Accès et recherche d'information

Auteurs : Baptiste Bouvier et Ancelin Serre
Date : 28/02/2019
POLYTECH GRENOBLE - INFO4

Sommaire

TP1 : Loi de Zipf

Dans cette première étape, l'essentiel du travail consistait à préparer les fichiers `cacm` en un corpus "*tokenizé*" afin de pouvoir, par la suite, dresser une courbe représentant la fréquence d'apparation des termes du vocabulaire selon leur rang. Cela illustrait ainsi la fameuse Loi de Zipf.

Pour effectuer ces tâches, nous avions déjà à dispositions le script `split_cacm.py` présent dans le répertoire `search_engine/preparation/` nous permettant de découper le fichier `cacm.all` en **3204** fichiers distincts.
Notre premier travail à donc été de *tokenizer* chacun de ces fichiers.
Cela s'est traduit par le script suivant :

```
pattern = r'[A-Za-z]\w{1,}'
tokenizer = RegexpTokenizer(pattern)

# Folder where CACM-XX are stored
folder_name = "../resources/results/"
# Folder where to store .flt files
tokenized_folder_name = "../resources/tokenized/"
safe_mkdir(tokenized_folder_name)

# Reading every file
for filename in os.listdir(folder_name):
    words = []
    with open(folder_name+filename, "r") as f:
        # Tokenizing each line of the file f
        for line in f:
            words += tokenizer.tokenize(line)

    # Lower every words
    words = [w.lower() for w in words]
    # Storing the result into a new file .flt
    with open(tokenized_folder_name+filename+".flt", "w") as output:
        for word in words:
            output.write(word+" ")

    print("File " + filename + " tokenized")
```

On voit bien ici que pour chaque fichier `cacm`, on utilise la fonction `tokenize()` en provenance du package *Python* `nltk` permettant de *tokenizer* un document selon un certain modèle défini dans la variable `pattern` avec une **expression régulière** chargée d'isoler chaque **terme** (pas de nombres ou autres caractères spéciaux) d'un fichier ligne après ligne.
Une fois cette étape intermédiaire réalisée, on a juste à réécrire les données *tokenizées* en ajoutant pour chaque document tout ses mots sur une ligne, tous séparés par un `espace`.
Ces fichiers seront finalement stockés dans le répertoire `search_engine/resources/tokenized/` avec l'extension `.flt`

Ces étapes étant désormais réalisées, il nous est maintenant aisé de calculer le lambda de la Loi de Zipf et de dresser son graphique. On trouve ainsi dans le script `zipf.py` :

```
# Folder where to store .flt files
folder_name = "../resources/tokenized/"
occurrences = {}
words = []
total = 0
# Reading every file
for filename in os.listdir(folder_name):
    with open(folder_name+filename, "r") as f:
        # Tokenizing each line of the file f
        for line in f:
            words += line.split()

# Counting occurences of each word
for word in words:
    occurrences[word] = occurrences.get(word, 0) + 1
    total += 1

# Let's construct the analysis data structure
analysis = []
for word in occurrences :
    analysis.append({
        "word" : word,
        "occurrences" : occurrences[word],
        "probability" : round((occurrences[word] / total) * 100, 4)
    })

# Sorting the values in desc order
analysis     = sorted(analysis, key=itemgetter('probability'), reverse=True)
# Computing lambda value
lambda_value = int(total / math.log(len(analysis)))
# Adding theoretical occurrence for each word
for i in range(0, len(analysis)):
    analysis[i]["theoretical"] = int(lambda_value/(i+1))
```

Dans ce code, on se préoccupe essentiellement de calculer le nombre d'occurrences de chaque mot du corpus. Cela nous permet de calculer leur probabilité d'apparition et in fine de calculer la valeur du lambda.

Ainsi, on créée un dictionnaire dans lequel pour chaque terme on sauvegarde sa probabilité d'apparition **pratique** (celle que l'on observe) et **théorique** (calculée selon le rang du terme et la valeur du lambda).
En représentant les deux courbes **pratiques** et **théoriques** via le package `matplotlib`, on observe aisément la précision donnée par la **Loi de Zipf**.

TP2 : Constitution de vocabulaire et représentation

Filtrage et racinisation des mots

Dans cette seconde étape, on commence par appliquer un filtre sur les fichiers *tokenizé* (extenstion `.flt`) venant filtrer les mots communs répertoriés dans le répertoire `search_engine/resources/cacm/` sous le nom `common_words`. Cela est réalisé via le code suivant présent dans la classe `DataPreprocessor.py` :

```
"""
Function used to remove stop words / common words
from each .flt tokenized files using a common words file.
"""
def _stop_list_filter(self):
    # Folder where to store .flt files
    # Reading every file
    for filename in os.listdir(self.utils.tk_path):
        words = []
        with open(self.utils.tk_path+filename, "r") as f:
            for line in f:
                curr_line = line.lower().split()
                words += [self.utils.get_root(word) for word in curr_line if word
not in self.utils.common_words]

        new_name = filename.replace(".flt", ".sttr")
        with open(self.utils.ft_path+new_name, "w") as f:
            for word in words:
                f.write(word+" ")
```

Il s'agit ici simplement de venir générer de nouveaux fichiers filtrés avec l'extension `.sttr` dans un répertoire donné en argument au programme (ou par défaut dans le répertoire `search_engine/resources/stop_words_filtered`).
On parcourt chaque document et pour chacun de leurs mots, on vérifie si ils font partie de la liste des mots communs. Si oui, on ne les réécrira pas, sinon on les conserve et on leur applique la **troncature de Porter** que l'on peut voir ici avec la fonction `get_root(word)` appelée depuis la classe `Utils` du `DataPreprocessor`.
Cette fonction de troncature vient également du package `nltk`

Génération du vocabulaire du corpus

Maintenant que le filtrage a été effectué, nous allons pouvoir générer le vocabulaire du corpus. Cela a été réalisé via la fonction suivante également présente dans la classe `DataProcessor` :

```
"""
Function used to initialize the corpus vocabulary.
```

```
    """
    def init_vocabulary(self):
        print("[DP] Initiating corpus vocabulary...")
        for document in self.files:
            for word in document["content"]:
                if word not in self.vocab:
                    self.vocab[word] = {    # We want fields to compute document
    frequency
                        "df" : 0,           # and inverse document frequency
                        "idf": 0
                    }

        # compute Document Frequency DF for each word of the vocabulary
        self._compute_df()
        # now that we have computed DF for each terms,
        # we can compute IDF using the following formula : idf_i = ln(N/df_i)
        self._compute_idf()
        print("[DP] OK")
```

On parcours chaque document et à chaque fois que l'on rencontre un nouveau mot, on l'ajoute dans le dictionnaire qui fait dans notre cas office de vocabulaire. On peut noter que cette fonction calcule aussi la **document frequency** et **l'inverse document frequency** pour chque terme du vocabulaire ce qui nous servira plus tard lors de la génération d'autres ressources. La **document frequency** décrit le nombre de documents du corpus dans lesquels le terme courant apparaît.

```
    """
    Function used to compute document frequency of each term in the vocabulary.
    """
    def _compute_df(self):
        for current_file in self.files:
            encountered = []
            for word in current_file["content"]:
                # Verify if we already encountered the word once in that file
                if word not in encountered:
                    self.vocab[word]["df"] += 1
                    encountered.append(word)

    """
    Function used to compute inverse document frequency of each term in the
    vocabulary.
    """
    def _compute_idf(self):
        N = len(self.files) # Number of document
        for term in self.vocab:
            self.vocab[term]["idf"] = math.log(N/self.vocab[term]["df"])
```

Les deux fonctions ci-dessus présentes dans la classe `DataPreprocessor` permettent comme leurs noms l'indiquent de calculer respectivement la **document frequency** et **l'inverse document frequency** pour

chaque terme du vocabulaire.
La strucuture finale ressemble à ça :

```
{
        "preliminari": {
                "df": 20,
                "idf": 5.076423034634259
        },
        "report": {
                "df": 100,
                "idf": 3.4669851222001586
        },
    ...
```

Génération de la représentation vectorielle de Salton

La représentation vectorielle de **Salton** consiste à générer pour chaque document un vecteur de pondération `TF.IDF` (**Term Frequency** et **Inverse Document Frequency**). Cela se caractérise dans notre cas par un dictionnaire dans lequel on trouve pour chaque document un nouveau dictionnaire contenant pour chaque terme de ce dernier la pondération `TF.IDF`. Cette donnée est obtenue via la fonction suivante :

```
"""
Function used to initialize salton representation using tf*idf.
"""
def init_salton(self):
    if not self.vocab:
        self.init_vocabulary()
    print("[DP] Initiating Salton vector representation...")
    for current_file in self.files:
        document = {
            "vector_tfidf" : {},
            "vector_tf"  : {}
        }
        # Computing tf for each term
        for term in current_file["content"]:
            if term in document["vector_tfidf"]:
                document["vector_tfidf"][term] += 1
                document["vector_tf"][term]  += 1
            else:
                document["vector_tfidf"][term] = 1
                document["vector_tf"][term]  = 1

        # Computing tf * idf for each term
        for term in document["vector_tfidf"]:
            document["vector_tfidf"][term] *= self.vocab[term]["idf"]

        # Adding the current vector to the salton representation
        self.salton_rep[current_file["name"]] = document

    print("[DP] OK")
```

Rien de très complexe dans cette fonction, on fait d'abord un calcul intermédiaire dans lequel on calcul la **Term Frequency** pour chaque terme de chaque document. Puis, on multiplie cette valeur par **l'Inverse Document Frequency** du terme en question calculée précédemment dans le vocabulaire.

N.B. On peut voir que la structure affichée ici ne correspond pas à celle sauvegardée dans le fichier `salton_representation.json` ce qui s'explique par le fait qu'on a modifié la structure avant de la sauvegarder. En effet, stocker la `TF` de chaque terme ne nous ait pas utile pour cette représentation d'où le choix de l'enlever avant la sauvegarde.
La structure finale ressemble à ça :


```
{
        "CACM-1.sttr": {
                "preliminari": 5.076423034634259,
                "report": 3.4669851222001586,
                "intern": 4.26549281841793,
                "algebra": 4.0117122976418305,
                "languag": 2.1750014405515095,
                "cacm": 0.00031215857909170155,
                "decemb": 2.4626835130032902,
                "perli": 5.58724865840025,
                "samelson": 6.462717395754149
    },
    ...
```


Génération de l'index inversé

Pour pouvoir effectuer des recherches dans le corpus, il nous manque encore un élément essentiel : **l'index inversé**. Il consiste à indexer pour chaque terme du vocabulaire, les documents dans lequel il est référencé. On génère ce dernier via la fonction suivante :

```
"""
Function used to generate the inverted index
Inverted index represents each word present in the corpus
with its df and its tf for each file it is in.
It doesn't compute anything, it just collects and regroups data.
"""
def init_inverted_index(self):
    if not self.salton_rep:
        self.init_salton()
    print("[DP] Initiating Inverted index...")
    for term in self.vocab:
        word = {
            "df" : self.vocab[term]["df"],
            "docs" : {}
        }
        for document in self.salton_rep:
            vector_tfidf = self.salton_rep[document]["vector_tfidf"]
```

```
            if term in vector_tfidf:
                word["docs"][document] = vector_tfidf[term]

        self.inv_index[term] = word
    print("[DP] OK")
```

La structure finale ressemble à ça :

```
{
        "preliminari": {
                "df": 20,
                "docs": {
                        "CACM-1.sttr": 5.076423034634259,
                        "CACM-1205.sttr": 5.076423034634259,
                        "CACM-1235.sttr": 5.076423034634259,
                        "CACM-1726.sttr": 5.076423034634259,
                        "CACM-1771.sttr": 5.076423034634259,
                    ...
```

Génération des normes de chaque document

Finalement, pour pouvoir calculer le **cosinus** entre la requête de l'utilisateur et le document trouvé, il nous faut au préalable calculer les **normes** de chaque document. Cela se fait en calculant *la racine carrée de la somme des pondérations de chaque terme au carré pour un même document*. C'est ce qui est fait dans la fonction ci-dessous présente dans la classe `DataPreprocessor` :

```
"""
Function used to compute vector norms of each document
of the corpus using the following formula :
||vector|| = sqrt(sum(wi**2))
with wi = idf_i in our case.
"""
def init_norms(self):
    if not self.salton_rep:
        self.init_salton()
    print("[DP] Initiating Norms list...")
    for doc in self.salton_rep:
        sum_square_wi = 0
        vector_tfidf = self.salton_rep[doc]["vector_tfidf"]
        for term in vector_tfidf:
            sum_square_wi += vector_tfidf[term]**2

        self.norms[doc] = math.sqrt(sum_square_wi)
    print("[DP] OK")
```

On obtient finalement la structure suivante :

```
{
        "CACM-1.sttr": 12.484303198993095,
        "CACM-10.sttr": 9.89108337886477,
        "CACM-100.sttr": 12.387697343297809,
        "CACM-1000.sttr": 13.415719264302341,
        "CACM-1001.sttr": 57.66180788477066,
    ...
```


TP3 : Recherche et évaluation

Dans cette dernière partie, on veut pouvoir lancer une recherche sur le corpus de document et observer les résultats de cette dernière. Pour arriver à cet objectif, nous avons suivi les étapes suivantes :

- Chargement en mémoire des différents fichiers nécessaires pour la recherche
- Acquisition et prétraitement de la requête utilisateur
- Traitement de la requête utilisateur et affichage des résultats

Chargement en mémoire des différents fichiers nécessaires pour la recherche

Pour charger les données nécessaires, on réutilise tout simplement la classe `DataPreprocessor` dans notre fichier `main.py`

```
if __name__ == "__main__":

    # Reading user arguments
    args = read_args()
    # Instantiating a new Utils object
    utils = Utils(**load_config(args.config)) if args.config else Utils()

    dp = DataPreprocessor(utils)  # Instantiating a new DataPreprocessor
    dp.init_vocabulary()          # Initializing the vocabulary
    dp.init_salton()              # Initializing the salton vector representation
    dp.init_inverted_index()      # Initializing the inverted index
    dp.init_norms()               # Initializing norms list
```

Acquisition et prétraitement de la requête utilisateur

Acquisition de la requête et du nombre de réponses attendues:

```
"""
Function used to get user input, his request and the
number of results he wants
:return:
    request, (str) his request
    nb_res, (int) number of results awaited
"""
def get_user_input():
    # Getting user request
```

```
    request = input("[SearchEngine] >> Enter your request : ")
    if not request: exit(0)
    # Getting number of request awaited
    str_nb  = input("[SearchEngine] >> Enter the number of results you want : ")
    nb_res  = 0
    # Casting it into an integer
    if str_nb and str_nb.isdigit():
        nb_res  = int(str_nb)
    else: exit(0)

    return (request, nb_res)
```

Instanciation d'une nouvelle requête dans la boucle infini de la fonction `main.py`:

```
# Looping while request isn't empty
while(True):
    # Instantiating and preprocessing a new request
    request = Request(request, dp.vocab, utils)
```

Le prétraitement de la requête se fait ainsi dans la classe `Request`. Dans cette dernière, on applique le même traitement que l'on appliqué aux documents, à savoir un filtrage et une racinisation ainsi qu'une mise en minuscule pour chacun des mots de la requête. De plus, on calcule la `TF` pour de chaque mot de la requête, leur **pondération** `TF.IDF` et la **norme** de la requête. Toutes ces fonctions sont présentes ci-dessous dans la classe `Request` :

```
    @request.setter
    def request(self, value):
        # Lowering every word and splitting it on spaces
        # in order to get a list
        reqst = value.lower().split()
        # Applying porter truncature for each word which are not in anti
dictionary
        reqst = [self.utils.get_root(req) for req in reqst if req not in
self.utils.common_words]
        self._request = reqst

    """
    Used to initialize the vector of term frequency
    """
    def _init_vector_tf(self):
        for word in self.request:
            # Computing TF only consists in adding 1
            # to the TF of the word every time we meet it
            self.vector_tf[word] = self.vector_tf.get(word, 0) + 1

    """
    Used to initialize the vector of TF.IDF weights
    """
    def _init_vector_tfidf(self):
```

```
        for word in self.vector_tf:
            # Verifying if the word exists in the vocabulary
            word_vocab = self.vocabulary.get(word, None)
            # Getting its Inverse Document Frequency
            word_idf   = word_vocab["idf"] if word_vocab else 0
            # Computing multiplication between TF and IDF
            self.vector_tfidf[word] = self.vector_tf[word] * word_idf

    """
    Used to initialize the request norm
    """
    def _init_norm(self):
        # Applying the following formula : sqrt(sum(wi**2))
        self.norm = math.sqrt(sum([self.vector_tfidf[word]**2 for word in
self.vector_tfidf]))
```

Traitement de la requête utilisateur et affichage des résultats

Finalement, pour effectuer la recherche, on instancie un nouvel objet `Search` et on lance la recherche avec ce dernier après lui avoir donné les données nécessaires à savoir, la requête, le nombre de réponses attendues, l'index inversé, les normes des documents et une instance d'un objet `Utils` permettant de simplifier des traitements.

```
    """
    Function used to execute the search in the corpus.
    """
    def search(self):
        # Step 1 : Getting corresponding lines for each request term
        cor_lines = self._get_corresponding_lines()
        # Step 2 : Starting the search by computing cosinus for each document
        self._compute_document_cosinus(cor_lines)
        # Step 3 : Generating a list of results sorted on cosinus in desc order
        self._init_final_results()
        # Step 4 : Printing final results
        print(self)

    """
    Function used to get corresponding lines
    for each request term in the inverted index
    :return:
        corresponding_lines, (dict)
    """
    def _get_corresponding_lines(self):
        # Getting corresponding lines for each request term
        # in the inverted index
        corresponding_lines = {}
        for word in self.request.vector_tfidf:
            if word in self.inverted_index:
                corresponding_lines[word] = self.inverted_index[word]["docs"]
        return corresponding_lines
```

```
    """
    Function used to compute document cosinus for each document
    by using the following formula for a document :
    cosinus_doc = sum(request_term_tfidf * document_term_tfidf) / request_norm *
document_norm
    """
    def _compute_document_cosinus(self, corresponding_lines):
        # Intermediate step to compute results
        for word in corresponding_lines:
            # Getting TF.IDF for current word in request
            req_tfidf = self.request.vector_tfidf[word]
            for doc in corresponding_lines[word]:
                # Getting TF.IDF for current word in current document
                doc_tfidf = corresponding_lines[word][doc]
                # If we already saved a value for the current doc,
                # we simply add the new one to it
                self.res[doc] = self.res.get(doc, 0) + (req_tfidf * doc_tfidf)

        # Dividing each value in the intermediate results by
        # the multiplication of request norm and document norm
        for doc in self.res:
            self.res[doc] /= (self.request.norm * self.norms[doc])

    """
    Function used to initialize the list named final_results
    which corresponds to the search results sorted on cosinus value in desc order
    """
    def _init_final_results(self):
        # Placing final results in a list
        # in order to be able to sort it in desc order
        self.final_result = []
        for doc in self.res:
            self.final_result.append({
                "name" : doc,
                "cosinus" : self.res[doc]
            })

        # Sorting final results on cosinus attribute in desc order
        self.final_result = sorted(self.final_result, key=itemgetter("cosinus"),
reverse=True)
```

La fonction `search()` résume relativement bien étape par étape ce qui est fait dans cette classe lors de l'exécution d'une recherche. On cherche dans un premier temps à obtenir tous les documents dans lesquels chaque terme apparaissent. On calcule ensuite le cosinus pour chacun des documents trouvés. Enfin, on restructure les résultats sous la forme d'une liste de dictionnaire afin de pouvoir proposer un affichage cohérent (dans l'ordre décroissant).
Exemple de résultat :

```
[SearchEngine] >> Enter your request : sorting algorithms for large volumes
[SearchEngine] >> Enter the number of results you want : 3
+-----------------------------------------+
| 1489 results found...
+-----------------------------------------+
+-----------------------------------------+
| 1 - [CACM-856] - [rel. ++] :
+-----------------------------------------+
  Sorting with Large Volume, Random Access, Drum Storage
  An approach to sorting records is described
  using random access drum memory.  The Sort program
  described is designed to be a generalized, self-generating
  sort, applicable to a variety of record statements.
+-----------------------------------------+
| 2 - [CACM-1724] - [rel. ++] :
+-----------------------------------------+
  A Generalized Partial Pass Block Sort
  The design of a partial pass block sort with
  arbitrary range of key and number of work files
  is described. The design is a generalization of the Partial
  Pass Column Sort by Ashenhurst and the Amphisbaenic
+-----------------------------------------+
| 3 - [CACM-866] - [rel. ++] :
+-----------------------------------------+
  Sorting on Computers
  CACM May, 1963
  Gotlieb, C. C.
```

On peut aussi voir des `++` dans l'entête de chaque réponse. Cela correspond à une extension que nous avons réalisé qui indique la pertinence de la requête. `+` indique faiblement pertinent et `++++` fortement pertinent.