UNIVERSITÉ D'AVIGNON
ET DES PAYS DE VAUCLUSE

M2 ILSEN – 2019/20
UE Ingénierie du document et de l'information
UCE Indexation & recherche

Vincent Labatut

Sujet | Examen de travaux pratiques

Durée : 1h15. Le barème est seulement indicatif, il peut être modifié. Les documents sont autorisés, y compris le Web. Tous les moyens de communication sont interdits. Votre rendu doit prendre la forme d'un projet Eclipse zippé. Votre code source doit être bien présenté, indenté, clair, simple, compilable. Les consignes en vigueur en TP restent valables lors de cet examen (cf. TP0), en particulier pour ce qui concerne le test des méthodes écrites. Les *éventuelles* réponses textuelles sont à écrire dans un fichier `reponses` (`.txt`, `.pdf`, `.odt`...) placé dans la racine de votre projet. Chaque étudiant est seul responsable du dépôt de son travail sur e-uapv.

1 Préparation

L'objectif de ce TP est d'implémenter un lexique de type *Permuterm* : lors de la construction de l'index, on n'insère donc plus directement les termes mais leurs différentes rotations. Pour mémoire, une rotation correspond à une ré-écriture d'un terme incluant un `$` supplémentaire pour en marquer la fin. Ainsi pour `chat`, on aura les rotations suivantes : `chat$`, `$chat`, `t$cha`, `at$ch`, et `hat$c`. L'intérêt (expliqué en cours) de cette approche est de pouvoir effectuer des recherches approchées.

Comme indiqué à l'avance, on se basera sur le code source produit auparavant lors de la série de TP, et ayant été préparé de la manière expliquée au préalable (cf. la première partie du sujet). Pour rappel, votre projet Eclipse doit porter un nom de la forme `NomPrenom`, où `Nom` est votre nom de famille et `Prenom` votre prénom. À la fin de la séance, vous devrez rendre une archive contenant ce projet, et portant le même nom. Attention de bien respecter ces consignes de nommage.

Téléchargez l'archive `classes.zip`. fournie avec ce sujet, et dézippez-la : vous allez obtenir un projet Eclipse. Copiez le dossier `src` de ce projet et collez-le dans le vôtre, de manière à écraser le dossier `src` existant. Le but de l'opération est d'intégrer à votre code source des classes squelettes additionnelles, à compléter lors de la séance.

Remarque : Tous les résultats donnés en exemple dans ce sujet sont obtenus sur la base d'index construits en filtrant les mots-vides et sans faire de racinisation.

Exercice 1 (1 pt.)

Cet exercice sert seulement à indiquer que les modifications du code source demandées en amont de la séance d'examen sont prises en compte de manière groupée.

2 Construction de l'index

Par rapport à ce qui a été fait en TP, la structure de données elle-même ne nécessite pas de modification : c'est seulement le *processus d'indexation* qui va devoir être adapté. Jusqu'à présent, celui-ci comportait 3 étapes réalisées dans `Index.indexCorpus` : la tokénisation, la normalisation et la construction. Nous allons utiliser une approche naïve consistant à rajouter une étape de *conversion* située entre les deux dernières, qui consistera à substituer à un token normalisé toutes les rotations correspondantes.

Exercice 2 (3 pt.)

La classe classe `Converter` située dans le package `processing` est destinée à traiter l'étape de conversion. Notez qu'elle est *sérialisable*, car le convertisseur sera enregistré plus tard avec le normalisateur et le tokénisateur.

Dans cette classe, complétez la méthode `public List<Token> convertIndexTokens(List <Token> tokens)` qui reçoit en paramètre une liste de tokens normalisés obtenus lors de l'indexation, et qui renvoie une nouvelle liste dans laquelle chaque token original a été remplacé par plusieurs tokens, chacun correspondant à l'une des rotations nécessaires à la représentation du token original. Bien entendu, le docId original n'est pas modifié.

Exemple : pour la liste contenant les tokens `abc` et `de`, situés respectivement dans les documents 1 et 99, on aura le résultat

```
[(abc$, 1), ($abc, 1), (c$ab, 1), (bc$a, 1), (abc$, 1), (de$, 99), ($de, 99), (e$d,
99), (de$, 99)]
```

Notez que le marqueur de fin de terme `$` est déjà défini sous la forme de la constante `CHAR_END`.

Exercice 3 (2 pt.)

Dans la classe `AbstractIndex`, ajoutez un champ `converter` destiné à stocker le convertisseur utilisé lors de l'indexation. Complétez ensuite la méthode `indexCorpus` de manière à :

- Instancier un convertisseur sous la forme d'un objet de classe `Converter` ;
- Réaliser l'étape supplémentaire de conversion en utilisant ce convertisseur ;
- Assigner ce convertisseur au champ `converter` de l'index créé.

À l'instar de ce qui a été fait pour les autres étapes, vous devez afficher le temps mis pour effectuer la conversion et le nombre de tokens présents dans la liste obtenue.

Exemple : sortie console obtenue pour le corpus de test `wp_test`

```
Tokenizing corpus...
631928 tokens were found, duration=802 ms

Normalizing tokens...
352716 tokens remaining after normalization, duration=1306 ms

Converting tokens...
3156324 tokens in total after conversion, duration=1144 ms

Building index...
  Sorting tokens...
  3156324 tokens sorted, duration=7011 ms

  Filtering tokens...
  1311016 tokens remaining, corresponding to 335591 terms, duration=684 ms

  Building posting lists...
  1311016 postings listed, lexicon=TREE, duration=587 ms
There are 335591 entries in the index, token list=LINKED, duration=8283 ms

Total duration=11535 ms
```

Exercice 4 (4 pt.)

Dans la classe `TestExam`, utilisez la méthode `testIndexation` pour tester votre nouvelle version de l'index.

Dans un premier temps, indexez le corpus `wp_test`, et comparez les statistiques ainsi que l'index obtenu avec ceux produits lors des TP (même paramétrage, mais sans le Permuterm). Présentez sous forme synthétique (par exemple dans une table) le nombre d'éléments (tokens, postings, termes, etc.) et le temps de calcul mesurés pour les différentes étapes du traitement (tokénisation, normalisation, etc.) et pour les deux versions de l'index (avec vs. sans Permuterm). Sur cette base, discutez comment les étapes du traitement affectent ces statistiques, et en quoi cette évolution diffère entre les deux versions de l'index.

En particulier, lors du filtrage, vous devriez observer une réduction supérieure (proportionnellement parlant) du nombre de tokens pour la version *avec* Permuterm. Avancez une explication

pour justifier cette différence. Examinez également les tailles des fichiers `.data` produits.

Lancez ensuite l'indexation du corpus `wp` complet, afin de confirmer les résultats obtenus sur `wp_test`. Il est possible que le traitement soit long, mais vous pouvez continuer le TP et revenir ensuite sur ce point plus tard dans la séance. Identifiez les points de votre discussion invalidés par ces nouveaux résultats, et le cas échéant, discutez-les.

3 Traitement des requêtes

L'intérêt principal du Permuterm est qu'il permet d'effectuer des recherches approchées en utilisant des jokers. On utilise ici `*` pour représenter une partie manquante d'un mot. Autrement dit, la requête `arb*` désigne tous les mots commençant par `arb`, tandis que `*bre` représente tous ceux finissant par `bre`, et `ar*re` tous ceux commençant par `ar` *et* finissant par `re`.

Notez qu'il est possible de combiner des expressions avec ou sans joker dans la *même* requête. Par exemple, la requête `arb* plante` signifie qu'on veut récupérer tous les documents contenant à la fois un mot commençant par `arb` et le mot `plante`. Par contre, on se limitera à un seul joker par mot *au plus* (i.e. pas de `a*b*e`).

Exercice 5 (2 pt.)

Dans `Converter`, complétez la méthode `public String convertQueryToken(String token)`, qui prend en paramètre une chaîne de caractères `token` représentant un token normalisé obtenu lors du traitement d'une requête. La méthode doit renvoyer une chaîne de caractères correspondant à la rotation appropriée, en fonction de la *présence* éventuelle d'un joker dans `token`, et de la *position* de ce joker. La chaîne résultat ne doit pas contenir de joker.

Exemples : `arbre` devient `arbre$` ; `arb*` devient `$arb` ; `*bre` devient `bre$` ; et `ar*re` devient `re$ar` (cf. le cours pour d'autres exemples).

Notez que le caractère joker `*` est déjà représenté sous la forme de la constante `CHAR_JOKER`.

Exercice 6 (3 pt.)

Dans la classe `AndQueryEngine`, il faut maintenant modifier la méthode `splitQuery` de manière à prendre en compte la présence potentielle de jokers dans la requête. Rappelons que cette méthode a originellement pour rôle de tokéniser la requête, puis de normaliser chacun des tokens obtenus et de récupérer les listes de postings qui leurs sont associées dans l'index. Il y a deux points à adapter dans `splitQuery`.

La première modification est *systématique* : il faut maintenant convertir les tokens normalisés au moyen de `convertQueryTokens`, afin d'obtenir la chaîne de caractère qui sera recherchée dans le lexique.

La seconde modification ne concerne que les token contenant un joker `*`. Dans ce cas-là, il est possible d'obtenir *plusieurs* listes de postings pour *un seul* token, puisque plusieurs termes du lexique peuvent lui correspondre. On va alors prendre l'union de ces listes, pour se ramener à une situation où on associe *une seule* liste de postings à un token donné. Le reste du traitement ne change pas.

Exemples : listes de listes de postings obtenues pour `wp_test`

Requête `arbre` : 15 postings

```
[[5, 16, 34, 47, 54, 64, 90, 95, 96, 99, 104, 121, 144, 146, 180]]
```

Requête `arb*` : 41 postings

```
[[5, 7, 16, 22, 25, 29, 33, 34, 36, 40, 42, 43, 45, 47, 54, 55, 64, 66, 79, 83...]]
```

Requête `*bre` : 148 postings

```
[[0, 1, 2, 3, 4, 5, 7, 9, 10, 13, 14, 15, 16, 17, 18, 19, 22, 23, 24, 25, 26...]]
```

Requête `ar*re` : 61 postings

```
[[4, 5, 7, 14, 16, 22, 25, 27, 29, 31, 34, 35, 40, 42, 45, 47, 54, 58, 60, 61...]]
```

Requête `le` : 0 postings

```
[ ]
```

Requête `le*` : 160 postings

```
[[2, 3, 4, 5, 6, 7, 10, 12, 13, 14, 16, 18, 19, 22, 23, 24, 25, 26, 27, 29, 30...]]
```

Requête `*le` : 193 postings

```
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21...]]
```

Requête `science université` : 33 et 49 postings

```
[[4, 5, 13, 23, 27, 29, 31, 36, 37, 41, 45, 66, 67, 72, 73, 77, 83, 96, 97, 104...],
 [3, 4, 5, 11, 13, 14, 16, 25, 33, 34, 35, 36, 46, 53, 55, 57, 61, 72, 74, 79, 80...]]
```

Requête `science univ*` : 33 et 87 postings

```
[[4, 5, 13, 23, 27, 29, 31, 36, 37, 41, 45, 66, 67, 72, 73, 77, 83, 96, 97, 104...],
 [3, 4, 5, 7, 11, 13, 14, 16, 22, 23, 25, 27, 29, 30, 33, 34, 35, 36, 37, 42, 46...]]
```

Requête `sci* univ*` : 90 et 87 postings

```
[[0, 4, 5, 7, 8, 13, 14, 16, 17, 19, 21, 23, 25, 27, 29, 31, 35, 36, 37, 39, 41...],
 [3, 4, 5, 7, 11, 13, 14, 16, 22, 23, 25, 27, 29, 30, 33, 34, 35, 36, 37, 42, 46...]]
```

Requête `sci*e un*té` : 60 et 70 postings

```
[[4, 5, 13, 14, 16, 17, 19, 23, 25, 27, 29, 31, 35, 36, 37, 39, 41, 45, 47, 50...],
 [3, 4, 5, 11, 13, 14, 16, 19, 22, 25, 30, 33, 34, 35, 36, 37, 42, 46, 47, 52, 53...]]
```

Exercice 7 (1 pt.)

Dans la classe `TestExam`, complétez la méthode `testQuery` de manière à charger l'index construit à l'Exercice 4 puis à tester une liste de requêtes contenant (ou pas) des jokers. Invoquez cette méthode depuis le `main` de `TestExam` et affichez les résultats obtenus pour les requêtes traitées.

Exemples : quelques requêtes résolues sur le corpus `wp_test` :

Requête `science université` : 12 postings

```
[4, 5, 13, 36, 72, 96, 97, 122, 124, 177, 181, 191]
```

Requête `science univ*` : 23 postings

```
[4, 5, 13, 23, 27, 29, 36, 37, 67, 72, 73, 83, 96, 97, 115, 122, 124, 131, 173...]
```

Requête `sci* univ*` : 56 postings

```
[4, 5, 7, 13, 14, 16, 23, 25, 27, 29, 35, 36, 37, 53, 55, 61, 67, 72, 73, 74, 78...]
```

Requête `sci*e un*té` : 34 postings

```
[4, 5, 13, 14, 16, 19, 25, 35, 36, 37, 47, 55, 61, 72, 74, 83, 85, 92, 96, 97, 99...]
```

Exercice 8 (2 pt.)

Complétez la méthode `convertQueryToken2` de manière à traiter le cas particulier où un même token contient *deux* jokers situés au *début* et à la *fin*. Modifiez `convertQueryToken2` de manière à utiliser cette méthode.

Exemple : pour la requête `*rbr*`, on obtient une liste de 61 postings sur `wp_test`

```
[[4, 5, 7, 14, 16, 22, 25, 27, 29, 31, 34, 35, 40, 42, 45, 47, 54, 58, 60, 61, 64...]]
```

Exercice 9 (2 pt.)

Supposons qu'on veuille autoriser n'importe quel nombre de jokers dans un token, et à n'importe quelle position, comme par exemple `a*b*e`. Quelles méthodes modifieriez-vous ? Comment vous y prendriez-vous ? Sans l'implémenter, expliquez cela dans le fichier `reponses`, et donnez l'algorithme général de votre approche. Inspirez-vous des mécanismes présentés en cours pour palier les limitations observées pour d'autres approches (par ex. k-grammes de caractères).