M1 Données et Connaissances

Recherche d'Information

Rapport de TP 1 à 5

19 octobre 2019

Table des matières

Informations

Voici le rapport qui présente notre le travail que nous avons réalisé pour les 5 premiers sujet de TP. Ce travail a été réalisé par **BARDY Benjamin** et **ORTIZ Diégo**. Vous pourrez retrouver les différents codes sources sur notre notre repository github.

TP1 : Indexation d'un corpus des documents textes avec Lucene

	Classes	Méthodes
Lecture du CSV	CSVParser	CSVRecord, CSVFormat
Indexation	IndexWriter, IndexCollection, IndexWriterConfig	index, indexDoc
Exécution de la requête	Query Simple	process, query

3.2.

3.3. — WhitespaceAnalyzer : Crée les tokens en utilisant les espaces comme séparateurs.

— SimpleAnalyzer : Crée les tokens en utilisant les espaces et les caractères spéciaux comme séparateurs, puis met chaque token en minuscule

— StopAnalyzer : Variation du SimpleAnalyzer, qui enlève les mots les plus courants (en anglais, « a », « the »,...). La liste de mots à enlever peut être passée en paramètre.

— StandardAnalyzer : Basé sur StopAnalyzer. Il peut aussi reconnaître les adresses mails, les noms de domaines, ou encore les noms de compagnie comme un seul token.

— KeywordAnalyzer : Le texte entier est compté comme un seul token. Utile pour les identifiants par exemple

— LanguageAnalyzer (ex : EnglishAnalyzer, FrenchAnalyzer) : Il existe des Analyzers spécifiques à la plupart des langues, car les règles de syntaxe et la grammaire changent d'une langue à l'autre.

— CustomAnalyzer : Il est possible de créer son propre Analyzer, avec ses propres critères pour créer les tokens.

Il existe plusieurs types d'analyseurs et ce pour plusieurs raisons. Tout d'abord, il faut

rappeler que le but est d'identifier les unités significatives linguistiques dans un texte. Cela dit ces « unités » ou tokens, peuvent dépendre de la langue du texte. Par exemple en Français il est assez facile de séparer ces tokens via la ponctuation ou les espaces (WhitespaceAnalyzer, SimpleAnalyzer). En revanche en Allemand la césure d'un token ne peut pas être basé sur des espaces. Un dernier exemple : l'arabe, qui se lit de droite à gauche pour les mots mais de gauche à droite pour les chiffres et ne possède pas l'alphabet latin ; pareil pour le chinois (SmartChineseAnalyser) où la signification des symboles n'a rien à voir avec celles de l'alphabet latin. Ainsi il est nécessaire d'avoir plusieurs types d'analyseurs car chaque langue a ses propres spécificités qui peuvent grandement influencer le processus de tokenisation, de normalisation, de filtrage de mot vide et de racinisation.

3.4. TextField est un champ tokenizé et indexé, il est donc possible de l'analyser. On peut le remplacer par StringField qui ne peut pas être analysé (c'est une simple chaîne de caractères)

5. **Questions ouvertes**

5.1. Lucene est une librairie open source en Java qui permet d'indexer et d'effectuer des recherches. Quant à Solr, elle n'est pas fondamentalement différente de Lucene. En effet Solr est un outil logiciel de moteur de recherche. A la spécificité que Solr utilise la librairie Lucene mais ne fait pas que ça, en effet Solr est un outil plus haut niveau qui ajoute des fonctionnalités comme la recherche Géospatiale, la recherche à facettes, une interface administration web etc ...

5.2. Les fichiers d'indexation sont dans le dossier indexRI. Ils sont nécessaires pour pouvoir ensuite calculer le score de pertinence de la requête suivant l'algorithme donné (que ce soit TF-IDF ou autre)

TP2 : Méthodes de pondération sur Lucene

4.1. Avec la requête 'title :France'

— Résultats avec BM25 :

1. Template :Location map France Île-de-France
2. France
3. Brest (France)
4. Versailles, France
5. Metropolitan France

— Résultats avec BM11 :

1. France
2. Brest (France)
3. Versailles, France
4. Metropolitan France
5. Anatole France

— Résultats avec BM15 :

1. Template :Location map France Île-de-France
2. Louis XV of France
3. Template :Tour de France Polka Dot Jersey
4. Category :Lists of communes of France

5. Category :Aviation in France

Les différents poids permettent de chercher des données de façon différentes en combinant les termes de manière différente.

Ici, on voit que BM11 n'a pas de titre contenant 'Template :' ou encore 'Category'. BM15, en revanche, a l'air de favoriser les titres plus longs.

5. Avec la requête "the white house" :

   — Résultats avec TFTotal IDFTotal : Found 5 hits of 7418.

   1. Wikipedia :Requests for deletion/Requests/2009/Captain Fantastic Faster Than Superman Spiderman Batman Wolverine Hulk And The Flash Combined
   2. R50/53 : Very toxic to aquatic organisms, may cause long-term adverse effects in the aquatic environment
   3. R52/53 : Harmful to aquatic organisms, may cause long-term adverse effects in the aquatic environment
   4. List of revocations of appointments to orders and awarded decorations and medals of the United Kingdom
   5. R51/53 : Toxic to aquatic organisms, may cause long-term adverse effects in the aquatic environment

   — Résultats avec TFTotal IDFSum : Found 5 hits of 26.

   1. Category :Episode list using the default LineColor
   2. Animal Crossing : Let's go to the city !

3. This Is What the Truth Feels Like
4. The Closer I Get to You
5. Wikipedia :Requests for oversightship/The Rambling Man

— Résultats avec TFTotal IDFSumSmooth : Found 5 hits of 7843.

1. The White House
2. White House
3. The White House plumbers
4. White House Plumbers
5. White House Down

— Résultats avec TFTotal IDFBir : Found 5 hits of 26.

1. Category :Episode list using the default LineColor
2. Animal Crossing : Let's go to the city !
3. This Is What the Truth Feels Like
4. The Closer I Get to You
5. Wikipedia :Requests for oversightship/The Rambling Man

— Résultats avec TFTotal IDFBirSmooth : Found 5 hits of 7843.

1. The White House
2. White House
3. The White House plumbers
4. White House Plumbers

5. White House Down

— Résultats avec TFLog IDFSum : Found 5 hits of 26.

1. Category :Episode list using the default LineColor
2. Animal Crossing : Let's go to the city !
3. This Is What the Truth Feels Like
4. The Closer I Get to You
5. Wikipedia :Requests for oversightship/The Rambling Man

— Résultats avec TFLog IDFSumSmooth : Found 5 hits of 7771.

1. The White House
2. White House
3. The White House plumbers
4. White House Plumbers
5. White House Down

— Résultats avec TFLog IDFBir : Found 5 hits of 26.

1. Category :Episode list using the default LineColor
2. Animal Crossing : Let's go to the city !
3. This Is What the Truth Feels Like
4. The Closer I Get to You
5. Wikipedia :Requests for oversightship/The Rambling Man

— Résultats avec TFLog IDFBirSmooth : Found 5 hits of 7771.

1. The White House
2. White House
3. The White House plumbers
4. White House Plumbers
5. White House Down

TFTotal IDFTotal préfère les titres qui sont plus longs.Il semblerait que les combinaisons de TF/IDF les plus efficaces soient les suivantes : TF Total IDF bir smooth et TFLog IDF Sum smooth.

7. Si on prend l'exemple d'une requête sur les titres avec « white house » contre une requête sur le texte, il apparaît que avec une recherche sur les textes on obtient un nombre de « hits » bien plus important (avec Tflog IDFsum smooth on obtient 7771 hits contre 64706 avec une recherche sur le texte). On remarque aussi que la qualité des meilleurs résultats s'affaiblie. En effet avec TF Total IFD Sum smooth avec une recherche sur les titres ont obtient le résultats suivant pour « the white house » : 1. The White House 2. White House 3. The White House plumbers 4. White House Plumbers 5. White House Down

   Alors que avec une recherche sur le texte on obtient un résultat bien moins pertinent : 1. Template :BillboardEncode/T 2. Survivor : All-Stars 3. Survivor : Palau 4. Survivor : The Amazon 5. Template :BillboardID/T

TP3 : Application de la RI pour la liaison référentielle

1-2.

	Bayern Munich	Mario (Gomez ou Götze)	Thomas Müller
BM11	18	134	437
TFtotal, IDFSum-Smooth	33	55	/
TFTotal, IDFBir	352	/	/
TFTotal, IDFBirSmooth	16	55	/
TFLog, IDFSum	184	/	/
TFLog, IDFSum-Smooth	29	52	/
TFLog, IDFBir	13	52	/
TFLog, IDFBirSmooth	29	58	/
BM25	17	/	/

On remarque que notre système a beaucoup de mal a retrouver la combinaison des 3 mots dans le texte. On peut aussi remarquer que "Bayern Munich" ainsi que "Mario Götze" se retrouvent très facilement et ce avec pratiquement toutes les variations de TF/IDF. On peut noter que les variations les plus efficaces sont TFLog IDFBirSmooth, TFLog IDFBir et TFTotal IDFBirSmooth ; chose que l'on avait déjà remarqué avec les résultats du TP2.

3.

	Bayern Munich (seul)	**Mario (Gomez ou Götze)** (seul)	**Thomas Müller** (seul)
TFLog IDFBir	4	25	138
BM25	2	9	190
TFTotal IDFTotal	6	7	75
TFLog IDFSum	2	25	162
TFLog IDFBirSmooth	8	9	190

On remarque le rang des différents mots clés et nettement moins élevé, ce qui est tout à fait normal. Une différence marquante est qu'on obtient tout nos mots clés et ce, toujours dans les 10 premiers résultats (rang). Cependant, "Thomas Müller" est toujours loin dans les différentes query et ce, quelque soit la variante TF/IDF.

4. Si on utilise le champ title avec nos 3 mots clés en tant que titre on ne retrouve aucun résultat. En effet après vérification il n'existe aucune page sur la version "simple" de Wikipedia qui contient en son titre à la fois "Thomas", "Munich" et "Mario" (dans le cas ou on spécifie "BooleanClause.Occur.MUST").

5. Pour l'exercice 5 nous avons réalisé une mini application pour mieux illustrer notre moteur de recherche miniature sur Wikipedia Simple. Pour faire cela nous avons utilisé des JSP ainsi qu'un servlet le tout en local avec TomCat. Le code source est disponible ici et nous avons même réalisé une courte **vidéo de présentation** étant donné que cela peut prendre du temps d'installer toutes les dépendances pour faire fonctionner l'application.

TP4 : PageRank

2. 5 pages sont utilisées et on réalise 3 itérations ;

3. Le programme se déroule de la manière suivante :

   — Premièrement, on donne à chaque page le même PageRank, qui vaut :

   $$PageRank_{INIT} = \frac{1}{\text{nombre total de pages}}$$

   Dans l'exemple, on utilise 5 pages.

   — On choisit le nombre d'itérations de calcul du PageRankà réaliser. Dans l'exemple, on réalise 2 itérations.

   — A chaque itération :

   (i) On commence par sauvegarder tous les PageRank

   (ii) Pour chaque lien entre deux noeuds e (le noeud entrant) et s (le noeud sortant), on compte le nombre de liens sortants de b grâce à la matrice d'adjacence

   (iii) On calcule ensuite le PageRank du noeud e :

   $$PageRank(e) = PageRank(s) * \frac{1}{\text{nombre de liens sortants de s}}$$

   — Puis, on applique le Damping Factor (généralement fixé à 0.85) au PageRank de chaque page p :

   $$PageRank_{FINAL}(p) = (1 - DampingFactor) + DampingFactor * PageRank(p)$$

   — Finalement, on affiche le PageRank de chaque page.

5. Je ne vais bien évidemment pas afficher le PageRank des 100 pages, mais on peut malgré tout regrouper les pages par score (arrondis à 5 chiffres après la virgule) :

PageRank	Pages
0.16847	30
0.09106	85
0.05304	71
0.01366	31, 35, 52, 77, 79
0.01052	88, 91
0.00738	Toutes les autres pages

Ainsi, les 3 pages les plus pertinentes sont 30, 85 et 71.

Question 5 partie bonus Nous avons aussi réalisé la question 5 après avoir généré via un script Python (voir le package tp4_python) la matrice d'adjacence correspondant au fichier AdjencyMatrix100.txt et après avoir lancé PageRank avec cette nouvelle matrice on obtient le même ordre qu'avec WebGraph (avec PageRank.java on obtient un score de 0.312 pour le meilleur résutlat contre 0.17 avec WebGraph ; on retrouve heuresement le même document le plus pertitant : le numéro 30).
On notera qu'on a aussi essayé de créer la matrice d'adjacence avec notre script python, malheureusement (et évidemment) le fichier est trop gros et on a une "MemoryError").

6. Il y a énormément plus de données ici (599330 pages au total), donc on va représenter les scores sous forme d'intervalles :

PageRank	Pages
Plus de 0.01	462979, 274143, 47778
Entre 0.01 et 0.008	10277
Entre 0.008 et 0.006	18413, 319119
Entre 0.006 et 0.004	4441, 432517, 318636, 123095, 70147
Entre 0.004 et 0.002	52889, 348825, 198442, 565840, 426394, 127328, 27501, 385461, 318293, 110252, 227168, 2023, 593888, 177917, 302538, 396899, 250005, 274619, 300750, 242546, 255486, 128702, 300927, 126767, 12656
Entre 0.002 et 0.001	455687, 195556, 259377, 564655, 341509, 96487, 200328, 13825, 922, 140194, 230602, 472929, 195379, 554, 102392, 146824, 176790, 182774, 247915, 219588, 181420, 197210, 135243, 583808, 483917, 424409, 189925, 247941, 510904, 517991, 364252, 183355, 318972, 119289, 165662, 183819, 188742, 510906, 250380, 28475, 262882
Moins de 0.001	Toutes les autres pages

Bien qu'il y ait quasiment 600000 pages en tout, il n'y en a que 77 qui ont un score supérieur à 0.001. Les 3 pages qui se démarquent vraiment sont les pages 462979 (score : 0.01206), 274143 (score : 0.01062) et 47778 (score : 0.1031).

TP5 : Évaluation d'un système de recherche d'information

3. *Pour cette partie, nous générons 10 résultats par mot-clés (soit les mots de la requête en majuscules), ce qui nous fait 140 résultats au total. D'après les tests, il y a aussi 15 documents jugés pertinents.*

Métriques	$TF_{total}IDF_{total}$	$TF_{log}IDF_{sum,smooth}$	$TF_{total}IDF_{BIR,smooth}$	BM25	BM11
num_rel_ret	1	9	7	5	7
map	0.0714	0.2728	0.1750	0.2245	0.2477
gm_ap	0.0000	0.0079	0.0017	0.0005	0.0019
R-prec	0.0714	0.1786	0.1071	0.1786	0.1786
bpref	0.0714	0.6071	0.4643	0.3214	0.4643
recip_rank	0.0714	0.2907	0.1929	0.2602	0.2834
ircl_prn.0.00	0.0714	0.2907	0.1929	0.2602	0.2834
ircl_prn.0.10	0.0714	0.2907	0.1929	0.2602	0.2834
ircl_prn.0.20	0.0714	0.2907	0.1929	0.2602	0.2834
ircl_prn.0.30	0.0714	0.2907	0.1929	0.2602	0.2834
ircl_prn.0.40	0.0714	0.2907	0.1929	0.2602	0.2834
ircl_prn.0.50	0.0714	0.2907	0.1929	0.2602	0.2834
ircl_prn.0.60	0.0714	0.2550	0.1571	0.1888	0.2120
ircl_prn.0.70	0.0714	0.2550	0.1571	0.1888	0.2120
ircl_prn.0.80	0.0714	0.2550	0.1571	0.1888	0.2120
ircl_prn.0.90	0.0714	0.2550	0.1571	0.1888	0.2120
ircl_prn.1.00	0.0714	0.2550	0.1571	0.1888	0.2120
P5	0.0143	0.1000	0.0857	0.0571	0.0714
P10	0.0071	0.0643	0.0500	0.0357	0.0500
P15	0.0048	0.0429	0.0333	0.0238	0.0333
P20	0.0036	0.0321	0.0250	0.0179	0.0250
P30	0.0024	0.0214	0.0167	0.0119	0.0167
P100	0.0007	0.0064	0.0050	0.0036	0.0050
P200	0.0004	0.0032	0.0025	0.0018	0.0025
P500	0.0001	0.0013	0.0010	0.0007	0.0010
P1000	0.0001	0.0006	0.0005	0.0004	0.0005

Sur le tableau, on peut voir que l'algorithme $TF_{log}IDF_{sum,smooth}$ est de loin le meilleur :

- Sur les 15 documents pertinents, c'est l'algorithme de qui en a retrouvé le plus avec 9.
- Sa moyenne des précisions est la plus élevée
- Son rappel interpolé est dès le départ le meilleur (et rappelle donc plus d'autres pages

pertinentes que les autres)

- Ses scores de précisions sont les plus élevés de tous

On peut aussi noter que $TF_{total}IDF_{total}$ ne marche absolument pas, et aussi que même si au départ $TF_{total}IDF_{BIR,smooth}$ a un map et des rappels interpolés plus faibles que BM11, il possède une meilleure précision sur ses 5 premiers résultats, et les deux sont équivalent pour le reste.

4-5. *Dans le même esprit que la question précédente, nous générons 100 résultats par mot-clés (soit les mots de la requête en majuscules), ce qui nous fait 1400 résultats au total.*

Métriques	$TF_{total}IDF_{total}$	$TF_{log}IDF_{sum,smooth}$	$TF_{total}IDF_{BIR,smooth}$	BM25	BM11
num_rel_ret	2	11	11	10	8
map	0.0729	0.3112	0.1884	0.2384	0.2496
gm_ap	0.0000	0.0173	0.0104	0.0087	0.0034
R-prec	0.0714	0.1786	0.1071	0.1786	0.1786
bpref	0.1429	0.7143	0.7143	0.6786	0.5357
recip_rank	0.0729	0.3264	0.2039	0.2741	0.2853
ircl_prn.0.00	0.0729	0.3264	0.2039	0.2741	0.2853
ircl_prn.0.10	0.0729	0.3264	0.2039	0.2741	0.2853
ircl_prn.0.20	0.0729	0.3264	0.2039	0.2741	0.2853
ircl_prn.0.30	0.0729	0.3264	0.2039	0.2741	0.2853
ircl_prn.0.40	0.0729	0.3264	0.2039	0.2741	0.2853
ircl_prn.0.50	0.0729	0.3264	0.2039	0.2741	0.2853
ircl_prn.0.60	0.0729	0.2960	0.1730	0.2026	0.2139
ircl_prn.0.70	0.0729	0.2960	0.1730	0.2026	0.2139
ircl_prn.0.80	0.0729	0.2960	0.1730	0.2026	0.2139
ircl_prn.0.90	0.0729	0.2960	0.1730	0.2026	0.2139
ircl_prn.1.00	0.0729	0.2960	0.1730	0.2026	0.2139
P5	0.0143	0.1143	0.0857	0.0571	0.0714
P10	0.0071	0.0714	0.0500	0.0357	0.0500
P15	0.0048	0.0476	0.0381	0.0286	0.0333
P20	0.0036	0.0357	0.0286	0.0250	0.0250
P30	0.0024	0.0262	0.0238	0.0190	0.0167
P100	0.0014	0.0079	0.0079	0.0071	0.0057
P200	0.0007	0.0039	0.0039	0.0036	0.0029
P500	0.0003	0.0016	0.0016	0.0014	0.0011
P1000	0.0001	0.0008	0.0008	0.0007	0.0006

Logiquement, tous les algorithmes ont réussi à trouver plus de documents pertinents.

Les métriques de chaque algorithme augmentent légèrement ou sont équivalentes à celles de la question précédente à part pour un algo : $TF_{log}IDF_{sum,smooth}$. Il arrive à être encore meilleur !

- Sa moyenne des précisions a visiblement augmenté

- Son rappel interpolé est encore loin devant les autres
- Ses scores de précisions sont encore meilleurs

On peut voir aussi que l'écart entre BM25 et BM11 se resserre doucement. Notamment, bien que toutes se métriques soient inférieures à celles de BM11, BM25 a réussi à trouver plus de documents pertinents que BM11, avec 10 contre 8.

On peut ainsi en conclure que de tous les algorithmes testés, l'algorithme de $TF_{log}IDF_{sum,smooth}$ est dans notre cas le meilleur et le plus performant.

Sources

1. Code source sur github de tous les TP : https://github.com/BenjiX34/TP_RI/
2. Code source sur github de l'application du TP3 : https://github.com/Diegoortizz/TP3RI
3. Vidéo youtube qui présente l'application du TP3 : https://www.youtube.com/watch?v=AGtuqIxLwKY&feature=youtu.be