Fouille de données – TD 3

RAPPEL: Installez les packages si vous ne l'avez pas déja fait. Pensez au pense-bête Python.

On utilise le dataset SMSSpamCollection (allez regarder la data!!), et on va produire un estimateur complet, et l'évaluer.
Faites tout votre TD dans un fichier `td3.py`, que vous rendrez via Moodle avant 23h59.

Pour **tester** votre travail: téléchargez `test3.tar.xz`, décompressez-le dans le même répertoire que votre fichier td3.py: `tar xf test3.tar.xz`, puis ensuite: `python3 test3.py`

Ne vous inquiétez pas si vous ne faites pas tout. On reprendra ce TD la semaine prochaine, pour le finir et pour en faire un peu plus. **C'est le TD le plus important du cours**, donc ça vaut le coup de le travailler plus que les autres.

Exercice 1: Séparation des données en training + test

Implémenter cette fonction dans votre fichier `td3.py` :

```
def split_lines(input, seed, output1, output2):
  """Distributes the lines of 'input' to 'output1' and 'output2' pseudo-randomly.

  The output files should be approximately balanced (50/50 chance for each line
  to go either to output1 or output2).

  Args:
    input: a string, the name of the input file.
    seed: an integer, the "seed" of the pseudo-random generator used. The split
        should be different with different seeds. Conversely, using the same
        seed and the same input should yield exactly the same outputs.
    output1: a string, the name of the first output file.
    output2: a string, the name of the second output file.
  """
```

On pourra utiliser le module `random` et sa fonction `seed` pour obtenir un générateur pseudo-aléatoire qui respecte les contraintes demandées.

Qu'est-ce qu'une "**seed**" ??? Si vous ne connaissez pas, regardez wikipedia (aussi en français,moins bien)

On peut utiliser `for line in open(file, 'r').readlines():` pour itérer sur les lignes d'un fichier `file` (par exemple).

Exercice 2: Encodage

Implémenter cette fonction dans votre fichier `td3.py` :

```
def tokenize_and_split(sms_file):
  """Parses and tokenizes the sms data, splitting 'spam' and 'ham' messages.

  Args:
    sms_file: a string, the name of the input SMS data file.

  Returns:
    A triple (words, l0, l1):
    - words is a dictionary mapping each word to a unique, dense 'word index'.
      The word indices must be in [0...len(words)-1].
    - l0 is a list of the 'spam' messages, encoded as lists of word indices.
    - l1 is like l0, but for 'ham' messages.
  """
```

On pourra (par exemple) utiliser `line.split()` pour décomposer une ligne en ses mots (“word”).

Exemple:
Si le fichier `'/tmp/tmp.txt'` contient le texte suivant:

```
ham    Hello World
spam   awesome stuff Hello
ham    Hello Hello ?
```

Alors `tokenize_and_split('/tmp/tmp.txt')` doit renvoyer:

```
({'Hello':0 'World':1 'awesome':2 'stuff':3 '?':4,},
 [[2, 3, 0]],
 [[0, 1], [0, 0, 4]])
```

Exercice 3: Fréquences de Bernoulli

Implémenter cette fonction dans votre fichier `td3.py` :

```
def compute_frequencies(num_words, documents):
  """Computes the frequency of words in a corpus of documents.

  Args:
    num_words: the number of words that exist. Words will be integers in
        [0..num_words-1].
    documents: a list of lists of integers. Like the l0 or l1 output of
        tokenize_and_split().

  Returns:
    A list of floats of length num_words: element #i will be the ratio
    (in [0..1]) of documents containing i, i.e. the ratio of indices j
    such that "i in documents[j]".
    If index #i doesn't appear in any document, its frequency should be zero.
  """
```

On pourra convertir une liste L en ensemble (sans doublons) avec la syntaxe `set(L)`.

Exemple:
`compute_frequencies(6, [[0, 1, 1], [0, 4, 0]])` doit renvoyer `[1.0, 0.5, 0.0, 0.0, 0.5, 0.0]`

Exercice 4: Training

Implémenter cette fonction dans votre fichier `td3.py`. Vous pourrez appeler les fonctions faites précedemment :

```
def naive_bayes_train(sms_file):
  """Performs the "training" phase of the Naive Bayes estimator.

  Args:
    sms_file: a string, the name of the input SMS data file.

  Returns:
    A triple (spam_ratio, words, spamicity) where:
    - spam_ratio is a float in [0..1] and is the ratio of SMS marked as 'spam'.
    - words is the dictionary output by tokenize_and_split().
    - spamicity is a list of num_words floats, where num_words = len(words) and
      spamicity[i] = (ratio of spams containing word #i (across all spams)) /
                     (ratio of SMS (spams and hams) containing word #i (across all SMS))
  """
```

Exemple:
Avec le fichier `'/tmp/tmp.txt'` décrit précédemment, `naive_bayes_train('/tmp/tmp.txt')` doit donner:

```
(0.3333333333333,
 {'Hello':0, 'World':1, 'awesome':2, 'stuff':3, '?':4,},  # L'ordre peut être différent mais pas les valeurs!
 [1.0, 0.0, 3.0, 3.0, 0.0])
```

Exercice 5: Prédiction

Implémenter cette fonction dans votre fichier `td3.py`:

```
def naive_bayes_predict(train_spam_ratio, train_words, train_spamicity, sms):
  """Performs the "prediction" phase of the Naive Bayes estimator.

  You should use the simple formula:
  P(spam|words in sms) = spam_ratio * Product[word in sms]( P(word|spam) / P(word) )
  Make sure you skip (i.e. ignore) the SMS words that are unknown (not in 'train_words').
  BE CAREFUL: if a word is repeated in the sms, it shouldn't appear twice here!

  Args:
    train_spam_ratio: see output of naive_bayes_train
    train_words: see output of naive_bayes_train
    train_spamicity: see output of naive_bayes_train
    sms: a string (which you can tokenize to obtain a list of words)

  Returns:
    The estimated probability that the given sms is a spam.
  """
  ...
```

Exemple:
Avec le fichier `'/tmp/tmp.txt'` décrit précédemment,
`naive_bayes_predict(*naive_bayes_train('/tmp/tmp.txt'), 'Hello dude')` doit donner `0.3333333333`,
Avec `'awesome stuff!'` il doit donner `1.0`, de même que pour `'awesome awesome awesome'`.
Avec `'Oh no!'` il doit donner `0.3333333333`, avec `'awesome ? ? ? awesome ? ?'` il doit donner `0.0`.

Exercice 6: Évaluation

Implémenter cette fonction dans votre fichier `td3.py`:

```
def naive_bayes_eval(test_sms_file, f):
  """Evaluates a spam classifier.

  Args:
    test_sms_file: a string, the name of the input 'test' SMS data file.
    f: a function. f(sms), where sms is a string (like "Hi. Where are you?",
        should return 1 if sms is classified as spam, and 0 otherwise.

  Returns:
    A pair of floats (recall, precision): 'recall' is the ratio (in [0,1]) of
    spams in the test file that were successfully identified as spam, and
    'Precision' is the ratio, among all sms that were predicted as spam by f, of
    sms that were indeed spam.
  """
  ...
```

Notez qu'en python il est simple de manipuler des objets "fonction".

Exemples:

`naive_bayes_eval('/tmp/tmp.txt', lambda x:1)` devrait renvoyer `(1.0, 0.33333333)`
`naive_bayes_eval('/tmp/tmp.txt', lambda x:0)` devrait renvoyer `(0.0, 1.0)`
`naive_bayes_eval('/tmp/tmp.txt', lambda sms:'awesome' in sms)` devrait renvoyer `(1.0, 1.0)`

Et pour tout essayer à la fois:

`spam_ratio, words, spamicity = naive_bayes_train('/tmp/tmp.txt')` suivi de

```
naive_bayes_eval('/tmp/tmp.txt',
                 lambda x:naive_bayes_predict(spam_ratio, words, spamicity, x)>0.5)
```

devrait renvoyer: `(1.0, 1.0) # Quelle star`

Exercice 7: Tuning manuel

Séparez le dataset SMSSpamCollection en un fichier `train` et un fichier `test` grâce à l'exo 1, entraînez votre classifieur (`naive_bayes_train`) sur le fichier `train`, puis évaluez votre superbe algo de prédiction sur le fichier `test`, à l'aide de l'exercice précédent, et notamment le dernier exemple. Qu'obtenez vous?

- Essayez avec d'autres valeurs seuils que 0.5.
- Essayez de trouver le meilleur compromis (recall, precision).
- **Matérialisez** votre estimateur final dans la fonction suivante:

```
def classify_spam(sms):
  """Returns True is the message 'sms' is predicted to be a spam."""
  ...
```

La version 0 pourrait donc être implémentée par

```
def classify_spam(sms):
    # Ici, spam_ratio, words, spamicity sont des constantes globales calculées
    # par votre programme UNE FOIS POUR TOUTES, et donc surtout pas recalculées par
    # cette fonction à chaque fois!! (demandez-moi si vous ne comprenez pas!).
    # Vous pouvez supposer la présence du fichier 'SMSSpamCollection' dans le répertoire.
    return naive_bayes_predict(spam_ratio, words, spamicity, sms) > 0.5
```

[Optionnel, car on fera beaucoup tout ça au TD4]
N'hésitez surtout pas à bidouiller d'autres choses! Par exemple:

- on pourra *copier* `tokenize_and_split`, `compute_frequencies`, `naive_bayes_train` et `naive_bayes_predict` (⚠ gardez les originaux pour bien valider les exercices précédents!), et en faire des versions tunées `better_tokenize_and_split`, etc, qui seront mieux adaptées à ce qu'on fait ici:
- On pourra jeter les mots trop rares par exemple..
- .. et/ou on pourra gérer différemment la "spamicité" des mots qui ne sont jamais dans un spam (au lieu de spamicity = 0, ???)
- Le package "re" de python est très pratique pour la manipulation avancée des chaînes de caractères. N'hésitez pas à me demander, par exemple, comment traiter la ponctuation…

Notez que **ce TD se prolongera la semaine prochaine** (évolution, tuning), cf TD4.