TP : Analyse de sentiments dans les critiques de films

Objectifs

1. Implémenter une manière simple de représenter des données textuelles
2. Implémenter un modèle d'apprentissage statistique basique
3. Utiliser ces représentations et ce modèle pour une tâche d'analyse de sentiments
4. Tenter d'améliorer les résultats avec des outils venus du traitement automatique du langage
5. Comparer les résultats avec une l'implémentation de Scikit-Learn, et avec d'autres méthodes de représentation ou d'apprentissage.

Dépendances nécessaires

Pour les objectifs 4. et 5., on aura besoin des packages suivants:

- The Machine Learning API Scikit-learn : http://scikit-learn.org/stable/install.html (http://scikit-learn.org/stable/install.html)
- The Natural Language Toolkit : http://www.nltk.org/install.html (http://www.nltk.org/install.html)

Les deux sont disponibles avec Anaconda: https://anaconda.org/anaconda/nltk (https://anaconda.org/anaconda/nltk) et https://anaconda.org/anaconda/scikit-learn (https://anaconda.org/anaconda/scikit-learn)

Entrée [ ]:

```
import os.path as op
import numpy as np
```

Charger les données

L'ensemble des données est disponible ici: https://ai.stanford.edu/~amaas/data/sentiment/ (https://ai.stanford.edu/~amaas/data/sentiment/) (pour faciliter la récupération, vous pouvez simplement décompresser cette archive (https://drive.google.com/file/d/1t_cai2X5VUt1yG2DHiMDBCfpRuz562wn/view?usp=sharing) dans le dossier du notebook)

On récupère les données textuelles dans la variable *texts*

On récupère les labels dans la variable y qui en contient *len(texts)* : 0 indique que la critique correspondante est négative tandis que 1 qu'elle est positive.

Entrée [ ]:

```
from glob import glob
filenames_neg = sorted(glob(op.join('.', 'data', 'imdb1', 'neg', '*.txt')))
filenames_pos = sorted(glob(op.join('.', 'data', 'imdb1', 'pos', '*.txt')))

texts_neg = [open(f, encoding="utf8").read() for f in filenames_neg]
texts_pos = [open(f, encoding="utf8").read() for f in filenames_pos]
texts = texts_neg + texts_pos

#Return an array of [1,len(texts)], filled with ones.
y = np.ones(len(texts), dtype=np.int)
y[:len(texts_neg)] = 0.

print("%d documents" % len(texts))
```

Idée principale

On dispose d'une critique étant en fait une liste de mots $s = (w_1, \ldots, w_N)$, et l'on cherche à trouver la classe associée c - qui dans notre cas, peut-être $c = 0$ ou $c = 1$. L'objectif est donc de trouver pour chaque critique s la classe $\hat{c}$ maximisant la probabilité conditionelle $P(c|s)$:

$$\hat{c} = \underset{c}{\operatorname{argmax}}\, P(c|s) = \underset{c}{\operatorname{argmax}}\, \frac{P(s|c)P(c)}{P(s)}$$

Hypothèse : P(s) est constante pour chaque classe :

$$\hat{c} = \underset{c}{\operatorname{argmax}}\, \frac{P(s|c)P(c)}{P(s)} = \underset{c}{\operatorname{argmax}}\, P(s|c)P(c)$$

Hypothèse naïve : les différentes variables (mots) d'une critique sont indépendantes entre elles :

$$P(s|c) = P(w_1, \ldots, w_N|c) = \Pi_{i=1..N} P(w_i|c)$$

On va donc pouvoir se servir des critiques annotées à notre disposition pour **estimer les probabilités** $P(w|c)$ **pour chaque mot** w **étant donné les deux classes** c. Ces critiques vont nous permettre d'apprendre à évaluer la "compatibilité" entre les mots et classes.

Vue générale

Entraînement: Estimer les probabilités

Pour chaque mot w du vocabulaire V, $P(w|c)$ est le nombre d'occurences de w dans une critique ayant pour classe c, divisé par le nombre total d'occurences dans c. Si on note $T(w, c)$ ce nombre d'occurences, on obtient:

$$P(w|c) = \text{Fréquence de } w \text{ dans } c = \frac{T(w, c)}{\sum_{w' \in V} T(w', c)}$$

Test: Calcul des scores

Pour faciliter les calculs et éviter les erreurs d'*underflow* et d'approximation, on utilise le "log-sum trick", et on passe l'équation en log-probabilités :

$$\hat{c} = \underset{c}{\operatorname{argmax}}\, P(c|s) = \underset{c}{\operatorname{argmax}} \left[\log(P(c)) + \sum_{i=1..N} log(P(w_i|c)) \right]$$

Représentation adaptée des documents

Notre modèle statistique, comme la plupart des modèles appliqués aux données textuelles, utilise les comptes d'occurences de mots dans un document. Ainsi, une manière très pratique de représenter un document est d'utiliser un vecteur "Bag-of-Words" (BoW), contenant les comptes de chaque mot (indifférement de leur ordre d'apparition) dans le document.

Si on considère l'ensemble de tous les mots apparaissant dans nos T documents d'apprentissage, que l'on note V (Vocabulaire), on peut créer **un index**, qui est une bijection associant à chaque mot w un entier, qui sera sa position dans V.

Ainsi, pour un document extrait d'un ensemble de documents contenant $|V|$ mots différents, une représentation BoW sera un vecteur de taille $|V|$, dont la valeur à l'indice d'un mot w sera son nombre d'occurences dans le document.

On peut utiliser la classe **CountVectorizer** de scikit-learn pour mieux comprendre:

Entrée [ ]:

```
from sklearn.feature_extraction.text import CountVectorizer

from sklearn.model_selection import cross_val_score
from sklearn.base import BaseEstimator, ClassifierMixin
```

Entrée [ ]:

```
corpus = ['I walked down down the boulevard', 'I walked down the avenue',
          'I ran down the boulevard', 'I walk down the city','I walk down the the a
vectorizer = CountVectorizer()

Bow = vectorizer.fit_transform(corpus)

print(vectorizer.get_feature_names())
Bow.toarray()
```

On affiche d'abord la liste contenant les mots ordonnés selon leur indice (On note que les mots de 2 caractères ou moins ne sont pas pris en compte).

Détail: entraînement

L'idée est d'extraire le nombre d'occurences $T(w, c)$ de chaque mot w pour chaque classe c, ce qui permettra de calculer la matrice de probabilités conditionelles $\boldsymbol{P}$ telle que:

$$\boldsymbol{P}_{w,c} = P(w|c)$$

Notons que le nombre d'occurences $T(w, c)$ peut être obtenu facilement à partir des représentations BoW de l'ensemble des documents.

Procédure:

 Drawing

Détail: test

Nous connaissons maintenant les probabilités conditionelles données par la matrice $\boldsymbol{P}$. Il faut maintenant obtenir $P(s|c)$ pour le document courant. Cette quantité s'obtient à l'aide d'un calcul simple impliquant la représentation BoW du document et $\boldsymbol{P}$.

Procédure:

 Drawing

Preprocessing du texte: obtenir les représentations BoW

Fonction **à compléter**. Elle renvoie la représentation BoW d'un document.

Quelques pointeurs pour les débutants en Python :

- `string_1.split(string_2)` : split the `string_1` variable using the `string_2` pattern
- `my_list.append(value)` : put the variable `value` at the end of the list `my_list`
- `words = set()` : create a set, which is a list of unique values
- `words.union(my_list)` : extend the set `words`
- `dict(zip(keys, values)))` : create a dictionnary
- `for k, text in enumerate(texts)` : syntax for a loop with the index, `texts` begin a list (of texts !)
- `len(my-list)` : length of the list `my_list`

Entrée [ ]:

```
def count_words(texts):
    """Vectorize text : return count of each word in the text snippets

    Parameters
    ----------
    texts : list of str
        The texts

    Returns
    -------
    vocabulary : dict
        A dictionary that points to an index in counts for each word.
    counts : ndarray, shape (n_samples, n_features)
        The counts of each word in each text.
    """

    words = set()
    for text in texts:
        pass
    n_features = 10
    counts = np.zeros((len(texts, n_features)))
    return vocabulary, counts
```

Naïve Bayes

Classe vide : fonctions **à compléter**

```
def fit(self, X, y)
```

Entraînement : va apprendre un modèle statistique basés sur les représentations X correspondant aux labels y.

```
def predict(self, X)
```

Testing : va renvoyer les labels prédits par le modèle pour les représentations X

Quelques pointeurs pour les débutants en Python :

Utiliser l'API Numpy pour travailler avec des tenseurs

- `X.shape` : for a `numpy.array` , return the dimension of the tensor
- `np.zeros((dim_1, dim_2,...))` : create a tensor filled with zeros
- `np.sum(X, axis = n)` : sum the tensor over the axis n
- `np.mean(X, axis = n)`
- `np.argmax(X, axis = n)`
- `np.log(X)`
- `np.dot(X_1, X_1)` : Matrix multiplication

Entrée [ ]:

```
class NB(BaseEstimator, ClassifierMixin):
    def __init__(self):
        pass

    def fit(self, X, y):
        return self

    def predict(self, X):
        return (np.random.randn(len(X)) > 0).astype(np.int)

    def score(self, X, y):
        return np.mean(self.predict(X) == y)
```

Expérimentation

On utilise la moitié des données pour l'entraînement, l'autre pour tester le modèle.

Entrée [ ]:

```
# Ici, on part d'un cinquième des données, pour des questions de temps de calcul
texts_red = texts[0::5]
y_red = y [0::5]

print('Nombre de documents:', len(y_red))
```

Entrée [ ]:

```
voc, X = count_words(texts_red)
```

Entrée [ ]:

```
nb = NB()
nb.fit(X[::2], y_red[::2])
print(nb.score(X[1::2], y_red[1::2]))
```

Cross-validation

Avec la fonction *cross_val_score* de scikit-learn

Entrée [ ]:

```
scores = cross_val_score(nb, X, y_red, cv=5)
print('Score de classification: %s (std %s)' % (np.mean(scores), np.std(scores)))
```

Evaluer les performances:

Quelles sont les points forts et les points faibles de ce système ? Comment y remédier ?

Pour aller plus loin:

Entrée [ ]:

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.pipeline import Pipeline
```

Scikit-learn

Améliorer les représentations

On utilise la function

`CountVectorizer`

de scikit-learn pour constituer notre corpus. Elle nous permettra d'améliorer facilement nos représentations BoW.

Tf-idf:

Il s'agit du produit de la fréquence du terme (TF) et de sa fréquence inverse dans les documents (IDF). Cette méthode est habituellement utilisée pour extraire l'importance d'un terme i dans un document j relativement au reste du corpus, à partir d'une matrice d'occurences $mots \times documents$. Ainsi, pour une matrice $\mathbf{T}$ de $|V|$ termes et D documents:

$$\mathrm{TF}(T, w, d) = \frac{T_{w,d}}{\sum_{w'=1}^{|V|} T_{w',d}}$$

$$\mathrm{IDF}(T, w) = \log\left(\frac{D}{|\{d : T_{w,d} > 0\}|}\right)$$

$$\text{TF-IDF}(T, w, d) = \text{TF}(X, w, d) \cdot \text{IDF}(T, w)$$

On peut l'adapter à notre cas en considérant que le contexte du deuxième mot est le document. Cependant, TF-IDF est généralement plus adaptée aux matrices peu denses, puisque cette mesure pénalisera les termes qui apparaissent dans une grande partie des documents.

Ne pas prendre en compte les mots trop fréquents:

On peut utiliser l'option

```
max_df=1.0
```

pour modifier la quantité de mots pris en compte.

Essayer différentes granularités:

Plutôt que de simplement compter les mots, on peut compter les séquences de mots - de taille limitée, bien sur. On appelle une séquence de n mots un n-gram: essayons d'utiliser les 2 et 3-grams (bi- et trigrams). On peut aussi tenter d'utiliser les séquences de caractères à la place de séquences de mots.

On s'intéressera aux options

```
analyzer='word'
```

et

```
ngram_range=(1, 2)
```

que l'on changera pour modifier la granularité.

Entrée [ ]:

```
## On peut définir un pipeline que l'on modifiera pour expérimenter.

pipeline_base = Pipeline([
    ('vect', CountVectorizer(analyzer='word', stop_words=None)),
    ('clf', MultinomialNB()),
])
scores = cross_val_score(pipeline_base, texts_red, y_red, cv=5)
print("Classification score: %s (std %s)",(np.mean(scores), np.std(scores)))
```

Natural Language Toolkit (NLTK)

Stemming

Permet de revenir à la racine d'un mot: on peut ainsi grouper différents mots autour de la même racine, ce qui facilite la généralisation. Utiliser:

```
from nltk import SnowballStemmer
```

Entrée [ ]:

```
from nltk import SnowballStemmer
stemmer = SnowballStemmer("english")
```

Exemple d'utilisation:

Entrée [ ]:

```
words = ['singers', 'cat', 'generalization', 'philosophy', 'psychology', 'philosoph
for word in words:
    print('word : %s ; stemmed : %s' %(word, stemmer.stem(word)))#.decode('utf-8'))
```

Transformation des données:

Classe vide : function **à compléter**

```
    def stem(X)
```

Entrée [ ]:

```
def stem(X):
    X_stem = []
    for text in X:
        pass
    return X_stem
```

Entrée [ ]:

```
texts_stemmed = stem(texts_red)
voc, X = count_words(texts_stemmed)
nb = NB()

scores = cross_val_score(nb, X, y_red, cv=5)
print('Score de classification: %s (std %s)' % (np.mean(scores), np.std(scores)))
```

Partie du discours

Pour généraliser, on peut aussi utiliser les parties du discours (Part of Speech, POS) des mots, ce qui nous permettra de filtrer l'information qui n'est potentiellement pas utile au modèle. On va récupérer les POS des mots à l'aide des fonctions:

```
    from nltk import pos_tag, word_tokenize
```

Entrée [ ]:

```
import nltk
from nltk import pos_tag, word_tokenize
```

Exemple d'utilisation:

Entrée [ ]:

```
pos_tag(word_tokenize(('I am Sam')))
```

Détails des significations des tags POS: [https://stackoverflow.com/questions/15388831/what-are-all-possible-pos-tags-of-nltk (https://stackoverflow.com/questions/15388831/what-are-all-possible-pos-tags-of-nltk)](https://stackoverflow.com/questions/15388831/what-are-all-possible-pos-tags-of-nltk)

Transformation des données:

Classe vide : fonction **à compléter**

```
def pos_tag_filter(X, good_tags=['NN', 'VB', 'ADJ', 'RB'])
```

Ne garder que les noms, adverbes, verbes et adjectifs pour notre modèle.

Entrée [ ]:

```
def pos_tag_filter(X, good_tags=['NN', 'VB', 'ADJ', 'RB']):
    X_pos = []
    for text in X:
        pass
    return X_pos
```

Entrée [ ]:

```
texts_POS = pos_tag_filter(texts_red)
voc, X = count_words(texts_POS)
nb = NB()

scores = cross_val_score(nb, X, y_red, cv=5)
print('Score de classification: %s (std %s)' % (np.mean(scores), np.std(scores)))
```

Stop-words

Les "stop-words" sont les mots apparaissant fréquemment dans les données et que l'on juge non représentatifs. On les considère comme du bruit. Une liste de stop-words est disponible dans le fichier *english.stop*

Entrée [ ]:

```
def readFile(fileName):
    """
     * Code for reading a file.  you probably don't want to modify anything here,
     * unless you don't like the way we segment files.
    """
    contents = []
    f = open(fileName)
    for line in f:
        contents.append(line)
    f.close()
    result = ('\n'.join(contents)).split()
    return result

sw = readFile('english.stop')
sw[0:50]
```

Transformation des données:

Classe vide : fonction **à compléter**

```
def filterStopWords(X)
```

Entrée [ ]:

```
def filterStopWords(X):
    """Filters stop words."""
    X_filtered = []
    for text in X:
        pass
    return X_filtered
```

Entrée [ ]:

```
texts_stop = pos_tag_filter(texts_red)
voc, X = count_words(texts_stop)
nb = NB()

scores = cross_val_score(nb, X, y_red, cv=5)
print('Score de classification: %s (std %s)' % (np.mean(scores), np.std(scores)))
```

Bonus: Utilisation d'un classifieur plus complexe ?

On peut utiliser les implémentations scikit-learn de classifieurs moins naïfs, comme la régression logistique ou les SVM.

Entrée [ ]:

```
from sklearn.svm import LinearSVC
from sklearn.linear_model import LogisticRegression
```

Entrée [ ]:

```

```