1 Introduction

Etude des fonctionnalités d'analyse prédictive de la librairie ''mlxtend'' pour Python.

Tout comme R, la popularité du langage Python en machine learning repose essentiellement sur la richesse des packages spécialisés qu'il propose. Mais, contrairement à R où nous disposons d'une multitude de librairies, le paysage de Python est dominé par quelques librairies, essentiellement ''scikit-learn'' (un de mes tutoriels les plus populaires de ces dernières années) et le tandem ''tensorflow / keras''[1]. C'est bien parce que ce sont là des packages de qualité qui couvrent des larges domaines d'applications, avec des modes opératoires cohérents qui nous facilitent grandement la vie. Mais ça peut être désavantageux parce qu'ils nous imposent une vision monolithique, voire restrictive, de la pratique de la data science, qui peut, parfois, biaiser notre perception des algorithmes ou des procédures de machine learning. Par exemple, de par les algorithmes internes utilisés, la régression logistique sous ''scikit-learn'' ne fournit pas les variances des coefficients, nous privant des calculs d'intervalles de confiance des coefficients estimés ou des tests de significativité.

C'est donc avec beaucoup d'intérêt que j'ai exploré le package ''mlxtend'' (**machine learning extensions**) de Sebastian Raschka. Je l'avais découvert initialement en cherchant des outils pour l'extraction des règles d'association sous Python. J'avais noté en lisant la documentation qu'il proposait des fonctionnalités assez intéressantes pour l'évaluation des modèles (cf. User guide / evaluate), que l'on retrouve peu dans les autres bibliothèques. De fil en aiguille, j'ai identifié d'autres procédures que je trouve assez judicieuses. Et surtout, plutôt que de s'opposer aux mastodontes déjà bien en place, il propose des outils qui s'interfacent avec ceux de scikit-learn par exemple, tirant parti de la puissance de ce dernier.

Dans ce tutoriel, via une trame d'analyse prédictive assez standard, nous essaierons de mettre en lumière l'intérêt et la pertinence des différentes fonctions que propose la libraire ''mlxtend''.

2 Importation et préparation des données

2.1 Chargement des échantillons d'apprentissage et de test

Nous utilisons les données ''spam'' accessible sur le serveur UCI Machine Learning Repository. L'objectif est d'identifier les courriels frauduleux (spam = yes) à partir de leurs caractéristiques (fréquences des mots, de certains caractères, nombre de caractères en majuscules). Nous avons

[1] Voir ''Python eats away at R: Top Software for Analytics, Data Science, Machine Learning in 2018: Trends and Analysis'', KDnuggets polls, Mai 2018 ; ''Top 8 Python Machine Learning Libraries'', KDnuggets Opinions, Octobre 2018.

au préalable scindé les données (**spam_mlxtend.xlsx**) en échantillons d'apprentissage (1601 observations, feuille ''spam_train'') et de test (3000, ''spam_test'').

Nous utilisons la version 3.7.1 de l'interpréteur Python.

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

```
3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)]
```

Nous chargeons les données d'apprentissage (TRAIN) à l'aide de la librairie ''pandas''. Nous disposons de 1601 observations et 56 variables, dont la cible ''spam''.

```
#changement de dossier
import os
os.chdir("... votre dossier ...")

#train
import pandas
DTrain = pandas.read_excel("spam_mlxtend.xlsx",sheet_name="spam_train")
print(DTrain.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1601 entries, 0 to 1600
Data columns (total 56 columns):
spam                         1601 non-null object
wf_make                      1601 non-null float64
wf_address                   1601 non-null float64
wf_all                       1601 non-null float64
wf_3d                        1601 non-null float64
wf_our                       1601 non-null float64
wf_over                      1601 non-null float64
wf_remove                    1601 non-null float64
wf_internet                  1601 non-null float64
wf_order                     1601 non-null float64
wf_mail                      1601 non-null float64
wf_receive                   1601 non-null float64
wf_will                      1601 non-null float64
wf_people                    1601 non-null float64
wf_report                    1601 non-null float64
wf_addresses                 1601 non-null float64
wf_free                      1601 non-null float64
wf_business                  1601 non-null float64
wf_email                     1601 non-null float64
wf_you                       1601 non-null float64
wf_credit                    1601 non-null float64
wf_your                      1601 non-null float64
wf_font                      1601 non-null float64
```

```
wf_000                        1601 non-null float64
wf_money                      1601 non-null float64
wf_hp                         1601 non-null float64
wf_hpl                        1601 non-null float64
wf_lab                        1601 non-null float64
wf_labs                       1601 non-null float64
wf_telnet                     1601 non-null float64
wf_857                        1601 non-null float64
wf_data                       1601 non-null float64
wf_415                        1601 non-null float64
wf_85                         1601 non-null float64
wf_technology                 1601 non-null float64
wf_1999                       1601 non-null float64
wf_parts                      1601 non-null float64
wf_pm                         1601 non-null float64
wf_direct                     1601 non-null float64
wf_cs                         1601 non-null float64
wf_meeting                    1601 non-null float64
wf_original                   1601 non-null float64
wf_project                    1601 non-null float64
wf_re                         1601 non-null float64
wf_edu                        1601 non-null float64
wf_table                      1601 non-null float64
wf_conference                 1601 non-null float64
cf_comma                      1601 non-null float64
cf_bracket                    1601 non-null float64
cf_sqbracket                  1601 non-null float64
cf_exclam                     1601 non-null float64
cf_dollar                     1601 non-null float64
cf_hash                       1601 non-null float64
capital_run_length_average    1601 non-null float64
capital_run_length_longest    1601 non-null int64
capital_run_length_total      1601 non-null int64
dtypes: float64(53), int64(2), object(1)
```

Nous chargeons également l'échantillon test (TEST) qui comporte 3000 observations.

```
#test
DTest = pandas.read_excel("spam_mlxtend.xlsx",sheet_name="spam_test")
print(DTest.shape)
```

```
(3000, 56)
```

2.2 Standardisation des variables

Via l'outil StandardScaler de la librairie ''scikit-learn'', nous standardisons les variables des deux ensembles de données en utilisant les paramètres (moyennes et écarts-type) calculés sur l'échantillon d'apprentissage. Il est par conséquent tout à fait normal que les variables transformées soient de moyenne nulle sur cet échantillon.

```
#transformation échantillon apprentissage
from sklearn.preprocessing import StandardScaler
stds = StandardScaler()
ZTrain = stds.fit_transform(X=DTrain.iloc[:,1:])
print(ZTrain.mean(axis=0))
```

```
[ 5.79729200e-17 -3.32858871e-17 -1.99715322e-17  8.87623655e-18
 -5.32574193e-17  2.21905914e-17  3.55049462e-17 -3.99430645e-17
 -8.87623655e-18  1.24267312e-16 -4.54907123e-17  1.23157782e-16
  8.87623655e-18 -4.21621236e-17  0.00000000e+00  2.21905914e-17
 -2.21905914e-17  5.10383602e-17 -6.65717741e-18  8.87623655e-18
 -2.21905914e-18 -2.80156216e-17  5.76955376e-17 -8.87623655e-17
 -4.43811827e-17  1.10952957e-18  1.55334140e-17 -2.21905914e-18
  7.76670698e-18 -6.65717741e-18 -9.98576612e-18  1.77524731e-17
 -1.99715322e-17 -8.87623655e-18 -6.04693615e-17  1.27595900e-17
  2.21905914e-18  8.87623655e-18 -1.44238844e-17 -4.16073588e-18
  4.88193010e-17  8.87623655e-18 -6.87908333e-17 -4.66002419e-17
 -2.99572984e-17  2.44096505e-17  2.21905914e-18 -3.66144758e-17
  2.66287096e-17  1.10952957e-17 -4.21621236e-17 -1.33143548e-17
  0.00000000e+00 -2.21905914e-18 -4.43811827e-17]
```

Il en est autrement pour l'échantillon test (les moyennes ne sont pas forcément nulles), et c'est tout aussi normal.

```
#et de l'échantillon test
ZTest = stds.transform(DTest.iloc[:,1:])
print(ZTest.mean(axis=0))
```

```
[ 2.91419019e-02  7.94909734e-02 -6.41282121e-03 -1.93968625e-04
  1.20715961e-02  1.48926666e-02 -1.05063997e-04  8.26136430e-02
  9.28410600e-02 -1.29249511e-03  5.10442610e-02  4.22698086e-02
 -7.92599971e-03 -8.49463753e-03  2.50510412e-02  2.17845751e-02
  6.09789473e-02  6.67487265e-03 -5.77890335e-02  2.38698263e-03
  1.13702575e-03  4.58049242e-02 -1.73232931e-02 -1.39578932e-02
 -1.56260293e-02  9.56819702e-03 -1.35350021e-02  9.77625868e-03
 -1.76682191e-02  7.41698162e-03  1.16412100e-01  6.67954542e-03
 -6.00690826e-03 -2.06173744e-02  8.57802451e-03 -3.40521073e-02
  1.41689304e-03 -7.30577407e-03 -2.55715810e-02  1.95178832e-02
  1.45998018e-02  3.83523590e-02 -2.20532320e-02 -3.97918820e-02
  1.49723546e-02  3.32022961e-03 -1.34188669e-02 -8.40746584e-03
  1.28819222e-02 -2.15086961e-02 -1.31968107e-02 -8.91132218e-03
  2.04902094e-02  6.56607427e-02  5.33414950e-02]
```

Par commodité, nous isolons les vecteurs représentant les variables cibles dans des structures spécifiques.

```
#isoler les cibles par commodité
YTrain = DTrain.iloc[:,0]
YTest = DTest.iloc[:,0]
```

Nous en profitons pour comptabiliser l'occurrence des classes sur l'échantillon test. Cela nous sera utile lorsqu'il faudra inspecter les matrices de confusion et les indicateurs de performances que nous calculerons plus loin.

```
#distribution de la cible en test
print(YTest.value_counts())
```

```
no     1821
yes    1179
Name: spam, dtype: int64
```

3 Apprentissage et évaluation des modèles

3.1 Le schéma ''holdout''

Le schéma ''holdout'' est le plus connu de la démarche d'élaboration et d'évaluation des modèles. Nous construisons le modèle sur un échantillon dédié, dit d'apprentissage, 1601 observations pour nous ; puis nous en évaluons les performances sur un échantillon distinct, dit de test, 3000 observations dans notre cas. Le fait d'utiliser des échantillons disjoints permet d'obtenir une estimation non-biaisée de la qualité du modèle.

Apprentissage. Nous utilisons un SVM (support vector machine) avec un noyau RBF (radial basis function) de la librairie ''scikit-learn'' (SVC). Nous veillons à ce que les probabilités d'affectation soient calculées en prédiction (probability = True), elles nous seront utiles lorsque nous travaillerons sur la combinaison de modèles plus bas (section 5).

```
#modélisation avec un SVM Rbf
from sklearn.svm import SVC
m_svm = SVC(gamma='scale',probability=True)
m_svm.fit(ZTrain,YTrain)
```

Prédiction et matrice de confusion. Pour évaluer les performances, nous calculons les prédictions sur l'échantillon test. Nous comptabilisons le nombre de prédictions 'no' et 'yes'.

```
#prédiction en test
p_svm = m_svm.predict(ZTest)

#distribution
import numpy
print(numpy.unique(p_svm,return_counts=True))
```

```
(array(['no', 'yes'], dtype=object), array([1912, 1088], dtype=int64))
```

Nous utilisons les outils de ''mlxtend'' à partir d'ici. Nous vérifions tout d'abord le numéro de la version utilisée (précaution que l'on devrait prendre systématiquement lorsqu'on utilise un package sous Python ou sous R).

```
#version de mlxtend
import mlxtend
print(mlxtend.__version__)
```

```
0.13.0
```

Puis, nous calculons la matrice de confusion en confrontant les classes observées et prédites.

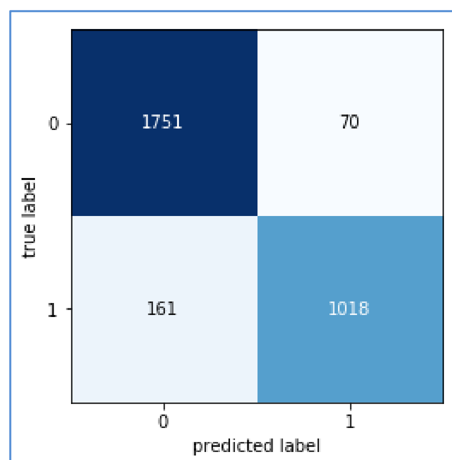
```
#matrice de confusion
from mlxtend.evaluate import confusion_matrix
cm_svm = confusion_matrix(YTest,p_svm)
print(cm_svm)
```

```
[[1751   70]
 [ 161 1018]]
```

Les sommes en ligne fournissent bien les classes observées ('no', 'yes') sur l'échantillon test (1821, 1179) ; les sommes en colonne correspondent aux classes prédites (1912, 1088).

Petite coquetterie, ''mlxtend'' propose une représentation graphique de la matrice de confusion.

```
#plotter la matrice de confusion
from mlxtend.plotting import plot_confusion_matrix
plot_confusion_matrix(conf_mat=cm_svm)
```



Concernant les codes utilisés : '0' correspond à la première modalité 'no', par ordre alphabétique, de la variable cible ''spam'' ; '1' représente 'yes'.

Indicateurs de performances. ''mlxtend'' propose plusieurs indicateurs de performances issus de la matrice de confusion, entres autres : le taux de reconnaissance (taux de succès), le taux d'erreur, le rappel (il faut indiquer la modalité positive 'yes'), la précision et le F1-Score.

```
#outil estimation des différents indicateurs
from mlxtend.evaluate import scoring

#taux de reconnaissance
```

```
print(scoring(YTest,p_svm,metric='accuracy')) # 0.923

#taux d'erreur
print(scoring(YTest,p_svm,metric='error')) # 0.077

#rappel
print(scoring(YTest,p_svm,metric='recall',positive_label="yes")) # 0.863

#precision
print(scoring(YTest,p_svm,metric='precision',positive_label="yes")) # 0.936

#F1-Score
print(scoring(YTest,p_svm,metric='f1',positive_label="yes")) # 0.898
```

3.2 Estimation par rééchantillonnage - 0.632 Bootstrap

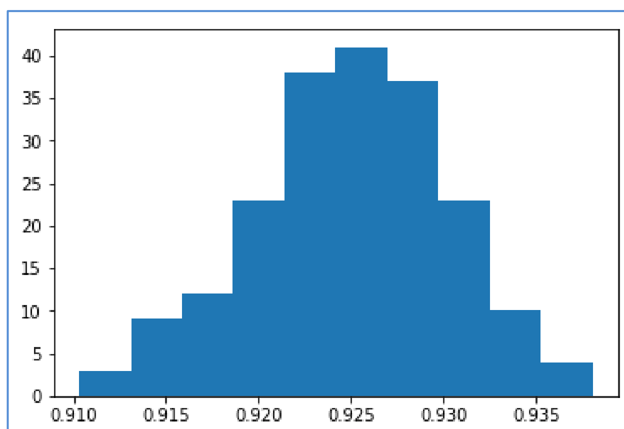
La stratégie ''holdout'' est difficilement tenable lorsque nous disposons d'une base avec un effectif réduit. Il nous faut utiliser la totalité des données disponibles pour élaborer le modèle afin de ne pas le pénaliser, puis passer par des techniques de rééchantillonnage pour en estimer les performances. La validation croisée est la méthode la plus souvent citée, mais d'autres approches existent, notamment le bootstrap, peu présente dans les librairies de machine learning, on se demande pourquoi parce qu'elle a largement fait ses preuves.

''mlxtend'' le propose justement, en particulier la variante 0.632 bootstrap, plus efficace. Après avoir construit le modèle sur les 1601 observations de TRAIN, **nous en évaluons les performances sur les mêmes données via un bootstrap**. Nous verrons si nous nous rapprochons ou pas de la mesure effectuée sur l'échantillon test à part (et ses 3000 observations).

```
#estimation 0.632 bootstrap de l'erreur
import numpy
from mlxtend.evaluate import bootstrap_point632_score
accuracies = bootstrap_point632_score(m_svm,ZTrain,YTrain,method='.632',n_splits=200)

#on dispose de 200 valeurs de l'accuracy
print(accuracies.shape) # (200,)
```

Nous demandons 200 itérations, nous disposions de 200 valeurs du taux de succès. Voici l'histogramme des fréquences des valeurs.

```
#histogramme
import matplotlib.pyplot as plt
plt.hist(accuracies)
```



La médiane peut servir d'indicateur synthétique.

```
#mediane des valeurs
print(numpy.median(accuracies)) # 0.925
```

Nous sommes assez proche de la valeur obtenue sur l'échantillon test (0.923).

Et puisque nous disposons d'une série de valeurs, nous pouvons en déduire l'intervalle de confiance empirique au niveau de confiance 90%.

```
#intervalle de confiance empirique
print('90%% CI : [%.2f,%.2f]' % (100*numpy.percentile(accuracies,5.0),100*numpy.percentile(accuracies,95.0)))
```

```
90% CI : [91.54,93.34]
```

Nous constatons, au moins pour cet exemple, que l'utilisation d'une technique de rééchantillonnage, mettant en œuvre le seul échantillon d'apprentissage, est une alternative tout à fait crédible à l'utilisation d'un échantillon test, dont la disponibilité est parfois hypothétique.

4 Comparaison de modèles

Dans cette section nous nous intéressons à la comparaison de modèles. Deux algorithmes sont appliqués au même jeu de données, lequel est le plus performant ? Nous constaterons qu'il y a plusieurs manières de répondre à la question.

4.1 Régression logistique

Nous opposerons la régression logistique de ''scikit-learn'' au SVM ci-dessus. Nous construisons le modèle, nous calculons les prédictions en test, nous en déduisons le taux de reconnaissance.

```
#un second modèle : régression logistique
from sklearn.linear_model import LogisticRegression
m_lr = LogisticRegression(solver='lbfgs')
m_lr.fit(ZTrain,YTrain)
```

```
#son taux de reconnaissance
p_lr = m_lr.predict(ZTest)
print(scoring(YTest,p_lr,metric='accuracy'))
```

```
0.9176666666666666
```

Puisque nous utilisons exactement le même échantillon test, 0.918 est comparable à 0.923, il semble moins bon. Mais doit-on tirer des conclusions définitives sachant que les performances ont été calculées à partir d'un échantillon, soumis aux variations d'échantillonnage ?

4.2 Comparaison sur échantillon test : test de McNemar

Le test de McNemar permet de comparer directement les performances prédictives de 2 modèles évalués sur le même échantillon test. La statistique de test est calculée à partir d'un tableau de contingence opposant les bonnes et mauvaises affectations des modèles.

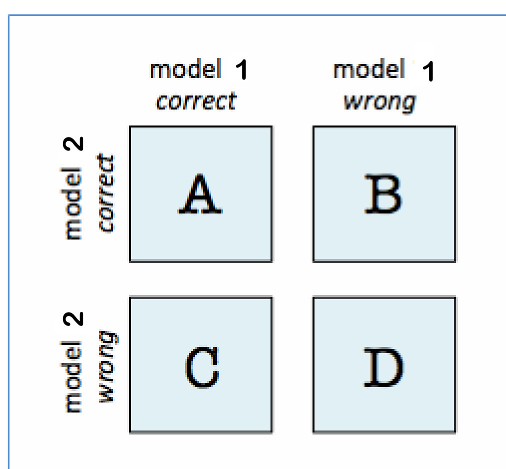


Figure 1 - Tableau pour la statistique de McNemar (le tableau en ligne est erroné)

Si B est significativement plus petit que C, le modèle 1 est meilleur que le modèle 2, puisqu'il classe plus souvent à bon escient. Et inversement.

Voyons ce qu'il en est en ce qui concerne nos deux classifieurs. L'outil mcnemar_table prend en entrée les classes observées et les prédictions des deux modèles.

```
#comparer les performances - Test de McNemar
#tableau de comparaison
from mlxtend.evaluate import mcnemar_table
tb = mcnemar_table(y_target=YTest,y_model1=p_svm,y_model2=p_lr)
print(tb)
```

```
[[2681   72]
 [  88  159]]
```

Il y a bien (72 + 159 = 231) mauvaises prédictions pour le 1er modèle (cf. la matrice de confusion du premier modèle SVM ci-dessus, page 6).

Les deux modèles concordent souvent à juste titre (A = 2681, sur 3000 individus), elles se trompent moins souvent de concert (D = 159). Et en cas de désaccord, le modèle 1 (SVM) semble légèrement moins meilleur, pas de manière flagrante (B = 72 vs. C = 88). Voyons si cet écart est significatif avec le test de McNemar. ''mlxtend'' produit la statistique de test et la p-value.

```
#statistique de test et conclusion
from mlxtend.evaluate import mcnemar
chi2, pval_mcnemar = mcnemar(ary=tb, corrected=False)
print('chi-squared %.5f' % chi2)
print('p-value: %.5f' % pval_mcnemar)
```

```
chi-squared 1.60000
p-value: 0.20590
```

Au risque 5%, les différences de performances ne sont pas significatives.

4.3 Comparaison par rééchantillonnage

De nouveau comment faire si nous ne disposons pas d'un échantillon test ?

Nous passons encore une fois par une technique de rééchantillonnage, la validation croisée en l'occurrence. Pour les deux modèles à confronter, la fonction ''K-fold cross-validated paired t test'' oppose l'indicateur de performance dans chaque fold, pour lesquels nous avons exactement les mêmes observations. Il s'agit bien d'une comparaison sur échantillons appariés.

```
#comparaison de performances par resampling
from mlxtend.evaluate import paired_ttest_kfold_cv
tcv, pval_cv = paired_ttest_kfold_cv(estimator1=m_svm,estimator2=m_lr,cv=30,scoring='accuracy',X=ZTrain,y=YTrain,random_seed=1)
print('t: %.5f' % tcv)
print('p-value: %.5f' % pval_cv)
```

```
t: 0.43766
p-value: 0.66487
```

Ici également, nous concluons que les performances ne sont pas significativement différentes.

5 Combinaison de modèles

La combinaison des modèles consiste à faire coopérer un pool de modèles élaborés sur un même et seul échantillon d'apprentissage, en espérant qu'elles se complètent en prédiction. ''mlxtend'' propose des outils pour différentes stratégies de votes. Voyons ce qu'il en est.

5.1 Combinaison par vote simple ou pondéré

Le première approche consiste à faire voter les modèles du pool, soit via un vote simple, soit en les pondérant, charge à nous de définir les valeurs appropriées des poids. La combinaison peut

reposer sur les classes prédites (voting = 'hard') ou sur les probabilités d'affectation (voting = 'soft'). L'expérience (les expérimentations) montre que cette seconde approche est souvent plus efficace.

Nous associons notre SVM (pour pouvoir réaliser un vote 'soft' il fallait bien demander le calcul des probabilités d'affectation, cf. section 3.1) et notre régression logistique (clfs=[m_svm,m_lr]). Nous leur attribuons un poids identique (weights = [1.0,1.0]). De nouveau, nous enchaînons les phases d'instanciation, d'apprentissage (elle n'était pas nécessaire ici en réalité puisque les classifieurs ont déjà été entraînés), de prédiction en test et de calcul de l'indicateur de performance.

```
#combinaison de classifieurs - Vote simple
from mlxtend.classifier import EnsembleVoteClassifier
m_ev = EnsembleVoteClassifier(clfs=[m_svm,m_lr], weights=[1.0,1.0], voting='soft')

#nécessité de réapprendre (on peut indiquer une option pour éviter cette phase)
m_ev.fit(ZTrain,YTrain)

#prédiction
p_ev = m_ev.predict(ZTest)

#performances
print(scoring(YTest,p_ev,metric='accuracy')) # 0.931
```

Le taux de reconnaissance est de 0.931, très légèrement meilleur (prudence, il faut faire un test pour pouvoir l'affirmer vraiment, cf. section 4.2) que les deux modèles pris individuellement, avec respectivement 0.923 (SVM) et 0.918 (régression logistique).

5.2 Stacking

Le stacking est une stratégie de combinaison de modèles où les pondérations (si on utilise la régression logistique comme métaclassifieur) sont apprises à partir des données. On s'appuie sur la validation croisée pour éviter le surapprentissage lors de la détermination des poids.

Avec ''mlxtend'', nous devons tout d'abord instancier le métaclassifieur.

```
#définir un meta-classifieur
meta_clf = LogisticRegression(solver='lbfgs')
```

Puis faire appel à l'outil StackingCVClassifier. Nous demandons une validation croisée en (cv = 5) blocs pour l'apprentissage des poids, et nous exploitons toujours les probabilités d'affectation (use_probas=True), plus précises que les classes prédites.

```
#combinaison de classifieurs - Stacking
from mlxtend.classifier import StackingCVClassifier
```

```
m_stacking = StackingCVClassifier(classifiers=[m_svm,m_lr],use_probas=True,meta_classifier=meta_clf,cv=5)

#apprentissage
m_stacking.fit(ZTrain,YTrain)

#prédiction
p_stacking = m_stacking.predict(ZTest)

#performances
print(scoring(YTest,p_stacking,metric='accuracy')) # 0.933
```

Le taux de reconnaissance est de 0.933, le meilleur résultat que nous ayons obtenu jusqu'à présent.

5.3 Comparaison de performances

Pour savoir s'il y a une différence notable est les performances des modèles, nous utilisons le test de Cochran, qui est une généralisation du test de McNemar à la comparaison de plus de 2 modèles.

```
#comparer les 4 modèles - Test de Cochran
from mlxtend.evaluate import cochrans_q
q, pval_cochran = cochrans_q(numpy.asarray(YTest),p_svm,p_lr,p_ev,p_stacking)
print('Q: %.5f' % q)
print('p-value: %.5f' % pval_cochran)
```

```
Q: 34.11066
p-value: 0.00000
```

Ah, il semble qu'il y ait une différence significative entre les modèles, l'un d'entre eux au moins se démarque des autres.

Pour confirmer cela, nous opposons le pire modèle (m_lr, la régression logistique, avec un taux de reconnaissance égal à 0.918) au meilleur (m_stacking avec 0.933) dans un test de McNemar (Remarque : un test post-hoc suggéré par les résultats n'est pas une très bonne idée généralement, il s'agit avant tout de montrer les capacités des outils de ''mlxtend'' dans ce tutoriel).

La table de comparaison nous indique…

```
#comparons le plus faible et le plus fort - sorte de test post hoc
#table
tbb = mcnemar_table(y_target=YTest,y_model1=p_lr,y_model2=p_stacking)
print(tbb)
```

```
[[2735   65]
 [  18  182]]
```

… que le modèle stacking (modèle 2) classe correctement (65 – 18 = 47) individus supplémentaires par rapport à la régression logistique (modèle 1). La différence est-elle significative ?

```
#test stat
chi2b, pval_mcnemarb = mcnemar(ary=tbb, corrected=False)
print('chi-squared %.5f' % chi2b)
print('p-value: %.5f' % pval_mcnemarb)
```

```
chi-squared 26.61446
p-value: 0.00000
```

Oui, apparemment.

6 Evaluation et sélection des variables

6.1 Feature importance permutation

Identifier d'emblée les variables importantes n'est pas facile pour certains classifieurs ''boîtes noires''. C'est le cas par exemple pour un SVM avec un noyau non linéaire. L'affaire devient inextricable lorsque nous souhaitons traiter un modèle combiné (m_stacking) qui en comporte (des classifieurs boîtes noires). Notre salut passe par les méthodes de monte-carlo, agnostiques (applicable à tout type de modèle). J'avais justement étudié récemment une approche (''Importance des variables dans les modèles'', février 2019) que je m'étais amusé à programmer sous R. La bibliothèque ''mlxtend'' la propose. Essayons de voir ce que cela donne sur notre modèle combiné m_stacking.

```
#importance des variables dans le stacking
from mlxtend.evaluate import feature_importance_permutation
fimp_mean,fimp_all = feature_importance_permutation(predict_method=m_stacking.predict,X=ZTest,y=YTest,metric='accuracy',num_rounds=20,seed=1)

#valeurs moyennes pour chaque variable
print(fimp_mean)
```

Pour rappel, l'approche consiste à appliquer plusieurs fois (num_rounds=20) le modèle (predict_method=m_stacking) sur des données judicieusement perturbées afin de mesurer l'influence de chaque variable sur le taux de reconnaissance (metric='accuracy'). Il n'est pas question de réapprendre le modèle. De fait, 20 répétitions du processus pour 55 variables n'est pas excessif en temps de calcul.

Nous affichons les valeurs moyennes pour chaque variable.

```
[0.00366667 0.0046     0.00105    0.00148333 0.00866667 0.00226667
 0.02135    0.0015     0.00483333 0.00066667 0.00316667 0.00295
 0.00078333 0.00021667 0.00246667 0.02046667 0.0119     0.00138333
 0.0014     0.00783333 0.00603333 0.00505    0.01545    0.00548333
 0.02361667 0.0059     0.0039     0.00075    0.00281667 0.002
 0.00396667 0.00306667 0.00563333 0.00401667 0.00371667 0.00075
 0.00183333 0.00341667 0.00206667 0.00641667 0.00128333 0.00291667
 0.00428333 0.00523333 0.00106667 0.0021     0.00403333 0.00233333
 0.00153333 0.01623333 0.017      0.00046667 0.00346667 0.01363333
 0.00433333]
```

Distinguer quelque chose là-dedans n'est pas facile. Nous organisons l'affichage de manière à mettre en évidence les 10 variables les plus influentes.

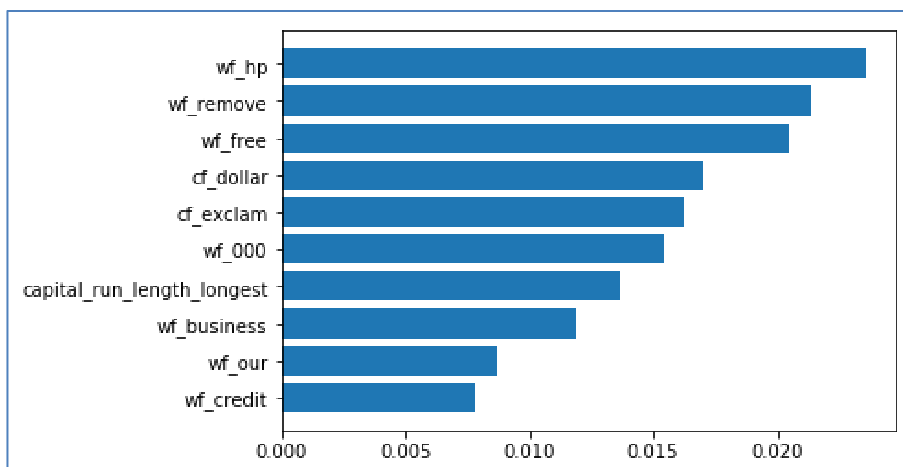
```
#index des valeurs triées de manière décroissante
index = numpy.argsort(fimp_mean)[::-1]

#tableau importance des variables
fimp = pandas.DataFrame({'variable':DTrain.columns[1:][index],'importance':fimp_mean[index]})

#affichage des 10 premiers
print(fimp.head(10))
```

```
                     variable  importance
0                       wf_hp    0.023617
1                   wf_remove    0.021350
2                     wf_free    0.020467
3                   cf_dollar    0.017000
4                   cf_exclam    0.016233
5                      wf_000    0.015450
6  capital_run_length_longest    0.013633
7                 wf_business    0.011900
8                      wf_our    0.008667
9                   wf_credit    0.007833
```

Les variables les plus influentes sont wf_hp (fréquence du mot 'hp'), wf_remove, etc. Ici commence une seconde étape d'interprétation et de compréhension du mécanisme d'affectation sous-jacent au modèle…. que nous n'aborderons pas dans ce tutoriel.

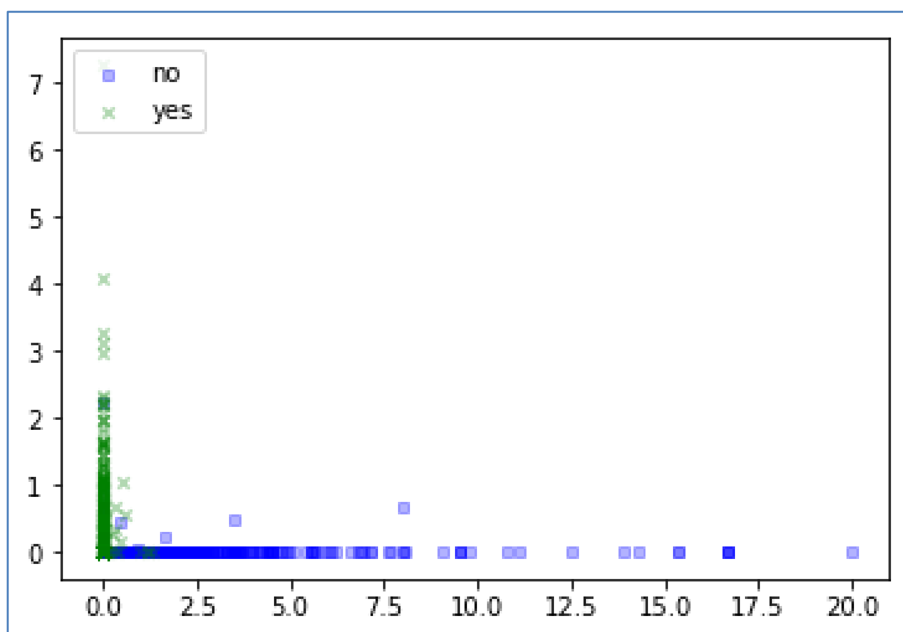
Nous pouvons également opter pour un affichage graphique, plus sympathique.

```
#affichage graphique
plt.barh(DTrain.columns[1:][index][:10][::-1],fimp_mean[index][:10][::-1])
```



Si l'on s'en tient aux deux premières variables, nous pouvons afficher les points dans le plan pour voir un peu ce qu'il en est de la séparabilité des classes.

```
#représentation des points dans l'espace des 2 variables les plus discriminantes
from mlxtend.plotting import category_scatter
category_scatter(x='wf_hp', y='wf_remove', label_col='spam', data=DTrain,
legend_loc='upper left',markersize=20,alpha=0.3)
```



Les deux variables sont (quasi)orthogonales. La discrimination se fait au niveau des faibles valeurs.

6.2 Sélection ''wrapper''

Sélection de variables – ''Wrapper'' en validation croisée. L'autre tâche clé concernant les variables prédictives consiste à sélectionner les plus pertinentes pour la prédiction. ''mlxtend'' propose une approche agnostique basé sur la recherche de la performance pure en validation

croisée sur l'échantillon d'apprentissage. L'outil Sequential Feature Selector procède de manière ascendante (forward) ou descendante (backward) pour identifier le meilleur sous-ensemble de descripteurs. Nous étudierons la première variante dans cette section. L'algorithme recherche variable qui maximise un critère d'évaluation en validation croisée. Puis, dans la seconde étape, il recherche la variable additionnelle qui améliore le mieux notre critère. Etc. Il s'arrête lorsque le nombre maximal de variables à sélectionner est atteint. En y regardant de plus près, il s'agit tout simplement de l'approche ''wrapper'' (Kohavi et John, 1997).

Mon idée initiale était d'appliquer la méthode au modèle combiné m_stacking. Mais la volumétrie des calculs m'en a dissuadé. Le nombre de validation croisée à effectuer, pour construire chaque modèle combiné, pour évaluer l'apport de chaque variable, est tout simplement prohibitif.

Finalement, je me suis rabattu sur la sélection pour le SVM avec un noyau RBF.

```
#selection de variables - approche wrapper sur validation croisée - 20 variables max
from mlxtend.feature_selection import SequentialFeatureSelector as SFS
sfs = SFS(m_lr,k_features=20,forward=True,scoring='accuracy',cv=5,verbose=True)

#recherche des meilleures variables
sfs.fit(ZTrain,numpy.asarray(YTrain=='yes',dtype='int'),custom_feature_names=DTrain.columns[1:])
```

Le nombre maximal de variables à sélectionner est (k_features = 20), le taux de reconnaissance sert d'indicateur de performances (scoring = 'accuracy'), nous procédons à une validation croisée à (cv = 5) folds. Ayant activé le mode ''bavard'' (verbose = True), nous disposons du détail des étapes.

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  55 out of  55 | elapsed:    0.8s finished
Features: 1/20[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  54 out of  54 | elapsed:    0.9s finished
Features: 2/20[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  53 out of  53 | elapsed:    1.0s finished
...
 [Parallel(n_jobs=1)]: Done  37 out of  37 | elapsed:    2.1s finished
Features: 19/20[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  36 out of  36 | elapsed:    2.0s finished
```

Ainsi, la première variable sélectionnée est 'cf_dollar' :

```
#1ere etape
sfs.subsets_[1]
{'feature_idx': (50,),
 'cv_scores': array([0.78504673, 0.76635514, 0.796875  , 0.784375  , 0.76175549]),
```

```
 'avg_score': 0.778881471010459,
 'feature_names': ('cf_dollar',)}
```

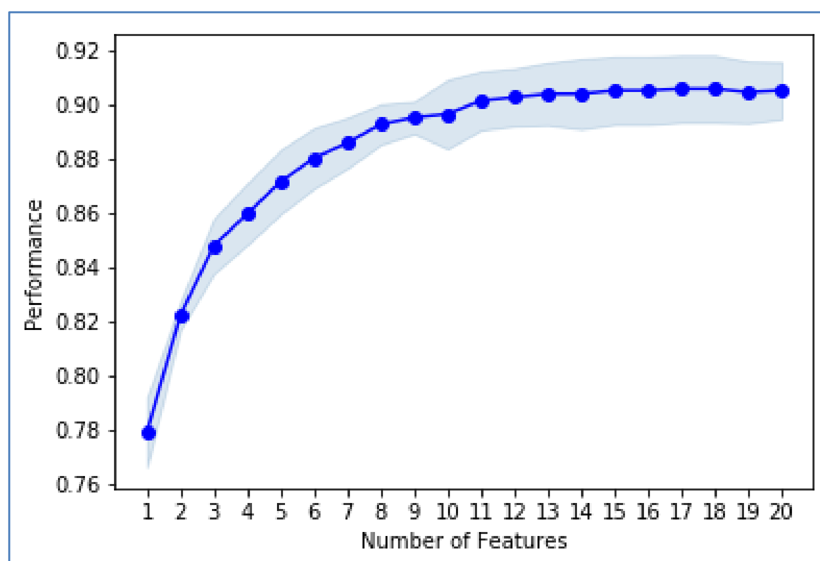
A la dernière étape, les 20 variables intégrées dans le modèle sont :

```
#20e étape
sfs.subsets_[20]
```

```
{'feature_idx': (0,
  3,
  6,
  8,
  13,
  15,
  22,
  24,
  27,
  28,
  29,
  31,
  36,
  38,
  42,
  43,
  44,
  49,
  50,
  53),
 'cv_scores': array([0.90031153, 0.92211838, 0.896875  , 0.89375   , 0.91222571]),
 'avg_score': 0.905056122374242,
 'feature_names': ('wf_make',
  'wf_3d',
  'wf_remove',
  'wf_order',
  'wf_report',
  'wf_free',
  'wf_000',
  'wf_hp',
  'wf_labs',
  'wf_telnet',
  'wf_857',
  'wf_415',
  'wf_pm',
  'wf_cs',
  'wf_re',
  'wf_edu',
  'wf_table',
  'cf_exclam',
  'cf_dollar',
  'capital_run_length_longest')}
```

Identification du nombre ''optimal'' de variables. ''mlxtend'' propose un outil graphique particulièrement intéressant pour suivre l'évolution du critère au fil des itérations. Il nous permet notamment de savoir à partir de quelle étape l'adjonction d'une variable n'est plus profitable au modèle. Nous devrions nous arrêter à ce niveau en vertu du principe de parcimonie.

```
#affichage graphique
from mlxtend.plotting import plot_sequential_feature_selection as plot_sfs
plot_sfs(sfs.get_metric_dict(), kind='std_dev')
```



Il est possible de définir une sorte de ''bande de confiance'' des valeurs à partir de l'écart-type empirique obtenu lors de la validation croisée (kind = 'std_dev'). Bon, les couleurs sont jolies certes, mais l'intérêt me paraît très relatif.

En revanche, nous notons dans le graphique qu'au-delà de 11 variables (à peu près), le gain semble marginal. Ces descripteurs sont :

```
#etape 11
sfs.subsets_[11]
```

```
{'feature_idx': (6, 8, 15, 24, 28, 36, 42, 43, 49, 50, 53),
 'cv_scores': array([0.89719626, 0.92211838, 0.9       , 0.896875  , 0.89028213]),
 'avg_score': 0.901294354681198,
 'feature_names': ('wf_remove',
  'wf_order',
  'wf_free',
  'wf_hp',
  'wf_telnet',
  'wf_pm',
  'wf_re',
  'wf_edu',
  'cf_exclam',
  'cf_dollar',
```

```
  'capital_run_length_longest')}
```

Avec un taux de reconnaissance en validation croisée de **0.901**.

Performances en test. Dans ce qui suit, nous reconstruisons le modèle SVM sur ces 11 descripteurs pour en apprécier les performances en test.

Tout d'abord, nous récupérons les numéros des variables concernées.

```
#récupération des numéros des variables concernées
num_sel_var = list(sfs.subsets_[11]['feature_idx'])
print(num_sel_var)
```

```
[6, 8, 15, 24, 28, 36, 42, 43, 49, 50, 53]
```

Nous restreignons les matrices de données (apprentissage et test) sur ces variables.

```
#restriction des matrices de données
ZTrainSel = ZTrain[:,num_sel_var]
ZTestSel = ZTest[:,num_sel_var]
```

Et nous reproduisons le schéma ''holdout''.

```
#apprentissage
m_svmSel = SVC(gamma='scale',probability=True)
m_svmSel.fit(ZTrainSel,YTrain)

#prédiction en test
p_svmSel = m_svmSel.predict(ZTestSel)

#taux de reconnaissance
print(scoring(YTest,p_svmSel,metric='accuracy')) # 0.909
```

Par rapport au modèle SVM sur l'ensemble des variables (**0.923**), il y a un gap non négligeable. Finalement, ne sélectionner que 11 variables paraît trop restrictif, il faudrait creuser un peu plus. Je m'arrête là en ce qui me concerne…

7 Conclusion

L'ennui naquit un jour de l'uniformité. C'est très bien qu'il y ait des packages performants pour le machine learning pour Python (scikit-learn, keras, …), mais il ne faut pas qu'ils prennent toute la place et imposent une vision hégémonique de la pratique de la data science.

Tout ce qui peut contribuer à la diversité est une bonne chose. Et c'est avec beaucoup d'intérêt et de plaisir que j'ai exploré à l'occasion de ce tutoriel les fonctionnalités de la librairie ''mlxtend'' qui, plutôt que de s'opposer aux mastodontes déjà bien en place, propose des outils complémentaires plutôt pertinents dans différents domaines de l'analyse prédictive.

8 Références

''Mlxtend : machine learning extensions'', http://rasbt.github.io/mlxtend/