Random Forest avec Python

Mathis Cordier

January 13, 2020

Nous présenterons dans ce notebook, un exemple d'analyse de données utilisant les forêts aléatoires. Nous exposerons les différentes étapes de manière la plus générale possible afin que ce TP soit reproductible quelque soit le jeu de données. Tout au long du TP nous utiliserons le package sklearn et dans la dernière partie nous utiliserons xgboost.

1 Préambule

Nous utiliserons le jeu de données Breast cancer, disponible sur Kaggle, qui traite du cancer du sein dans le Wisconsin. Nous disposons d'une matrice de design X à 569 individus et 30 variables explicatives quantitatives et d'une variable qualitative à expliquer Y représentant le diagnostic d'une tumeur "Bénigne" ou "Maligne" de l'individu. Ces 2 états sont présents en quantité à peu près équivalentes dans ce jeu de données, on peut donc utiliser des méthodes d'analyses classiques.

```
In [2]: import pandas as pd
        import numpy as np
        import os
        import matplotlib.pyplot as plt
```

```
In [3]: X = pd.read_csv("./breast_cancer.csv").iloc[:,1:32]
        Y = np.array(X.iloc[:,0])
        X = X.iloc[:,1:]
        X.head()
```

```
Out[3]:    radius_mean  texture_mean  perimeter_mean  area_mean  smoothness_mean  \
        0        17.99         10.38          122.80     1001.0          0.11840
        1        20.57         17.77          132.90     1326.0          0.08474
        2        19.69         21.25          130.00     1203.0          0.10960
        3        11.42         20.38           77.58      386.1          0.14250
        4        20.29         14.34          135.10     1297.0          0.10030

           compactness_mean  concavity_mean  concave_points_mean  symmetry_mean  \
        0           0.27760          0.3001              0.14710         0.2419
        1           0.07864          0.0869              0.07017         0.1812
        2           0.15990          0.1974              0.12790         0.2069
        3           0.28390          0.2414              0.10520         0.2597
        4           0.13280          0.1980              0.10430         0.1809
```

```
   fractal_dimension_mean           ...             radius_worst  \
0                 0.07871           ...                    25.38
1                 0.05667           ...                    24.99
2                 0.05999           ...                    23.57
3                 0.09744           ...                    14.91
4                 0.05883           ...                    22.54

   texture_worst  perimeter_worst  area_worst  smoothness_worst  \
0          17.33           184.60      2019.0            0.1622
1          23.41           158.80      1956.0            0.1238
2          25.53           152.50      1709.0            0.1444
3          26.50            98.87       567.7            0.2098
4          16.67           152.20      1575.0            0.1374

   compactness_worst  concavity_worst  concave_points_worst  symmetry_worst  \
0             0.6656           0.7119                0.2654          0.4601
1             0.1866           0.2416                0.1860          0.2750
2             0.4245           0.4504                0.2430          0.3613
3             0.8663           0.6869                0.2575          0.6638
4             0.2050           0.4000                0.1625          0.2364

   fractal_dimension_worst
0                  0.11890
1                  0.08902
2                  0.08758
3                  0.17300
4                  0.07678

[5 rows x 30 columns]
```

On peut obtenir les niveaux pour Y :

- "B" correspondant à une tumeur bénigne
- "M" correspondant à une tumeur maligne

```
In [4]: list(np.unique(Y))

Out[4]: ['B', 'M']

In [5]: X.describe()

Out[5]:        radius_mean  texture_mean  perimeter_mean    area_mean  \
       count   569.000000    569.000000      569.000000   569.000000
       mean     14.127292     19.289649       91.969033   654.889104
       std       3.524049      4.301036       24.298981   351.914129
       min       6.981000      9.710000       43.790000   143.500000
```

```
25%      11.700000     16.170000       75.170000   420.300000
50%      13.370000     18.840000       86.240000   551.100000
75%      15.780000     21.800000      104.100000   782.700000
max      28.110000     39.280000      188.500000  2501.000000

       smoothness_mean  compactness_mean  concavity_mean  concave_points_mean  \
count       569.000000        569.000000      569.000000           569.000000
mean          0.096360          0.104341        0.088799             0.048919
std           0.014064          0.052813        0.079720             0.038803
min           0.052630          0.019380        0.000000             0.000000
25%           0.086370          0.064920        0.029560             0.020310
50%           0.095870          0.092630        0.061540             0.033500
75%           0.105300          0.130400        0.130700             0.074000
max           0.163400          0.345400        0.426800             0.201200

       symmetry_mean  fractal_dimension_mean           ...             \
count     569.000000              569.000000           ...
mean        0.181162                0.062798           ...
std         0.027414                0.007060           ...
min         0.106000                0.049960           ...
25%         0.161900                0.057700           ...
50%         0.179200                0.061540           ...
75%         0.195700                0.066120           ...
max         0.304000                0.097440           ...

       radius_worst  texture_worst  perimeter_worst   area_worst  \
count    569.000000     569.000000       569.000000   569.000000
mean      16.269190      25.677223       107.261213   880.583128
std        4.833242       6.146258        33.602542   569.356993
min        7.930000      12.020000        50.410000   185.200000
25%       13.010000      21.080000        84.110000   515.300000
50%       14.970000      25.410000        97.660000   686.500000
75%       18.790000      29.720000       125.400000  1084.000000
max       36.040000      49.540000       251.200000  4254.000000

       smoothness_worst  compactness_worst  concavity_worst  \
count        569.000000         569.000000       569.000000
mean           0.132369           0.254265         0.272188
std            0.022832           0.157336         0.208624
min            0.071170           0.027290         0.000000
25%            0.116600           0.147200         0.114500
50%            0.131300           0.211900         0.226700
75%            0.146000           0.339100         0.382900
max            0.222600           1.058000         1.252000

       concave_points_worst  symmetry_worst  fractal_dimension_worst
count            569.000000      569.000000               569.000000
mean               0.114606        0.290076                 0.083946
```

```
std                0.065732        0.061867                0.018061
min                0.000000        0.156500                0.055040
25%                0.064930        0.250400                0.071460
50%                0.099930        0.282200                0.080040
75%                0.161400        0.317900                0.092080
max                0.291000        0.663800                0.207500

[8 rows x 30 columns]
```

2 Préparation des données

On va scinder le jeu de données en 2 afin d'obtenir une base d'entraînement et une base de test.

```
In [6]: from sklearn.model_selection import GridSearchCV, train_test_split
        from sklearn.metrics import confusion_matrix, accuracy_score
```

On choisit de conserver 50% du jeu de données pour la base d'entraînement et 50% pour la base de test.

```
In [7]: test_rate = 0.5
        Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y,
                                                        test_size=test_rate,
                                                        random_state=0)
        print(Xtrain.shape)
        print(Xtest.shape)
```

```
(284, 30)
(285, 30)
```

On obtient alors une base de données d'entrainement de 284 individus et une base de données de test de 285 individus.

3 Choix de l'algorithme

Nous utiliserons 3 types d'algorithmes différents :

- Random forest
- Gradient boosting
- Extrem gradient boosting

3.1 Random Forest

```
In [8]: from sklearn.ensemble import RandomForestClassifier
```

Nous disposons d'un jeu de données d'entrainement de 284 individus et 30 variables. Nous pouvons commencer par effectuer un random forest avec le paramètrage par défaut de sklearn. On appellera cette méthode la méthode naïve.

```
In [9]: rf = RandomForestClassifier(random_state=1)
        rf.fit(Xtrain,Ytrain)

Out[9]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                    max_depth=None, max_features='auto', max_leaf_nodes=None,
                    min_impurity_decrease=0.0, min_impurity_split=None,
                    min_samples_leaf=1, min_samples_split=2,
                    min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=None,
                    oob_score=False, random_state=1, verbose=0, warm_start=False)
```

On peut maintenant prédire les résultats pour la population d'entraînement.

```
In [10]: Ypred = rf.predict(Xtest)
         100*accuracy_score(Ytest,Ypred)

Out[10]: 94.03508771929825
```

On obtient alors 94.03% de bonnes prédiction qui est déjà un très bon résultat avec l'utilisation brute de la fonction.

```
In [11]: confusion_matrix(Ytest,Ypred)

Out[11]: array([[177,   7],
                [ 10,  91]], dtype=int64)
```

On remarque également que ce modèle a tendance à prédire plus de tumeurs bénignes qui sont réellement malignes que l'inverse.

On passe maintenant à la méthode plus fine dans laquelle nous allons adapter les paramètres à notre base de données.

Afin de définir les paramètres les plus adaptés à nos données, nous commençons par définir l'espace auquel appartiennent les paramètres puis nous procédons à une validation croisée 4-folds.

```
In [12]: N_core = -1 #Permet d'utiliser tous les coeurs du processeur
         parametres = {'max_features':[10,12,14],
                       'n_estimators':[200,500,800,1400,2000],
                       'criterion':['gini','entropy'],
                       'max_depth':[None,5,10]}
```

Par défaut, on fixe `max_features`, le nombre maximal de variables utilisées par arbres à entre 30% et 40% du nombre total de variables (ici 31). Le paramètre `n_estimators` indique le nombre d'arbres à générer dans le Random forest. Lorsque `max_depth` vaut 'None', les noeuds de l'arbres sont étendues tant que les feuilles sont pures.

```
In [13]: rf = RandomForestClassifier(random_state=2)
         tuning = GridSearchCV(estimator = rf, param_grid = parametres,
                               scoring='accuracy', n_jobs=N_core, cv=4)
```

On parcourt ensuite la grille de paramètres ci-dessus et on définit la meilleure combinaison de paramètres par validation croisée.

```
In [14]: fitted = tuning.fit(Xtrain,Ytrain)
```

On peut alors comparer les différents modèles entre-eux pour retenir le meilleur. Les meilleurs paramètres sont :

```
In [15]: tuning.best_params_

Out[15]: {'criterion': 'gini',
          'max_depth': None,
          'max_features': 10,
          'n_estimators': 200}
```

Le meilleur score correspondant au modèle ci-dessus :

```
In [16]: 100*tuning.best_score_

Out[16]: 94.36619718309859
```

Il est important de prendre également en compte l'écart type de la prédiction. Pour comparer les meilleurs modèles, on peut les mettre dans un tableau. On affiche les modèles ayant les meilleurs scores puis si 2 modèles ont un taux de bonnes prédiction similaires, on prend celui ayant l'écart type le plus faible.

```
In [17]: grid_results_ = pd.DataFrame(columns=['Score','Score_std','MinScore'])
         for i in range(len(tuning.cv_results_['mean_test_score'])):
             d = tuning.cv_results_['params'][i]
             sc = 100*tuning.cv_results_['mean_test_score'][i]
             sc_std = 100*tuning.cv_results_['std_test_score'][i]
             d['Score'] = sc
             d['Score_std'] = sc_std
             d['MinScore'] = sc-sc_std
             grid_results_ = grid_results_.append(d,ignore_index=True)
         grid_results_.sort_values(by=['MinScore','Score','Score_std'],
                                   ascending=[False,False,True]).head(10)
```

```
Out[17]:         Score  Score_std   MinScore criterion max_depth  max_features  \
         20  94.014085   1.490326  92.523758      gini         5          12.0
         51  94.014085   1.490326  92.523758   entropy      None          12.0
         53  94.014085   1.490326  92.523758   entropy      None          12.0
         66  94.014085   1.490326  92.523758   entropy         5          12.0
         67  94.014085   1.490326  92.523758   entropy         5          12.0
         68  94.014085   1.490326  92.523758   entropy         5          12.0
         81  94.014085   1.490326  92.523758   entropy        10          12.0
         83  94.014085   1.490326  92.523758   entropy        10          12.0
         52  93.661972   1.163267  92.498705   entropy      None          12.0
         57  93.661972   1.163267  92.498705   entropy      None          14.0

             n_estimators
         20         200.0
```

```
51          500.0
53         1400.0
66          500.0
67          800.0
68         1400.0
81          500.0
83         1400.0
52          800.0
57          800.0
```

Ici, on retient le plus petit modèle qui obtient le meilleur résultat, soit celui formé par les critères :

- critère : entropie
- profondeur maximale automatique
- 12 variables maximales par arbre

On choisit ce modèle plutôt que celui de la première ligne car le critère de profondeur automatique paraît plus adapté car il permet de donner de la flexibilité aux arbres en fonction des variables qui sont sélectionnées pour chacun d'eux.

On atteint ainsi 96.48% de bonnes prédictions en validation croisée avec un écart type de 0.68% sur la population d'entraînement avec une validation croisée 4-folds.

Nous pouvons maintenant prédire le type de tumeurs pour les 285 individus de la base de test.

```
In [18]: rf = RandomForestClassifier(criterion='entropy',
                                     max_features=12,
                                     n_estimators=4000)
         rf.fit(Xtrain,Ytrain)
```

```
Out[18]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
                       max_depth=None, max_features=12, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=4000, n_jobs=None,
                       oob_score=False, random_state=None, verbose=0,
                       warm_start=False)
```

On peut maintenant prédire les résultats pour la population d'entraînement.

```
In [19]: Ypred = rf.predict(Xtest)
         100*accuracy_score(Ytest,Ypred)
```

```
Out[19]: 95.08771929824562
```

Nous obtenons une amélioration de la prédiction.

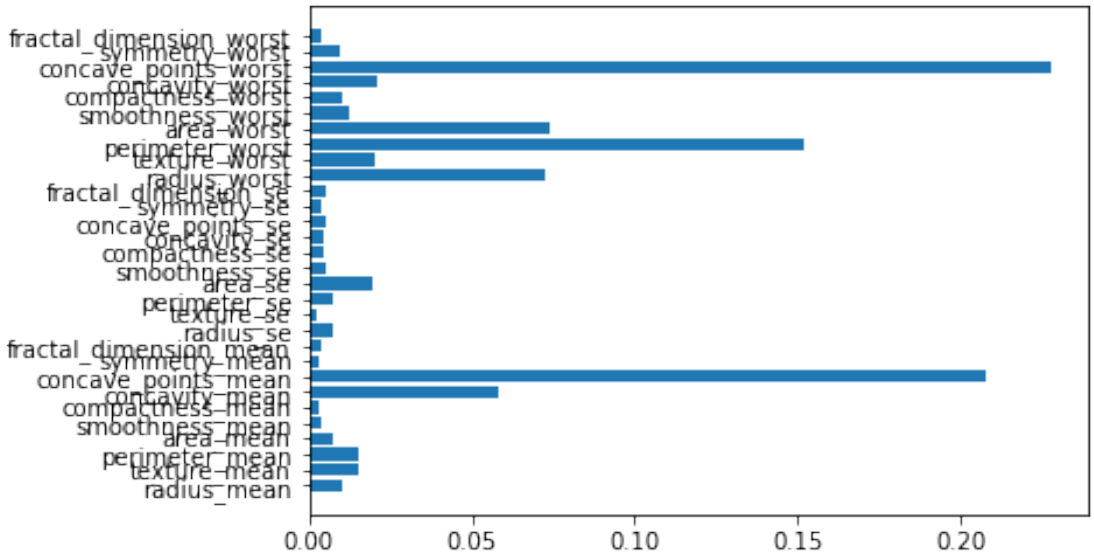
```
In [20]: confusion_matrix(Ytest,Ypred)
```

```
Out[20]: array([[181,   3],
                [ 11,  90]], dtype=int64)
```

On remarque ici que les prédictions gagnées par rapport au premier modèle se sont faites sur les tumeurs bénignes. On a perdu des bonnes prédictions sur les tumeurs malignes.

On peut aussi comparer l'importance des variables dans le modèle.

```
In [21]: use = rf.feature_importances_
         feat = list(Xtrain.columns)
         p = plt.barh(feat,use)
```



Nous allons maintenant voir des méthodes dérivées de cette dernière. La suivante étant le Gradient Boosting.

3.2 Gradient Boosting

```
In [22]: from sklearn.ensemble import GradientBoostingClassifier
```

Nous utilisons le même package que pour le random forest classique. Un grand avantage de ce package réside dans le fait que la procédure dans le cas du boosting est exactement la même que la procédure précédente.

Nous commençons alors par la méthode naïve sur la classification par gradient boosting.

```
In [23]: gb = GradientBoostingClassifier(random_state=3)
         gb.fit(Xtrain,Ytrain)

Out[23]: GradientBoostingClassifier(criterion='friedman_mse', init=None,
                           learning_rate=0.1, loss='deviance', max_depth=3,
                           max_features=None, max_leaf_nodes=None,
                           min_impurity_decrease=0.0, min_impurity_split=None,
                           min_samples_leaf=1, min_samples_split=2,
                           min_weight_fraction_leaf=0.0, n_estimators=100,
```

```
                        n_iter_no_change=None, presort='auto', random_state=3,
                        subsample=1.0, tol=0.0001, validation_fraction=0.1,
                        verbose=0, warm_start=False)
```

On prédit alors les valeurs sur l'échantillon test.

```
In [24]: Ypred = rf.predict(Xtest)
         100*accuracy_score(Ytest,Ypred)

Out[24]: 95.08771929824562
```

On obtient une nouvelle fois 95.09% de bonnes prédictions.

```
In [25]: confusion_matrix(Ytest,Ypred)

Out[25]: array([[181,   3],
                [ 11,  90]], dtype=int64)
```

On retrouve ici exactement la même prédiction que la méthode fine de Random Forest précédente.

Passons maintenant à la méthode fine de Gradient Boosting.

```
In [26]: N_core = -1 #Permet d'utiliser tous les coeurs du processeur
         parametres = {'max_features':[5,6,7,8],
                       'learning_rate':[0.1,0.05,0.01],
                       'max_depth':[None], #default=3
                       'loss':["deviance","exponential"],
                       'n_estimators':[2000]}
```

Dans le cas où `loss` vaut `exponential`, la fonction de classification suivante va utiliser l'algorithme Adaboost. La variable `learning_rate` indique le paramètre de pas de la descente de gradient.

```
In [27]: gb = GradientBoostingClassifier(random_state=5)
         tuning = GridSearchCV(estimator = gb, param_grid = parametres,
                               scoring='accuracy', n_jobs=N_core, cv=3)
```

On parcourt maintenant les différentes combinaisons possibles de paramètres afin de trouver le meilleur modèle.

```
In [28]: fitted = tuning.fit(Xtrain,Ytrain)
```

On retrouve les paramètres du meilleur modèle ci-dessous.

```
In [29]: tuning.best_params_

Out[29]: {'learning_rate': 0.1,
          'loss': 'exponential',
          'max_depth': None,
          'max_features': 6,
          'n_estimators': 2000}
```

```
In [30]: 100*tuning.best_score_
```

```
Out[30]: 95.4225352112676
```

On obtient une très bonne erreur de validation croisée.

```
In [31]: grid_results_ = pd.DataFrame(columns=['Score','Score_std','MinScore','max_features',
                                               'learning_rate','max_depth','loss'])
         for i in range(len(tuning.cv_results_['mean_test_score'])):
             d = tuning.cv_results_['params'][i]
             sc = 100*tuning.cv_results_['mean_test_score'][i]
             sc_std = 100*tuning.cv_results_['std_test_score'][i]
             d['Score'] = sc
             d['Score_std'] = sc_std
             d['MinScore'] = sc-sc_std
             grid_results_ = grid_results_.append(d,ignore_index=True)
         grid_results_.sort_values(by=['MinScore','Score','Score_std'],
                                   ascending=[False,False,True]).head(10)
```

```
Out[31]:         Score  Score_std   MinScore max_features  learning_rate max_depth  \
         3   94.718310   0.026348  94.691962            8           0.10      None
         7   94.718310   0.026348  94.691962            8           0.10      None
         14  94.718310   0.026348  94.691962            7           0.05      None
         13  95.070423   0.474257  94.596166            6           0.05      None
         1   95.070423   0.510301  94.560122            6           0.10      None
         5   95.422535   0.974861  94.447674            6           0.10      None
         10  94.366197   0.484051  93.882147            7           0.05      None
         18  94.366197   0.484051  93.882147            7           0.01      None
         22  94.366197   0.484051  93.882147            7           0.01      None
         9   94.718310   0.840534  93.877776            6           0.05      None

                    loss  n_estimators
         3      deviance        2000.0
         7   exponential        2000.0
         14  exponential        2000.0
         13  exponential        2000.0
         1      deviance        2000.0
         5   exponential        2000.0
         10     deviance        2000.0
         18     deviance        2000.0
         22  exponential        2000.0
         9      deviance        2000.0
```

On avait mis 2000 arbres dans la validation croisée. Maintenant qu'on a choisi notre modèle, nous pouvons encore augmenter le nombre d'arbres pour avoir une prédiction encore meilleure.

```
In [32]: gb = GradientBoostingClassifier(max_features=5,
                                         n_estimators=10000,
                                         learning_rate=0.01,
```

```
                                        loss='deviance')
        gb.fit(Xtrain,Ytrain)
```

```
Out[32]: GradientBoostingClassifier(criterion='friedman_mse', init=None,
                       learning_rate=0.01, loss='deviance', max_depth=3,
                       max_features=5, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=10000,
                       n_iter_no_change=None, presort='auto', random_state=None,
                       subsample=1.0, tol=0.0001, validation_fraction=0.1,
                       verbose=0, warm_start=False)
```

```
In [33]: Ypred = gb.predict(Xtest)
         100*accuracy_score(Ytest,Ypred)
```

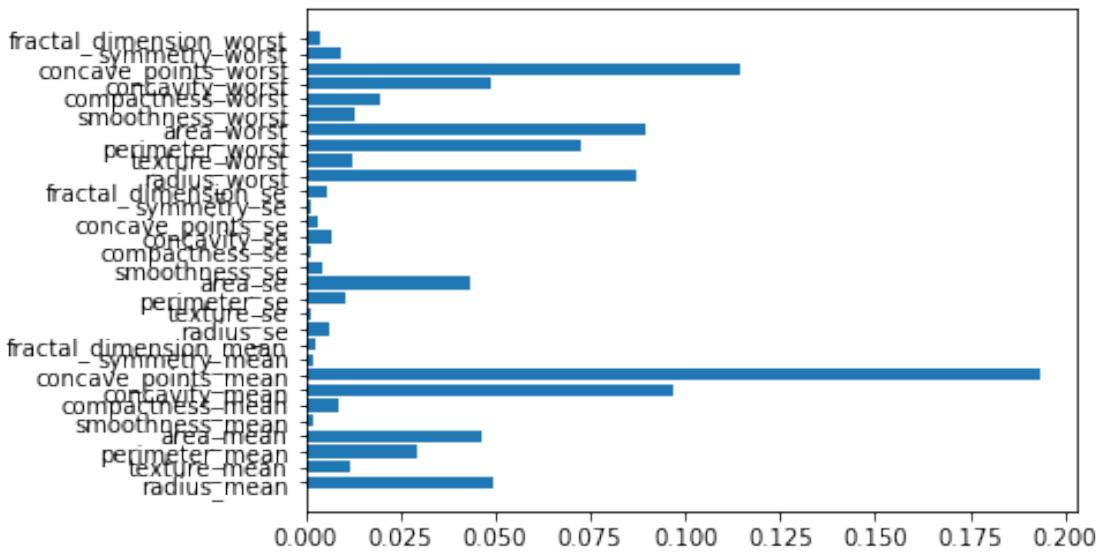
```
Out[33]: 96.49122807017544
```

```
In [34]: confusion_matrix(Ytest,Ypred)
```

```
Out[34]: array([[183,   1],
                [  9,  92]], dtype=int64)
```

On obtient une forte amélioration du score en passant à la prédiction.
Ci-dessous sont représentées les variables et leur importance dans le modèle.

```
In [35]: use = gb.feature_importances_
         feat = list(Xtrain.columns)
         p = plt.barh(feat,use)
```



Nous espérons encore améliorer notre prédiction en passant à la méthode d'Extrem Gradient Boosting.

3.3 Extrem Gradient Boosting

```
In [36]: import xgboost as xgb
```

Pour que les fonctions du package interprètent bien Ytrain et Ytest, nous devons transformer les facteurs "B" et "M" respectivement en 0 et 1. Nous notons également que ce package présente l'avantage d'étre compatible avec les fonctions sklearn que nous utilisions précédemment.

```
In [37]: Ytrain[Ytrain=="B"] = 0
         Ytrain[Ytrain=="M"] = 1
         Ytest[Ytest=="B"] = 0
         Ytest[Ytest=="M"] = 1
         Ytrain = Ytrain.astype('int8')
         Ytest = Ytest.astype('int8')
         Ytrain, Ytest
```

```
Out[37]: (array([0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0,
                 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1,
                 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0,
                 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1,
                 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1,
                 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0,
                 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0,
                 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0,
                 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0,
                 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
                 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1,
                 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0,
                 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0],
                dtype=int8),
          array([1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1,
                 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0,
                 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0,
                 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0,
                 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1,
                 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0,
                 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0,
                 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0,
                 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0,
                 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0,
                 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1,
                 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0],
                dtype=int8))
```

```
In [38]: xmat = xgb.DMatrix(data=Xtrain,label=Ytrain)
```

Méthode naïve :

```
In [39]: xg_class = xgb.XGBClassifier()
         xg_class.fit(Xtrain,Ytrain)

Out[39]: XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
                colsample_bynode=1, colsample_bytree=1, gamma=0, learning_rate=0.1,
                max_delta_step=0, max_depth=3, min_child_weight=1, missing=None,
                n_estimators=100, n_jobs=1, nthread=None,
                objective='binary:logistic', random_state=0, reg_alpha=0,
                reg_lambda=1, scale_pos_weight=1, seed=None, silent=None,
                subsample=1, verbosity=1)

In [40]: Ypred = xg_class.predict(Xtest)
         100*accuracy_score(Ytest,Ypred)

Out[40]: 96.14035087719299

In [41]: confusion_matrix(Ytest,Ypred)

Out[41]: array([[181,   3],
                [  8,  93]], dtype=int64)
```

On remarque que sans même paramétrer la méthode nous obtenons un score presque identique à celui de la méthode précédente.

Méthode fine :

Pour optimiser les paramètres, nous consultons la documentation de xgboost : https://xgboost.readthedocs.io/en/latest/parameter.html

```
In [42]: xg_class = xgb.XGBClassifier()
         N_core = -1 #Permet d'utiliser tous les coeurs du processeur
         parametres = {'learning_rate': [0.01,0.05,0.1],
                       'gamma' : [0,0.25,0.5],
                       'max_depth': [3,4,5],
                       'subsample':[0.5,0.75,1],
                       'colsample_bytree':[0.5,0.75,1],
                       'colsample_by_node':[0.5,0.75,1],
                       "objective":["binary:logistic"],
                       "n_estimatros":[500]}

         tuning = GridSearchCV(estimator = xg_class, param_grid = parametres,
                               scoring='accuracy', n_jobs=N_core, cv=3)

In [43]: tuning.fit(Xtrain,Ytrain)

Out[43]: GridSearchCV(cv=3, error_score='raise-deprecating',
                estimator=XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
                colsample_bynode=1, colsample_bytree=1, gamma=0, learning_rate=0.1,
                max_delta_step=0, max_depth=3, min_child_weight=1, missing=None,
                n_estimators=100, n_jobs=1, nthread=None,
                objective='binary:logistic', random_state=0, reg_alpha=0,
```

```
                reg_lambda=1, scale_pos_weight=1, seed=None, silent=None,
                subsample=1, verbosity=1),
                fit_params=None, iid='warn', n_jobs=-1,
                param_grid={'learning_rate': [0.01, 0.05, 0.1], 'gamma': [0, 0.25, 0.5], 'max_de
                pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
                scoring='accuracy', verbose=0)
```

```
In [44]: tuning.best_params_
```

```
Out[44]: {'colsample_by_node': 0.5,
          'colsample_bytree': 0.5,
          'gamma': 0,
          'learning_rate': 0.1,
          'max_depth': 4,
          'n_estimatros': 500,
          'objective': 'binary:logistic',
          'subsample': 0.5}
```

```
In [45]: 100*tuning.best_score_
```

```
Out[45]: 95.07042253521126
```

On obtient ici une prédiction un peu moins bonne qu'avec la méthode naïve. Nous avons tout de même bien gagner en précision avec la méthode xgboost.

```
In [46]: grid_results_ = pd.DataFrame(columns=['Score','Score_std','MinScore'])
         for i in range(len(tuning.cv_results_['mean_test_score'])):
             d = tuning.cv_results_['params'][i]
             sc = 100*tuning.cv_results_['mean_test_score'][i]
             sc_std = 100*tuning.cv_results_['std_test_score'][i]
             d['Score'] = sc
             d['Score_std'] = sc_std
             d['MinScore'] = sc-sc_std
             grid_results_ = grid_results_.append(d,ignore_index=True)
         grid_results_.sort_values(by=['MinScore','Score','Score_std'],
                                   ascending=[False,False,True]).head(10)
```

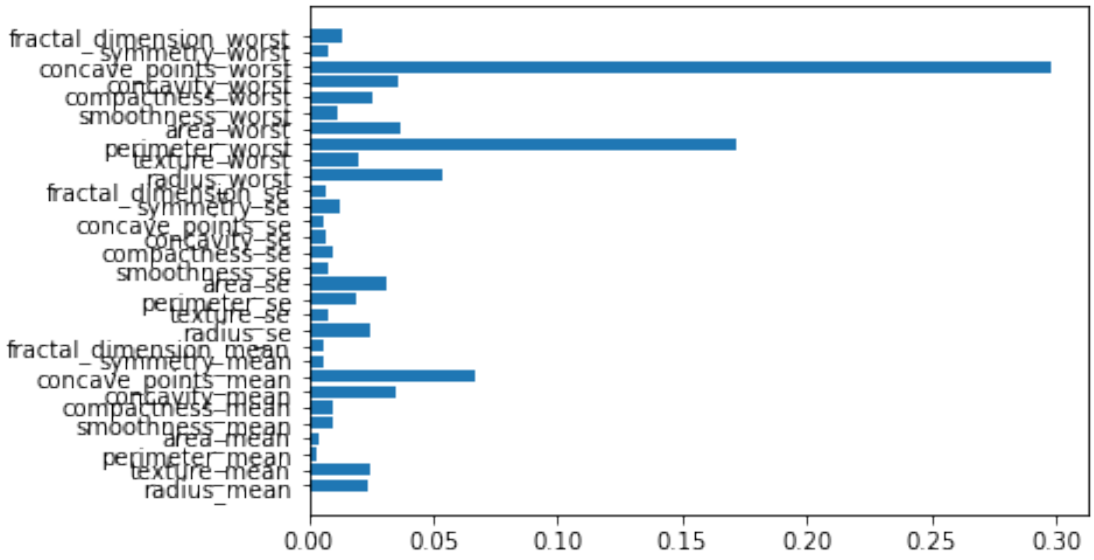
```
Out[46]:          Score  Score_std   MinScore  colsample_by_node  colsample_bytree  \
         21   95.070423   0.982959  94.087463               0.50               0.5
         24   95.070423   0.982959  94.087463               0.50               0.5
         45   95.070423   0.982959  94.087463               0.50               0.5
         48   95.070423   0.982959  94.087463               0.50               0.5
         51   95.070423   0.982959  94.087463               0.50               0.5
         264  95.070423   0.982959  94.087463               0.75               0.5
         267  95.070423   0.982959  94.087463               0.75               0.5
         288  95.070423   0.982959  94.087463               0.75               0.5
         291  95.070423   0.982959  94.087463               0.75               0.5
         294  95.070423   0.982959  94.087463               0.75               0.5
```

```
      gamma  learning_rate  max_depth  n_estimatros        objective  subsample
21     0.00            0.1        4.0         500.0  binary:logistic        0.5
24     0.00            0.1        5.0         500.0  binary:logistic        0.5
45     0.25            0.1        3.0         500.0  binary:logistic        0.5
48     0.25            0.1        4.0         500.0  binary:logistic        0.5
51     0.25            0.1        5.0         500.0  binary:logistic        0.5
264    0.00            0.1        4.0         500.0  binary:logistic        0.5
267    0.00            0.1        5.0         500.0  binary:logistic        0.5
288    0.25            0.1        3.0         500.0  binary:logistic        0.5
291    0.25            0.1        4.0         500.0  binary:logistic        0.5
294    0.25            0.1        5.0         500.0  binary:logistic        0.5
```

```
In [47]: xg_class = xgb.XGBClassifier(gamma=0,
                                      n_estimators=10000,
                                      objective='binary:logistic',
                                      subsample=0.5,
                                      learning_rate=0.1,
                                      max_depth=4,
                                      colsample_by_node=0.5,
                                      colsample_by_tree=0.5)
         xg_class.fit(Xtrain,Ytrain)

Out[47]: XGBClassifier(base_score=0.5, booster='gbtree', colsample_by_node=0.5,
                colsample_by_tree=0.5, colsample_bylevel=1, colsample_bynode=1,
                colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
                max_depth=4, min_child_weight=1, missing=None, n_estimators=10000,
                n_jobs=1, nthread=None, objective='binary:logistic', random_state=0,
                reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
                silent=None, subsample=0.5, verbosity=1)

In [48]: Ypred = xg_class.predict(Xtest)
         100*accuracy_score(Ytest,Ypred)

Out[48]: 95.78947368421052

In [49]: confusion_matrix(Ytest,Ypred)

Out[49]: array([[178,   6],
                [  6,  95]], dtype=int64)
```

Nous avons perdu un peu de précision mais nous avons vraiment réduit le nombre de tumeurs faussement prédites comme étant bénignes.

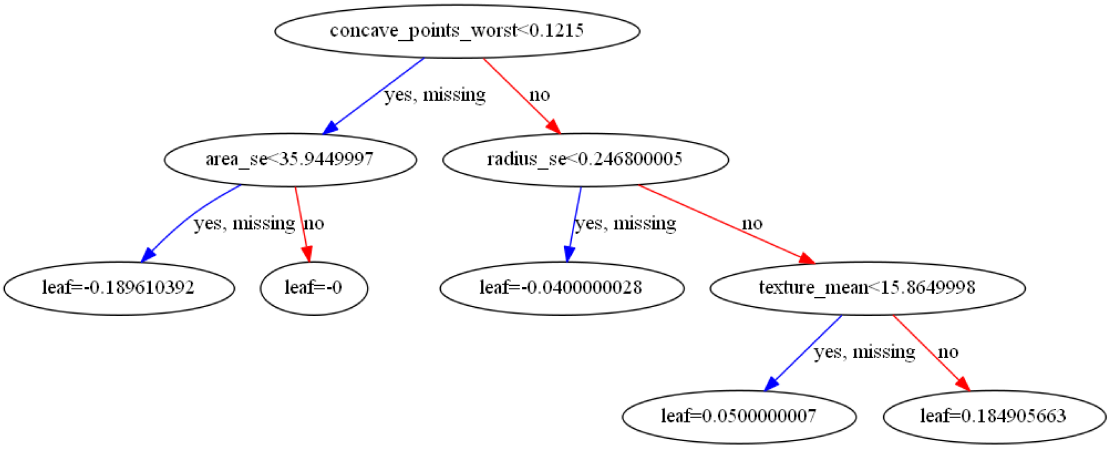
```
In [50]: use = xg_class.feature_importances_
         feat = list(Xtrain.columns)
         p = plt.barh(feat,use)
```



On voit bien l'évolution du choix des variables entre les algorithmes précédents et celui-ci. En effet, ce dernier utilise presque toutes les variables mais n'en utilise que très peu à forte dose.

On peut également dessiner un des arbres de décision obtenu (ici le premier).

```
In [53]: from matplotlib.pylab import rcParams
         rcParams['figure.figsize'] = 80,50
         p = xgb.plot_tree(xg_class)
```



Pour conclure, nous voyons que l'implémentation en Python des méthodes de forêts aléatoires sont assez simples. La difficulté réside dans le choix de hyper-paramètres du problème qui vont jouer un gros rôle dans la qualité des prédictions. De plus, la manière dont on utiliser les packages en Python est très similaire à celle que l'on connaît dans R. Les packages sont donc très faciles à prendre en main.