scaled_importance    percentage
----------  ---------------------  -------------------  ------------
plasma      1                      1                    0.26422
age         0.922294               0.922294             0.243689
pregnant    0.647399               0.647399             0.171056
bodymass    0.400116               0.400116             0.105719
pedigree    0.276527               0.276527             0.073064
diastolic   0.238319               0.238319             0.0629688
triceps     0.229113               0.229113             0.0605363
serum       0.0709525              0.0709525            0.0187471
```

Par rapport au modèle linéaire (perceptron simple), nous constatons que : (1) PLASMA est toujours en première position ; (2) les positions sont modifiées pour certaines variables, AGE notamment arrive en seconde position maintenant, quasiment au même niveau que la première, c'est étonnant, aucun des modèles précédents n'avait suggéré ce positionnement, à tort ou à raison, la question reste posée.

Représentation intermédiaire. H2O produit les coordonnées des individus dans les couches intermédiaires avec la commande deepFeatures(). Comme nous avons 2 neurones dans la couche cachée, nous pouvons projeter les individus dans une espace de représentation 2D (une sorte de plan factoriel supervisé). Inspectons les valeurs pour les 10 premiers individus de l'échantillon d'apprentissage.

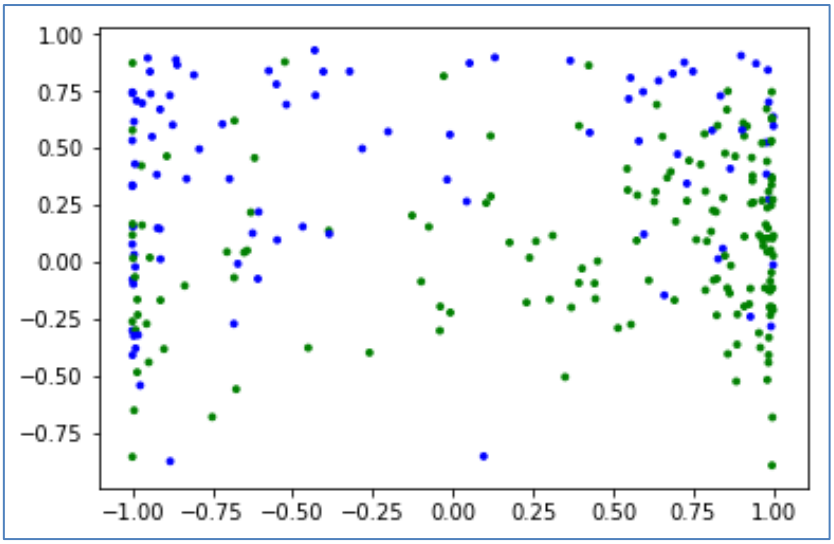
```
#PMC -- coordonnées dans la couche cachée
coordPmc = pmc.deepfeatures(pimaTest,0).as_data_frame()
print(coordPmc.head(10))
```

```
   DF.L1.C1  DF.L1.C2
0  0.685449  0.827108
1 -1.000000  0.076823
2  0.927492 -0.241484
3  0.991998 -0.284199
4  0.810772  0.225380
5 -0.999800  0.117698
6  0.653203  0.549985
7 -0.863971  0.887957
8  0.998619  0.637089
9  0.543016  0.408412
```

Les valeurs varient entre -1 et +1 puisque la fonction tangente hyperbolique (Tanh) fait office de fonction d'activation.

Nous récupérons les colonnes de valeurs pour projeter les individus dans le plan en les coloriant selon leur classe d'appartenance.

```
#récupération des colonnes
C1 = coordPmc.iloc[:,0]
C2 = coordPmc.iloc[:,1]

#code couleur
couleur = (pimaTest.as_data_frame().loc[:,"diabete"] == "positive").astype('int')

#librairie graphique
import matplotlib.pyplot as plt

#affichage des points dans le plan
plt.scatter(C1,C2,c=pandas.Series(['green','blue'])[couleur],s=8)
```



Il est difficile de trouver une séparation linéaire qui permettrait d'isoler parfaitement les points de couleurs différentes, d'où un taux d'erreur élevé. Pour remédier à cela, nous pourrions augmenter le nombre de neurones ou rajouter des couches, sans garantie de succès d'ailleurs. Si la discrimination est intrinsèquement difficile, en particulier parce qu'il manque des variables explicatives plus pertinentes, s'exciter à surdimensionner les modèles ne peut qu'aboutir au surapprentissage.

Performances en test. Voyons justement ce qu'il en est des performances en test.

```
#prediction - de nouveau voir le seuil d'affectation
predPmc = pmc.predict(pimaTest).as_data_frame()
print(predPmc.head(10))
```

```
    predict  negative  positive
0  positive  0.413450  0.586550
```

```
1  positive  0.494807  0.505193
2  negative  0.978730  0.021270
3  negative  0.983258  0.016742
4  negative  0.879521  0.120479
5  positive  0.458264  0.541736
6  positive  0.645782  0.354218
7  positive  0.060442  0.939558
8  positive  0.684084  0.315916
9  positive  0.722006  0.277994
```

```
#F1-score
print(metrics.f1_score(pimaTest.as_data_frame()["diabete"],predPmc.predict,pos_label="positive"))
```

```
0.6948356807511737
```

Le meilleur F1-Score que l'on ait obtenu pour l'ensemble des modèles testés jusqu'à présent.

3.7 Bilan

Les résultats obtenus dans cette section n'ont pas valeur de preuve, loin s'en faut. Il s'agit d'une expérimentation sur une base (PIMA), avec une version de la subdivision apprentissage-test des données, où ce dernier (échantillon test) est de taille assez réduite quand même (259 observations). Rappelons aussi que j'ai sciemment pénalisé le gradient boosting pour mieux mettre en valeur les outils additionnels de H20 dans la section suivante.

Bon, il reste qu'un petit tableau récapitulatif permet d'avoir une vue globale de notre étude.

ML Method - Pima Dataset	F1-Score
GLM (Logistic Regression)	0.667
Random Forest	0.670
Gradient Boosting	0.333
Naive Bayes	0.650
Simple Perceptron	0.681
Multilayer Perceptron	0.695

Le perceptron multicouche est le plus performant semble-t-il. Mais les écarts entre les modèles ne sont pas mirifiques non plus. Estimés sur un échantillon test de taille réduite, ils ne sont certainement pas significatifs.

4 Plus loin avec l'apprentissage supervisé

La profusion des paramètres est le véritable goulot d'étranglement de l'utilisation des librairies de machine learning modernes. Ils sont souvent en nombre très importants pour les différentes méthodes. C'est un avantage si on les comprend et qu'on sait les manipuler. C'est une plaie si l'on s'y noie. J'ai moi-même du faire évoluer mes enseignements et leur accorder plus de place lorsque je présente les techniques de data mining aux étudiants. Les paramètres sont souvent interdépendants. Sans aller dans trop de détails, il faut au moins tracer les grandes lignes pour savoir comment les orienter en fonction des caractéristiques des problèmes et des données que nous traitons. Ou bien, et

c'est que nous essaierons de mettre en avant dans cette section, utiliser des outils additionnels qui nous permettent d'identifier « automatiquement » les bonnes valeurs et, ainsi, nous affranchir des manipulations périlleuses.

4.1 Utilisation d'un échantillon de validation

Il est d'usage en analyse prédictive de scinder les données en 2 parties : la première, dite échantillon d'apprentissage sert à construire le modèle ; la seconde, dite échantillon test, est un arbitre impartial qui permet d'en évaluer les performances. Nous avons adopté ce schéma lors de l'étude des différents algorithmes de H2O (section 3). Nous en étions arrivés à la conclusion que le perceptron multicouche était le plus performant, avec la réserve liée à la faible taille de l'ensemble de test utilisé.

Dans cette section, nous explorons l'utilisation d'un troisième échantillon de données dit « de validation ». Nous sommes dans un schéma à 3 ensembles de données : apprentissage, validation, test. Le rôle de cet échantillon supplémentaire est assez particulier, il participe certes au processus de modélisation, mais intervient différemment selon l'algorithme étudié. (1) Il peut servir par exemple d'échantillon permettant de réduire la taille de l'arbre dans une phase de post-élagage, on parle de « pruning » set ». (2) Il peut également intervenir pour mesurer les performances des modèles selon les valeurs attribuées aux différents paramètres, il sert alors à sélectionner la combinaison la plus performante. On le qualifie dans ce cas de « tuning set ». (3) Il peut également servir à suivre l'évolution de la fonction de perte au fil de l'apprentissage pour les algorithmes itératifs, et permettre ainsi de stopper le processus lorsque l'on constate une stagnation au bout d'un certain nombre de passages sur les données. On parle alors de « validation set ». En effet, la fonction de perte présente la fâcheuse tendance d'évoluer positivement constamment sur l'échantillon de travail, laissant à croire qu'on améliore continuellement le modèle au fil des itérations, alors qu'en réalité nous sommes simplement en train d'ingérer les singularités des données d'entraînement. L'utilisation d'un ensemble de données à part, dont le seul rôle est de surveiller les performances, permet de relativiser cette fausse impression et surtout de stopper suffisamment tôt l'apprentissage, économisant les ressources de calcul mais aussi nous préservant du surapprentissage.

Dans cette section, nous allons exploiter cette idée d'échantillon de validation pour mieux guider la construction de l'ensemble d'arbres du Gradient Boosting, qui s'était révélée si catastrophique précédemment (section 3.4). Nous suspections une combinaison funeste lors de notre modélisation : trop d'arbres, trop complexes. Pour le premier point, nous avions constaté que la fonction de perte stagnait à partir d'un certain nombre d'arbres (Figure 2). Essayons voir si l'échantillon de validation nous permettra de déterminer efficacement le point d'arrêt de l'empilement des arbres.

Création de l'échantillon de validation. L'échantillon test (pimaTest) étant toujours à part pour l'évaluation des performances, la création de l'échantillon de validation doit venir de la partition de l'ensemble pimaTrain. Nous utilisons de nouveau la fonction split_frame() pour produire pimaLearning (352 observations) et pimaValidation (157 observations).

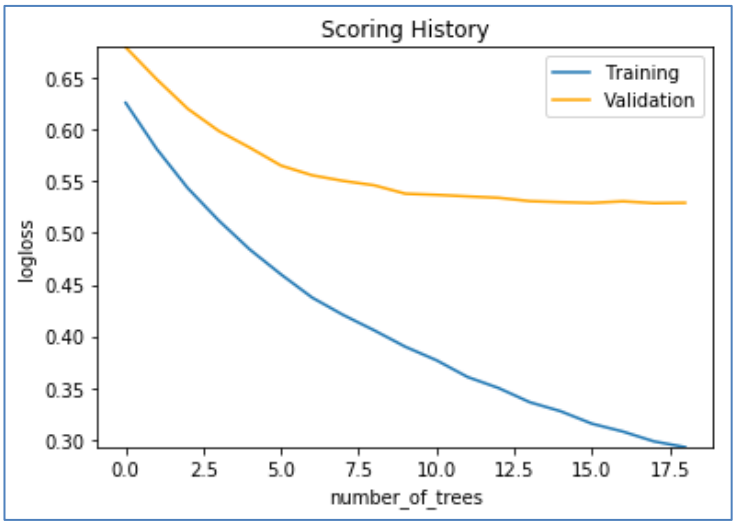
```
#subdivision
pimaLearning, pimaValidation = pimaTrain.split_frame(ratios=[0.70],seed=1)

#dimensions - learning set
pimaLearning.shape #(352, 9)

#dimensions - validation set
pimaValidation.shape      #(157, 9)
```

Apprentissage avec échantillon de validation. Pour faire intervenir l'échantillon de validation lors de la construction du modèle, nous ajoutons les paramètres : (stopping_metric = ''log_loss''), l'évolution de logarithme de la perte est utilisée pour monitorer l'évolution de l'apprentissage ; (stopping_rounds = 3), si après 3 passages sur les données, la perte n'évolue pas dans le bon sens, le processus est stoppé ; (stopping_tolerance = 1e-3) représente l'écart à considérer pour conclure à l'absence d'évolution de l'indicateur. Ensuite, nous indiquons l'échantillon de validation lors de l'appel de train().

```
#instanciation
gbBis=H2OGradientBoostingEstimator(seed=100,ntrees=1500,max_depth=5,
stopping_metric="log_loss",stopping_rounds=3,stopping_tolerance=1e-3)

#apprentissage
gbBis.train(x=pimaTrain.columns[:-1], y="diabete", training_frame=pimaLearning, validation_frame=pimaValidation)
```

L'évolution du processus d'optimisation que l'on peut retracer avec plot() est particulièrement édifiant (à comparer avec la Figure 2).

```
#évolution
gbBis.plot()
```



18 arbres suffisent en réalité. Tous sont de profondeur 5. C'est bien le paramètre nombre d'arbres qui est en jeu. L'échantillon de validation permet de stopper tôt le processus et évite les calculs supplémentaires superflus.

```
#résumé
gbBis.summary()
```

```
Model Summary:
  number_of_trees  number_of_internal_trees  model_size_in_bytes  min_depth  max_depth  mean_depth  min_leaves  max_leaves  mean_leaves
--  ---------------  -------------------------  -------------------  ---------  ---------  ----------  ----------  ----------  -------------
   18               18                         4644                 5          5          5           10          21          15.8889
```

Performances en test. Mais est-ce que la réduction plus que drastique du nombre d'arbres joue sur les performances en déploiement ? Calculons les prédictions sur l'échantillon test.

```
#prediction - de nouveau voir le seuil d'affectation
predGbBis = gbBis.predict(pimaTest).as_data_frame()
print(predGbBis.head(10))
```

```
    predict  negative  positive
0  positive  0.283294  0.716706
1  positive  0.270484  0.729516
2  negative  0.937999  0.062001
3  negative  0.902753  0.097247
4  positive  0.738218  0.261782
5  negative  0.893637  0.106363
6  positive  0.717226  0.282774
7  positive  0.247857  0.752143
8  positive  0.707718  0.292282
9  positive  0.364520  0.635480
```

```
#F1-score
print(metrics.f1_score(pimaTest.as_data_frame()["diabete"],predGbBis.predict,pos_label="positive"))
```

```
0.6883720930232559
```

Oui, la contribution de l'échantillon de validation est décisive dans notre expérimentation. Elle a permis de réduire le nombre de d'arbres, jouant sur le temps de calcul (18 arbres à construire vs. 1500) et neutralisant le phénomène de surapprentissage (F1-Score = 0.6883 vs. 0.3333).

4.2 Validation croisée

La partition des données en 3 sous-échantillons n'est pas toujours possible sur les petites bases. La validation croisée est une alternative possible. Elle permet d'obtenir une mesure des performances crédible sur l'échantillon d'apprentissage, au prix certes d'un surcroît de calculs. Comme dans la section précédente, elle peut de ce fait servir à monitorer le processus d'apprentissage et stopper à opportunément le processus de modélisation. Dans cette section, nous avons plutôt choisi de montrer sa propriété d'estimation honnête des performances en généralisation en comparant le F1-Score en validation croisée (basé uniquement sur l'échantillon d'apprentissage) et en test (mesuré sur l'échantillon dédié).

Apprentissage et validation croisée. Nous instancions de nouveau un Gradient Boosting en laissant les paramètres par défaut (ntrees = 50, max_depth_tree = 5) sauf (nfolds = 3) pour indiquer une validation croisée en 3 portions. De manière interne, 4 versions du modèle seront instanciées : 1 sur la

totalité des données d'apprentissage, 3 correspondantes aux sessions apprentissage-test sur les 3 configurations des données (2 premières portions contre la dernière, et ainsi de suite en faisant tourner).

```
#instanciation
gbCV = H2OGradientBoostingEstimator(seed=100,nfolds=3)

#apprentissage
gbCV.train(x=pimaTrain.columns[:-1],y="diabete",training_frame=pimaTrain)
```

cross_validation_metrics_summary() affiche le détail des calculs.

```
#résumé des résultats
gbCV.cross_validation_metrics_summary()
```

```
Cross-Validation Metrics Summary:
                        mean      sd          cv_1_valid    cv_2_valid    cv_3_valid
----------------------  --------  ----------  ------------  ------------  ------------
accuracy                0.769191  0.0266593   0.715909      0.794118      0.797546
auc                     0.794159  0.0201015   0.755324      0.822581      0.804574
err                     0.230809  0.0266593   0.284091      0.205882      0.202454
err_count               39.3333   5.36449     50            35            33
f0point5                0.662747  0.0391365   0.587467      0.718954      0.681818
f1                      0.681339  0.0210688   0.642857      0.715447      0.685714
f2                      0.703803  0.00710214  0.709779      0.711974      0.689655
lift_top_group          1.99904   0.568855    1.49153       1.37097       3.13462
logloss                 0.562309  0.0300321   0.622007      0.526728      0.538192
max_per_class_error     0.301902  0.00578991  0.307692      0.290323      0.307692
mcc                     0.507217  0.0384677   0.430968      0.554218      0.536466
mean_per_class_accuracy 0.757741  0.0152335   0.72751       0.776135      0.769577
mean_per_class_error    0.242259  0.0152335   0.27249       0.223865      0.230423
mse                     0.181434  0.011883    0.205161      0.168391      0.17075
precision               0.652037  0.0497459   0.555556      0.721311      0.679245
r2                      0.188873  0.0573542   0.079375      0.273222      0.214023
recall                  0.721566  0.0211753   0.762712      0.709677      0.692308
rmse                    0.425507  0.0137449   0.452947      0.410355      0.413219
specificity             0.793916  0.0508188   0.692308      0.842593      0.846847
```

Nous disposons des résultats pour chaque indicateur. Si l'on s'en tient au F1-Score, les sessions ont fourni les valeurs (0.642857, 0.715447 et 0.684714). L'estimation en validation croisée du F1-Score est la moyenne, soit 0.681339.

Evaluation sur l'échantillon test. Voyons ce qu'il en est sur l'échantillon test.

```
#prediction - de nouveau voir le seuil d'affectation
predGbCV = gbCV.predict(pimaTest).as_data_frame()
print(predGbCV.head(10))

#F1-score
print(metrics.f1_score(pimaTest.as_data_frame()["diabete"],predGbCV.predict,pos_label="positive"))
```

```
0.6590909090909091
```

Le F1-Score en test est de 0.659. Cette valeur représente une autre estimation du F1-Score en généralisation. Nous constatons qu'elle est moins optimiste. Laquelle est la bonne, celle-ci ou celle produite en validation croisée ? Nous sommes bien en mal de le dire. Ma préférence va vers l'échantillon test lorsque sa taille est suffisamment élevée. Dans le cas présent, son faible effectif entache sa crédibilité. Disons que nous disposons de deux estimations du F1-Score du modèle et que la vraie valeur est « dans ces eaux-là » (la science statistique n'est pas toujours très précise…).

4.3 Recherche des paramètres optimaux

La validation croisée, en produisant une estimation « honnête » des performances, peut jouer un rôle important dans le paramétrage des modèles. La démarche, très simpliste, consiste à croiser différentes de valeurs des paramètres et, pour chaque combinaison, mesurer – en validation croisée – un indicateur de performance, tel que le F1-Score par exemple, mais ça pourrait être également le taux d'erreur, l'AUC (aire sous la courbe ROC), etc. Cette recherche en grille (H2OGridSearch), avec l'option par défaut « systématique », a le mérite de l'exhaustivité. Mais elle peut être coûteuse en temps de calcul et, surtout, elle peut aboutir au sur-apprentissage si nous multiplions les associations. Il faut savoir rester raisonnable.

Définition des valeurs de paramètres à croiser. Nous travaillons de nouveau avec le Gradient Boosting. Nous souhaitons croiser différentes valeurs du nombre d'arbres (ntrees) et leur profondeur maximale (max_depth). Nous pouvons maintenant instancier l'outil en lui passant la grille des paramètres à traiter.

```
#définir les paramètres à tester
gbParametres = {'ntrees':[20,50,100,1000] , 'max_depth':[1,2,5,20]}

#classe pour la grille de recherche
from h2o.grid.grid_search import H2OGridSearch

#instancier
gbGrid = H2OGridSearch(model = H2OGradientBoostingEstimator, hyper_params = gbParametres)
```

Evaluation des configurations en validation croisée. Nous lançons les traitements en indiquant le nombre de portions de la validation croisée (nfolds=5). Nous obtenons un tableau de résultats que nous trions selon les valeurs décroissantes du F1-Score.

```
#lancer les calculs -- 5-validation croisée
gbGrid.train(x=pimaTrain.columns[:-1],y="diabete",training_frame=pimaTrain,nfolds=5,seed=100)

#affichage des résultats
gbModels = gbGrid.get_grid(sort_by='f1',decreasing=True)
print(gbModels.summary())
```

Grid Summary:

Model Id	number_of_trees	number_of_internal_trees	model_size_in_bytes	min_depth	max_depth	mean_depth	min_leaves	max_leaves	mean_leaves
Grid_GBM_py_7_sid_962a_model_python_1546462206476_931_model_6	50	50	5426	2	2	2	3	4	3.98
Grid_GBM_py_7_sid_962a_model_python_1546462206476_931_model_2	20	20	2170	2	2	2	4	4	4
Grid_GBM_py_7_sid_962a_model_python_1546462206476_931_model_10	100	100	10753	2	2	2	3	4	3.9
Grid_GBM_py_7_sid_962a_model_python_1546462206476_931_model_9	100	100	8384	1	1	1	2	2	2
Grid_GBM_py_7_sid_962a_model_python_1546462206476_931_model_5	50	50	4188	1	1	1	2	2	2
Grid_GBM_py_7_sid_962a_model_python_1546462206476_931_model_3	20	20	5980	5	5	5	14	24	19.1
Grid_GBM_py_7_sid_962a_model_python_1546462206476_931_model_13	1000	1000	84825	1	1	1	2	2	2
Grid_GBM_py_7_sid_962a_model_python_1546462206476_931_model_4	20	20	10564	8	14	9.35	30	42	37.2
Grid_GBM_py_7_sid_962a_model_python_1546462206476_931_model_1	20	20	1670	1	1	1	2	2	2
Grid_GBM_py_7_sid_962a_model_python_1546462206476_931_model_7	50	50	12865	5	5	5	9	24	15.8
Grid_GBM_py_7_sid_962a_model_python_1546462206476_931_model_8	50	50	27459	8	19	11.96	30	44	38.7
Grid_GBM_py_7_sid_962a_model_python_1546462206476_931_model_12	100	100	55939	8	20	13.2	30	44	39.39
Grid_GBM_py_7_sid_962a_model_python_1546462206476_931_model_11	100	100	24195	5	5	5	8	24	14.57
Grid_GBM_py_7_sid_962a_model_python_1546462206476_931_model_16	1000	1000	560115	8	20	17.507	26	44	38.86
Grid_GBM_py_7_sid_962a_model_python_1546462206476_931_model_15	1000	1000	239204	5	5	5	6	25	14.295
Grid_GBM_py_7_sid_962a_model_python_1546462206476_931_model_14	1000	1000	107629	2					

Nous pouvons afficher les performances en validation croisée des modèles.

```
#affichage des performances
gbModels.show()
```

```
                      f1
0             ...          0.7131782945736435
1             ...          0.7061855670103093
2             ...          0.7032418952618454
3             ...          0.6990291262135923
4             ...          0.6923076923076924
5             ...          0.6871794871794872
6             ...          0.6745843230403801
7             ...          0.6742209631728044
8             ...          0.6738544474393531
9             ...          0.6734693877551019
10            ...           0.668076109936575
11            ...          0.6610169491525424
12            ...           0.657210401891253
13            ...          0.6498855835240274
14            ...          0.6324435318275153
15            ...          0.6315789473684211
```

Le meilleur modèle combine (ntrees = 50) et (max_depth = 2) avec un F1-Score de 0.713. Ce que confirme l'accès explicite à ses paramètres.

```
#paramètres du premier (meilleur) modèle
gbModels.get_hyperparams(0)
```

```
Hyperparameters: [ntrees, max_depth]
[50, 2]
```

Récupération du meilleur modèle et performances en test. Nous accédons au meilleur modèle, nous affichons pour vérifications les paramètres associés.

```
#récupération du meilleur modele
gbBest = gbModels[0]

#vérification - nombre d'arbre
gbBest.get_params()['ntrees']['actual_value']      #50

#vérification - profondeur max
gbBest.get_params()['max_depth']['actual_value']    #2
```

Et nous l'appliquons sur notre échantillon test.

```
#prédiction sur le test
predGbBest = gbBest.predict(pimaTest).as_data_frame()

#F1-score
print(metrics.f1_score(pimaTest.as_data_frame()["diabete"],predGbBest.predict,pos_label="positive"))
```

```
0.6378378378378379
```

Finalement, notre modèle, dont les paramètres ont été détectés par validation croisée, ne s'avère pas être si bon que cela en test. Notamment parce qu'avec des effectifs aussi faibles, les résultats sont fatalement très instables. Mais la démarche mérite d'être soulignée. Elle est très prisée des data scientists parce qu'elle répond à une vision de la recherche systématique de la performance, et qu'elle est assez mécanique, facile à mettre en place même sans connaissance approfondie des techniques de machine learning utilisées.

4.4 H2OAutoML

L'outil H2OAutoML (Automatic Machine Learning) correspond au même état d'esprit. Mieux même, il prétend détecter pour nous le meilleur modèle possible. Il procède toujours par validation croisée pour l'exploration de l'espace des solutions mais, au-delà de la recherche des paramètres optimaux, permet aussi de tester différentes familles d'algorithmes et de les combiner. « H2O's AutoML can be used for automating the machine learning workflow, which includes automatic training and tuning of many models within a user-specified time-limit. Stacked Ensembles – one based on all previously trained models, another one on the best model of each family – will be automatically trained on collections of individual models to produce highly predictive ensemble models which, in most cases, will be the top performing models in the AutoML Leaderboard. ». Le discours est séduisant, c'est le moins qu'on puisse dire. H2O met à la portée des néophytes toute la puissance de l'intelligence artificielle (je suis ébloui par ce que je viens d'écrire, argh…). Bon, trêve de plaisanterie, regardons ce que cela donne sur notre base « pima ». Rien ne vaut l'épreuve du feu (des données).

Paramétrage et instanciation. Il y a deux manières de limiter la recherche des solutions sous AutoML : fixer le nombre de modèles à essayer, ou fixer une durée limite des calculs. Je trouve cette seconde option particulièrement intéressante dans un contexte d'études réelles. On peut toujours essayer d'optimiser, mais on ne peut pas le faire indéfiniment. Sachant que la durée de calcul pour chaque méthode n'est pas toujours maîtrisée, pouvoir borner le temps que l'on consacre à l'exploration correspond à la pratique réelle des data scientists. Nous choisissons d'allouer 180 secondes (3 minutes) à AutoML puis nous faisons appel à train(). La performance de chaque modèle est mesurée en (nfolds = 5) validation croisée.

```
#chargement de la classe
from h2o.automl import H2OAutoML

#instanciation
aml = H2OAutoML(seed=100,nfolds=5,max_runtime_secs=180)

#lancement des calculs
aml.train(x=pimaTrain.columns[:-1],y="diabete",training_frame=pimaTrain)
```

35 modèles ont été évalués, nous affichons leurs identifiants avec la valeur de l'AUC correspondante.

```
#récupérer le tableau des modèles
lb = aml.leaderboard

#nombre de modèles
print(lb.nrow) #35

#afficher les modèles -- tri par défaut AUC pour le classement binaire
result = lb.head(rows=lb.nrow).as_data_frame()
result.loc[:,["model_id","auc"]]
```

```
0   StackedEnsemble_BestOfFamily_AutoML_20190103_0...  0.844782
1           GBM_grid_1_AutoML_20190103_082421_model_4  0.842537
2           GLM_grid_1_AutoML_20190103_082421_model_1  0.841935
3    StackedEnsemble_AllModels_AutoML_20190103_082421  0.841823
4           GBM_grid_1_AutoML_20190103_082421_model_8  0.835036
5                       GBM_5_AutoML_20190103_082421  0.834073
6                       GBM_2_AutoML_20190103_082421  0.834030
7                       GBM_3_AutoML_20190103_082421  0.829488
8          GBM_grid_1_AutoML_20190103_082421_model_18  0.829334
9                       GBM_4_AutoML_20190103_082421  0.829282
10         GBM_grid_1_AutoML_20190103_082421_model_20  0.828895
11                      GBM_1_AutoML_20190103_082421  0.828731
12          GBM_grid_1_AutoML_20190103_082421_model_2  0.826744
13  DeepLearning_grid_1_AutoML_20190103_082421_mod...  0.825213
14         GBM_grid_1_AutoML_20190103_082421_model_12  0.823020
15  DeepLearning_grid_1_AutoML_20190103_082421_mod...  0.820250
16  DeepLearning_grid_1_AutoML_20190103_082421_mod...  0.813524
17  DeepLearning_grid_1_AutoML_20190103_082421_mod...  0.805412
18         GBM_grid_1_AutoML_20190103_082421_model_16  0.805214
19         GBM_grid_1_AutoML_20190103_082421_model_10  0.803933
```

```
20                        DRF_1_AutoML_20190103_082421  0.803391
21             DeepLearning_1_AutoML_20190103_082421  0.802840
22                        XRT_1_AutoML_20190103_082421  0.799073
23         GBM_grid_1_AutoML_20190103_082421_model_13  0.795993
24         GBM_grid_1_AutoML_20190103_082421_model_19  0.791821
25          GBM_grid_1_AutoML_20190103_082421_model_9  0.788639
26          GBM_grid_1_AutoML_20190103_082421_model_1  0.785155
27         GBM_grid_1_AutoML_20190103_082421_model_17  0.783555
28          GBM_grid_1_AutoML_20190103_082421_model_7  0.782704
29          GBM_grid_1_AutoML_20190103_082421_model_5  0.744547
30         GBM_grid_1_AutoML_20190103_082421_model_14  0.727825
31          GBM_grid_1_AutoML_20190103_082421_model_6  0.723610
32         GBM_grid_1_AutoML_20190103_082421_model_11  0.661523
33          GBM_grid_1_AutoML_20190103_082421_model_3  0.643347
34         GBM_grid_1_AutoML_20190103_082421_model_15  0.555395
```

Un petit mot sur la manière de procéder d'AutoML pour bien comprendre ces résultats. Il essaie différentes familles d'algorithmes, couplés avec différentes valeurs des paramètres clés (qu'il choisit lui-même). Jusque là pas de surprise. Ensuite il « empile », au sens du stacking[1], des groupes de modèles : tous les modèles (StackedEnsemble_AllModels_AutoML), les meilleurs modèles de chaque famille (StackedEnsemble_BestOfFamily). Pour notre jeu de données, c'est ce dernier qui s'est révélé le plus intéressant au sens de l'AUC (0.8447 en validation croisée).

Nous pouvons afficher les propriétés du modèle leader.

```
#meilleur modèle
aml.leader
```

```
Model Details
=============
H2OStackedEnsembleEstimator :  Stacked Ensemble
Model Key:  StackedEnsemble_BestOfFamily_AutoML_20190103_082421
No model summary for this model


ModelMetricsBinomialGLM: stackedensemble
** Reported on train data. **

MSE: 0.11071156038576531
RMSE: 0.3327334674867638
LogLoss: 0.36310177919539594
Null degrees of freedom: 508
Residual degrees of freedom: 504
Null deviance: 652.4944896569244
Residual deviance: 369.6376112209131
AIC: 379.6376112209131
```

[1] Le stacking (stacked ensembles) consiste à créer un ensemble de modèles de familles différentes (ou de la même famille mais avec des paramètres différents) sur les mêmes données puis, à partir de leurs prédictions en validation croisée qui font office de variables explicatives, et toujours la variable cible, de créer un méta-modèle (un modèle de modèles). L'approche n'est performante que si les modèles individuels le sont, tout en étant dissemblables les uns des autres, c.-à-d. ne font pas leurs erreurs sur les mêmes individus. Ainsi, les modèles se compensent efficacement.

```
AUC: 0.932399187998899
pr_auc: 0.8646492848189802
Gini: 0.864798375997798

Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.4117348789161613:
          negative    positive    Error    Rate
--------  ----------  ----------  -------  ------------
negative  282         54          0.1607   (54.0/336.0)
positive  15          158         0.0867   (15.0/173.0)
Total     297         212         0.1356   (69.0/509.0)

Maximum Metrics: Maximum metrics at their respective thresholds
metric                        threshold    value     idx
----------------------------  -----------  --------  -----
max f1                        0.411735     0.820779  183
max f2                        0.294157     0.878378  232
max f0point5                  0.531571     0.806658  129
max accuracy                  0.450817     0.866405  164
max precision                 0.901897     1         0
max recall                    0.0854117    1         340
max specificity               0.901897     1         0
max absolute_mcc              0.411735     0.723102  183
max min_per_class_accuracy    0.439794     0.861272  168
max mean_per_class_accuracy   0.411735     0.87629   183
Gains/Lift Table: Avg response rate: 33,99 %, avg score: 36,94 %
```

group	cumulative_data_fraction	lower_threshold	lift	cumulative_lift	response_rate	score	cumulative_response_rate	cumulative_score	capture_rate	cumulative_capture_rate	gain	cumulative_gain
1	0.0117878	0.869197	2.9422	2.9422	1	0.883804	1	0.883804	0.0346821	0.0346821	194.22	194.22
2	0.021611	0.865328	2.9422	2.9422	1	0.866668	1	0.876015	0.0289017	0.0635838	194.22	194.22
3	0.0314342	0.858013	2.9422	2.9422	1	0.860219	1	0.871079	0.0289017	0.0924855	194.22	194.22
4	0.0412574	0.849049	2.9422	2.9422	1	0.852983	1	0.86677	0.0289017	0.121387	194.22	194.22
5	0.0510806	0.843653	2.9422	2.9422	1	0.84674	1	0.862918	0.0289017	0.150289	194.22	194.22
6	0.100196	0.784206	2.70682	2.82682	0.92	0.817576	0.960784	0.840692	0.132948	0.283237	170.682	182.682
7	0.151277	0.724536	2.37639	2.67472	0.807692	0.755113	0.909091	0.811795	0.121387	0.404624	137.639	167.472
8	0.200393	0.653207	2.00069	2.50952	0.68	0.689984	0.852941	0.781939	0.0982659	0.50289	100.069	150.952
9	0.300589	0.526242	2.24991	2.42299	0.764706	0.589467	0.823529	0.717782	0.225434	0.728324	124.991	142.299
10	0.400786	0.426401	1.61532	2.22107	0.54902	0.473306	0.754902	0.656663	0.16185	0.890173	61.5324	122.107
11	0.500982	0.319678	0.634591	1.90377	0.215686	0.367614	0.647059	0.598853	0.0635838	0.953757	-36.5409	90.3774
12	0.599214	0.249557	0.29422	1.63991	0.1	0.282357	0.557377	0.546968	0.0289017	0.982659	-70.578	63.9913
13	0.699411	0.143663	0.11538	1.42151	0.0392157	0.208644	0.483146	0.498501	0.0115607	0.99422	-88.462	42.1511
14	0.799607	0.0839275	0.0576901	1.25061	0.0196078	0.111084	0.425061	0.449955	0.00578035	1	-94.231	25.0614
15	0.899804	0.0435711	0	1.11135	0	0.0565368	0.377729	0.406146	0	1	-100	11.1354
16	1	0.0371031	0	1	0	0.0397857	0.339882	0.369438	0	1	-100	0

```
ModelMetricsBinomialGLM: stackedensemble
** Reported on cross-validation data. **

MSE: 0.15242395931892277
RMSE: 0.3904151115401692
LogLoss: 0.4601557692708877
Null degrees of freedom: 508
Residual degrees of freedom: 504
Null deviance: 656.161616837367
Residual deviance: 468.4385731177636
AIC: 478.4385731177636
AUC: 0.8447822047894301
```

```
pr_auc: 0.6925992286788722
Gini: 0.6895644095788602

Confusion Matrix (Act/Pred) for max f1 @ threshold = 0.238570360454366:
          negative    positive    Error    Rate
--------  ----------  ----------  -------  -------------
negative  221         115         0.3423   (115.0/336.0)
positive  18          155         0.104    (18.0/173.0)
Total     239         270         0.2613   (133.0/509.0)

Maximum Metrics: Maximum metrics at their respective thresholds
metric                          threshold    value     idx
------------------------------  -----------  --------  -----
max f1                          0.23857      0.699774  239
max f2                          0.147832     0.817121  291
max f0point5                    0.599947     0.701079  108
max accuracy                    0.599947     0.78389   108
max precision                   0.934221     1         0
max recall                      0.0354528    1         393
max specificity                 0.934221     1         0
max absolute_mcc                0.23857      0.525509  239
max min_per_class_accuracy      0.311093     0.757225  192
max mean_per_class_accuracy     0.23857      0.776846  239
Gains/Lift Table: Avg response rate: 33,99 %, avg score: 34,05 %
```

group	cumulative_data_fraction	lower_threshold	lift	cumulative_lift	response_rate	score	cumulative_response_rate	cumulative_score	capture_rate	cumulative_capture_rate	gain	cumulative_gain
1	0.0117878	0.921223	2.45183	2.45183	0.833333	0.926884	0.833333	0.926884	0.0289017	0.0289017	145.183	145.183
2	0.021611	0.896247	2.35376	2.40725	0.8	0.906435	0.818182	0.917589	0.0231214	0.0520231	135.376	140.725
3	0.0314342	0.889217	2.35376	2.39053	0.8	0.892736	0.8125	0.909823	0.0231214	0.0751445	135.376	139.053
4	0.0412574	0.873818	1.76532	2.24167	0.6	0.879784	0.761905	0.902671	0.017341	0.0924855	76.5318	124.167
5	0.0510806	0.867641	2.9422	2.37639	1	0.87033	0.807692	0.896451	0.0289017	0.121387	194.22	137.639
6	0.100196	0.801412	2.11838	2.24991	0.72	0.831747	0.764706	0.864734	0.104046	0.225434	111.838	124.991
7	0.151277	0.744028	2.15007	2.2162	0.730769	0.773035	0.753247	0.83377	0.109827	0.33526	115.007	121.62
8	0.200393	0.664301	2.23607	2.22107	0.76	0.699962	0.754902	0.800974	0.109827	0.445087	123.607	122.107
9	0.300589	0.47229	1.49994	1.98069	0.509804	0.564051	0.673203	0.722	0.150289	0.595376	49.9943	98.0694
10	0.400786	0.326154	1.38456	1.83166	0.470588	0.400287	0.622549	0.641572	0.138728	0.734104	38.4563	83.1662
11	0.500982	0.253647	1.1538	1.69609	0.392157	0.285471	0.576471	0.570352	0.115607	0.849711	15.3803	69.609
12	0.599214	0.18569	0.764971	1.54345	0.26	0.225415	0.52459	0.513805	0.0751445	0.924855	-23.5029	54.3447
13	0.699411	0.112473	0.576901	1.40498	0.196078	0.150613	0.477528	0.461774	0.0578035	0.982659	-42.3099	40.4981
14	0.799607	0.062317	0.11538	1.24339	0.0392157	0.0859229	0.422604	0.414678	0.0115607	0.99422	-88.462	24.3385
15	0.899804	0.0435192	0	1.10493	0	0.0494273	0.375546	0.374006	0	0.99422	-100	10.493
16	1	0.0301414	0.0576901	1	0.0196078	0.0391869	0.339882	0.340458	0.00578035	1		

Les résultats en validation croisée sont ceux qui nous intéressent le plus. On nous annonce un F1-Score de 0.699774. Voyons ce qu'il en est en test.

Performance en test du meilleur modèle. Nous appliquons le modèle leader sur notre échantillon test et nous mesurons le F1-Score.

```
#prédiction sur le test
predAml = aml.predict(pimaTest).as_data_frame()

#F1-score
print(metrics.f1_score(pimaTest.as_data_frame()["diabete"],predAml.predict,pos_label="positive"))
```

```
0.6694560669456067
```

Empiler 35 modèles pour obtenir des performances équivalentes aux autres approches « simples » (section 3.7), on se demande où est l'intérêt…. Encore une fois, c'est la démarche ici qui est intéressante. Après arrive toujours une limite où, malgré tous nos efforts, nous ne pouvons plus tirer de l'information utile (pour la prédiction) des données, même en les triturant dans tous les sens.

5 Conclusion

J'avais entendu parler de H2O depuis un moment déjà. A l'époque, après un rapide coup d'œil, je n'avais pas constaté d'éléments qui pourraient justifier l'écriture d'un tutoriel spécifique à son sujet. Plus récemment, je l'ai de nouveau inspectée de plus près en m'intéressant aux packages pour le Deep Learning. J'ai finalement décidé de me pencher attentivement sur la plateforme avec pour premier objectif l'évaluation de sa capacité à paralléliser les algorithmes de machine learning. Pour avoir moi-même étudié ce sujet à plusieurs reprises, je sais combien la tâche est difficile. Force est de constater qu'elle sait le faire efficacement, indubitablement (section 2.3). Elle a su exploiter au mieux le processeur multicœur de ma machine. Dans un deuxième temps, à force de m'escrimer dessus, je me suis rendu compte que H2O proposait des fonctionnalités intéressantes pour le machine learning. J'ai pris plaisir à les explorer et à les exposer.

6 Références

H2O.ai – 3.22.1.1, http://docs.h2o.ai/h2o/latest-stable/h2o-docs/index.html (© Copyright 2016-2018 H2O.ai. Last updated on Dec 28, 2018.)

H2O Tutorials, http://docs.h2o.ai/h2o-tutorials/latest-stable/index.html

The H2O Python Module, http://h2o-release.s3.amazonaws.com/h2o/master/3574/docs-website/h2o-py/docs/intro.html (© Copyright 2015, H2O.ai.).