**PROJET DE RECHERCHE D’INFORMATION**

TABLE DES MATIERES

[ARCHITECTURE GENERALE 1](#_Toc506294242)

[INDEXATION DE LA COLLECTION 1](#_Toc506294243)

[RECHERCHE BOOLEENNE 2](#_Toc506294244)

[RECHERCHE VECTORIELLE 3](#_Toc506294245)

[PARTIE I : TRAITEMENT LINGUISTIQUE 3](#_Toc506294246)

[PARTIE II : EVALUATION POUR LA COLLECTION CACM 5](#_Toc506294247)

[MESURES DE PERFORMANCE 5](#_Toc506294248)

[MESURES DE PERTINENCE 5](#_Toc506294249)

# ARCHITECTURE GENERALE

Les données de CACM (cacm.all, common\_words, qrels.text, query.text) sont stockées dans le dossier data.

Les données retraitées sont dans le dossier data\_clean.

Les modules intermédiaires sont précédés par M\_ , ceux qui produisent des résultats sont précédés par R\_ , et ceux qui servent à analyser le code sont précédés de A\_ .

## INDEXATION DE LA COLLECTION

Afin d’indexer ces données, on utilise le module **R\_indexation.py**. J’utilise le concept de map-reduce. Il y a plusieurs étapes dans ce processus :

1. La fonction « create\_collection » : elle crée la collection de documents, de classe *Documents* (créée dans le module **M\_Documents.py**), et une liste de textes prêts à être indexés, et les enregistre dans le dossier clean\_data.
2. La fonction « create\_index » : elle crée l’index inversé. Pour chaque document de CACM\_collection\_txt.json, elle fait appel à la fonction « tokenize\_tf », puis une fois tous les documents tokenizés, elle appelle la fonction « aggregate\_idf » qui aggrège les données en un index.
   1. La fonction « tokenize\_tf » (map) : elle prend en entrée le texte d’un document et son doc ID. La fonction va lire chaque mot, si ce n’est pas un stopword ou un nombre, elle va le lemmatiser puis l’ajouter à la liste des keywords. Ensuite j’utilise la fonction FreqDist qui renvoie le nombre d’occurrence par mot. Enfin je crée un liste composée d’éléments de la forme (terme, (docID, term\_freq)) avec term\_freq =
   2. La fonction « aggregate\_idf » (reduce) : elle prend en entrée la liste complète de mots significatifs lemmatisés construite par tokenize\_tf, et crée un nouveau dictionnaire qui agrège pour chaque terme ses occurrences dans les différents documents. Ensuite elle repasse sur ce nouveau dictionnaire afin de calculer l’IDF. La fonction renvoie une liste d’éléments de la forme : (terme, collection\_freq, [posting\_list: (docID, tf-idf)]) avec tf-idf de la forme

## RECHERCHE BOOLEENNE

Module concernés : **M\_boolean\_treebuilder.py** et **R\_Research.py**

Afin de répondre à des requêtes booléennes du type "master and not (algorithmic or access)", j’ai créé un parser de requêtes booléennes sous forme d’arbre, inspiré de la série d’article du [blog de Ruslan](https://ruslanspivak.com) « Let’s Build A Simple Interpreter »

J’ai défini des classes pour parser la requête :

* Token : pour chaque entité (i.e. : ‘(‘, mot, ‘AND’….) de la requête afin de lui associer une action
* Lexer : pour prétraiter le texte de la requête et associer à chaque entité une action en utilisant des instances de la classe token

Des classes pour créer l’arbre de recherche :

* BinOp : classe de nœuds binaires pour les tokens AND et OR
* UnaryOp : classe de nœuds unaires pour le token NOT
* Operand : classe de feuilles stockant les ID des documents comportant les mots de la requête
* Parser : classe qui construit l’arbre de manière récursive et par ordre de priorité des opérations :
  + Fonction « factor » (cas de base) : si c’est un token de type Operand, alors on a atteint une feuille, si c’est un NOT alors on rappelle factor après avoir ajouté le nœud NOT. Si c’est une parenthèse gauche, alors on explore l’intérieur en appelant la fonction « expr »
  + Fonction « term » : appelle une 1e fois « factor » pour déterminer ce qu’il y a à gauche puis crée un nœud AND (s’il y a un token AND) et appelle « factor » à droite
  + Fonction « expr » (entrée dans la récurrence) : appelle une 1e fois « term » pour déterminer ce qu’il y a à gauche puis crée un nœud OR (s’il y a un token OR) et appelle « term » à droite

Des classes pour visiter l’arbre et trouver les bons doc ID pour la requête :

* NodeVisitor : crée le cas de base de visite de nœud
* Interpreter : visite de manière récursive l’arbre. Dès que le visiteur tombe sur un neud de type binaire il rappelle la fonction « visite » de la classe NodeVisitor pour chaque argument de la fonction d’évaluation . Il y a 3 cas de figure :
  + Nœud AND : appel à la fonction « intersect »
  + Nœud OR : appel à la fonction « union »
  + Nœud NOT : renvoie l’intégralité des ID moins les ID de ce qu’il y a en dessous

Enfin les fonctions :

* « Get\_postings » : pour un terme en entrée, la fonction renvoie la liste des postings associés dans l’index inversé
* « intersect » : prend 2 listes de postings (ordonnées) et renvoie l’intersection des deux
* « union » : prend 2 listes de postings (ordonnées) et renvoie l’union des deux sans doublons

La recherche booléenne se fait dans le module **R\_research.py** grâce à la fonction « boolean\_research ». Elle crée les tokens, l’arbre et récupère la liste des docs ID correspondant à la requête, tout en imprimant la liste des 10 premiers résultats (Doc ID + Titre)

## RECHERCHE VECTORIELLE

Modules concernés : **M\_vectorial.py** et **R\_research.py**

La fonction « vect\_search » de **M\_vectorial.py** calcule le TF de chaque mot (grâce à la fonction « tokenizer\_tf » de **R\_indexation.py**) puis calcule le cosinus entre la requête et chaque document de la collection. Elle renvoie une liste ordonnée de doc IDs par ordre décroissant de mesure du cosinus.

La recherche vectorielle est ensuite récupérée dans le module **R\_research.py** grâce à la fonction « vectorial\_research ». Elle fait appel à la fonction « vect\_search » de **M\_vectorial.py** qui calcule la proximité entre la requête et les documents de la collection et récupère la liste des docs ID, tout en imprimant la liste des 10 premiers résultats (Doc ID + Titre)

# PARTIE I : TRAITEMENT LINGUISTIQUE

**Q1:**

il y a 192,032 tokens dans la collection

**Q2:**

nombre de mots significatifs: 106,507 mots

sans doublons dans chaque doc: 76,654 mots

taille du vocabulaire (sans nombres): 8,780 mots

**Q3:**

collection entière - 192,032 tokens / 8,780 mots

demi collection - 55,133 tokens / 4,823 mots

=> M = kT^b

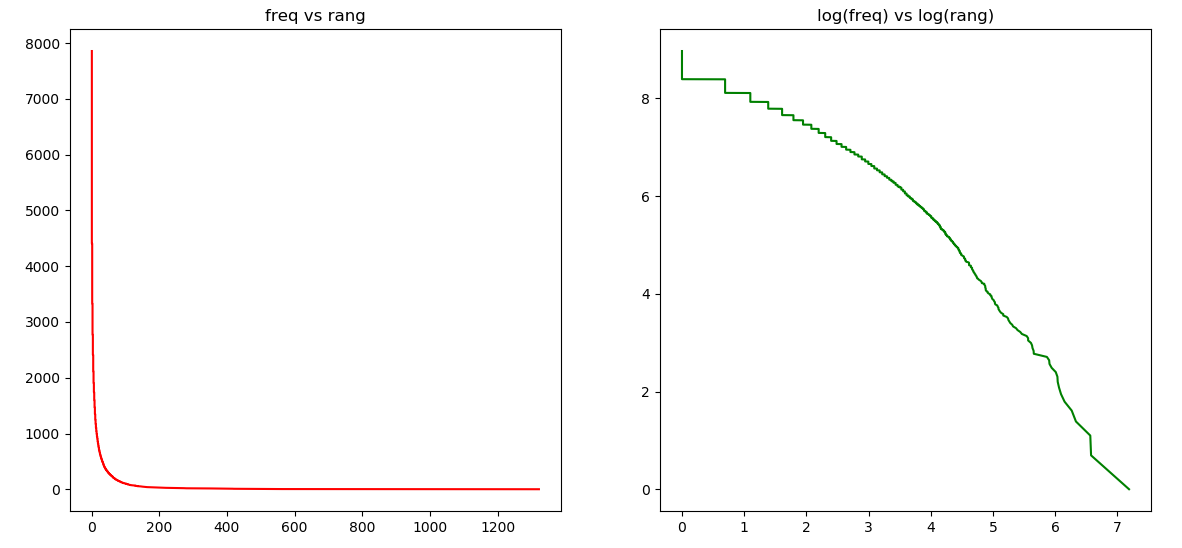
=> M = 25.534601029258393 \* math.pow(T, 0.48006550908028656)

**Q4 :**

Pour T = 1,000,000

M = 19,387 mots dans le vocabulaire

**Q5 :**



# PARTIE II : EVALUATION POUR LA COLLECTION CACM

## MESURES DE PERFORMANCE

- Temps de calcul indexation = 10.584 seconds

- Temps de réponse à une requête booléenne = 4.865 seconds pour la requête : "(document or master) and not (data or access)"

- Temps de réponse à une requête vectorielle = 6.219 secondes pour la requête : " code optimization for space efficiency"

- Taille index inversé = 2123 Ko

- Taille index documents = 85 Ko

## MESURES DE PERTINENCE

### Courbe rappel/précision pour les 8 premières questions :

Module **: A\_EvaluationCACM.py** et **A\_CACM\_QA.py**

Le module **A\_CACM\_QA.py** récupère les questions et les réponses des documents **qrels.text** et **query.text** et les stocke en format JSON.

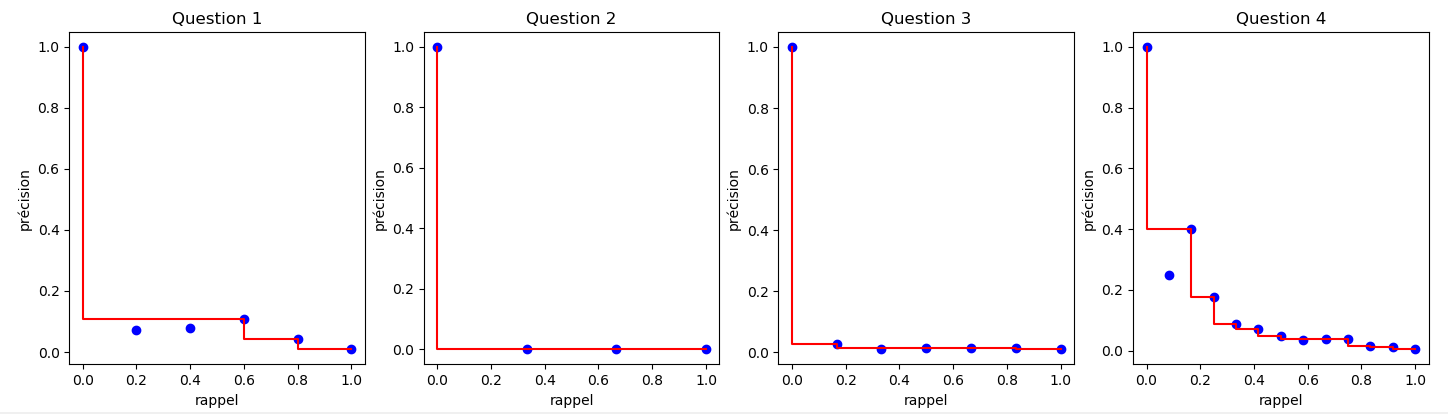
Le traçage des courbes de rappel/précision pour chaque question se fait dans le module **A\_EvaluationCACM.py** . La fonction « vectorial\_evaluation » va, pour chaque question, faire appel à la fonction « vect\_search » et calculer la précision et le rappel à chaque nouveau document retourné.

"1": " What articles exist which deal with TSS (Time Sharing System), an operating system for IBM computers? "

"2": " I am interested in articles written either by Prieve or Udo Pooch "

"3": " Intermediate languages used in construction of multi-targeted compilers; TCOLL "

"4": " I'm interested in mechanisms for communicating between disjoint processes, possibly, but not exclusively, in a distributed environment. I would rather see descriptions of complete mechanisms, with or without implementations, as opposed to theoretical work on the abstract problem. Remote procedure calls and message-passing are examples of my interests. "



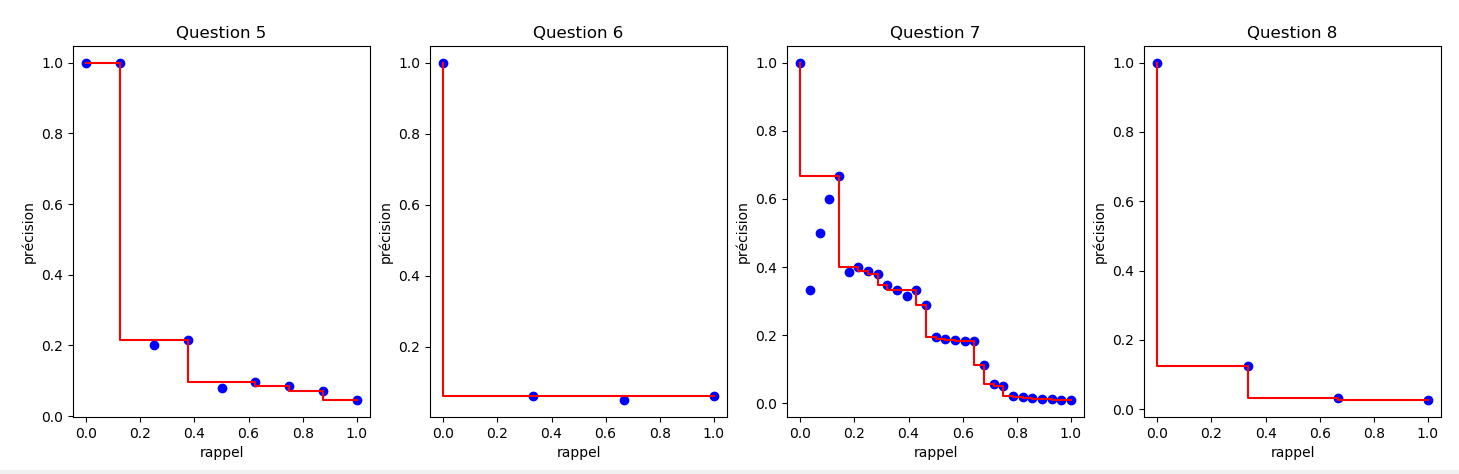


"5": " I'd like papers on design and implementation of editing interfaces, window-managers, command interpreters, etc. The essential issues are human interface design, with views on improvements to user efficiency, effectiveness and satisfaction. "

"6": " Interested in articles on robotics, motion planning particularly the geometric and combinatorial aspects. We are not interested in the dynamics of arm motion. "

"7": " I am interested in distributed algorithms - concurrent programs in which processes communicate and synchronize by using message passing. Areas of particular interest include fault-tolerance and techniques for understanding the correctness of these algorithms. "

"8": " Addressing schemes for resources in networks; resource addressing in network operating systems "





### F-Measure, E-Measure, R-Measure & Mean Average Precision

Module : **A\_Measures.py**

Moyenne des scores pour les 64 questions aux mesures à différents rangs (@K = 0, 5, 10, 15, […], 50). Certaines requêtes détériorent les résultats car elles n’ont pas de bonnes réponses dans les documents.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Rang @k | Mean Average Precision | Rappel (R-measure) | F1-measure | E1-measure |
| 0 | 1 | 0. | 0. | 1. |
| 5 | 0.3484375 | 0.1152616 | 0.11516427 | 0.88483573 |
| 10 | 0.26916419 | 0.16289327 | 0.13431491 | 0.86568509 |
| 15 | 0.23559476 | 0.20909557 | 0.15448776 | 0.84551224 |
| 20 | 0.21671542 | 0.24560089 | 0.15868021 | 0.84131979 |
| 25 | 0.2023464 | 0.28198504 | 0.15881925 | 0.84118075 |
| 30 | 0.19094674 | 0.30995553 | 0.15873463 | 0.84126537 |
| 35 | 0.18161017 | 0.32972536 | 0.15525611 | 0.84474389 |
| 40 | 0.1736591 | 0.35611882 | 0.1517104 | 0.8482896 |
| 45 | 0.16652589 | 0.37214981 | 0.14617007 | 0.85382993 |
| 50 | 0.16018964 | 0.39657532 | 0.14430263 | 0.85569737 |

