Kodex · Lab

Korantin Auguste

Rapport de stage de fin d'études

Maitre de stage :
Emmanuel Navarro

Tuteur de stage :
Jean-Christophe Buisson

Objectifs :

Développement sur la base de travaux de P. Magistry (ELeVE) d'une bibliothèque d'indexation de n-grammes permettant le calcul à la fois de scores d'autonomie et de pertinence (type TF-IDF). Dans un second temps, utilisation de cette bibliothèque pour mettre en place des expérimentations d'extraction de mots-clés sur de larges collections de documents.

13 avril — 11 septembre

Remerciements

Je tiens à remercier mon maitre de stage (et ancien professeur de TP à l'n7 !) Emmanuel Navarro, pour avoir rendu ce stage possible, ses conseils et sa disponibilité malgré un emploi du temps chargé. Ainsi que tous les membres de Kodex · Lab, en particulier Pierre Magistry pour son aide précieuse, et Yannick Chudy pour sa sympathie et pour m'avoir fait découvrir le coworking avec Tau !

Je remercie aussi mon tuteur de stage, Jean-Christophe Buisson.

Dédicace aux membres du club d'informatique de l'n7, net7, aux contributeurs des logiciels libres que j'ai utilisé tous les jours, à mes amis et proches ainsi qu'à tous les êtres vivants de la planète Terre (pourquoi pas).

Table des matières

1 Introduction **4**
1.1 Kodex · Lab . . . 4
1.2 Contexe du projet . . . 4
1.3 Objectifs du stage . . . 5

2 Calcul d'autonomie **6**
2.1 EleVE : travaux de Pierre Magistry . . . 6
2.1.1 Autonomie d'un n-gramme . . . 6
2.1.2 Segmentation du mandarin . . . 7
2.2 Cahier des charges & critères d'intérêt . . . 8
2.3 Pistes théoriques . . . 9
2.3.1 Réflexions préliminaires . . . 9
2.3.2 Stockage . . . 10
2.4 Solutions existantes . . . 11
2.4.1 ELeVE existant . . . 11
2.4.2 Kenlm . . . 13
2.4.3 Tries en Python . . . 13
2.4.4 Xapian . . . 14
2.5 Travail effectué . . . 14
2.5.1 Backend RAM . . . 14
2.5.2 Backend incrémental . . . 16
2.5.3 Backend Neo4j . . . 17
2.5.4 Backend « merge » . . . 19
2.5.5 Backend C++ . . . 19
2.6 Premiers résultats/benchmarks . . . 20
2.7 Seconde itération : Leveldb . . . 21
2.8 Benchmark finaux . . . 23

3 Autonomie & extraction **24**
3.1 État de l'art . . . 25
3.1.1 Évaluation . . . 26
3.2 Méthodologie . . . 26
3.2.1 Évaluation via le F-score. . . . 26
3.2.2 Filtres . . . 27
3.2.3 Évaluation via la capture corpus/gold. . . . 28
3.3 Jeux d'évaluation . . . 28
3.4 Évaluation . . . 31

4 Conclusion **33**
4.1 Évaluations sur tous les corpus . . . 35

1 Introduction

Ce rapport présente le stage de fin d'étude que j'ai eu l'opportunité d'effectuer à Kodex · Lab, d'avril à septembre 2015.

C'est en contactant Emmanuel Navarro, ancien encadrant de TP à l'n7 que ce stage m'a été proposé. Le sujet (traitement du langage naturel) et le contexte me semblant propices à découvrir des domaines qui m'intéressent, j'ai accepté le stage.

L'objectif du stage est d'utiliser l'autonomie (une mesure utilisée pour la segmentation de texte) pour améliorer les résultats sur une tâche d'extraction de mots-clés, en ayant au préalable implémenté un moteur de calcul de l'autonomie efficace.

Après avoir présenté la société Kodex · Lab, on développe sur les contextes sur lesquels se basent les deux objectifs du stage, avant de revenir sur les objectifs précis de ce dernier.

1.1 Kodex · Lab

Kodex · Lab est une spin-off de recherche fondée par cinq chercheurs et ingénieurs, en 2014 :

- Bruno Gaume
- Emmanuel Navarro
- Pierre Magistry
- Yannick Chudy
- Yann Desalle

Leur activité se fait sur trois axes :

Modélisation psycholinguistique consistant à proposer des outils de diagnostic de pathologies basés sur la manière d'utiliser le langage, à destination des médecins, orthophonistes ...

Moteur d'exploration et de navigation concernant tous les services liés à l'extraction de données utile depuis de gros corpus de textes : extraction de mots-clés, visualisation de données, segmentation de texte, etc.

Services en ligne à savoir, mise en place d'interfaces de programmation permettant d'effectuer automatiquement des traitements (comme ceux du point précédent) sur des données brutes.

1.2 Contexe du projet

Mon stage s'insère suite à, et vise à combiner deux tâches classiques de traitement automatique du langage, que voici.

Segmentation. Mon stage se base sur les travaux de Pierre Magistry, associé fondateur de Kodex · Lab ayant fait sa thèse sur la segmentation du mandarin.

Le mandarin est une langue constituée de signes qui se suivent. Un concept (un « mot ») peut pourtant être constitué de plusieurs caractères.

Se pose alors la question du découpage de textes en mandarin en « mots », tâche préalable à tout traitement automatisé de texte. C'est une tâche délicate, car un mot

n'a pas de définition précise, et des locuteurs pourront choisir des découpages différents, selon leur sensibilité.

Les méthodes de l'état de l'art s'appuient sur de l'apprentissage supervisé (donc basé sur de gros corpus d'exemples déjà découpés), et Pierre Magistry a développé une méthode qui se base sur des statistiques intrinsèques au « langage » (c'est-à-dire un gros corpus de texte non découpé en mots) pour effectuer le découpage. C'est donc une méthode non-supervisée qui constitue une piste vers une définition mathématique et calculable de ce qu'est un mot.

Or, une des pistes de réflexion exploitable, c'est d'utiliser le même algorithme sur des langues européennes : si on l'applique à l'échelle des mots (un token n'est pas une lettre de l'alphabet, mais un mot), on doit pouvoir en tirer une mesure servant à découper une phrase en groupes de mots. Les groupes de mots extraits ont alors la propriété d'aller tout le temps ensemble, comme les groupes de caractères du mandarin.

Extraction de termes clés. En parallèle des travaux de Pierre Magistry, il faut aussi savoir que l'extraction de termes clés est une tâche assez importante et commune en traitement du langage naturel : il s'agit d'extraire des termes pertinents (constitués d'un ou plusieurs mots) de documents d'un corpus.

L'idée étant d'utiliser des avancées récentes en segmentation de texte, afin d'améliorer les résultats des systèmes d'extraction de termes clés.

1.3 Objectifs du stage

Comme nous l'avons déjà dit, il s'agit d'appliquer des algorithmes de segmentation du mandarin à l'extraction de mots-clés. Plusieurs étapes ont été identifiées :

Calcul d'autonomie. Un prototype a été développé par Pierre Magistry pour calculer les scores d'autonomie sur du texte. Ce prototype présente toutefois des faiblesses en terme de performance. Il s'agira de développer une nouvelle implémentation plus performante afin de travailler sur de gros corpus.

L'objectif est aussi d'avoir un code de qualité. En particulier, il devra être couvert par une suite de tests automatiques, et bien documentés.

Extraction de termes clés. Le lien entre extraction de termes clés et autonomie est simple : on va chercher à voir dans quelle mesure l'autonomie permet d'améliorer les algorithmes d'extraction de mots-clés, voir quel rôle elle peut jouer. On verra par la suite qu'elle aide surtout à un filtrage préalable avant l'extraction et le tri des candidats : on limite ainsi le nombre de candidats, et donc de n-grammes à indexer et traiter.

Tout cela est rendu possible, car l'extraction du mandarin utilise une mesure d'*autonomie* (définie en 2.1.1) appliquée à des groupes de caractères, pour connaitre leur probabilité d'être des « mots » autonomes. Si l'on applique cette même mesure aux mots du français, on pourra extraire des expressions comme « New York », constituée de mots qui apparaissent tout le temps ensemble : ce sont probablement de bons candidats pour l'extraction de termes clés.

2 Calcul d'autonomie

Cette section présente le travail effectué sur la partie de calcul d'autonomie, indépendamment de la tâche d'extraction de mots-clés.

La section 2.2 décrit le cahier des charges fixé et les critères d'intérêt, qui ont évolué avec le temps. On voit ensuite en 2.4 et 2.5 les solutions envisagées et celles testées. Puis, on verra qu'au terme de benchmarks il a été nécessaire de nous réorienter vers une seconde solution présentée en 2.7, qui donne des résultats très satisfaisants que l'on étudiera.

2.1 EleVE : travaux de Pierre Magistry

Nous définissons ici la notion d'« autonomie » d'une suite de caractères (ou de tokens). Pour plus d'informations, se référer à sa thèse (Magistry 2013) ou à cet article : Magistry et Sagot (2012).

2.1.1 Autonomie d'un n-gramme

Un n-gramme est une liste de *tokens* de longueur n. Un *token* étant ici un mot du français (on verra que la définition de token pourra être modifiée par la suite).

Entropie de branchement à droite (RBE). Pour un n-gramme donné, on définit d'abord son entropie de branchement à droite (ou **RBE** pour *Right Branching Entropy*) comme étant l'entropie des tokens qui suivent ce n-gramme.

Prenons par exemple le n-gramme « harry ». Dans les textes analysés, il sera suivi 30 fois par « potter », 6 fois par « est », et 4 fois par « truman ». La RBE de « harry » est donc de :

$$-\frac{30}{40}\log_2\left(\frac{30}{40}\right)-\frac{6}{40}\log_2\left(\frac{6}{40}\right)-\frac{4}{40}\log_2\left(\frac{4}{40}\right)\approx 1.05$$

40 étant le nombre total de tokens qui suivent « harry » (30 + 6 + 4).

Calculons maintenant la RBE du n-gramme « harry potter ». On va supposer qu'il est suivi 7 fois par « est », 5 fois par « et », 8 fois par « le », 5 fois par « dit », et 1 fois par « décida », « pensa », « prit », « tu », et « cria ». La RBE de « harry potter » est alors de :

$$-\frac{7}{30}\log_2\left(\frac{7}{30}\right)-\frac{5}{30}\log_2\left(\frac{5}{30}\right)-\frac{8}{30}\log_2\left(\frac{8}{30}\right)-\frac{5}{30}\log_2\left(\frac{5}{30}\right)-\frac{1}{30}\log_2\left(\frac{1}{30}\right)\times 5\approx 2.68$$

30 étant le nombre total de tokens qui suivent « harry potter » (...et donc le nombre de fois où « potter » suit « harry » !).

Comme il y a beaucoup plus d'incertitude sur le mot qui va suivre (probablement un mot de liaison ou un verbe, mais aucun mot ne ressort vraiment plus que les autres), l'entropie est plus élevée.

Variation d'entropie de branchement à droite (RVBE). On définit la variation d'entropie de branchement à droite (**RVBE** pour *Right Variation of Branching Entropy*)

comme d'un n-gramme comme étant la RBE du n-gramme moins la RBE du n-gramme privé de son dernier token.

Ainsi, la RVBE de « harry potter » est donc de :

$$2.68 - 1.05 = 1.63$$

Variation d'entropie de branchement à gauche. On définit de même l'entropie de branchement à gauche (**LBE**) comme l'entropie des tokens qui précèdent le n-gramme. Et la variation d'entropie de branchement à gauche comme la LBE du n-gramme moins la LBE du n-gramme privé de son premier token.

Là aussi, la LBE de « harry potter » sera élevée. La LBE de « potter » quasi-nulle (car il est quasi-systématiquement précédé de « harry »). Et donc la LVBE plutôt élevée.

Variation d'entropie de branchement normalisée. Une des avancées de la thèse de Pierre Magistry consiste à normaliser ces variations d'entropie de branchement. C'est-à-dire que pour chaque longueur de n-gramme, on va calculer leur RVBE (resp. LVBE). On calcule alors la moyenne et l'écart-type de toutes ces valeurs (pour une longueur donnée), que l'on mémorise.

Pour calculer une entropie de branchement normalisée on calcule le z-score de l'entropie de branchement, c'est-à-dire que l'on lui soustrait la moyenne que l'on a calculée et que l'on divise le tout par l'écart type.

Autonomie. Ayant les variations d'entropie de branchement normalisées à gauche et à droite, on définit alors l'autonomie d'un n-gramme comme la moyenne de ces deux valeurs.

Intuitivement, on voit donc que plus un n-gramme a une autonomie élevée, plus ce n-gramme a des chances d'apparaître tel quel dans le texte, avec les tokens qui le précèdent et ceux qui le suivent qui changent.

Ainsi, un n-gramme comme « harry potter » aura une forte autonomie. « potter » aura une autonomie négative, car il est quasi-systématiquement précédé de « harry ».

2.1.2 Segmentation du mandarin

On peut appliquer cette mesure d'autonomie à un cas particulier : la segmentation du mandarin.

Le mandarin se compose en effet d'une suite de caractères non séparés par des espaces (comme si on enlevait les espaces entre les mots du français). Une tâche préliminaire à tout algorithme de traitement du langage consiste alors à découper tout texte en « mots ».

L'algorithme va chercher pour chaque phrase, le découpage en mots qui maximise la somme des autonomies de tous les mots du découpage (on pourra utiliser de la programmation dynamique).

Les résultats obtenus avec cette méthode arrivent au même niveau que l'état de l'art des méthodes non supervisées (avec des calculs bien plus simples), mais restent en dessous de ceux des méthodes supervisées qui se basent sur un corpus d'entraînement annoté à

la main.

2.2 Cahier des charges & critères d'intérêt

L'objectif initial était donc de développer un moteur qui permettra de calculer les mesures d'autonomies de n-grammes dans un texte, ce qui pourra donc servir à la tâche de segmentation de texte (et particulièrement de mandarin), tout en proposant d'autres mesures statistiques utiles à l'extraction de termes clés, comme le TF-IDF[1] de n-grammes (mais pour l'extraction de termes clés, l'autonomie sera aussi une valeur qui nous intéressera).

Un seul moteur (ou « backend ») permettra donc d'effectuer de la segmentation de texte, mais fournira aussi des informations supplémentaires utiles à la tâche d'extraction de mots-clés.

On le verra par la suite, mais développer un tel moteur n'est pas une tâche aisée, car le volume de données à traiter est potentiellement très grand. Les n-grammes sont de plus répartis selon une loi de Zipf : quelques n-grammes vont apparaître en masse dans la collection, mais la grande majorité n'apparaîtra que très peu (et beaucoup n'apparaîtront même qu'une seule fois).

Cahier des charges du moteur pour n fixé et $k \in [\![0, n]\!]$. Il permettra les requêtes suivantes :

- Insertion d'un n-gramme (associé à l'ID d'un document).
- Nombre d'occurrences d'un k-gramme ($k \leq n$), dans le corpus d'apprentissage complet.
- Autonomie d'un n-gramme.
- Nombre de documents dans lequel un k-gramme apparait au moins une fois.
- Liste des $(k+1)$-grammes qui commencent par un k-gramme.

Ainsi que, si le backend stocke les postings, la récupération d'un objet qui représente la liste des documents dans lesquels un k-gramme donné apparait. Cet objet permettra de lister les ID des documents en question et de récupérer le nombre d'occurrences dans chaque document.

On verra toutefois que par la suite, ce cahier des charges a évolué et s'est simplifié : on a décidé de se concentrer sur les calculs d'entropie et d'autonomie, et de ne plus gérer les postings (et donc ne plus pouvoir calculer le nombre de documents dans lesquels apparait un mot, seulement la fréquence brute). Ceci a grandement simplifié le moteur (plus de détails en 2.7).

Critères d'intérêt. Si l'on souhaite comparer de manière objective les différents systèmes, il faut d'abord définir un ensemble de critères :

Durée des requêtes. Le temps moyen mis pour demander la mesure d'autonomie d'un

[1] Le TF-IDF est une mesure très courante du « poids » d'un mot dans un document, qui se base sur la fréquence du mot dans le document comparé au nombre de documents dans lesquels ce mot apparait dans une collection.

n-gramme. On est également intéressé par le temps moyen mis pour obtenir le nombre d'occurrence d'un n-gramme ou le nombre de documents distincts dans lequel il se trouve. On pourra exprimer cette mesure par le nombre de requêtes par seconde.

Durée de construction. Le temps mis pour ajouter l'ensemble des documents (donc pour construire notre modèle) est aussi un critère important. On pourra l'exprimer en tokens/seconde (nombre de tokens total divisé par le temps total).

Consommation mémoire/disque. Consommation mémoire du modèle. Soit en RAM, soit sur disque (voire dans certains cas, les deux).

Online/Offline. Une fois que notre modèle est construit, peut-on le mettre à jour dynamiquement (donc y ajouter des n-grammes) ? Si oui, le modèle est dit « online ». S'il ne l'est pas, on le qualifie de « offline » et on doit donc procéder avec une étape d'entraînement et une étape de requêtes, les deux étant totalement dissociées.

2.3 Pistes théoriques

On va maintenant étudier les différentes possibilités d'implémentation, et on comparera les résultats en pratique.

2.3.1 Réflexions préliminaires

On peut voir le stockage des n-grammes de multiples manières, parmi lesquelles :

Liste triée de n-grammes. Tout d'abord, une manière très simple pour stocker les données et qui fait une bonne introduction, consisterait à prendre les n-grammes du texte, les trier par ordre lexicographique et les écrire dans un fichier, suivi du nombre de fois ou ils apparaissent :

```
1  le petit chat 2
2  le petit chien 3
3  petit chat qui 1
4  petit chat minou 1
5  petit chien aboyant 1
6  petit chien gentil 1
7  petit chien méchant 1
```

Pour obtenir le nombre d'occurrences d'un k-gramme ($k < n$), il suffit de sommer les occurrences des n-grammes qui commencent par ce k-gramme, ce qui se fait assez facilement avec une lecture séquentielle.

Cette structure de données est conceptuellement très simple, puisqu'elle peut être construite via une série de passes :

- Extraction des n-grammes du corpus dans un fichier, en vrac.
- Tri du fichier des n-grammes extraits (via la commande `sort`).
- Recopie du fichier en comptant les lignes identiques, et ajoutant le compteur à la fin des lignes.

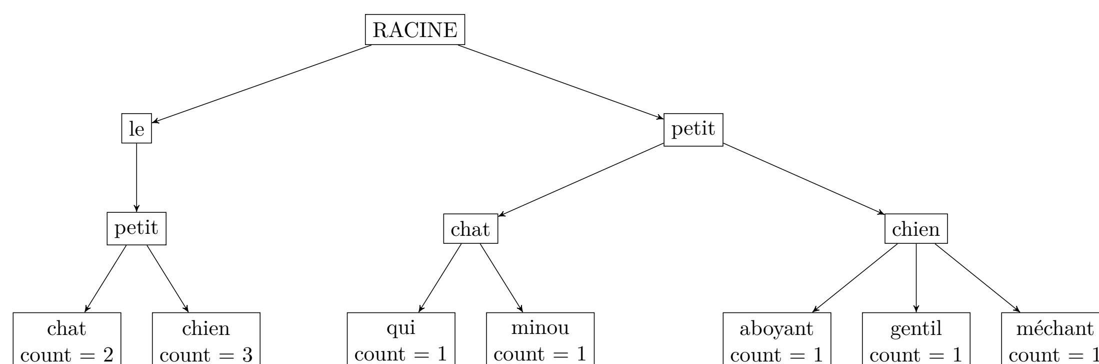


Fig. 1 – Trie

Le tout repose sur des outils éprouvés, et a une complexité linéaire en la taille de corpus, sauf pour la passe de tri (qui utilise un outil optimisé).

En utilisant les compteurs des k-grammes, on peut calculer l'entropie, et donc l'autonomie. Le seul inconvénient est que les requêtes seront très lentes : calculer l'entropie du n-gramme vide demande les compteurs de tous les 1-grammes, et donc une lecture (très rapide, certes) de tout le fichier. De plus, le modèle est « offline » : si on veut rajouter un n-gramme on doit réécrire tout le fichier en l'insérant au bon endroit.

Toutefois, cet exemple est extrêmement simple et sa construction très rapide, c'est donc une structure de données de référence que l'on pourra garder en tête.

De plus, des systèmes de stockage clé/valeur, extrêmement simples et rapides, ne sont donc pas totalement à exclure. Redis[2], par exemple, dispose même d'une fonction pour lister les clés qui commencent par une certaine valeur...

Trie. L'utilisation d'une structure de données de type *trie* est très judicieuse, et c'est celle qui est utilisée par la version initiale d'ELeVE (cf. section 2.4.1). On pourra représenter les n-grammes de la liste ci-dessus, par le trie 1.

C'est une structure de données qui se prête très bien au stockage du modèle, puisque obtenir tous les n-grammes qui commencent par un k-gramme est trivial (il suffit d'aller voir les fils du nœud pour le k-gramme en question).

De plus, on peut stocker des valeurs précalculées pour les k-grammes (nœuds qui ne sont pas des feuilles) directement dans l'arbre, ce qui est extrêmement pratique.

2.3.2 Stockage

Stockage des n-grammes. Pour n fixé, on souhaite ajouter chaque n-gramme rencontré dans notre backend. Il se charge ensuite de conserver le nombre d'occurrences de chaque n-gramme, que nous pouvons récupérer.

Toutefois, pour $k < n$, nous devons aussi pouvoir récupérer le nombre d'occurrences. Ce qui implique soit de stocker le nombre d'occurrences pour tous les k-grammes, soit

[2] `http ://redis.io`

de le déduire en sommant le nombre d'occurrences des n-grammes qui commencent par notre k-gramme.

Stockage des postings ? Une question importante est la suivante : pour un n-gramme, va-t-on stocker la liste des documents dans lesquels il apparait (avec le nombre d'occurrences). Disposer d'une telle donnée n'est pas utile pour mesurer l'autonomie, mais l'est pour calculer le TF-IDF : pour le calculer, on a besoin du nombre de documents qui contiennent au moins une occurrences du n-gramme.

On peut très bien stocker directement ce compteur, sans pour autant stocker la liste des documents. Sauf qu'avec cette méthode, on ne peut pas connaitre le nombre de documents distincts qui contiennent une occurrence d'un k-gramme (pour $k < n$).

Si on ne stocke pas les postings, on devra alors stocker le nombre de documents distincts pour tous les k-grammes, de longueur inférieure ou égale à n.

Le stockage des postings permet aussi de créer un vrai moteur de recherche d'information, ce qui peut toujours être intéressant si on souhaite l'étendre.

2.4 Solutions existantes

2.4.1 ELeVE existant

De base, Pierre Magistry a développé un prototype utilisé pour sa thèse. Il est développé en Python, avec une partie en CPython (fichier `.pyx`) pour les performances.

Il contient beaucoup de code qui ont servi pour les tests, et les deux fichiers les plus intéressants sont `LM.py` et `DTrie.pyx`. Le premier implémente l'algorithme de segmentation et quelques autres fonctions utiles, et le second implémente une structure de type Trie, avec tous les calculs d'entropie, de normalisation, etc. (je vais en parler très bientôt).

Structure Trie. La version initiale de ELeVE utilise une structure de données de type Trie, très ressemblante à la structure présentée précédemment.

Toutefois, il y a quelques différences : tout d'abord, on ne maintient pas des compteurs seulement au niveau des feuilles, mais sur tous les nœuds. Cela consomme plus de mémoire, mais permet d'éviter de devoir parcourir tout le sous-arbre du nœud qui nous intéresse jusqu'aux feuilles pour obtenir les compteurs.

Une seconde valeur est stockée dans tous les nœuds, c'est la valeur d'entropie. Initialement, elle est nulle partout.

Le modèle est un modèle *offline* : une fois que l'on a construit le trie dans son intégralité, on appelle une fonction qui va calculer les variations d'entropie et les coefficients de normalisation :

1. Calculer les valeurs d'entropie de tous les nœuds (via les compteurs des fils qui ont été tenus à jour pendant la construction).
2. Pour tous les nœuds, mettre à jour l'attribut d'entropie précédemment calculé en y soustrayant l'entropie du nœud parent (calcul de la variation d'entropie).
3. Pour chaque profondeur, calculer la moyenne et l'écart type et normaliser l'attribut entropie en mettant à jour tous les nœuds du niveau.

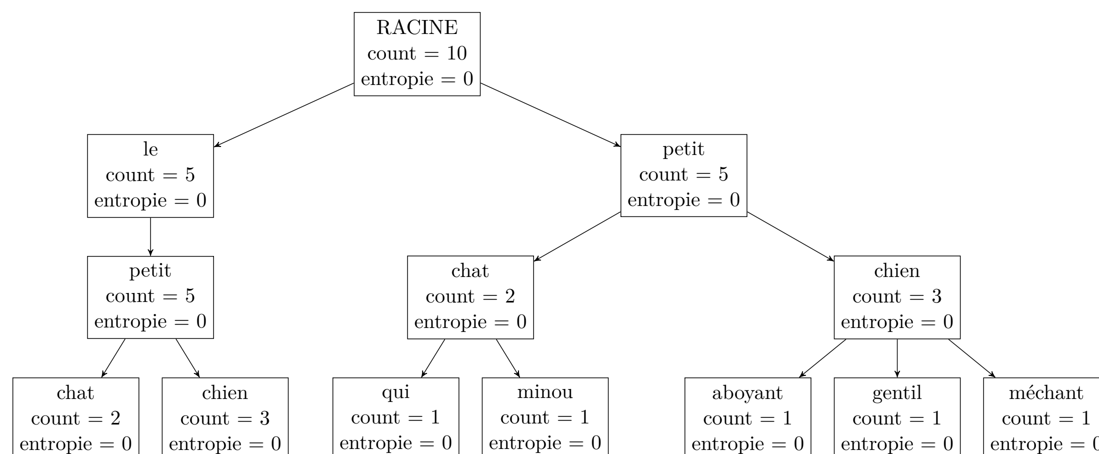


FIG. 2 – Trie d'ELeVE en cours de construction.

Ainsi, on a toute une phase de calcul après la construction du modèle qui débouche sur les variations d'entropie de branchement normalisées. Une fois ce calcul effectué, faire une requête de variation d'entropie normalisée demande juste d'accéder à la valeur calculée.

Toutefois, on ne peut plus faire une requête de l'entropie « simple » d'un nœud, vu que tous les calculs sont effectués en place, modifiant cet attribut.

Si on souhaite ajouter un n-gramme, les valeurs d'entropie de branchement normalisée ne changeront donc pas. Toutefois, on peut conserver les compteurs à jour et il suffirait alors de relancer la phase de calcul et de normalisation des entropies pour mettre à jour le modèle.

Ainsi, on obtient le trie de la figure 2 une fois les n-grammes d'exemple ajoutés :

Une fois les calculs d'entropie effectués, chaque nœud aurait sont champ entropie qui vaudrait alors l'entropie de branchement normalisé du n-gramme qui lui correspond.

Pour le stockage de cette structure, deux backends ont étés implémentés :

Stockage RAM. Ce trie peut être stocké de deux manières différentes : la première étant en RAM. Les nœuds sont des objets Python classiques, chaque nœud ayant un attribut « children » qui est un dictionnaire associant au token du fils le nœud fils en question. Il est implémenté dans `DTrie.pyx` en CPython (et donc compilé en fichier C). La consommation mémoire de ce backend est très élevée (cf. tableau 1), pour des raisons identiques à celles de mon propre backend RAM, détaillées en 2.5.1.

Stockage SQLite. Le trie peut aussi être construit et stocké dans une base SQLite. Petite différence : les nœuds ont des champs pour l'entropie de branchement, sa variation, et sa variation normalisée. On peut donc faire des requêtes sur toutes ces valeurs. Le gros inconvénient de ce backend est sa lenteur, comme on le verra dans les tests.

2.4.2 Kenlm

`Kenlm` est un outil pour générer et interroger des modèles de langues. Il analyse des grosses quantités de texte, et va en extraire tous les k-grammes ($k \leq n$). Ensuite, on pourra lui donner une phrase, et il nous donnera la probabilité qu'elle « existe » selon le corpus qu'il a utilisé pour apprendre. Pour cela, il supporte également une requête plus simple qui est la probabilité qu'un token suive un k-gramme. Pour plus d'informations, aller voir l'article de Heafield (2011).

C'est un des outils les plus performants existants pour ce genre de tâche, et même s'il ne calcule pas les mêmes statistiques que nous, son fonctionnement n'est pas si éloigné.

Par contre, il est de type *offline*, et peut fonctionner avec deux stockages distincts :

Table de hachage. S'il stocke un modèle d'ordre n, il va utiliser n tables de hachage, une table par longueur de k-gramme. C'est un modèle très performant à requêter, mais qui demande pas mal d'espace de stockage. De plus, le calcul d'autonomie se base sur le fait qu'on puisse voir tous les tokens successeurs d'un k-gramme (pour le calcul d'entropie), ce qui est impossible avec des tables de hachage. C'est donc une solution qui n'est pas envisageable dans notre cas.

Trie. Le second stockage possible est un trie, encodé sous forme de listes : de manière comparable aux tables de hachage, on va avoir n listes très longues qui stockent chacune les k-grammes pour $k \in [\![0, n]\!]$. Mais contrairement aux tables de hachage, les données sont contigües en mémoire, et peuvent être compressées de manière efficace !

Pour plus d'informations, les structures implémentées sont proches de celles décrites dans l'article de Pauls et Klein (2011).

Il serait donc théoriquement possible de modifier cette structure de données pour supporter les calculs d'entropie, mais une telle modification s'avèrera probablement très couteuse en temps puisqu'elle nécessiterait de modifier assez lourdement le cœur de Kenlm. L'ajout du support pour le TF-IDF (et pour les postings), serait encore plus lourd. Mais les structures de données et techniques utilisées ont constitué une source d'inspiration.

2.4.3 Tries en Python

Un autre point sur lequel il a été intéressant de se pencher : les implémentations de dictionnaires / tries en Python. Ceci pour deux raisons :

- Des implémentations naïves en Python utilisent souvent (on le verra) des dictionnaires dans leur structure de données. C'est une structure très volumineuse que l'on peut potentiellement remplacer par une structure plus spécialisée et efficace.
- Si on peut trouver une implémentation efficace de trie, qui peut être étendue à notre cas d'usage, on n'aurait qu'à l'adapter à nos besoins et la charge de travail serait drastiquement réduite.

Il y a deux implémentations qui ont retenu mon attention :

- marisa-trie : `https ://github.com/kmike/marisa/trie/`

— DAWG : `https ://dawg.readthedocs.org/`

Elles sont toutes les deux plutôt équivalentes, et gèrent en interne un trie. Toutefois, on n'a pas accès à sa structure, donc les opérations ne sont pas très flexibles.

Mais surtout, les clés doivent forcément être des chaines unicode. Dans le cas où on segmente du mandarin ce n'est pas un problème (nos n-grammes sont des chaines de caractères mandarins), mais si on travaille sur du français nos n-grammes sont des objets Python (des listes de chaines de caractères) et ces implémentations ne sont plus utilisables.

Les utiliser dans mon implémentation naïve pour remplacer un dictionnaire Python parait être une bonne idée, mais de même, on ne peut stocker comme valeur que des buffers. Or il faudrait stocker des pointeurs vers d'autres objets Python. On peut envisager d'écrire leur adresse encodée dans un buffer, mais ils ne seraient plus référencés et le garbage collector supprimerait les objets. Les sérialiser est inenvisageable : on perd tout l'intérêt d'un arbre si on doit tout désérialiser/resérialiser de manière récursive à chaque modification de l'arbre.

Ainsi, cette piste ne semble pas exploitable, mais l'explorer était plutôt intéressant, car dans certains cas ces bibliothèques pourraient nous fournir une solution très efficace et prête à l'emploi !

2.4.4 Xapian

Une autre piste explorée est l'utilisation d'un index inverse classique pour stocker nos données : il aurait l'avantage de permettre les requêtes de type TF-IDF très facilement, et nous simplifierait grandement la conception du backend.

Toutefois, la difficulté est de faire tous les calculs d'entropie et de les stocker. En particulier, Xapian permet d'obtenir tous les termes qui commencent par une chaine. Il permet aussi de stocker des données clé-valeur.

2.5 Travail effectué

Je présente ici les backend que j'ai développé ou explorés :

2.5.1 Backend RAM

Pour commencer, j'ai implémenté un backend en RAM. Il a pour but d'être le plus simple possible : au lieu d'un backend performant, nous voulions quelque chose de très simple qui puisse constituer une implémentation de référence la plus lisible possible (et sans bugs).

Son fonctionnement est proche du backend en RAM de l'ancienne version d'ELeVE, en gardant seulement le nécessaire : il maintient un trie en mémoire, contenant en tout point un compteur à jour du nombre de fois ou chaque k-gramme a été vu. En plus de ce compteur, chaque nœud a également un attribut avec son entropie (initialement nulle).

Il implémente aussi une méthode `update_stats` qui permet de calculer les entropies sur tous les nœuds de l'arbre, mais aussi de mettre à jour les constantes de normalisation donnant, pour chaque profondeur, la moyenne et l'écart type des variations d'entropie.

Si on effectue une requête et que des n-grammes ont été ajoutés entre temps sans appel à cette méthode, alors elle est appelée automatiquement.

Pour obtenir la variation d'entropie de branchement, il suffit d'accéder au nœud du n-gramme voulu en retenant son parent, puis de prendre l'entropie du nœud moins l'entropie du parent. Pour la normaliser, il suffit d'utiliser la moyenne et l'écart-type du niveau qui ont été calculés par `update_stats` en même temps que les entropies.

Tests automatiques. Avec ma version d'ELeVE, j'ai aussi développé une série de tests. Certains sont très simples et vérifient que les résultats sont conformes à ceux obtenus à la main, sur de très petits exemples. D'autre tirent profit de ce backend en RAM : ils génèrent des milliers de n-grammes au hasard, construisent un arbre, et comparent le résultat de très nombreuses requêtes avec les mêmes requêtes sur le backend RAM de référence.

Ainsi, même si ce backend RAM est remplacé par d'autres backends plus performants, il aura toujours un rôle en tant que référence simple aussi bien pour un humain qui voudrait lire son code que pour s'assurer que les backends plus sophistiqués sont corrects et se calquent dessus.

Consommation mémoire. Le backend RAM est difficilement utilisable en pratique à cause de sa consommation mémoire : Chaque nœud du trie est un objet Python, qui contient d'autres objets Python, dont un dictionnaire. Or, les objets Python nécessitent un peu d'espace pour définir leur type, un compteur de référence, etc. Chaque nœud contient un dictionnaire Python, qui est un objet particulièrement gros :

```
1 >>> import sys
2 >>> sys.getsizeof({})
3 288
```

Un dictionnaire Python vide prend en effet 288 octets (alors qu'il y a une optimisation pour les dictionnaires de moins de quatre éléments qui sont stockés directement dans l'objet). Comme les mots sont souvent répartis selon la loi de Zipf (loi de puissances), beaucoup de nœuds du trie contiennent des dictionnaires avec seulement quelques éléments (ou même très souvent, un seul !). Cette méthode de stockage n'est donc pas efficace du tout.

Pour limiter la consommation mémoire sans modifier la structure du code, il y a toutefois une optimisation : deux types de nœuds existent (nœuds internes ou feuilles). Les nœuds feuilles ne contiennent pas de dictionnaire de leurs fils, et les nœuds internes ne contiennent pas de dictionnaire des postings. Cette optimisation réduit presque la consommation mémoire de moitié (cf. tableau 1), comparée à ELeVE qui a la même structure sans cette optimisation.

2.5.2 Backend incrémental

Un inconvénient du backend RAM est qu'il nécessite cette phase de calcul des entropies et des coefficients de normalisation. J'ai essayé d'éliminer cette étape en créant un backend incrémental, mais je me suis rendu compte que c'était difficilement possible. Toutefois, j'en ai tiré des conclusions plutôt intéressantes et il me semble intéressant de les développer.

Entropie. Première constatation : on peut mettre à jour les entropies des nœuds affectés dès qu'on ajoute un n-gramme. On devra modifier les entropies de n nœuds (de la racine jusqu'au nœud feuille correspondant au n-gramme). Mais la mise à jour d'un nœud nécessite l'examen de tous ses fils. Ainsi, l'ajout du n-gramme « le petit chat » va nécessiter le parcours de tous les fils de la racine du trie, puis de tous les fils du nœud pour « le », etc. On a donc de nombreux parcours que l'on va chercher à éliminer.

Pour éviter de devoir parcourir tous les fils d'un nœud mis à jour, on peut calculer l'entropie de manière incrémentale.

Si c_i est notre suite des compteurs d'occurrences, et $N = \sum_{i=1}^{n} c_i$, l'entropie h va s'écrire :

$$h = -\sum_{i=1}^{n} \frac{c_i}{N} \log_2 \left(\frac{c_i}{N} \right)$$

Si on sort le $\frac{1}{N}$ de la somme, qu'on développe le $\log_2 \frac{c_i}{N}$ et qu'on développe l'expression obtenue en deux sommes, on se rend compte qu'on retombe sur la définition de N, et on peut exprimer l'entropie de cette façon :

$$h = \log_2 N - \frac{1}{N} \sum_{i=1}^{n} c_i \log_2 c_i \tag{1}$$

En conservant dans chaque nœud la somme partielle des $c_i \log_2 c_i$ au lieu de l'entropie, on a les propriétés suivantes :

— On peut calculer l'entropie via l'équation 1 avec N qui est le compteur du nœud, et la somme des $c_i \log_2 c_i$ qui est stockée dans le nœud. Ce calcul ne demande aucun parcours, juste un petit calcul.

— Si un des c_i (donc le compteur d'un nœud fils) change, on soustrait de la somme partielle l'ancien $c_i \log_2 c_i$ avant d'y ajouter le nouveau.

On peut donc calculer l'entropie de manière incrémentale. Toutefois, si on doit dans tous les cas parcourir l'arbre pour y calculer les constantes de normalisation de chaque niveau, l'intérêt est limité puisqu'on pourrait en profiter pour calculer l'entropie au passage (et ça nous épargne le petit calcul que l'on doit faire chaque fois que l'on souhaite obtenir l'entropie).

Constantes de normalisation. Pour ce qui est des constantes de normalisation des variations d'entropie, on peut aussi les calculer de manière incrémentale : il y a un algorithme tout simple pour calculer la moyenne et l'écart type d'une suite de cette

manière[3]. Il consiste à définir deux suites :

$$\begin{cases} A_0 = 0 \\ A_k = A_{k-1} + \frac{x_k - A_{k-1}}{k} \end{cases} \qquad \begin{cases} Q_0 = 0 \\ Q_k = Q_{k-1} + (x_k - A_{k-1})(x_k - A_k) \end{cases}$$

A_k vaut alors la moyenne, est Q_k est une somme partielle qui permet de calculer la variance $\sigma_k^2 = \frac{Q_k}{k}$.

Toutefois, dans notre cas il faut aussi pouvoir la mettre à jour, en ayant modifié des valeurs déjà intégrées. C'est possible pour la moyenne puisqu'il suffit d'ajouter à A la différence entre la nouvelle et l'ancienne valeur de x (le tout divisé par k).

Mais il est impossible de le faire pour l'écart-type : si on modifie une valeur, on voit bien que la modification engendrée sur la moyenne nécessiterait un recalcul de l'écart à la moyenne pour toutes les valeurs...

On n'est donc pas loin de l'objectif que l'on s'est fixé, mais il manque le calcul itératif de l'écart type de la variation d'entropie de branchement sur chaque niveau, ce qui rend le recours à une fonction de calcul obligatoire... De plus, les performances de ce backend étant très inférieures à celle du backend simple en RAM, il n'est pas inclus pas dans les tests. Mais il contient des astuces fort intéressantes, qui pourraient être réutilisées dans un backend plus efficace.

2.5.3 Backend Neo4j

Comme on l'a vu, le backend simple en RAM est très consommateur. Une autre solution simple consiste à utiliser une base de données existante.

Plutôt que de se tourner vers un SGBD SQL classique, on a décidé de garder la structure de trie utilisée pour le backend RAM, mais en la stockant dans une base de données Neo4j, qui est un SGBD orienté graphe, capable seulement de stocker des nœuds et des relations entre eux.

Le backend utilise `py2neo`[4] et le langage de requête `Cypher`[5]. Je ne vais pas parler de la structure de données, puisque c'est exactement la même que celle du backend RAM (excepté une optimisation de l'indexation des nœuds dont on va parler).

Il faut par contre savoir que les arbres ont forcément un nom, et que le nœud racine de chaque arbre est associé à une « étiquette » qui porte le nom de l'arbre. Ceci permet d'avoir plusieurs arbres dans une seule base de données Neo4J.

Ce backend était initialement extrêmement lent et inutilisable, et on va voir les différents points qui m'ont permis de l'améliorer avec les problèmes qui sont restés :

Performance des requêtes, transaction. Toute requête faite à Neo4j est faite sous forme d'une requête HTTP, ce qui a un cout non-négligeable. Si c'est une requête en

[3]Il est décrit plus en détail ici : `http ://en.wikipedia.org/wiki/Standard_deviation#Rapid_calculation_methods`
[4]Bibliothèque en Python pour Neo4j : `http ://py2neo.org`
[5]Langage de requête équivalent au SQL, mais orienté graphe. `http ://neo4j.com/developer/`

écriture, alors Neo4j va synchroniser les données sur le disque. Au final, chaque ajout d'un n-gramme nécessite n requêtes, chacune nécessitant d'écrire des données sur le disque et d'attendre qu'elles aient été écrites[6].

Pour corriger ce problème, j'ai utilisé les transactions : je groupe les requêtes d'écriture dans une transaction, qui va toutes les effectuer d'un coup en une seule requête HTTP et une seule écriture sur le disque. Les performances sont alors clairement améliorées (gain d'un ordre de grandeur).

Par contre, il n'est pas possible d'effectuer des transactions trop grosses, auquel cas la connexion HTTP avec le serveur Neo4j est réinitialisée… Il faut donc limiter la taille des transactions de manière systématique.

Indexation des nœuds. Un second goulot d'étranglement était la façon dont le serveur calculait les requêtes : lorsque j'ajoutais un n-gramme, la consultation de l'arbre était particulièrement lente. Je me suis rendu compte que de passer les tokens en attributs des nœuds, en les rendant uniques (au lieu de stocker le token, je stocke le chemin complet qui mène au nœud) et en mettant un index dessus améliorait les choses de manière drastique (performances $\approx \times 5$). J'ai donc conservé cette méthode.

À noter qu'au final, on perd un peu l'intérêt de la structure arborescente, puisque les chemins complets sont stockés dans chaque nœud...

Fusion des arbres. Pour permettre de faire des grosses transactions qui ajoutent de nombreux n-grammes d'un coup, ainsi que d'éviter que des n-grammes proches de la racine (et même la racine) soient mis à jour tout le temps, j'ai développé un système un peu différent : au lieu d'avoir une méthode qui ajoute un n-gramme, j'ai une méthode qui prend en paramètre un autre arbre (qui sera en mémoire, et dans lequel on ajoute les n-grammes), et copie son contenu dans la base de données. Ainsi, un arbre est d'abord construit en mémoire, et quand il devient trop gros il est copié dans la base de données, et on recommence la procédure avec un nouvel arbre en mémoire vide. Je vais détailler cette méthode dans la section qui va suivre.

Limitations. Une limitation reste la suppression des arbres : si la base de données contient un arbre avec des millions de nœuds, et que l'on veut le supprimer, on va faire une requête pour demander de supprimer tous les nœuds (et leurs fils) depuis la racine. Neo4j construira en mémoire une liste des nœuds à supprimer, et si la liste est trop grosse la requête ne pourra simplement pas s'exécuter, car il n'y aura pas assez de mémoire. Il n'y a pas de solution connue à ce problème[7].

`cypher-query-language/`

[6] En pratique, cela se traduisait par mon SSD qui passait son temps à travailler, avec des performances extrêmement faibles.

[7] `https ://stackoverflow.com/questions/14690522/deleting-all-nodes-and-relationships-in-neo4j-using-cypher-`

2.5.4 Backend « merge »

Pour permettre à des backends plus lents (comme Neo4j) d'être compétitifs, ou pour développer ensuite des backend en lecture seule qui seraient copiés à chaque mise à jour, j'ai développé un backend « merge ». Il prend deux backends, l'un étant en RAM et très rapide pour l'ajout de n-gramme, et le second pouvant stocker un nombre très grand de n-grammes.

Quand le backend RAM est plein, il appelle une méthode `merge` du second backend qui va alors se mettre à jour en intégrant les n-grammes stockés en RAM. Comme expliqué dans la partie sur Neo4j, ce fonctionnement permet à des backends plus lents de rester intéressants.

On pourrait même imaginer que si le backend RAM est plein, on en instancie un nouveau et on lance un merge en parallèle, pendant que le nouveau backend se remplit.

2.5.5 Backend C++

Le backend RAM naïf en Python étant particulièrement consommateur, en pratique il est difficilement utilisable[8]. Le backend Neo4j, même avec le travail effectué sur les performances, reste trop lent (cf. tableau 1).

Afin de comparer, j'ai développé un prototype de backend en C++, en RAM, avec pour objectif de limiter au maximum la consommation mémoire. Là aussi, un trie similaire à celui du backend Python est utilisé. Mais il est composé de trois types de nœuds :

- Un nœud « Token », qui contient une liste d'ID de tokens associés au pointeur vers le nœud fils pour le token en question.
- Un nœud « Feuille », qui contient une liste d'ID de documents associés à un compteur. Le compteur vaut le nombre d'occurrences du n-gramme représenté par le nœud dans le document avec l'ID en question.
- Un nœud « Index », qui contient une liste d'entiers (tokens ou ID de documents) associés au pointeur vers un nœud fils.

Dans tous ces nœuds, les données sont stockées dans un tableau contigu en mémoire, et les ID de tokens ou de documents y sont en ordre croissant. Donc l'insertion d'un n-gramme pourra nécessiter l'insertion d'un élément au milieu des listes de ces nœuds, opération ayant une complexité linéaire en la taille du tableau.

Comme un nœud (typiquement, la racine) peut avoir beaucoup de fils, il serait donc peu efficace d'avoir un seul tableau. C'est là que les nœuds « Index » entrent en jeu : si un nœud a sa liste qui devient trop grande, il est découpé en deux et remplacé par un nœud d'index, qui pointe vers les deux moitiés du nœud d'origine, et pourra transférer les insertions/requêtes à la bonne moitié. Un nœud d'index ne contient pas forcément deux fils : si un de ses fils grossit trop, il va le découper et se rajouter un fils. Si le nœud d'index a lui aussi trop de fils alors il sera découpé en deux de la même manière.

Pistes d'optimisations. Il s'avère que même si la structure en C++ prend bien moins de place que celle en Python, il est encore possible de l'améliorer significativement.

[8] Il n'était pas utilisable avec 8 Go de RAM sur les corpus les plus gros.

Prenons un n-gramme qui apparait une fois, dans un document. On devra stocker n nœuds qui sont des objets alloués sur le tas, avec chacun un objet `std::vector` de 24 octets[9]. Notre vecteur allouant sur le tas un tableau contenant un entier de 4 octets (ID du document, ou du token) et un pointeur vers le fils, de 8 octets. Pour un n-gramme qui apparait seul dans l'arbre, on a donc une consommation de :

$$n \times (\overbrace{24}^{\text{vecteur du tableau des fils}} + \underbrace{4 + 8}_{\text{tableau des fils : ID du token/document + pointeur vers le fils}}) = n \times 36 \text{ octets}$$

Pour les petits nœuds (une dizaine d'octets), il serait intéressant de pouvoir les insérer tels quels dans le nœud parent, ce qui permet d'éviter les 8 octets pour le pointeur. Comme les entiers sont quasiment toujours stockés par ordre croissant, il suffit de stocker la différence avec le dernier entier et de l'encoder en VLQ (pour *Variable-length quantity*[10]) et on a moyen de réduire énormément la taille de ce que l'on stocke : il est évident qu'on peut utiliser moins de $4 \times 36 = 144$ octets pour stocker un 4-gramme qui apparait une fois.

Si on stocke le tout en *inline* (sans pointeur), on a juste 4 ID à stocker (chacun peut faire 4 octets, sans encodage VLQ) ainsi que 4 headers de blocs qui indiquent qu'il y a un fils qui suit et qu'il est *inline* (on va dire pour être très conservateur qu'ils feront 4 octets également), et on arrive à une consommation mémoire de $(4 + 4) \times 4 = 32$ octets, plus de 4 fois moins qu'avant.

On se rapproche un peu des techniques décrites dans les articles sur le stockage des modèles de langue ou des index inversés pour les moteurs de recherche.

2.6 Premiers résultats/benchmarks

Le benchmark consiste en l'ajout de 4 millions de n-grammes de longueur 4. Les tokens sont des nombres (répartis selon une distribution exponentielle[11] avec $\lambda = 10^{-2}$). La machine de test disposant de 16 Go de RAM, il est important de constater que les backends disque font leur lecture dans le cache RAM, en pratique. Les résultats du benchmark sont en figure 1.

[1] Le backend C++ étant un prototype à l'heure du benchmark, seule la construction de l'arbre est implémentée. Je n'ai donc pas pu tester le reste.

[2] La structure de données n'intègre pas les valeurs d'entropie, si elle devait l'intégrer plutôt que de les calculer à la volée. La consommation devrait donc probablement être un peu supérieure.

[3] J'ai dû arrêter le script au bout de quelques heures puisqu'il ne terminait pas. Au bout de 3h il était en train de calculer les statistiques d'entropie.

[9] Obtenu avec `sizeof(std::vector<T>)`, sur une machine 64 bits.

[10] `https://en.wikipedia.org/wiki/Variable-length_quantity`

[11] Fonction `random.expovariate`(λ) de la bibliothèque standard de Python.

Backend	Temps de construction (dont mise à jour)	Temps de requête	Conso. Mémoire
ELeVE Pierre en RAM	82s (33s)	512s	5,3 Go
ELeVE Pierre SQLite	1605s (746s)	2242s	800 Mo (1,2 Go disque)
RAM	100s (31s)	362s	2,9 Go
Merge(RAM + Neo4j)	>3h[3]	N/A[3]	50 Mo (4 Go disque)
C++	35s (N/A[1])	N/A[1]	360 Mo[2]

Tab. 1 – Benchmark des différents backends

Conclusion. On voit que la consommation mémoire varie énormément d'un backend à l'autre, mais est globalement élevée. On doit pourtant pouvoir faire les expérimentations rapidement et en RAM, ce qui est n'est pas commode avec le backend RAM naïf et 8 Go de RAM.

Ensuite, pouvoir traiter de gros corpus est indispensable, donc il est nécessaire d'avoir un backend disque. Une idée qui a été développée ensuite pour pallier à ces problèmes est une structure de données totalement précalculée et stockée sur disque, en lecture seule. Si on veut l'agrandir, on la fusionne avec un arbre en mémoire (qui doit contenir le plus de données possible, d'où l'importance d'un backend RAM efficace).

Pour les calculs d'autonomie, on doit stocker une seconde structure pour le texte de droite à gauche. Il faudrait que cette structure contienne seulement les compteurs d'occurrences (pour calculer les entropies) : pas besoin de stocker les postings ou toutes les informations nécessaires pour les requêtes de type recherche d'information, puisque ce sont les mêmes de droite à gauche ou de gauche à droite.

Pertinence d'un seul moteur ? On s'est ensuite posé des questions sur la pertinence d'utiliser un moteur pour les calculs d'autonomie et les statistiques type TF-IDF. En effet, les calculs d'autonomie impliquent de construire deux modèles (de gauche à droite et de droite à gauche), avec une profondeur de $n+1$ si on veut les autonomies sur des n-grammes.

D'un autre côté, les statistiques comme le TF-IDF ne nécessitent pas cette profondeur supplémentaire. De plus, le fait de construire un modèle de droite à gauche est complètement redondant, puisqu'on y trouvera les mêmes informations. Il faudrait au moins pouvoir désactiver le stockage des postings dans l'arbre construit à l'envers.

2.7 Seconde itération : Leveldb

Au vu des performances décevantes des divers backend, des problèmes de complexité en stockant les postings seulement sur les feuilles, et n'ayant toujours pas de backend utilisable sur de gros corpus, il a été nécessaire de réfléchir à nouveau au problème dans un second temps, à partir des conclusions qu'on a émises.

Nous avons fini par décider de recentrer le backend sur sa tâche première : les calculs d'autonomie. Et donc de supprimer toute la gestion des postings, qui peut être faite par un autre moteur spécialisé (type Xapian ou Elasticsearch) faisant très bien ce genre de

tâche.

Ainsi :

- Un travail de refactoring/simplification a donc été effectué, pour ne garder que l'essentiel dans eleve.
- Le backend C++ a été terminé (ce qui a demandé beaucoup de travail pour corriger quelques bugs coriaces). Le but étant d'avoir un backend RAM très efficace et peu consommateur.

Ensuite, l'idée était d'utiliser ce backend RAM efficace combiné à un nouveau backend disque en lecture seule, lui aussi en C++. Ce qui permettrait de créer un backend merge, qui stockerait les données temporairement en RAM, avant de faire des fusions périodiques avec le disque (le moins possible, d'où l'intérêt d'un backend RAM efficace). On pourrait alors interroger la structure sur disque.

La structure sur disque contiendrait, elle, n listes triées (à la façon de Kenlm).

C'est en pensant à la façon exacte de stocker le tableau associatif des tokens vers leurs IDs sur disque que je me suis intéressé de près à Leveldb. Il s'agit d'un système de stockage clé-valeur extrêmement efficace, qui, en interne, stocke une liste triée comme je souhaitais le faire dans mon backend disque. Je me suis alors rendu compte qu'il serait envisageable d'utiliser Leveldb pour stocker également mes n listes triées.

J'ai alors décidé de consacrer un peu de temps à une expérimentation avec Leveldb qui s'est avérée fructueuse :

Leveldb. Leveldb[12] est une base de données clé-valeurs qui se présente sous la forme d'une bibliothèque C++. Il a été conçu en restant extrêmement simple, et, basiquement, stocke un tableau associatif clé-valeurs par ordre lexicographique (ce qui est donc exactement mon idée originale).

Par contre, il permet aussi l'ajout de nouvelles données. Pour cela, il utilise un système de « niveaux » (d'où le nom Leveldb) : de nouvelles données sont ajoutées dans un tableau plus petit. Une fois qu'il atteint une taille critique, il est fusionné avec le tableau de niveau supérieur qui est reconstruit. Une documentation[13] détaille ce processus.

Ainsi, il est particulièrement adapté à une phase comprenant de nombreuses écritures. Pour les calculs d'entropies, qui demandent de lister tous les fils d'un nœud, il s'avère extrêmement efficace puisqu'il stocke des listes triées qui sont triviales à parcourir avec un itérateur. Et pour les requêtes d'entropie, il s'agit de récupérer un nœud (par sa clé), ce qui est là encore trivial et très rapide.

Comme c'est une simple bibliothèque, le cout d'appel aux fonctions d'ajout et de mise à jour est quasi-nul. Les clés et valeurs sont des buffers contenant des données arbitraires.

Stockage. Pour stocker les n-grammes, je stocke donc n listes dans Leveldb (en plus de la racine).

Le premier octet de la clé indique la taille du k-gramme. Ensuite, pour chaque token on ajoute a la clé un octet nul puis le token en question. Voici quelques listes représentant

[12] `https ://github.com/google/leveldb`

[13] `https ://leveldb.googlecode.com/svn/trunk/doc/impl.html`

des k-grammes avec la clé correspondante :

```
[] -> "\x00" # noeud racine
["le"] -> "\x01\x00le"
["le", "petit"] -> "\x02\x00le\x00petit"
["le", "petit", "chat"] -> "\x03\x00le\x00petit\x00chat"
```

Le fait de préfixer les clés par leur taille permet de m'assurer que, pour k fixé, tous les k-grammes sont stockés de manière contigüe dans une même liste. C'est une sorte de stockage du trie parcouru en largeur. On ne les préfixerait pas par leur longueur, on aurait un stockage dans une seule liste, d'un parcours en profondeur du trie.

La valeur associée a la clé est un couple de flottants avec le compteur d'occurrences et l'entropie.

Implémentations. J'ai d'abord implémenté ce backend en Python, qui dispose d'un wrapper pour Leveldb très bien conçu, Plyvel[14]. Au final, le code du backend est particulièrement court et simple.

Je me suis rapidement rendu compte que le plus lent n'était pas les appels à Leveldb, mais le reste du code Python (très bonne nouvelle). J'ai donc restructuré le code pour le rendre le plus rapide possible, sans perdre en lisibilité.

Puis j'ai créé un fichier `.pxd` qui définit statiquement les types de données utilisés et permet d'utiliser `cython` pour compiler ce code Python en C, et donc gagner en performance.

Me rendant compte que du code généré avec `cython` devait quand même repasser par le wrapper pour leveldb et restait beaucoup trop utilisateur de Python, j'ai implémenté mon propre backend C++ utilisant Leveldb directement. Ce backend n'est au final pas très complexe et s'est avéré être le plus rapide.

2.8 Benchmark finaux

Contrairement à la première partie ou le benchmark s'effectuait juste sur un trie, il s'agit d'un benchmark effectué en indexant un corpus avec toutes les informations nécessaires pour calculer l'autonomie (script `autonomy-gold.py`).

Le temps de construction représente le temps mis pour ajouter toutes les phrases au backend. Le temps de mise à jour (Màj dans le tableau 2) représente le temps mis par le backend pour calculer les entropies et constantes de normalisation. Le temps de requête représente le temps mis pour faire diverses requêtes d'autonomie et de compteurs sur le backend.

On constate donc que le backend RAM en C++ est d'une efficacité très satisfaisante, aussi bien en terme de performance qu'en terme de consommation mémoire. Le backend Leveldb, quant à lui, est bien plus lent, mais reste suffisamment rapide pour être utilisé en pratique : ce dernier a été capable d'indexer l'intégralité de Wikipédia en français en moins d'une semaine. Le modèle final obtenu faisant une vingtaine de gigaoctets.

[14] `https ://plyvel.readthedocs.com`

Backend	Temps de construction	Temps de Màj	Temps de requête	Conso. Mémoire
RAM	1m 10s	59s	1m	2,5 Go
RAM C++	10s	2s	20s	100 Mo
Leveldb	7m 6s	11m 1s	7m 46s	193 Mo disque
Leveldb (cython)	5m 12s	8m 59s	5m 55s	193 Mo disque
Leveldb (C++)	3m 4s	3m 2s	3m 54	193 Mo disque

Tab. 2 – Benchmark final des backends

Il est intéressant de noter les différences de performance entre les trois versions des backend Leveldb. L'existence de ces différences de performances est une bonne nouvelle, car elle veut dire que l'on n'est pas dans une situation où tout le temps est passé dans le système de stockage disque, comme c'était le cas avec Neo4j : ici, sérialiser des n-grammes en Python pur (qui est une tâche dépendant seulement du CPU) devient au moins aussi lent que leur insertion dans Leveldb. D'où l'intérêt du backend Leveldb C++ qui fait en sorte de réduire au maximum tous ces calculs.

Conclusion. Nous avons donc obtenu un système de calcul d'autonomie constitué de deux *backends* complémentaires : l'un en RAM, extrêmement performant et capable de travailler sur des corpus relativement gros, et l'autre sur disque qui a quand même des performances tout à fait acceptables.

L'outil développé est spécialisé uniquement dans le calcul d'entropie/d'autonomie et ne contient pas toutes les données nécessaires au calcul du TF-IDF. Ce qui le rend d'autant plus simple, et donc il a plus de chance d'être réutilisé/modifié s'il est utilisé dans ce contexte précis.

Toutefois, il permet quand même d'avoir accès aux fréquences brutes (compteurs d'occurrences), et pourrait donner accès aux tokens qui suivent un n-gramme, etc.

3 Autonomie & extraction de termes clés

Nous présentons ici le second objectif du stage, à savoir étudier l'apport de l'autonomie dans la tâche d'extraction de termes clés.

Tout d'abord, la section 3.1 présente l'état de l'art des méthodes d'extraction de mots-clés. Puis en 3.2 nous présentons la méthodologie générale adoptée pour l'évaluation, et les choix effectués pour cette dernière. Ensuite les corpus d'évaluation sont présentés en section 3.3, et enfin nous verrons les résultats et interprétations en 3.4.

Comme vu en 1.2, l'extraction de termes clés consiste en la sélection d'un petit nombre d'expressions (pouvant être des mots seuls, ou des termes composés de quelques mots) représentant le mieux possible un texte. C'est une définition qui n'est pas forcément très précise, car dans certains cas on cherchera à extraire des termes d'un thesaurus (vocabulaire) correspondant au domaine des documents, dans d'autre cas à trouver des « tags » pour faciliter la recherche ou l'indexation de documents, ou encore trouver quelques expressions dans chaque document pour faciliter la compréhension au premier abord par les lecteurs, etc.

3.1 État de l'art

Les algorithmes d'extraction de termes clés peuvent se classer en deux méthodes.

Méthodes supervisées. Partant d'un jeu de documents annotés, c'est-à-dire pour lequel on a des mots-clés de référence, on va typiquement appliquer un algorithme de machine learning qui va, pour un jeu de documents inconnus, essayer d'en déduire quels sont les mots-clés qui nous intéressent. Pour cela, on doit trouver des caractéristiques pertinentes (nombre d'occurrences du mot-clé dans le document, étiquetage morphosyntaxique...).

Méthodes non-supervisées. Ce sont des méthodes qui vont principalement effectuer des statistiques sur les documents, et qui ne vont pas avoir besoin d'exemple de documents avec mots-clés de références.

Indépendamment du fait qu'ils soient supervisés ou non, les algorithmes d'extraction vont sélectionner les termes à extraire basés sur leurs caractéristiques (*features*). Ils utilisent principalement les *features* suivantes.

TF-IDF du terme (l'IDF étant calculé par rapport au corpus de test ou à un corpus externe).

Position de la première occurrence du terme dans le document. Un terme au milieu du document étant probablement moins important qu'un terme dans l'introduction ou dans la conclusion.

Keyphraseness : le mot-clé apparait-il fréquemment dans le *gold standard* de référence ?

Longueur du terme en mots. Des termes plus longs ayant plus facilement un sens non ambigu.

Étiquettage grammatical des mots qui composent le terme. Par exemple, un nom suivi d'un adjectif semble plus pertinent qu'un verbe.

D'autres *features* plus exotiques sont aussi utilisées :

- le degré d'incidence, ou le Pagerank du terme dans un graphe qui lie des termes entre eux. Ce graphe peut être construit à partir de Wikipédia, d'un graphe de cooccurrence des mots (Mihalcea et Tarau 2004), etc.
- l'appartenance à un thesaurus externe,
- la présence dans des historiques de requêtes sur des moteurs de recherche,
- proximité sémantique avec les autres candidats (avec par exemple des statistiques de cooccurrence),
- dispersion du mot dans le document : distance entre sa première et dernière occurrence, pour détecter des mots dans l'introduction et la conclusion.

Des approches plus originales ont aussi été présentées. Par exemple, Liu et al. (2010) proposent d'utiliser une LDA (*latent dirichlet allocation*) sur chaque document afin de rechercher les articles de Wikipédia les plus semblables et de renvoyer les titres de ces articles.

Pour un état de l'art plus détaillé, nous renvoyons à la revue de référence Hasan et Ng (2014). En complément, il est intéressant de citer Beliga (2014) et Bougouin (2013).

3.1.1 Évaluation

Il faut un moyen d'évaluer toutes ces méthodes d'extraction, et le les comparer entre elles. Pour cela, on se base en général sur un corpus de documents où chaque document est annoté d'une liste des termes intéressants extraits par des humains : c'est le *gold standard*.

On fait alors tourner les systèmes d'extraction sur ces mêmes documents, et on peut calculer le rappel, la précision et le F-score, mesures d'évaluation classiques (elles sont définies en 3.2.1).

Toutefois, si on extrait un terme très proche d'un terme du *gold* mais légèrement différent (au pluriel alors qu'il est au singulier dans le *gold*, par exemple), il sera compté comme différent. Pour éviter ce genre de soucis, d'autres méthodes d'évaluation ont vu le jour (comme ROUGE ou BLEU), et le sujet de l'évaluation de ces méthodes est un domaine de recherche à part entière qui a fait l'objet de nombreuses publications. Plus d'informations sont données dans Kim, Baldwin et Kan (2010) (qui cherche à *évaluer les méthodes d'évaluation*).

Cohérence. Une mesure qui s'avère très pertinente, mais qui est bien plus dure à mettre en pratique est la mesure de cohérence. Elle est décrite par Medelyan, Frank et Witten (2009), et décrit la proximité entre deux extractions de termes clés d'un même document.

On peut la calculer sur un système ou une personne comme la moyenne sur tous les documents de la moyenne de la proximité avec les extractions des autres systèmes/personnes.

Pour la mettre en pratique, il faut toutefois disposer d'extractions de références faites par des humains, et disposer d'un bon nombre de ces annotations. Un tel corpus est CiteULike, décrit dans l'article cité ci-dessus.

3.2 Méthodologie

On développe ici la façon dont on mesure l'impact de la prise en compte de l'autonomie. Pour cela, on construit un procédé expérimental que l'on va ensuite mettre en œuvre.

Pour évaluer l'intérêt de l'autonomie, on se base sur des évaluations classiques basées sur le rappel, la précision et le F-score (3.2.1), pour des méthodes d'extraction de termes clés simples. On va ensuite modifier ces méthodes d'extraction pour y ajouter une prise en compte de l'autonomie (via des filtres, en section 3.2.2), et voir si les scores sont meilleurs ou moins bons. On s'intéresse aussi au pourcentage de capture sur le corpus comparé à celui sur le gold (3.2.3).

3.2.1 Évaluation via le F-score.

Tout d'abord, des mesures incontournables, et qui nous permettront de nous comparer aux algorithmes existants : la précision, le rappel et le F-score.

Le rappel représente le ratio du nombre de termes corrects extraits par le nombre de termes à trouver au total. La précision représente quant à elle le ratio du nombre de

FIG. 3 – Affichage des résultats de l'extraction de l'algorithme baseline.

termes corrects extraits par le nombre de termes extraits.

Le F-score est la moyenne harmonique de la précision p et du rappel r :

$$f_1 = \frac{2 \times p \times r}{p + r}$$

Ces mesures peuvent permettre de se comparer avec les autres implémentations, sur des corpus de référence contenant des textes associés à leur *gold standard* (termes de référence extraits du texte).

Algorithme d'extraction baseline. On effectue nos évaluations sur une méthode d'extraction consistant à trier tous les n-grammes du texte par TF-IDF décroissant. Notre façon de prendre en compte l'autonomie qui s'est avérée la plus simple et efficace consiste à filtrer les n-grammes candidats en éliminant ceux qui ont une autonomie trop faible.

Une fois qu'on a filtré puis trié les n-grammes candidats, on garde les 20 premiers qui sont considérés comme les termes du gold. C'est une méthode très simple, mais qui nous permet d'évaluer facilement l'influence du filtrage sur l'autonomie.

Au préalable, j'ai essayé d'autres manières de prendre en compte l'autonomie (calculer un score à partir du TF-IDF et de l'autonomie sans aucun filtrage), mais c'est le filtrage qui s'est avéré être la méthode la plus efficace.

3.2.2 Filtres

Entrainement sur des corpus externes. Avant de lister les filtres, il faut savoir que, l'autonomie étant calculée à partir d'un corpus de référence, on a pris en compte deux mesures différentes : la première à partir d'une mesure d'autonomie entrainée sur tout le corpus (qui sera donc très performante concernant les termes spécifiques au corpus), et la seconde à partir d'une mesure d'autonomie entrainée sur Wikipédia en anglais (corpus très gros contenant énormément de termes techniques, donc fort intéressants).

Filtres Les filtres sont les suivants :

aucun pas de filtrage ;

autonomie corpus >1 seuil de 1 sur l'autonomie (entrainée sur le corpus) ;

autonomie wp >1 seuil de 1 sur l'autonomie (entrainée sur Wikipédia) ;

count wp >4 filtrage des n-grammes qui apparaissent moins de 4 fois dans Wikipédia ;

stopwords filtrage des n-grammes qui commencent ou finissent par un mot vide (liste de `nltk.corpus.stopwords`) ;

Les filtres « stopwords et ... » filtrent à la fois ceux qui commencent ou finissent par un mot vide, et ceux qui sont filtrés par le second filtre.

Ces filtres sont déjà des résultats d'un processus d'expérimentation : le seuil de 1 n'a pas été choisi au hasard, mais est empiriquement un des plus intéressants. Le filtre qui seuil sur le nombre d'occurrences des n-grammes dans Wikipédia n'a rien à voir avec l'autonomie, mais a été obtenu par un arbre de décision qui l'a détecté comme étant un très bon critère. Le filtrage par stopwords est une sorte de pré-filtre qui permet d'améliorer les résultats de manière importante sans perte significative de termes du gold, et qui semblait être très complémentaire à l'autonomie.

3.2.3 Évaluation via la capture corpus/gold.

Nous mesurerons aussi la proportion des n-grammes du corpus gardé par les filtres par rapport à la proportion des termes du *gold* que l'on garde. Le but étant évidemment d'éliminer un maximum de candidats des documents tout en gardant 100% (ou presque) du gold.

On ne mesure plus une hausse ou une baisse d'efficacité de l'algorithme comme pour le F-score, mais il s'agit d'une mesure objective de l'efficacité de nos filtres sur l'autonomie, qui est donc très complémentaire.

Pour trouver un seuil convenable sur l'autonomie, on a généré de telles courbes de capture corpus/gold en faisant varier le seuil. On obtient les courbes en figure 6. Pour ces courbes, l'autonomie est entrainée sur les corpus, et non sur Wikipédia. Si chaque corpus comprend plusieurs courbes qui font varier le seuil, c'est, car il y a un second critère de nombre d'occurrences minimales, qui permet d'éliminer les autonomies considérées comme non fiables, car se basant sur trop peu d'occurrences. Ces courbes nous ont fait choisir un seuil de 1 sur l'autonomie.

3.3 Jeux d'évaluation

Nous présentons ici les jeux d'évaluation que nous avons utilisés, qui sont des collections de documents (scientifiques, pour la plupart) assortis de termes pertinents qui représentent le document (*gold standard*). Ce sont des jeux de données de la littérature, et qui sont librement accessibles.

Le tableau 3 donne une synthèse des caractéristiques des différents corpus.

J'ai converti tous les jeux d'évaluation au format MAUI[15]. Cet outil propose de nombreux jeux d'évaluation et son format et très simple : une archive contenant des fichiers `.txt` et `.key` pour le *gold standard*.

Inspec. Ce corpus provient de l'article de Hulth (2003), article référence du domaine qui présente des approches supervisées de base et leurs résultats. Le meilleur F-score observé, indépendamment de la méthode, est de 33.9 (méthode supervisée travaillant sur l'ensemble des n-grammes).

Ce corpus est composé d'abstracts d'articles scientifiques dont les mots-clés ont été extraits par des experts, en deux *gold standard* :

controlled, termes qui font partie du thesaurus associé à la collection ;

uncontrolled, termes choisis sans aucune restriction.

Le corpus `controlled` nous intéresse peu, car 80% des termes du *gold standard* ne sont pas présents dans les textes. On utilisera donc le corpus `uncontrolled` pour nos évaluations (30% de termes introuvables). La plupart des termes à trouver sont des digrammes.

Des exemples de documents avec les gold associés sont en figure 4.

semeval2010-maui. Ce corpus (Kim, Medelyan et al. (2010)) contient plusieurs centaines d'articles scientifiques de l'« ACM Digital Library », dans des domaines distincts. Ils ont été convertis en texte par `pdftotext`, mais le texte généré a été soigneusement analysé pour éviter des mots incorrectement convertis depuis le PDF. Les termes extraits ne proviennent pas d'un thesaurus ou vocabulaire particulier.

Nous ne donnerons pas d'exemple, puisqu'il s'agit d'articles de recherche variés et qu'ils ont un format assez hétérogène. Ils sont plutôt longs, et peuvent contenir des formules (converties en texte), etc.

citeulike180. Ce corpus provient de Medelyan, Frank et Witten (2009), et contrairement à ceux du dessus il a été généré par des utilisateurs lambda qui ont tagué des articles scientifiques. Un filtre très strict a été appliqué pour ne retenir que les articles tagués plusieurs fois par des utilisateurs différents.

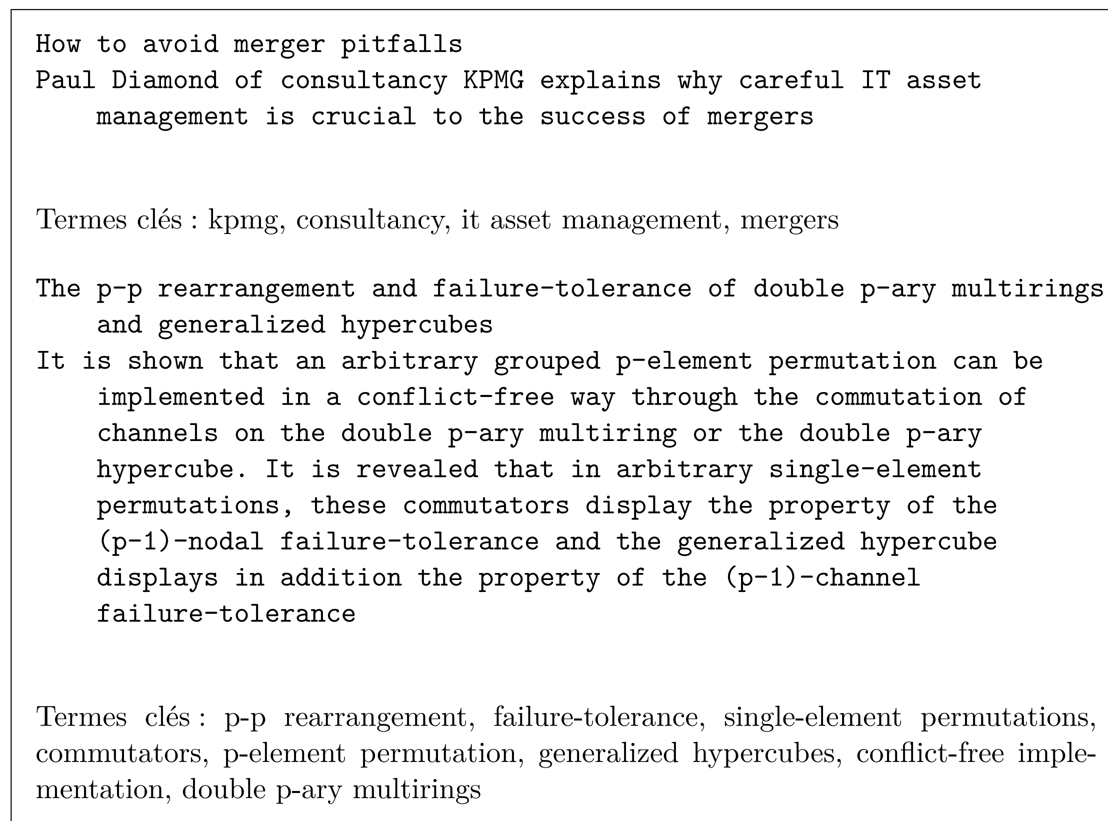
Le corpus initial était constitué d'un dossier listant les tags de chaque utilisateur (ce qui sert à calculer la cohérence). Ne la calculant pas, je l'ai converti en deux corpus :

citeulike180-all qui contient la fusion des tags des utilisateurs ;

citeulike180-filtered qui contient les tags ayant au moins deux utilisateurs l'ayant choisi.

Sur citeulike180-all, 32% des termes ne sont pas présents dans les textes sources (le rappel maximum est donc de 68% si on ne fait que de l'extraction depuis les textes). La grande majorité des termes à extraire sont des mots simples. Cf tableau 3.

[15] `https ://code.google.com/p/maui-indexer`



```
How to avoid merger pitfalls
Paul Diamond of consultancy KPMG explains why careful IT asset
    management is crucial to the success of mergers
```

Termes clés : kpmg, consultancy, it asset management, mergers

```
The p-p rearrangement and failure-tolerance of double p-ary multirings
    and generalized hypercubes
It is shown that an arbitrary grouped p-element permutation can be
    implemented in a conflict-free way through the commutation of
    channels on the double p-ary multiring or the double p-ary
    hypercube. It is revealed that in arbitrary single-element
    permutations, these commutators display the property of the
    (p-1)-nodal failure-tolerance and the generalized hypercube
    displays in addition the property of the (p-1)-channel
    failure-tolerance
```

Termes clés : p-p rearrangement, failure-tolerance, single-element permutations, commutators, p-element permutation, generalized hypercubes, conflict-free implementation, double p-ary multirings

Fig. 4 – Exemple de documents/gold du corpus Inspec.

NLM-500 et fao780. Ces corpus utilisent tous les deux un thesaurus (comme inspec-contr), et sont donc moins intéressants pour nous. Ils proviennent là encore de Medelyan, Frank et Witten (2009).

fao780 contient 780 publications du FAO (de taille variable, souvent très longue). NLM-500 contient 500 publications médicales, de taille moyenne.

Le tableau 3 présente quelques statistiques utiles à une première approche des jeux d'évaluation.

Pour limiter la quantité de résultats, nous avons utilisé citeulike180-all comme jeu d'évaluation de référence, car il contient plus de mots dans le *gold* que les autres, ce qui permet d'avoir des F-scores relativement élevés, et donc les variations de ce dernier sont plus pertinentes. Il est aussi un représentant très correct des autres corpus vis-à-vis des variations des résultats en passant d'une méthode à l'autre. C'est donc sur ce corpus que je travaillerai en priorité, bien que les résultats des autres corpus soient aussi fournis.

Nom du corpus	Taille compressée	Nb. docs.	mots/doc	Nb. evals	termes/eval nb. tokens	% termes introuvables
inspec-uncontr	746.7 Kio	2000	124.4	500	9.8 1.3 + 5.2 + 2.4 + 0.9	30.8%
semeval2010-maui	3.8 Mio	244	8020.5	100	15.2 3.1 + 8.2 + 2.9 + 1.1	17.6%
citeulike180-all	2.7 Mio	183	6994.6	183	17.4 13.4 + 3.5 + 0.5 + 0.0	32.4%
citeulike180-filtered	2.7 Mio	183	6994.6	183	5.2 4.4 + 0.7 + 0.1 + 0.0	15.2%
inspec-contr	721.8 Kio	2000	124.4	500	4.5 1.0 + 2.6 + 0.8 + 0.1	81.6%
NLM-500	4.5 Mio	500	4757.0	500	14.2 5.1 + 6.2 + 2.3 + 0.7	64.1%
fao780	45.0 Mio	779	29473.7	779	8.0 3.4 + 4.3 + 0.3 + 0.0	28.0%
TALN_full	10.4 Mio	1063	4333.2	5	15.6 5.2 + 7.4 + 2.4 + 0.6	2.6%
TALN-abstract	308.7 Kio	990	122.5	5	15.6 5.2 + 7.4 + 2.4 + 0.6	0.0%

TAB. 3 – Caractéristiques des jeux d'évaluation. Le champ « nb. tokens » avec la somme désigne le nombre de 1-grammes, 2-grammes, 3-grammes, et 4-grammes ou plus, dans le *gold* de chaque document.

3.4 Évaluation

On analyse dans cette section les résultats obtenus via notre algorithme d'extraction assorti des différents filtres développés.

Le tableau 4 présente la précision le rappel et le F-score obtenus avec l'algorithme baseline d'extraction par TF-IDF, et différents filtres.

filtre	précision	rappel	F-score	capture corpus	capture gold
aucun	11.89	16.07	13.66	100.0	100.0
autonomie corpus >1	13.25	17.8	15.19	60.62	88.75
autonomie wp >1	12.46	16.64	14.25	76.99	91.28
count wp >4	13.11	17.52	15.0	48.15	88.21
stopwords	12.79	17.18	14.66	44.54	**99.08**
stopwords et autonomie corpus >1	13.61	18.24	15.59	28.54	87.9
stopwords et autonomie wp >1	12.95	17.35	14.83	36.72	90.55
stopwords et count wp >4	**14.26**	**19.07**	**16.32**	**17.65**	87.49

TAB. 4 – Métriques évaluées sur citeulike180-all. Les colonnes « capture corpus » et « capture gold » présentent respectivement la proportion de n-grammes conservés après l'application de chaque filtre respectivement dans le corpus et dans le gold.

Des résultats pour les autres corpus sont disponibles en annexe (du tableau 5 à 10)

Des nuages de points des proportions de capture des n-grammes du gold et de ceux du corpus sont toutefois plus visuels et faciles à interpréter. Une telle figure pour notre corpus de référence est en 5.

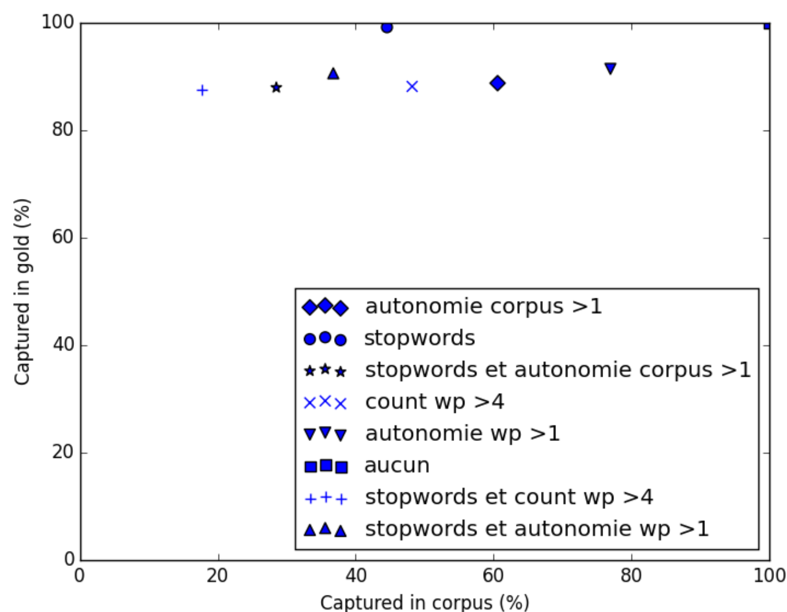


Fig. 5 – Capture gold/corpus sur citeulike180-all.

Observations générales. L'extraction de termes clés est une tâche assez délicate, et des F-scores faibles par rapport aux *gold standard* sont normaux, puisque même un humain ne serait pas capable d'extraire la même chose que dans le *gold standard* : c'est une tâche très subjective. Les scores obtenus sont plutôt cohérents par rapport à la littérature, en particulier pour une méthode non-supervisée qui ne peut pas s'adapter à la manière d'annoter les corpus, ou réutiliser des mots-clés déjà apparus dans le *gold standard* sur des documents similaires.

Ainsi, notre *baseline* par TF-IDF et filtre sur l'autonomie donne des résultats relativement faibles. Mais il suffit qu'on note une nette amélioration via un filtre sur l'autonomie pour montrer que l'autonomie est une mesure utile à la tâche d'extraction de mots-clés.

De plus, même si le F-score n'augmente pas forcément, l'intérêt du filtrage est aussi de supprimer un maximum de candidats du corpus. Un filtre qui garde très peu de termes du corpus sera très intéressant à utiliser en amont d'un algorithme complexe d'extraction de mots-clés afin de réduire drastiquement le nombre de termes candidats et donc d'améliorer les performances.

Autonomie. On constate donc que la plupart des filtres conservent autour de 90% des termes du gold (éliminant les termes les plus particuliers, qui n'auraient pour la plupart pas un bon TF-IDF), tout en conservant de moins de la moitié des termes du corpus, voire jusqu'à seulement 20%.

En outre, on observe dans tous les cas une amélioration du F-score, de 1 à 3 points, ce qui témoigne de l'apport de nos filtres.

Le filtrage stopwords est intéressant en tant que pré-filtre, car il capture quasiment tout le gold, et arrive à éliminer la moitié du corpus. Il est de plus très complémentaire avec les filtrages sur l'autonomie, et c'est sa combinaison avec d'autres filtres qui apporte les meilleurs résultats.

Par contre, les améliorations apportées par l'autonomie sont à nuancer par le filtre sur le nombre d'occurrences dans Wikipédia : ce filtre, assez trivial et n'utilisant pas l'autonomie, s'avère être plus efficace que ceux basés sur l'autonomie. C'est aussi celui qui élimine la plus grande quantité de termes du corpus, de loin (avec 17% des termes gardés).

Analyse de l'autonomie. L'analyse des termes éliminés du gold car ayant une autonomie faible est très intéressante, et est une des pistes de réflexion pour des études complémentaires. La figure 7 en annexe présente un extrait des détails de n-grammes (autonomies, compteurs d'occurrences, variation d'entropie, etc.), pour des termes du gold de notre corpus de référence.

On voit que de nombreux termes sont composés de deux mots très communs, ce qui va forcément donner une autonomie assez faible (car la variation d'entropie ne sera pas favorable). C'est le cas pour un terme comme « small world » (visible en figure 7), qui a une autonomie Wikipédia très faible, car les deux mots qui le composent sont présents des centaines de milliers de fois dans Wikipédia, ce qui donne une variation d'entropie très négative.

Par exemple, on pourrait tenter de définir une nouvelle mesure inspirée de l'autonomie, mais qui prendrait en compte le fait qu'un couple de mots très communs peut nous intéresser même si son autonomie est faible. Ou encore, étudier l'intérêt d'une combinaison entre les autonomies du corpus et de Wikipédia.

Il faudrait aussi étudier plus en détail l'influence de la taille des n-grammes sur l'autonomie : il est possible que selon la longueur du n-gramme candidat, il faille mettre des seuils différents. L'autonomie étant la variation d'entropie **normalisée**, on peut douter de l'intérêt d'une telle étude. Mais le cas particulier des 1-grammes est un bon exemple : peut-être que de laisser passer tous les 1-grammes candidats pourrait être une stratégie intéressante, car l'autonomie à moins de sens pour les 1-grammes.

Une autre manière d'évaluer l'intérêt de l'autonomie serait tout simplement reprendre un système existant comme MAUI, et y intégrer l'autonomie comme une *feature* supplémentaire. On verrait alors une amélioration (ou non) des scores, si l'algorithme d'apprentissage détecte une corrélation entre autonomie et terme potentiellement clé.

4 Conclusion

J'ai donc pu développer un outil performant et utile, qui peut avoir un nombre important d'applications, et est capable de travailler sur des jeux de données très gros comme Wikipédia.

Pour l'aspect extraction de termes clés, mes résultats s'avèrent au final un peu décevants, car même si l'autonomie est discriminante, elle ne l'est pas plus que d'autres

critères triviaux qui marchent aussi bien, voir mieux. Il serait intéressant de retourner à la définition de l'autonomie et de tenter de définir une nouvelle mesure inspirée de l'autonomie, mais n'ayant pas ses défauts.

Plus généralement, mon stage aura surtout été l'occasion de découvrir énormément de choses dans le domaine du traitement du langage naturel (domaine qui, avec la sécurité informatique, m'attire beaucoup pour plus tard) : j'ai pu lire de nombreux articles scientifiques, comprendre beaucoup de concepts de base, etc.

Développer le système de calcul d'autonomie aura été une tâche passionnante, car il s'agit de trouver une structure de données et des algorithmes les plus efficaces possible, pour un traitement qui n'est pas très compliqué, mais qui peut s'effectuer à l'échelle des milliards de n-grammes.

Enfin, j'ai pu découvrir la problématique de l'extraction de mots-clés et toutes les méthodes associées. J'ai ensuite dû utiliser des mesures comme le f-score, faire des tests, mettre en évidence mes résultats via des visualisations, etc.

Mon stage aura donc été très enrichissant, et je tiens encore à remercier tous ceux qui l'ont rendu possible.

Pour finir, nous souhaiterions publier un article scientifique qui résumerait notre démarche et les résultats obtenus, qui restent intéressants et mériteraient d'être développés.

Annexes

4.1 Évaluations sur tous les corpus

filtre	précision	rappel	F-score	capture corpus	capture gold
aucun	3.77	5.81	4.57	100.0	100.0
autonomie corpus >1	4.3	6.6	5.21	59.91	83.6
autonomie wp >1	3.81	5.78	4.59	78.43	86.4
count wp >4	4.77	7.25	5.75	46.03	71.8
stopwords	4.31	6.67	5.24	41.5	99.21
stopwords et autonomie corpus >1	4.73	7.27	5.73	25.86	82.94
stopwords et autonomie wp >1	4.3	6.56	5.2	34.77	85.98
stopwords et count wp >4	5.46	8.28	6.58	15.63	71.21

Tab. 5 – NLM-500

filtre	précision	rappel	F-score	capture corpus	capture gold
aucun	6.14	14.55	8.63	100.0	100.0
autonomie corpus >1	3.83	9.0	5.37	63.93	60.59
autonomie wp >1	5.38	12.84	7.58	76.79	78.99
count wp >4	5.02	12.04	7.09	62.06	51.53
stopwords	8.18	19.63	11.55	42.04	99.02
stopwords et autonomie corpus >1	5.28	13.07	7.52	27.77	59.79
stopwords et autonomie wp >1	6.96	16.91	9.86	34.47	78.16
stopwords et count wp >4	7.94	19.71	11.32	23.33	51.0

Tab. 6 – inspec-uncontr

filtre	précision	rappel	F-score	capture corpus	capture gold
aucun	1.47	6.55	2.4	100.0	100.0
autonomie corpus >1	0.93	4.31	1.53	63.93	64.23
autonomie wp >1	1.32	6.05	2.17	76.79	79.41
count wp >4	1.57	6.96	2.56	62.06	83.89
stopwords	1.58	7.02	2.58	42.04	99.91
stopwords et autonomie corpus >1	0.98	4.69	1.62	27.77	64.14
stopwords et autonomie wp >1	1.46	6.79	2.4	34.47	79.36
stopwords et count wp >4	1.98	9.38	3.27	23.33	83.84

Tab. 7 – inspec-contr

Références

Beliga, Slobodan (2014). “Keyword extraction : a review of methods and approaches”. Mém.de mast. University of Rijeka, Department of Informatics.

Bougouin, Adrien (2013). “État de l’art des méthodes d’extraction automatique de termes-clés”. In : *Rencontre des Étudiants Chercheurs en Informatique pour le Traite-*

filtre	précision	rappel	F-score	capture corpus	capture gold
aucun	1.47	6.55	2.4	100.0	100.0
autonomie corpus >1	0.93	4.31	1.53	63.93	64.23
autonomie wp >1	1.32	6.05	2.17	76.79	79.41
count wp >4	1.57	6.96	2.56	62.06	83.89
stopwords	1.58	7.02	2.58	42.04	99.91
stopwords et autonomie corpus >1	0.98	4.69	1.62	27.77	64.14
stopwords et autonomie wp >1	1.46	6.79	2.4	34.47	79.36
stopwords et count wp >4	1.98	9.38	3.27	23.33	83.84

TAB. 8 – semeval2010-maui

filtre	précision	rappel	F-score	capture corpus	capture gold
aucun	15.8	33.82	21.54	100.0	100.0
autonomie corpus >1	13.25	28.78	18.15	67.86	64.19
stopwords	15.9	34.01	21.67	96.96	99.79
stopwords et autonomie corpus >1	13.35	28.97	18.28	65.72	64.08

TAB. 9 – wikinews (les autonomies wikipédia ne sont pas affichées, car ce corpus est en français, et Wikipédia en anglais)

ment Automatique des Langues (RECITAL). Sables d'Olonne, France. URL : https ://hal.archives-ouvertes.fr/hal-00821671.

HASAN, Kazi Saidul et Vincent NG (2014). "Automatic Keyphrase Extraction : A Survey of the State of the Art". In : *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1 : Long Papers)*, p. 1262–1273.

HEAFIELD, Kenneth (2011). "KenLM : Faster and smaller language model queries". In : *Proceedings of the Sixth Workshop on Statistical Machine Translation*. Association for Computational Linguistics, p. 187–197.

HULTH, Anette (2003). "Improved Automatic Keyword Extraction Given More Linguistic Knowledge". In : *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing*. EMNLP '03. Stroudsburg, PA, USA : Association for Computational Linguistics, p. 216–223. DOI : 10.3115/1119355.1119383. URL : http ://dx.doi.org/10.3115/1119355.1119383.

KIM, Su Nam, Timothy BALDWIN et Min-Yen KAN (2010). "Evaluating N-gram Based Evaluation Metrics for Automatic Keyphrase Extraction". In : *Proceedings of the 23rd International Conference on Computational Linguistics*. COLING '10. Beijing, China : Association for Computational Linguistics, p. 572–580. URL : http ://dl.acm.org/citation.cfm?id=1873781.1873846.

KIM, Su Nam, Olena MEDELYAN et al. (2010). "SemEval-2010 Task 5 : Automatic Keyphrase Extraction from Scientific Articles". In : *Proceedings of the 5th International Workshop on Semantic Evaluation*. SemEval '10. Los Angeles, California : Association for Computational Linguistics, p. 21–26. URL : http ://dl.acm.org/citation.cfm?id=1859664.1859668.

LIU, Zhiyuan et al. (2010). "Automatic Keyphrase Extraction via Topic Decomposition". In : *Proceedings of the 2010 Conference on Empirical Methods in Natural Language*

filtre	précision	rappel	F-score	capture corpus	capture gold
aucun	6.31	27.53	10.27	100.0	100.0
autonomie corpus >1	7.05	30.38	11.44	60.62	93.23
autonomie wp >1	6.75	29.32	10.97	76.99	95.88
count wp >4	7.08	30.89	11.52	48.15	96.72
stopwords	6.78	29.27	11.0	44.54	99.89
stopwords et autonomie corpus >1	7.35	31.55	11.92	28.54	93.13
stopwords et autonomie wp >1	7.1	30.76	11.54	36.72	95.77
stopwords et count wp >4	7.68	33.23	12.47	17.65	96.62

Tab. 10 – citeulike180-filtered

Processing. EMNLP '10. Cambridge, Massachusetts : Association for Computational Linguistics, p. 366–376. url : http://dl.acm.org/citation.cfm?id=1870658.1870694.

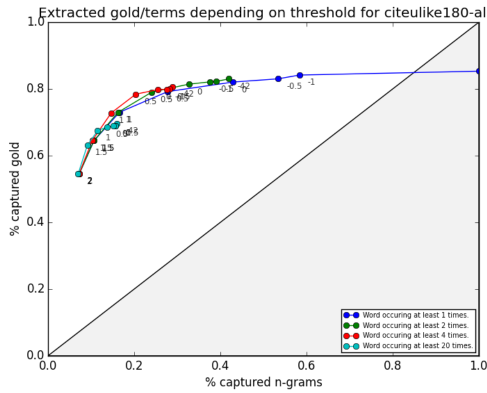
Magistry, Pierre (2013). "Segmentation en mots non-supervisée et estimation de la lexicalité : le cas du mandarin". Thèse de doct.

Magistry, Pierre et Benoît Sagot (2012). "Unsupervized Word Segmentation : the case for Mandarin Chinese". In : *ACL - Annual Meeting of the Association for Computational Linguistics - 2012*. ACL. Jeju, South Korea. url : https://hal.inria.fr/hal-00701200.

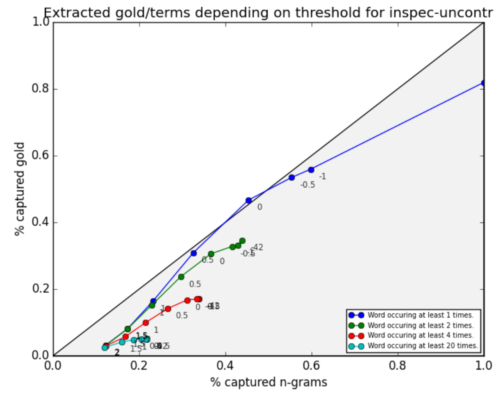
Medelyan, Olena, Eibe Frank et Ian H. Witten (2009). "Human-competitive Tagging Using Automatic Keyphrase Extraction". In : *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing : Volume 3 - Volume 3*. EMNLP '09. Singapore : Association for Computational Linguistics, p. 1318–1327. isbn : 978-1-932432-63-3. url : http://dl.acm.org/citation.cfm?id=1699648.1699678.

Mihalcea, Rada et Paul Tarau (2004). "TextRank : Bringing Order into Texts". In : *Proceedings of EMNLP 2004*. Sous la dir. de Dekang Lin et Dekai Wu. Barcelona, Spain : Association for Computational Linguistics, p. 404–411.

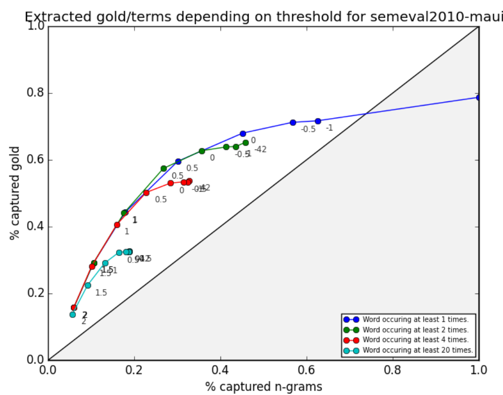
Pauls, Adam et Dan Klein (2011). "Faster and Smaller N-gram Language Models". In : *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics : Human Language Technologies - Volume 1*. HLT '11. Portland, Oregon : Association for Computational Linguistics, p. 258–267. isbn : 978-1-932432-87-9. url : http://dl.acm.org/citation.cfm?id=2002472.2002506.



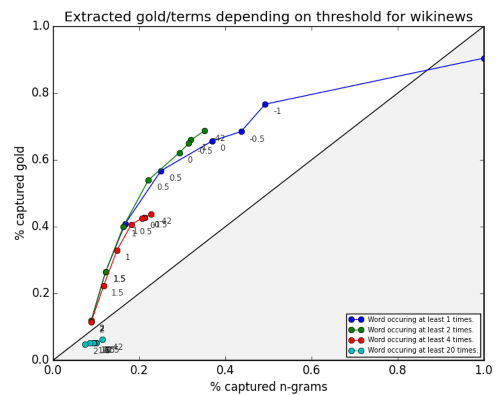
(a) citeulike180-all



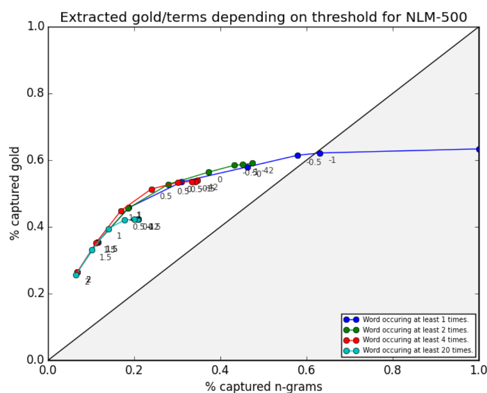
(b) inspec-uncontr



(c) semeval2010-maui



(d) wikinews



(e) NLM-500

FIG. 6 – Capture des n-grammes du texte par rapport à la capture des n-grammes du *gold*.

```
evolution
    autonomy : 3.38 / 6.00
    ev : -6.37 / -7.34
    count : 905.00 / 35338.00
    l_count : [905.0] / [35338]
    l_ev : ['-6.37'] / ['-7.34']
    l_autonomy : ['3.38'] / ['6.00']
bioinformatics
    autonomy : 4.29 / 3.32
    ev : -4.97 / -11.65
    count : 371.00 / 1147.00
    l_count : [371.0] / [1147]
    l_ev : ['-4.97'] / ['-11.65']
    l_autonomy : ['4.29'] / ['3.32']
worms
    autonomy : 1.77 / 4.06
    ev : -8.83 / -10.46
    count : 24.00 / 5915.00
    l_count : [24.0] / [5915]
    l_ev : ['-8.83'] / ['-10.46']
    l_autonomy : ['1.77'] / ['4.06']
neurodynamics
    autonomy : -10.00 / 0.33
    ev : -10.00 / -16.49
    count : 0.00 / 11.00
    l_count : [0.0] / [11]
    l_ev : ['nan'] / ['-16.49']
    l_autonomy : ['nan'] / ['0.33']
text mining
    autonomy : 2.05 / 1.06
    ev : 0.22 / -5.75
    count : 61.00 / 139.00
    l_count : [373.0, 108.0] / [91454, 44628]
    l_ev : ['-6.81', '-8.62'] / ['-7.34', '-8.56']
    l_autonomy : ['3.09', '1.90'] / ['6.01', '5.25']
neural coding
    autonomy : 0.30 / 1.44
    ev : -4.92 / -4.08
    count : 1.00 / 33.00
    l_count : [425.0, 693.0] / [7159, 6768]
    l_ev : ['-6.53', '-8.02'] / ['-9.81', '-10.03']
    l_autonomy : ['3.27', '2.30'] / ['4.44', '4.35']
```

```
bayesian statistics
    autonomy : 1.40 / 1.67
    ev : -1.78 / -3.27
    count : 23.00 / 129.00
    l_count : [304.0, 228.0] / [1726, 35939]
    l_ev : ['-7.16', '-6.44'] / ['-11.16', '-9.79']
    l_autonomy : ['2.86', '3.33'] / ['3.63', '4.49']
round robin
    autonomy : -10.00 / 2.02
    ev : -10.00 / -2.06
    count : 0.00 / 3734.00
    l_count : [21.0, 5.0] / [192583, 27324]
    l_ev : ['-9.16', '-10.27'] / ['-8.72', '-9.88']
    l_autonomy : ['1.56', '0.84'] / ['5.19', '4.45']
behavioral control
    autonomy : 0.35 / 0.55
    ev : -4.85 / -7.61
    count : 2.00 / 42.00
    l_count : [122.0, 598.0] / [8021, 256873]
    l_ev : ['-7.81', '-5.88'] / ['-9.36', '-5.38']
    l_autonomy : ['2.44', '3.70'] / ['4.74', '7.24']
literature mining
    autonomy : 0.80 / 0.08
    ev : -3.40 / -9.63
    count : 2.00 / 1.00
    l_count : [184.0, 108.0] / [88424, 44628]
    l_ev : ['-6.85', '-8.62'] / ['-8.52', '-8.56']
    l_autonomy : ['3.06', '1.90'] / ['5.25', '5.25']
small world
    autonomy : 1.71 / 0.66
    ev : -0.80 / -7.26
    count : 144.00 / 547.00
    l_count : [1013.0, 386.0] / [421496, 975283]
    l_ev : ['-6.23', '-7.49'] / ['-4.72', '-7.01']
    l_autonomy : ['3.46', '2.64'] / ['7.57', '6.12']
infant speech perception
    autonomy : -10.00 / 1.63
    ev : -10.00 / -2.84
    count : 0.00 / 3.00
    l_count : [35.0, 81.0, 61.0] / [13251, 61086, 14313]
    l_ev : ['-9.66', '-7.46', '-8.00'] / ['-8.27', '-6.76', '-7.84']
    l_autonomy : ['1.23', '2.67', '2.31'] / ['5.39', '6.35', '5.71']
```

Fig. 7 – Détails des *features* pour quelques n-grammes. Chaque ligne contient les valeurs pour un entrainement du système sur le corpus, puis sur Wikipédia. On affiche l'autonomie, la variation d'entropie, le nombre d'occurrences, puis ces mêmes valeurs, mais pour chacun des tokens des n-grammes.