Atelier : détection des SMS : Spam / non Spam (Ham)

1 Objectif

Text mining / fouille de textes. Catégorisation de textes sous Python.

L'objectif de la catégorisation de textes est d'associer aussi précisément que possible des documents à des classes prédéfinies. Nous nous situons dans le cadre de l'apprentissage supervisé, avec une variable cible catégorielle, souvent binaire. La particularité réside dans la nature des observations qui sont des documents textuels. Mettre en œuvre directement les techniques prédictives n'est pas possible, il faut passer obligatoirement par une phase de préparation des données.

La représentation par sac de mots (*bag of words*) est souvent privilégiée pour ramener la description du corpus de textes en tableau individus (documents) – variables (termes) : on parle de matrice documents termes. Elle est accolée à la variable cible pour former l'ensemble de données. Elle ne fait que commencer en réalité car la matrice a pour singularité d'avoir une dimensionnalité élevée (plusieurs milliers de descripteurs) et d'être creuse (beaucoup de valeurs sont nulles). Certaines techniques de machine learning sont plus adaptées que d'autres. La réduction de la dimensionnalité notamment revêt une importance considérable, d'une part pour améliorer les performances, d'autre part pour rendre interprétables les résultats car, au final, au-delà de la prédiction pure, il s'agit aussi de comprendre la nature de la relation entre les documents et les classes.

Dans cet atelier, nous décrirons le processus de catégorisation de textes sous Python en exploitant essentiellement les capacités de text mining du package scikit-learn, lequel fournira également les méthodes de data mining (régression logistique). Nous traiterons des SMS à classer en messages « spam » (courrier indésirable, délictueux) ou « ham » (légitime).

2 Processus de catégorisation de textes

Une partie des observations est extraite aléatoirement, on parle d'échantillon d'apprentissage, elle est dévolue à la construction du modèle ; la partie restante, dite

échantillon test, est consacrée à la mesure des performances. La répartition 2/3 vs. 1/3 est souvent utilisée mais il n'y a pas de règle véritable en la matière.

Dans le cadre de la catégorisation de textes, cela se traduit par une partition du corpus **AVANT** la construction des matrices documents termes. Il ne faut pas que les observations du corpus de test puissent intervenir dans la constitution du dictionnaire (la liste des termes). On peut schématiser la démarche comme suit :

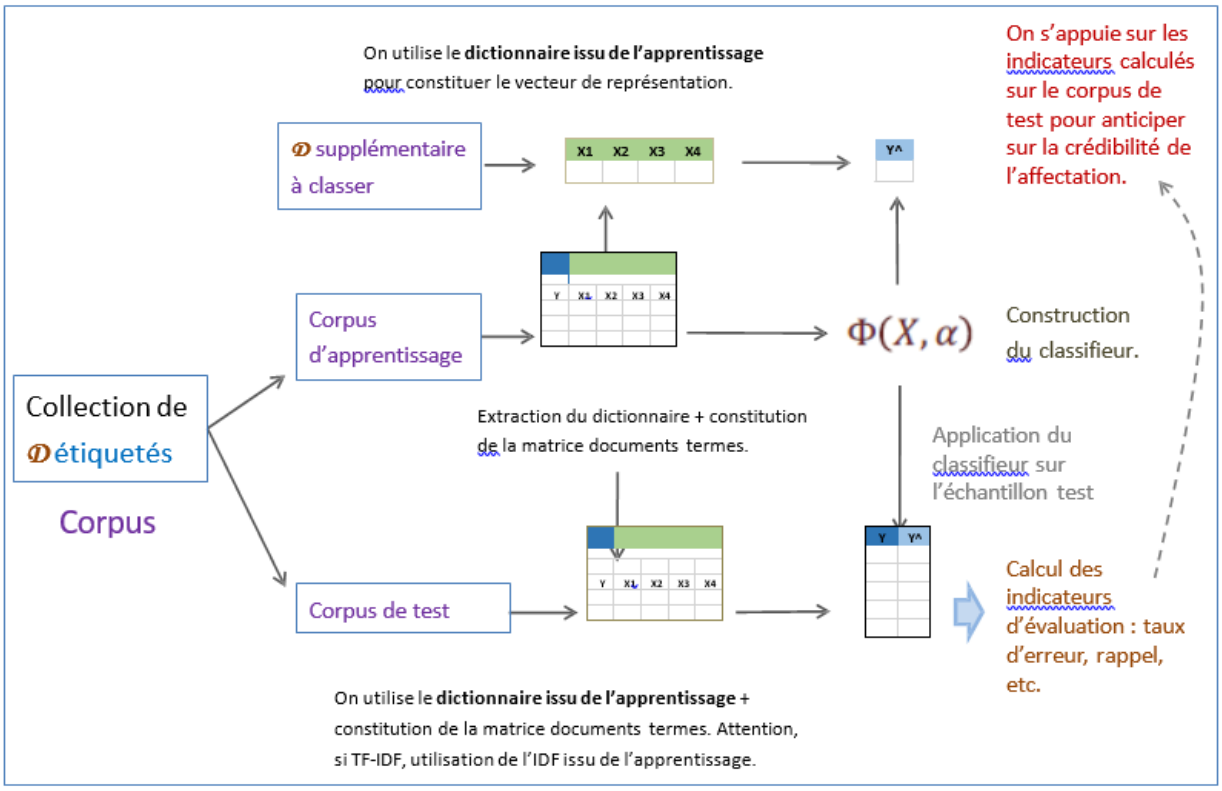

Figure 1 - Processus de catégorisation de textes

Cette contrainte – partition des corpus avant la construction des matrices documents termes – prend tout son sens lorsque nous utilisons le classifieur final en déploiement, lors du classement d'un document supplémentaire qui n'est pas disponible lors de la phase de modélisation. Il est évident qu'il ne doit intervenir en rien dans la construction du modèle : s'il introduit des termes inconnus dans la phase de modélisation, ils doivent être ignorés ; de même, nous ne connaissons pas le nombre de documents à classer lors du cycle de vie du modèle, le travail doit se baser exclusivement sur les informations issues de l'échantillon d'apprentissage.

Nous devons nous placer dans des conditions identiques dans les phases d'apprentissage et d'évaluation. La matrice document terme utilisée pour la modélisation doit provenir exclusivement du corpus d'apprentissage ; le dictionnaire, et les indicateurs qui en résultent,

seront ensuite exploités pour élaborer la matrice documents termes en test.

3 Identification des spam sous Python

3.1 Importation du corpus

Le fichier « **SMSSPamCollection.txt** » recense **n = 5572** messages, ventilés en 2 classes « spam » et « ham ». Les données se présentent comme suit :

```
classe message
ham    Go until jurong point, crazy.. Available only in bugis n great world
ham    Ok lar... Joking wif u oni...
spam   Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005
ham    U dun say so early hor... U c already then say...
```

La première ligne correspond aux noms des variables, les documents sont par la suite listés avec en première position la classe d'appartenance, en seconde le message. Le caractère tabulation "\t" fait office de séparateur de colonnes. Nous utilisons le package Pandas pour importer les données dans une structure de type

`<class 'pandas.core.frame.DataFrame'>`, nous disposons de `2 colonnes` et `5572 lignes`.

```
#numpy
import numpy as np

#change the current directory
import os
os.chdir("... your directory ...")

#importation of the corpus
import pandas
spams = pandas.read_table("SMSSpamCollection.txt",sep="\t",header=0)

#type of the object
print(type(spams))

#size of the dataset
print(spams.shape)
```

3.2 Description des données

Une petite inspection est toujours souhaitable lorsque l'on appréhende un jeu de données.

Nous énumérons les colonnes, leurs types et nous calculons les statistiques descriptives.

```
#list of columns
print(spams.columns)

#type of columns
print(spams.dtypes)

#description
print(spams.describe())
```

Les colonnes sont de type générique ''object''.

```
#list of columns
Index(['classe', 'message'], dtype='object')
#type of columns
classe     object
message    object
dtype: object
#description
       classe                 message
count    5572                    5572
unique      2                    5169
top       ham  Sorry, I'll call later
freq     4825  30
```

La classe la plus fréquente est `''ham''` avec 4825 observations ; nous observons que le texte `'' Sorry, I'll call later''` revient 30 fois dans les SMS échangés.

Nous pouvons calculer explicitement la distribution de fréquences des classes :

```
#frequency distribution of the class attribute
print(pandas.crosstab(index=spams["classe"],columns="count"))
```

Le dénombrement de ''ham'' est confirmé, nous disposons aussi de celui des ''spam''. Nous constatons que cette dernière est plus rare.

```
#frequency distribution of the class attribute
col_0   count
classe
ham      4825
spam      747
```

3.3 Partition en corpus d'apprentissage et de test

Nous créons les corpus d'apprentissage et de test avec, respectivement, ntrain = 3572 et ntest = (5572 – 3572) = 2000 documents, via une partition au hasard, stratifiée selon les classes pour préserver les proportions de ''spam'' et ''ham'' dans les sous-ensembles. Nous utilisons la procédure train_test_split du module sklearn.model_selection.

```
#subdivision into train and test sets
from sklearn.model_selection import train_test_split
spamsTrain, spamsTest = train_test_split(spams,train_size=3572,random_state=1,stratify=spams['classe'])

#frequency distribution of the class attribute
#train set
freqTrain = pandas.crosstab(index=spamsTrain["classe"],columns="count")
print(freqTrain/freqTrain.sum())
#test set
freqTest = pandas.crosstab(index=spamsTest["classe"],columns="count")
print(freqTest/freqTest.sum())
```

Les proportions sont bien respectées, la précision de l'évaluation n'en sera qu'améliorée.

```
#train set
col_0      count classe
ham     0.865901
spam    0.134099
#test set
col_0   count classe
ham     0.866
spam    0.134
```

3.4 Construction de la matrice documents termes pour l'apprentissage

Nous pouvons maintenant construire la matrice de documents termes pour le corpus d'apprentissage. Nous choisissons la pondération binaire. L'opération est réalisée en deux temps. (1) Nous instancions l'objet de calcul CountVectorizer. (2) Nous appelons la méthode fit_transform() en passant en paramètre les messages du corpus d'apprentissage spamsTrain.

```
#import the CountVectorizer tool
from sklearn.feature_extraction.text import CountVectorizer

#instantiation of the objet – binary weighting
parseur = CountVectorizer(binary=True)

#create the document term matrix
XTrain = parseur.fit_transform(spamsTrain['message'])
```

La méthode crée le dictionnaire et la matrice documents termes que nous affectons à la variable XTrain.

Nous pouvons afficher le nombre et la liste des termes qui composent le dictionnaire.

```
#number of tokens
print(len(parseur.get_feature_names()))

#list of tokens
print(parseur.get_feature_names())
```

Nous observons 6789 termes. Les énumérer serait trop fastidieux. Nous remarquons simplement que la casse a été harmonisée, tous les caractères sont en minuscule.

Pour calculer la fréquence des termes, nous utilisons XTrain. Il est au format « matrice creuse », nous le transformons en matrice « numpy » que nous stockons dans la variable mdtTrain.

```
#transform the sparse matrix into a numpy matrix
mdtTrain = XTrain.toarray()

#type of the matrix
print(type(mdtTrain))

#size of the matrix
print(mdtTrain.shape)
```

Nous disposons d'un matrice documents termes de dimension (3572, 6789). Nous calculons le nombre de documents dans lesquels apparaissent chaque terme, *nous pouvons utiliser la somme puisque nous avons choisi la pondération binaire*. Il ne reste plus qu'à trier le vecteur de fréquences pour mettre en évidence les termes les plus fréquents.

```
#frequency of the terms
freq_mots = np.sum(mdtTrain,axis=0)
print(freq_mots)

#arg sort
index = np.argsort(freq_mots)
print(index)

#print the terms and their frequency
imp = {'terme':np.asarray(parseur.get_feature_names())[index],'freq':freq_mots[index]}
print(pandas.DataFrame(imp))
```

Si l'on s'en tient aux 5 termes les plus fréquents, nous avons :

```
#5 most frequent terms
522              and
528               in
647              the
1042              you
1091               to
```

Le terme ''to'' apparaît dans 1091 documents, ''you'' dans 1042, etc.

3.5 Modélisation à l'aide de la régression logistique

Nous sommes prêts pour lancer la modélisation. Nous choisissons d'utiliser la régression logistique du package scikit-learn. Nous importons et instancions la classe LogisticRegression dédiée, nous lançons le processus avec la méthode fit().

```
#import the class LogistiRegression
from sklearn.linear_model import LogisticRegression

#instatiate the object
modelFirst = LogisticRegression()

#perform the training process
modelFirst.fit(mdtTrain,spamsTrain['classe'])
```

Nous obtenons un vecteur de coefficients avec 6789 éléments, ainsi que la constante.

```
#size of coefficients matrix
print(modelFirst.coef_.shape) #(1, 6789)

#intercept of the model
print(modelFirst.intercept_) #-4.4777
```

Dans le cadre binaire, la fonction de classement (*Note : le logit ici en l'occurrence puisqu'il s'agit de la régression logistique*) est unique, elle s'écrit :

$$D = a_0 + a_1 \times T_1 + a_2 \times T_2 + \cdots + a_p \times T_p$$

Où p est le nombre de termes, a_j est le coefficient pour le terme T_j, a_0 étant la constante du modèle (intercept). La règle de prédiction s'écrit :

Si D(document) > 0 Alors Prédiction = "spam" Sinon Prédiction = "ham"

3.6 Evaluation sur l'échantillon test

Pour appliquer le modèle sur le corpus de texte, nous devons construire la matrice documents termes correspondante, *en utilisant le dictionnaire issu de l'apprentissage*.

Nous appliquons la méthode transform() de l'objet **parseur** instancié lors de la phase d'apprentissage (section 3.4) sur le corpus de test spamsTest issu de la partition aléatoire des données (section 3.3).

```
#create the document term matrix
mdtTest = parseur.transform(spamsTest['message'])

#size of the matrix
print(mdtTest.shape)
```

Nous avons une matrice (2000, 6789) cette fois-ci : 2000 lignes parce qu'il y a 2000 documents dans le corpus de test, 6789 colonnes parce que 6789 termes ont été extraits du corpus d'apprentissage (page 6).

Nous calculons les prédictions du modèle sur les documents de l'échantillon test…

```
#prediction for the test set
predTest = modelFirst.predict(mdtTest)
```

… et déduisons les différents indicateurs de performances en les confrontant avec les valeurs observées de l'attribut cible sur l'échantillon test. Nous utilisons les procédures du module metrics de scikit-learn.

```
#import the metrics class for the performance measurement
from sklearn import metrics

#confusion matrix
mcTest = metrics.confusion_matrix(spamsTest['classe'],predTest)
print(mcTest)

#recall
print(metrics.recall_score(spamsTest['classe'],predTest,pos_label='spam'))

#precision
print(metrics.precision_score(spamsTest['classe'],predTest,pos_label='spam'))

#F1-Score
print(metrics.f1_score(spamsTest['classe'],predTest,pos_label='spam'))

#accuracy rate
print(metrics.accuracy_score(spamsTest['classe'],predTest))
```

Nous obtenons respectivement :

<table>
<tr><th>Indicateur</th><th>Valeur</th></tr>
<tr><td>Matrice de confusion</td><td>
<table>
<tr><td></td><td colspan="2">Prédiction</td></tr>
<tr><td></td><td>Ham</td><td>Spam</td></tr>
<tr><td>Ham</td><td>1732</td><td>0</td></tr>
<tr><td>Spam</td><td>38</td><td>230</td></tr>
</table>
</td></tr>
<tr><td>Rappel</td><td>0.858</td></tr>
<tr><td>Précision</td><td>1.0</td></tr>
<tr><td>F1-Score</td><td>0.924</td></tr>
<tr><td>Taux de succès</td><td>0.981</td></tr>
</table>

Le modèle est plutôt pas mal, il n'y a aucun faux positif c.-à-d. à chaque fois que le modèle désigne un « spam », il le fait à bon escient. Après, il laisse quand même passer 14.2% des spams, des SMS frauduleux parviennent jusqu'à l'utilisateur si nous déployons ce système.

3.7 Réduction de dimensionnalité 1 – Stop words et fréquence des termes

Une étude très succincte du dictionnaire montre que les termes qui reviennent très souvent ne sont pas porteurs de sens : « to », « you », « the », … On parle de *stop words* (mots vides en français). Ils ne permettent pas – a priori – de discriminer les documents, il paraît pertinent de les supprimer du dictionnaire. A l'inverse, nous pouvons aussi considérer que les termes trop rares ne sont pas décisifs car anecdotiques. Nous les retirons également du

dictionnaire.

Dans cette section, nous réitérons l'analyse précédente en introduisant ces deux options lors de l'instanciation de la classe CountVectorizer : stop_words = 'english' pour le retrait des mots vides, min_df = 10 pour retirer les termes qui apparaissent dans moins (strictement) de 10 documents.

```
#rebuild the parser with new options : stop_words='english' and min_df = 10
parseurBis = CountVectorizer(stop_words='english',binary=True, min_df = 10)
XTrainBis = parseurBis.fit_transform(spamsTrain['message'])
#number of tokens
print(len(parseurBis.get_feature_names()))
#document term matrix
mdtTrainBis = XTrainBis.toarray()
#instatiate the object
modelBis = LogisticRegression()
```

```
#perform the training process
modelBis.fit(mdtTrainBis,spamsTrain['classe'])
#create the document term matrix for the test set
mdtTestBis = parseurBis.transform(spamsTest['message'])
#prediction for the test set
predTestBis = modelBis.predict(mdtTestBis)
#confusion matrix
mcTestBis = metrics.confusion_matrix(spamsTest['classe'],predTestBis)
print(mcTestBis)
#recall
print(metrics.recall_score(spamsTest['classe'],predTestBis,pos_label='spam'))
#precision
print(metrics.precision_score(spamsTest['classe'],predTestBis,pos_label='spam'))
#F1-Score
print(metrics.f1_score(spamsTest['classe'],predTestBis,pos_label='spam'))
#accuracy rate
print(metrics.accuracy_score(spamsTest['classe'],predTestBis))
```

Avec plus de 12 fois moins de termes (**541 termes** vs. 6789 pour être précis), nous obtenons des performances équivalentes :

<table>
<tr><th>Indicateur</th><th>Valeur</th></tr>
<tr><td>Matrice de confusion</td><td>
<table>
<tr><td></td><td colspan="2">Prédiction</td></tr>
<tr><td></td><td>Ham</td><td>Spam</td></tr>
<tr><td>Ham</td><td>1731</td><td>1</td></tr>
<tr><td>Spam</td><td>37</td><td>231</td></tr>
</table>
</td></tr>
</table>

Rappel	0.862
Précision	0.996
F1-Score	0.924
Taux de succès	0.981

Nous avons un modèle plus simple tout en préservant la qualité de prédiction.

3.8 Réduction de dimensionnalité 2 – Post traitement du modèle

3.8.1 Stratégie de sélection de variables

Est-il possible de réduire encore la dimensionnalité ? S'intéresser aux propriétés du modèle prédictif produit par la régression logistique constitue une autre piste. Certains coefficients de la fonction de classement sont *quasiment* nuls, ils pèsent de manière négligeable dans la décision. Une stratégie simple consiste (1) à retirer les termes correspondants du dictionnaire, (2) à ré-estimer les paramètres du modèle composé des termes restants.

Il se justifie exclusivement par des considérations de capacité de calcul face à une forte dimensionnalité. Dans la philosophie « machine learning », la démarche exploratoire par essais et erreurs (*trial and error*) fait souvent par **sélection pas à pas**.

Ces réserves étant émises. Essayons quand même d'implémenter notre nouvelle stratégie.

Tout d'abord il nous faut caractériser les coefficients du modèle. Nous les passons en valeur absolue et nous calculons plusieurs quantiles.

```
#absolute  value of the coefficients
coef_abs = np.abs(modelBis.coef_[0,:])

#percentiles of the coefficients (absolute value)
thresholds = np.percentile(coef_abs,[0,25,50,75,90,100])
print(thresholds)
```

Nous obtenons différentes valeurs

```
#percentiles of the coefficients (absolute value)
[ 0.01367356  0.17817203  0.30258512  0.60639769  1.03953052  2.70949586]
```

La plus petite valeur des coefficients en valeur absolue est `0.01367356`, la plus grande `2.70949586`. Nous optons pour le 1er quartile **0.17817203** pour définir le seuil. Nous identifions les numéros des termes correspondants.

```
#identify the coefficients "significantly" higher than zero
#use 1st quartile as threshold
indices = np.where(coef_abs > thresholds[1])
print(len(indices[0]))
```

405 descripteurs ont été retenus (contre 541 précédemment, après élimination des stop words et des termes peu fréquents, section 3.7).

Nous créons les matrices documents termes correspondantes, en apprentissage et en test.

```
#train and test sets
mdtTrainTer = mdtTrainBis[:,indices[0]]
mdtTestTer = mdtTestBis[:,indices[0]]

#checking
print(mdtTrainTer.shape)
print(mdtTestTer.shape)
```

Nous obtenons respectivement $(3572, 405)$ et $(2000, 405)$.

Remarque : Nous pouvons procéder directement à partir des matrices documents termes de l'analyse précédente parce que nous nous appuyons sur une pondération simple (présence / absence des termes). Dans le cas où la pondération tiendrait compte de la longueur des documents (ex. fréquence relative des termes), plutôt que de se lancer dans des calculs compliqués, il aurait été plus judicieux de filtrer le dictionnaire puis de réitérer la construction des matrices.

Il nous reste à reproduire le processus de modélisation (**modelTer**) et d'évaluation.

```
#instatiate the object
modelTer = LogisticRegression()
#train a new classifier with selected terms
modelTer.fit(mdtTrainTer,spamsTrain['classe'])
#prediction on the test set
predTestTer = modelTer.predict(mdtTestTer)
#confusion matrix
mcTestTer = metrics.confusion_matrix(spamsTest['classe'],predTestTer)
print(mcTestTer)
#recall
print(metrics.recall_score(spamsTest['classe'],predTestTer,pos_label='spam'))
#precision
print(metrics.precision_score(spamsTest['classe'],predTestTer,pos_label='spam'))
#F1-Score
print(metrics.f1_score(spamsTest['classe'],predTestTer,pos_label='spam'))
#accuracy rate
print(metrics.accuracy_score(spamsTest['classe'],predTestTer))
```

Avec un **F1-Score** de **0.926,** nous constatons que la qualité de la modélisation n'est en rien affectée par la réduction de la dimensionnalité. Voici le détail des résultats :

Indicateur	Valeur		
Matrice de confusion		Prédiction	
		Ham	Spam
	Ham	1731	1
	Spam	36	232
Rappel	0.866		
Précision	0.996		
F1-Score	0.926		
Taux de succès	0.981		

Nous sommes passés de 6789 à **405 termes**, tout en préservant les performances prédictives du dispositif. Le bilan est plutôt positif.

3.8.2 Interprétation des résultats – Influence des termes dans le modèle

Essayons d'identifier les termes les plus discriminants. Pour ce faire, nous trions le dictionnaire en fonction de la valeur absolue des coefficients du modèle :

```
#selected terms
sel_terms = np.array(parseurBis.get_feature_names())[indices[0]]

#sorted indices of the absolute value coefficients
sorted_indices = np.argsort(np.abs(modelTer.coef_[0,:]))

#print the terms and theirs coefficients
imp = {'term':np.asarray(sel_terms)[sorted_indices],'coef':modelTer.coef_[0,:][sorted_indices]}
print(pandas.DataFrame(imp))
```

Les 10 termes les plus discriminants dans le modèle sont (avec les coefficients associés) :

```
1.760636      text
1.798298      http
1.823208      free
1.884867        50
1.948201       txt
1.999089       new
2.058226      150p
2.201104   service
2.249400     claim
2.715046        uk
```

Les coefficients de ces termes étant positifs, tous concourent à la désignation des « spam » c.-à-d. lorsqu'ils sont présents dans les documents, les chances d'avoir affaire à un « spam » augmentent. L'analyse fine des résultats commence à ce stade. Il est à prévoir vraisemblablement qu'il faudra affiner le dictionnaire pour améliorer la pertinence du dispositif.

3.8.3 Réserves à propos de la sélection de variables

Répétons-le encore une fois, l'approche décrite dans cette section pour éliminer les termes non pertinents est particulièrement discutable. Retirer en bloc les descripteurs dont les coefficients estimés sont quasi-nuls – en dehors de toute autre considération – n'est valable que s'ils (les descripteurs) sont deux à deux indépendants. En pratique, nous aurions dû tenir compte de leurs covariances pour effectuer les tests de nullité des coefficients. Ou encore passer par des tests de rapport de vraisemblance. Dans les deux cas, l'inflation des calculs rend la démarche impraticable sur des jeux de données pouvant contenir plusieurs centaines voire milliers de descripteurs.

Le choix du seuil est également sujet à caution. Les algorithmes de machine learning sont par nature paramétrés. Utiliser le 1er quartile comme seuil n'est pas plus hasardeux que choisir un risque de première espèce α dans un test classique de significativité (10% ? 5 % ? 1 %

? la question reste ouverte souvent, il faut prendre α plutôt comme un paramètre de contrôle des algorithmes dans un processus de statistique exploratoire).

3.9 Déploiement

Une des finalités de la catégorisation de textes est de produire une fonction permettant d'assigner automatiquement une classe (« spam » ou « ham ») à un nouveau document. Dans cette section, nous détaillons les différentes étapes des opérations pour montrer que la tâche est loin d'être triviale.

Nous souhaitons classer la phrase « this is a new free service for you only » à l'aide de notre troisième modèle **modelTer** sachant que la sélection de variables opérée (section 3.8) va compliquer un peu les choses.

Description compatible avec la matrice documents termes. Nous transformons le document en un vecteur de présence absence des termes présents dans le dictionnaire :

```
#document to classify
doc = ['this is a new free service for you only']

#get its description
desc = parseurBis.transform(doc)
print(desc)
```

Python nous dit qu'il a recensé les termes n° 166, 315 et 405. Nous avons une description « spare » des données c.-à-d. seules les valeurs différentes de 0 (zéro) sont recensées.

```
  (0, 166)      1
  (0, 315)      1
  (0, 405)      1
```

De quels termes s'agit-il ?

```
#which terms
print(np.asarray(parseurBis.get_feature_names())[desc.indices])
#print(np.asarray(parseurBis.get_feature_names())[desc.indices])
['free' 'new' 'service']
```

De 'free', 'new' et 'service'.

Par conséquent, les autres termes ('this', 'is', 'a', 'for', 'you', 'only') ne sont pas pris en compte car absents du dictionnaire. Ils ne pèseront en rien dans le classement du message.

Application de la sélection de variables. Puis, nous transformons le vecteur en matrice dense pour pouvoir pratiquer la sélection de variables.

```
#dense representation
dense_desc = desc.toarray()

#apply var. selection
dense_sel = dense_desc[:,indices[0]]
```

Prédiction de la classe d'appartenance. Nous pouvons appliquer la méthode predict() de l'objet modelTer.

```
#prediction of the class membership
pred_doc = modelTer.predict(dense_sel)
print(pred_doc)
```

Python nous annonce la classe 'spam'.

Crédibilité de la prédiction. Une prédiction, c'est bien. Disposer d'une indication sur la crédibilité de la prédiction, c'est mieux. Nous faisons calculer les probabilités d'appartenance aux classes avec la méthode predict_proba().

```
#prediction of the class membership probabilities
pred_proba = modelTer.predict_proba(dense_sel)
print(pred_proba)
```

L'appartenance du message à la classe 'spam' ne semble faire de doute avec une probabilité d'appartenance égale à 0.9215.

```
#print(pred_proba)
[[ 0.07846502  0.9215349]]
```

Vérification manuelle des calculs. Puisque nous disposons des coefficients du modèle, nous pouvons reproduire les calculs :

```
#checking - logit
logit = 1.823208 + 1.999089 + 2.201104 + modelTer.intercept_

#probability – logistic function
import math
p_spam = 1/(1+math.exp(-logit))
print(p_spam)
```

Il ne fallait surtout pas oublier la constate (**modelTer.intercept_**) dans le calcul du logit.

Nous obtenons *exactement* (notre précision est limitée par le nombre de chiffres après la virgule pris en compte dans le calcul manuel) la même valeur qu'avec predict_proba().

```
#print(p_spam)
0.9215349...
```

4 Conclusion

L'analyse statistique des données textuelles est une branche passionnante du data mining. Elle met à l'épreuve à la fois nos connaissances en statistique et en informatique. Cet atelier trace dans ses grandes lignes le processus de catégorisation de textes sur un problème de détection de SPAMS dans des messages de type SMS. Le problème est suffisamment réaliste pour qu'on se rende compte à la fois de la saveur du domaine et des difficultés qu'elle représente.

Et encore, nous avons fait simple. Les pistes de développement sont nombreuses : réduire la dimensionnalité via des techniques basées sur les caractéristiques des termes (ex. stemming, lemmatisation, …), via de techniques statistiques de sélection plus agressives ou

plus sophistiquées (ex. méthodes de ranking basées sur la corrélation), via des méthodes de changement de représentation (ex. topic modeling) ; exploiter un autre système de pondération ; utiliser d'autres méthodes prédictives (ex. SVM, random forest, gradient boosting, etc…