MOTEUR DE RECHERCHE

par **Alexandre DUFOUR**

Ce document s'inscrit dans le périmètre du projet tutoré de l'un des groupes de la classe d'Année SPEciale de l'IUT Informatique LYON 1. Réunissant :

- DUFOUR Alexandre
- FERRAND Gérome
- LAURENT Jeremy
- MOUSSU Nathan
- VATON Juliette

Ce groupe était placé sous le tutorat de M. VIDAL VINCENT.

Ce document représente les travaux de **Alexandre DUFOUR**, au sujet d'un programme de **moteur de recherche**.

A terme, le code inscrit dans ce document sera ré-employé dans un programme plus conséquent, réunissant les applications suivantes :

- Moteur de recherche (affecté à DUFOUR Alexandre)
- Détecteur de spams (affecté à LAURENT Jérémy)
- Analyseur d'opinion (affecté à VATON Juliette)

Notre OBJECTIF :

Le but de cette application est de permettre à un utilisateur de rechercher un texte dans un corpus à partir d'une requête tapée au clavier. Comme pour un moteur de recherche web, la requête est préférablement une suite de mots-clés en rapport avec le texte recherché.

A propos du CORPUS :

L'application n'est pas réalisée pour un corpus particulier. Le corpus utilisé ici sera le jeu de données 20newgroups, simplement car il est déjà utilisé par l'application de catégorisation de texte, ce qui permet de pouvoir partager les connaissances.
20newsgroups est une collection de près de 20 000 documents répartis sur 20 thèmes (recueilli par Ken Lang). Cette collection est devenue très populaire parmi les informaticiens désireux de faire des expériences dans le domaine du text mining.
(Nous trouverons plus d'infos sur le dataset ici : http://qwone.com/~jason/20Newsgroups/)

Vers de la DOCUMENTATION COMPLEMENTAIRE :

Cette présentation a été conçue grâce aux documentations suivantes :

https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html
http://www.nltk.org/

Plan du document :

1. Principe de l'algorithme

L'algorithme se base sur l'analyse de la fréquence des mots dans les textes, et fonctionne en deux phases distinctes :

- Une première phase de travail préliminaire de traitement du corpus, qui a pour but de construire une matrice représentatrice de ce dernier
- La phase de recherche à proprement parler, qui consiste représenter de manière vectorielle la requête de l'utilisateur et trouver la ligne de la matrice du corpus (donc le texte) lui correspondant le plus

1.1. Traitement du corpus

Le but est à partir du corpus constitué de textes sous la forme d'une chaîne de caractère de construire une matrice représentatrice de ce dernier. Chaque ligne de la matrice représentera un texte et chaque colonne représentera un mot présent dans le corpus. **La valeur dans la matrice à la place (i, j) correspond à l'importance du mot j pour le texte i par rapport au reste du corpus.** Cette valeur doit permettre de choisir parmis les textes, ainsi l'importance du mot doit être comprise comme importance au sein du texte en lui-même (répétition du mot dans le texte) et importance dans ce texte par rapport aux autres textes (mot présent dans le texte mais peu dans le reste du corpus).

1.1.1 Transformation des textes

Pour construire la matrice, on commence par "découper" les textes en listes de mots (**tokenisation**). On va ensuite éliminer les mots trop communs (**stopwords**), que l'on considère comme ayant trop peu de sens pour être pris en compte (cela peut aussi être la ponctuation). On va ensuite appliquer aux mots des transformations, afin de rassembler ceux ayant un sens identique ou proche, de deux types :

- **lemmatisation** : Rassembler des mots sous un même mot-clé
- **stemming** (ou radicalisation) : Rassembler des mots sous un même radical

On utilise pour cela la bibliothèque **nltk** (Natural Language ToolKit) de python. On définit ainsi la fonction "split" suivante :

In [1]:
```
# Import des fonctions et outils utilisé
from nltk.corpus import stopwords          # liste des mots non pris en compte
from nltk.tokenize import word_tokenize    # fonction de tokenisation
from nltk.stem import WordNetLemmatizer    # fonction de lemmatisation
from nltk.stem.porter import PorterStemmer # fonction de stemming

# "Hyperparamètres"
stop_words = set(stopwords.words('english'))
# On peut modeler la liste des mots non pris en compte
stop_words.update(stop_words,{'.',',','!','?','\'s', '<', '>', ':', ';', '/', '(', ')', '-', '_', '{', '}', '--', '...'})
lmtzr=WordNetLemmatizer()
ps = PorterStemmer()
```

In [2]:
```
def split(text):
    """ Fonction qui prend en paramètre un texte (chaine de caractère) et qui
    renvoie la liste de ses mots ayant été filtrés puis ayant subits certaines
    transformations: lemming puis stemming """

    # Tokenisation ************************************************************
    # Découpage du texte en mots (words est une liste de chaine de caractère)
    words = word_tokenize(text)

    # Stop-words **************************************************************
    # Filtrage des mots : on supprime de words ceux qui sont contenus dans
    # stop_words car ils sont supposés avoir trop peu de sens.
    # (mots trop communs, mots de liaisons, ponctuation)
    words_clean = []
    for word in words:
        if word.lower() not in stop_words:
            words_clean.append(word)

    # Lemming *****************************************************************
    # Transformation des mots en un unique mot-clé (lemme) les représentant.
    # Ex: divided, dividing, divided, divides -> divide
    words_lemmed = [lmtzr.lemmatize(word) for word in words_clean]

    # Stemming ****************************************************************
    # Transformation des mots en un unique radical les représentant.
    # Ex: divided, dividing, divided, divides -> divid
    words_stemmed = [ps.stem(word) for word in words_lemmed]

    return words_stemmed
# End
```

On peut alors identifier les **"tokens"** (mots distincts) du corpus, qui forment donc une base de représentation des textes sous forme vectorielle.

1.1.2 Scoring

Il s'agit maintenant d'affecter un score pour chaque token dans chaque texte, ce qui formera la matrice désirée. Ce score, comme dit précédemment, est basé sur la fréquence du token, et doit être représentatif à la fois de l'importance du token dans le texte en lui-même et par rapport aux autres textes du corpus.

On commence par récupérer les fréquences d'apparition des tokens grâce à la fonction "count" suivante :

In [3]:
```
def count(words, wordbase):
    """ Fonction qui prend en paramètre un texte 'splité' en mots et la liste
    des tokens du corpus, et renvoie le vecteur contenant le nombre d'occurence
    dans le texte des tokens du corpus."""

    vector = [0 for i in range(len(wordbase))]
    for i in range(len(wordbase)):
        if wordbase[i] in words:
            vector[i] += 1
    return vector
# End
```

On peut alors récupérer la matrice représentant la fréquence de chaque token dans chaque texte. On va alors à partir de ces fréquences calculer un score pour chaque token dans chaque texte selon la formule de **TF-IDF** (Term Frequency - Inverse Document Frequency).

La partie TF indique l'importance du token dans le texte. On choisit ici de prendre **TF(t, m) = log (1 + F(t, m))**, où F(t, m) est la fréquence du token m dans le texte t.

La partie IDF indique le degré de spécification d'un token, c'est-à-dire que ce critère est d'autant plus important que le token apparaît dans peu de texte. On peut alors dire qu'il est spécifique aux textes dans lesquels il apparraît. On choisit ici de prendre **IDF(m) = log(N / nt(m))**, où N est le nombre de textes du corpus et nt(m) est le nombre de textes dans lesquels apparaît le token m.

Le score d'un token m pour un texte t dans la matrice (qu'on appelle M) est donné par **M(t, m) = TF(t, m) * IDF(m)**

On peut maintenant définir la fonction "preliminaryWork" qui à partir d'un corpus sous la forme d'une liste de chaîne de caractère, chaque chaîne étant un texte, va construire la matrice des scores des tokens dans les textes.

In [4]:
```
def preliminaryWork(corpus):
    """ Fonction qui prend en paramètre un corpus de texte sous la forme d'une
    liste de chaines de caractère, et qui renvoie la liste des mots utilisés
    comme base pour représenter les textes qui le compose, et la matrice de ces
    textes dans cette base, les coordonnées étant calculées par la formule du
    TF-IDF."""

    # Découpage, tri et transformation des textes (voir split)
    corpus_words = []
    for i in range(len(corpus)):
        corpus_words.append(split(corpus[i]))

    # Construction de la liste des mots du corpus (intersection des mots des
    # textes). Wordset est un objet de type set, intéressant car il permet de
    # faire l'intersection seul, mais pas ordonné. On construit donc wordbase
    # à partir des mots de wordset pour pouvoir associer 1 mot à 1 coordonnée.
    wordset = set()
    for words in corpus_words:
        wordset = wordset.union(set(words))
    wordbase = [word for word in wordset]

    # Construction de la matrice représentant les textes dans la base wordbase.
    # On l'initialise avec les vecteurs dont les coordonnées sont les
    # occurences brutes.
    matrix = []
    for words in corpus_words:
        matrix.append(count(words, wordbase))

    # Calcul du nombe de textes contenant chaque mot
    nt = [0 for m in range(len(wordbase))]
    for m in range(len(wordbase)):
        for line in matrix:
            if line[m] > 0:
                nt[m] += 1

    # Calcul du tf-idf de chaque mot dans chaque texte
    # TF(mot dans un texte) = log(1 + nb d'occurence de ce mot dans ce texte)
    # IDF(mot) = log(nombre de textes total / nombre de texte cntenant ce mot)
    # TF-IDF(mot dans un texte) = TF(mot dans un texte) * IDF(mot)
    for t in range(len(matrix)):
        for m in range(len(wordbase)):
            matrix[t][m] = math.log(1 + matrix[t][m]) * math.log(len(corpus) / nt[m])

    return (matrix, wordbase)
#End
```

On renvoie aussi la liste des tokens (wordbase) pour les traitements à suivre.

1.2 Traitement d'une recherche

Une fois que la matrice représentatrice du corpus a été construite, on va vouloir traiter les requêtes de l'utilisateur. Pour cela, on va procéder en deux temps :

- vectoriser la requête, comme on a vectorisé les textes du corpus
- identifier le texte le plus proche de la requête en calculant la similarité entre la requête et chaque texte

1.2.1 Vectorisation de la requête

On cherche à mettre la requête sous la forme d'un vecteur. Afin que les vecteurs de la requête et des textes soient comparables et que les calculs aient un sens, il faut **appliquer à la requête le même traitement qu'on a appliqué au textes**. Ainsi, on s'assure que les coordonnées de la requêtes portent bien sur les bons tokens.

Pour les coordonnées du vecteur, on va simplement associer à chaque token sa fréquence dans la requête.

Pour cela, on réutilise les fonctions "split" et "count" utilisées pour le traitement du corpus, afin de définir la fonction "vectorisation" suivante :

In [5]:
```
def vectorisation(text, wordbase):
    """ Fonction qui prend en paramètre un texte sous la forme d'une chaine de
    caractère, et la liste des mots du corpus et qui renvoie le vecteur
    représentant le texte dans la base du corpus."""

    return count(split(text), wordbase)
# End
```

1.2.2 Calcul de similarité

On va calculer pour chaque texte un score correspondant à sa proximité à la requête. Pour cela on calcule pour chaque vecteur de texte du corpus (donc ligne de la matrice) sa similarité avec le vecteur de la requête.

On utilise ici la similarité cosinus qui se calcule de la manière suivante :
Sim(t) = <R, T> / (||R|| * ||T||)
Où T est le vecteur du texte t (= la ligne M[t]), R le vecteur de la requête, < , > un produit scalaire et || || la norme associée.

On va donc calculer la liste de ces valeurs pour chaque test, puis sélectionner les meilleurs résultats (le nombre de résultat choisi étant arbitraire). On définit donc les fonctions suivantes :

In [6]:
```
import math

# Nombre de textes conservés pour une requête
nbTop = 10;

def scal(v1, v2):
    """ Fonction qui calcule le produit scalaire entre deux vecteurs de même
    taille et la renvoie."""

    scal = 0
    for i in range(len(v1)):
        scal += v1[i] * v2[i]
    return (scal)
#End


def norm(v):
    """ Fonction qui calcule la norme 2 d'un vecteur et la renvoie."""
    n = math.sqrt(scal(v, v))
    if (n == 0):
        n = 1
    return (n)
#End


def iMax(similarity):
    """ Fonction qui renvoie l'indice du maximum de la liste passée en
    paramètre """
    imax = 0
    for i in range(1,len(similarity)):
        if (similarity[i] > similarity[imax]):
            imax = i
    return (imax)
#End


def top(similarity, nbTop):
    """ Fonction qui renvoie la liste des indices des nbTop éléments
    les plus grands de la liste passée en paramètre (similarity),
    dans l'ordre décroissant. """
    order = []
    cptTop = 0
    imax = iMax(similarity)
    while (similarity[imax] >= 0 and cptTop < nbTop):
        order.append(imax)
        # On met à -1 l'élément dont on vient de prendre l'indice pour ne plus le prendre en compte
        similarity[imax] = -1
        cptTop += 1
        imax = iMax(similarity)
    return (order)
#End


def research(request, matrix, wordbase):
    """ Fonction qui prend en paramètre une requête sous la forme d'une
    chaîne de caractère, la matrice représentatrice du corpus et la liste
    des tokens, et qui renvoie la liste des indices des textes dans le
    corpus correspondant le plus à la requête. """
    # Vectorisation de la requête
    vector = vectorisation(request, wordbase)
    # Calcul de la liste des score de similarité pour chaque vecteur de la matrice
    similarity = []
    for v2 in matrix:
        similarity.append(scal(vector, v2) / (norm(vector) * norm(v2)))
    # On renvoie les meilleurs résultats
    return (top(similarity, nbTop))
#End
```

2. Exécution de l'application

Les fonctions définies précédemment permettent de faire les calculs nécessaires à l'application, et d'avoir des résultats exploitables, il faut alors définir comment se déroule les interractions avec l'utilisateur : récupération de la requête et affichage des résultats. Pour des raisons de simplicité, tout ce fait dans la console python.

On définit une fonction d'affichage des résultats.

In [ ]:
```
def printResearch(order, corpus):
    """ Fonction qui à partir des résultats d'une requête (liste d'indices) et
    du corpus utilisé affiche les textes dans la console, le plus représentatif
    en premier"""
    print("*\t*\t*\t*\t*\t*\t*\t*")
    print("*\t*\tRésultats de la recherche\t*\t*")
    print("*\t*\t*\t*\t*\t*\t*\t*\n\n")
    input("Appuyer sur une touche...\n")
    for i in range(len(order)):
        print("*\t*\t*\t*\t*\t*\t*\t*")
        print("*\t*\tNuméro " + str(i + 1) + " - Texte " + str(order[i]) + "\t*\t*\t*")
        print("*\t*\t*\t*\t*\t*\t*\t*\n\n")
        print(corpus.data[order[i]])
        # Pause entre l'affichage de chaque texte
        input("Appuyer sur une touche...\n")
    print("Fin")
#End
```

Et enfin la fonction principale.

In [ ]:
```
# Corpus utilisé
from sklearn.datasets import fetch_20newsgroups
corpus = fetch_20newsgroups()

# Nombre de texte qu'on prend en compte (optionnel)
nbTxt = 100

###############################################################################

# main

# Construction de la matrice
(matrix, wordbase) = preliminaryWork(corpus.data[:nbTxt])

request = input("Recherche : ")
while request != "exit":
    # Traitemtn d'une recherche
    order = research(request, matrix, wordbase)
    printResearch(order, corpus)
    request = input("Recherche : ")

#End
```