Projet : Moteur de recherche

L'objet de ce projet est d'implémenter un moteur de recherche de type "Google" qui permet de restituer des documents en réponse à une requête.
De façon technique, ce projet permettra d'ajouter de nouvelles compétences java :
- L'utilisation de librairies externes et de leur JavaDoc
- L'utilisation des HashMap

Consignes. Le projet est à rendre pour le **18 décembre 2017**. Vous devrez remettre les documents suivants par mail, compressés dans un fichier zip, intitulé : "MoteurRI_Nom1_Nom2.zip" :
- le projet java dans son intégralité
- un document d'une page recensant les différents `main` à exécuter pour tester votre code.
- la javadoc associée
- tout autre information qui vous semblera utile à l'évaluation/compréhension du projet

De plus, il vous sera demandé lors de la dernière séance de TME de montrer et d'expliquer votre code au chargé de TD. Pensez donc à vérifier que votre projet fonctionne sur les machines de la fac avant le TME.

Données fournies. En plus du sujet, vous disposez des éléments suivants :
- Un fichier `cisi.txt` qui contient la collection de documents CISI où les documents sont stockés au format texte dans le fichier `cisi.txt` [1].
- Une librairie `indexation.jar` [2] avec sa javadoc [3].

Principe d'un moteur de recherche. Un moteur de recherche repose sur deux grands principes :
- L'indexation des documents, qui permet de pré-traiter et stocker les documents afin de faciliter leur interrogation. Les pré-traitements incluent :
  1. la suppression de certains mots ("le", "la", "les", ...),
  2. la normalisation des mots ("chevaux" → "cheval", "amie" → "ami")
  3. la pondération calculant l'importance des mots dans les documents.
- L'interrogation des documents à partir d'une requête qui retourne la liste ordonnée des documents qui répondent à la requête. Afin de correspondre aux termes stockés des documents indexés, la requête doit elle-aussi passer par une étape de pré-traitement. Il existe plusieurs modèles qui permettent de trier les documents. Nous les verrons plus loin.

Dans ce projet, nous utiliserons la librairie d'indexation fournie pour indexer les documents, puis nous implémenterons la partie interrogation. De plus, nous nous intéresserons à l'évaluation de la qualité des modèles d'interrogation.

Exercice 1 – Indexation de la collection

La classe `Index` permet de construire, à partir d'une collection de documents fournie, un objet qui contient simultanément deux index :
- L'index classique qui pour chaque document recense les termes présents et leur occurrence. Par exemple pour les documents d_1 : "souris grise ordinateur gris" et d_2 : "soleil grise pluie", l'index sera le suivant : "d_1 : (souris :1),(gris :2), (ordinateur :1)" et "d_2 : (soleil :1),(gris :1), (pluie :1)".
- L'index inverse qui pour chaque mot identifie la liste des documents qui le contiennent et le nombre d'occurrences dans le document (par exemple "souris : (d_1 :1)", "gris : (d_1 :2),(d_2 :1)", ...).

Q 1.1 Prendre connaissance de la JavaDoc :
- Comment récupérer le texte dans un `Document` ?
- Quel est la hierarchie des classes de la librairie ?
- A quoi correspondent les méthodes `getTfsForDoc` et `getTfsForStem` de la classe `Index` ?

Q 1.2 Écrire dans une classe `MainIndexation` un programme principal qui créer un `Index` utilisant un `ParserCISI` et un `Stemmer` définis dans la librairie. S'en servir pour générer un index pour la collection de documents CISI.

Q 1.3 Cette étape d'indexation peut prendre un certain temps. Pourquoi les méthodes `serialize` et `deserialize` de la classe `Index` peuvent alors être utile ?

Q 1.4 Écrire dans une classe `TestIndexation` un programme principal qui :

1. http://www-connex.lip6.fr/~soulier/data/2I002/projet2017/data/cisi/cisi.txt
2. http://www-connex.lip6.fr/~soulier/data/2I002/projet2017/indexation.jar
3. http://www-connex.lip6.fr/~soulier/data/2I002/projet2017/doc/index.html

— Charge l'`Index` précédemment construit
— Affiche le texte du document "55"
— Affiche l'ensemble des mots inclus dans le document "55" et leur occurrence.
— Affiche l'ensemble des documents qui contiennent le mot "attempt" son occurrence dans chaque document.

Q 1.5 Les méthodes de la classe `Index.java` utilisent un objet HashMap<,>. Renseignez-vous sur cet objet et son fonctionnement : initialisation, insertion, accesseur, parcours de l'objet, ...

Q 1.6 Faire quelque tests dans la classe `TestIndexation` et compléter le programme principal avec les fonctionnalités suivantes :
— Afficher le nombre termes indéxés differents contenus dans un document.
— Afficher le nombre de documents qui contiennent un terme donné.
— Afficher le texte de tous les documents qui contiennent un terme donné.
— Générer et afficher une Hashmap<String,Double> qui associe a chaque document un score aléatoire.

Exercice 2 – Interrogation des documents

L'interrogation des documents permet de mettre en correspondance des documents par rapport à une requête donnée au moyen d'un score qui dénote l'importance du document pour la requête. En d'autres termes, pour une requête donnée, il s'agit de calculer pour chaque document un score de pertinence pour la requête. L'objectif final est de définir une liste ordonnée de documents en fonction de ce score de pertinence. Comme il existe différents scores, nous définissons des modèles (appelés `IRModel`) pour les représenter.
On dénombre deux grandes familles de modèles :

1. Le modèle booléen (`Boolean`) qui retourne les documents qui contiennent tous les mots de la requête. Dans ce cas, le score de chaque document vaut soit 1 soit 0 selon que le document contient les tous les mots de la requête ou non. Il peut être calculé directement à partir des index classique et inverse.
2. Les modèles vectoriels qui 1) ont une façon particulière de représenter les documents sous la forme d'un vecteur de poids, où chaque élément du vecteur représente le poids d'un mot et 2) calcule un score d'importance à partir de ces vecteurs de poids. On dénombrera deux méthodes de pondération des mots :
   — la fréquence (TF). Par exemple, les documents sont représentés sous la forme de vecteur de fréquence, et donc "d_1 : (souris :1/4),(gris :1/2), (ordinateur :1/4)".
   — la fréquence combinée à la fréquence inverse (TFIDF). Ce poids correspond à la multiplication du TF avec l'IDF du terme correspondant où $IDF(terme) = ln(\frac{N}{nb\ Doc\ qui\ contiennent\ terme})$, calculable depuis l'index inverse.

   Un modèle vectoriel utilisera ces poids pour calculer leur score. On s'intéressera à deux scores différents et on implémentera les modèles vectoriels correspondant (avec d et q étant respectivement les vecteurs de poids des termes des documents et de la requête) :
   — Le produit cartésien (`VectorialCart`) :

$$score(q,d) = q \cdot d = \sum_{i \in q} q_i * d_i \tag{1}$$

   avec q_i le poids du terme i dans la requête q et d_i le poids du terme i dans le document d.
   — Le cosinus (`VectorialCos`) :

$$score(q,d) = \frac{q \cdot d}{|q| * |d|} = \frac{\sum_{i \in q} q_i * d_i}{\sqrt{\sum_{i \in q} q_i^2} * \sqrt{\sum_{i \in d} d_i^2}} \tag{2}$$

Une fois les scores calculés pour chaque document, les modèles doivent retourner une liste ordonnée de documents. Pour vous guider, le code d'un programme principal vous est donné.

Q 2.1 Dessiner le schéma UML qui modélise l'architecture de l'étape d'interrogation d'un moteur de recherche.

Q 2.2 Implémenter cette étape.

Dans `mainInterrogation.java`

```
1 public static void main(String[] args){
2         // initialise l'index
3         Index index=Index.deserialize("cisi");
4         String query = "young boy";
5         // Modele booleen
6         ModelIR mod=new Boolean(index);
7         mod.runModel(query);
8
9         // Modele Vectoriel Produit Cartesien avec ponderation TF
10        WeighterTF w  = new WeighterTF(index);
11        IRModel modCos = new VectorielCartesien(index, w);
12        System.out.println(modCos.runModel(query));
13
14        // Modele Vectoriel Cosinus avec ponderation TFIDF
15        WeighterTFIDF w  = new WeighterTFIDF(index);
16        IRModel modCart = new VectorielCartesien(index, w);
17        System.out.println(modCart.runModel(query));
18 }
```

Dans `IRModel.java`

```
1 public LinkedHashMap<String,Double> runModel(String query){
2         HashMap<String,Integer> queryProcessed= getQueryProcessed(query);
3         HashMap<String,Double> docsScore= getDocScores(queryProcessed);
4         return getRanking(docsScore);
5 }
```

Exercice 3 – Évaluation du moteur de recherche

Remarque : Pour cet exercice, il n'est pas nécessaire d'avoir réussi à coder tous les modèles de l'exercice 2. Vous pouvez utiliser la liste de documents générée aléatoirement dans l'exercice 1.

Nous souhaitons évaluer la qualité des modèles implémentés dans le moteur de recherche. Pour cela, nous disposons d'un ensemble de requêtes pour lesquelles nous connaissons les documents qui sont reliés. Vous trouverez ces informations respectivement dans les fichiers `cisi.qry`[4] et `cisi.rel`[5] de chaque jeu de données.
Le fichier `cisi.rel` inclut 4 colonnes, seulement les deux premières nous intéressent :

```
1 idRequete     idDoc
2 01            123     0       0
3 01            456     0       0
4 ....
5 02            654     0       0
6 ...
```

Il existe trois mesures d'évaluation permettant de calculer la qualité des listes ordonnées retournées par le moteur de recherche. Pour une liste D de documents et une requête q, on regardera les N premiers documents D_N et on les compareras aux réponses attendues *DocAttendus* (colonnes 1 et 2 dans le fichier `cisi.rel`) avec les mesures suivantes :

- La précision : $\frac{|D_N \cap DocAttendus|}{|D_N|}$
- Le rappel : $\frac{|D_N \cap DocAttendus|}{|DocAttendus|}$
- La F-mesure : $2 \cdot \frac{precision \cdot rappel}{precision + rappel}$

où $|X|$ correspond au nombre d'éléments dans l'ensemble X.

Q 3.1 Dessiner le schéma UML modélisant l'étape d'évaluation du moteur de recherche

Q 3.2 Implémenter cette étape, sans oublier de tester l'implémentation pour $N = 10$.

Exercice 4 – Bonus

Q 4.1 Réaliser l'interface de l'application "Moteur de recherche". Vous pouvez intégrer une partie des fonctionnalités ou la totalité. Vous avez également la possibilité de rajouter d'autres contraintes/modèles/etc... Carte blanche !

4. http://www-connex.lip6.fr/~soulier/data/2I002/projet2017/data/cisi/cisi.qry
5. http://www-connex.lip6.fr/~soulier/data/2I002/projet2017/data/cisi/cisi.rel

Annexe

Vous trouverez ici un ensemble de codes pour vous aider. Bien entendu, leur utilisation est optionnelle. Vous avez le droit de faire vos propres codes.

```
// pour formater la requete sous la forme d'un HashMap
public HashMap<String,Integer> getQueryProcessed(String query){
                TextRepresenter textRep=index.getTextRepresenter();
                HashMap<String,Integer> ret=textRep.getTextRepresentation(query);
                return ret;
}


// pour trier une HashMap - ordonnancement des documents
public LinkedHashMap<String,Double> getRanking(HashMap<String,Double> docScores){
        List<Map.Entry<String, Double>> entries =  new ArrayList<Map.Entry<String,
            Double>>(docScores.entrySet());
        Collections.sort(entries, new Comparator<Map.Entry<String, Double>>() {
        public int compare(Map.Entry<String, Double> a, Map.Entry<String, Double> b){
                return b.getValue().compareTo(a.getValue());
        }
        });
        LinkedHashMap<String, Double> ret = new LinkedHashMap<String, Double>();
        for (Map.Entry<String, Double> entry : entries) {
                ret.put(entry.getKey(), entry.getValue());
        }
        return ret;
}
```