Louis Lumière
école nationale supérieure

Un Moteur de Recherche Sémantique pour Sonothèques

Théo Serror
Section Son – Promotion 2018

Directeur interne : Sylvain Lambinet

Directeur externe : Aymeric Devoldère

Rapporteur : Franck Jouanny

Remerciements

Je tiens à remercier chaleureusement :

Sylvain LAMBINET de m'avoir accompagné avec attention, de la mise au point du projet jusqu'à la rédaction finale.

Aymeric DEVOLDÈRE, Valérie DELOOF, Raphaël SOHIER, Gurwal COÏC-GALLAS, Guillaume BOUCHATEAU et Marie DOYEUX d'avoir pris le temps de me faire partager leurs réflexions quant à leur pratique.

Cyril HOLTZ, Niels BARLETTA, Guillaume COUTURIER et Aymeric DEVOLDÈRE, pour *HAL*, d'avoir dès le début porté de l'intérêt à ce projet. Ces premières réflexions furent d'une grande aide pour démarrer et source d'une importante motivation.

Alex-Adrien AUGER, enfin, pour son expertise logicielle, son regard critique et sa participation décisive au développement de la partie pratique de ce mémoire.

Résumé

Le faible coût des supports de stockage numérique permets aux monteurs son d'avoir à disposition des sonothèques composées de plusieurs dizaines de milliers de sons. Se souvenir de tous ces sons est alors impossible. Et malgré l'avènement de moteurs de recherche puissants comme ceux de *Google* et de *Spotify*, les outils à disposition des monteurs sons sont encore limités à une simple recherche littérale de chaînes de caractères.

Dans ce travail, nous analysons les conséquences du fonctionnement de ces moteurs sur la pratique du montage son. À partir de ces observations, nous envisageons la représentation du contenu sémantique des noms de fichiers comme une solution possible.

Nous étudions pour cela deux approches. D'abord, une décomposition sémantique manuelle, avec la construction d'une structure hiérarchisée pour recueillir cette décomposition. Cette décomposition est appliquée au cas particulier des sons « d'ambiance d'intérieur ». Nous explorons ensuite une méthode automatique : l'*analyse sémantique latente*, appliquée à des sons issus de sonothèques commerciales. Dans les deux cas, nous étudions l'implication de telles représentations sur le processus de recherche.

Enfin, nous proposons, à travers le développement d'un gestionnaire de sonothèques rudimentaire, une implémentation concrète de ces deux méthodes.

Mots-clefs : moteur de recherche, sémantique, sonothèque, montage son, lexème, analyse sémantique latente, base de données, recherche d'information.

Abstract

With digital storage devices being cost effective, sound designers can nowadays gather sound libraries consisting of dozens of thousands samples. Therefore, it is impossible to remember all these samples. Despite the advent of powerful search engines, such as Google's and Spotifiy's engines, the tools available to sound designers are still limited to simple characters string matching.

This paper analyses the consequences of these search engine's operation towards sound designers' creative workflow. From these observations, we study how sounds' filenames' semantic content can be a potential solution.

Two separate solutions are then studied. The first consists in a manual decomposition of filenames' semantic content with the construction of a hierarchical structure to collect such a decomposition. This decomposition is carried out over the example of "indoor ambiences". An automatic solution is then explored: *latent semantic analysis*, carried out over samples from retail sound libraries. In both case, the consequences of such representations regarding the sound retrieval process are studied.

Finally, through the programming of a simplistic asset management software, implementations for both methods are presented.

Keywords: search engine, semantic, sound library, sound design, lexeme, latent semantic analysis, database, information retrieval.

Table des matières

Remerciements....3
Résumé....4
Abstract....5
Introduction....8
I Observation et analyse des mécanismes de recherche....10
1 Fonctionnement des outils existants....10
1.a Indexation....10
1.b Recherche littérale....12
2 Deux approches du nommage des sons....15
2.a Un nommage descriptif des sons : Aymeric Devoldère....16
2.b Un nommage synthétique des sons : Valérie Deloof....17
2.c Conséquences communes....19
2.c.α Une charge supplémentaire pour la mémoire du monteur....20
2.c.β De la difficulté d'échanger ses sons avec d'autres monteurs....21
3 Les pistes d'une solution....23
3.a Les solutions habituellement envisagées....23
3.a.α Formaliser l'information : les conventions de nommage....23
3.a.β Augmenter la puissance d'indexation : les métadonnées....25
3.b La représentation du *sens* comme une solution alternative....27
II Recherche et représentation du contenu sémantique des sons……29
1 La diversité morphologique....29
1.a La racinisation : un traitement des formes fléchies....30
1.a.α Présentation de l'algorithme de Porter....*30*
1.b Le traitement des abréviations et des formes fautives....32
2 Recherche et représentation du contenu sémantique....34
2.a Désignation des sources sonores et hyponymie....35
2.b L'espace des cas d'utilisations....38
2.c L'isotopie....41
2.c.α Représentation des lexèmes contextuels....42
2.c.β Décomposition des lexèmes principaux....45
3 Analyse sémantique latente....46
3.a Introduction à l'*ASL*....47
3.b Une mesure de l'occurrence des mots : la *TF-IDF*....49
3.c Description de l'analyse sémantique latente....50
3.c.α Construction de la matrice des occurrences....50
3.c.β Réduction du rang....52
3.c.γ Mesure des similitudes....53
4 Conclusions provisoires....54

III Implémentation du moteur de recherche..56

1 Schéma fonctionnel...57
2 Environnement de travail : *Electron*...59
2.a.α Gestion des bases de données dans l'application.......................................61
3 Traitement des variétés morphologiques..62
3.a Implémentation des cas particuliers...62
3.b Implémentation de la racinisation...64
4 Représentation manuelle du contenu sémantique..65
4.a Représentation et stockage des données...65
4.a.α Représentation des lexèmes contextuels...66
4.a.β Représentation des lexèmes quantificateurs..67
4.a.γ Représentation du contenu des lexèmes principaux....................................68
4.b Recherche...70
5 Implémentation de l'*ASL*...73
5.a Corpus choisi : les sonothèques commerciales..74
5.b Implémentation de l'*ASL*..75
5.b.α Formatage des données avec semanticpy*..76*
*5.b.β Exécution de l'*ASL *avec* semanticpy*...77*
5.b.γ Recherche de sons dans l'espace des concepts avec semanticpy*.............79*

Conclusion...81
Bibliographie...84
Annexes...87

A Note sur les expressions régulières...88
B Éléments de sémantique..90
C Abréviations et formes fautives...92
D Graphe des lexèmes contextuels..94
E Implémentation de l'algorithme de Porter (*Python*)...95
F Mots-vides pour l'anglais (*stopwords*)...100
G Implémentation *Python* de TF-IDF et ASL...101

Introduction

La révolution numérique a considérablement modifié le travail du monteur son en transformant ses outils. Ainsi, si écouter un son demandait de charger une bobine sur un banc de montage et si monter ce son nécessitait d'abord de le repiquer, ces deux opérations sont aujourd'hui réalisables en un clic.

L'époque où il était encore possible de répertorier dans un cahier le contenu de sa sonothèque et d'en mémoriser la majeure partie est révolue. L'accès quasi-instantané aux sons disponibles, conjugué à l'explosion des capacités de stockage a permis la constitution par les monteurs de sonothèques incroyablement fournies : on rencontre maintenant des sonothèques de plusieurs téraoctets, constituées de plusieurs centaines de milliers de sons. Cette croissance a permis la composition de bandes sons de plus en plus complexes, autant qu'elle s'en est alimentée.

Mais à l'heure des moteurs de recherche et de suggestion tels que *Google* et *Spotify*, les outils à la disposition des monteurs restent rudimentaires : il s'agit de recherche littérale sur un ensemble d'informations renseignées à la main. Autrement dit, ces logiciels ne profitent que de la puissance d'*itération* de l'outil numérique (parcourir et filtrer *très vite* une grande quantité de données), mais pas de sa puissance de *représentation* (c'est à dire établir de façon autonome des liens et des hiérarchies entre ces données).

Le second constat qui motive ce travail est le caractère « naïf » de ces moteurs de recherche qui pousse le monteur son à adopter des comportements contre-intuitifs pour « aider » le moteur de recherche à « comprendre » ce qu'il cherche. Une telle contorsion demande du temps et de l'énergie, pouvant mener à une frustration vis à vis de l'outil – qui manque alors son objectif : « faciliter la vie de l'utilisateur ».

En analysant cette *interaction* clef entre l'outil et l'utilisateur, nous ferons émerger la représentation du sens dans le nom des sons comme un moyen de créer des liens de haut niveau entre ces sons. Nous montrerons que ces liens permettent de rendre la recherche plus intuitive en

la rapprochant des structures de relation du langage. Nous proposerons alors deux méthodes de construction/représentation de ces liens, pour enfin en proposer une implémentation.

L'exploitation d'une sonothèque renvoie à une double problématique : celle de la désignation du sens et du traitement automatique du langage. Si longtemps ce type de traitement s'est appuyé sur des égalités strictes de caractères, l'introduction du langage naturel vise à rapprocher les méthodes de recherche du fonctionnement cognitif de la mémoire. Elle suppose d'être en mesure de prendre en compte à la fois la variabilité des descripteurs des sons et la variabilité sémantique de ces descripteurs ; et la problématique de la représentation du sens dans le contexte du langage cinématographique.

Ce travail se trouve donc à l'intersection de trois champs : la sémantique, l'informatique et le cinéma.

I Observation et analyse des mécanismes de recherche

En étudiant le fonctionnement des outils existants d'une part et le comportement des monteurs quant à la recherche des sons d'autre part, on constate que la rencontre entre les défauts de l'outil et les stratégies de contournement de ces défauts relèvent d'une interaction complexe. Analyser les conséquences de cette interaction permet d'expliquer les écueils des solutions proposées jusque-là, puis d'esquisser les pistes d'une solution alternative.

1 Fonctionnement des outils existants

Nous nous intéressons ici au fonctionnement élémentaire commun à tous les logiciels de gestion de sonothèque. On trouve un comparatif plus détaillé des logiciels du marché dans le mémoire de Simon Cacheux (2008)[1] [1] et plus récemment sur le site de Paul Virostek (2017)[2] [2].

Réduits à leur plus simple expression, ces outils visent à :

- indexer une partie du système de fichier de l'utilisateur dans une base de données,
- effectuer une recherche littérale sur les entrées de cette base de données.

1 Cacheux, Simon, *Méthode de recherche et de classification des sons en sonothèque*. Mémoire de master. Paris : École Nationale Supérieure Louis-Lumière, 2008, 87 p.

2 Virostek, Paul, *An Introduction to Sound FX Metadata Apps 2 - Comparing Apps*. https://creativefieldrecording.com/2014/06/17/an-introduction-to-sound-fx-metadata-apps-2-comparing-apps/, 2017 [consulté le 21 avril 2018]

1.a Indexation

L'indexation consiste à rassembler toutes les informations dont le logiciel a besoin dans une base de données (c'est-à-dire un tableau à double entrée). Ainsi, il n'est pas nécessaire de parcourir le disque de l'utilisateur et de lire les fichiers en question pour retrouver l'information. Chaque ligne du tableau correspond alors à un fichier (un son), chaque colonne contenant une information sur ce fichier.

Parmi les colonnes de ce tableau, on retrouve toujours :

- Le nom du fichier – c'est généralement ce qui permet à l'*utilisateur* de retrouver le fichier en question.
- Le chemin d'accès du fichier – c'est ce qui permet au logiciel de retrouver le fichier sur le disque pour éventuellement le lire.

On trouve ensuite un certain nombre de métadonnées, variant d'un logiciel à l'autre. Ces métadonnées servent à renseigner des informations supplémentaires sur le son en question comme :

- Une description plus précise du son.
- Une catégorie/sous-catégorie pour classer ce son.
- Un identifiant de séquence/plan/prise dans le cas des rushes issus d'un tournage.
- Le microphone utilisé pour la prise de son.
- *Etc.*.

Ces métadonnées peuvent appartenir à une norme ou à un format défini par l'éditeur du logiciel. Dans le cas des sons, on trouve par exemple les normes *ID3*, *Broadcast extension* et *iXML*.

Exemple :

Duration	Samp	Filename	Description	Category	Microphone	Location	Show
00:35.151	48000	MACHINE Bip alarme - Pousse-seringue 1 - OFF Porte ouverte.wav	Trois bips bre...	Machine	AB DPA 4006	Argenteuil	Hippocrate La Série
00:29.986	48000	MACHINE Bip alarme - Pousse-seringue 1 - Plan moyen.wav	Trois bips bre...	Machine	AB DPA 4006	Argenteuil	Hippocrate La Série
01:50.385	48000	MACHINE Bip alarme - Pousse-seringue 1 - Plan serré.wav	Trois bips bre...	Machine	AB DPA 4006	Argenteuil	Hippocrate La Série
01:34.771	48000	MACHINE Bip alarme - Pousse-seringue 2 - OFF Porte fermée.wav	Un bip long et...	Machine	AB DPA 4006	Argenteuil	Hippocrate La Série
01:09.028	48000	MACHINE Bip alarme - Pousse-seringue 2 - OFF Porte ouverte.wav	Un bip long et...	Machine	AB DPA 4006	Argenteuil	Hippocrate La Série
01:42.067	48000	MACHINE Bip alarme - Pousse-seringue 2 - Plan moyen.wav	Un bip long et...	Machine	AB DPA 4006	Argenteuil	Hippocrate La Série
01:39.383	48000	MACHINE Bip alarme - Pousse-seringue 2 - Plan serré.wav	Un bip long et...	Machine	AB DPA 4006	Argenteuil	Hippocrate La Série
00:13.626	48000	MACHINE Bip alarme - Pousse-seringue 3 - Plan serré.wav	Deux bips lon...	Machine	AB DPA 4006	Argenteuil	Hippocrate La Série
01:07.498	48000	MACHINE Bip alarme - Pousse-seringue 4 - OFF Porte fermée.wav	Trois bips nas...	Machine	AB DPA 4006	Argenteuil	Hippocrate La Série
01:03.242	48000	MACHINE Bip alarme - Pousse-seringue 4 - OFF Porte ouverte.wav	Trois bips nas...	Machine	AB DPA 4006	Argenteuil	Hippocrate La Série
01:13.359	48000	MACHINE Bip alarme - Pousse-seringue 4 - Plan moyen.wav	Trois bips nas...	Machine	AB DPA 4006	Argenteuil	Hippocrate La Série
00:54.655	48000	MACHINE Bip alarme - Pousse-seringue 4 - Plan serré.wav	Trois bips nas...	Machine	AB DPA 4006	Argenteuil	Hippocrate La Série
00:54.069	48000	MACHINE Bip alarme douce - ECG - OFF Porte fermée.wav	Trois bips lon...	Machine	AB DPA 4006	Argenteuil	Hippocrate La Série
01:31.837	48000	MACHINE Bip alarme douce - ECG - OFF Porte ouverte.wav	Trois bips lon...	Machine	AB DPA 4006	Argenteuil	Hippocrate La Série

Exemple de sonothèque indexée par Soundminer

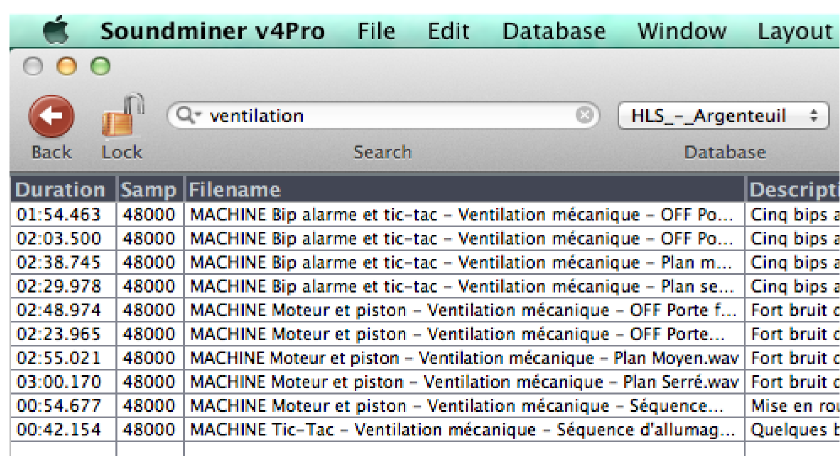
Sur cet extrait de sonothèque indexée par le logiciel *Soundminer*, on peut lire quelques unes des informations récupérées sur chaque fichier. Ici, on trouve de gauche à droite, la durée du son, la fréquence d'échantillonnage, le nom du fichier, la description, la catégorie, le dispositif de prise de son utilisé, le lieu de la prise de son, et le projet duquel est extrait le son.

Une fois l'indexation réalisée, l'utilisateur peut parcourir cet index pour trouver les sons qui lui conviennent.

1.b Recherche littérale

Concrètement, tous les outils du marché se présentent avec une « barre de recherche » dans laquelle l'utilisateur peut entrer des mots-clefs. Le moteur renvoie alors toutes les entrées de la sonothèque correspondant à ces mots-clefs.

Exemple :

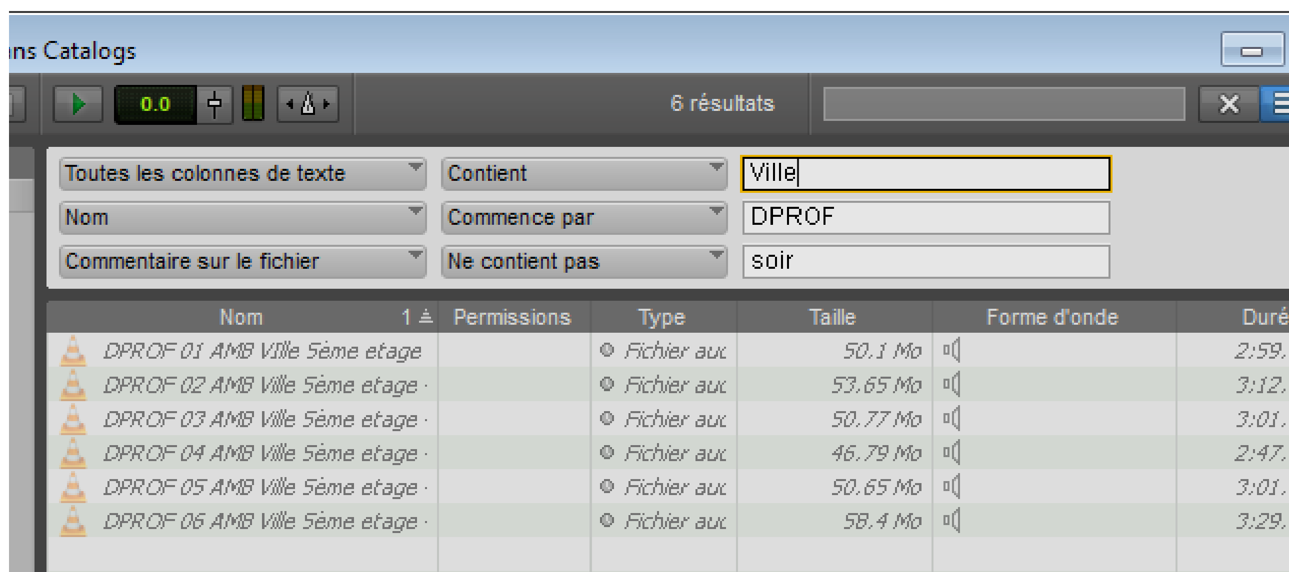


Duration	Samp	Filename	Descript
01:54.463	48000	MACHINE Bip alarme et tic-tac - Ventilation mécanique - OFF Po...	Cinq bips a
02:03.500	48000	MACHINE Bip alarme et tic-tac - Ventilation mécanique - OFF Po...	Cinq bips a
02:38.745	48000	MACHINE Bip alarme et tic-tac - Ventilation mécanique - Plan m...	Cinq bips a
02:29.978	48000	MACHINE Bip alarme et tic-tac - Ventilation mécanique - Plan se...	Cinq bips a
02:48.974	48000	MACHINE Moteur et piston - Ventilation mécanique - OFF Porte f...	Fort bruit c
02:23.965	48000	MACHINE Moteur et piston - Ventilation mécanique - OFF Porte...	Fort bruit c
02:55.021	48000	MACHINE Moteur et piston - Ventilation mécanique - Plan Moyen.wav	Fort bruit c
03:00.170	48000	MACHINE Moteur et piston - Ventilation mécanique - Plan Serré.wav	Fort bruit c
00:54.677	48000	MACHINE Moteur et piston - Ventilation mécanique - Séquence...	Mise en rou
00:42.154	48000	MACHINE Tic-Tac - Ventilation mécanique - Séquence d'allumag...	Quelques b

Résultats de recherches dans Soundminer

Dans cet exemple, seuls les sons dont le nom contient le mot « ventilation » sont renvoyés à l'utilisateur.

Certains logiciels permettent d'effectuer la recherche sur plusieurs colonnes simultanément, ou bien selon différentes combinaisons, comme dans l'exemple suivant, issu de l'*Espace de travail* de *Pro Tools*.



Résultats de recherche dans "l'espace de travail" de Pro Tools

Pour comprendre d'où émergent les limitations de ces outils, il convient de détailler le processus du point de vue du moteur de recherche.

Considérons que l'utilisateur entre la requête suivante : `passage auto nuit` dans la barre de recherche. Comment le moteur de recherche interprète-t-il chaque mot-clef et quel lien fait-il entre les différents mots-clefs d'une requête ?

Si la requête est interprétée comme *une seule* chaîne de caractères, alors le moteur ne distingue pas les mots-clefs et ne renverra pas, par exemple `passage auto` **`la`** `nuit`.

Ce comportement n'est évidemment pas satisfaisant. C'est pourquoi les moteurs de recherche « découpe » la requête (à chaque espace, par exemple). Ainsi `passage auto nuit` devient la liste `["passage","auto","nuit"]`. On peut alors comparer les mots-clefs individuellement.

Toutefois, la recherche d'un mot-clef peut s'effectuer d'au moins deux façons :

- Par égalité stricte : dans ce cas, `auto` ne renvoie pas `automobile`.
- Par inclusion : dans ce cas, on a bien `automobile`, mais aussi `automatique`.

La plupart des moteurs effectuent implicitement une recherche par inclusion (avec les avantages et les inconvénients que cela implique).

Enfin, les mots-clefs entre eux peuvent être liés par différents opérateurs logiques. La plupart des moteurs remplacent implicitement l'espace par l'opérateur « ET » (&). Ainsi, `["passage","auto","nuit"]` devient `"passage"&"auto"&"nuit"`. Certains moteurs permettent aussi d'utiliser d'autres opérateurs comme « OU » (|) et « SAUF » (-). On peut ainsi effectuer la requête suivante :

`"passage"&("auto"-"automatique"|"moto")&"nuit"`.

Les moteurs se chargent donc de traduire la requête de l'utilisateur en une représentation plus proche du langage machine (*Soundminer* par exemple, interprète la virgule « , » comme un « OU » et le signe moins « - » comme un « SAUF »). La représentation utilisée dans la majorité des langages de programmation suit le formalisme des expressions régulières (*cf.* annexe sur la question). Certains logiciels permettent d'ailleurs d'effectuer directement des recherches sous forme d'expression régulières, mais cela demande de l'entraînement.

In fine, l'utilisation naturelle d'un moteur de recherche suivrait le schéma suivant :

- L'utilisateur cherche une « pluie battante ».
- Il tape donc « pluie battante » dans la barre de recherche.
- Le moteur renvoie tous les sons correspondant à des pluies battantes.
- L'utilisateur choisit alors le(s) son(s) qui lui convien(nen)t parmi les sons proposés.

Cependant, il n'y a pas une façon unique de nommer les sons. La requête qui donnera le meilleur résultat dépend de la manière dont sont nommés les sons. Or la façon de nommer les sons est éminemment subjective et dépend évidement de la personnalité du monteur son.

Dès lors, on constate qu'il y a une interaction forte entre les défauts apparents de l'outil et le comportement du monteur : le monteur se heurtant à un « défaut » du moteur de recherche (défaut relatif à une utilisation particulière) modifie son comportement pour contourner l'obstacle. Mais cette contorsion a un coût.

En esquissant une typologie du nommage des sons, on peut en savoir plus sur cette interaction.

2 Deux approches du nommage des sons

L'objectif de ce mémoire étant de trouver une solution pour faciliter la recherche des sons pendant le montage, il était indispensable d'être témoin des obstacles rencontrés par des monteurs son professionnels. J'ai observé et échangé avec plusieurs monteurs son au travail, pour comprendre comment chacun utilise (le cas échéant) un moteur de recherche sur sa sonothèque.

Concrètement, cela a consisté à relever :

- Les mots-clefs entrés dans le moteur de recherche – c'est-à-dire la *requête*.
- Un extrait des résultats proposés par le moteur.
- Le(s) son(s) éventuellement sélectionné(s).

Ces observations ont été complétées par un échange avec les monteurs à propos des liens entre leur façon de monter, de nommer et de chercher les sons.

Deux grandes tendances ressortent de ces observations : une approche **descriptive** et une approche **synthétique**. Ces deux catégories n'épuisent certainement pas la question du nommage des sons, qui n'est d'ailleurs pas directement l'objet de ce travail. Plutôt que de catégories mutuellement exclusives, il s'agit des deux extrémités d'une *dimension*. Nous espérons montrer qu'il est pertinent de projeter la réflexion sur cette dimension en montrant qu'elle a des causes et des conséquences pratiques.

Pour nourrir d'exemples ce qui suit, j'ai choisi d'étudier le cas d'un « représentant » de chacune de ses catégories. Il s'agit de Valérie Deloof et Aymeric Devoldère, tous deux monteurs son membres de la société *HAL*. Je les ai choisi en exemple car leur façon de monter (et, inévitablement, leur filmographie) se distingue bien l'une de l'autre. De plus, au moment où je les ai observés travailler, Valérie venait d'acquérir la sonothèque de *HAL*, tandis qu'Aymeric venait d'acquérir celle de Valérie. Il était d'autant plus intéressant de pouvoir les observer chercher en « territoire inconnu ».

> **Précaution :** il ne s'agit pas là d'une étude comportementale exhaustive des monteurs son, mais d'acquérir une compréhension empirique des mécanismes à l'œuvre dans la recherche des sons et l'exploitation des sonothèques. On espère induire de ces observations des hypothèses saines pour la suite du travail.

2.a Un nommage descriptif des sons : Aymeric Devoldère

Les sons d'Aymeric sont généralement nommés de façon descriptive. Les noms sont assez longs et le vocabulaire utilisé est plutôt précis (à la manière de certaines sonothèques du commerce). Cette précision entraîne l'utilisation d'un vocabulaire varié et conduit à souvent employer plusieurs mots pour décrire la même *idée* de son. Quelles sont les conséquences possibles d'une telle pratique ?

- Effectuer de multiples requêtes pour parcourir toutes les formes d'une même idée. **Exemples :** la requête `oiseau`, ne renverra jamais le son `pouillot véloce – gazouille.wav`.
- Nommer les sons de façon redondante, pour augmenter leur « score » de recherche (*i.e.* leur chance d'apparaître comme résultat de recherche). **Exemple :** nommer un son `OISEAUX pouillot véloce – gazouille.wav` alors que l'idée `oiseau` est intégralement contenu dans `pouillot véloce`.[3]
- Trouver une « requête élémentaire » (le plus souvent, un radical) qui englobe toutes formes. Cela permet parfois aussi de s'affranchir des fautes d'orthographes (une des plus courantes : `oisos` pour `oiseaux`). Mais cela augmente le nombre de résultats « indésirables ». **Exemple :** `fri` pour chercher `frigo, frigidaire, réfrigérateur, fridge`. Problème : cette requête renvoie aussi `friture`.

D'autre part, cette approche descriptive du nommage des sons tend à faire figurer des « commentaires » dans le nom des fichiers.

3 On notera que quand il existe un champ de métadonnée « Catégorie », l'information « OISEAUX » y a sa place. Cependant, très peu de monteurs utilisent effectivement tous les champs de métadonnées disponibles. Et l'objet de ce travail est précisément de ne pas les contraindre à modifier leurs habitudes.

Exemple :

- `Vent dans les feuilles – quelques oiseaux en fond`. Il est très vraisemblable que ce son ne corresponde pas du tout à la requête `oiseaux`.
- `Enfants jouent dans une cour de récréation – bruit de circulation au loin`. *Idem*, ce son n'est sûrement pas un son de circulation. Ici, `circulation` désigne vraisemblablement une nuisance.

Peut-on esquisser un lien entre cette façon de nommer/chercher et une pratique particulière du montage son ? De l'aveu d'Aymeric : « cela tient aussi aux films que je fait. » En simplifiant, on peut qualifier sa façon de monter de « systématique » : tous les événements sonores sont recréés au montage son, avec parfois de nombreux éléments pour un même événement. Une des conséquences pratiques de cette approche concerne la version internationale du montage : Aymeric n'en monte pas. Ou plutôt : son montage son constitue en lui-même une version internationale (*i.e.* il ne manque rien d'autre que la voix quand on supprime le son direct). Cette façon de monter requiert souvent des sons très spécifiques. Il n'est de ce fait pas anodin qu'Aymeric possède et achète de nombreuses sonothèques du commerce – sonothèques qui viennent souvent remplir une niche particulière, et sont nommées de manière très descriptive.

2.b Un nommage synthétique des sons : Valérie Deloof

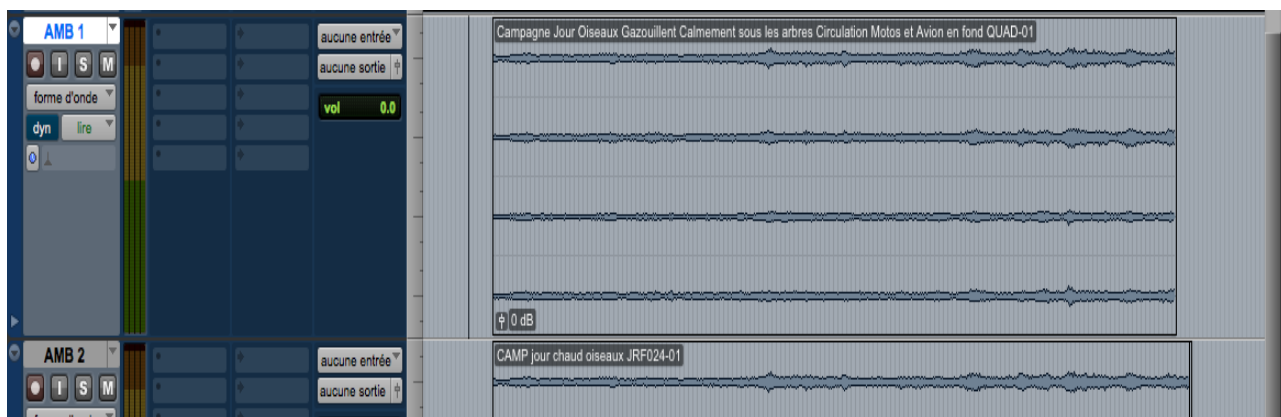
Contrairement à Aymeric, Valérie a tendance à nommer ses sons de façon beaucoup plus synthétique. Chaque nom commence par un mot, en majuscule, (éventuellement une abréviation) renvoyant à une catégorie, et se termine par un code, renvoyant au film dont il est issu.

Exemple : `CAMP jour chaud oiseaux JRF024`, où `CAMP` signifie `Campagne`.

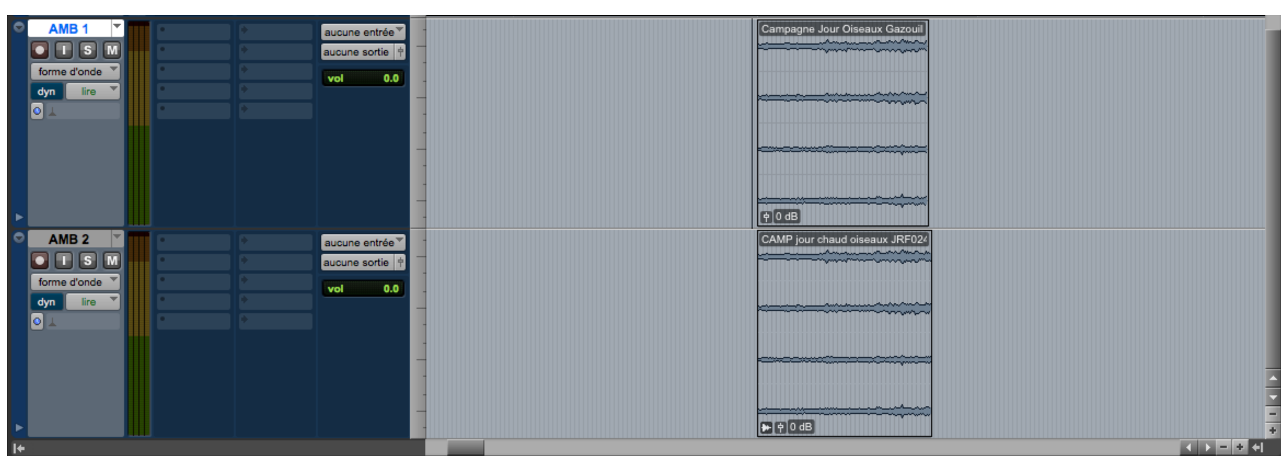
Valérie donne trois raisons à cela :

- Imposer à elle-même et à tout monteur qui reçoit ses sons d'en écouter le maximum pour trouver ce qu'il cherche – puisque le nom est loin d'épuiser le contenu du son. C'est effectivement ce que j'ai constaté en regardant Aymeric parcourir les sons de Valérie. Non seulement il ne connaissait généralement pas ces sons (reçus depuis peu), mais chaque requête renvoyait généralement plus de résultats quand effectuée sur la sonothèque de Valérie.

- C'est visuellement reposant, notamment pour le mixeur qui sait directement de quel son il s'agit. En effet, un nom court est moins susceptible d'être tronqué à l'affichage :



Affichage en gros plan : on peut lire correctement les deux noms



Affichage en plan large : le nom le plus long est tronqué

- Le temps disponible pour dérusher (et donc nommer) les sons étant court, les noms le sont d'autant plus.

Valérie raconte aussi qu'avant l'arrivée sur le marché des supports de stockage de grande capacité, elle notait tous ses sons dans un cahier, et les accompagnait d'une longue description. Elle passait donc beaucoup de temps à lire. Maintenant que des disques durs peuvent accueillir tous ses sons, **elle a converti le temps passé à lire en temps passé à écouter**.

Une autre caractéristique des sons de Valérie : le vocabulaire utilisé se rapporte toujours à la source (supposée) du son, jamais à une éventuelle connotation ou émotion. Valérie explique que l'émotion ressentie varie d'un monteur à l'autre, d'un moment à l'autre et du contexte d'utilisation

du son. Pour elle, renseigner une émotion préjuge trop de l'utilisation future du son et risque d'induire le monteur en erreur. On remarquera que cet état d'esprit est très cohérent avec sa volonté « d'écouter beaucoup de sons ».

Quelles sont les conséquences d'une telle façon de nommer les sons ?

Nommer les sons avec peu de mots et avec la volonté d'en écouter beaucoup conduit à utiliser un vocabulaire restreint. Cela permet, en théorie, de limiter le problème des synonymes – pourvu que l'on soit très rigoureux au moment de nommer les sons. En effet, avec peu de représentants par champ lexical, un son nommé avec un vocabulaire plus imagé ou « en bordure » du champ en question a toutes les chances d'être perdu. Cette situation est d'autant plus probable quand on reçoit la sonothèque d'un autre monteur – susceptible d'utiliser d'autres représentants d'un champ lexical donné.

Exemple : En cherchant des sons de la catégorie `Foule`, pour trouver une foule agressive, il était impossible de tomber sur le son dont le descripteur principal était `Meute` mais désignant des êtres humains (son nommé par un autre monteur). Dans un cas moins caricatural : toujours dans le champ lexical des `Foules`, les sons désignés par `Figurants` ont de grandes chances d'être oubliés.

Comme Aymeric, Valérie fait un lien entre son rapport à sa sonothèque et les films pour lesquels elle travaille. Ainsi, elle construit beaucoup plus la bande son autour du direct – qu'elle monte parfois elle-même. Ainsi, plutôt que de reconstruire systématiquement tous les événements sonores, l'accent est porté sur la cohérence avec le son direct. De fait, nombre de ses sons sont des bouts-à-bouts de prises du son direct du film. Plutôt que d'avoir un son pour remplir dimension spécifique d'un événement, l'objectif est d'utiliser ensemble des sons qui ont été enregistrés de la même façon, peut-être au même moment et par la même personne. Plus qu'une question technique, il s'agit là d'une question d'intention, sonore et surtout cinématographique.

L'étude de ces deux façons de nommer les sons permet de remarquer qu'elles ont, au moment de la recherche des sons, des conséquences similaires.

2.c Conséquences communes

Les approches **descriptives** et **synthétiques** du nommage sons, plutôt que d'être deux catégories mutuellement exclusives, sont deux extrémités d'une même dimension. Étudier les

conséquences communes de ces deux approches sur le montage son permet de le montrer. Ces conséquences sont, d'une part un effort de mémoire supplémentaire de la part du monteur, d'autre part une difficulté d'échanger ses sons avec d'autres monteurs.

2.c.α Une charge supplémentaire pour la mémoire du monteur

Au moment de la recherche des sons, on relève d'après les observations précédentes (2.a et 2.b) des cas fréquents où la mémoire vient soit *en renfort* du moteur de recherche, soit s'y substituer complètement.

En renfort, d'abord, pour tous les cas où le monteur doit se souvenir de toutes les *formes* du son qu'il cherche. Il peut s'agir des abréviations mais aussi des synonymes plus ou moins proches.

Exemples :

- Pour chercher un son de `porte`, il faut aussi chercher `pte` et `prte`.
- Peut-être que le chant d'`oiseau` que l'on cherche se cache dans un son nommé `hirondelle`.
- `Carriole`, `Charrette` et `Diligence` sont sûrement des sons proches, sans pour autant que ces objets soient vraiment identiques.

Il est ensuite des cas où le monteur abandonne complètement le moteur de recherche pour faire appel à la mémoire des sons qu'il possède, qu'il a déjà montés ou qu'il a enregistrés lui-même pour un film/un décor/une séquence similaire, *et cætera*.

La mémoire permet d'aller vite – on se souvient du nom du son ou du film dans lequel on l'a utilisé, donc on le trouve plus rapidement. De ce fait, les monteurs son rencontrés rangent tous, d'une façon ou d'une autre, leurs sons par les films dont ils sont issus.

Cependant, cette dépendance à la mémoire n'est pas sans risque. Certes, il est plus « facile » de monter un son dont on se souvient : on sait où le trouver, on connaît précisément son contenu et on a peut-être même déjà une idée des transformations qu'on va lui appliquer. Le risque est alors de n'utiliser qu'un sous-ensemble restreint des sons à disposition. D'autant que plus on utilise un son, plus on est susceptible de s'en souvenir… Ce risque est d'autant plus grand que le temps pour monter est court.

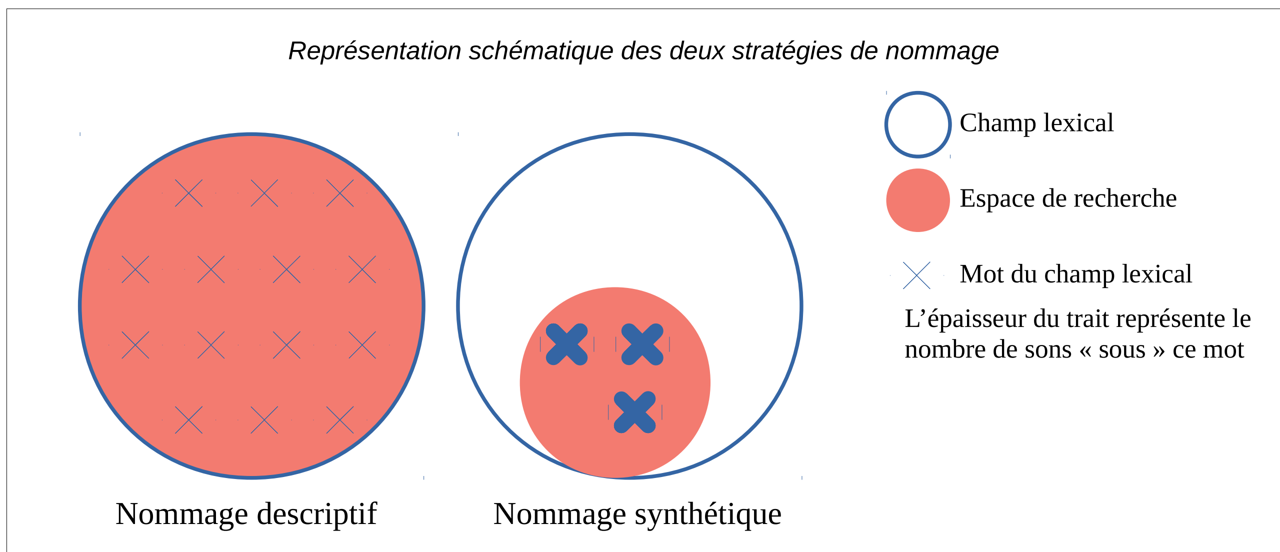
On envisage ce phénomène comme un « risque » en considérant que le montage son consiste à *choisir* (et pas à chercher) le « meilleur » son à chaque instant du film, parmi tous les sons *disponibles*. En effet, le choix de monter (ou de ne pas monter) tel ou tel son concrétise une partie du mouvement créatif du montage son en ce qu'il témoigne d'une interprétation de la narration, de la mise en scène, de la volonté du réalisateur, *et cætera*. Autrement dit, le choix des sons est une « particule élémentaire » du montage son comme processus créatif. Nous soutenons que la recherche n'en est pas une.

En effet, si l'on avait pleine mémoire et pleine conscience de tous les sons disponibles, nul besoin de chercher : il ne reste plus qu'a *décider*. Cela revient à dire que la recherche dans une base de données est une béquille de la mémoire qu'on a des éléments de cette base. Or on a montré (ou plutôt : observé) que, dans le cadre du montage son, la recherche s'appuie – en partie – sur la mémoire. On peut donc avancer que la recherche *devient* partie du processus créatif « faute de mieux », *in extremis*.

Si les deux approches du nommage décrites induisent un travail de mémorisation supplémentaire pour le monteur, elles rendent aussi délicat le partage des sons d'un monteur à l'autre.

2.c.β De la difficulté d'échanger ses sons avec d'autres monteurs

On a vu que, du point de vue de la recherche des sons, la principale distinction entre une approche descriptive et une approche synthétique concerne le nombre de représentants d'un champ lexical. On peut représenter cette distinction comme suit :



Représentation schématique des deux stratégies de nommage

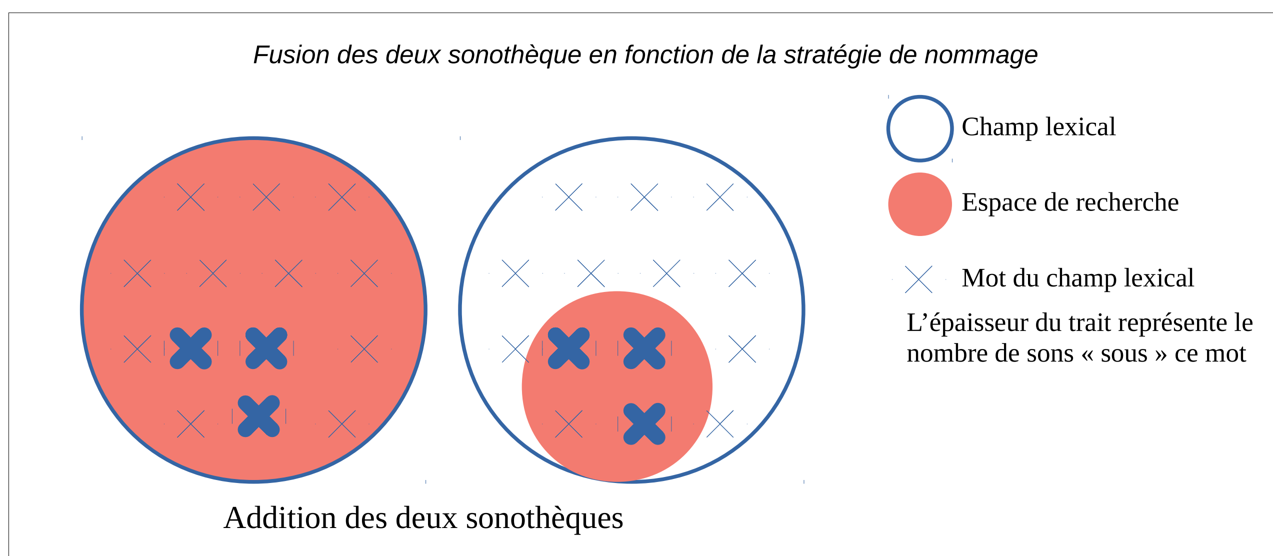
On voit donc que :

- Dans le premier cas, on peut rechercher des éléments particuliers, mais qu'une recherche sur tous le champ lexical est plus longue.
- Dans le second cas, on peut discriminer moins d'éléments particuliers, mais on récupère plus vite tous les éléments du champ.

On passe donc continument d'un cas à l'autre en augmentant le nombre de représentants du champ lexical – agrandissant dans le même temps la surface des requêtes à couvrir.

Que se passe-t-il si un monteur « descriptif » reçoit une sonothèque « synthétique » ? Certains mots de champ vont maintenant renvoyer beaucoup plus de résultats (qui pourraient être « ventilés » sur d'autres représentants du champ lexical). Ainsi le monteur va probablement devoir écouter plus de sons – et c'est peut-être l'objectif.

Que se passe-t-il si un monteur « synthétique » reçoit une sonothèque « descriptive » ? Soit il doit apprendre les nouveaux mots utilisés pour ce champ lexical, soit une partie des sons est perdue.



Ayant évalué les liens entre le fonctionnement des moteurs de recherche et le comportement du monteur son, nous pouvons maintenant étudier les voies d'amélioration de cet outil.

3 Les pistes d'une solution

Nous avons montré que les limites propres des moteurs de recherche pour les sonothèques poussent les monteurs à adopter des comportements visant à contourner ces limites. Ce faisant, de nouveaux problèmes adviennent, qui peuvent créer une frustration pour l'utilisateur. En effet, le monteur son peut avoir l'impression de passer plus de temps à composer une requête qu'à effectivement écouter les sons. Ou bien il effectue des requêtes très générales qui lui renvoient plusieurs centaines de sons, le moteur fournissant alors une valeur ajoutée bien faible.

Toutefois, plusieurs solutions ont été envisagées pour rendre les moteurs de recherche pour les sonothèques plus performants tel que le formalisme et les métadonnées. Malheureusement ces solutions ne font que déplacer le problème, nous verrons comment.

Avec l'ambition de ne pas demander plus d'efforts à l'utilisateur, nous envisagerons le problème sous l'angle de la question du *sens*, pour proposer une solution alternative.

3.a Les solutions habituellement envisagées

3.a.α Formaliser l'information : les conventions de nommage

Il s'agit d'établir un certain nombre de règles pour nommer les sons, en utilisant toujours les mêmes mots pour désigner les même choses. Le plus souvent, on choisi un certain nombre de paramètres utiles pour décrire le son, puis on choisi quelles sont les valeurs de chacun de ces paramètres.

Exemples :

Pour nommer les sons de véhicules, on pourrait décider de faire figurer dans le nom :

- La catégorie : véhicule.
- La sous-catégorie : voiture, camion, moto, vélo, *etc.*
- Le modèle/La marque : *Peugeot*, *Ferrari*, moteur V8, *etc.*
- Le mouvement : passage gauche/droite, démarrage, arrêt, accident, *etc.*
- Le sol : gravier, bitume, terre, *etc.*
- La valeur de plan : gros plan, plan moyen, plan large, plan lointain, *etc.*

Cependant, cette solution a trois défauts majeurs.

Tout d'abord, certains défauts évoqués plus haut persistent. La question du champ lexical persiste : avec quelle précision doit-on renseigner les paramètres ? Doit-on séparer `moto/mobylette/scooter` ou bien les rassembler sous `deux-roues` (voir même, faire figurer les deux informations) ? Ensuite, l'évaluation de certain paramètres est subjective ou relative au contexte : un plan moyen pour une séquence « sonnera » peut-être comme un plan large pour la séquence suivante (et cette appréciation dépend évidement du monteur) : l'échange des sons n'est donc toujours pas résolu.

Ensuite, cette solution revient à un classement arborescent des sons et en a les défauts sans vraiment en avoir les avantages. D'abord, un son est condamné à n'appartenir qu'à une feuille de l'arbre : si c'est un `Véhicule`, il ne peut plus appartenir à la catégorie `Sport` (*quid* alors des `voitures de course` ?). Ensuite, si cette solution est appropriée pour un ensemble *fini* d'éléments comme, par exemple, les sonothèques commerciales dont le contenu n'évolue plus

une fois achetées, elle ne l'est pas pour une sonothèque en perpétuelle expansion comme celle d'un monteur son. En effet, si la catégorie `Véhicule` était pertinente au moment du premier classement, l'acquisition d'un très grand nombre de sons de cette catégorie peut la rendre trop « large ». Il faut alors repenser le classement, ce qui est illusoire quand il faut renommer plusieurs centaines de milliers de sons.

Enfin, la mise en pratique de cette solution la disqualifie presque immédiatement. En effet, adopter une telle convention demande une grande rigueur et une grande cohérence dans le nommage des sons. Cela demande du temps et de l'énergie (toujours vérifier que l'on utilise bien le bon vocabulaire pour les bons sons) pour, *in fine*, rajouter beaucoup d'informations redondantes (il semble évident qu'une `moto` est un `véhicule`). S'ajoute à cela qu'il n'existe pas de convention largement adoptée par les monteurs sons. En proposer une sans avoir les moyens de la faire adopter est donc une entreprise vaine.

On peut d'ailleurs remarquer que, si cette solution n'est pas répandue, c'est peut-être parce qu'elle « assèche » le nom du son de son contenu émotionnel. En effet, le nom `Voiture de charlotte` est d'office exclu d'une telle convention, alors qu'il renvoie peut-être le monteur au souvenir de l'enregistrement de ce son ou au film dont il est issu.

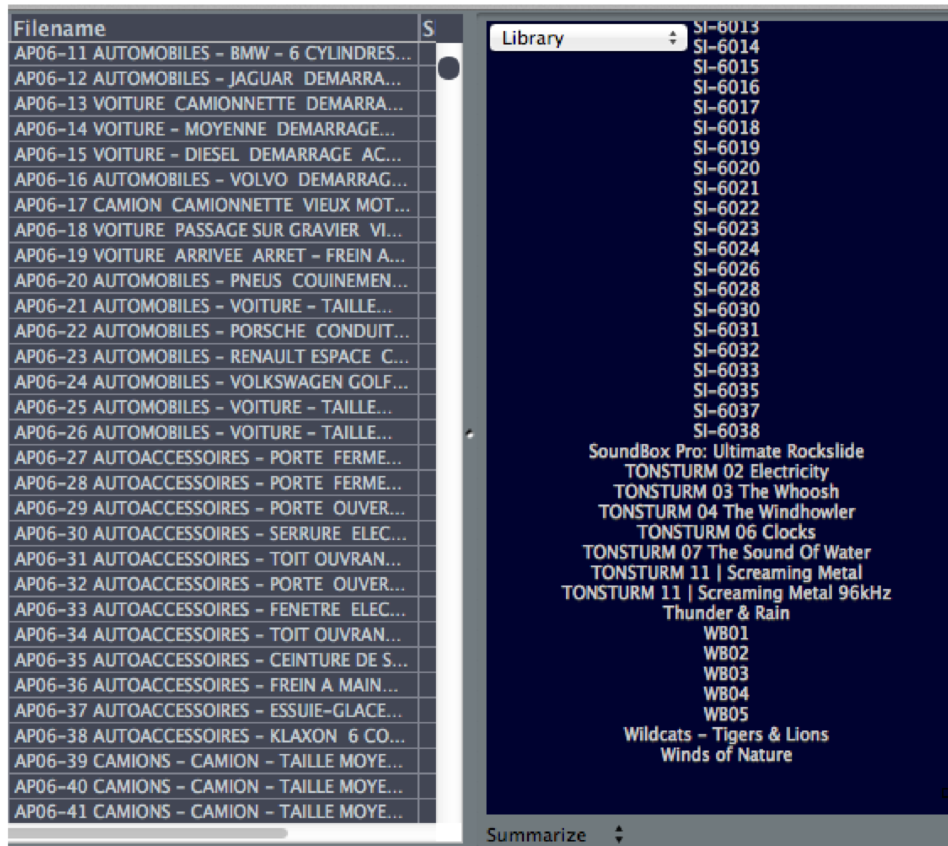
Finalement, cela revient à contraindre le monteur à adopter un comportement particulier, ce que nous cherchons précisément à éviter. Étudions donc la piste des métadonnées.

3.a.β Augmenter la puissance d'indexation : les métadonnées

Certains logiciels de gestion de sonothèque permettent de renseigner de nombreuses métadonnées supplémentaires pour compléter celle des normes existantes comme le *iXML*. C'est le cas de *Soundminer,* par exemple, qui propose de renseigner les champs supplémentaires : *Category, ChannelLayout, Composer, Conductor, Description, Designer, EntryDate, FeaturedInstrument, FXName, Index, Key, Keywords, Library, Location, Lyrics, Manufacturer, Microphone, Mood, Notes, Performer, Popularity, Publisher, Rating, RecMedium, RecType, Show, Source, SubCategory, Usage et Version*.

L'objectif est d'augmenter la quantité d'information ingérées par le moteur de recherche pour améliorer l'indexation et donc faciliter la recherche. Ainsi l'utilisateur a accès en un clic à tous les sons issus d'un film donné ou encore peut voir toutes les sous-catégories existantes dans un ensemble de sons.

Exemple : (Issu de *Soundminer*)



Le panneau "Summarize" de Soundminer donne un accès rapide à des sous ensembles de la base de données

En cliquant sur un titre du panneau de droite, on accède directement à tous les sons issus de la sous-sonothèque en question.

Il convient toutefois de remarquer que cette solution n'est qu'une déclinaison de la précédente à laquelle elle ajoute une facilité d'usage. En effet, il s'agit essentiellement de distribuer l'information sur différents champs de métadonnées plutôt que dans le nom du fichier. On se heurte alors aux même écueils : la cohérence du classement et l'effort demandé au monteur pour renseigner tous ces champs.

Cette information est en outre largement propriétaire et donc non transportable d'un logiciel à l'autre. Ainsi, les champs renseignés dans *Soundminer* ne seront lus ni dans l'*Espace de travail* de *Pro Tools*, ni dans *BaseHead*. Cela limite encore l'échange des sons entre monteurs qui utiliseraient des logiciels différents.

Ayant étudié les raisons pour lesquelles les solutions existantes échouent à résoudre profondément le problème, nous allons donc tenter de proposer une solution dont le point de départ est la représentation du sens contenu dans le nom des sons tels qu'ils sont nommés spontanément par les monteurs.

3.b La représentation du *sens* comme une solution alternative

En reprenant les observations faites en 1 et 2, on peut constater que les obstacles rencontrés lors de la recherche des sons sont de deux sortes : morphologiques d'une part, sémantiques d'autre part.

Les obstacles **morphologiques** regroupent : les synonymes simples (« frigidaire » ; « réfrigérateur » ; *etc.*) ; les différentes traduction d'un mot (essentiellement entre anglais et français) ; les abréviations (« pte » ; « fd air » ; « camp » ; *etc.*); les déclinaisons (« léger » ; « légère » ; « légèrement ») et toutes les formes fautives (« oisos » ; « piafs » ; *etc.*). Dans ces cas, si *le sens* des mots ne fait aucun doute, la multiplicité des formes (*i.e.* la « diversité morphologique ») oblige le monteur à s'en souvenir et à chercher parmi toutes ces formes possibles.

Les obstacles **sémantiques** concernent essentiellement les habitudes de nommage.

Exemples :

- Utilisation d'un vocabulaire **imagé,** souvent polysémique, contre un vocabulaire plus **analytique** et donc plus monosémique.
- Renseignement de la **connotation** du son contre le renseignement de la **source** uniquement.

Ainsi, le monteur son qui reçoit la sonothèque d'un collègue doit « apprendre », « se plier à » la façon de penser/de nommer et, ultimement, à la façon de *monter* de son collègue.

Observant cela, on peut envisager une solution qui ne contraigne pas le comportement du monteur. En effet, plutôt que de *modifier* l'information disponible (par des conventions ou des métadonnées), nous faisons l'hypothèse que les noms des sons contiennent *en puissance* toute l'information nécessaire. Il s'agit donc de faire en sorte que le moteur « comprenne » mieux l'information déjà présente.

En effet, utiliser un moteur de recherche *littérale* revient à considérer que l'information contenue dans le nom d'un son est épuisée par la somme des mots qui le constituent. Ce faisant, on omet l'interaction entre les mots, le contexte, qui participe de l'*effet de sens*.

La partie suivante est donc consacrée à l'analyse et à la représentation de ce contenu sémantique.

II Recherche et représentation du contenu sémantique des sons

En observant le fonctionnement des moteurs de recherche du marché et en le mettant en relation avec le comportement des monteurs son, nous avons relevé deux types d'obstacles à la recherche des sons : la diversité morphologique (abréviations, déclinaisons, *etc.*) d'abord ; puis la représentation du contenu sémantique. Autrement dit : comment savoir qu'une église est, le plus souvent, un lieu « grand, calme et réverbérant »?

Dans cette partie, nous allons donc d'abord tenter de résoudre la question de la diversité morphologique, car c'est un prérequis à l'étude de la suite. Ainsi nous pourrons proposer, une représentation conceptuelle des liens sémantiques entre les sons et une méthode manuelle de construction de ces liens. Cependant, mesurant les limites de cette construction manuelle, nous étudierons la faisabilité d'une solution automatique.

1 La diversité morphologique

La première partie nous a permis d'esquisser une typologie des variétés morphologiques existantes pour un *lexème* donné. On distingue donc :

- Les formes fléchies (ou « déclinaisons »). **Exemple :** `léger, légère, légèrement` représentent tous le même *lexème* (représenté, par exemple, par `léger`).
- Les abréviations. **Exemple :** `porte/pte, quelques/qques`, *etc.*

- Les formes fautives. **Exemple :** `oiseaux/oisos/piafs`.

S'il existe une approche algorithmique à la question des formes fléchies, on verra que les abréviations et les formes fautives doivent faire l'objet d'une documentation minutieuse.

1.a La racinisation : un traitement des formes fléchies

La racinisation, ou désuffixation (*stemming*, en anglais) désigne le procédé consistant à transformer des formes fléchies en leur racine. Ceci fait, on peut alors identifier chaque *lexème* à un représentant unique : sa racine.

Exemples :

- `Agitation -> Agit / Agité -> Agit`
- `Légère -> Léger / Légèrement -> Léger`

Il existe plusieurs algorithmes de racinisation, dont ceux de Porter[4][3] et de Paice/Husk[5][4]. Nous choisissons de nous intéresser à l'algorithme de Porter, car il en existe plusieurs transpositions en langue française, notamment l'algorithme de Carry[6][5] et *via* le langage *Snow*[7][6].

1.a.α Présentation de l'algorithme de Porter

L'algorithme de Porter permet de supprimer automatiquement les suffixes des mots, par application successive d'un ensemble de règles.

On note v une voyelle (`Y` est une voyelle, s'il est précédé d'une consonne) et c une consonne (`Y` est une consonne, s'il est précédé d'une voyelle), ainsi que $V=v^{+}$ (respectivement $C=c^{+}$) une séquence *d'au moins une* voyelle (respectivement *une* consonne).

Avec ces notations, n'importe quel mot peut être représenté sous l'une des quatre formes suivantes :

4 Rijsbergen, Cornelis J. et al., *New models in probabilistic information retrieval*. London : British Library, 1980, pp. 98-106

5 Paice, Chris, Method for evaluation of stemming algorithms based on error counting. *Journal of the American Society for Information Science*. Vol. 47, 1996, pp. 632–349

6 Paternostre, M. et al., *Carry, un algorithme de désuffixation pour le français*. Rapport technique du projet Galilei, Institut Paul Otlet, 2002, 15 p.

7 Porter, Martin, *French stemming algorithm*. http://snowball.tartarus.org/algorithms/french/stemmer.html, 2006 [consulté le 21 avril 2018]

$$\begin{cases} CVCV\ldots C \\ VCVC\ldots C \\ CVCV\ldots V \\ VCVC\ldots V \end{cases}$$

Ou, avec le formalisme des expressions régulières (*cf.* annexe A : Note sur les expressions régulières) : $(C)?(VC)^m(V)?$ On dit alors que la racine « est d'ordre m ».

Exemples :

$$\begin{array}{l} m=0\ :\ I,\ BY,\ TREE,\ \ldots \\ m=1\ :\ TROUBLE,\ TREES,\ CEASE,\ \ldots \\ m=2\ :\ TROUBLES,\ PRIVATE,\ \ldots \end{array}$$

L'algorithme de Porter était d'abord destiné à la langue anglaise, on conserve les exemples dans cette langue pour l'instant.

On définit ensuite les règles selon lesquelles on va supprimer les suffixes, *et les conditions d'application* de ces règles – avec le formalisme suivant : $(condition)\ suffixe_1 \rightarrow suffixe_2$. Chaque suffixe est remplacé par un suffixe plus court, sous certaines conditions.

Exemples :

$(m>1)\ EMENT \rightarrow \emptyset$: Si la racine précédant « EMENT » est d'ordre supérieur à 1, alors on supprime « EMENT ». On évite ainsi la réduction $D\acute{E}MENT \rightarrow D$.

$(m>1\ et\ ({}^*S\,ou\,{}^*T))\ ION \rightarrow \emptyset$: Si la racine (d'ordre > 1) précédant « ION » se termine par « S » ou « T », alors on supprime « ION ». Ainsi : $ADOPTION \rightarrow ADOPT$.

Si une chaine de caractères remplie simultanément plusieurs conditions, alors on effectue celle correspondant au plus long suffixe.

Exemple : Avec les conditions suivantes,

$$\begin{cases} SSES & \rightarrow & SS \\ SS & \rightarrow & SS \\ S & \rightarrow & \emptyset \end{cases} \Rightarrow CARRESSES \rightarrow CARRESS.$$

L'ensemble des règles (environ une cinquantaine dans l'algorithme de Porter[8][3]) est classé en sept étapes, définissant ainsi l'ordre dans lequel les règles peuvent s'appliquer. Ceci permet de réduire « petit à petit » les suffixes les plus complexes.

Exemple :

$$GENERALIZATIONS \xrightarrow[]{{}^{*}S \rightarrow \emptyset} GENERALIZATION \xrightarrow[(m>0)]{IZATION \rightarrow IZE} GENERALIZE \xrightarrow[(m>0)]{ALIZE \rightarrow AL} GENERAL \xrightarrow[(m>1)]{AL \rightarrow \emptyset} \mathbf{GENER}$$

Il existe un ensemble de règles suivants le même principe pour la langue française : l'algorithme *Carry*. Malheureusement, l'adresse documentant ces règles n'est plus maintenue. Toutefois, comme mentionné plus haut, un *stemmer* pour la langue française a été ajouté au projet *Snowball* (un langage de programmation mis au point par Porter et dédié à la racinisation). On trouve des implémentations de ce *stemmer* dans différents langages de programmation, notamment en *C*[9][6] et en *Python*[10][7].

Par souci d'autonomie de ce mémoire, on reproduit, en annexe, l'intégralité des règles utilisées dans *PyStem*[11]. De plus, les détails d'une implémentation sont présentés dans la troisième partie.

1.b Le traitement des abréviations et des formes fautives

Observons, dans le tableau suivant, quelques exemples de sons issus de la sonothèque de Raphael Sohier, monteur son :

RecID	Duration	Filename
1	01:26.340	eloignement pas off + pte lourde.L.wav
2	00:07.845	montee escalier pas .L.wav
3	00:21.418	pas calme int apart deplacement .L.wav
10	00:17.024	pas int apart rapide saccades.L.wav
11	00:41.386	passage pas int apart .L.wav
14	01:20.532	presence humaine cage escalier apart .L.wav

8 Rijsbergen, Cornelis J. *et al.*, *op. cit.*
9 Porter, Martin, *op. cit.*
10 Boulton, R., *PyStemmer 1.3.0*. https://pypi.org/project/PyStemmer/1.3.0/, 2013 [consulté le 21 avril 2018]
11 *Ibid.*

21	01:27.125	ecoulement eau discret+ fond robinet .L.wav
30	01:26.016	amb cave + activitee diff verrou bois planche.L.wav
31	00:39.957	fd air cave .L.wav
34	00:55.381	vx cave hommes + buzz sub .L.wav
35	00:25.952	voix homme comissariat off .L.wav
36	02:23.872	voix hommes calme discussion .L.wav
66	00:28.928	activitee bar rangement .L.wav
71	02:21.994	Club Muscu R Brouhaha Discution Rires Hommes.L.wav
171	02:54.005	Crachotement Optique 16+Schratc.wav
172	01:14.319	Fond Air Labo ventil+Rev.L.wav
181	04:03.280	Pluie int rafales contre vitre-EFRT_01.L.wav
182	01:01.840	Pluie int oisos css 2-EFRT_01.L.wav

On voit se dessiner, à travers ce tableau, des tendances dans la formation des abréviations :

1. Conservation de la première et de la dernière lettre. **Exemple :** `Porte/Pte`.
2. Suppression des voyelles. **Exemple :** `Voix/Vx`.
3. Racinisation « agressive » du mot. **Exemple :** `Ambiance/Amb`.
4. Compression de certaines terminaisons. **Exemple :** `Oiseaux/Oisos`.

On voit toutefois que ces règles peuvent être incompatibles entre elles : selon (1.), on a `Porte/Pte`, mais selon (2.), on a `Porte/Prt`. De même, (1.) et (3.) sont, par construction, mutuellement exclusives. Si ces formes abrégées correspondent à des processus linguistiques connus (le faible poids informationnel des voyelles par exemple)[12][8], leur variabilité rend le traitement automatique délicat.

On se propose donc de documenter « à la main » les abréviations les plus courantes – et qui auraient échappé à la racinisation présentée en 1.a. Porter lui-même souligne que pour les cas

12 Opillard, Thierry, La dégradation Cambridge : même pas vrai ?. *Les Actes de Lecture*. Vol. n. 102, 2008, pp. 23-28

rares[13], ajouter des règles complexes peut alourdir le programme (en temps et en mémoire) par rapport à la documentation (exhaustive, autant que faire se peut) des cas en question.

On peut néanmoins envisager d'implémenter quelques règles simples comme la suppression des voyelles ou la transformation `*eau -> *o`.

Le dernier point à considérer s'agissant des abréviations et des formes fautives, c'est qu'il ne s'agit pas de simples équivalences entre des chaînes de caractères.

Exemples :

- `pte` ~ `porte` ⇒ `com`**`pte`** ~ `com`**`porte`**.
- `sil` ~ **`sil`**`ence` mais on a aussi `sil` ~ `fu`**`sil`**.

On utilise la relation $S_1 \sim S_2$ pour signifier que la recherche du mot-clef S_1 provoquera la recherche simultanée du mot-clef S_2 . On pourra profiter de la transitivité de cette relation - $(S_1 \sim S_2 \text{ et } S_2 \sim S_3) \Rightarrow S_1 \sim S_3$ - pour ne renseigner les équivalences que deux à deux.

Dans les exemples précédents, en considérant qu'une abréviation conserve la première lettre du mot, on effectue les transformations :

- `sil -> (^|\W)sil\w*` (*i.e.* `sil` doit être placé en *début* de mot).
- `Porte -> ((^|\W)porte($|[^\w-])|(^|\Wpte($\W))` (ainsi, on a `porte` et `pte`, mais pas `porteur` et `porte-avion`).

Maintenant que l'on a réduit la diversité morphologique (formes fléchies, abréviations et formes fautives), on peut prétendre à la constitution, pour chaque son, d'un nom de fichier « épuré ».

Exemple :

Nom d'origine	Nom « épuré »
montee escalier pas .L.wav	Monte escalier pas
pas calme int apart deplacement .L.wav	Pas calme intérieur appartement déplace
pas nerveux int cave .L.wav	Pas nerveux intérieur cave
pas escalier cave homm seul .L.wav	Pas escalier cave homme seul
passage pas int apart .L.wav	Passe pas intérieur appartement
pte vitree bois ouv ferm .L.wav	Porte vitre bois ouvre ferme

13 Rijsbergen, Cornelis J. *et al.*, *op. cit.*

Ainsi, on peut maintenant considérer que parmi tous les noms de fichiers de la sonothèque, chaque lexème dispose d'un représentant unique. Ce qui permet de se poser sereinement la question des liens entre les différents lexèmes.

2 Recherche et représentation du contenu sémantique

On a vu dans la première partie que, si les moteurs de recherches « standards » (c'est-à-dire littéraux) permettent de retrouver des *mots-clefs* (tous les sons contenant le mot `vent` par exemple), les relations de haut niveau sont à la charge du monteur. Par relation de haut niveau, on désigne les relations entre les *concepts* qui dépassent leur simple représentation lexicale. Par exemple, la séquence de lettres qui compose un mot est une information de bas niveau et l'apposition de deux mots constitue une relation de bas niveau, tandis que l'*effet de sens* résultant de cette apposition est une relation de plus haut niveau. Si la langue est un *support* de ces relations en ce qu'elle permet de les exprimer, les relations elles-mêmes sont tirées du réel.

Exemple : `Pouillot véloce - gazouille` n'indique pas *a priori* la relation hyponymique « le pouillot véloce est un oiseau ». Toutefois, si on disposait de l'information « seuls les oiseaux gazouillent », on pourrait peut-être deviner cette relation. Et il convient de noter que ce n'est pas la langue *en elle-même* qui renseigne cette information, mais la structure du *réel* (et les relations que nous y observons/construisons).

Pour obtenir une représentation de ces relations de haut niveau, il convient d'analyser une tension rencontrée dans la première partie : la tension entre le renseignement *de la source* et le renseignement *des cas d'utilisation*.

Nous verrons que sur ces deux espaces – celui des *sources* et celui des *cas d'utilisation* – se projètent respectivement deux classes de relations : hyponymie et isotopie.

2.a Désignation des sources sonores et hyponymie

On a constaté dans la première partie que, pour l'essentiel, le nom des sons correspond à une description (toujours personnelle et plus ou moins précise) de la source sonore. Notre hypothèse pour expliquer cela (hypothèse soutenue par les monteurs sons rencontrés) : la source

sonore, réelle ou figurée, est la seule chose *absolument certaine* au moment du nommage du son. Soit parce que le monteur a lui-même enregistré le son et en a la mémoire, soit parce que l'information est présente dans « l'annonce » prononcée par le preneur de son au moment de l'enregistrement. Cet effet de la mémoire explique d'ailleurs pourquoi il est généralement plus laborieux de nommer/dérusher des sons que l'on n'a pas enregistré soi-même.

Se pose alors une question, déjà évoquée sous l'angle de la disjonction « nommage synthétique/nommage descriptif » : « Avec quelle précision convient-il de décrire la source ? »

Exemple :

α. `oiseau gazouille` contre `pouillot véloce gazouille`.

β. `Usine – activité machines et ventilation` contre `Imprimerie – activité machine et ventilation`.

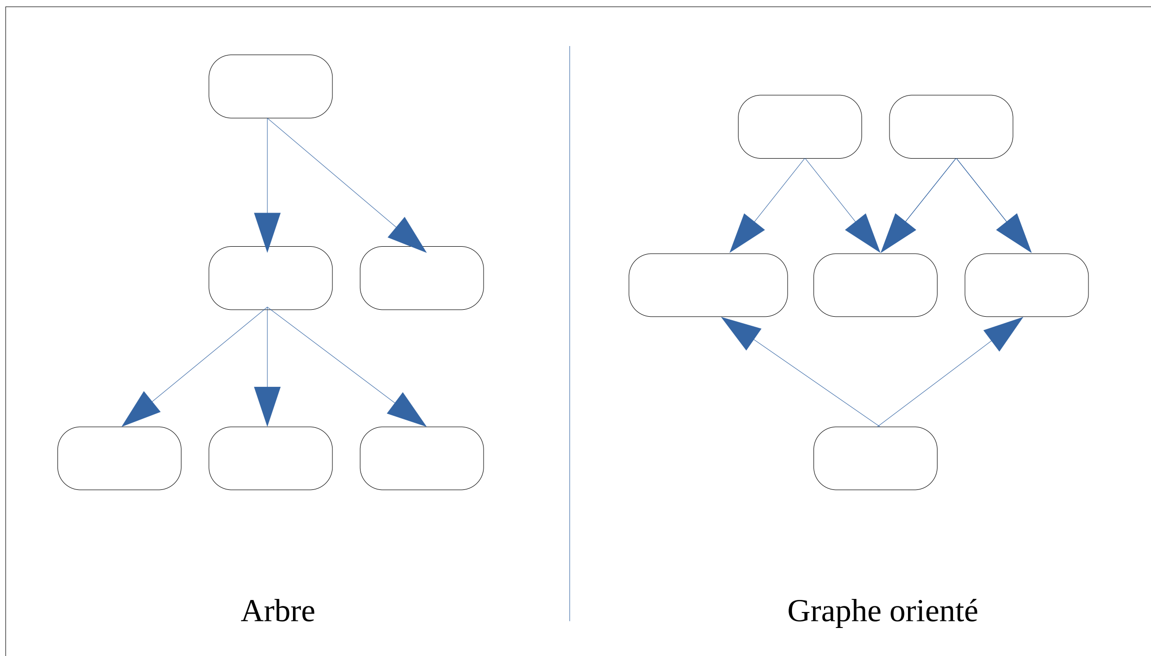
On a étudié, dans la première partie, les implications de cette disjonction, vues du moteur de recherche. Analysons-là maintenant avec les outils d'analyse sémantique habituellement appliqués au langage naturel.

Dans α, il s'agit de remarquer que le *syntagme* `pouillot véloce` présuppose la manifestation du *sème*[14] `oiseau`. De même, le *sème* `oiseau` présuppose la manifestation des sèmes `animal` (comme élément de la disjonction `animal/végétal` par exemple) et `volant` (sauf peut-être pour les autruches).

Cette relation de « présupposition logique » est nommée *hyponymie*. En adoptant les notations utilisées par Greimas dans *Sémantique structurale*[15][9], on note : $s_1 \rightarrow s_2 \ (général \rightarrow particulier)$. **Ce qui se lit, en français :** « s_2 présuppose s_1 », ou encore, en mathématiques : $s_2 \Rightarrow s_1$. *s_2* est dit *hyponyme* de *s_1*, tandis que *s_1* est dit *hyperonyme* de *s_2*.

Dans le langage courant, on peut associer cette relation avec la définition des catégories/sous-catégories, à ceci près qu'elle ne correspond pas à une structure arborescente (comme les dossiers/sous-dossiers d'un système de fichiers), mais à un graphe orienté :

14 En tant qu'unité minimale de signification. Plus de détails dans l'annexe B : Éléments de sémantique.
15 Greimas, Algirdas Julien, *Sémantique structurale*. Paris : Presses Universitaires de France, 1966, 294 p.



Ces exemples permettent déjà d'esquisser une méthode permettant de deviner les catégories auxquelles appartient un lexème ou un syntagme donné par contagion de la relation d'hyponymie. Autrement dit, si l'on parvient à documenter et à représenter cette relation, on peut alors chercher parmi les « voisins » d'un lexème.

Un tel voisinage peut se formuler comme suit : pour deux lexèmes *s* et *l*, si *l* appartient à l'ensemble des contextes sémiques de *s* (noté *Cs(s)*) et que *σ* est un hyperonyme de *l*, alors *σ* est un hyperonyme de *s*.

Formellement : $\begin{cases} l \in Cs(s) \\ \sigma \rightarrow l \end{cases} \Rightarrow \sigma \rightarrow s$

De cette façon, on peut construire progressivement *l'ensemble des hyperonymes* de *s*, sans avoir à les documenter un par un. De fait, on n'utilisera pas une présupposition stricte, mais une présupposition « la plus probable », établie à partir de la co-occurrence des lexèmes deux à deux. Autrement dit : deux lexèmes sont en relation s'ils sont présents dans les mêmes noms de fichiers.

Documenter les hyperonymies permet donc de mettre en relation des éléments, parfois mutuellement exclusifs, qui sont des cas particuliers d'un ensemble plus général. Autrement dit, *s* et *l* sont en relation *s'ils ont un hyperonyme commun*.

Formellement : $s \| l \Leftrightarrow (\exists \sigma,\ \sigma \rightarrow s \text{ et } \sigma \rightarrow l)$

Cette relation à lieu :

- du particulier au particulier – **Exemple :** `voiture || camion`
- du particulier au général – **Exemple :** `voiture || véhicule`.

Cependant, si l'on déduit l'hyponymie du nom des sons, alors on risque de ne pas tenir compte des morphologies *sonores* communes donc des cas d'utilisations communs, si les sources sont « trop éloignées ». C'est pourquoi on s'intéresse à une deuxième relation : *l'isotopie* qui constitue l'espace des cas d'utilisation.

2.b L'espace des cas d'utilisations

Si l'étude de la source sonore donne des informations sur les utilisations potentielles d'un son, elle ne les épuise pas. En effet, un des procédés du montage consiste à utiliser un son pour figurer une autre source que la sienne. C'est aussi un des ressorts essentiels du travail de bruitage.

L'exemple des sons issus du bruitages est à ce titre très particulier : ils sont nommés suivant la source qu'ils sont sensés figurer – c'est-à-dire leur cas d'utilisation *supposé* au moment de l'enregistrement – et non pour décrire la source réelle du son.

Exemple :

- `Pas dans la neige` pour `Toile remplie de riz`
- `Vent dans les feuilles` pour `Sac à gravat frotté doucement`

Toutefois, on voit immédiatement se dresser une barrière infranchissable : on ne peut renseigner *tous* les cas d'utilisation d'un son. Non seulement il semble illusoire d'envisager simultanément, au moment du nommage, tous ces cas, mais en plus cela représente une quantité d'information que ne peut supporter le nom du fichier.

Pour autant, nous soutenons que l'écoute d'un son (*i.e.* la perception) et donc sa source (en tant que cause matérielle) contiennent *en puissance* tous les cas d'utilisation du son en question.

Il s'agit là de l'application d'un postulat de Bergson dans *Matière et mémoire*[16][10]. En effet, au début de *Matière et mémoire*, Bergson explique que : *« Notre représentation de la matière est la mesure de notre action possible sur les corps »*, prenant alors l'exemple de la photographie : *« comment ne pas voir que la photographie, si photographie il y a, est déjà prise, déjà tirée dans l'intérieur même des choses […] .»* Et Bergson de caractériser ensuite le lien direct entre représentation et perception : *« Toute perception attentive suppose véritablement, au sens étymologique du mot, une réflexion, c'est-à-dire la projection extérieure d'une image activement créée, identique ou semblable à l'objet, et qui vient se mouler sur ses contours. »* La représentation survenant donc *au moment* de la perception, qui est la sélection de nos actions potentielles sur l'objet en question.

Dans le cas du son enregistré (matériau utilisé par le monteur) sous forme de fichiers, le nom du fichier est une représentation de l'enregistrement. En effet, on monte l'enregistrement pour ce qu'il *est* et, plus précisément, pour ce que l'on en perçoit, pas pour son nom. Mais cette représentation contiendrait donc *en puissance* l'ensemble des utilisations potentielles de ce son (c'est-à-dire *« la mesure de notre action possible sur [ce] corps. »).* Notre objectif est de trouver la trace de ces potentialités dans l'*isotopie*.

> **Aparté : Opportunité d'un historique des cas d'utilisations**.
>
> Si l'on ne peut prévoir tous les cas d'utilisation au moment du nommage du son, une piste consiste à les mémoriser. Ainsi, en « regardant » quels sons sont utilisés ensembles, on pourrait tendre vers la représentation de liens sémantiques qui dépassent ceux habituellement exprimés dans le langage, et qui tiendrait à une isotopie du son décorrélé de sa source (puisqu'on ne s'intéresserait plus qu'au son une fois utilisé comme élément de narration pour un film.
>
> On pourrait donc imaginer un programme qui « apprenne », grâce à des modèles d'apprentissage par renforcement, quels sons sont susceptibles d'être utilisés ensembles, pour faire des propositions « d'ordre élevés », plutôt que de proposer deux sons systématiquement utilisés ensemble, en proposer un troisième lié au premier mais pas au deuxième.
>
> Une telle piste pose plusieurs question.

16 Bergson, Henri, *Matière et mémoire*. Paris : Flammarion, 2012, 349 p.

D'abord : comment détecter que deux sons ont un cas d'utilisation commun ? Une piste consiste à choisir les sons qui apparaissent « en même temps » dans une session de travail. Mais cette information est vraisemblablement bruitée par l'infinité des cas particuliers que constituent les films et leurs séquence. De plus obtenir cette information est un problème d'ingénierie complexe pour que le procédé soit transparent pour l'utilisateur. Par exemple : si le procédé est automatique, comment permettre à l'utilisateur d'introduire des cas d'exception, sans alourdir la tache ? Comment faire pour que l'information soit transmise directement de la station de travail au logiciel de gestion de sonothèque, quand ces stations protègent la plupart de leur protocoles de communication.

Ensuite : pour que les modèles d'apprentissage par renforcement commencent à établir des liens significatifs entre les objets, il faut un échantillon de données conséquents (typiquement : plusieurs millions d'itérations). Si de tels échantillons existent pour la reconnaissance d'image, il n'en est pas de même pour le montage son. D'autant qu'il n'est pas évident que l'on puisse « mutualiser » l'information (autrement dit, le « modèle » d'un monteur s'applique-t-il à ses collègues ?). Ce faisant, **le temps nécessaire pour obtenir un programme performant peut être très long.[17][11]**

Enfin se pose **la question de la collecte de ces informations.** Quelle est la valeur de cette information ? Les monteurs seraient-ils prêts à la céder ? En effet, on touche, avec l'utilisation des sons particuliers, au cœur du mouvement créatif du montage son.

L'étude de ces questions constituerait un travail à part entière que nous ne mènerons pas ici, d'autant que sa réalisation concrète est largement hors de portée dans le cadre de ce mémoire.

17 *cf.* Sutton, Richard et Barto Andrew, *Reinforcement Learning: An Introduction*. Cambridge : The MIT Press, 2012, 334 p.

2.c L’isotopie

L’isotopie est définie, notamment dans *Sémantique structurale*[18], comme la redondance de catégories sémantiques au sein d’un énoncé, pour en résoudre l’ambiguité. Elle peut donc être définie entre deux lexèmes comme : ’intersection de leurs noyaux sémiques respectifs, augmentée de l’intersection de l’ensemble de leurs contextes sémiques potentiels.

Comment ce concept se traduit-il concrètement dans le cas des sons (et, plus précisément, de leur noms ?

On prendra comme exemple, dans la suite, l’ensemble des sons d’ambiances d’intérieurs. Avec une définition large : des sons plutôt longs (typiquement d’au moins une minute), qui ne visent pas un événement sonore ponctuel particulier, enregistrés dans des lieux clos.

Exemples :

Nom d’origine	Nom « épuré »
Fond int appart circul.L.wav	silence interieur appartement circul
Hopital amb int bureau (porte ouverte) vie discussion moy.wav	Hopital ambiance interieur bureau porte ouverte vie discussion moyenne
fond air interieur calme Leonard matin 8h oiseaux circulation Froncardo.L.wav	Fond air interieur calme leonard matin 8h oiseau circul froncardo
vent sifflant int eglise.L.wav	Vent siffl interieur eglise
Silence int chez Manon Salon fenêtre ouverte un peu de vie extérieure.L.wav	Silence interieur chez manon salon fenetre ouverte un peu de vie exterieur
INT APP HLM JOUR SALON FO.L.wav	Interieur appartement hlm jour fenetre ouverte
Fond racc int refrigerateur .L.wav	Silence raccord interieur refrigerateur
CIRCUL INT MANSARDE JOUR FENÊTRE OUVERTE.L.wav	Circul intérieur mansarde jour fenêtre ouverte

On peut, à travers ces exemples, distinguer trois catégories fonctionnelles de lexèmes :

18 Greimas, Algirdas Julien, *op. cit.*

- Lexèmes « principaux » : C'est, le plus souvent, le nom du lieu (`bureau, hôtel, hôpital, église, ...`). Ils contiennent généralement le noyau sémique du son.
- Lexèmes « contextuels » : souvent des adjectifs (`calme, agité, vide, ...`) mais parfois aussi des noms (`nuit, activité, brouhaha, ...`). Ils caractérisent le lieu. Dans le sémème (en tant qu'*effet de sens*), ils peuvent s'identifier au contexte sémique.
- Lexèmes « quantificateurs » : principalement des adverbes (`très, peu, sans, quelques, ...`). Associés aux lexèmes « contextuels » qui appartiennent à une disjonction, ils en changent la valeur le long de cette disjonction. **Exemple :** $((calme/agité) \rightarrow calme) \Rightarrow très\ calme <_{(calme/agité)} calme$.

Ce classement permet de réduire la question de l'isotopie entre les sons à l'isotopie des lexèmes principaux d'une part et à celle des lexèmes contextuels d'autre part.

On traitera d'abord des lexèmes contextuels, car on verra que cela facilite l'étude des lexèmes principaux.

2.c.α Représentation des lexèmes contextuels

L'étude des lexèmes contextuels revient à l'étude des « dimensions » selon lesquelles peuvent s'étendre une classe de sons. Recenser ces dimensions permet ensuite de placer les sons dans l'espace de ces dimensions et de mesurer la distance qui les sépare.

Exemple : dans le cas des ambiances d'intérieurs, on peut relever les disjonctions suivantes :

- `réverbération : mat / (hall`[19] `~ réverbérant)`
- `agitation : calme / (activité ~ agité ~ animé)`
- `voix : aucune voix / (voix ~ discussion ~ paroles)`

Pour une classe de sons donnée, on fait l'hypothèse que le vocabulaire utilisé pour décrire chaque dimension est relativement restreint. La réduction morphologique restreignant encore ce vocabulaire.

---

19 Anglicisme utilisé comme adjectif désignant une réverbération longue.

Exemple :

$$\begin{pmatrix} activité & agité & animé \\ activités & agitée & animées \\ actif & agitation & animation \end{pmatrix} \Leftrightarrow \begin{pmatrix} act & agit & anim \end{pmatrix}$$

De plus, on peut noter que certains lexèmes contextuels opèrent à un niveau supérieur, sur plusieurs dimensions à la fois.

Exemple :

$$calme \rightarrow nuit \;\; ; \begin{pmatrix} calme \\ aucune\ voix \end{pmatrix} \rightarrow nuit$$

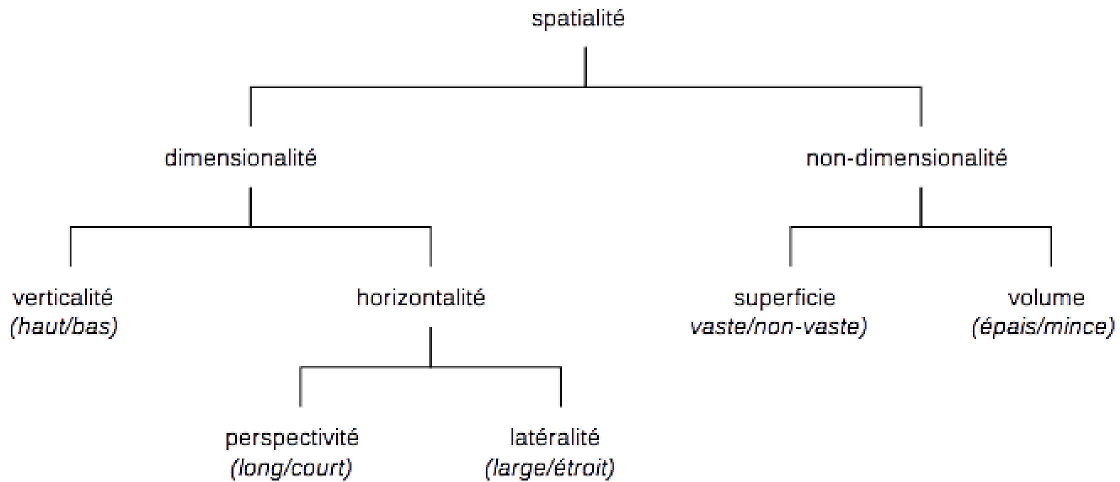
Opérant à plus haut niveau, ils ont aussi une priorité inférieure. On rappelle que la relation se lit « `nuit` *présuppose* `calme` ».

Exemple : `Hôpital la nuit - salle d'attente animée`. Ici, la relation $(calme \rightarrow nuit)$ est niée par la caractérisation `animée`, de plus bas niveau, qui prend alors le dessus.

Exemple : `Hôpital la nuit - salle d'attente vide - qques voix` (*σ*). On a alors $\begin{pmatrix} calme \\ voix \end{pmatrix} \rightarrow \sigma$.

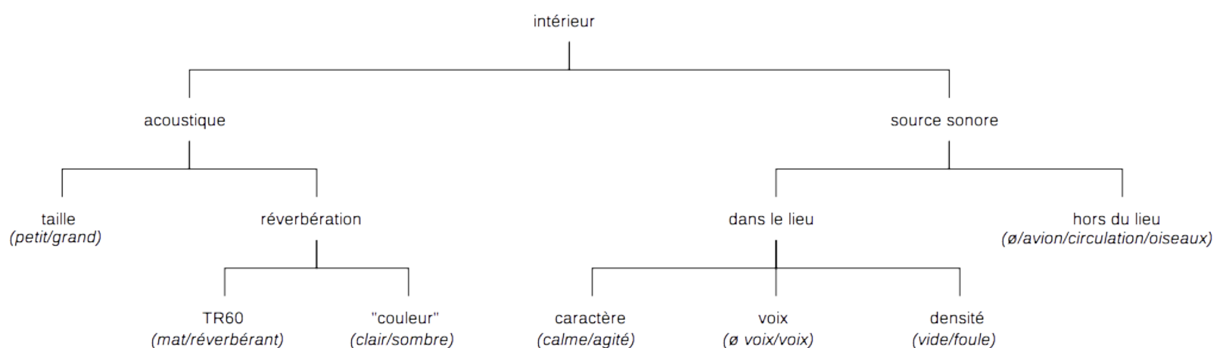
L'ensemble de ces dimensions, une fois recensées, peut-être hiérarchisé. On trouve chez Greimas[20] l'exemple suivant :

20 Greimas, Algirdas, Julien, *op. cit.*



Arbre des catégiries sémique de spatialité

Quand au cas des ambiances d'intérieurs, une telle arborescence peut être esquissée :



Arbre des catégories sémiques décrivant les ambiances d'intérieur

À partir d'une telle arborescence, nous pouvons commencer à mesurer des liens entre les concepts. **Exemple :** `calme` est plus proche de `aucune voix` que de `mat`.

L'enjeu déterminant réside alors dans l'exhaustivité et la pertinence des disjonctions recensées. Ce recensement représente un travail considérable, qui doit être quasiment mené « de zéro ». En effet, si des bases de données existent pour le langage naturel (dont *WordNet*[21][12] et

---

21 Princeton University, *WordNet* | *A Lexical Database for English*. https://wordnet.princeton.edu/, 1998 [consulté le 21 avril 2018]

WOLF[22][13]), les mots utilisés pour décrire les sons ne sont pas toujours ceux utilisés pour décrire leurs sources dans le langage naturel.

Une fois cette structure, c'est à dire l'arbre reliant les différentes dimensions entre elles et les éventuelles relations transversales, correctement documentée, on a alors défini l'espace des contextes sémiques disponibles pour une classe de sons. Autrement dit : « quelles sont les dimensions selon lesquelles peuvent se développer ces sons, et comment ces dimensions sont-elles liées ? ». Voyons maintenant comment cela peut nous renseigner sur le noyau sémique (c'est-à-dire les lexèmes « principaux »).

2.c.β Décomposition des lexèmes principaux

L'objectif ici est de pouvoir mettre en relation des lexèmes renvoyant à des significations complexe. Pour les traiter, on les décompose à l'aide des lexèmes contextuels présentés plus haut.

Exemple : Pourquoi y a-t-il de grandes chances pour que `Fond d'air - église` ressemble à `Grand hall calme` ? Sûrement parce qu'implicitement :

$$\begin{aligned} \textit{église} &\leftarrow \begin{pmatrix} \textit{grand} \text{ (relativement à } \textit{taille}) \\ \textit{réverbérant} \text{ (relativement à } TR60) \end{pmatrix} \\ \textit{hall} &\leftarrow \left(\textit{réverbérant} \text{ (relativement à } TR60)\right) \end{aligned}$$

En effet, on remarque que le noyau sémique est le plus souvent implicite[23] et que le contexte sémique vient le préciser, c'est-à-dire ajouter des dimensions, ou l'infléchir, changer sa valeur suivant une dimension.

Exemple :

On désigne rarement `une chaise à quatre pieds` ou `une chaise avec dossier` (on considère généralement qu'il s'agit là de pléonasmes). En revanche `une chaise` **`en bois/ à trois pieds/sans dossier`** apporte une information supplémentaire

On peut donc envisager, pour chaque lieu, de documenter ses caractéristiques « les plus probables » selon chacune des dimensions précédemment recensées.

---

22 Sagot Benoît et Fišer Darja, *Building a free French wordnet from multilingual resources*. May 2008, Marrakech, Morroco.

23 *cf.* Greimas, Algirdas Julien, *op. cit.*

On peut alors mettre en relation des lexèmes principaux, même quand le contexte sémique n'est pas formulé pour toutes les dimensions.

In fine, l'effet de sens est donc représenté par :

$$Ns+Cs = \begin{pmatrix} dimension_1 & : & valeur_1 \\ dimension_2 & : & valeur_2 \\ dimension_3 & : & valeur_3 \\ \vdots & : & \vdots \end{pmatrix} + \begin{pmatrix} dimension_2 & : & contexte_2 \\ dimension_3 & : & contexte_3 \times quantification_3 \end{pmatrix}$$

$$Ns+Cs = \begin{pmatrix} dimension_1 & : & valeur_1 \\ dimension_2 & : & contexte_2 \\ dimension_3 & : & contexte_3 \times quantification_3 \\ \vdots & : & \vdots \end{pmatrix}$$

Où N_s et C_s désignent respectivement le *noyau sémique* et le *contexte sémique*. Toutefois, cette décomposition doit se faire manuellement et est soumise aux biais de celui qui l'effectue (biais qui semble difficile à évaluer).

Nous avons donc vu comment tenir compte de l'hyponymie permet de relier les sources sonores entre elle ; puis comment, en décomposant les lexèmes pour en obtenir l'isotopie, on peut supposer des cas d'utilisation communs. Cependant, représenter ces deux relations suppose une documentation minutieuse dont il est difficile de mesurer les biais.

On se propose donc d'étudier une méthode « automatique » qui, à défaut d'être *a priori* plus performante, sera certainement plus facile à évaluer ainsi qu'à déployer sur un champ plus large que celui des ambiances d'intérieurs. Il l'agit de l'*analyse sémantique latente*.

3 Analyse sémantique latente

Jusqu'ici, nous avons :

- décrit les outils conceptuels nécessaires pour représenter les liens sémantiques (noyau sémique, contexte sémique, lexème principaux/contextuels/quantificateurs, …),
- proposé une méthode manuelle de documentation de ces liens.

Cependant, la quantité de données à traiter, typiquement plusieurs téraoctets de données, représentant plusieurs centaines de milliers d'entrées, et la diversité des classes de sons sont

telles que, dans le temps de ce mémoire, nous n'aurons la possibilité que de réaliser la décomposition des « ambiances d'intérieur » (*cf.* Partie III).

D'où la nécessité d'étudier une approche automatique du problème.

Deux traits communs à toutes les sonothèques nous permettent d'envisager *a priori* une méthode statistique :

- Le caractère ordonné de la sonothèque, c'est-à-dire que les mots dans les noms des sons ne sont pas associés « au hasard » : les corrélations mesurées auront donc une chance d'être significatives sur le plan sémantique.
- La grande quantité d'échantillons assure que les résultats observés convergent vers les modèles mathématiques utilisés pour les décrire (loi faible des grands nombres).

L'*analyse sémantique latente* (*ASL*) est une méthode d'indexation, utilisée en recherche de l'information, pour deviner des liens entre des termes à travers un corpus de document en mesurant l'occurrence des termes dans ce corpus.[24][14]

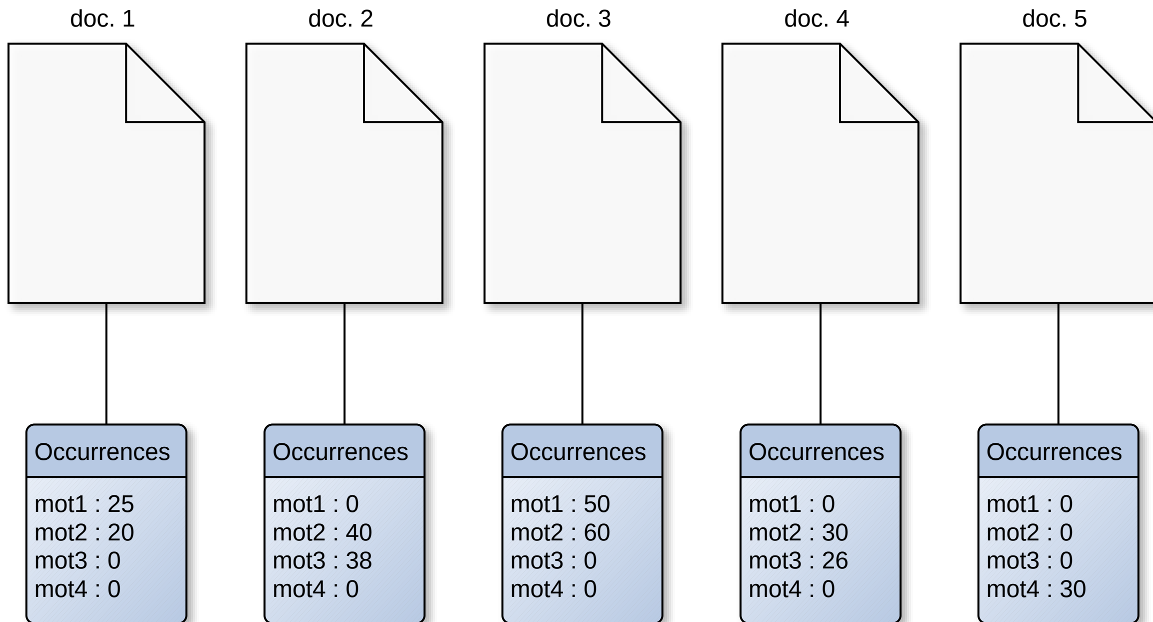
Pour présenter l'*ASL*, nous allons commencer par présenter une version simplifiée du problème. Nous étudierons ensuite une mesure de l'occurrence des termes dans un ensemble de documents (ici : l'ensemble des noms des sons). Enfin, nous verrons comment *l'ASL* permet d'obtenir une mesure de la corrélation entre les termes.

3.a Introduction à l'*ASL*

L'hypothèse la plus simple pour induire *a priori* un lien sémantique entre des mots consiste à supposer que deux mots qui apparaissent souvent ensemble dans un ensemble de documents ont un lien sémantique. Pour un ensemble de documents donné, on compte alors les occurrences de chaque mot dans chaque document.

24 *cf.* Landauer, Thomas et al., An Introduction to Latent Semantic Analysis. *Discourse Processes*. Vol. 25, 1998, pp. 259-284

Exemple :



Représentation schématique de l'analyse sémantique latente

Dans cet exemple, on fait alors l'hypothèse que $mot_1 \sim mot_2$ et $mot_2 \sim mot_3$. Mais surtout, si suppose la transitivité de $\sim$, alors $mot_1 \sim mot_3$, alors qu'ils ne se rencontrent dans aucun document.

On voit toutefois apparaître au moins deux limites à cette approche simplifiée :

- Comme souligné en 2, la co-occurrence d'un noyau sémique et d'un contexte sémique ne donne pas d'information sur ce qu'« est » ce noyau (c'est-à-dire sa décomposition complète), mais seulement sur ce qu'il « peut être ». Autrement dit, si, à travers l'échantillon analysé, une partie du noyau demeure implicite, alors on n'arrive pas à la détecter (**Exemple :** s'il n'est jamais dit q'une `église` est `calme`, alors on ne peut le deviner *ex nihilo*).
- Certains mots apparaissent très souvent dans un ensemble de document, sans participer à l'isotopie du document. C'est le cas de ce qu'on appelle les *mot vides* (*stop-words* en anglais).

Étudions maintenant une mesure plus précise de l'occurrence des termes dans un ensemble de documents : la *Term Frequency – Inverse Document Frequency* (*TF-IDF*), littéralement : *Fréquence du terme – Fréquence inverse du document*.

3.b Une mesure de l'occurrence des mots : la *TF-IDF*

Il s'agit, dans le champs de la recherche d'information, d'une mesure de l'importance d'un terme relativement à un corpus de document.[25][15]

TF-IDF signifie *term frequency-inverse document frequency,* que l'on peut traduire par « fréquence du terme – fréquence inverse de document »[26].

- ***Term frequency*** **:** c'est le nombre d'occurrence d'un terme (*t*) dans un document (*d*). On peut la mesurer de plusieurs façon, notamment : *binaire* (1 si $t \in d$, 0 sinon), *brute* (directement le nombre d'occurrence de *t* dans *d*, noté $f_{t,d}$), *normalisée* (fréquence divisée par la fréquence du terme le plus fréquent dans le document $\frac{f_{t,d}}{max_{(v \in d)} f_{v,d}}$).
- ***Inverse document frequency*** **:** cette mesure vise à réduire l'importance des mots « trop fréquents » dans le corpus (typiquement, les déterminants ou les pronoms). On considère alors que les mots « rares » donnent plus d'informations sur le contenu sémantique du document. On la définie comme le logarithme (en base 10) du rapport entre le nombre de documents (*D*) et le nombre de documents dans lesquels *t* est présent. Soit : $idf_t = \log \frac{D}{Card(d | t \in d)}$ (le logarithme permet de conserver une commune mesure entre des documents ou des corpus de tailles variées).

In fine, pour un terme *t* dans un document *d*, on a donc la mesure : $tfidf(t,d) = tf_{t,d} . idf_t$.

Autrement dit, la *TF-IDF* du terme *t* dans le document *d* est le rapport entre le nombre d'occurrences du terme dans le document et du nombre d'occurrence du terme dans *tous le corpus*.

---

25 Baeza-Yaetes, Ricardo et Bibeiro-Neto Berthier, *Modern Information Retrieval*. New York : ACM Press, 1999, 513 p.

26 Nous n'avons pas trouvé d'équivalent en Français utilisé dans la littérature sur le sujet.

On peut encore améliorer la mesure de l'*IDF* en supprimant *à l'avance* les *mots-vides*. On trouve des listes pour ces mots dans le projet *CLEF* ([16] pour l'anglais[27] et [17] pour le français[28]).

On note cependant une précaution à prendre quant à la mesure de l'*IDF* : il arrive qu'un mot soit présent *dans* tous les noms de fichiers d'une collection. Il correspond généralement à la catégorie à laquelle appartiennent ces sons, autrement dit, c'est un hyperonyme de ces sons. Or, comme il est « trop courant », l'*IDF* minorera son importance. Une solution consiste alors à analyser plusieurs collections différentes *en même temps*. Ainsi, ce mot sera courant dans un sous-ensemble, mais rare dans la totalité de l'échantillon et ainsi favorisé par l'*IDF*.

Maintenant que nous disposons d'une mesure robuste de l'importance des termes dans un ensemble de documents, voyons comment on peut mesurer les liens entre ces termes.

3.c Description de l'analyse sémantique latente

L'analyse sémantique latente (*latent semantic analysis* en Anglais, *ALS* dans la suite) se déroule en trois étapes :

- construction de la matrice des occurrences,
- décomposition de la matrice en valeurs singulières et réduction du rang,
- calcul des similitudes entre les termes-concepts.

Pour la suite, on considère un dictionnaire de *m* termes et un corpus de *n* documents.

3.c.α Construction de la matrice des occurrences

Il s'agit de la matrice dont les lignes représentent les termes du dictionnaire et dont les colonnes représentent les documents du corpus. Autrement dit, c'est un tableau dont les cases contiennent « l'importance du terme *t* dans le document *d* ». Ainsi, la case de coordonnées (i, j) a pour valeur $tfidf(t_i, d_j)$.

Formellement, on a :

27 Savoy, Jacques, *IR Multilingual Resources at UniNE - English stopwords*. http://members.unine.ch/jacques.savoy/clef/englishST.txt, 2016 [consulté le 21 avril 2018]
28 Savoy, Jacques, *IR Multilingual Resources at UniNE - French stopwords*. http://members.unine.ch/jacques.savoy/clef/frenchST.txt, 2016 [consulté le 21 avril 2018]

$$X \in M_{m,n}(\mathbb{R}),\ \forall (i,j) \in [\![1;m]\!] \times [\![1;n]\!],\ (X)_{i,j} = x_{i,j} = tfidf(t_i, d_j)$$
$$\text{où} \forall (i,j) \in [\![1;m]\!] \times [\![1;n]\!], \begin{cases} t_i \text{ désigne le terme d'index } i \text{ dans le dictionnaire.} \\ d_j \text{ désigne le document d'index } j \text{ dans le corpus.} \end{cases}$$

X a donc l'aspect suivant :

$X = \begin{pmatrix} x_{1,1} & \cdots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{m,1} & \cdots & x_{m,n} \end{pmatrix}$ (attention, X n'est pas carrée, mais de dimension $m \times n$),

avec, en ligne, $\boldsymbol{t_i^T} = (x_{i,1} \ \cdots \ x_{i,n})$ le vecteur mesurant l'importance du terme t_i dans le corpus

et, en colonne, $\boldsymbol{d_j} = \begin{pmatrix} x_{1,j} \\ \vdots \\ x_{m,j} \end{pmatrix}$ le vecteur mesurant l'importance de chaque terme dans le document d_j.

La corrélation entre deux termes (t_i, t_p) est alors donnée par le produit scalaire $(\boldsymbol{t_i} | \boldsymbol{t_p}) = \boldsymbol{t_i^T} . \boldsymbol{t_p}$. Ainsi, le produit $X . X^T$ contient tous ces produits scalaires : la case (i, p) ayant pour valeur $\boldsymbol{t_i^T} . \boldsymbol{t_p}$.

De même, $X^T . X$ contient tous les produits scalaires $\boldsymbol{d_j^T} . \boldsymbol{d_q}$, qui mesurent la corrélation entre les documents (d_j, d_q) .

Cependant, la matrice ainsi obtenue n'est pas directement exploitable, pour au moins deux raisons :

- Pour un document donné, il est probable que la majorité des termes du dictionnaire en soient absent. Autrement dit, *X* est essentiellement composée de valeurs nulles (on désigne une telle matrice comme « creuse »). Ces valeurs nulles diminuent d'autant l'efficacité en mémoire de l'analyse.
- Les termes « anecdotiques », apparaissant très peu à travers le corpus, bruitent l'information contenue dans la matrice. Supprimer ces termes permettra d'améliorer le rapport signal/bruit des résultats d'analyse.

Pour réduire la taille de la matrice *X*, on peut « préparer » les documents, selon les mécanismes décrits en 1. (traitement de la diversité morphologique, racinisation), et en 3.b (suppression des *mots vides*).

Pour plus d'efficacité, on procède à une *réduction du rang*[29] de *X*. L'idée est de supprimer les termes de faible importance, ou de les combiner entre eux.

3.c.β Réduction du rang

Pour pouvoir réduire la taille de la matrice des occurrences (*X*), on ne peut se contenter d'en supprimer des lignes. En effet, supprimer « directement » un terme (une ligne) supprime aussi les liens de ce termes avec les autres et donc quantité de liens indirectes (cf. 3.a : $mot_1 \sim mot_2$ et $mot_2 \sim mot_3$ implique $mot_1 \sim mot_3$).

On effectue donc une *décomposition en valeurs singulières* de la matrice. Cette décomposition se présente comme suit :

Pour X une matrice rectangulaire de dimensions $m \times n$, il existe une factorisation de X de la forme : $X = U.\Sigma.V^T$,
où U et V sont deux matrices orthogonales carrées de dimensions respectives m et n,
et où Σ est une matrice rectangulaire diagonale de dimension $m \times n$.

Les coefficients diagonaux de Σ sont appelés *valeurs singulières* de *X*, tandis que les vecteurs de *U* (respectivement V) sont appelés *vecteurs singuliers à gauche* (respectivement *à* droite) de X.

Si l'on écrit les produits de corrélations évoqués en 3.c.α suivant cette décomposition, on obtient :[30]

(1) $X.X^T = (U.\Sigma.V^T).(U.\Sigma.V^T)^T = (U.\Sigma.V^T).((V^T)^T.\Sigma^T.U^T)$
$= U.\Sigma.V^T.V.\Sigma^T.U^T = U.\Sigma.\Sigma^T.U^T$

et

(2) $X^T.X = (U.\Sigma.V^T)^T.(U.\Sigma.V^T) = ((V^T)^T.\Sigma^T.U^T).(U.\Sigma.V^T)$
$= V.\Sigma^T.U^T.U.\Sigma.V^T = V.\Sigma^T.\Sigma.V^T$

$\Sigma^T.\Sigma$ et $\Sigma.\Sigma^T$ étant diagonales (comme produits de matrices diagonales), on a donc :

(1) $\Rightarrow$ U est une base des vecteurs propres de $X.X^T$
et
(2) $\Rightarrow$ V est une base des vecteurs propres de $X^T.X$

---

29 Le rang d'une matrice est la dimension de l'espace engendré par les vecteurs qui la constituent.

30 $(A.B)^T = B^T.A^T$ (par transposition) et car $U^T.U = V^T.V = I_d$ puisque U et V sont orthogonales.

Les deux produits $X.X^T$ et $X^T.X$ ont donc les mêmes valeurs propres non nulles, à savoir les coefficients diagonaux non nuls de $\Sigma.\Sigma^T$.

D'où la décomposition :

$$X=\left(\left(\mathbf{u_1}\right)\cdots\left(\mathbf{u_l}\right)\right).\begin{pmatrix}\sigma_1 & \cdots & 0\\ \vdots & \ddots & \vdots\\ 0 & \cdots & \sigma_l\end{pmatrix}.\begin{pmatrix}(& \mathbf{v_1} &)\\ & \vdots & \\ (& \mathbf{v_l} &)\end{pmatrix}=U.\Sigma.V^T$$

Alors, dans *U*, seule la *i*-ème ligne contribue au vecteur $\boldsymbol{t_i}$. On nomme $\boldsymbol{\hat{t}_i^T}$ ce vecteur ligne de *U*.

De même, dans V^T, seule la *j*-ème colonne contribue au vecteur $\boldsymbol{d_j}$. On nomme $\boldsymbol{\hat{d}_j}$ ce vecteur colonne de V^T.

Pour réduire le rang (et ultimement la taille) de *X*, on sélectionne les *k* plus grandes valeurs singulières de *X*, et leurs vecteurs singuliers associés.[31]

Une fois le rang de *X* réduit, comment peut-on mesurer l'intensité des relations entre les termes et les documents ?

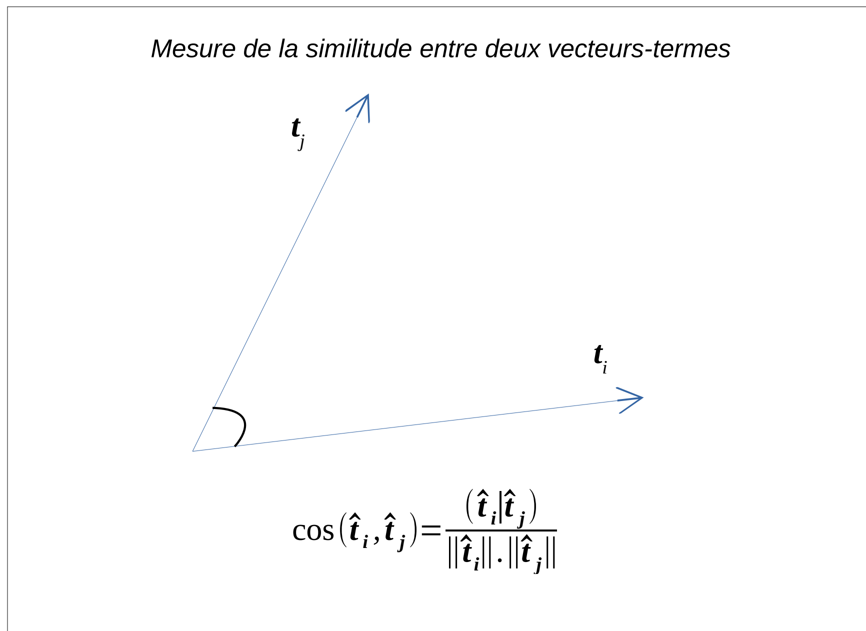
3.c.y Mesure des similitudes

La réduction de rang *k* de *X*, que l'on note maintenant $X_k=U_{k.}\Sigma_{k.}V_k^T$ « projette » les vecteurs *termes* et *documents* dans l'espace des *concepts* (autrement dit, chaque concept est une des *k* dimensions selon lesquelles on a décomposé la matrice *X*).

Les *k* composantes du vecteur $\boldsymbol{\hat{t}_i}$ mesurent donc l'importance du terme t_i dans chacun de ces concepts (et réciproquement pour les composantes du vecteur $\boldsymbol{\hat{d}_j}$).

On peut donc mesurer la similitude entre deux termes en mesurant le cosinus de l'angle entre les vecteurs qui les représentent :

31 L'application du théorème de Eckart-Young-Mirsky prouve qu'il s'agit alors de la meilleure approximation de rang *k* de la matrice *X*, pour la norme choisie (Frobennius, dans le cas de l'*ALS*).



Avec une telle mesure, on a :

- $\cos(\hat{t}_i, \hat{t}_j)$ tend vers 1 quand $\hat{t}_i$ devient parallèle à $\hat{t}_j$. (*i.e.* t_i et t_j partagent les mêmes *concepts*).
- $\cos(\hat{t}_i, \hat{t}_j)$ tend vers 0 quand $\hat{t}_i$ devient perpendiculaire à $\hat{t}_j$. (*i.e.* t_i et t_j **ne partagent pas** les mêmes *concepts*).

On peut ainsi mesurer la *similitude* entre une requête, considérée soit comme un document, soit comme une collection de termes, et des termes/documents. Il suffit pour cela de traduire la requête *q* (comme *query*, en Anglais) dans « l'espace des concepts » :

$$\hat{q} = \Sigma_k^{-1} . U_k^T . q$$

4 Conclusions provisoires

L'objectif de cette partie était de trouver une méthode de représentation du contenu sémantique présent dans les noms des sons d'une sonothèque. Pour cela, nous avons séparé la tâche en deux.

D'abord, le traitement de la diversité morphologique (formes fléchies, fautes d'orthographe, *et cætera*). Cette question est réglée en deux temps :

- De façon algorithmique (avec le procédé de racinisation).
- De façon manuelle (en documentant les abréviations et les formes fautives).

La résolution de cette question nous a permis de considérer, dans la suite, que chaque *lexème* disposait d'un représentant unique. On peut alors se poser la question des relations entre les lexèmes.

Pour représenter les relations entre les lexèmes, nous avons proposé deux méthodes :

- Une méthode « manuelle », consistant à documenter les différents noyaux et contextes sémiques, pour en proposer une représentation hiérarchisée.
- Une méthode « automatique », l'analyse sémantique latente, qui consiste à mesurer la co-occurrence de termes dans un ensemble de documents pour en déduire des relations.

La partie qui suit est dédiée à l'implémentation de ces différentes méthodes.

III Implémentation du moteur de recherche

Jusqu'ici, nous avons montré pourquoi tenir compte des liens de sens entre les sons permet d'améliorer l'efficacité de la recherche (en réduisant les résultats absurdes et en élargissant le spectre des résultats pertinents) ; puis nous avons proposé une représentation de ces liens sémantiques entre les sons à partir de leur nom.

L'objectif de cette section est de proposer une implémentation concrète des concepts exposés jusqu'ici, pour montrer que leur application est possible. Nous nous attelons donc au développement d'un gestionnaire de sonothèque élémentaire qui servira de cadre de travail. Ensuite, nous y greffons un moteur chargé de renseigner le contenu sémantique de chaque son et de traduire les requêtes utilisateurs en tenant compte de ces contenus.

Pour clarifier le propos, l'implémentation de la méthode manuelle et celle de la méthode automatique d'analyse sémantique seront présentées séparément.

Nous proposons d'abord un résumé du fonctionnement du logiciel, sous forme de schéma pour ensuite présenter l'environnement dans lequel il est développé (*Electron*). Seront ensuite exposés, dans cet ordre, le traitement des variétés morphologiques (commun aux deux méthodes d'analyse sémantique), l'implémentation de l'analyse sémantique manuelle et enfin celle de l'analyse sémantique latente.

> **Aparté :** Pour l'implémentation concrète du programme et notamment pour la mise au point d'une application autonome dotée d'une interface graphique utilisable, j'ai été largement épaulé par Alex-Adrien Auger. Le programme étant encore en cours de

> développement au moment de l’écriture de ces lignes, le lien suivant permet d’accéder au code source ainsi qu’aux ressources supplémentaires [18] :
>
> https://github.com/alexadrien/ElectronSQL
> À cette adresse, le code est disponible à la lecture et au téléchargement. De plus, le projet est susceptible d’être divisé en plusieurs « branches », permettant de travailler simultanément sur plusieurs versions ou sur plusieurs fonctionnalité. Toutefois, la branche `master` contient toujours une version fonctionnelle du programme.
>
> Par souci d’autonomie de ce mémoire, nous mettrons toutefois certaines ressources en annexes, dans leur état au jour de l’impression. Le cas échéant, un lien vers la version présente sur le dépôt sera fourni.
>
> De la même façon, nous mettons à disposition un dépôt contenant **toutes** les ressources de programmation utilisées dans la suite, dans leur état au jour de l’impression. Cela inclut les différents modules tiers utilisés, les bibliothèques de code source ainsi qu’une version « figée » de l’application exactement telle qu’elle est décrite dans la suite. L’objectif est de garantir l’autonomie de ce travail – même si les librairies utilisées venaient à être modifiées voir supprimées. Ce dépôt se trouve à l’adresse suivante [19] : https://github.com/C-rror/ressources_PPM.

1 Schéma fonctionnel

Si on omet la question de l’interface utilisateur, le programme peut se décomposer en deux fonctions principales : **l’indexation** (c’est-à-dire l’ajout des sons à une base de données) et **la recherche** (c’est-à-dire la comparaison entre la requête utilisateur et la base de données).

L’indexation se déroule selon le schéma suivant :

- Le programme parcourt récursivement[32] le répertoire choisi par l’utilisateur
- Pour chaque son trouvé :

32 c’est-à-dire qu’il parcourt aussi tous les sous-répertoires.

- le nom et le chemin sont ajoutés à la base de données,
- le nom est « nettoyé » de ses abréviations en suivant les règles figurant dans un tableau,
- le nom est racinisé en suivant l'algorithme de Porter,
- le nom ainsi épuré est ajouté à la base de données (colonne « nom épuré »),
- le programme construit le contenu sémantique du nom en interrogeant le graphe des lexèmes et des contextes sémiques,
- le contenu ainsi construit est ajouté à la base de données (colonne « contenu sémantique »),

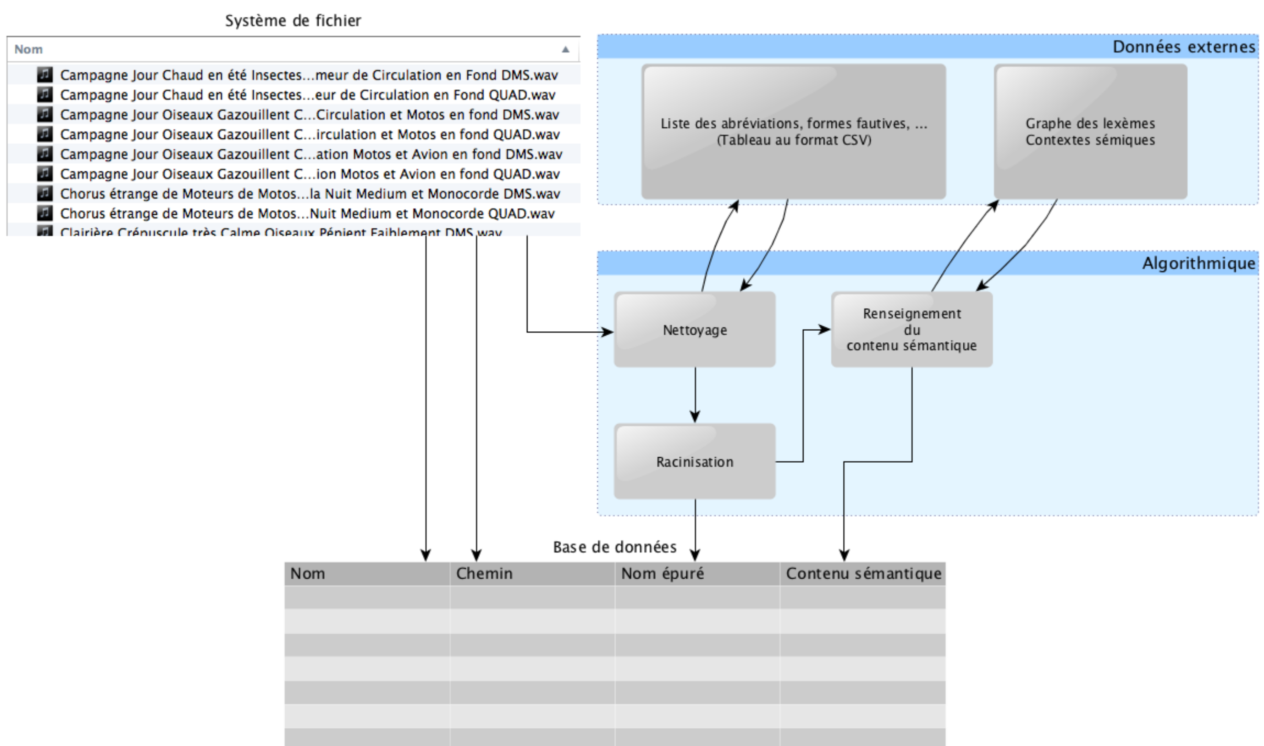


Schéma fonctionnel de l'indexation des fichiers par l'application

La recherche se déroule de façon analogue : on traite la requête de l'utilisateur comme si c'était le nom d'un son, puis on compare ce nom à la base de données.

D'où le schéma suivant :

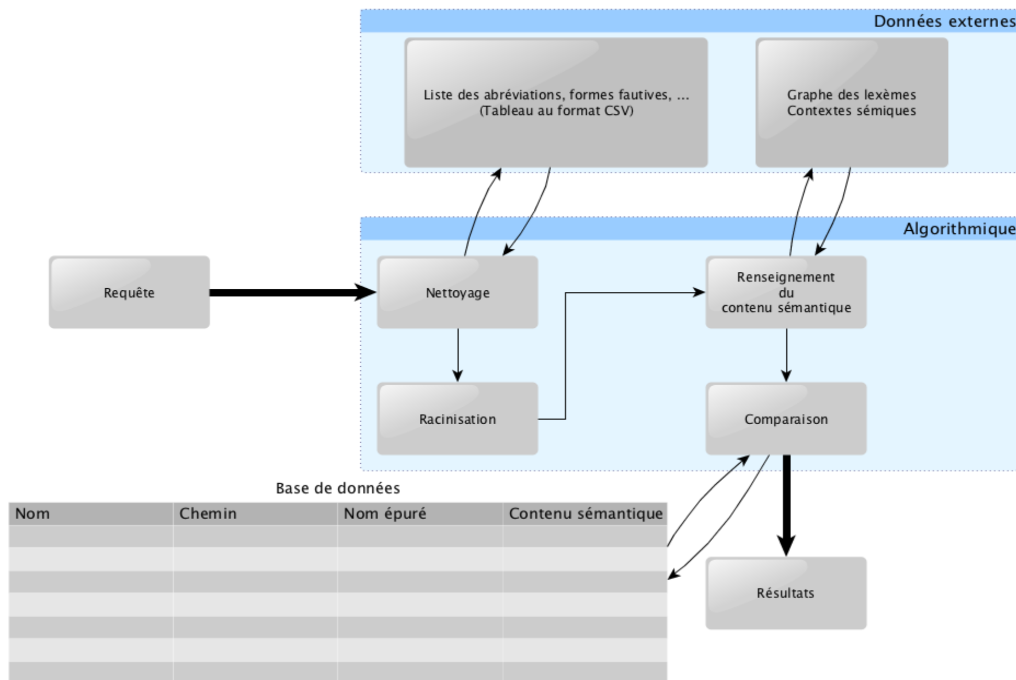


Schéma fonctionnel du traitement de la requête utilisateur

En suivant ce schéma, on comprend que le travail de ce mémoire porte sur les deux groupes fonctionnels « Données externes » et « algorithmique ».

Ainsi on en déduit les fonctionnalités de base de l'environnement de travail. Celui-ci doit être capable de :

- parcourir un répertoire donné du système de fichier,
- ajouter les sons de ce répertoire à une base de données,
- proposer un modèle de base de données sur lequel il est possible d'effectuer des requêtes complexes.

Nous avons choisi de développer une application dans l'environnement *Electron*.

2 Environnement de travail : *Electron*

Pour obtenir un prototype convaincant, nous tenions à proposer un programme qui soit autonome (c'est-à-dire une application de bureau, plutôt qu'un script exécuté dans un

environnement de développement), multiplateforme (c'est-à-dire fonctionnant au moins sous *Windows* et *OSX*) et avec une interface utilisateur simple à mettre en place. L'environnement *Electron* remplit ces trois conditions.

Electron est un environnement (plus précisément un *cadriciel* – *framework* en Anglais) permettant « d'emballer » une application *web* (conçue pour s'exécuter dans un navigateur comme *Mozilla Firefox* ou *Google Chrome*) dans une application *de bureau* (c'est-à-dire un logiciel autonome).

Ce *framework* a donc deux avantages déterminant pour nous, qui sont les avantages des applications *web* : le programme développé est multiplateforme et la confection d'une interface utilisateur est grandement facilité, puisqu'on utilisera les langages dédiés aux interfaces d'application *web* (*HTML, JavaScript* et *CSS*). Enfin, il permet de livrer un logiciel autonome (l'utilisateur n'a besoin d'installer que le logiciel « final »).

L'application se présente donc ainsi :

Recherche
Recherche Sémantique
Recherche Littérale
Sonothèque
Créer
Supprimer
Ajouter...
En cours :
Sonothèque #1

Nom	Chemin	ID
eloignement pas off + porte lourde.L.wav	/Volumes/Sonotheque Raph 1/Autres films/ prise de son homm libre/deplacement int	1
montee escalier pas .L.wav	/Volumes/Sonotheque Raph 1/Autres films/ prise de son homm libre/deplacement int	2
pas calme int apart deplacement .L.wav	/Volumes/Sonotheque Raph 1/Autres films/ prise de son homm libre/deplacement int	3
pas int apart rapide saccades.L.wav	/Volumes/Sonotheque Raph 1/Autres films/ prise de son homm libre/deplacement int	10
passage pas int apart .L.wav	/Volumes/Sonotheque Raph 1/Autres films/ prise de son homm libre/deplacement int	11
presence humaine cage escalier apart .L.wav	/Volumes/Sonotheque Raph 1/Autres films/ prise de son homm libre/deplacement int /presence off	14
ecoulement eau discret+ fond robinet .L.wav	/Volumes/Sonotheque Raph 1/Autres films/ prise de son homm libre/eau	21
fond air cave + activitee diff verrou bois planche.L.wav	/Volumes/Sonotheque Raph 1/Autres films/ prise de son homm libre/fond air	30
fond air cave .L.wav	/Volumes/Sonotheque Raph 1/Autres films/ prise de son homm libre/fond air	31
voix cave hommes + buzz sub .L.wav	/Volumes/Sonotheque Raph 1/Autres films/ prise de son homm libre/humain int	34
voix hom comissariat off .L.wav	/Volumes/Sonotheque Raph 1/Autres films/ prise de son homm libre/humain int	35
voix homm calm discussion .L.wav	/Volumes/Sonotheque Raph 1/Autres films/ prise de son homm libre/humain int	36
activitee bar rangement .L.wav	/Volumes/Sonotheque Raph 1/Autres films/ prise de son homm libre/vaisselle	66

Proposition d'interface graphique utilisateur

On dispose donc des fonctionnalités élémentaires d'un gestionnaire de sonothèque. La partie supérieure comporte une barre de recherche pour recueillir les requêtes utilisateurs, les fonctions

pour créer/supprimer une sonothèque ou ajouter des sons dans une sonothèque existante et un menu déroulant pour consulter les différentes sonothèques indexées.

À titre d'expérience, on rend accessible sur l'interface graphique la mise en fonction de la recherche sémantique et de la recherche littérale, pour éventuellement comparer les deux.

Le panneau central sert à l'affichage des résultats de la recherche, ou, à défaut, à l'affichage de la totalité de la base de données.

Enfin, dans la partie inférieure, on trouve un lecteur audio rudimentaire pour écouter le son sélectionné.

2.a.α Gestion des bases de données dans l'application

Pour la gestion des bases de données, *Electron* dispose d'un moteur embarqué : *SQLite*. Ce moteur, directement intégré à l'application permet de créer et de modifier des bases de données à l'aide d'un « langage de requête structurée » (*Structured Query Language*, *SQL* en anglais). La base de données est modifiée à l'aide de requêtes suivant une syntaxe particulière.

Exemple :

Quand l'utilisateur clique sur « Créer », il renseigne un nom pour la base de données (disons `sonothèque_1`). Le programme exécute alors la requête

```
CREATE TABLE sonothèque_1 (
    Nom string,
    Chemin string
);
```

Créant ainsi un nouveau tableau contenant les colonnes « Nom » et « Chemin ».

Ensuite, quand l'utilisateur ajoute des sons à la sonothèque, via le bouton « Ajouter ... », le programme parcourt le répertoire choisi par l'utilisateur et, pour chaque son rencontré, effectue la requête

```
INSERT INTO sonothèque_1 (Nom, Chemin)
VALUES (nom_du_fichier, /chemin/du/fichier);
```

L'intérêt d'une telle base de données est de permettre la recherche à travers le contenu. Ainsi, pour un mot-clef entré par l'utilisateur dans la barre de recherche, le programme exécute la requête

```
SELECT * FROM sonothèque_1 WHERE Nom LIKE %motclef%;
```

Où `SELECT *` signifie que l'on souhaite afficher (`SELECT`) toutes les colonnes (`*`) du tableau en question (`sonothèque_1`).

(Ce cas correspond à une recherche « littérale » : on cherche tous les sons qui contiennent `motclef`).

Ayant donc à disposition un cadre de travail disposant des fonctionnalités essentielles, nous pouvons commencer à implémenter des blocs fonctionnels supplémentaires. Le premier concerne les différentes formes que peut prendre chaque mot.

3 Traitement des variétés morphologiques

En suivant le même schéma que dans la deuxième partie (*cf.* La diversité morphologique , p. 28), les abréviations et les formes fautives (qui sont une collection de cas particuliers) sont traités avant les formes fléchies (qui, elles, sont résolues de façon algorithmiques).

3.a Implémentation des cas particuliers

Il s'agit ici de « nettoyer » le nom du fichier de toutes ses abréviations (`sdb`, `pte`, `sil`, …) et de ses formes fautives (`piaf`, `oiso`, `chuchotis`, …) pour que la recherche ne s'y heurte plus.

Pour cela on renseigne, pour chaque mot susceptible d'être « nettoyé », un mot-clef de référence suivi de toutes les formes qu'il peut prendre.

Exemples :

- `quelques : qques, quelque, qlqs, qqs, qques`
- `fond d'air : fd a, f d'r, fd air`
- `voix : vx`
- `porte ouverte : PO`

Dans certains cas, on peut renseigner d'un seul coup plusieurs formes à l'aide d'une expression régulière. C'est important pour les mots qui seraient susceptibles de ne pas être rassemblés par la racinisation.

Exemples :

- `discussion : discut(.*)`
- `oiseaux : oisx, oiso(s)?, piaf(s)?`

Ces données sont rassemblées dans un tableau au format *CSV* (*comma separated values*, « valeurs séparées par des virgules »). Ce format est lu par la plupart des langages de programmation (dont *JavaScript*) ainsi que par les tableurs des logiciels de bureautiques (*LibreOffice* par exemple). Ainsi on dispose d'un format aussi simple à intégrer au programme qu'à modifier « à la main » pour ajouter des cas particuliers. Il faut toutefois s'assurer que chaque case du tableau est unique (sinon, un mot pourrait se trouver remplacé plusieurs fois).

On trouve une reproduction de ce tableau dans l'annexe C : Abréviations et formes fautives.[33]

On peut donc proposer une implémentation de la fonction réalisant le nettoyage du nom de fichier. En voici une version en *pseudo-code* :

```
table = open('./equivalence.csv')

def clean(name) : # Où name est une chaîne de caractères.
    word_list = name.split('\w') # Sépare name à chaque caractère non-aplhanumérique.
    out_list = [] # Liste recueillant les mots « nettoyés ».
    for word in word_list :
        for line in table : # Si le mot est une abréviation, on le remplace...
            if word in table : out_list.append(line[0]) # par le mot de la colonne de gauche.
    return out_list

# Exemple :
print clean('vx_appart PO')
>>['voix','appartement', 'porte ouverte']
```

Une fois le nom du fichier (ou la requête utilisateur) nettoyé de ses scories, celui-ci est prêt à subir la réduction de l'algorithme de racinisation.

3.b Implémentation de la racinisation

Comme expliqué dans la deuxième partie (*cf.* « La racinisation : un traitement des formes fléchies », p. 29), les algorithmes de racinisation utilisent un ensemble de règles pour réduire les mots à leur radical. Ces règles varient d'une langue à l'autre. La documentation de ces règles, pour le français, a déjà été effectuée ; ainsi il existe des modules pour les appliquer dans différents langages de programmation. Dans notre cas – le développement d'une application

33 Sur le dépôt, ce fichier se trouve à la racine : ./equivalence.csv

Electron, dont les fonctions sont écrites en *JavaScript* – nous avons choisi `jslingua`[34] [20]. Ce module propose une implémentation de l'algorithme de Porter pour l'anglais, l'arabe, le japonais et le français.

Ce module a, pour ce projet, l'avantage d'être directement intégrable à l'environnement de travail, et dispose de fonctionnalités étendues (comme la conjugaison, même pour les verbes irréguliers du troisième groupe, type « aller ») pour générer des formes fléchies à partir du radical. Cela permet par exemple de détecter les pluriels et les traiter différemment quand ils portent un sens différent au son.

Ainsi, la racinisation du nom d'un fichier, après le nettoyage s'effectue comme suit :

```
import JsLingua # Importation du module JsLingua.
french_stemmer = JsLingua.nserv('morpho', 'fra') # Chargement en mémoire du stemmer.

def stem_filename(clean_name) : # Où clean_name est la liste des mots « nettoyés ».
    out_name = []
    for word in clean_name :
        radical = french_stemmer.stem(word) # Racinisation du mot.
        out_name.append(radical) # Ajout du mot à la liste finale.
    return out_name

# Exemple :
print stem_filename(['fond', 'air', 'amphi', 'bois', 'legers', 'craquements']
>>['fond', 'air', 'amphi', 'bois', 'leger', 'craqu']
```

Il convient ensuite d'ajouter ce « nom épuré » à la base de données, en face du fichier auquel il correspond, en exécutant la requête *SQL* suivante :

```
UPDATE sonothèque SET Nom_épuré = stem_filename(clean_name) WHERE Nom = filename
```

On rassemble tous les traitements que subit un nom de fichier dans la fonction suivante :

```
def traitement(filename) :
    clean_name = clean(filename)
    stemmed_name = stem_filename(clean_name)

    database.execute('UPDATE database SET Nom_épuré='+stemmed_name+'WHERE Nom='+filename)
```

On dispose maintenant d'un programme pour obtenir un nom de fichier où chaque lexème ne dispose que d'un représentant unique. On peut donc procéder à l'analyse sémantique du nom du fichier, en commençant par la méthode « manuelle ».

34 Aries, Abdelkrim, *Javascrip libraries to process text*. https://github.com/kariminf/jslingua, 2016 [consulté le 21 avril 2018]

4 Représentation manuelle du contenu sémantique

Tout comme pour le fonctionnement général du programme, présenté en 1., on sépare la présentation entre l'indexation d'une part et la recherche d'autre part : il faut d'abord *ajouter* l'information du contenu sémantique à la base de données pour pouvoir ensuite la *chercher*.

4.a Représentation et stockage des données

Pour reprendre les trois « catégories » de lexèmes présentées « L'isotopie », p. 40, il faut pouvoir représenter :

- le graphe des lexèmes contextuels,
- les relations qu'entretiennent les lexèmes quantificateurs avec ce graphe,
- le contenu des lexèmes principaux sur ce graphe.

Comme pour les variétés morphologiques, cette représentation doit à la fois être intégrable facilement à l'environnement de travail (autrement dit, ce doit être un format lisible en *JavaScript*) et suffisamment lisible pour être éditable « à la main » au fur et à mesure que l'analyse se précise ou que des champs lexicaux sont ajoutés.

Le format adopté est le *JSON* (*JavaScript Object Notation*)[35][21]. C'est un format d'échange de données dérivé de la syntaxe du *JavaScript*. Il supporte deux structures de données :

- Les « objets » : une succession paires `entrée/valeur`, analogue à un dictionnaire.
- Les « listes » : une succession de `valeurs`.

Sachant qu'une `valeur` peut être une chaîne de caractères, un nombre, un objet, une liste ou `false/true/null`.

Exemple :

```
{
    ''id'' : ''nom'',
    ''type'' : ''fichier'',
    ''metadonnées'' :
    {
        ''extension'' : ''wav'',
        ''durée'' : 640,
        ...}}
```

35 On trouve le détail de ce format à l'adresse json.org.

Cette structure est interprétée directement en *JavaScript* avec la méthode `JSON.parse()` :

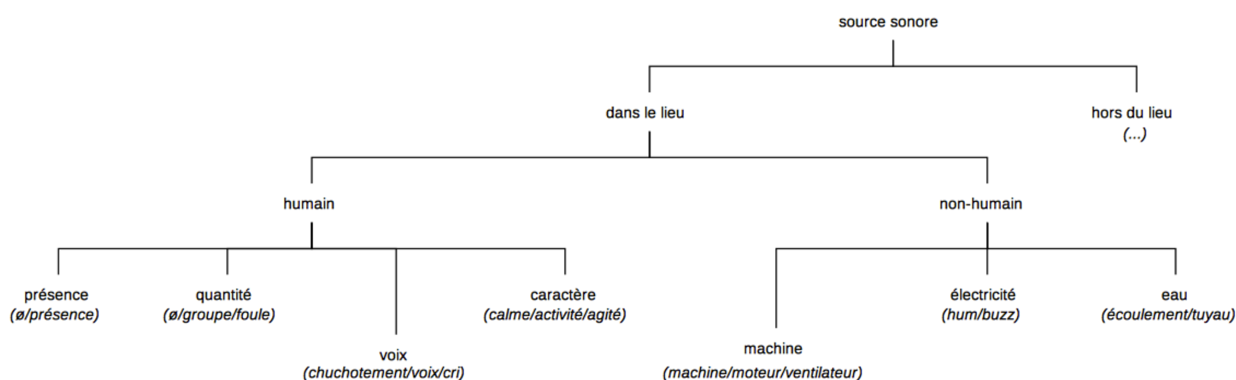
```
var json_string = '{"résultat":true, "occurences":42}';
objet = JSON.parse(json);

// objet.occurences -> 42
// objet.résultat -> true
```

Ainsi, obtient-on des représentations pour les trois catégories de lexèmes qui sont lisibles tant par le programme que par le développeur.

4.a.α Représentation des lexèmes contextuels

Si on reprend, en la précisant, une branche de l'arbre présenté dans la deuxième partie :



Précision d'une partie de l'arbre des lexèmes : « dans le lieu »

Une représentation en JSON de la branche « `dans le lieux` » se fait comme suit :

```
{
1 : "dans_le_lieu",
children" : [
    {
    2 : "humain",
    "children" : [
        { 3 : "presence", "values" : ["none", "presence"] },
        { 4 : "quantite", "values" : ["groupe", "foule"] },
        { 5 : "voix", "values" : ["chuchotement", "voix", "cri"], "children" : [4]},
        { 6 : "caractere", "values" : ["calme", "activite", "agite"] },
        ]
    },
    {
    7 : "non-humain",
    "children" : [
        { 8 : "machine", "values" : ["machine", "moteur", "ventilateur"] },
        { 9 : "electricite", "values" : ["hum", "buzz", "neon", "induction"] },
        { 10 : "eau", "values" : ["ecoulement", "tuyau"] }]}]}
```

Dans cet exemple, on a ajouté un lien entre la feuille `voix` et la feuille `quantité` en ajoutant `quantite` aux « enfants » de `voix`. On note d'ailleurs que ce lien est à sens unique, puisque `voix` n'est pas un « enfant » de `quantité`. C'est ainsi que l'on passe d'une structure d'arbre à une structure de graphe orienté.

Dans ce graphe, chaque nœud a un identifiant unique, un nombre, qui permet d'y faire référence. Cela nous permet de faire référence à des nœuds du graphe dans le contexte d'autres fichiers, comme c'est le cas pour les lexèmes quantificateurs.

4.a.β Représentation des lexèmes quantificateurs

Pour les quantificateurs (`aucun, beaucoup, sans, quelques, ...`) il faut renseigner trois informations :

- le quantificateur en question : `aucun, beaucoup,` *etc.*,
- l'opération qu'il réalise : décrémentation (-), incrémentation (+), suppression (`null`),
- des paires de lexèmes contextuels désignant le lexème auquel il est apposé, et le lexème qu'il modifie.

Ces deux derniers points sont rassemblés dans une même liste `[opération, lexème1, lexème2]`.

Exemple :

- `aucune voix` ne renseigne pas sur la valeur du lexème `5 [chuchotement, voix, cri]`, mais sur la valeur du lexème `4 [groupe, foule]`, en le mettant à zéro (`null`).
- `Aucune agitation` en revanche, renseigne directement la valeur du lexème `6 [calme, activite, agite]`, en la minorant (-).

Ainsi, on peut représenter le quantificateur `aucun` comme suit :

```
{"id" : "aucun", "application" : [[null,5,4], [-,6,6]] }
```

Ceci fait, on dispose de tous les éléments pour décrire le contexte sémique d'un lexème donné. Il faut maintenant renseigner, pour chaque lexème, son contenu sémantique *implicite* (comme décrit dans la section « Décomposition des lexèmes principaux », p. 44).

4.a.y Représentation du contenu des lexèmes principaux

Il s'agit de faire le lien entre le « nom épuré » décrit en 3 et le graphe des lexèmes contextuels. Pour chaque lexème documenté (dans ce travail, on s'intéresse à ceux décrivant les « ambiances d'intérieur »), on renseigne les nœuds du graphe sur lesquels il a une valeur, et on précise la valeur sur ce nœud.

Exemple :

Si on considère que `église <- grand, calme, réverbérant`, alors on peut représenter `église` comme suit :

```
{ "église" : [
    { id("taille") : "grand" },
    { id("caractère") : "calme" },
    { id("réverbération") : "réverbérant"}
    ]}
```

Où « `id("taille")` » correspond à l'identifiant unique du nœud `"taille"` tel qu'il est renseigné dans le graphe des lexèmes contextuels.

On remarquera qu'avec cette représentation, deux « synonymes » sont simplement deux lexèmes possédant la même décomposition sémantique. Ainsi n'est-il plus nécessaire de renseigner explicitement les synonymes (par exemple spécifier `église = chapelle`) et introduire une nuance entre deux termes consiste simplement à modifier leur décomposition sémantique (sans risque de « corrompre » le lien qui existerait entre eux).

Évidemment faut-il aussi représenter les lexèmes contextuels de cette façon, afin qu'ils soient interprétés au moment de la lecture du « nom épuré » (la représentation obtenu étant pour le moins tautologique).

Exemple :

```
{ "grand" : [ { id("taille") : "grand" } ] }
```

Enfin, pour analyser le « nom épuré » comme la combinaison des lexèmes qui le constituent, faut-il attribuer aux lexèmes des priorités.

Exemple :

Si on considère que `cantine <- grand, foule, activité, voix, présence`, et que `vide <- ∅ foule, calme, ∅ voix, ∅ présence`, a lors on traduit le fait que le son `cantine vide – quelques présences` a le contenu sémantique `grand, ∅ foule, calme, ∅ voix, présence`.

On voit alors[36], que les lexèmes principaux ont la priorité la plus basse, modifiés par des lexèmes de priorité intermédiaire. Enfin, les lexèmes contextuels appartenant au graphe ont la priorité la plus haute, puisque leur présence dans le nom du son renvoie directement au graphe.

Le fichier final, rendant compte du contexte sémique de chaque lexème a donc l'allure suivante :

```
{
    "priorité" : 1,
    "lexèmes" : [
        { "grand" : [ { id("taille") : "grand" } ] },
        ...
    ]
},

{
    "priorité" : 3,
    "lexèmes" : [
        { "vide" : [
            { id("quantité") : null },
            { id("caractère") : "calme" },
            { id("voix") : null },
            { id("présence") : null }
            ]},
        ...
    ]
},

{
    "priorité" : 3,
    "lexèmes" : [
        { "église" : [
            { id("taille") : "grand" },
            { id("caractère") : "calme" },
            { id("réverbération") : "réverbérant"}
            ]},
        ...
    ]
}
```

36 Comme souligné dans L'isotopie, p. 40.

Ainsi, pour construire le contenu sémantique d'un « nom épuré », suffit-il de lire la décomposition de chacun de ses lexèmes. En cas de conflit, on favorise le lexème de plus haute priorité.

On trouve la version « au jour de l'impression » de ce fichier dans l'annexe D : Graphe des lexèmes contextuels.[37]

En reprenant l'exemple précédent, on en déduit le contenu sémantique suivant, pour le son `cantine vide - quelques présences` :

```
[
    { id("taille") : "grand" },
    { id("quantité") : null },
    { id("caractère") : "calme" },
    { id("voix") : null },
    { id("présence") : "présence" }
]
```

Il convient enfin d'ajouter cette information à la base de données, dans la colonne « Contenu sémantique ». Si `contenu` est la liste recueillant le contenu sémantique du fichier de nom `filename`, alors il suffit d'appeler la fonction :

```
database.execute('UPDATE database SET Contenu_sémantique='+contenu+'WHERE Nom='+filename)
```

Ayant effectué toutes les étapes de la partie « indexation » du programme (*cf.* 1), on s'intéresse au fonctionnement de la recherche.

4.b Recherche

Comme l'indique le schéma fonctionnel en 1, la recherche se décompose en deux étapes :

- le traitement de la requête de l'utilisateur,
- la comparaison de cette requête « traitée » avec la base de données.

Le traitement de la requête est le même que celui des noms de fichier : la requête utilisateur (une suite de mots-clefs séparés par des caractères d'espacement) est séparée en une liste dont les éléments sont nettoyés et racinisés. On obtient une « requête épurée » (à l'instar des « noms épurés ») dont on construit le contenu sémantique selon le procédé décrit en 4.

37 Sur le dépôt, ce fichier se trouve à la racine : ./lexeme_tree.js

La question restante est celle de la comparaison avec la base de données. L'approche élémentaire consiste à sélectionner les fichiers dont le contenu sémantique est identique à celui de la requête. En nommant `contenu_requête` le contenu sémantique de la requête, il suffit d'exécuter la requête :

```
database.execute('SELECT * FROM database WHERE Contenu_sémantique='+contenu_requête)
```

Représenter aussi précisément le contenu sémantique des sons permet d'envisager une recherche « par similitude ». Trois méthodes supplémentaires pour sélectionner les résultats sont alors possibles :

- Sélectionner les sons dont le contenu sémantique est **inclus** dans celui de la requête. Ce sont les sons dont le contenu est « moins précis » que celui de la requête – ils rassemblent *a priori* des cas plus généraux.
- Sélectionner es sons qui contiennent **au moins** le contenu sémantique de la requête. Ces sons correspondent à des cas plus précis que ceux représentés par la requête utilisateur.[38]
- Sélectionner les sons qui ont en commun une certaine proportion de leur contenu avec la requête, l'enjeu étant de mesurer cette proportion.

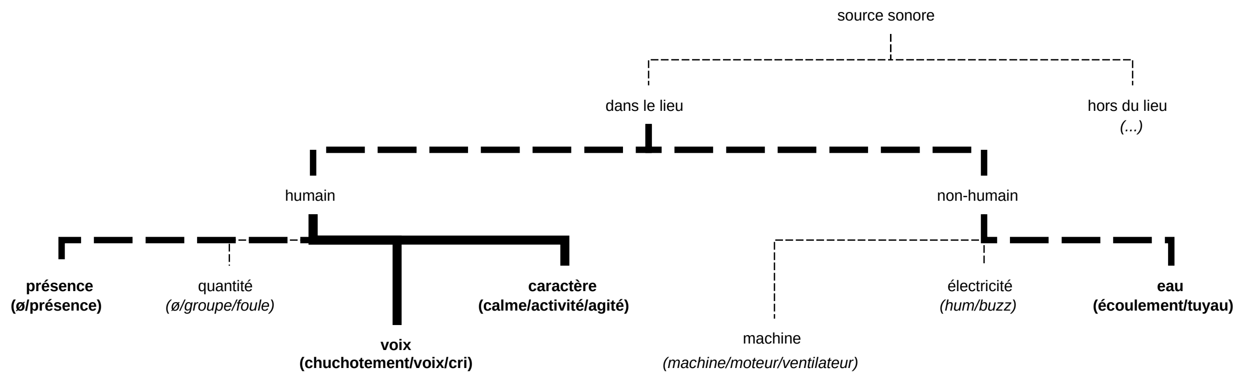
Pour mesurer la « distance » entre deux contenus sémantiques (entre la requête et un nom de fichier, ou bien entre deux noms de fichiers), on s'appuie sur l'organisation en graphe des lexèmes contextuels.

En effet, on peut par exemple définir la distance entre deux lexèmes comme *le plus court chemin* entre ces deux nœuds.

Exemple :

---

38 C'est d'ailleurs le mode de fonctionnement standard des moteurs de recherche littéraux, où `passage voiture` renvoie tous les sons **contenant** `passage voiture` – dont `passage rapide voiture de sport`.



En gras continu : le chemin de « voix » à « caractère ». En gras discontinu : le chemin de « présence » à « eau »

Entre `voix` et `caractère`, on compte un seul nœud, on a donc : $dist(voix, caractère)=1$. De la même façon, on compte trois nœuds entre `présence` et `eau`, on a donc $dist(présence, eau)=3>dist(voix, caractère)$.

Cependant, on peut rencontrer sur un même nœud des valeurs opposées (comme `chuchotement` et `cri` par exemple). C'est pourquoi, quand c'est possible, ces valeurs sont ordonnées. Et on définit la distance entre ces deux valeurs comme étant maximale, c'est-à-dire $dist(\text{“chuchotement”}, \text{“cri”})=+\infty$.[39]

Ayant défini la distance entre deux lexèmes, on définit ensuite la distance entre un lexème l et une liste de lexèmes $\Lambda=(\lambda_{1,} \lambda_{2,} \lambda_{3,} ...)$ (donc une décomposition sémantique) comme : $dist(l, \Lambda)=\min_{\lambda \in \Lambda}[dist(l, \lambda)]$. Ainsi la distance entre deux décompositions sémantiques Λ_1 et Λ_2 est-elle définie comme la somme des distances entre leurs lexèmes respectifs : $dist(\Lambda_{1,} \Lambda_2)=\sum_{l \in \Lambda_1} dist(l, \Lambda_2)$.

Disposant de cette mesure de similitude entre deux décompositions sémantiques, on l'applique à la recherche sur la base de données, en fixant un seuil au-dessous duquel on sélectionne les résultats :

```
database.execute('SELECT * FROM database WHERE'+dist(contenu_requête,contenu_sémantique)<seuil)
```

---

39 Pour économiser du temps de calcul au moment de la recherche, on remarquera que l'on peut calculer l'ensemble de ces distances entre lexèmes « à l'avance », à partir du graphe des lexèmes contextuel et les renseigner dans un tableau.

En étudiant comment s'effectue l'indexation des fichiers dans la base de données – à l'aide du gestionnaire *SQLite* -, comment est représentée et stockée la structure qui lie les lexèmes entre eux et enfin comment se construit le contenu sémantique d'un nom de fichier à partir de la combinaison des lexèmes qui le constituent, nous avons pu implémenter une méthode de recherche des sons par similitude du contenu sémantique, plutôt que par correspondance de chaîne de caractères.

Toutefois, la documentation des lexèmes qui constituent une sonothèque et leur mise en relation est un travail fastidieux[40]. C'est pourquoi, dans le cadre de ce projet, nous avons restreint l'étude au seul cas des ambiances d'intérieur.

Cependant, il était nécessaire d'envisager une solution qui soit applicable à (virtuellement) tous les sons, quelque soit leur catégorie sémique. C'est pourquoi nous présentons dans la section suivante une implémentation de l'*Analyse Sémantique Latente*.

5 Implémentation de l'*ASL*

Au moment de l'écriture de ces lignes, l'*ASL* n'est pas encore implémentée dans la version disponible de l'application. Nous présenterons donc ici un prototype développé sous la forme d'un script dans un autre langage (*Python*). Ce script ne dispose pas des fonctionnalités élémentaires du gestionnaires de sonothèques – il n'a, notamment, pas d'interface graphique. L'objectif est de démontrer dans ces lignes la faisabilité de l'implémentation, l'intégration à l'application « mère » étant une question d'ingénierie logicielle.

On rappelle que les résultats de l'*ASL* sont relatifs au corpus de données choisi. Aussi convient-il de présenter plus précisément le corpus sélectionné.

5.a Corpus choisi : les sonothèques commerciales

Nous avons choisi « d'entraîner » l'algorithme d'analyse sémantique sur des sonothèques « commerciales ». Pour justifier ce choix, examinons d'abord les conditions nécessaires à la mise en application de l'*ASL*.

40 Comme mentionné dans la section « L'isotopie », p. 40.

L'objectif premier était d'analyser des sons appartenant à un maximum de catégories sémiques différentes. Il fallait donc trouver un corpus réunissant des sons « de toutes sortes » et en grande quantité.

S'attaquer à un spectre de sons aussi large à une conséquence pratique : s'il était réaliste – dans le temps de ce mémoire – de recenser les abréviations rencontrées dans le champ (restreint) des « ambiance d'intérieur »), appliquer cette tâche à « tous » les sons possibles semblait illusoire. Aussi fallait-il disposer d'un ensemble dont les noms étaient au maximum dépourvus des scories lexicales et orthographiques.

Dès lors, appliquer l'*ASL* aux sonothèques « personnelles » de monteurs son semblait être une opération périlleuse, ne serait-ce que par l'impossibilité de traiter toutes les abréviations.

En revanche, les sonothèques éditées pour une diffusion commerciale sont généralement vierges d'abréviations. En effet, pour toucher un public le plus large possible, les éditeurs limitent au maximum les abréviations et autres expressions issues de jargons particuliers, puisqu'ils peuvent être source d'ambiguité. De plus, les éditeurs ne sont pas soumis aux mêmes contraintes temporelles que les monteurs au moment du dérushage. On a donc le plus souvent affaire à des sonothèques nommées avec rigueur (les mêmes mots sont toujours utilisés dans les mêmes cas) et disposant généralement de métadonnées supplémentaires (le plus souvent sont renseignés les champs « catégorie » et « description »).

Ces sonothèques commerciales offrent en outre un volume de données important et « prêtes à l'emploi ». En effet, la plupart des éditeurs mettent gratuitement à disposition pour chaque sonothèque le tableau, au format *CSV* ou *XLSX*[41], listant les sons et leurs métadonnées.

La seule contrainte concernant ces données est qu'elles ne sont disponibles qu'en langue anglaise.

Nous avons donc, pour entraîner l'algorithme, téléchargé tous les tableaux disponibles chez les éditeurs suivants : *SoundIdeas*, *BOOM Library* et *Pro Sound Effects*.

5.b Implémentation de l'*ASL*

Avant de passer par l'analyse sémantique latente, les données doivent, comme dans le cas de l'analyse manuelle, être racinisées. On utilise pour ce prototype un portage *Python* de

41 Format propriétaire du logiciel *Microsoft Excel*.

l'algorithme de Porter (pour l'anglais), écrit par Vivake Gupta[42][22]. On trouve l'intégralité du code source en annexe[43].

De plus, comme mentionné dans la deuxième partie, il convient de retirer les « mots-vides » avant de passer à l'analyse sémantique. Pour rappel, les mots-vides sont des mots à forte occurrence mais portant peu de sens sur le sujet dont il est question (les pronoms par exemple). On trouve de nombreuse listes de mots-vides pour l'anglais et le français.

> **Remarque :** Les listes de mots-vides contiennent la plupart des lexèmes quantificateurs (`aucun, sans, plusieurs, quelques, ...`). S'ils font bien sens du point de vue de la description des sons, il est toutefois cohérent de ne pas en tenir compte dans l'*ASL*, puisqu'ils font sens « indépendamment » du sujet. Par exemple, mesurer un lien entre `aucun` et `voiture` puis entre `aucun` et `voix` produira un lien entre `voiture` et `voix` difficile à interpréter (en l'occurrence : ils seraient liés par leur faculté à être absents…).

Par souci de cohérence avec le *stemmer* utilisé (celui de Porter), on utilisera la liste de mots-vides proposée par Martin Porter[44][23]. On reproduit cette liste en annexe[45].

Comme décrit la deuxième partie[46], il nous faut maintenant un programme pour calculer la *TF-IDF* de chaque mot du corpus et construire la matrice des occurrences puis effectuer la décomposition en valeurs singulières.

Pour cela, nous allons utiliser le module `semanticpy` de Joseph Wilk[47][24], qui intègre lui-même le *stemmer* de Porter mentionné plus haut[48]([25], [26]).

Du point de vue de notre application, le module `semanticpy` expose une classe d'objets, `VectorSpace`, qui se construit de la façon suivante :

---

42 Gupta, Vivake, *Porter Stemming Algorithm*. https://tartarus.org/martin/PorterStemmer/python.txt, 2008 [consulté le 21 avril 2018]
43 *cf.* annexe E : *Implémentation de l'algorithme de Porter (Python)*
44 Porter, Martin, *An English stop word list*. http://snowball.tartarus.org/algorithms/english/stop.txt, /ca 2000 [consulté le 21 avril 2018]
45 *cf.* annexe F : Mots-vides pour l'anglais (stopwords)
46 *cf.* « Analyse sémantique latente », p. 45
47 Wilk, Joseph, *A collection of semantic functions for python*. https://github.com/josephwilk/semanticpy, 2008 [consulté le 21 avril 2018]
48 On trouve deux articles de vulgarisation de l'*ASL* consultables sur le blog de Joseph Wilk, blog.josephwilk.net/project.

```
from semanticpy.vector_space import VectorSpace
vector_space = VectorSpace( [document1, document2, ...] )
```

Où `document1`, `document2`, … correspondent à des chaînes de caractères. Dans notre cas, chaque `document` correspond à un nom de fichier issu des sonothèques commerciales utilisées (tel que décrit en 5.a).

Concrètement, à sa construction, l'objet `VectoSpace` effectue les opérations successives de l'*ASL* :

1. formatage des données, suppression des mots-vides (*stopwords*), racinisation (*stemming*) ;
2. construction de la matrice des occurrences (calcul de la *TF-IDF* de chaque terme) ;
3. décomposition de la matrice en valeurs singulières et réduction du rang de la matrice.

5.b.α Formatage des données avec `semanticpy`

En interne (c'est-à-dire au sein du module `semanticpy`), le formatage est réalisé par la classe d'objet `Parser`[49]. Cet objet fait appel à un fichier contenant la liste des mots-vides (liste que nous avons remplacée par la liste proposée par Martin Porter et présente annexe F).

La méthode `Parser._tokenise(string)` convertit toutes les lettres en caractère minuscule et supprime les marques de ponctuation puis racinise tous les mots contenus dans `string` à l'aide de la méthode `Parser.stemmer.stem(word)`.[50]

Ensuite, tous les mots vides sont supprimés à l'aide de la méthode `Parser._remove_stop_words(list)`. Enfin, les doublons sont supprimés de la liste du vocabulaire.

On peut résumer cette étape de formatage par le code suivant :[51]

```
def nettoyage(document_list) :
    vocabulary_string = " ".join(document_list) #Tous les documents sont joints dans une seule
                                                #chaîne de caractères.

    #-- Equivalent de la méthode Parser._tokenise()--#
```

49 Classe d'objet décrite dans ./semanticpy/parser.py.
50 Cette méthode fait précisément appel au *stemmer* reproduit en annexe E.
51 Si l'on rassemble ici toutes les étapes dans un même bloc fonctionnel, elles sont toutefois réparties dans des fichiers différents du code source d'origine. On espère que ce code évitera au lecteur de reconstituer « à la main » le trajet de l'information à partir du code source du module.

```
    vocabulary_string = Parser._clean(vocabulary_string) #Minuscules et ponctuation
    words = vocabulary_string.split(" ") #Transforme vocabulary_string en une liste de mots
    tokenised_vocab_list = [Parser.stemmer.stem(word) for word in words] #Stem les mots de la liste

    #-- Equivalent de Parser._remove_stop_words() --#
    stop_file = open("./english_stop_words.txt", 'r') #Ouvre le fichier contenant les mots-vides
    stopwords = stop_file.read().split() #Convertit le fichier en liste de mots.

    clean_word_list = [word for word in tokenised_vocab_list if word not in stopwords]
    #Ainsi clean_word_list contient tous les mots racinisés qui ne sont pas des mots-vides
    unique_vocabulary_list = set((item for item in clean_word_list)) #Supprime les doublons

    return unique_vocabulary_list
```

`unique_vocabulary_list` correspond donc au « vecteur des termes » de l'analyse, L'index de chaque terme correspondant à sa position dans la liste. On construit ensuite la matrice *TF-IDF* du corpus.

*5.b.β Exécution de l'*ASL *avec* `semanticpy`

Pour construire cette matrice, on fait cette fois appel à des méthodes de la classe `VectorSpace`. La méthode `_build()` commence par construire la matrice des occurrences, autrement dit : on compte le nombre d'apparition de chaque mot dans chaque document, avant d'effectuer la pondération de « fréquence inverse dans le document » (autrement dit : si un mot est trop présent, c'est qu'il a peu de sens).

On peut résumer la construction de la matrice des occurrences par le code suivant :

```
def frequency_matrix(document_list) :

    #On assemble les vecteurs des occurrences pour chaque document
    matrix = [frequency_vector(document) for document in document_list]

    return matrix

#-- Où frequency_vector(document) est défini comme suit : --#

def frequency_vector(document) :
    vector = [0]*len(unique_vocabulary_list) #On crée un vecteur de la taille du dictionnaire
    #On effectue le nettoyage décrit plus haut et on racinise chaque mot du document
    word_list = Parser._remove_stop_words(Parser._tokenise(document.split(" ")))

    for word in word_list :
        #On incrémente le nombre d'occurrences de chaque mot.
        #On notera que l'ordre des mots est le même dans vector et unique_vocabulary_list.
        vector[unique_vocabulary_list.index(word)] += 1
    return vector
```

Une fois la matrice obtenue, on y applique deux transformations successives : une pour obtenir la matrice *TF-IDF*, l'autre pour effectuer la décomposition en valeur singulière et la réduction du rang.

Ces deux transformations sont respectivement décrites dans `./semanticpy/transform/tfidf.py` et `./semanticpy/transform/lsa.py`. On les reproduit en annexe G : « Implémentation Python de TF-IDF et ASL ».[52]

Avec les noms de variable utilisés plus haut, cela se traduit par la séquence :

```
matrix = frequency_matrix(document_list)
matrix = tfidf(matrix)
matrix = lsa(matrix)
```

On obtient un outil capable de créer la matrice mesurant les liens entre les concepts et les documents. Cette matrice est accessible *via* un attribut de la classe `VectorSpace` :

```
#Construction de l'espace des concepts
vector_space = VectorSpace( [document1, document2, ...] )

#On accède à la matrice par l'attribut collection_of_document_term_vectors
lsa_matrix = vector_space.collection_of_document_term_vectors
```

Ayant construit « l'espace des concepts », on peut maintenant effectuer une recherche sur le corpus de données, grâce au module `semanticpy` qui implémente deux méthodes de la classe `VectorSpace`.

5.b.y Recherche de sons dans l'espace des concepts avec `semanticpy`

On a vu dans la section « Mesure des similitudes » p. 52, que la construction de la matrice représentant l'espace des concepts permet de mesurer la similitude entre deux documents (représentés par leurs vecteurs-termes). Ainsi peut-on, de la même façon qu'en 4.b, envisager deux méthodes de recherche :

- Une recherche par requête. On traite la requête comme un document et on cherche tous les documents « similaires » à cette requête.
- Une recherche par similitude « d'un son à l'autre ». On cherche tous les sons « qui ressemblent à un son donné » (vu de la matrice des concepts).[53]

---

52 À toutes fins utiles : cette double opération est résumée dans le code source de la fonction VectorSpace._build() par la ligne :

```
matrix = reduce(lambda matrix,transform: transform(matrix).transform(), transforms, matrix)
```

53 On rappelle ici que la « similitude » entre deux vecteurs-termes est mesurée par le cosinus entre ces deux vecteurs.

En terme d'implémentation, ces deux méthodes s'implémentent de façon identique, à ceci près que, dans le second cas, on profite du fait que le document est déjà intégré à la matrice.

Concrètement, `semanticpy` propose déjà ces deux méthodes pour la classe `VectorSpace` :

- `search(query)` renvoie tous les documents, classés par ordre de similitude avec la liste de mots-clefs `query`.
- `related(document_id)` renvoie tous les documents, classés par ordre de similitude avec le document du corpus d'index `document_id`.

On peut résumer ces deux fonctions avec le code suivant :

```
def search(query) :

    query_vector= frequency_vector(query) #Création du vecteur-terme correspondant à la requête

    #Création de la liste des similitude : on la calcule pour chaque document du corpus
    resultats = [ cosine(query_vector, document_vector) for document_vector in
collection_of_document_term_vectors]

    #On renvoie la liste des résultats, classée par ordre décroissant de similitude
    return resultat.sort( reverse=True )

def related(document_id) :

    #Le vecteur-terme est directement une ligne de la matrice !
    query_vector = collection_of_document_term_vectors[document_id]

    resultats = [ cosine(query_vector, document_vector) for document_vector in
collection_of_document_term_vectors]

    return resultat.sort( reverse=True )
```

Avec `semanticpy`, nous avons tous les outils nécessaires pour effectuer l'*analyse sémantique latente* de l'ensemble des sonothèques commerciales : les données sont d'abord formatées, puis racinisées avant de construire la « matrice des concepts » par *TF-IDF* et décomposition en valeurs singulières.

Le module nous permet ensuite d'effectuer des recherches sur l'ensemble des sonothèques, soit par une requête, soit par comparaison entre le nom d'un son et les autres.

En présentant l'implémentation de deux méthodes de représentation du contenu sémantique des noms de fichier : celle issue de la décomposition sémantique « manuelle » d'abord, appliquées aux ambiances d'intérieur ; puis la représentation des liens entre les sons par analyse sémantique latente, appliquée à un corpus de sonothèques du commerce, nous avons montré

que chacune de ces méthodes permet d’effectuer soit une recherche par mots-clefs, soit une recherche par similitudes entre les sons de la sonothèque.

Conclusion

Le point de départ de ce travail est le constat d'une frustration de monteurs son quant à l'utilisation des moteurs de recherche pour sonothèques, contraignant les monteurs à dépenser une énergie importante à la mise au point d'une façon de nommer les sons pour que le moteur « comprenne » leur requête, ou bien les conduisant à abandonner simplement l'utilisation du moteur pour ne plus faire confiance qu'à leur mémoire.

Observant et analysant le lien entre le monteur et l'outil, nous avons constaté que cette frustration n'est pas directement liée à des défauts intrinsèques du moteur de recherche, mais à une interaction forte entre le fonctionnement du moteur et le comportement du monteur : le monteur rencontre un obstacle et tente de le contourner en adaptant son utilisation (il essaie de « faire comprendre » au moteur ce qu'il cherche). Ce faisant, le monteur est amené à adopter des comportements contre-intuitifs, susceptibles de rendre la recherche très fastidieuse – au lieu de laisser place à l'association créative d'idées et de sons. En observant des monteurs son au travail, nous avons constaté que ces difficultés avaient pour cause possible l'absence de représentation du contenu sémantique des sons à travers leurs noms, forçant le monteur à l'expliciter par des noms complexes ou par de nombreuses recherches successives.

La représentation de ces liens nécessite au préalable de résoudre la question des abréviations, des erreurs d'orthographe et des déclinaisons, pour réduire chaque unité de sens à un unique représentant lexical. Cette question est réglée par une documentation la plus exhaustive possible des abréviations et autres formes fautives, puis par l'application de l'algorithme de Porter, réduisant chaque mot à son radical.

Une fois les unités sémantiques réduites à un seul représentant, nous avons exploré deux méthodes de construction des liens sémantiques.

La première consiste à organiser les catégories sémiques sur un graphe. Cela revient à se demander, pour une classe donnée de sons (dans notre cas, les « ambiances d'intérieur »), quelles sont les dimensions qui peuvent caractériser cette classe de sons. Ensuite, en effectuant la décomposition sémique des lexèmes de cette classe, on obtient, par combinaison, la projection

du nom du son sur le graphe reliant les lexèmes entre eux, mesurant ainsi la distance d'un son à l'autre.

La seconde approche est automatique. C'est l'*analyse sémantique latente*. Elle consiste à mesurer l'occurrence des termes dans un ensemble de documents (dans notre cas, les noms de sons issus d'un corpus de sonothèques du commerce). Par transformation algébrique, on obtient alors une matrice permettant de mesurer la similitude entre deux noms de fichier.

Ces deux méthodes permettent, l'une comme l'autre, d'effectuer la recherche de deux façons : soit en cherchant les sons « les plus proches » de la requête utilisateur, soit en cherchant tous les sons « en lien avec un son donné ».

L'étude de ces deux méthodes nous a conduit à développer un gestionnaire de sonothèques dont le moteur d'indexation et de recherche tient compte de la représentation de ces liens sémantiques.

S'agissant du lien entre ce travail et la pratique du montage son, il convient de noter qu'il ne s'agit pas de rationaliser le processus de recherche : il ne dit pas quels sons « vont ensemble ». En revanche, il s'agit de tirer parti des informations du langage utilisé pour nommer les sons pour suggérer l'existence de liens entre les sons – ce que fait déjà *en soi* le nom d'un son. Ainsi, ce travail ne propose en aucun cas une méthode pour trouver « le bon son » pour une séquence donnée d'un film. En revanche, nous pensons qu'en fluidifiant le processus de recherche, nous laissons plus de place au mouvement créatif. Avec un outil qui ne demande aucune opération supplémentaire au monteur, libre à lui de se perdre dans sa sonothèque. Nous espérons même que cet outil puisse susciter des associations d'idées en rapprochant des sons autrement éloignés.

Au sujet des associations d'idées, un prolongement naturel de ce travail explorerait les liens que l'on peut tirer entre les sons à partir de leur utilisation réalisée (dans une bande son par exemple). Quelle conclusion peut-on tirer de la co-utilisation de deux sons dans une même séquence, dans un même film ? Quelles informations résident dans la recherche successive de plusieurs mot-clefs ? Dans le temps passé à écouter certains sons plutôt que d'autres ?

Ces réflexions pourraient mener à la conception d'un moteur de recherche dont le profil s'adapterait progressivement à son utilisateur – s'approchant en cela des méthodes de filtrage collaboratif des plateformes de diffusion de musique en ligne. Il faudrait cependant être vigilant quand au temps nécessaire à la convergence d'un tel profil et la question éthique posée par la

collecte des données d'utilisation des sons d'un monteur. Enfin, il faudrait évidemment s'assurer qu'un tel outil propose bien un agrandissement du territoire créatif du monteur, en suscitant des associations de sons et d'idées imprévues, plutôt qu'en encourageant des routines de montage.

Bibliographie

[1] : Cacheux, Simon, *Méthode de recherche et de classification des sons en sonothèque*. Mémoire de master. Paris : École Nationale Supérieure Louis-Lumière, 2008, 87 p.

[2] : Virostek, Paul, *An Introduction to Sound FX Metadata Apps 2 - Comparing Apps*. https://creativefieldrecording.com/2014/06/17/an-introduction-to-sound-fx-metadata-apps-2-comparing-apps/, 2017 [consulté le 21 avril 2018]

[3] : Rijsbergen, Cornelis J. et al., *New models in probabilistic information retrieval*. London : British Library, 1980, pp. 98-106

[4] : Paice, Chris, Method for evaluation of stemming algorithms based on error counting. *Journal of the American Society for Information Science*. Vol. 47, 1996, pp. 632–349

[5] : Paternostre, M. et al., *Carry, un algorithme de désuffixation pour le français*. Rapport technique du projet Galilei, Institut Paul Otlet, 2002, 15 p.

[6] : Porter, Martin, *French stemming algorithm*. http://snowball.tartarus.org/algorithms/french/stemmer.html, 2006 [consulté le 21 avril 2018]

[7] : Boulton, R., *PyStemmer 1.3.0*. https://pypi.org/project/PyStemmer/1.3.0/, 2013 [consulté le 21 avril 2018]

[8] : Opillard, Thierry, La dégradation Cambridge : mème pas vrai ?. *Les Actes de Lecture*. Vol. n. 102, 2008, pp. 23-28

[9] : Greimas, Algirdas Julien, *Sémantique structurale*. Paris : Presses Universitaires de France, 1966, 294 p.

[10] : Bergson, Henri, *Matière et mémoire*. Paris : Flammarion, 2012, 349 p.

[11] : Sutton, Richard et Barto Andrew, *Reinforcement Learning: An Introduction*. Cambridge : The MIT Press, 2012, 334 p.

[12] : Princeton University, *WordNet | A Lexical Database for English*. https://wordnet.princeton.edu/, 1998 [consulté le 21 avril 2018]

[13] : Sagot Benoît et Fišer Darja, *Building a free French wordnet from multilingual resources*. May 2008, Marrakech, Morroco.

[14] : Landauer, Thomas et al., An Introduction to Latent Semantic Analysis*. Discourse Processes*. Vol. 25, 1998, pp. 259-284

[15] : Baeza-Yaetes, Ricardo et Bibeiro-Neto Berthier, *Modern Information Retrieval*. New York : ACM Press, 1999, 513 p.

[16] : Savoy, Jacques, *IR Multilingual Resources at UniNE - English stopwords*. http://members.unine.ch/jacques.savoy/clef/englishST.txt, 2016 [consulté le 21 avril 2018]

[17] : Savoy, Jacques, *IR Multilingual Resources at UniNE - French stopwords*. http://members.unine.ch/jacques.savoy/clef/frenchST.txt, 2016 [consulté le 21 avril 2018]

[18] : Auger, Alex-Adrien et Serror, Théo, *ElectronSQL*. https://github.com/alexadrien/ElectronSQL, 2018 [consulté le 21 avril 2018]

[19] : Serror, Théo, *ressources_PPM*. https://github.com/C-rror/ressources_PPM, 2018 [consulté le 21 avril 2018]

[20] : Aries, Abdelkrim, *Javascrip libraries to process text*. https://github.com/kariminf/jslingua, 2016 [consulté le 21 avril 2018]

[21] : ECMA International, *JavaScript Object Notation*. https://json.org, 2017 [consulté le 21 avril 2018]

[22] : Gupta, Vivake, *Porter Stemming Algorithm*. https://tartarus.org/martin/PorterStemmer/python.txt, 2008 [consulté le 21 avril 2018]

[23] : Porter, Martin, *An English stop word list*. http://snowball.tartarus.org/algorithms/english/stop.txt, /ca 2000 [consulté le 21 avril 2018]

[24] : Wilk, Joseph, *A collection of semantic functions for python*. https://github.com/josephwilk/semanticpy, 2008 [consulté le 21 avril 2018]

[25] : Wilk, Josep, *Building a Vector Space Search Engine in Python*. http://blog.josephwilk.net/projects/building-a-vector-space-search-engine-in-python.html, 2007 [consulté le 21 avril 2018]

[26] : Wilk, Joseph, *Latent Semantic Analysis in Python*. http://blog.josephwilk.net/projects/latent-semantic-analysis-in-python.html, 2007 [consulté le 21 avril 2018]

Annexes

A Note sur les expressions régulières

Le terme *expression régulière* désigne une chaîne de caractères, suivant un formalisme particulier, servant à représenter un ensemble plus large de chaînes de caractères. L'utilisation des expressions régulières permet de décrire avec concision des opérations complexes.

Exemple :

- `p(or)?te` correspond à la fois aux deux chaines `pte` et `porte`.
- `A*` correspond aux chaînes `A, AA, AAA, AAAA, ...`
- `lég(er|ère)s?` correspond aux chaînes `léger, légère, légers` et `légères`.

Il existe plusieurs jeux de symboles, variants d'une implémentation à une autre. On note ici les symboles les plus courants :

- `?` : correspond à zéro ou une occurrence. `ab? -> a, ab.`
- `*` : correspond à « au moins zéro » occurrence. `A* -> Ø, A, AA, AAA, ...`
- `+` : correspond à « au moins une » occurrence. `A+ -> A, AA, AAA, ...`
- `.` : correspond à exactement un caractère. `a. -> aa, ab, ac, a1, ...`
- $str_1|str_2$; correspond à l'opérateur « OU ». `a(a|b) -> aa, ab.`
- `[liste]` : choisit un des caractères parmi la liste. `[acd] -> a, c, d.`
- `[^liste]` : exclut les caractères de la liste. `[^acd] -> b, e, f, ...`
- `(expr)`: permet d'appliquer un opérateur à `expr` toute entière. `(ab)? -> Ø, ab.`
- `expr{n}` : correspond à exactement `n` occurrences de `expr`.
- `expr{n,m}` : correspond à au moins `n` occurrences, au plus `m` occurrences, de `expr`.
- `expr{n,}` : correspond à au moins `n` occurrences de `expr`.
- `^expr` : chercher `expr` en début de ligne.
- `expr$` : chercher `expr` en fin de ligne.

Raccourcis :

Certaines expressions sont interprétées comme des raccourcis pour des expressions plus longues (ces raccourcis peuvent varier d'une implémentation à l'autre).

- `[A-Za-z]` : un caractère parmi ceux compris entre `A` et `Z` ou `a` et `z`.
- `\w` : un caractère alphanumérique. Équivalent à `[A-Za-z0-9_]`.
- `\W` : un caractère non alphanumérique. Équivalent à `[^A-Za-z0-9_]`.
- `\b` : position de début ou fin de mot.
- `\B` : position ne correspondant ni à un début ni à une fin de mot.
- `\s` : un caractère d'espacement.

On trouve de nombreux modules en ligne pour vérifier le comportement d'expressions régulières.[54]

54 Notamment : https://www.regexpal.com/

B Éléments de sémantique

Lexème : (aussi appelé *unité lexicale*) désigne le constituant lexical d'une unité sémantique autonome (appelée parfois *lemme*). On peut, schématiquement identifier *lexème/lemme* à *signifiant/signifié*.

Un lexème peut avoir plusieurs *représentants*. **Exemple :** `**léger, légère**` et `légèrement` sont trois représentants du même lexème. On choisit généralement le radical comme représentant d'un lexème (`léger` dans l'exemple précédent).

Sème : c'est l'unité minimale de signification, indépendamment de sa représentation. Les sèmes peuvent s'assembler en *sémèmes*, pour former une unité lexicale.

Exemple :

- `fauteuil = pour s'asseoir + avec bras + avec dossiers`
- `chaise = pour s'asseoir + avec dossier (+ sans bras)`
- `tabouret = pour s'asseoir (+ sans dossier + sans bras)`

Dans cet exemple, on a plusieurs *sémèmes* décomposés comme des sommes de *sèmes.*

Noyau et Contexte sémiques : Pour un sémème donné, on sépare son contenu sémique en deux. D'abord, un minimum sémique permanent, comme un invariant, que l'on nomme *noyau sémique* (noté *Ns*). Ensuite, une partie recevant les variations de sens provenant du contexte, que l'on nomme *contexte sémique* (noté *Cs*).

Exemple : avec le sémème `tête`. On peut remarquer que, quelque soit son contexte d'utilisation, il contient le sème `extrémité`.

- `Une tête couronnée,`
- `la tête d'un arbre, ...`

Tandis que le sème `partie du corps humain` n'est pas présent dans toutes les manifestations de ce sémème : il l'est dans `tête couronnée` et `tête de mort`, mais pas dans `la tête d'un arbre` et `prendre la tête (d'un mouvement)`.

Greimas remarque, dans *Sémantique structurale*, que le noyau sémique est le plus souvent implicite (sinon, il y a pléonasme), tandis que le contexte sémique vient le préciser (*i.e.* ajouter une catégorie sémique) ou l'infléchir (*i.e.* changer sa valeur sur une catégorie sémique présupposée).

C Abréviations et formes fautives

On reproduit ici un extrait du tableau des abréviations présentées dans la troisième partie.

quelques	qq	qqs	qques	qlqs	quelque
appartement	app	appart	apprt		
fond d'air	fond	fd a	f d'r	fond air	fd air
circulation	circul	circu(s)?	circl		
restaurant	resto(s)?	restau			
discussion	discut(.*)				
voix	vx				
parole	parl(.*)				
figuration	figu				
oiseaux	oisx	oiso(s)?	piaf(s)?		
chuchotement	chuchotis	chuchot(.*)	murmur(.*)		
interieur	int				
toilettes	wc				
etrange	strange				
presence	pres	prsce			
reverberation	reverberant	reverb	rvrb	rvb	rev
porte	pte	prte			
porte ouverte	PO				
porte fermée	PF				
fenetre ouverte	FO				
fenetre fermee	FF				

elements	elem	elts
passage	pass	

D Graphe des lexèmes contextuels


```
/*
{ "id" : "object_name",
    "children" : [
        { "id" : "child1", "values" : ["val1", "val2"] },
        { "id" : "child2", "values" : ["val1", "val2"] },
    ]
}
*/

{ "id" : "interieur",
    "children" : [
        { "id" : "acoustique",
            "children" : [
                { "id" : "taille", "values" : ["petit", "grand"] },
                { "id" : "stereo", "values" : ["none", "large"]},
                { "id" : "reverberation",
                    "children" : [
                        { "id" : "TR60", "values" : ["mat", "reverberant"] },
                        { "id" : "couleur", "values" : ["clair", "sombre"] }
                    ]
                }
            ]
        },
        { "id" : "source_sonore",
            "children" : [
                { "id" : "dans_le_lieu",
                    "children" : [
                        { "id" : "humain",
                            "children" : [
                                { "id" : "presence", "values" : ["none", "presence"] },
                                { "id" : "quantite", "values" : ["groupe", "foule"] },
                                { "id" : "voix", "values" : ["chuchotement", "voix", "cri"] },
                                { "id" : "caractere", "values" : ["calme", "activite", "agite"] },
                            ]
                        },
                        { "id" : "non-humain",
                            "children" : [
                                { "id" : "machine", "values" : ["machine", "moteur", "ventilateur"] },
                                { "id" : "electricite", "values" : ["hum",
                                "buzz", "neon", "induction"] },
                                { "id" : "eau", "values" : ["ecoulement", "tuyau"] },

                            ]
                        }

                    ]
                },
                { "id" : "hors_du_lieu",
                    "children" : [
                        { "id" : "avion", "values" : ["none", "avion"] },
                        { "id" : "circulation", "values" : ["none", "circulation"] },
                        { "id" : "oiseaux", "values" : ["none", "oiseaux"] }
                    ]
                }
            ]
        }
    ]
}
```

E Implémentation de l'algorithme de Porter (*Python*)

```
#!/usr/bin/env python

"""Porter Stemming Algorithm
This is the Porter stemming algorithm, ported to Python from the
version coded up in ANSI C by the author. It may be be regarded
as canonical, in that it follows the algorithm presented in

Porter, 1980, An algorithm for suffix stripping, Program, Vol. 14,
no. 3, pp 130-137,

only differing from it at the points maked --DEPARTURE-- below.

See also http://www.tartarus.org/~martin/PorterStemmer

The algorithm as described in the paper could be exactly replicated
by adjusting the points of DEPARTURE, but this is barely necessary,
because (a) the points of DEPARTURE are definitely improvements, and
(b) no encoding of the Porter stemmer I have seen is anything like
as exact as this version, even with the points of DEPARTURE!

Vivake Gupta (v@nano.com)

Release 1: January 2001

Further adjustments by Santiago Bruno (bananabruno@gmail.com)
to allow word input not restricted to one word per line, leading
to:

release 2: July 2008
"""

import sys

class PorterStemmer:

    def __init__(self):
        """The main part of the stemming algorithm starts here.
        b is a buffer holding a word to be stemmed. The letters are in b[k0],
        b[k0+1] ... ending at b[k]. In fact k0 = 0 in this demo program. k is
        readjusted downwards as the stemming progresses. Zero termination is
        not in fact used in the algorithm.

        Note that only lower case sequences are stemmed. Forcing to lower case
        should be done before stem(...) is called.
        """

        self.b = ""  # buffer for word to be stemmed
        self.k = 0
        self.k0 = 0
        self.j = 0   # j is a general offset into the string

    def cons(self, i):
        """cons(i) is TRUE <=> b[i] is a consonant."""
        if self.b[i] == 'a' or self.b[i] == 'e' or self.b[i] == 'i' or self.b[i] == 'o' or self.b[i]
== 'u':
            return 0
        if self.b[i] == 'y':
            if i == self.k0:
                return 1
            else:
                return (not self.cons(i - 1))
        return 1

    def m(self):
        """m() measures the number of consonant sequences between k0 and j.
        if c is a consonant sequence and v a vowel sequence, and <..>
        indicates arbitrary presence,
```

```
       <c><v>       gives 0
       <c>vc<v>     gives 1
       <c>vcvc<v>   gives 2
       <c>vcvcvc<v> gives 3
       ....
    """
    n = 0
    i = self.k0
    while 1:
        if i > self.j:
            return n
        if not self.cons(i):
            break
        i = i + 1
    i = i + 1
    while 1:
        while 1:
            if i > self.j:
                return n
            if self.cons(i):
                break
            i = i + 1
        i = i + 1
        n = n + 1
        while 1:
            if i > self.j:
                return n
            if not self.cons(i):
                break
            i = i + 1
        i = i + 1

def vowelinstem(self):
    """vowelinstem() is TRUE <=> k0,...j contains a vowel"""
    for i in range(self.k0, self.j + 1):
        if not self.cons(i):
            return 1
    return 0

def doublec(self, j):
    """doublec(j) is TRUE <=> j,(j-1) contain a double consonant."""
    if j < (self.k0 + 1):
        return 0
    if (self.b[j] != self.b[j-1]):
        return 0
    return self.cons(j)

def cvc(self, i):
    """cvc(i) is TRUE <=> i-2,i-1,i has the form consonant - vowel - consonant
    and also if the second c is not w,x or y. this is used when trying to
    restore an e at the end of a short  e.g.

       cav(e), lov(e), hop(e), crim(e), but
       snow, box, tray.
    """
    if i < (self.k0 + 2) or not self.cons(i) or self.cons(i-1) or not self.cons(i-2):
        return 0
    ch = self.b[i]
    if ch == 'w' or ch == 'x' or ch == 'y':
        return 0
    return 1

def ends(self, s):
    """ends(s) is TRUE <=> k0,...k ends with the string s."""
    length = len(s)
    if s[length - 1] != self.b[self.k]: # tiny speed-up
        return 0
    if length > (self.k - self.k0 + 1):
        return 0
    if self.b[self.k-length+1:self.k+1] != s:
        return 0
    self.j = self.k - length
```

```
        return 1

    def setto(self, s):
        """setto(s) sets (j+1),...k to the characters in the string s, readjusting k."""
        length = len(s)
        self.b = self.b[:self.j+1] + s + self.b[self.j+length+1:]
        self.k = self.j + length

    def r(self, s):
        """r(s) is used further down."""
        if self.m() > 0:
            self.setto(s)

    def step1ab(self):
        """step1ab() gets rid of plurals and -ed or -ing. e.g.

           caresses  ->  caress
           ponies    ->  poni
           ties      ->  ti
           caress    ->  caress
           cats      ->  cat

           feed      ->  feed
           agreed    ->  agree
           disabled  ->  disable

           matting   ->  mat
           mating    ->  mate
           meeting   ->  meet
           milling   ->  mill
           messing   ->  mess

           meetings  ->  meet
        """
        if self.b[self.k] == 's':
            if self.ends("sses"):
                self.k = self.k - 2
            elif self.ends("ies"):
                self.setto("i")
            elif self.b[self.k - 1] != 's':
                self.k = self.k - 1
        if self.ends("eed"):
            if self.m() > 0:
                self.k = self.k - 1
        elif (self.ends("ed") or self.ends("ing")) and self.vowelinstem():
            self.k = self.j
            if self.ends("at"):   self.setto("ate")
            elif self.ends("bl"): self.setto("ble")
            elif self.ends("iz"): self.setto("ize")
            elif self.doublec(self.k):
                self.k = self.k - 1
                ch = self.b[self.k]
                if ch == 'l' or ch == 's' or ch == 'z':
                    self.k = self.k + 1
            elif (self.m() == 1 and self.cvc(self.k)):
                self.setto("e")

    def step1c(self):
        """step1c() turns terminal y to i when there is another vowel in the stem."""
        if (self.ends("y") and self.vowelinstem()):
            self.b = self.b[:self.k] + 'i' + self.b[self.k+1:]

    def step2(self):
        """step2() maps double suffices to single ones.
        so -ization ( = -ize plus -ation) maps to -ize etc. note that the
        string before the suffix must give m() > 0.
        """
        if self.b[self.k - 1] == 'a':
            if self.ends("ational"):   self.r("ate")
            elif self.ends("tional"):  self.r("tion")
        elif self.b[self.k - 1] == 'c':
            if self.ends("enci"):      self.r("ence")
            elif self.ends("anci"):    self.r("ance")
```

```
        elif self.b[self.k - 1] == 'e':
            if self.ends("izer"):      self.r("ize")
        elif self.b[self.k - 1] == 'l':
            if self.ends("bli"):       self.r("ble") # --DEPARTURE--
            # To match the published algorithm, replace this phrase with
            #   if self.ends("abli"):      self.r("able")
            elif self.ends("alli"):    self.r("al")
            elif self.ends("entli"):   self.r("ent")
            elif self.ends("eli"):     self.r("e")
            elif self.ends("ousli"):   self.r("ous")
        elif self.b[self.k - 1] == 'o':
            if self.ends("ization"):   self.r("ize")
            elif self.ends("ation"):   self.r("ate")
            elif self.ends("ator"):    self.r("ate")
        elif self.b[self.k - 1] == 's':
            if self.ends("alism"):     self.r("al")
            elif self.ends("iveness"): self.r("ive")
            elif self.ends("fulness"): self.r("ful")
            elif self.ends("ousness"): self.r("ous")
        elif self.b[self.k - 1] == 't':
            if self.ends("aliti"):     self.r("al")
            elif self.ends("iviti"):   self.r("ive")
            elif self.ends("biliti"):  self.r("ble")
        elif self.b[self.k - 1] == 'g': # --DEPARTURE--
            if self.ends("logi"):      self.r("log")
        # To match the published algorithm, delete this phrase

    def step3(self):
        """step3() dels with -ic-, -full, -ness etc. similar strategy to step2."""
        if self.b[self.k] == 'e':
            if self.ends("icate"):     self.r("ic")
            elif self.ends("ative"):   self.r("")
            elif self.ends("alize"):   self.r("al")
        elif self.b[self.k] == 'i':
            if self.ends("iciti"):     self.r("ic")
        elif self.b[self.k] == 'l':
            if self.ends("ical"):      self.r("ic")
            elif self.ends("ful"):     self.r("")
        elif self.b[self.k] == 's':
            if self.ends("ness"):      self.r("")

    def step4(self):
        """step4() takes off -ant, -ence etc., in context <c>vcvc<v>."""
        if self.b[self.k - 1] == 'a':
            if self.ends("al"): pass
            else: return
        elif self.b[self.k - 1] == 'c':
            if self.ends("ance"): pass
            elif self.ends("ence"): pass
            else: return
        elif self.b[self.k - 1] == 'e':
            if self.ends("er"): pass
            else: return
        elif self.b[self.k - 1] == 'i':
            if self.ends("ic"): pass
            else: return
        elif self.b[self.k - 1] == 'l':
            if self.ends("able"): pass
            elif self.ends("ible"): pass
            else: return
        elif self.b[self.k - 1] == 'n':
            if self.ends("ant"): pass
            elif self.ends("ement"): pass
            elif self.ends("ment"): pass
            elif self.ends("ent"): pass
            else: return
        elif self.b[self.k - 1] == 'o':
            if self.ends("ion") and (self.b[self.j] == 's' or self.b[self.j] == 't'): pass
            elif self.ends("ou"): pass
            # takes care of -ous
            else: return
        elif self.b[self.k - 1] == 's':
            if self.ends("ism"): pass
```

```
            else: return
        elif self.b[self.k - 1] == 't':
            if self.ends("ate"): pass
            elif self.ends("iti"): pass
            else: return
        elif self.b[self.k - 1] == 'u':
            if self.ends("ous"): pass
            else: return
        elif self.b[self.k - 1] == 'v':
            if self.ends("ive"): pass
            else: return
        elif self.b[self.k - 1] == 'z':
            if self.ends("ize"): pass
            else: return
        else:
            return
        if self.m() > 1:
            self.k = self.j

    def step5(self):
        """step5() removes a final -e if m() > 1, and changes -ll to -l if
        m() > 1.
        """
        self.j = self.k
        if self.b[self.k] == 'e':
            a = self.m()
            if a > 1 or (a == 1 and not self.cvc(self.k-1)):
                self.k = self.k - 1
        if self.b[self.k] == 'l' and self.doublec(self.k) and self.m() > 1:
            self.k = self.k -1

    def stem(self, p, i, j):
        """In stem(p,i,j), p is a char pointer, and the string to be stemmed
        is from p[i] to p[j] inclusive. Typically i is zero and j is the
        offset to the last character of a string, (p[j+1] == '\0'). The
        stemmer adjusts the characters p[i] ... p[j] and returns the new
        end-point of the string, k. Stemming never increases word length, so
        i <= k <= j. To turn the stemmer into a module, declare 'stem' as
        extern, and delete the remainder of this file.
        """
        # copy the parameters into statics
        self.b = p
        self.k = j
        self.k0 = i
        if self.k <= self.k0 + 1:
            return self.b # --DEPARTURE--

        # With this line, strings of length 1 or 2 don't go through the
        # stemming process, although no mention is made of this in the
        # published algorithm. Remove the line to match the published
        # algorithm.

        self.step1ab()
        self.step1c()
        self.step2()
        self.step3()
        self.step4()
        self.step5()
        return self.b[self.k0:self.k+1]
```

F Mots-vides pour l'anglais (*stopwords*)

i
me
my
myself
we
our
ours
ourselves
you
your
yours
yourself
yourselves
he
him
his
himself
she
her
hers
herself
it
its
itself
they
them
their
theirs
themselves
what
which
who
whom
this
that
these
those
am
is
are
was
were
be
being
have
has
had
having
do
does
did
doing
would
should
could
ought
i'm
you're
he's
she's
it's
we're
they're
i've
you've
we've
they've
i'd
you'd
he'd
she'd
we'd
they'd
i'll
you'll
he'll
she'll
we'll
they'll
isn't
aren't
wasn't
weren't
hasn't
haven't
hadn't
don't
didn't
won't
wouldn't
shan't
shouldn't
can't
cannot
couldn't
mustn't
let's
that's
who's
what's
here's
there's
when's
where's
why's
how's
a
an
the
and
but
if
or
because
as
until
while
of
at
by
for
with
about
against
between
into
through
during
before
above
below
to
from
up
down
in
out
on
off
over
under
again
further
then
once
here
there
when
where
why
how
all
any
both
each
few
more
most
other
some
such
no
nor
not
only
own
same
so
than
too
very

been	doesn't	after

G Implémentation *Python* de TF-IDF et ASL

TF-IDF :

```
global matrix
def tfidf(matrix) :

    rows,cols = matrix.shape
    transformed_matrix = matrix.copy()

    for row in xrange(0, rows): #Pour chaque document

        word_total = reduce(lambda x, y: x+y, matrix[row] )
        word_total = float(word_total)

        for col in xrange(0, cols): #Pour chaque terme
            transformed_matrix[row,col] = float(transformed_matrix[row,col])

            if transformed_matrix[row][col] != 0:
                transformed_matrix[row,col] = _tf_idf(row, col, word_total)

    return transformed_matrix


def _tf_idf(row, col, word_total):
    global matrix

    term_frequency = matrix[row][col] / float(word_total)
    inverse_document_frequency = log(abs(len(matrix) / float(_get_term_document_occurrences(col))))

    return term_frequency * inverse_document_frequency


def _get_term_document_occurrences(col):
    """ Dans combien de documents apparaît le terme ..."""
    global matrix
    term_document_occurrences = 0
    rows, cols = matrix.shape

    for n in xrange(0,rows):
        if matrix[n][col] > 0: #Le terme est dans le document n
            term_document_occurrences +=1
    return term_document_occurrences
```

ASL :

```
from scipy import linalg.svd, dot #Importe la décomposition et le produit matriciel.
def asl(self, dimensions=1):
global matrix
    # Calcule la DVS de la matrice : SVD of objects matrix: U . SIGMA . VT = MATRIX
    # Réduit le rang de sigma par un certain facteur ('dimensions'), donnant sigma'.
    # On multiplie ensuite les matrices :  U . SIGMA' . VT = MATRIX'
    rows,cols = matrix.shape

    if dimensions <= rows: #C’est bien une réduction
        #On utilise une fonction du module scipy, linalg.svd, pour la décomposition
        u,sigma,vt = linalg.svd(matrix)

        #Réduction du rang, construction de SIGMA'
        for index in xrange(rows - dimensions, rows):
            sigma[index] = 0

        #Reconstruction de MATRIX'
        transformed_matrix = dot(dot(u, linalg.diagsvd(sigma, len(self.matrix), len(vt))) ,vt)
```

```
return transformed_matrix
```