HénoÏk WILLOT et Haifei ZHANG

LO17 Structuration et Indexation

I. Grandes lignes du projet

Le projet comprend deux parties: la préparation du Corpus et l'indexation du Corpus. Les principales étapes de mise en œuvre des deux parties sont décrites séparément ci-dessous.

Pour préparer le corpus , nous avons analysé la structure d'un fichier HTML pour déterminer les balises à extraire et leur contenu. Nous avons créé deux scripts Perl pour extraire automatiquement les informations et les écrire dans un fichier XML. Enfin, nous utilisons la méthode d'échantillonnage pour vérifier l'exactitude et l'exhaustivité de la méthode.

Pour l'indexation du corpus, Il y a trois étapes principales. Premièrement, nous avons déterminé la stoplist en calculant les coefficients "$tf_{i,\ j} \times idf_i$". Ensuite, nous avons utilisé "successeurs.pl" et "filtronc.pl" pour créer des lemmes. Enfin, nous avons utilisé "index.pl" et "indexText.pl" pour créer des fichiers inverses.

II. Préparation du Corpus: génération du fichier XML

1. L'analyse des fichiers html

Le dossier BULLETINS, contenant tous les articles de notre corpus, contient 326 fichiers HTML. Nous devons extraire les informations dont nous avons besoin de ces 326 fichiers

Premièrement, nous sélectionnons au hasard quelques fichiers à ouvrir dans le navigateur: nous avons constaté que le format de la page Web est le même. La seule différence est la partie centrale de l'article (c'est-à-dire la partie à traiter).

Ici, nous avons choisi 70165.html comme exemple. Nous avons découvert dans la section article que nous pouvons extraire: le numéro du bulletin, la date, la rubrique, le titre de l'article, le texte (brut) de l'article, la ou les ressource(s) images et leur(s) légende(s) respective(s), ainsi que les informations de contact.

Deuxièmement, nous avons examiné le document HTML. Heureusement, les informations que nous devons extraire sont identifiées de manière unique. Par exemple, l'image aura une balise <img>. En termes de texte, nous avons les balises <p> et les styles correspondants. Les autres informations à extraire sont spécifiées par une balise et un style particulier pour chaque type.

```
<p><span class="style32">BE France 270  </span>
<span class="style55">&gt;&gt;</span><span class="
style42">  31/05/2012</span></p></td>
```

Comme indiqué ci-dessus, le numéro de séquence est inclus dans la chaîne entre et </ span>. La date est incluse dans et </ span>.

2. Le choix des balises html discriminantes

Après analyse, nous avons décidé d'utiliser les balises suivantes. pour extraire les informations correspondantes.

Numero :

Date :

Rubrique :

Titre :

Images : <img src="">

LegendeImages :

Texte : tous (sauf images et LegendeImages) entre et </td>

3. Les algorithmes Perl

Pour extraire automatiquement les informations correspondantes à partir des 326 fichiers, nous devons écrire des scripts Perl.

Le premier script appelé "corpus.pl" peut récupérer tous les fichiers du dossier BULLETINS. Nous avons stocké tous les noms de fichiers dans un tableau. En parcourant ce tableau et en transmettant chaque élément en tant que paramètre à la commande UNIX, le deuxième script Perl qui extrait les informations pour chaque fichier html est exécuté. Pour appeler la commande UNIX dans le script perl, nous utilisons la fonction system().

```perl
#!/usr/bin/perl

my $dir = "../BULLETINS";
my $file;
my @dir;

# opendir (DIR, $dir) or die "can't open the directory!";
# @dir = readdir DIR;
@dir = `ls $dir`;
system("touch corpus.XML");
system("echo -n '<corpus>\n' >> corpus.XML");

$count = 1;
foreach $file (@dir) {
    if ( $file =~ /[0-9]*\.htm/) {
    #print "$file\n";
    $instruction = "perl extrait_des_informations.pl ../BULLETINS/$file";
    system($instruction);
    print $count++."\n";
    }
}

system("echo -n '</corpus>\n' >> corpus.XML");

```

Le second script est utilisé pour extraire des informations d'un fichier HTML. Nous avons utilisé l'opérateur “<>” pour lire le contenu du fichier ligne par ligne. Pour chaque ligne de contenu, nous allons effectuer l'opération correspondante.

```perl
while(<>){
    #Opération de traitement de l'information
}
```

Pour chaque ligne de contenu, nous utilisons des expressions régulières pour extraire les informations correspondantes. En perl, la variable “$_” représente la chaîne d'origine et lorsque nous utilisons les parenthèses “( )” autour d’une partie de l'expression régulière, le contenu de la chaîne d’origine qui *match* l’expression contenue entre les parenthèses est stocké dans des variables de “$1” à “$9”. Nous avons utilisé formellement cette technologie pour extraire des informations. Auparavant, nous avons analysé les identifiants uniques pour chaque message et maintenant on les utilise.

Extraire des numéros, des dates, des rubiques et des titres est simple. L'extraction de chaque information ne nécessite que deux lignes de code. Comme indiqué ci-dessous:

```
if(/<span class="style32">BE France (\d+)  <\/span>/){
    $numero = $1;
}
if(/<span class="style42">  (\d+\/\d+\/\d+)<\/span>/){
    $date = $1;
}
if(/<span class="style42">(.+)<br><\/span>/){
    $rubrique = $1;
}
if(/<span class="style17">(.+)<\/span>/){
    $titre = $1;
```

L'extraction de texte est un peu plus compliquée. Ce dont nous avons besoin est un morceau de texte brut (pas d'information sur les images et la légende, pas de sauts de ligne, pas de balises html) et le tout mis sur une seule ligne. Dans l'analyse précédente, nous savons que le texte est entièrement contenu entre <p class = "style96"> et </ span> </ p> </ td> (bien sûr, nous avons besoin de supprimer les informations sur l'image et les balises HTML supplémentaires). Donc, voici la question de quand le texte se commence et quand le texte se termine. Nous avons utilisé la méthode suivante pour résoudre ce problème.

```
if (~ /<p class="style96"><span class="style95">/) {
        chomp;
        $texte .= $_;
        do {
            $_ = <>;
            chomp;
            $texte .= $_;
            } until ($_ =~ /<\/span><\/p><\/td>/)
}
```

A ce moment là, "$texte" possède les informations sur l'image, nous devons ensuite extraire l'adresse URL de l'image et la légende correspondante. Une chose à noter ici est que certaines images ont des légendes et d'autres ne l'ont pas. Par conséquent, le meilleur moyen consiste à extraire l'adresse URL d'une image et leur légende correspondante (chaîne ou vide) à chaque fois et stockez-les dans un hash

```
%images = ();
while($texte_brut =~ /<img src="(.+?)"(.*?)style88/g) {
    $url = $1;

    if ($2 =~ /<span class="style21"><strong>(.*?)<\/strong>/) {
        $legend = $1;
    } else {
        $legend = "";
    }
    $images{$url} = $legend;
}
```

Ensuite, nous avons supprimé toutes les étiquettes non liées au texte brut (images, légendes, caractères gras, sauts de ligne, etc.).

```
#Supprimer les balises supplémentaires du texte
$texte =~ s/.+<p class="style96"><span class="style95">|<\/div><span class="style95">|<a href=.+?>|<\/a>//g;
$texte =~ s/<\/span><div style="text-align: center"><img src.+?>|<span class="style21"><strong>.+?<\/span>//g
$texte =~ s/<span class="style88">.+?<\/span>//g
$texte =~ s/<\/span>|<br \/>|<\/div><span class="style95">|<strong>|<\/strong>||<\/p><\/td>/ /g;
```

Jusqu'ici, nous avons correctement extrait les informations d'un fichier html, il suffit maintenant de tout écrire dans un fichier XML, en utilisant un format approprié.

4. La vérification la qualité de l'extraction

Nous avons utilisé diverses méthodes pour vérifier l'exactitude et l'exhaustivité du traitement de l'information.

1) Nous avons vérifié que nous avons traité 326 fichiers en comptant le nombre de balises <titre> dans le fichier XML.

2) En comparant les résultats avec les camarades de classe, nous avons confirmé que le nombre de 155 images extraites semble correct.

3) Grâce à un échantillonnage aléatoire, nous avons vérifié le texte des cinq articles et les informations des autres tags. Les résultats sont tous corrects.

III. Indexation

1. Déterminer la *stoplist* et éliminer son contenu du corpus

Afin de pouvoir procéder à un indexation pertinente, il faut tout d'abord éliminer les mots qui ont le moins de spécification, le moins de significativité, c'est à dire ceux qui sont régulièrement présents dans les articles et en grande quantité. Pour pouvoir les identifier, nous allons les classer selon le critère "$tf_{i,j} \times idf_i$". Ce critère permet de quantifier la pertinence d'un mot "i" dans un fichier "j". Il se décompose en deux parties. La partie "idf_i" représente une grandeur exprimant sa présence dans tous les fichiers. La partie représente sa présence dans le fichier "j". Le produit des deux permet de donner une bonne appréciation de la valeur du mot "i" dans le fichier "j".

Tout d'abord, calculons le "df_i" qui permet ensuite d'obtenir "idf_i" avec la formule $idf_i = log\frac{N}{dfi}$.

Il représente le nombre de fichiers dans lequel le mot "i" apparaît. Il est possible de le faire simplement à l'aide de commandes UNIX. Avant de le calculer nous utilisons le fichier "segmente_TT.pl" fourni sur le site de l'UV pour séparer tous les mots présents dans le titre et le texte d'un article, en les plaçant chacun sur une ligne différente. Nous devons ensuite éliminer les doublons au sein d'un même article (étant donné que l'on recherche uniquement le nombre d'articles où le mot apparaît) et sommer les occurrences restantes.

```
LANG=C
cat corpus.XML | perl segmente_TT.pl -f | sort | uniq -c > result2.txt
awk '$2 ~/^[0.9]/ {next} {print}' result2.txt > ./temp/result.txt
rm -rf result2.txt
awk '{print $2}' ./temp/result.txt | uniq -c > ./temp/DFi.txt
```

Le fichier obtenu est très grand au vu du corpus (~14000) ce qui logique car comme il contient beaucoup d'articles scientifiques, il est normal de trouver une grande variété de mots.

Le fichier "DFi.txt" obtenu contient deux colonnes, la première contient le "df_i" et la seconde le mot "i".

Ex :

```
9571        1 peinture
9572        3 peintures
9573        1 pelco
9574        1 peltier
9575        1 penant
```

Nous pouvons maintenant calculer “idf_i“ en remplaçant la valeur dans la première colonne par la formule énoncée plus haut. Cette fois-ci il est plus difficile de le faire à l’aide du SHELL, nous utilisons donc un nouveau script Perl “calcul_idfi.pl”.

```
while(<>){
  chomp;
  if(/  ([0-9]+?) [0-9].?/){
    $_ =~ s/  [0-9]+? /0 /;
  } else {
    if(/  ([0-9]+?) (.+)/){
      $dfi = $1;
    }
    $idfi = log10($taille/$dfi);
    $_ =~ s/  [0-9]+? /$idfi /;
  }
  $texte = $texte."$_\n";
}
```

Ce script va parcourir le fichier “DFi.txt” ligne par ligne et va récupérer la valeur “DFi” grâce à l’expression régulière “([0-9]+?)” pour la stocker dans la variable “$dfi”. Le nombre de fichiers est stocké dans “$taille” et le calcul de “idf_i“ est stocké dans “$idfi”. La ligne qui utilise la variable “$_” permet de remplacer “DFi” par “idf_i“. “$texte” est notre variable de sortie, on lui concatène donc tous les résultats les uns à la suite des autres.

Il y a dans ce script une ligne qui peut paraître surprenante au premier abord :
“if(/ ([0-9]+?) [0-9].?/)”. Cette expression régulière permet de *match* les mots du texte et du titre qui commence par un chiffre. En effet il y a quelques mots dans le corpus tels que “1er” ou “105” qui auront une valeur de “idf_i“ forte car ils sont peu utilisés dans un article et peu dans le corpus, alors qu’ils ne contiennent aucune information. Cette partie du code permet de fixer la valeur de “idf_i“ à 0 pour ce type de mots afin qu’ils aillent directement dans la *stoplist*.

Maintenant que pour chaque mot “i” du lexique un “idf_i“ lui a été associé, il faut revenir à la liste des occurrences des mots par fichier et multiplier cette valeur par le “idf_i“ du mot. Pour ce faire nous allons utiliser un autre script Perl “calcul_tfIdf.pl”.

```
while(<FIC1>){
  if(/([0-9]\.[0-9]+?) .+/){
    $val = $1;
  }
  $cle = $_;
  $cle =~ s/([0-9]\.[0-9]*)//;
  $cle =~ s/\n//;
  $val =~ s/\s+//;
  $cle =~ s/\s+//;

  $hash{$cle} = $val;
}

while(<FIC2>){
  chomp;
  if(/([0-9]+?)\s(.+?)\s([0-9]*\.htm)/){
    $val = $1;
    $mot = $2;
    $fichier = $3;
  }
  $val = $hash{$mot}*$val;
  $texte = $texte."$val    $mot    $fichier\n";
}
```

Ce script va tout d’abord créer un dictionnaire, *hash* sous Perl, associant le “idf_i“ à sa clé le mot “i”. Ensuite il va parcourir toutes les lignes du fichier contenant les occurrences des mots dans chaque fichier et va remplacer le nombre d'occurrence par son produit avec le “idf_i“ associé.

L’opérateur “<FIC1>” permet de réaliser la même boucle que l’opérateur “<>” mais celà sur un fichier que l’on a ouvert directement dans le script et sauvegardé sous le nom “FIC1”. Ainsi nous commençons par stocker la valeur “idf_i“ dans la variable “$val”. La clé est stockée dans la variable “$cle” en extrayant le mot à partir de la ligne actuelle (contenue dans “$_”) et en en retirant la valeur présente avant. On s’assure aussi de retirer les séparateurs qui auraient pu rester dans les variables. Le *hash* est ensuite rempli avec “$hash{$cle} = $val” qui va créer l’entrée de la valeur de “$cle” dans le *hash* et lui associer sa valeur contenue dans “$val”.

La seconde boucle s'effectue dans le fichier que l'on avait obtenu au cours du script "DFi.sh" alors appellé "result.txt". Nous allons identifier chaque composante qu'il y a sur une ligne, le nombre d'occurence du mot "i" dans le fichier "j" dans la variable "$val", le mot "i" dans la variable "$mot" et enfin le nom du fichier "j" dans la variable "$fichier". On affecte ensuite à la variable "$val" la valeur du "$tf_{i,j} \times idf_i$" et on concatène "$val", "$mot" et "$fichier" dans la variable "$texte" qui est notre variable de retour. Nous stockons le résultat dans le fichier "tfIdf.txt".

Nous avons maintenant enfin un moyen de décider quels mots doivent être retirés du corpus. Mais le contenu du fichier "tfIdf.txt" n'est pas utilisable tel quel. En effet si nous avions décidé d'utiliser ces informations telle quel pour décider de quels mots retirer, nous aurions alors ordonné le fichier dans l'ordre croissant des "$tf_{i,j} \times idf_i$" et choisi un seuil en dessous duquel le mot est alors exclu. Mais cette façon de faire ne prends pas en compte qu'un mot peut être faiblement significatif dans un article mais fortement dans un autre ! A partir du moment ou pour un fichier la valeur se trouve en dessous du seuil, le mot est éliminé peu importe ses autres valeurs.

Pour palier ce problème, nous allons donc travailler sur la moyenne des "$tf_{i,j} \times idf_i$", qui permet de bien approcher la véritable "utilité" de garder le mot dans le corpus. En effet un mot qui est régulièrement peu significatif et lorsqu'il l'est plus ne l'est pas si fortement aura une moyenne faible ce qui correspond bien à son manque de significativité. En revanche, un mot dont certaines valeur sont extrêmement faible mais qui possède des occurrences de valeurs plus élevées aura une moyenne qui sera tirée vers le haut, ce qui lui permettra de dépasser le seuil s'il est en moyenne plus significatif.

Nous allons donc tout commencer par calculer les moyennes pour chaque mot et ensuite les ordonner. Cette partie est réalisée grâce à un script Perl pour calculer la moyenne d'un mot, et de la commande UNIX "sort" sur le résultat pour l'ordonner. Etant donné que le fichier "tfIdf.txt" est ordonné alphabétiquement, c'est à dire que toutes les occurrences d'un même mot sont regroupées, il suffit de faire une simple boucle while qui lit toutes les lignes, et tant que nous rencontrons le même mot, nous incrémentons notre compte d'occurrences et sommons les "$tf_{i,j} \times idf_i$", jusqu'à ce que le mot change et à ce moment là on calcule la moyenne et nous la stockons. Une fois tout le fichier parcouru, une utilisation de la fonction "sort" et il est maintenant possible de décider d'un seuil de tolérance pour établir la *stoplist*.

Nous avons choisi de mettre le seuil à 0.762372386508878 qui correspond au mot autre.

```
0.749592458736717          ne
0.753680751957319          domaine
0.754049267902694          cela
0.755154685027757          depuis
0.757721789518788          autour
0.757721789518788          centre
0.75807524033542          déjà
0.758780576088405          permet
0.759509821529089          chercheurs
0.761490751732538          y
0.762329619243382          laquelle
0.762372386508878          autre
0.764947712048936          contexte
0.765199287576116          terme
0.76586362767688          afin
0.767139336813699          tous
0.774838990908871          précise
0.777912706671894          non
0.778119597689883          trois
0.77982747657292          particulier
```

En effet, les mots présents avant n'ont soit aucune valeur d'information (verbes de narration, particules etc), soit sont trop généraux pour notre corpus d'articles scientifiques comme "chercheurs" ou "domaine". On peut remarquer que le mot "non" apparaît après et nous savons qu'il n'as pas de vrai valeur, mais le rajouter nous inciterait soit à abandonner les mots "terme" et "contexte" qui peuvent être utiles, soit à augmenter le seuil et mettre des exceptions pour certains mots ce qui représente une dose de travail trop grande pour une différence qui ne sera pas drastique sur la qualité de la recherche (excepté si un être humain décide manuellement de tous les mots à garder mais dans ce cas là l'étude menée n'aura servie à rien).

Il suffit maintenant de parcourir notre fichier "tfIdf.txt" et de récupérer les mots avec une valeur associée inférieure à 0.762372386508878 et les écrire dans un fichier "stoplist.pl". Une fois ce fichier créé, nous pouvons maintenant épurer notre corpus. Nous allons donc créer un filtre à partir de cette *stoplist* qui permettra à chaque fois qu'un de ces mots est rencontré, de le remplacer par du vide et ainsi l'éliminer du texte. Pour réaliser celà nous allons utiliser le script

“filtreCorpus.pl” généré par le script “newcreefiltre.pl” fourni sur le site de l’UV. Ce dernier permet à partir d’un flux de données à deux colonnes (ou une colonne et ainsi la seconde sera considérée comme étant vide) de générer un script Perl qui va utiliser des expression régulières pour *match* les mots contenu dans la première colonne et le remplacer par ceux contenus dans la seconde.

Une fois le script généré, nous l’appliquons sur le corpus pour obtenir une épuration des textes et des titres des articles.

2. Création les lemmes associés au reste des mots

Maintenant que notre corpus est épuré, il ne contient plus (en majorité) que des mots intéressant qui permettent d’effectuer une recherche sur un thème. Afin de pouvoir donner suite à une recherche en langage naturel, il est nécessaire de définir des méthodes pour appliquer une proximité entre des mots proches. En effet un mot au singulier, au féminin et au pluriel représentent exactement la même chose et doivent être retournée en réponse à la recherche. Pour ce faire nous allons associer un lemme à chaque mot, ce qui va permettre de représenter les mots proches par le même mot et faciliter la recherche car il y aura moins de données dans la base de mots.

Tout d’abord, récupérons la liste des mots présents dans le corpus. Nous possédons déjà une liste ordonnée des mots du corpus selon les valeurs des “$tf_{i,j} \times idf_i$” croissantes. Il suffit d’en extraires les mots dont la significativité est avérée, c’est à dire au dessus du seuil que nous nous sommes fixé. Les mots du corpus seront ordonnés dans l’ordre alphabétique pour procéder à la lemmisation avec une complexité modérée, et stockés dans un fichier “motsCorpus.txt”, en une seule colonne.

Ensuite nous allons utiliser un algorithme de troncation pour permettre de préparer la lemmisation. Cette étape va associer à chaque mot une suite de chiffres correspondant au nombre de lettre différentes qu’il existe parmis les mots possédant le même préfixe. Pour ce faire nous allons utiliser le script “successeur_P16.pl” qui permet de faire ce calcul de façon optimisée. Nous fournissons à ce script le fichier “motsCorpus.txt” en entrée et nous stockons la sortie dans le fichier “successeurs.txt”. Ce fichier se compose de deux colonnes, dans la première se situe les successeurs (les suites de chiffres) et dans la seconde le mot originel.

Nous utilisons ensuite un autre algorithme fourni sous la forme du script “filtronc_P16.pl”. Ce script prend en entrée un flux de donnée séparé en deux colonnes du même type que la sortie de “successeurs_P16.pl”. Il va permettre d’associer à tous les mots du flux le lemme calculé avec le contenu du successeur associé. La sortie va donc nous permettre d’avoir en premier les mots du corpus et de les associer directement avec leur lemme. Nous pouvons ainsi réutiliser le script “newcreeFiltre.pl” afin, de nouveau, modifier notre fichier corpus.XML pour qu’il ne contienne que des lemmes. Nous donnons la sortie de “filtronc_P16.pl” à “newcreeFiltre.pl” et nous obtenons le script à appliquer au corpus pour qu’il soit prêt à être utilisable pour former les tables inverses pour l’indexation.

3. Création des tables inverses

Maintenant que le contenu du corpus est fixe et est utilisable pour l’indexation, il ne nous reste plus qu’à générer des listes d’affectations, ou liste d’index inverse. Ces listes vont permettre d’associer à un terme du lexique toutes ses apparitions dans le corpus, ce qui permet d’accéder rapidement à cette liste d’apparition lorsque l’on recherche un terme donné.

Nous allons tout d’abord commencer par créer la table sur le titre et le contenu de l’article. Pour cela nous allons utiliser de nouveau le script “segmente_TT.pl” sur le corpus constitué de lemmes pour lier chaque apparition des lemmes à leur rubrique, ainsi qu’à leur fichier ainsi qu’au numéro de bulletin. Nous utilisons alors toutes les options passables en argument au script pour afficher toutes ces variables. Maintenant que toutes les apparitions de chaque lemme sont listées, nous devons les “regrouper”, avoir dans la table une seule occurrence du lemme suivi par une liste de “rubrique fichier numéro”. Pour ce faire nous utilisons le script “indexTexte.pl” qui effectue cela en utilisant un hash Perl intermédiaire pour associer à une clé unique (le lemme) toutes les valeurs de “rubrique fichier numéro” qui lui sont associées. Il suffit ensuite d’afficher la clé suivi de toutes les valeurs associées avant de passer à une autre.

Ensuite nous allons créer une table inverse des dates et une table inverse des rubriques. Cette fois-ci nous utilisons le script “index.pl” fourni sur le site. Le fonctionnement de ce script est différent de celui sur le texte. Il va se baser sur une balise (ici <date> et <rubrique>) et va utiliser la chaîne qui était contenue entre les balises comme terme (même si elle est composée de

plusieurs mots) et va effectuer la même action que “indexTexte.pl”, c’est à dire écrire pour un terme une liste de “fichier numéro” à la suite pour permettre de lister tous les fichiers et leur numéro de corpus contenant ce terme.

Ainsi en effectuant ce travail sur les balises titre et texte, date et rubrique notre recherche sera plus fournie et en plus de pouvoir nous donner les articles où figurent des occurrences d’un mot, nous pourrons récupérer un bulletin dont on connaît le numéro, ou les articles pour une date.