POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Extraction de taxonomie par regroupement hiérarchique de plongements vectoriels de graphes de connaissances

FÉLIX MARTEL
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Juillet 2020

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Extraction de taxonomie par regroupement hiérarchique de plongements vectoriels de graphes de connaissances

présenté par **Félix MARTEL**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Michel GAGNON, président
Amal ZOUAQ, membre et directrice de recherche
Christopher PAL, membre

REMERCIEMENTS

Je remercie vivement Amal Zouaq pour son encadrement exigeant mais bienveillant, et pour ses conseils toujours pertinents.

Merci également à Michel Gagnon, qui m'a accueilli très chaleureusement à Montréal et m'a fait découvrir le monde du Web sémantique.

Ce travail a été soutenu par le Fond d'excellence en recherche d'Apogée Canada. Les ressources de calcul provenaient de Calcul Québec [1] et Calcul Canada [2].

1. www.calculquebec.ca
2. www.computecanada.ca

RÉSUMÉ

Les graphes de connaissances jouent aujourd'hui un rôle important pour représenter et stocker des données, bien au-delà du Web sémantique ; beaucoup d'entre eux sont obtenus de manière automatique ou collaborative, et agrègent des données issues de sources diverses. Dans ces conditions, la création et la mise à jour automatique d'une taxonomie qui reflète le contenu d'un graphe est un enjeu crucial.

Or, la plupart des méthodes d'extraction taxonomique adaptées aux graphes de grande taille se contentent de hiérarchiser des classes pré-existantes, et sont incapables d'identifier de nouvelles classes à partir des données. Dans ce mémoire, nous proposons une méthode d'extraction de taxonomie expressive applicable à grande échelle, grâce à l'utilisation de plongements vectoriels. Les modèles de plongement vectoriel de graphe fournissent une représentation vectorielle dense des éléments d'un graphe, qui intègre sous forme géométrique les régularités des données : ainsi, deux éléments sémantiquement proches dans le graphe auront des plongements vectoriels géométriquement proches.

Notre but est de démontrer le potentiel du regroupement hiérarchique non-supervisé appliqué aux plongements vectoriels sur la tâche d'extraction de taxonomie. Pour cela, nous procédons en deux étapes : nous montrons d'abord qu'un tel regroupement est capable d'extraire une taxonomie sur les classes existantes, puis qu'il permet de surcroît d'identifier de nouvelles classes et de les organiser hiérarchiquement, c'est-à-dire d'extraire une taxonomie expressive.

Pour l'extraction de taxonomie sur les classes existantes, nous proposons deux méthodes capables d'associer des classes existantes à des groupes d'entités en tenant compte de la structure d'arbre qui existe entre ces groupes ; cela permet de transformer l'arbre de clustering issu du regroupement hiérarchique en une taxonomie. La première de ces méthodes consiste à trouver une injection optimale des classes vers les clusters en résolvant un problème d'optimisation linéaire. La seconde est un lissage de la méthode précédente, conçue pour mieux tenir compte du bruit dans les données. Nous appliquons ces deux méthodes à DBpedia, et montrons qu'elles sont toutes deux capables de surpasser une méthode basée sur un regroupement supervisé.

Pour l'extraction de taxonomie expressive, nous présentons une méthode d'extraction d'axiomes capable de tirer profit d'un arbre de clustering pour obtenir des exemples positifs et négatifs pertinents, et induire des axiomes à partir de ces exemples. Nous y ajoutons un mécanisme de tirage aléatoire capable d'augmenter récursivement la spécificité des entités traitées, et donc de construire progressivement une taxonomie complète. Sur DBpedia, notre approche

est capable de reconstituer la taxonomie de référence sur les classes existantes, mais aussi de décrire ces classes au moyen d'axiomes logiques et d'identifier de nouvelles classes pertinentes.

ABSTRACT

Knowledge graphs are the backbone of the Semantic Web, and have been succesfully applied to a wide range of areas. Many of these graphs are built automatically or collaboratively, and aggregate data from various sources. In these conditions, automatically creating and updating a taxonomy that accurately reflects the content of a graph is an important issue.

However, among scalable taxonomy extraction approaches, most of them can only extract a hierarchy on existing classes, and are unable to identify new classes from the data. In this thesis, we propose a novel taxonomy extraction method based on knowledge graph embeddings that is both scalable and expressive. A knowledge graph embedding model provides a dense, low-dimensional vector representation of the entities of a graph, such that similar entities in the graph are embedded close to each other in the embedding space.

Our goal is to show how these graph embeddings can be combined with unsupervised hierarchical clustering to extract a taxonomy from a graph. We first show that unsupervised clustering is able to extract a taxonomy on existing classes. Then, we show that it can also be used to identify new classes and organize them hierarchically, thus creating an expressive taxonomy.

For the non-expressive taxonomy extraction task, we introduce two methods for mapping existing classes to clusters of entities. The first of these methods solves a linear optimization problem in order to find an optimal injective function from classes to clusters. The second one can be seen as a smoothed version of the first one, designed to better handle noise and uncertainty in the data. In both cases, the resulting mapping is used to transform the clustering tree into a taxonomy. We run experiments with these two methods on DBpedia, and show that they both outperform a method based on supervised clustering.

For the expressive extraction task, we propose an axiom extraction method that leverages the clustering tree to define positive and negative samples, and induces new axioms from these samples. Since samples are chosen based on the similarity of their embeddings, this method effectively narrows down the search space to relevant subsets of the full graph. We also add a resampling mechanism, which allows us to extract increasingly specific axioms. We try our method on DBpedia, and show that the predicted taxonomy is able to rebuild the reference taxonomy with good precision, and that it can also identify new relevant classes and describe them with logical axioms.

NOTATIONS

Ensembles

— $\mathbb{N}$: l'ensemble des entiers naturels, $\mathbb{N} = \{1, 2, 3, \ldots\}$.
— $\mathbb{R}$: l'ensemble des nombres réels.
— $\mathbb{C}$: l'ensemble des nombres complexes.
— $\mathbb{R}^d$ (resp. $\mathbb{C}^d$) : l'ensemble des vecteurs réels (resp. complexes) de dimension d.
— $\mathbb{R}^{m\times n}$ (resp. $\mathbb{C}^{m\times n}$) : l'ensemble des matrices réelles (resp. complexes) de dimension m par n.

Fonctions

— $\lfloor x \rfloor$: le plus grand entier inférieur ou égal à x.
— $\Re(z)$: la partie réelle d'un nombre complexe $z \in \mathbb{C}$.
— $\Im(z)$: la partie imaginaire d'un nombre complexe $z \in \mathbb{C}$.

Vecteurs, matrices

Dans tout le document, les caractères gras minuscules ($\mathbf{e}, \mathbf{u}, \ldots$) désignent des vecteurs et les caractères gras majuscules ($\mathbf{A}, \mathbf{M}, \mathbf{X}, \ldots$) désignent des matrices.

— $\mathbf{I_d}$: la matrice identité de dimension $d \times d$.
— $\mathbf{o_d}$: le vecteur nul de dimension d.
— $\mathbf{0_{d\times d}}$: la matrice nulle de dimension $d \times d$.
— $\mathbf{M}^\top$: la transposée de la matrice $\mathbf{M}$.
— $\overline{\mathbf{M}}$: la matrice conjuguée d'une matrice complexe $\mathbf{M}$.
— $\mathrm{diag}(d_1, d_2, \ldots, d_k)$: la matrice diagonale de dimension $k\times k$, dont la i-ème coordonnée diagonale vaut d_i.

Pour deux vecteurs $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$, on utilisera le produit scalaire usuel sur $\mathbb{R}^n$:

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{n} \mathbf{u_i} \times \mathbf{v_i} \tag{1}$$

Ce produit scalaire définie une norme, appelée norme euclidienne, pour tout vecteur $\mathbf{u} \in \mathbb{R}^n$:

$$\|\mathbf{u}\|_2 = \sqrt{\mathbf{u} \cdot \mathbf{u}} = \sqrt{\sum_{i=1}^{n} u_i^2} \tag{2}$$

La norme euclidienne définit à son tour une distance, la *distance euclidienne*, qui vaut, pour $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$:

$$d_{\text{euc}}(\mathbf{u}, \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\|_2 = \sqrt{\sum_{i=1}^{n} (\mathbf{u_i} - \mathbf{v_i})^2} \tag{3}$$

La distance cosinus, qui n'est pas une distance mathématique mais plutôt une mesure de dissimilarité, est définie sur $\mathbb{R}^n$ par :

$$d_{\cos}(\mathbf{u}, \mathbf{v}) = 1 - \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|_2 \times \|\mathbf{v}\|_2} \tag{4}$$

La norme de Frobenius est définie pour toute matrice réelle ou complexe de dimension de $m \times n$ par :

$$\|\mathbf{M}\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |M_{i,j}|^2} \tag{5}$$

Graphes de connaissances

— $\mathcal{E}(G)$: l'ensembles des entités (ou *sommets*) d'un graphe G.
— $\mathcal{R}(G)$: l'ensemble des relations (ou *arêtes*) d'un graphe G.
— $h \xrightarrow{r} t$: la propriété «h est lié à t par la relation r».

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Les graphes de connaissances constituent l'ossature du Web sémantique, et trouvent aujourd'hui des applications toujours plus variées : ils servent pour la recherche d'information [1, 2], la recommandation de contenu [3–5] ou la réponse automatique aux questions [6–8]. Au-delà de l'informatique, ils sont désormais utilisés dans des domaines aussi divers que les sciences biomédicales [9, 10], la recherche historique [11, 12] ou les sciences sociales [13].

Un graphe de connaissances est constitué d'entités reliées les unes aux autres par des relations, mais aussi d'une ontologie, c'est-à-dire d'une collection d'axiomes logiques qui en décrit le contenu : ces axiomes définissent et caractérisent des classes (autrement dit, des groupes d'entités ayant des caractéristiques communes), établissent des liens logiques entre classes, relations et entités et peuvent imposer des contraintes sur les données. Toutefois, si l'extraction automatique de graphes de connaissances à partir du Web est désormais courante [14], la création automatique d'ontologie demeure un problème ouvert. On introduit ici le contexte et les enjeux de ce problème, et les difficultés qui y sont liées.

1.1 Éléments de la problématique

Comme n'importe quelle base de données, un graphe de connaissances stocke de l'information de manière structurée et la rend accessible par le biais de requêtes ; toutefois, par rapport aux bases de données relationnelles classiques, un graphe de connaissances permet une représentation plus flexible des données : il est facile d'ajouter ou de retirer des triplets, de modifier le schéma du graphe, d'agréger des sources diverses et de lier des graphes de connaissances entre eux, ce qui constitue la clé du Web des données (*Linked Data*). Par rapport au texte, qui constitue un autre moyen dominant pour représenter la connaissance humaine, le graphe de connaissances présente l'avantage d'avoir une sémantique formelle, qui fournit une interprétation non-ambigüe des faits qu'il contient et permet donc son utilisation par des machines.

Pour tirer pleinement profit de cette structure, il faut disposer d'une ontologie aussi complète que possible. En effet, une ontologie permet d'éviter les incohérences au sein d'un graphe [15], d'attribuer automatiquement un type aux entités [16] ou d'inférer de faits nouveaux [17]. Les ontologies facilitent également la fusion de graphes de connaissances, c'est-à-dire la mise en correspondance de plusieurs graphes, grâce à des méthodes d'alignement [18]. Dans le cas d'un graphe extrait automatiquement, une ontologie peut être utilisée pour l'extraction, le nettoyage ou l'uniformisation des données [19, 20]. Les ontologies constituent également

une piste prometteuse dans le domaine de l'intelligence artificielle explicable (*explainable AI*) [21–23].

Or construire une ontologie manuellement est coûteux, d'autant plus que la plupart des graphes sont dynamiques et changent au cours du temps. En particulier, de nouvelles classes peuvent émerger des données ; ces nouvelles classes peuvent effectivement correspondre à l'apparition de nouveaux types d'entités, mais elles peuvent aussi signaler des incomplétudes ou des mésusages du graphe de connaissances. En effet, lorsqu'un vocabulaire ontologique (c'est-à-dire un ensemble de classes et de relations) est utilisé par des utilisateurs humains ou des programmes de population automatique de graphes, il peut se produire des incompréhensions, par ces utilisateurs, de l'intention du concepteur de l'ontologie, autrement dit un conflit entre l'usage *prévu* et un ou plusieurs usages *réels* [24]. On peut par exemple voir émerger des sous-classes qui n'utilisent qu'une partie des relations prévues : de telles sous-classes ne sont que des artefacts liés à l'extraction, et n'ont pas d'équivalence dans le monde réel. Détecter ce genre de sous-classes permet donc de corriger des mauvaises pratiques et d'uniformiser la représentation des données. Dans ces conditions, pouvoir construire automatiquement, à partir d'un graphe, une ontologie qui corresponde aux données et à leur utilisation est un enjeu important pour l'extraction et la maintenance automatique et à grande échelle de graphes de connaissances [25].

Le problème de l'extraction automatique d'ontologies est très étudié, et notamment sous une forme particulière : l'extraction de *taxonomies*. Une taxonomie est une hiérarchie entre les classes d'un graphe ; on peut la voir comme une ontologie réduite à des axiomes de la forme $A \sqsubset B$, qui indique que la classe A est une sous-classe de B, et donc que toutes les entités qui font partie de A font également partie de B. De tels axiomes sont appelés axiomes de *subsumption* ; un exemple de taxonomie est donné à la figure 1.1.

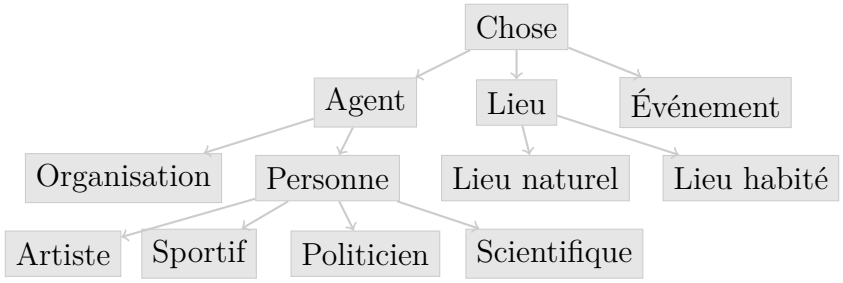


Figure 1.1 Un exemple de taxonomie généraliste.

Pour extraire une taxonomie, on peut s'appuyer sur le contenu d'un graphe de connaissances [26–30] ou sur un corpus textuel [31–36]. Ce choix d'un graphe ou d'un corpus délimite deux grandes familles d'approches, qui utilisaient à l'origine des méthodes très différentes :

inférence de nouveaux axiomes par des méthodes symboliques ou statistiques dans le premier cas [26, 28], identification de motifs lexico-syntaxiques dans un texte dans le second cas [37, 38]. Aujourd'hui, on observe une certaine convergence des méthodes grâce à l'utilisation de représentations vectorielles denses des éléments manipulés (qu'il s'agisse d'entités [39], de mots [40] ou de classes [41]). Ces représentations vectorielles visent à traduire certaines régularités sous forme géométrique, afin que des éléments similaires (par exemple, des mots aux sens apparentés) soient représentés par des vecteurs géométriquement proches. Dans le cas d'un graphe, on parle de *plongements vectoriels de graphe* (ou *knowledge graph embeddings* en anglais) ; dans le cas du texte, on parle de *plongements lexicaux* (ou *word embeddings* en anglais). Un exemple de tels plongements est représenté à la figure 1.2.

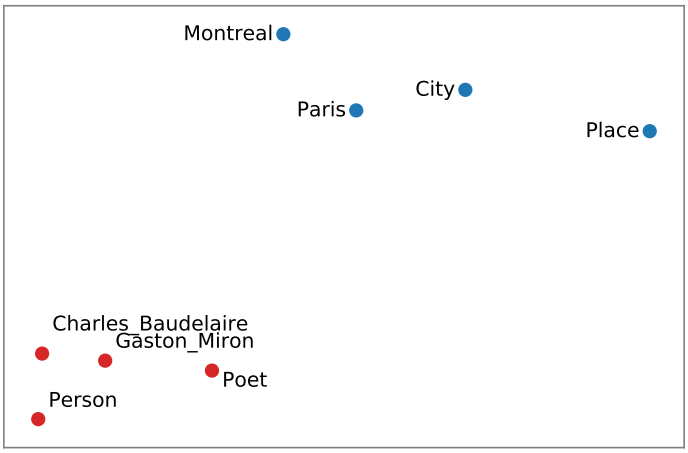


Figure 1.2 Un exemple de plongements vectoriels de graphe, représentés en deux dimensions [2].

Qu'ils soient entraînés sur un graphe ou sur du texte, ces plongements permettent l'application de diverses techniques issues de l'apprentissage automatique au problème de l'extraction de taxonomie, par exemple en entraînant un classificateur capable de détecter la subsumption [33], ou en regroupant les plongements sur la base de leur proximité géométrique [34, 42].

Toutefois, la plupart de ces méthodes se contentent d'organiser hiérarchiquement des classes pré-existantes, et ne sont pas capables de caractériser ces classes (que ce soit au moyen d'axiomes logiques, de descriptions textuelles ou même de mots-clés), ni d'identifier de nouvelles classes à partir des données. Dans le présent mémoire, nous proposons au contraire

2. Ces plongements ont été obtenus sur DBpedia [14] avec le modèle TransE [39]. Leur dimension est ramenée de $d = 50$ à $d = 2$ grâce à une PCA.

une identification non-supervisée de groupes d'entités cohérents, ce qui permet à la fois de détecter des classes, nouvelles ou pré-existantes, et de les organiser hiérarchiquement au sein d'une taxonomie.

Cette identification se base sur un regroupement hiérarchique ascendant (ou *clustering* hiérarchique) des plongements vectoriels, qui produit un arbre de clustering organisant hiérarchiquement des groupes (ou *clusters*) d'entités en fonction de la distance entre leurs plongements. À partir de cet arbre, on peut produire une taxonomie sur les classes pré-existantes grâce à une méthode d'association entre les classes et les clusters. L'arbre de clustering permet également l'extraction d'une taxonomie *expressive*, c'est-à-dire dont les classes peuvent être soit des classes nommées, soit des classes complexes décrites par des axiomes logiques. Pour ce faire, nous proposons une méthode d'extraction d'axiomes qui cherche spécifiquement à expliquer la partition d'un cluster en deux sous-clusters au sein de l'arbre de clustering. Chacun de ces deux sous-clusters est constitué d'entités géométriquement proches, donc sémantiquement similaires ; on exploite donc la géométrie des plongements pour restreindre l'espace de recherche à un sous-ensemble pertinent du graphe. Cette restriction de l'espace de recherche diminue suffisamment la complexité en temps de l'algorithme pour permettre son application à un graphe de grande taille.

1.2 Objectifs de recherche

Dans ce mémoire, nous cherchons à extraire une taxonomie expressive à partir des plongements vectoriels d'un graphe de connaissances. Notre hypothèse est en effet que les plongements vectoriels intègrent dans leur géométrie des notions de proximité entre entités, mais aussi des informations de nature taxonomique, et qu'il doit donc être possible de les utiliser à la fois pour l'identification de groupes d'entités et pour la hiérarchisation de ces groupes. Notre question de recherche s'énonce donc ainsi :

> *Comment les plongements vectoriels de graphe peuvent-ils contribuer à l'extraction de taxonomie ?*

Pour résoudre ce problème, nous proposons d'utiliser un regroupement hiérachique non-supervisé sur les plongements vectoriels, ce qui permet de créer une structure hiérarchique sur des groupes d'entités. Pour transformer cette structure en une taxonomie expressive ou non-expressive, il est nécessaire de répondre aux sous-questions suivantes :

> ***Q1.*** *Comment assigner des concepts à des groupes d'entités en tenant compte de la structure d'arbre entre ces groupes ?*

Q2. *Comment décrire un groupe d'entités à l'aide d'axiomes logiques expressifs ?*

1.3 Plan du mémoire

On présente d'abord un panorama de la littérature existante au chapitre 2. On y introduit les concepts fondamentaux du Web sémantique, et notamment la logique descriptive. On décrit ensuite les techniques existantes pour extraire automatiquement des ontologies ou des taxonomies, que ce soit à partir de textes ou de graphes.

Le chapitre 3 est consacré aux modèles de plongement, qui permettent une représentation vectorielle des éléments d'un graphe et servent de base à notre travail. On présente plusieurs familles de modèles, et on propose une méthode afin d'identifier le modèle le plus performant pour notre extraction de taxonomie.

Les deux chapitres suivants sont consacrés à notre méthodologie d'extraction de taxonomie à partir de graphes. Nous montrons d'abord que le regroupement non-supervisé de plongements vectoriels peut servir à reconstituer une taxonomie sur les classes existantes (chapitre 4). Pour cela, nous proposons deux méthodes pour assigner un type à des groupes qui émergent d'un processus de regroupement hiérarchique. Nous présentons ensuite une méthode pour identifier de nouvelles classes à partir du regroupement non-supervisé, et donc pour produire une taxonomie expressive à partir d'un graphe (chapitre 4). Cette méthode repose sur un algorithme d'extraction d'axiomes qui exploite le regroupement pour réduire la taille de l'espace de recherche.

CHAPITRE 2 CONTEXTE ET TRAVAUX CONNEXES

On présente ici le contexte général de nos travaux, ainsi qu'un aperçu de la littérature existante. La section 2.1 introduit les concepts fondamentaux du Web sémantique : les graphes de connaissances, qui offrent une représentation structurée de la connaissance, et la logique descriptive, qui permet d'enrichir ces graphes de règles logiques dont l'ensemble constitue une taxonomie ou une ontologie, selon les cas.

Nous présentons ensuite les techniques existantes pour extraire automatiquement des taxonomies ou des ontologies. Cette tâche d'extraction automatique a en effet fait l'objet de nombreux travaux : nous résumons ici les principales approches ainsi que les idées notables du domaine, en distinguant notamment les approches basées sur du texte, que l'on présente à la section 2.2.1, et les approches basées sur les graphes, qui sont décrites dans la section 2.2.2.

Notons enfin que les plongements vectoriels de graphe, qui constituent le socle de notre travail, font l'objet d'un chapitre dédié (chapitre 3).

2.1 Web sémantique et logique descriptive

2.1.1 Graphes de connaissances

Un graphe de connaissances est une collection structurée de données permettant de représenter des informations, ou des *faits*, sur le monde réel. La définition exacte d'un graphe de connaissances varie d'un auteur à l'autre [43] ; parmi les caractéristiques fréquemment retenues, on notera l'interconnexion des données et le caractère multi-relationnel, l'existence d'une sémantique formelle et de mécanismes de raisonnement, et parfois une certaine taille critique. Les paragraphes suivants visent à définir et expliciter ces différentes propriétés.

Concepts de base Dans ce travail, on adopte une définition très générale d'un graphe de connaissances vu comme un ensemble d'*entités* liées entre elles par des *relations*. Pour un ensemble d'entités $\mathcal{E}$ et un ensemble de relations $\mathcal{R}$, un graphe de connaissances est simplement un ensemble de triplets $\mathcal{KG} \subseteq \mathcal{E} \times \mathcal{R} \times \mathcal{E}$. Si (h, r, t) est un triplet du graphe, alors l'entité h est reliée à l'entité t par la relation r ; dans ce cas, on dit que h est le *sujet* du triplet, et t son objet. Le choix des notations s'explique par l'anglais, h désignant la tête (*head*) et t la queue (*tail*) du triplet. Dans la suite, on désignera par *voisinage* d'une entité l'ensemble des triplets qui impliquent cette entité en tant que sujet ou objet, et on notera

$h \xrightarrow{r} t$ pour indiquer que le triplet (h, r, t) est présent dans le graphe.

En pratique, lorsqu'on parle de graphe de connaissances, on s'attend à quelques restrictions supplémentaires. D'une part, les données d'un graphe de connaissances doivent être *multi-relationnelles*, c'est-à-dire qu'il doit exister plusieurs relations possibles entre les entités ($|\mathcal{R}| > 1$) ; d'autre part, les entités sont reliées *mutuellement* entre elles. Ainsi, un système reliant des fichiers à des métadonnées textuelles ne saurait être qualifié de graphe de connaissances ; dans un graphe de connaissances, les objets d'un triplet sont, au moins pour partie, sujets d'autre triplets, et le graphe forme donc un réseau d'entités interconnectées.

Le format dominant pour représenter un tel graphe est le format RDF, pour *Resource Description Framework* [44]. En RDF, les éléments sont de deux types : soit des identifiants, soit des littéraux. Les **identifiants**, ou IRI, pour *International Resource Identifier*, sont simplement des chaînes de caractères qui désignent des ressources du monde réel : lieux, personnes, documents, etc. Les IRI servent également à représenter les relations du graphe. RDF n'impose pas de mécanisme pour relier les IRI aux entités qu'elles désignent : c'est au créateur du graphe d'établir cette correspondance. Dans le cadre du Web des données, les IRI prennent la forme d'un lien HTTP. Les **littéraux** permettent de représenter des valeurs fixes : nombres, textes, booléens, dates, sommes d'argent. Contrairement aux identifiants, l'interprétation des littéraux est définie une fois pour toutes ; la manière d'interpréter un littéral est indiquée par son *format* (parfois appelé *datatype* en anglais). Par exemple, le littéral `1901-01-28^^xsd:date` est constitué de la valeur `1901-01-28` au format `xsd:date`, et s'interprète de manière unique comme la date du 28 janvier 1901.

Un *vocabulaire* est un ensemble d'IRI, présentant une certaine cohérence. Les IRI d'un même vocabulaire partagent quasi systématiquement une racine commune, que l'on peut alors abréger par un *préfixe*. Ainsi, toutes les IRI du vocabulaire standard RDF commencent par `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, que l'on abrège en `rdf:`. Par exemple, les IRI `http://www.w3.org/1999/02/22-rdf-syntax-ns#type` et `http://www.w3.org/1999/02/22-rdf-syntax-ns#Property` sont abrégées en `rdf:type` et `rdf:Property`, respectivement.

Un exemple : DBpedia Il existe de nombreux graphes de connaissances publics ; certains sont restreints à un domaine spécifique, comme UniProt [45] ou DrugBank [46], d'autres sont généralistes, comme YAGO [47] ou Wikidata [48]. Parmi ces graphes de connaissances généralistes, l'un des plus notables est DBpedia [14], sur lequel on s'appuiera dans toute la suite de ce travail, dans sa version anglaise. Les données de DBpedia sont obtenues de manière automatique en agrégeant différentes sources structurées ou semi-structurées issues de

Wikipédia. Le cœur de DBpedia est son système d'extraction, qui convertit Wikipédia en un ensemble de triplets RDF à l'aide de différentes heuristiques ; cet extracteur s'appuie essentiellement sur les infoboîtes de Wikipédia, des encarts constitués de paires clé-valeur et dont le format est relativement uniformisé à l'échelle de Wikipédia. Outre les infoboîtes, l'extracteur utilise également les liens de redirection et de désambiguïsation, les catégories des pages, les éventuelles informations de géolocalisation, etc. À cet extracteur s'ajoutent différentes procédures pour nettoyer et uniformiser les résultats, ainsi que pour agréger des données provenant d'autres sources. DBpedia fournit également un point d'entrée pour effectuer des requêtes sur le graphe [1].

Les entités de DBpedia sont regroupées au sein d'un vocabulaire `http://dbpedia.org/resource/`, généralement abrégé en `dbr:`. Le voisinage d'une entité donnée `dbr:<Nom de l'entité>` peut être consulté en ligne à l'adresse dbpedia.org/page/<Nom de l'entité>. Les relations et les classes de DBpedia, elles, font partie du vocabulaire `http://dbpedia.org/ontology/`, abrégé en `dbo:`. DBpedia fait parfois appel à des relations ou des classes issues d'autres vocabulaires, dont FOAF [2] et Dublin Core [3].

Dans la version que nous utilisons, DBpedia contient environ 1 300 relations et 3,4 millions d'entités pour un total de 63,6 millions de triplets. Les entités sont réparties selon 589 types, dont les plus fréquents sont les personnes (`dbo:Person`), les lieux (`dbo:Place`), les espèces (`dbo:Species`), les événements (`dbo:Event`), les œuvres (`dbo:Work`), les organisations (`dbo:Organisation`). Comme les données de DBpedia sont extraites automatiquement de Wikipédia et que Wikipédia est modifié régulièrement, le graphe DBpedia évolue lui aussi régulièrement.

Typologie des relations Il est utile de détailler les propriétés que peuvent avoir les relations dans un graphe de connaissances. Nous citons ici deux propriétés importantes qui serviront dans la suite du mémoire, à savoir la transitivité et la symétrie :

- une relation r est **transitive** si l'existence des liens $x \xrightarrow{r} y$ et $y \xrightarrow{r} z$ implique l'existence de $x \xrightarrow{r} z$: c'est le cas notamment des relations `dbo:ancestor` et `dbo:isPartOf` ;
- une relation r est **symétrique** si $t \xrightarrow{r} h$ est vérifiée dès que $h \xrightarrow{r} t$ est vérifiée. Par exemple, la relation `owl:sameAs`, qui indique que deux IRI correspondent à la même entité, est naturellement symétrique. Plus généralement, toutes les relations qui impliquent une équivalence ou une réciprocité sont symétriques, comme `dbo:spouse` ou `dbo:colleague` ;

1. dbpedia.org/sparql
2. xmlns.com/foaf/spec/
3. dublincore.org/

— à l'inverse, une relation r est **antisymétrique** si $h \xrightarrow{r} t$ et $t \xrightarrow{r} h$ ne peuvent jamais être vérifiées simultanément. C'est le cas, en pratique, de la plupart des relations hiérarchiques comme `rdfs:subClassOf` ou `dbo:parent` ;

Une relation peut très bien n'être ni symétrique, ni antisymétrique. La relation `dbo:influencedBy`, qui marque l'influence intellectuelle d'une personne sur un autre, est parfois mutuelle (pensons par exemple à Russell et Wittgenstein, ou Beauvoir et Sartre) et parfois non.

On peut également classer les relations selon les cardinalités relatives de leurs sujets et de leurs objets :

— une relation ***one-to-many*** (de un à plusieurs, en français) lie chaque sujet à plusieurs objets : par exemple, pour la relation `dbo:product`, une marque est reliée à tous les produits qu'elle commercialise, tandis qu'un produit ne possède qu'une seule marque ;

— à l'opposé, une relation ***many-to-one*** (de plusieurs à un) lie plusieurs sujets à un même objet ; c'est le cas de la relation `foaf:gender` qui, dans DBpedia, relie un million et demi de sujets à seulement trois objets (`"male"`, `"female"` et `"non-binary"`) ;

— une relation ***one-to-one*** (un pour un) relie un sujet à un unique objet, et réciproquement. On peut penser à la relation qui unit un pays à sa capitale (`dbo:capital`), ou une entité à son label (`rdfs:label`) ;

— enfin, une relation ***many-to-many*** (de plusieurs à plusieurs) relie un sujet à plusieurs objets, et un objet à plusieurs sujets. Cette catégorie comprend par exemple la relation `dbo:influencedBy` citée plus haut : en général, un artiste ou un penseur est influencé par plusieurs personnes, et en influence d'autres à son tour ;

Pour déterminer à quelle catégorie appartient une relation, il suffit de comparer le nombre moyen d'objets par sujet au nombre moyen de sujets par objet [49]. Pour une relation r donnée, on note tph_r (de l'anglais *tails per head*) le nombre moyen d'objets reliés à une entité par la relation r :

$$tph_r = \frac{\sum_{h \in \mathcal{E}} |\{t \in \mathcal{E} : h \xrightarrow{r} t\}|}{|\{h \in \mathcal{E} : \exists y \in \mathcal{E}, h \xrightarrow{r} y\}|} \tag{2.1}$$

Et, réciproquement, on note hpt_r (de l'anglais *heads per tail*) le nombre moyen de sujets qui sont reliés à un objet par la relation r.

Dès lors, une relation r est *one-to-many* si $hpt_r = 1$ et $tph_r > 1$, *many-to-one* si $hpt_r > 1$ et $tph_r = 1$, *one-to-one* si $hpt_r = tph_r = 1$, et *many-to-many* si $hpt_r > 1$ et $tph_r > 1$. En pratique, pour les graphes de connaissances extraits automatiquement, cette classification est parfois trop stricte ; certains auteurs comme [39, 49] proposent de tenir compte du bruit des données en caractérisant les relations *one-to-many* par $hpt_r \leq 1,5$ et $tph_r > 1,5$, *many-to-one* par $hpt_r > 1,5$ et $tph_r \leq 1,5$, *one-to-one* par $hpt_r \leq 1,5$ et $tph_r \leq 1,5$ et *many-to-many* par $hpt_r > 1,5$ et $tph_r > 1,5$. On désignera cette catégorisation par le qualificatif d'*assouplie*,

par opposition à la catégorisation *stricte* présentée plus haut.

La répartition des relations entre les différentes catégories est présentée au tableau 2.1. On peut voir que, au sens strict du terme, les relations *one-to-one*, *one-to-many* et *many-to-one* sont rares ; les relations *many-to-many* constituent plus des trois quarts des relations du graphe. En revanche, si l'on assouplit les conditions pour tenir compte du bruit des données, on trouve une majorité de relations *many-to-one* et *one-to-one*.

Tableau 2.1 Répartition des différents types de relations dans DBpedia.

	Catégorisation	
	stricte	assouplie
one-to-one	7,54%	35,39%
one-to-many	2,54%	7,69%
many-to-one	12,26%	46,27%
many-to-many	77,66%	10,66%

Comparer hpt_r et tph_r permet également d'aller au-delà des quatre catégories ci-dessus, et de mieux appréhender la diversité des relations. Examinons par exemple les deux relations *many-to-many* suivantes : `dbo:influencedBy` et `rdf:type`[4]. Pour la relation `dbo:influencedBy`, on obtient $hpt_r = 2,7$ et $tph_r = 2,9$. Autrement dit, une personne est influencée par 2,9 personnes en moyenne, et en influence 2,7[5]. On a donc un équilibre entre sujets et objets. À l'inverse, pour `rdf:type`, on trouve $tph_r = 3,57$ et $hpt_r = 26228$: une entité a en moyenne 3,57 types, et un type a en moyenne 26 228 instances. On a donc, dans un cas, un équilibre entre sujets et objets, et de l'autre un net déséquilibre. Ces considérations permettent de nuancer les quatre catégories décrites plus haut, et servent notamment au chapitre 3, sections 3.1.2 et 3.2.2.

2.1.2 Ontologies et logique descriptive

La logique descriptive est un terme général englobant une famille de systèmes logiques, conçus précisément pour décrire des données multi-relationnelles. La logique descriptive cherche à obtenir un compromis entre deux aspects : l'expressivité – la capacité à décrire des situations complexes – et la calculabilité – la possibilité d'effectuer des raisonnements en un temps acceptable. Selon l'équilibre souhaité entre ces deux aspects, on autorise ou non certains types d'axiomes : $\mathcal{SROIQ}$ [50] est le système le plus large et le plus expressif, qui autorise

4. `rdf:type` est une relation *many-to-many*, car une entité a généralement plusieurs types, et un type a plusieurs instances.

5. Plus rigoureusement : une personne qui exerce de l'influence l'exerce en moyenne sur 2,7 personnes ; une personne influencée par d'autres est en moyenne influencée par 2,9 personnes.

l'ensemble des axiomes de la logique descriptive ; $\mathcal{EL}$ et DL-lite sont parmi les plus simples [51] ; $\mathcal{ALC}$ se situe entre ces deux extrêmes.

La logique descriptive manipule trois types d'éléments : des entités (ou *individus*), des relations, et des concepts. Les concepts sont des ensembles d'entités, et les relations décrivent des liens entre deux entités. Entités, relations et concepts peuvent être combinés entre eux pour former des *axiomes* ; les types de combinaisons possibles varient selon la logique descriptive choisie. Une ontologie (parfois appelée un *schéma*) est alors simplement un ensemble d'axiomes, chacun d'entre eux devant être vrai dans la situation décrite. Ici, on présente un aperçu des types d'axiomes les plus utilisés ; pour une description plus complète, on pourra consulter [52].

La forme la plus simple d'un concept est un concept nommé (parfois appelé un *type*), comme `Personne`, `Lieu`, etc. On peut également définir un *concept singleton*, c'est-à-dire un concept contenant une unique instance : {`alice`} ou {`bob`} par exemple. Enfin, on dispose de deux concepts spéciaux : le concept universel $\top$, qui contient toutes les entités, et le concept vide $\bot$, qui n'en contient aucune. On peut alors assembler et relier ces différents types de concepts, en utilisant les axiomes qui suivent.

Une première forme d'axiome est la **subsumption** entre un concept A et un concept B, notée $A \sqsubseteq B$, qui indique que les instances de A sont également instances de B. Par exemple :

$$\texttt{Athlète} \sqsubseteq \texttt{Personne} \tag{2.2}$$

On peut aussi indiquer l'**équivalence** entre deux concepts A et B, en écrivant $A \equiv B$. Deux concepts sont équivalents s'ils partagent les mêmes instances, ce qui correspond à une subsumption mutuelle $A \sqsubseteq B$ et $B \sqsubseteq A$. Par exemple :

$$\texttt{Personne} \equiv \texttt{Humain} \tag{2.3}$$

Pour combiner deux concepts A et B, on possède un opérateur de disjonction et un opérateur de conjonction. La **disjonction** (ou *union)* de A et de B, notée $A \sqcup B$, est un nouveau concept qui contient tous les individus qui sont instances de A ou instances de B. Cela permet par exemple d'exprimer l'idée qu'un animal est soit un vertébré, soit un invertébré :

$$\texttt{Animal} \equiv \texttt{Vertébré} \sqcup \texttt{Invertébré} \tag{2.4}$$

La **conjonction** (ou *intersection*) de A et de B, notée $A \sqcap B$, est un concept dont les instances sont tous les individus qui sont à la fois instances de A et instances de B. Cela

permet d'exprimer des axiomes de la forme :

$$\texttt{Mère} \equiv \texttt{Parent} \sqcap \texttt{Femme} \tag{2.5}$$

Le **complémentaire** d'un concept A, noté A^{-}, contient l'ensemble des individus qui ne sont pas instances de A. On peut ainsi ré-écrire la relation entre vertébrés et invertébrés à l'aide du complémentaire, en disant qu'un invertébré est un animal qui n'est pas vertébré :

$$\texttt{Invertébré} \equiv \texttt{Animal} \sqcap (\texttt{Vertébré}^{-}) \tag{2.6}$$

Enfin, on souhaite pouvoir relier les concepts à des relations, et pas seulement les concepts entre eux. Pour cela, on dispose de la **restriction existentielle** $\exists R.C$, qui représente l'ensemble des individus reliés à une instance de C par la relation R. Cela permet par exemple de définir les poètes comme l'ensemble des personnes qui écrivent des poèmes :

$$\texttt{Poète} \equiv \texttt{Personne} \sqcap \exists \texttt{écrit}.\texttt{Poème} \tag{2.7}$$

L'utilisation d'un concept singleton dans un quantificateur existentiel, qui donne un axiome de la forme $\exists R.\{e\}$, permet de représenter tous les individus reliés à l'individu e par la relation R, par exemple :

$$\texttt{PoèteCanadien} \equiv \texttt{Poète} \sqcap \exists \texttt{nationalité}.\{\texttt{Canada}\} \tag{2.8}$$

Enfin, si l'on souhaite seulement représenter les individus qui sont sujets d'une relation, sans restriction sur le type de l'objet associé, on peut utiliser le concept universel :

$$\texttt{Possédant} \equiv \exists \texttt{possède}.\top \tag{2.9}$$

Il existe également la **restriction universelle**, notée $\forall R.C$. Un individu x en fait partie à la condition que tous les individus y tels que $x \xrightarrow{R} y$ soient des instances de C (cela inclut donc le cas où x n'est pas sujet de la relation R).

Enfin, on peut également obtenir de nouvelles relations à partir de relations existantes. La **composition de relations**, notée $R_1 \circ R_2$, produit une nouvelle relation qui relie x à y si et seulement si l'on peut trouver z tel que $x \xrightarrow{R_1} z$ et $z \xrightarrow{R_2} y$. L'inverse d'une relation R, notée R^{-} ou parfois R^{-1}, est une relation qui relie x à y si et seulement si y est relié à x par R.

Sémantique d'une ontologie On a présenté ici une vision intuitive de la logique descriptive et de ses types d'axiomes. Derrière cette vision intuitive, il existe une sémantique formelle des axiomes logiques et une notion d'interprétation, dont on présente ici les grandes lignes. Une ontologie correspond à une description partielle du monde ; plusieurs états du monde peuvent correspondre à cette description. Une *interprétation* consiste justement à faire correspondre l'ontologie à un état du monde particulier, en reliant chaque entité apparaissant dans l'ontologie à un individu, chaque concept nommé à un ensemble d'individus et chaque relation à un ensemble de paires d'individus. Les axiomes décrits plus haut possèdent tous une *sémantique* formelle, qui est simplement une traduction mathématique ensembliste de la description intuitive qu'on en a faite. Par exemple, la conjonction de concepts $A \sqcap B$ se traduit sémantiquement par l'intersection des ensembles d'individus associés à A et B. Cette sémantique permet de vérifier, de façon univoque, si un axiome est vérifié dans l'interprétation choisie. Une interprétation *satisfait* l'ontologie $\mathcal{O}$ si tous les axiomes de $\mathcal{O}$ sont vérifiés ; un axiome α est une *conséquence* de cette ontologie s'il est vérifié dans toutes les interprétations qui satisfont $\mathcal{O}$, ce que l'on notera $\mathcal{O} \vdash \alpha$.

On dit qu'une ontologie est *consistante* s'il existe au moins un état du monde (une interprétation) qui satisfait cette ontologie ; dans le cas contraire, l'ontologie ne peut jamais être satisfaite, et présente donc peu d'intérêt. Le *raisonnement* logique consiste principalement à vérifier la consistance d'une ontologie d'une part, et d'autre part à déduire de nouveaux axiomes qui sont des conséquences logiques de l'ontologie de départ.

Le formalisme OWL Les axiomes de la logique descriptive peuvent être représentés informatiquement dans le langage OWL (*Web Ontology Language*) [53], que l'on peut voir comme une extension de RDF pour la représentation d'ontologies. Un axiome de subsumption $A \sqsubset B$ est représenté en OWL par un triplet `A rdfs:subClassOf B` ; l'équivalence $\equiv$, la conjonction $\sqcap$ et la disjonction $\sqcup$ sont représentées respectivement par les relations `owl:equivalentClass`, `owl:intersectionOf` et `owl:unionOf`.

On peut voir un graphe de connaissances comme la combinaison d'un graphe RDF, qui relie les entités entre elles au moyen de relations, et d'une ontologie OWL qui décrit des axiomes reliant les concepts et les relations. Ces deux éléments sont liés entre eux, principalement par la relation `rdf:type` qui relie les entités (RDF) aux concepts (OWL). La tâche d'extraction d'ontologie, qui fait l'objet de ce mémoire, peut dès lors être vue comme la création ou l'enrichissement de l'ontologie OWL à partir du graphe RDF. Cette division RDF/OWL est parfois simplificatrice, mais elle décrit assez fidèlement le cas de DBpedia et le problème que nous traitons dans ce mémoire.

2.2 Extraction automatique d'ontologie et de taxonomie

Le problème d'extraction d'axiomes logiques n'est pas nouveau et prend des formes très variées. Dans le cas général, on cherche soit à construire une ontologie à partir de zéro [28,54–57], soit à étendre une ontologie existante en produisant de nouveaux axiomes [31,58,59]. Un sous-problème fréquent est l'extraction de *taxonomie* [30,35,56,60] : il s'agit alors d'extraire uniquement des axiomes de subsumption, c'est-à-dire ayant la forme $A \sqsubseteq B$, afin d'obtenir une hiérarchie sur les concepts. Si on autorise ces concepts à prendre la forme d'axiomes plus complexes, on parlera d'extraction de taxonomie *expressive*.

On peut distinguer deux grandes familles de méthodes, selon la source des données utilisées : les méthodes basées sur le texte, et les méthodes basées sur un graphe de connaissances. Étant donné la très large disponibilité de corpus textuels étendus (et particulièrement en langue anglaise), les méthodes utilisant le texte sont aujourd'hui les plus fréquentes, et se concentrent particulièrement sur l'extraction de taxonomies. Elles font l'objet de la section 2.2.1. Les méthodes basées sur un graphe ont toutefois leurs avantages : en particulier, elles peuvent fonctionner pour les graphes spécifiques à un domaine et pour lesquels il n'existe pas ou peu de données textuelles. On décrit ces méthodes dans la section 2.2.2. Notons que l'émergence conjointe des plongements lexicaux (pour le texte) et des plongements vectoriels de graphe permet d'envisager une convergence de ces deux familles de méthodes [61].

2.2.1 Extraction d'ontologies à partir de texte

La recherche de taxonomies ou d'ontologies à partir de textes est explorée depuis longtemps [37] ; aujourd'hui, l'abondance actuelle des ressources textuelles [62, 63] combinée à l'émergence des plongements lexicaux en ont fait l'approche dominante pour l'extraction de taxonomies : il nous paraît donc important de la présenter ici.

Dans le contexte textuel, on parle plus volontiers d'*hyperonymie* que de subsumption. Un terme v est un *hyperonyme* d'un terme u s'il est plus général que lui. Dans ce cas, on dit que u est un *hyponyme* de v. Par exemple, «animal» est un hyperonyme de «chat», ce qui est analogue à la relation de subsumption $\texttt{Chat} \sqsubseteq \texttt{Animal}$. La principale différence entre l'hyperonymie et la subsumption est que la première concerne indifféremment les entités et les concepts : ainsi, «Garfield» est un hyponyme de «chat», lui-même hyponyme d'«animal». Dans le cas de la subsumption, on distinguerait pourtant ces deux cas, en disant que «Garfield» est une instance de «chat», et «chat» est subsumé par «animal». Cette distinction entre sous-classes et instances est discutée notamment dans [64]. Pour cette raison, l'hyperonymie est parfois désignée sous le nom de relation *is-a* («est un», en français) : Garfield *est un*

chat, un chat *est un* animal [65]. L'identification d'hyperonymie produit donc une hiérarchie contenant indistinctement des instances et des concepts. Transformer cette hiérarchie en une taxonomie contenant uniquement des concepts exige donc de distinguer automatiquement les instances des concepts, ce qui se fait généralement par des méthodes d'analyses textuelles telles que l'analyse syntaxique et l'étiquetage morpho-syntaxique [66–69].

Les méthodes textuelles pour l'extraction de taxonomie comportent généralement trois étapes : identifier les concepts parmi les mots du vocabulaire, identifier les paires hyponyme-hyperonyme probables, et filtrer ces paires pour obtenir une taxonomie.

Utilisation de motifs lexico-syntaxiques

Une première approche consiste à définir des motifs lexico-syntaxiques, appelés motifs de Hearst [37] (ou *Hearst patterns*, en anglais), pour identifier les relations d'hyperonymie. Par exemple, le motif NP `{,` NP `}* {, } or other` NP (où NP indique l'emplacement de noms) trouve une occurrence dans la phrase «*Bruises, wounds, broken bones or other injuries*» et permet de déduire que «*bruise*», *wound*» et «*broken bone*» sont des hyponymes de «*injury*» [6]. Ces motifs sont généralement définis à la main, et fixés pendant toute la durée de l'extraction. Toutefois, certaines approches apprennent de nouveaux motifs au cours de l'extraction [70, 71]. Appliquées à des corpus textuels étendus, les méthodes basées sur des motifs de Hearst ont généralement une bonne précision [38], mais un mauvais rappel [31], car la langue naturelle présente une grande variabilité et il est difficile de définir des motifs capables de couvrir tous les cas d'hyperonymie.

Les paires hyponyme-hyperonyme extraites sont ensuite filtrées, afin d'éliminer les éventuelles erreurs d'extraction. Dans sa forme la plus simple, le filtrage consiste simplement à compter le nombre d'occurrences de chaque paire, et de définir un seuil minimal. En faisant cela, on pénalise toutefois les mots rares au détriment des mots fréquents : par exemple, il est plus probable d'extraire la paire *(France, pays)* que *(France, république)*, simplement parce que «pays» est un terme plus commun que «république». Pour résoudre ce problème, [72] utilise la PPMI (*Positive Pointwise Mutual Information*, ou information mutuelle ponctuelle positive) entre deux termes. Si $w(x, y)$ désigne le nombre d'occurrences d'une paire (x, y) dans les résultats d'extraction, et W le nombre total de paires extraites, on peut définir une fréquence d'extraction $p(x, y) = w(x, y)/W$ pour la paire (x, y) ; pour le terme x, on définit également une fréquence d'apparition en tant qu'hyponyme : $p^-(x) = \sum_y w(x, y)/W$ et en

6. Exemple tiré de [37].

tant qu'hyperonyme : $p^+(x) = \sum_y w(y,x)/W$. La PPMI est alors calculée selon la formule :

$$PPMI(x,y) = \max\left(0, \log \frac{p(x,y)}{p^-(x)p^+(y)}\right) \tag{2.10}$$

La PPMI permet de résoudre le problème soulevé plus haut : on aura certes toujours p(France, pays) $> p$(France, république), mais également p^+(pays) $> p^+$(république), et donc normalement une PPMI comparable pour les deux paires *(France, pays)* et *(France, république)*.

Pour mieux gérer les mots rares, [38] propose de lisser les valeurs de PPMI obtenues en réduisant la dimension de la matrice de PPMI avec une SVD tronquée. Cela donne une représentation de rang faible des mots, qui permet alors de décider si deux mots sont hyperonymes l'un de l'autre, même si la paire exacte n'a pas été vue sur le corpus d'entraînement. Notons que cette méthode n'inclut pas de mécanisme pour distinguer les concepts des instances.

Enfin, on peut étendre ces méthodes à l'extraction d'ontologies, et pas simplement de taxonomies. OntoCmaps [73] effectue pour cela une analyse syntaxique complète des textes traités, et détecte des motifs dans l'arbre syntaxique obtenu. Parmi ces motifs, on trouve les motifs de Hearst décrits plus haut, qui servent à former des liens taxonomiques, mais également des motifs plus complexes, associés à des règles de transformation, qui permettent de transformer des passages textuels en axiomes logiques.

Méthodes distributionnelles, méthodes à plongement

Les méthodes distributionnelles reposent sur l'«hypothèse d'inclusion distributionnelle» (ou DIH, pour *Distributional Inclusion Hypothesis*) [74] : si le terme A est un hyperonyme du terme B, alors les contextes dans lesquels B apparaît forment un sous-ensemble des contextes dans lesquels A apparaît. Le contexte d'un mot peut être les $2N$ mots adjacents au sein du corpus (N mots précédents et N mots suivants), ou les éléments voisins (parents ou enfants) au sein d'un arbre de dépendance [**?**]. Intuitivement, l'hypothèse DIH signifie que l'on peut remplacer les occurrences de B par A tout en gardant un texte valide.

Pour mettre en pratique cette idée, les méthodes distributionnelles construisent une représentation vectorielle des termes, ou plutôt des contextes des termes ; on peut alors évaluer si un terme A est un hyperonyme d'un terme B en utilisant une fonction de classification, qui combine les représentations de A et de B et prédit le résultat. Avant de décrire ces méthodes, il nous faut présenter une étape préalable : l'extraction des concepts.

Extraction des concepts L'extraction de taxonomie à partir de texte suppose en effet deux étapes : d'abord, identifier, parmi tous les termes du corpus, ceux qui constituent des concepts ; ensuite, identifier les relations d'hyperonymie entre ces concepts. Les méthodes basées sur des motifs de Hearst effectuent simultanément ces deux étapes ; ce n'est pas le cas des méthodes distributionnelles, qui supposent une étape préalable d'*extraction des concepts*. Dans certains cas, les concepts sont considérés comme des données d'entrée. Autrement, il est nécessaire de filtrer le corpus, puis d'identifier les concepts : cela consiste généralement à ne garder que les noms, à l'aide d'un étiquetage morpho-syntaxique, puis à les filtrer à l'aide de règles linguistiques ou statistiques de façon à en exclure les instances [75].

Approches distributionnelles Les approches distributionnelles classiques ne sont pas supervisées [76], et se fondent sur des représentations creuses des termes. Pour cela, on définit une notion de *co-occurrence*, variable d'un modèle à l'autre : la co-occurrence peut être l'apparition dans une même fenêtre de contexte, ou le fait d'être sujet et objet d'un même verbe dans un arbre de dépendance. Pour un vocabulaire de N termes $\{x_1, \ldots, x_N\}$, on peut alors représenter un terme t par un vecteur $\mathbf{t}$ à N dimensions, tel que $\mathbf{t}_i$ contient la fréquence de co-occurrence entre t et x_i. Pour évaluer si y est un hyperonyme de x, on peut alors comparer $\mathbf{y}$ et $\mathbf{x}$. Avec des mesures de similarité usuelles (indice de Jaccard, divergence de Jensen-Shannon ou autre), il est difficile de distinguer l'hyperonymyie d'autres formes de relations entre les mots (par exemple, la synonymie ou la co-hyponymie, c'est-à-dire le fait d'avoir un hyperonyme commun) [77]. C'est pourquoi des mesures spéciiques à la la recherche de l'hyperonymie ont été proposées, telles que la métrique *WeedsPrec* [76] :

$$WeedsPrec(x, y) = \frac{\sum\limits_{\substack{i=1,\ldots,N \\ \mathbf{y}_i > 0}} \mathbf{x}_i}{\sum\limits_{i=1,\ldots,N} \mathbf{x}_i} \tag{2.11}$$

On retrouve dans cette formule l'hypothèse DIH : si les contextes de x forment un sous-ensemble des contextes de y, alors $\mathbf{y}_i$ est positif chaque fois que $\mathbf{x}_i$ est positif, et on a donc $WeedsPrec(x, y) = 1$. À l'inverse, si les contextes de x et y sont disjoints, on a $WeedsPrec(x, y) = 0$. On peut alors estimer que y est un hyperonyme de x si le score $WeedsPrec(x, y)$ est supérieur à un certain seuil.

Utilisation de plongements lexicaux Depuis l'introduction de Word2Vec [40], la recherche s'est tournée vers l'exploitation de plongements lexicaux pour représenter vectoriellement les termes. Les plongements lexicaux sont des représentations denses des mots, géné-

ralement obtenus à partir de très grands corpus et qui intègrent la sémantique des mots sous forme géométrique. Deux approches sont possibles : soit utiliser des plongements lexicaux généralistes [33–36], soit définir un modèle de plongement lexical spécifique à l'extraction d'hyperonymes [60, 78–82].

Plongements génériques : Dans le premier cas, les plongements utilisés sont génériques – typiquement des variantes de Word2Vec. Ils sont donc d'abord conçus pour représenter la *similarité* entre les mots (deux mots apparaissants dans les mêmes contextes auront des plongements géométriquement proches), sans notion de hiérarchie ou d'hyperonymie. Pour identifier l'hyperonymie, [33] utilise une méthode supervisée : l'idée est d'apprendre une relation linéaire entre les plongements des hyponymes et ceux des hyperonymes. On dispose de paires d'hyponymes-hyperonymes $\{(x_k, y_k)\}$, d'un plongement lexical $\mathbf{w}$ pour chaque mot w du vocabulaire, et on souhaite trouver une matrice $\mathbf{M}$ telle que $\mathbf{M}\mathbf{x}_k \approx \mathbf{y}_k$, ce qui correspond à un problème classique de régression linéaire. En pratique, une unique relation linéaire est insuffisante pour couvrir la variété des paires d'hyperonymes, et on utilise donc une relation linéaire par morceaux : chaque paire (x_k, y_k) est représentée par le vecteur $\mathbf{u}_k = \mathbf{x}_k - \mathbf{y}_k$, les vecteurs $\mathbf{u}_k$ sont alors regroupés en m groupes $C_1, \ldots, C_m$ avec l'algorithme des k-moyennes, et une régression linéaire $\mathbf{M}_i$ est apprise sur chaque groupe C_i. Les k-moyennes produisent un partitionnement de l'espace : pour classifier une paire inconnue (x, y), il suffit donc de trouver le groupe C_i auquel la paire appartient, puis de mesurer la distance entre $\mathbf{M}_i\mathbf{x}$ et $\mathbf{y}$: si elle est inférieure à un certain seuil, alors y est un hyperonyme de x.

Plongements spécifiques : Lorsqu'on choisit, au contraire, de définir un modèle de plongement lexical spécifique à la recherche d'hyperonymes, l'enjeu est de trouver une géométrie capable de représenter la notion de hiérarchie. La méthode du *Dynamic-weighting Neural Network* [81] utilise en entrée une liste de paires hyponymes-hyperonymes et un corpus textuel étendu : chaque fois qu'un hyponyme et un hyperonyme sont trouvés dans la même phrase du corpus, on extrait un triplet d'entraînement comportant lesdits hyponyme et hyperonyme, ainsi que les mots contextuels, c'est-à-dire tous les mots situés entre l'hyponyme et l'hyperonyme. On fournit alors ces mots contextuels et l'hyponyme à un réseau de neurones, dont l'objectif est de prédire l'hyperonyme. Comme dans Word2Vec, les poids de ce réseau servent alors de plongements lexicaux. HyperVec [78] utilise également une liste de paires hyponymes-hyperonymes et un corpus textuel pour apprendre des plongements lexicaux présentant les caractéristiques suivantes : (a) un hyponyme et un hyperonyme ont des plongements proches au sens de la similarité cosinus, et (b) le plongement d'un hyponyme a une norme euclidienne inférieure à ses hyperonymes.

Une approche classique consiste à choisir un espace D pour les plongements (typiquement

l'espace euclidien à d dimensions) et un ordre partiel $\preceq$ sur cet espace, puis à entraîner les plongements lexicaux tels que $\mathbf{x} \preceq \mathbf{y}$ si y est un hyperonyme de x. Si l'on adopte ce paradigme, il devient intéressant de raisonner avec des *zones d'hyponymie* : la zone d'hyponymie $Z(\mathbf{y})$ associée à un plongement $\mathbf{y}$ est l'ensemble des points $\mathbf{x}$ de D tels que $\mathbf{x} \preceq \mathbf{y}$. Dès lors, un terme x est un hyponyme de y si son plongement est contenu dans la zone d'hyponymie de y. De plus, par transitivité de $\preceq$, on a une équivalence entre $\mathbf{x} \preceq \mathbf{y}$ et $Z(\mathbf{x}) \subseteq Z(\mathbf{y})$. Ce cadre permet de visualiser et de comparer plusieurs géométries de plongement : on donne trois exemples de géométries, et on illustre les zones d'hyponymies associées à ces trois choix à la figure 2.1.

Le modèle des *Order Embeddings* [82] utilise comme ordre partiel sur $\mathbb{R}^d_+$ la relation :

$$\mathbf{x} \preceq \mathbf{y} \iff \forall i = 1, \dots, d, x_i \geq y_i \tag{2.12}$$

En particulier, l'élément racine de la taxonomie aura un plongement nul. En dimension 2, la zone d'hyponymie d'un point A est le quart de plan supérieur droit de A, c'est-à-dire l'ensemble des points situés en haut à droite de A (figure 2.1a). Ce choix de $\preceq$ a un inconvénient : il interdit d'avoir des classes disjointes. En effet, quels que soient les plongements $\mathbf{x}$ et $\mathbf{y}$ considérés, leurs zones d'hyponymies auront une intersection non-vide.

Pour dépasser ce problème, le modèle des *Box-Lattice Embeddings* [83] se place dans l'hypercube $[0, 1]^d$, c'est-à-dire l'ensemble des vecteurs de $\mathbb{R}^d$ dont toutes les coordonnées sont comprises entre 0 et 1, et représente chaque terme x par deux plongements $\mathbf{x}^m$ et $\mathbf{x}^M$, qui vérifient $x_i^m < x_i^M$ pour tous $i = 1, \dots, d$. La relation d'ordre entre deux paires de plongements s'écrit :

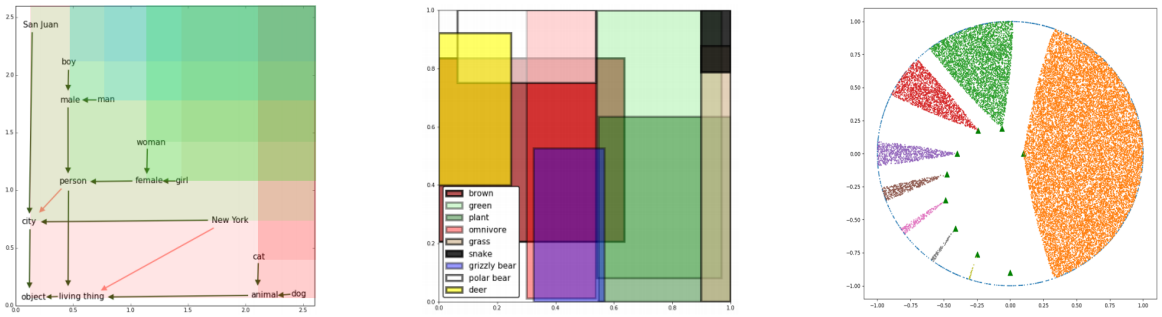
$$(\mathbf{x}^m, \mathbf{x}^M) \preceq (\mathbf{y}^m, \mathbf{y}^M) \iff \forall i = 1, \dots, d, y_i^m \leq x_i^m < x_i^M \leq x_i^M \tag{2.13}$$

Ce choix de relation d'ordre consiste en fait à choisir comme zones d'hyponymie des pavés de $\mathbb{R}^d$:

$$Z(\mathbf{x}) = \{\mathbf{u} \in [0, 1]^d : \forall i = 1, \dots, d, x_i^m \leq u_i \leq x_i^M\} \tag{2.14}$$

À nouveau, la relation d'hyperonymie entre deux termes se traduit par une relation d'inclusion entre leurs pavés : y est un hyperonyme de x si et seulement si $Z(\mathbf{x}) \subseteq Z(\mathbf{y})$. La figure 2.1b en donne une illustration.

Enfin, plusieurs travaux ont suggéré de quitter l'espace euclidien et de privilégier un espace hyperbolique [60,79,84–86], qui offre une représentation naturelle des relations hiérarchiques. Comme montré par [86], un plongement $\mathbf{u}$ dans une boule de Poincaré induit naturellement une zone d'hyponymie conique, dont la largeur dépend de la norme de $\mathbf{u}$. Des exemples de ces «cônes d'hyponymie» sont donnés à la figure 2.1c.



(a) *Order Embeddings* (b) *Box-Lattice Embeddings* (c) Plongements de Poincaré

Figure 2.1 Zones d'hyponymies pour différents modèles de plongement lexical utilisés pour la détection d'hyperonymes. La zone d'hyponymie associée à un concept contient tous les hyponymes de ce concept. Les illustrations proviennent, de gauche à droite, de [82], [83] et [86].

Regroupement hiérarchique sur les plongements lexicaux

Parmi les méthodes qui reposent sur des plongements lexicaux, on s'intéresse ici à celles qui réalisent un regroupement ou partitionnement des données (en anglais : *clustering*).

Un algorithme de regroupement hiérarchique aboutit à la même structure d'arbre qu'une taxonomie : une fois que l'on dispose de représentations vectorielles pour les concepts à hiérarchiser, il est donc naturel d'appliquer un tel algorithme pour obtenir une taxonomie. Plusieurs méthodes ont été proposées dans ce sens. [34] propose un partitionnement divisif de plongements lexicaux : on part de l'ensemble de tous les concepts, représentés par des plongements lexicaux génériques, et on y applique la méthode des k-moyennes. Le nombre de clusters k est obtenu de manière non-supervisée, en combinant plusieurs métriques classiques telles que le critère du coude (en anglais : *elbow method*), la statistique du *gap* et l'examen de la *silhouette* (voir [87, p. 126–130] pour une discussion sur ces métriques). Chaque cluster est alors étiqueté à l'aide d'une combinaison de motifs de Hearst, d'analyse statistique et de supervision humaine. Cette nouvelle étiquette est soit l'un des concepts contenus dans le cluster, soit un nouveau nom fourni par un annotateur humain. Puisque la méthode demande une supervision humaine, elle ne résout pas la question cruciale de l'étiquetage automatique des clusters, que nous examinons dans ce mémoire, dans les sections 4.2.2 (cas non-expressif) et 5.2.3 (cas expressif). TaxoGen [42] propose une méthode pour extraire une taxonomie à partir d'un regroupement hiérarchique de plongements vectoriels. TaxoGen repose sur un partitionnement adaptatif des données, capable de décider, pour chaque cluster, si une instance doit être propagée vers les sous-clusters (c'est-à-dire qu'elle peut être rattachée à un concept plus spécifique que le concept associé au cluster courant), ou si elle doit rester dans

le cluster courant (ce qui signifie qu'elle n'est instance d'aucun concept plus spécifique). En revanche, TaxoGen ne propose pas de mécanisme pour associer un nom à chaque cluster : le résultat est donc une hiérarchie de groupes d'instances, et pas un ensemble d'axiomes de subsumption. À nouveau, la question de l'étiquetage automatique des clusters n'est pas résolue. VDGEC [88] extrait de corpus textuels un graphe de co-occurrence de concepts, qui relie entre eux les concepts apparaissants fréquemment ensemble. Ce graphe est alors utilisé pour calculer des représentations vectorielles des concepts ; ces vecteurs sont ensuite regroupés hiérarchiquement pour donner une taxonomie. La différence essentielle entre la méthode VDGEC et la nôtre est la nature des plongements vectoriels manipulés : dans VDGEC, on utilise directement les *concepts*, alors que nous utilisons les *instances*. Manipuler les instances ajoute une complexité supplémentaire, puisque les données à manipuler sont plus nombreuses, et qu'il est nécessaire de concevoir un mécanisme pour relier chaque concept à un groupe d'instances ; en contrepartie, cela autorise une plus grande souplesse, particulièrement en permettant d'identifier de nouveaux groupes d'instances pertinents, et donc de nouveaux concepts. Nous démontrons cette possibilité dans le chapitre 5.

2.2.2 Extraction d'ontologies à partir d'un graphe

Ici, on décrit les approches existantes pour extraire ou étendre une ontologie à partir d'un graphe de connaissances. Par rapport à l'utilisation de corpus textuels, l'utilisation d'un graphe permet d'éviter les ambiguïtés syntaxiques ou lexicales, puisque l'on travaille directement à partir de données structurées. On cherche généralement à exprimer les axiomes extraits dans le formalisme de la logique descriptive ; certaines des méthodes présentées ici utilisent d'autres systèmes logiques (règles d'association, programme logique) ; sous réserve que ces systèmes soient moins expressifs que la logique descriptive $\mathcal{SROIQ}$, le passage de l'un à l'autre ne pose pas de problème.

Une première approche consiste à utiliser des techniques issues de l'intelligence artificielle symbolique pour effectuer des raisonnements à partir de règles existantes et déduire de nouveaux axiomes. D'autres méthodes identifient au contraire des régularités dans le graphe, soit par un examen statistique (fréquences relatives des relations et des concepts, calculs de co-occurrences), soit en plongeant les éléments du graphe dans un espace vectoriel qui reflète la structure du graphe. Ces différentes méthodes sont décrites dans les sections suivantes.

Méthodes symboliques

Les méthodes symboliques constituent une première famille d'approches capables de construire de nouveaux axiomes à partir d'axiomes existants. On désigne sous ce nom toutes les mé-

thodes qui manipulent directement des règles logiques. Ces méthodes sont pour la plupart mal adaptées aux graphes de connaissances créés automatiquement ou semi-automatiquement, et ce pour deux raisons essentielles : d'une part, ces méthodes demandent beaucoup de ressources, et ne fonctionnent que sur des ontologies réduites ; d'autre part, elles sont incapables de gérer correctement l'incomplétude et le bruit qui caractérise ces graphes. On en présente succinctement trois grandes familles dans les paragraphes qui suivent.

Les **raisonneurs**, comme Fact++ [89] ou Hermit [90], appliquent des règles de démonstration pour déduire des axiomes nouveaux. Un exemple de ces règles de démonstration est le *modus ponens*, qui consiste simplement en $P \wedge (P \implies Q) \vdash Q$: si P est vraie et P implique Q, alors Q est vraie. Ces raisonneurs peuvent être utiles pour vérifier si une ontologie donnée est consistante, mais ils ne peuvent pas réellement être utilisés pour l'extraction de taxonomie : d'une part, ils nécessitent une ontologie de départ ; d'autre part, ce sont des méthodes *déductives* et donc incapables d'inférer de nouvelles règles à partir des données.

Une autre approche symbolique [91] est la **programmation logique inductive** (ou ILP, pour *Inductive Logic Programming*) [92, 93] : il s'agit cette fois d'une méthode *inductive*, c'est-à-dire qui produit de nouveaux axiomes à partir d'exemples. La programmation logique inductive part d'une série d'axiomes (la *théorie préalable*, éventuellement vide) et d'exemples positifs et négatifs ; elle cherche à produire de nouveaux axiomes qui sont vérifiés par les exemples positifs mais pas par les exemples négatifs. De manière informelle et à titre d'exemple, on se donne une propriété P, une théorie préalable vide, un ensemble d'exemples positifs $E^+ = \{P(0), P(2), P(4), P(8)\}$ et $E^- = \{P(1), P(3), P(5)\}$ – autrement dit, $0, 2, 4, 8$ vérifient P et $1, 3, 5$ ne la vérifient pas [7]. Alors, un exemple de théorie induite est donné par :

$$\text{Si } P(x) \text{ est vraie, alors } P(x+2) \text{ est vraie.} \tag{2.15}$$

$$P(0) \text{ est vraie.} \tag{2.16}$$

Cette théorie ne pourrait être obtenue par un raisonneur, car elle n'est pas une conséquence logique de E^+ et de E^- : il s'agit simplement d'une théorie plausible étant donné les faits observés. Pour obtenir des théories induites pertinentes, l'ILP nécessite donc un choix judicieux d'exemples positifs et négatifs. Pour obtenir une théorie induite, on part d'une théorie initiale, possiblement vide, puis on l'améliore itérativement : si elle est trop générale, on la spécialise ; si elle est trop spécifique, on la généralise [92, p. 169]. En pratique, et comme pour les raisonneurs, les méthodes d'ILP passent difficilement à l'échelle pour les graphes de connaissances de grande taille [94], même si des heuristiques moins coûteuses en calcul ont

7. Exemple tiré de [92].

été proposées [95]. De plus, les données d'un graphe réel sont incomplètes et bruitées, une situation pas ou mal gérée par la programmation logique.

Dans notre mémoire (section 5.2.3), nous proposons une méthode d'extraction d'axiomes dont le point de départ est proche de l'ILP : on se donne en effet un ensemble d'exemples positifs et négatifs, et on cherche à trouver des axiomes décrivant l'un et pas l'autre. On résout le problème du passage à l'échelle en restreignant drastiquement l'espace de recherche grâce à l'utilisation de plongements vectoriels et du tirage aléatoire (*sampling*). On reprend l'idée d'une amélioration itérative des axiomes par généralisation/spécialisation, mais on gère le bruit et l'incomplétude des données en utilisant des méthodes statistiques plutôt que symboliques.

Une troisième approche symbolique repose sur l'**analyse formelle de concept** (AFC, en anglais : *Formal Concept Analysis*), introduite mathématiquement par [96] et appliquée à la construction d'ontologie par [26]. L'AFC considère un ensemble d'entités possédant des attributs, et cherche à produire une hiérarchie de *concepts* : dans ce contexte, un concept est une paire (A, B) où A est un ensemble d'entités et B un ensemble d'attributs tels que B est l'ensemble des attributs vérifiés par tous les éléments de A, et inversement A est l'ensemble des entités qui vérifient tous les attributs de B. L'ensemble des concepts est muni d'une relation d'ordre $\sqsubseteq$: on a $(A, B) \sqsubseteq (A', B')$ si $A \subseteq A'$ (ou, de manière équivalente, $B' \subseteq B$), analogue à la subsumption décrite plus haut. Munie de cette relation d'ordre, l'AFC induit naturellement une hiérarchie sur les concepts : l'enjeu est donc de construire une série de concepts pertinents. Pour obtenir ces concepts, on utilise une règle de dérivation : si A est un ensemble d'entités, on désigne par A' l'ensemble des attributs communs à tous les éléments de A ; de la même manière, si B est un ensemble d'attributs, on désigne par B' l'ensemble des entités qui possèdent tous les attributs de B. En partant d'une entité x, on peut ainsi obtenir $\{x\}'$ l'ensemble des attributs de x, puis $\{x\}''$ l'ensemble des entités qui ont exactement les mêmes attributs que x : la paire $(\{x\}'', \{x\}')$ constitue un premier concept. En ajoutant une nouvelle entité y à $\{x\}''$, on obtient un nouvel ensemble d'entités $\{x\}'' \cup \{y\}$, que l'on peut à nouveau dériver en un ensemble d'attributs $(\{x\}'' \cup \{y\})'$, lui-même dérivé pour former un second concept $((\{x\}'' \cup \{y\})'', (\{x\}'' \cup \{y\})')$; et ainsi de suite. Par ailleurs, un ensemble de concepts $C_1, \ldots, C_N$ possède toujours un *sous-concept commun maximal*, c'est-à-dire le plus grand concept S qui vérifie $\forall i, S \sqsubseteq C_i$. De même, chaque ensemble de concepts possède un *superconcept commun minimal*, c'est-à-dire le plus petit concept S tel que $\forall i, C_i \sqsubseteq S$. De plus, il existe un concept minimal, qui contient tous les attributs (et potentiellement aucune entité), et un concept maximal, qui contient toutes les entités (et potentiellement aucun attribut). À partir de ces notions, de nombreux algorithmes ont été proposés pour l'identification et la hiérarchisation de concepts [97–99] ; toutefois, la conception d'algorithmes

efficaces et applicables à grande échelle demeure un problème ouvert [27, 100]. L'AFC peut être étendue aux relations binaires, c'est-à-dire à des triplets (h, r, t) où h, t sont des entités reliées par la relation r : on parle alors d'**analyse relationnelle de concept** (ARC, ou *Relational Concept Analysis* en anglais) [101].

Méthodes statistiques

Les méthodes statistiques visent à extraire des règles logiques en observant des régularités statistiques dans un graphe de connaissances : par exemple, en comptant les co-occurrences de certaines classes ou de certaines relations dans le voisinage d'une entité.

On présente ici la méthode de *Statistical Schema Induction* (SSI, en français : induction statistique d'ontologie) [28], qui se rapproche du travail que nous présentons à la section 5.2.3. Cette approche repose sur l'apprentissage de *règles d'association* [102], qui sont essentiellement des implications logiques inférées d'après des co-occurrences observées dans les données. Une règle d'association s'écrit sous la forme $X \to Y$. On peut par exemple imaginer une règle $\{\texttt{Personne}, \exists\texttt{écrit.Poème}\} \to \texttt{Poet}$: si une même entité est de type `Personne` et vérifie l'axiome $\exists\texttt{écrit.Poème}$, alors cette entité est probablement de type `Poète`.

Pour extraire ces règles, on se donne un *tableau de transactions* $\mathbf{T}$, c'est-à-dire une matrice dont les lignes représentent les entités $e_1, \ldots, e_n$ du graphe, et les colonnes des prédicats logiques $P_1, \ldots, P_m$. Ces prédicats incluent les concepts nommés C (comme `Artist`, `Person`, etc.) et différentes formes de restrictions existentielles comme $\exists r.C$ ou $\exists r.\top$. On note $\mathcal{E}$ l'ensemble des entités, et $\mathcal{P}$ l'ensemble des prédicats. La coordonnée (i, j) de $\mathbf{T}$ contient 1 si e_i vérifie le prédicat P_j, et 0 sinon. Soit $X \subseteq \mathcal{P}$ un ensemble de prédicats, alors le *support* de X est l'ensemble des entités qui vérifient tous ces prédicats simultanément :

$$\mathrm{Supp}(X) = \frac{|\{e \in \mathcal{E} : \forall P \in X, P(e)\}|}{|\mathcal{E}|} \tag{2.17}$$

L'indice de confiance associée à une règle $X \to Y$ se calcule alors par :

$$\mathrm{Conf}(X \to Y) = \frac{\mathrm{Supp}(X \cup Y)}{\mathrm{Supp}(X)} \tag{2.18}$$

Au numérateur, on compte le nombre d'entités qui vérifient à la fois les prédicats de X et de Y, que l'on rapporte au nombre d'entités qui vérifient les prédicats de X (notons que l'on a nécessairement $\mathrm{Supp}(X \cup Y) < \mathrm{Supp}(X)$, car une entité qui vérifie à la fois les prédicats de X et de Y vérifie *a fortiori* ceux de Y). Un indice égal à 1 signifie que toutes les entités qui vérifient X vérifient également Y, et donc que la règle $X \to Y$ est parfaitement validée par les

données ; dans la pratique, on définit plutôt un seuil τ au-delà duquel la règle d'association est considérée comme valide.

Le nombre de combinaisons de prédicats possibles croît exponentiellement : les énumérer est donc impossible, même pour des jeux de données réduits. Plusieurs heuristiques ont été développées, dont la plus connue est Apriori [103], elle-même améliorée à plusieurs reprises, par exemple en minimisant le nombre de lectures du graphe [104] ou en implémentant un mécanisme de parallélisme avec MapReduce [105].

Une fois les règles d'association extraites, on peut les filtrer et les convertir en axiomes de subsumption. Cette approche produit des axiomes de subsumption isolés les uns des autres, alors qu'on souhaite généralement obtenir une hiérarchie contenant tous les concepts nommés et, éventuellement, de nouveaux concepts expressifs. Dans notre propre méthode, nous reprenons l'idée d'extraire des axiomes en examinant statistiquement un tableau de co-occurrence entités-axiomes. Notre méthode propose toutefois deux différences essentielles : d'une part, elle s'applique sur un groupe d'entités restreint, qui a été défini de manière non supervisée en se basant sur des plongements vectoriels ; d'autre part, l'extraction d'axiomes sert uniquement à *étiqueter* des groupes d'entités, et non pas à obtenir des liens de subsumption. Dans notre cas, les relations de subsumptions sont identifiées par regroupement hiérarchique.

Utilisation de plongements vectoriels de graphe pour l'extraction d'ontologies

Un modèle de plongement vectoriel de graphe est un modèle capable de construire des représentations vectorielles des entités et des relations d'un graphe de connaissances. Il s'agit d'une approche relativement récente qui présente une analogie forte avec les plongements lexicaux utilisés en traitement de la langue. Les modèles de plongement constituent un aspect important de notre travail, et sont décrits en détail dans le chapitre 3 – on renvoie donc le lecteur à ce chapitre pour une plus ample présentation. Dans cette section, il suffit de savoir qu'un modèle de plongement associe à chaque entité e du graphe un vecteur $\mathbf{e} \in \mathbb{R}^d$ (plus rarement $\mathbb{C}^d$), de telle sorte que deux entités impliquées dans des relations similaires aient des plongements géométriquement proches.

Les plongements vectoriels de graphes de connaissances sont très utilisés pour découvrir des régularités au niveau des instances (ce qui permet notamment d'évaluer la validité d'un triplet inconnu ou de prédire de nouveaux triplets), mais ils ont encore été peu exploités pour la manipulation de concepts ou l'extraction de taxonomie ou d'ontologie. Dans cette optique, [106] étudie les liens entre les axiomes logiques et la géométrie de différents modèles de plongement, dans le but de pouvoir traduire un axiome sous forme de plongement vectoriel sans perte d'information ou d'expressivité. Une idée qui commence à se développer consiste à

utiliser les plongements vectoriels pour évaluer *a priori* la plausibilité d'un axiome logique, ce qui permet de réduire l'espace de recherche. Dans cette lignée, IterE [29] apprend conjointement des plongements vectoriels et des règles logiques, les deux se renforçant mutuellement. Des plongements sont initialement appris à partir du graphe de connaissances ; ensuite, une marche aléatoire dans le graphe permet de générer des axiomes potentiels. Ces axiomes se voient alors attribuer un score de vraisemblance, grâce à une table de correspondance entre axiomes logiques et plongements. Par exemple, l'axiome $A \equiv B$ est traduit géométriquement par $\mathbf{A} = \mathbf{B}$, où $\mathbf{A}$ et $\mathbf{B}$ sont les plongements vectoriels des concepts A et B. Le score de vraisemblance de $A \equiv B$ est alors défini comme la similarité géométrique des plongements de A et de B : dans le cas idéal où $\mathbf{A} = \mathbf{B}$, la similarité est maximale et l'axiome obtient un score de 1. Plus on s'éloigne de ce cas idéal, plus le score diminue. Cette approche permet une évaluation peu coûteuse des axiomes candidats. Enfin, les axiomes ayant un score de vraisemblance suffisamment haut servent à générer de nouveaux triplets d'entraînement : on peut alors répéter ces trois étapes pour améliorer itérativement les plongements et les axiomes. Cette méthode est uniquement évaluée sur des graphes de petite taille (moins de 50 000 entités) ; savoir si elle est applicable à plus large échelle reste une question ouverte. RLvLR [107] repose sur un principe similaire. Il cherche à extraire des règles logiques de la forme $x \xrightarrow{r_1} z_1 \xrightarrow{r_2} z_2 \xrightarrow{r_3} \ldots \xrightarrow{r_k} y \implies x \xrightarrow{r} y$. entre les relations. Pour une relation r donnée, on récupère les triplets impliquant r, puis on effectue des marches aléatoires autour de ces triplets pour générer des règles candidates. Ces règles sont alors filtrées en exploitant la similarité des plongements des relations impliquées dans chaque règle. Cela permet de restreindre l'espace de recherche et donc d'appliquer la méthode à des graphes de grande taille comme Wikidata ou DBpedia. En contrepartie, l'expressivité des axiomes extraits est limitée ; de plus, les axiomes extraits ne sont pas organisés entre eux sous forme de taxonomie.

Plus proche de notre propre méthode, **TIEmb** [30] extrait une taxonomie (non-expressive) à partir de plongements vectoriels d'entités. Comme notre méthode, TIEmb considère comme données d'entrée un ensemble de paires $\{(\mathbf{e}_i, t_i)\}_{i=1,\ldots,n}$, où $\mathbf{e}_i \in \mathbb{R}^d$ est le plongement vectoriel de l'entité e_i, et t_i est son type, et produit en sortie une taxonomie, c'est-à-dire un ensemble d'axiomes de subsumption de la forme $t_i \sqsubset t_j$. Pour cela, chaque type t est représenté par un sphère S_t à d dimensions, dont le centre est un vecteur $\mathbf{c}_t$ défini par :

$$\mathbf{c}_t = \frac{1}{n_t} \sum_{\substack{i \in [1,n] \\ t_i = t}} \mathbf{e}_i \tag{2.19}$$

Avec $n_t = |\{i \in [1, n] : t_i = t\}|$ le nombre d'entités de type t dans les données. $\mathbf{c}_t$ représente le centroïde des instances de t.

Le rayon r_t de S_t est lui défini comme la distance moyenne entre les instances de t et le centroïde $\mathbf{c}_t$:

$$r_t = \frac{1}{n_t} \sum_{\substack{i \in [1,n] \\ t_i = t}} \|\mathbf{e}_i - \mathbf{c}_t\|_2 \tag{2.20}$$

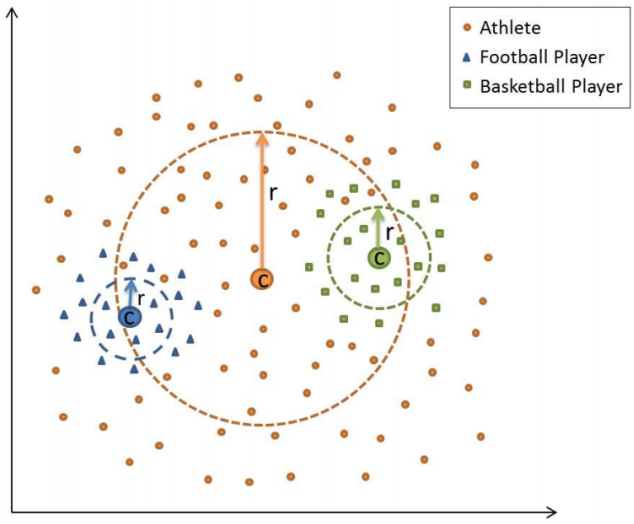


Figure 2.2 Plongements et sphères associées à trois classes dans le cadre de la méthode TIEmb, dans le cas bidimensionnel. Illustration tirée de [30].

Dès lors, l'axiome $t \sqsubset t'$ est prédit si et seulement si $S_t \subset S_{t'}$: la sphère d'un supertype englobe les sphères de ses sous-types, comme illustré à la figure 2.2. Enfin, une étape finale de filtrage est appliquée pour enlever les cycles et obtenir une taxonomie valide.

TIEmb présente l'avantage d'être conceptuellement simple, et peu coûteux en calcul, ce qui permet de l'appliquer à très grande échelle. Une différence cruciale avec notre approche est que le regroupement des entités au sein des sphères est entièrement supervisé, puisqu'il repose sur les types t_i contenus dans les données, et pas sur la géométrie des plongements : TIEmb ne permet donc pas d'identifier de nouveaux groupes cohérents d'entités et donc de détecter de nouvelles classes. Dans le chapitre 4, on compare notre méthode d'extraction de taxonomie non-expressive à TIEmb ; dans le chapitre 5, on illustre les possibilités offertes par le regroupement non-supervisé en extrayant une taxonomie expressive de DBpedia.

Résumé On a présenté un aperçu de différentes approches pour extraire automatiquement des taxonomies ou des ontologies. Parmi ces méthodes, celles qui manipulent directement un graphe ou des textes se heurtent à deux difficultés : d'une part, les données sont *sparse* (parfois traduit par «disséminées» ou «creuses» en français), c'est-à-dire que de nombreux

éléments (mots ou entités) sont rarement observés ; d'autre part, les données sont *discrètes*, et généraliser les observations faites sur un élément à d'autres éléments est compliqué.

Les plongements vectoriels permettent de surmonter ces deux difficultés, et ont été appliqués à l'extraction d'ontologies avec une grande variété de méthodes et d'objectifs. Toutefois, aucune des méthodes présentées ne permet d'extraire une ontologie complète et de haute qualité sans supervision humaine. Les modèles capables d'induction complexe ne sont généralement pas applicables sur des graphes de grande taille comme DBpedia. À l'inverse, les modèles capables de passer à l'échelle se contentent de hiérarchiser des classes existantes, et sont incapables d'identifier de nouvelles classes à partir des données. Dans la suite, nous proposons une méthode capable d'extraire une taxonomie expressive à grande échelle, en combinant plongements vectoriels de graphe, regroupement hiérarchique non-supervisé et extraction d'axiomes à partir de statistiques sur le graphe. Au préalable, nous présentons en détail les modèles de plongement vectoriel dans le chapitre 3.

CHAPITRE 3 PLONGEMENTS VECTORIELS DE GRAPHES DE CONNAISSANCES

Ce chapitre introduit les modèles de plongement vectoriels, utilisés pour représenter les entités et les relations d'un graphe de connaissances sous forme vectorielle. Ces représentations vectorielles, dont la géométrie intègre une partie de l'information sémantique contenue dans le graphe, permettent l'application d'outils d'apprentissage automatique, comme la complétion de triplets [108], la détection de triplets invalides [109], mais aussi l'extraction automatique de taxonomie qui fait l'objet de ce mémoire [30, 42]. Dans ce chapitre, on présente dans le détail les modèles utilisés dans la suite de notre travail, en distinguant deux façons d'envisager les modèles de plongement – l'une géométrique, l'autre algébrique. On s'attache notamment à mettre en évidence les hypothèses de chaque modèle, les types de relations qu'il s'attache à modéliser, et les implications des choix effectués pour la modélisation.

Enfin, on présente une nouvelle tâche sur laquelle évaluer un modèle de plongement de graphe. Cette tâche vise explicitement à mesurer la capacité d'un modèle à envoyer des entités appartenant à une même classe dans une même région de l'espace – une propriété essentielle pour pouvoir extraire des informations taxonomiques à partir des plongements vectoriels, et un préalable aux méthodes présentées dans les chapitres suivants de ce mémoire.

3.1 Généralités

3.1.1 Introduction et motivation

Dans toute cette section, on considérera un graphe de connaissances $\mathcal{KG} \subseteq \mathcal{E} \times \mathcal{R} \times \mathcal{E}$ dont on notera $n = |\mathcal{E}|$ le nombre d'entités, et $m = |\mathcal{R}|$ le nombre de relations.

Représentation des entités et des relations d'un graphe

La représentation des données est une question cruciale en apprentissage automatique ; or la représentation standard des entités dans un graphe de connaissances est très inadaptée à la plupart des algorithmes. En effet, dans un graphe de connaissances RDF, une ressource est représentée par un identifiant unique, l'IRI (*International Resource Identifier*). Une IRI est purement symbolique et ne contient pas nécessairement d'information sur la ressource qu'elle représente. Cette représentation discrète ne permet pas, par exemple, de comparer deux entités entre elles : pour deux entités e et e', la seule donnée de leurs IRI respectives ne permet pas de déduire si ces entités sont proches l'une de l'autre.

Certains auteurs combinent IRI et plongements lexicaux pour obtenir des représentations vectorielles des entités d'un graphe [110]. Pour cela, on découpe l'IRI (ou éventuellement le nom de l'entité, tel qu'indiqué par la relation `rdfs:label`) en mots, on récupère les plongements lexicaux de ces mots (c'est-à-dire des représentations vectorielles de mots en faible dimension, comme par exemple Word2Vec [40]), et on combine ces plongements, par exemple en les moyennant. Une telle représentation permet alors de mesurer une distance entre entités, et permet l'application d'algorithmes d'apprentissage automatique. Elle présente toutefois un inconvénient sévère : les plongements lexicaux sont appris sur des corpus textuels génériques et extérieurs au graphe. En ce sens, ces plongements échouent à représenter le contenu et les propriétés intrinsèques du graphe.

Une autre représentation possible d'une entité e est sa matrice d'adjacence $\mathrm{Adj}(e)$ [111] : une matrice booléenne de dimension $m \times n$, dont la coordonnée (i, j) vaut 1 si le triplet (e, r_i, e_j) est valide et 0 sinon. Cette représentation est déjà plus informative que la précédente : on peut comparer les représentations de deux entités e et e', par exemple en calculant le coefficient de Jaccard :

$$\mathrm{Jaccard}(e, e') = \frac{\sum_{i,j} \mathrm{Adj}(e)_{i,j} \wedge \mathrm{Adj}(e')_{i,j}}{\sum_{i,j} \mathrm{Adj}(e)_{i,j} \vee \mathrm{Adj}(e')_{i,j}} \tag{3.1}$$

Cette représentation possède toutefois ses limites. Elle est discrète, donc des triplets proches mais différents seront encodés différemment. Ainsi, la proximité des triplets `(dbr:Montréal, dbo:region, dbr:Québec)` et `(dbr:Island_of_Montréal, dbo:country, dbr:Canada)` échappe complètement à l'indice de Jaccard. Elle est également creuse et de grande dimension : les entités sont représentées par des points épars dans un espace quasiment vide, une situation qui complique l'usage de la plupart des algorithmes d'apprentissage automatique.

Un **modèle de plongement** est une méthode pour obtenir une représentation vectorielle dense et sémantiquement cohérente des entités d'un graphe, à partir des triplets contenus dans le graphe. Dense, c'est-à-dire de faible dimension et avec peu de zéros, par opposition à la représentation sous forme de matrice d'adjacence. Sémantiquement cohérente, car on souhaite que la géométrie des plongements reflète le graphe de connaissances original : deux entités sémantiquement proches dans le graphe doivent avoir des plongements géométriquement proches. Enfin, contrairement à la méthode basée sur des plongements lexicaux, cette représentation vectorielle est basée sur le graphe de connaissances et non sur des ressources externes, et doit donc tenir compte des spécificités de ce graphe.

En toute généralité, un modèle de plongement vectoriel est un modèle qui associe à chaque entité $e \in \mathcal{E}$ un vecteur $\mathbf{e} \in \mathbb{R}^d$, et à chaque relation $r \in \mathcal{R}$ un vecteur $\mathbf{r} \in \mathbb{R}^{d'}$ (vecteur étant ici à comprendre au sens large de «point dans un espace vectoriel», ce qui inclut donc les matrices). On note $\mathbf{E} = \{\mathbf{e}\}_{e \in \mathcal{E}}$ l'ensemble des plongements d'entité, $\mathbf{R} = \{\mathbf{r}\}_{r \in \mathcal{R}}$ l'ensemble

des plongements de relation et $\Theta = (\mathbf{E}, \mathbf{R})$ l'ensemble des paramètres du modèle.

Les deux applications principales d'un modèle de plongement sont d'une part la complétion de triplet [108], et d'autre part la classification de triplet [109]. Dans le premier cas, l'objectif est de prédire l'entité manquante dans un triplet. Par exemple, pour un triplet incomplet `(dbr:China, dbo:capital, ?)`, le modèle devra prédire `dbr:Beijing` à partir des seuls plongements vectoriels de `dbr:China` et `dbo:capital`. La classification de triplet consiste à prédire si un triplet inconnu est valide ou non, à partir des plongements de ce triplet. Dans les deux cas, on voit qu'un modèle de plongement doit être capable d'intégrer les régularités du graphe sous forme géométrique.

3.1.2 Procédure d'entraînement

Pour entraîner un modèle de plongement, on a besoin d'un ensemble de triplets valides Δ_+ (généralement, $\Delta_+ = \mathcal{KG}$), et un ensemble de triplets invalides Δ_-. On attribue l'étiquette 1 aux triplets de Δ_+, et 0 à ceux de Δ_-, ce qui donne un jeu d'entraînement supervisé Δ :

$$\Delta = \{((h,r,t),1)\}_{(h,r,t)\in\Delta_+} \cup \{((h',r',t'),0)\}_{(h',r',t')\in\Delta_-} \tag{3.2}$$

La phase d'entraînement diffère d'un modèle à l'autre, mais l'enjeu est essentiellement d'apprendre au modèle à retrouver la validité $y \in \{0,1\}$ d'un triplet (h,r,t) à partir des seuls plongements $\mathbf{h}, \mathbf{r}, \mathbf{t}$. Dans tous les cas, un modèle définit une perte $J(\Theta)$ à partir de Δ et des plongements Θ. Ceux-ci sont initialisés aléatoirement, puis la perte est minimisée itérativement en mettant à jour Θ par descente de gradient.

Corruption de triplets valides

Par définition, tous les triplets présents dans le graphe sont valides, or un ensemble d'exemples négatifs Δ_- est nécessaire pour l'entraînement. Il faut donc une procédure pour construire des triplets invalides. La méthode la plus simple pour construire un triplet invalide consiste à choisir aléatoirement une triplet valide $(h,r,t) \in \mathcal{KG}$, et à le *corrompre*, c'est-à-dire à choisir h ou t, et à le remplacer par une autre entité $h' \in \mathcal{E} \setminus \{h\}$ ou $t' \in \mathcal{E} \setminus \{t\}$ pour donner un triplet corrompu (h',r,t) ou (h,r,t'). Plus rarement, on choisit de corrompre la relation en tirant aléatoirement $r' \in \mathcal{R} \setminus \{r\}$ et en créant le triplet invalide (h,r',t). On suppose que les triplets corrompus sont effectivement invalides : c'est l'hypothèse du monde localement fermé. Cette hypothèse est empiriquement vérifiée en moyenne, dans tout graphe suffisamment grand. En effet, pour h et r fixés, l'ensemble des entités t telles que (h,r,t) est valide est, en moyenne, de petite taille comparé à l'ensemble des entités t telles que (h,r,t) est invalide.

Toutefois, si elle est vérifiée en moyenne, cette hypothèse devient fragile pour certaines relations, notamment *many-to-one* ou *one-to-many*. Considérons par exemple la relation `rdf:type` qui lie une entité à son type. Dans la version de DBpedia utilisée dans ce mémoire, il y a plus de trois millions d'entités, chacune étant sujet d'une relation `rdf:type`, contre environ 500 types ; parmi ces trois millions, un million environ sont de type `dbo:Person`. Donc, en corrompant un triplet $(h, \texttt{rdf:type}, \texttt{dbo:Person})$ en un nouveau triplet $(h', \texttt{rdf:type}, \texttt{dbo:Person})$, on a une probabilité d'environ 1/3 d'obtenir un triplet valide, qui constituera alors un faux négatif. À l'inverse, si on remplace l'objet du triplet, c'est-à-dire qu'on produit un triplet corrompu $(h, \texttt{rdf:type}, t)$ avec t une entité aléatoire, la probabilité d'obtenir un faux négatif est d'environ 10^{-6} (une entité a généralement entre un et cinq types, d'où l'ordre de grandeur d'un sur un million). [49] propose donc d'amender cette procédure de corruption des triplets, en tenant compte du type de la relation r : *one-to-one*, *one-to-many*, *many-to-one*. Pour mesurer ce type, on définit deux quantités pour toute relation r : le nombre moyen de sujets par objet pour une relation, noté *hpt*, et le nombre moyen d'objets par sujet, noté *tph*. Dans une relation *one-to-many*, une même entité est reliée à de nombreuses autres entités, soit un nombre *tph* élevé par rapport à *hpt* ; inversement, dans une relation *many-to-one*, c'est *hpt* qui est élevé par rapport à *tph*. Enfin, les relations *one-to-one* et *many-to-many*, *tph* et *hpt* auront le même ordre de grandeur. Pour corrompre un triplet (h, r, t), on choisit alors de corrompre le sujet h avec probabilité $\frac{tph}{tph+hpt}$, et de corrompre l'objet t avec probabilité $\frac{hpt}{tph+hpt}$. Ce mode de corruption est appelé *bern* (car le choix de l'objet ou du sujet suit une loi de Bernoulli, de paramètre $\frac{tph}{tph+hpt}$), par opposition à *unif* qui désigne le choix équiprobable de l'objet ou du sujet.

3.2 Modèles de plongement

Cette section présente les modèles de plongement utilisés dans la suite de ce mémoire. On y distingue deux grandes façons de concevoir ces modèles : la première est une approche algébrique, qui considère un graphe de connaissances comme un tenseur d'adjacence, et applique sur ce tenseur des techniques de réduction de la dimension. Cette approche algébrique est illustrée par trois modèles : RESCAL [112], DistMult [113] et ComplEx [114]. Une seconde approche est davantage géométrique : elle consiste à définir des contraintes géométriques pour les triplets valides du graphe, et à entraîner les plongements afin qu'ils respectent ces contraintes. Cette approche géométrique est ici représentée par trois modèles, qui sont TransE [39], TransH [49], TransD [115]. Enfin, on présente RDF2Vec [116], une troisième approche inspirée des modèles de plongements lexicaux.

3.2.1 Approche algébrique : RESCAL et ses variantes

Présentation

Soit $\mathcal{KG}$ un graphe de connaissances ayant n entités et m relations, on note $e_1, e_2, \ldots, e_n$ ses entités et $r_1, r_2, \ldots, r_k$ ses relations. Ce graphe de connaissances peut être représenté sous la forme d'un tenseur [1] d'adjacence $\mathcal{T} \in \{0,1\}^{n \times m \times n}$. Pour une relation $r_k \in \mathcal{R}$, et deux entités $e_i, e_j \in \mathcal{E}$, on a :

$$\mathcal{T}_{i,j,k} = \begin{cases} 1 \text{ si } (e_i, r_k, e_j) \in \mathcal{KG} \\ 0 \text{ sinon} \end{cases} \tag{3.3}$$

Pour une relation $r \in \mathcal{R}$, on note $\mathbf{X}^{(r)}$ la matrice d'adjacence associée à cette relation : $\mathbf{X}^{(r)}$ est une matrice booléenne carrée, de dimension n et dont la coordonnée (i, j) contient 1 si le triplet (e_i, r, e_j) est valide, et 0 sinon. On peut donc voir $\mathcal{T}$ comme une collection des matrices $\mathbf{X}^{(r)}$: $\mathcal{T} = [\mathbf{X}^{(r_1)}, \mathbf{X}^{(r_2)}, \ldots, \mathbf{X}^{(r_m)}]$, comme l'illustre la figure 3.1. Notons ici que le cas où $\mathcal{T}_{i,j,k} = 0$ ne signifie pas que le triplet (e_i, r_k, e_j) est invalide : dans l'hypothèse du monde ouvert, un triplet absent du graphe peut être soit invalide, soit inconnu.

Les modèles algébriques présentés dans cette section approximent les matrices $\mathbf{X}^{(r)}$ par des matrices de rang faible, et extraient de ces approximations des représentations en faible dimension des entités et des relations du graphe. Ils peuvent être rapprochés de techniques de réduction de la dimension comme l'analyse sémantique latente (ou LSA, pour *Latent Semantic Analysis*), fréquente en traitement des langues naturelles.

Dans toute la suite, $d \in \mathbb{N}$ désigne la dimension des plongements. On choisit généralement d entre 50 et 500.

RESCAL [112]

Le modèle RESCAL [112] s'appuie sur une factorisation approximative de $\mathbf{X}^{(r)}$, de rang d, qui s'écrit comme suit :

$$\mathbf{X}^{(r)} \approx \mathbf{E} \cdot \mathbf{W}_r \cdot \mathbf{E}^\top \tag{3.4}$$

Où $\mathbf{E} \in \mathbb{R}^{n \times d}$ contient la représentation vectorielle des entités, et $\mathbf{W}_r \in \mathbb{R}^{d \times d}$ modélise l'interaction entre les composants du sujet et de l'objet pour la relation r.

Notons $\mathbf{E} = (\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n)^\top$: $\mathbf{e}_i$ désigne la i-ème ligne de $\mathbf{E}$ et constitue la représentation vectorielle en d dimension de l'entité e_i, soit le plongement vectoriel recherché. L'équation

1. Dans le cas qui nous occupe, un tenseur peut être vu comme la généralisation d'une matrice à trois dimensions

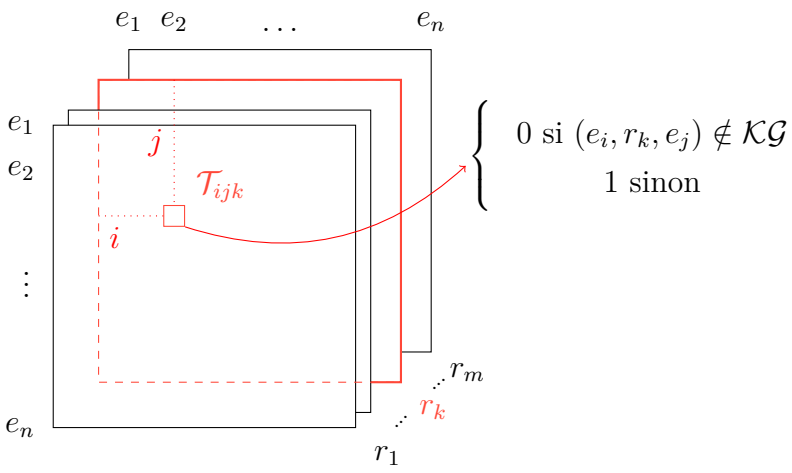


Figure 3.1 Représentation d'un graphe de connaissances $\mathcal{KG}$ sous forme de tenseur d'adjacence $\mathcal{T}$.

3.4 présente deux caractéristiques importantes : d'une part, la représentation des entités ne dépend pas de la relation r ; d'autre part, les entités sont représentées de la même manière selon qu'elles soient sujet ou objet dans la relation. On peut résumer RESCAL avec les deux caractéristiques suivantes :

— les entités sont représentées par un vecteur de dimension d, qui est le même pour chaque relation et quelle que soit la position de l'entité dans le triplet.

— une relation consiste en une interaction entre les représentations vectorielles de son sujet et de son objet. Cette interaction se traduit mathématiquement par une matrice carrée, a priori non symétrique. La matrice étant asymétrique, le rôle des deux entités impliquées (sujet ou objet) est pris en compte.

Ce dernier point est visible lorqu'on ré-écrit l'équation 3.4 pour un triplet (e_i, r_k, e_j) donné. Notons d'abord $\hat{\mathbf{X}}^{(r)}$ l'approximation de rang d de $\mathbf{X}^{(r)}$:

$$\hat{\mathbf{X}}^{(r)} = \mathbf{E} \cdot \mathbf{W}_r \cdot \mathbf{E}^\top$$

Et $\hat{\mathcal{T}} = [\hat{\mathbf{X}}^{(r_1)}, \hat{\mathbf{X}}^{(r_2)}, \ldots, \hat{\mathbf{X}}^{(r_m)}]$ l'approximation de rang d du tenseur d'adjacence complet. On peut alors approximer la validité d'un triplet (e_i, r_k, e_j) par :

$$\hat{\mathcal{T}}_{i,j,k} = \mathbf{e}_i^\top \cdot \mathbf{W}_{r_k} \cdot \mathbf{e}_j \tag{3.5}$$

Cette équation est fondamentale, puisqu'elle permet d'estimer la validité d'un triplet à partir de ses plongements. On la ré-écrit sous forme plus générique, en interprétant la quantité $\hat{\mathcal{T}}_{i,j,k}$

comme le *score* associé au triplet (e_i, r_k, e_j) :

$$s(h, r, t) = \mathbf{h}^\top \cdot \mathbf{W}_r \cdot \mathbf{t} \tag{3.6}$$

Le but est donc d'obtenir des plongements tels que le score s d'un triplet valide soit élevé, et le score d'un triplet invalide soit faible. Pour les obtenir, on définit un programme d'optimisation non-linéaire : les paramètres à entraîner sont les plongements des entités $\mathbf{E}$ et les m matrices de relation $\mathbf{R} = \{\mathbf{W}_r\}_{r \in \mathcal{R}}$. On note $\Theta = (\mathbf{E}, \mathbf{R})$ ces paramètres. La fonction de perte mesure l'écart entre le tenseur d'adjacence initial et son approximation de rang d, avec un terme de régularisation g supplémentaire :

$$J(\Theta) = \frac{1}{2} \sum_{r \in \mathcal{R}} \left\| \mathbf{X}^{(r)} - \hat{\mathbf{X}}^{(r)} \right\|_F^2 + g(\Theta) \tag{3.7}$$

On rappelle que $\| \cdot \|_F$ désigne la norme de Frobenius, une norme matricielle définie par $\|\mathbf{M}\|_F = \sqrt{\sum_{i,j} |M_{i,j}|^2}$. Le terme g est caractérisé par un paramètre λ qui indique l'intensité de la régularisation, et défini par :

$$g(\Theta) = \frac{1}{2} \lambda \left(\|\mathbf{E}\|_F^2 + \sum_{r \in \mathcal{R}} \|\mathbf{W}_r\|_F^2 \right) \tag{3.8}$$

Pour harmoniser les notations avec les autres modèles de plongement, on peut réécrire la perte $J(\Theta)$ à l'aide de la fonction de score s et de l'ensemble d'entraînement Δ défini dans la section 3.1.2 :

$$J(\Theta) = \frac{1}{2} \sum_{(h,r,t), y \in \Delta} \left(y - s(h, r, t) \right)^2 + g(\Theta) \tag{3.9}$$

La perte est donc égale aux carrés de la distance entre le score d'un triplet et son label, plus un terme de régularisation. Muni de cette fonction de perte, l'entraînement des plongements se fait en initialisant aléatoirement les paramètres, puis par descente de gradient sur J jusqu'à atteindre un nombre maximal d'itérations, ou jusqu'à remplir un critère de convergence. Les auteurs [112] proposent le critère de convergence suivant :

$$\frac{\frac{1}{2} \sum_{r \in \mathcal{R}} \left\| \mathbf{X}^{(r)} - \hat{\mathbf{X}}^{(r)} \right\|_F^2}{\|\mathcal{T}\|_F^2} < \epsilon \tag{3.10}$$

Dans l'article original, on prend $\epsilon = 10^{-5}$.

Examinons maintenant quelques propriétés du modèle RESCAL. Notons $w_{i,j}$ la coordonnée

i, j de $\mathbf{W}_r$ et u_i la i-ème coordonnée d'un vecteur $\mathbf{u}$, on peut réécrire l'équation 3.6 sous la forme :

$$s(h, r, t) = \sum_{i,j=1}^{d} h_i \cdot t_j \cdot w_{i,j} \tag{3.11}$$

Ainsi, toutes les combinaisons $h_i t_j$ de $\mathbf{h}$ et de $\mathbf{t}$ apparaissent dans la fonction de score, avec un coefficient $w_{i,j}$. En ce sens, RESCAL propose une expressivité maximale et permet de modéliser une large palette de relations. Par exemple, si $\mathbf{W_r}$ est symétrique (c'est-à-dire, $\mathbf{W}_r = \mathbf{W}_r^\top$), on a $w_{i,j} = w_{j,i}$ pour tous $i, j = 1, \ldots, d$, et donc :

$$s(h, r, t) = \sum_{i,j=1}^{d} \mathbf{h_i} \cdot \mathbf{t_j} \cdot d_{i,j} = \sum_{i,j=1}^{d} \mathbf{h_j} \cdot \mathbf{t_i} \cdot w_{j,i} = \sum_{i,j=1}^{d} \mathbf{h_j} \cdot \mathbf{t_i} \cdot w_{i,j} = s(t, r, h) \tag{3.12}$$

Une matrice symétrique induit donc une fonction de score symétrique elle aussi ; cela permet de modéliser des relations symétriques. Inversement, si $\mathbf{W_r}$ est antisymétrique (c'est-à-dire, $\mathbf{W}_r = -\mathbf{W}_r^\top$), on aura une fonction de score antisymétrique, ce qui permet de modéliser une relation antisymétrique.

Cette expressivité se paie toutefois par un nombre élevé de paramètres (d^2 par relation) : cela demande plus de mémoire que les modèles décrits dans les sections suivantes, et surtout cela complique l'entraînement du modèle, en compliquant la régularisation et favorisant le sur-apprentissage. Pour ces raisons, on n'utilise pas RESCAL dans ce travail, mais deux modèles apparentés : DistMult et ComplEx, présentés dans les deux sections suivantes.

DistMult [113]

DistMult [113], bien qu'il ne soit pas présenté comme tel par ses auteurs, peut être vu comme une simplification de RESCAL pour le rendre plus facile à entraîner et moins sujet au sur-apprentissage. Pour cela, on conserve comme fonction de score un produit trilinéaire entre les plongements des éléments du triplets (équation 3.6), mais on impose que la matrice $\mathbf{W_r}$ soit diagonale – soit d paramètres par relation au lieu de d^2. Cela revient à représenter une relation r par un vecteur $\mathbf{r}$ de dimension d, et à poser $\mathbf{W_r} = \mathbf{I_{d\times d}} \cdot \mathbf{r}$. La fonction de score s'écrit donc :

$$s(h, r, t) = \mathbf{h}^\top \cdot \mathbf{I_{d\times d}} \mathbf{r} \cdot \mathbf{t} \tag{3.13}$$

Ce qui se réécrit sous forme non-matricielle :

$$s(h, r, t) = \sum_{i=1}^{d} h_i r_i t_i \tag{3.14}$$

Ainsi, la fonction de score peut être vue comme un produit scalaire à trois vecteurs. Comparé à RESCAL, l'expressivité est très réduite : seules les combinaisons de la forme $\mathbf{h_i t_i}$ sont utilisées pour le calcul du score. De plus, s est toujours symétrique, donc le sujet et l'objet d'un triplet ne sont pas différenciés.

L'entraînement de DistMult se fait avec une fonction de perte spécifique, la perte par marge maximale, qui s'écrit comme suit :

$$J(\Theta) = \sum_{(h,r,t)\in\Delta_+} \sum_{(h',r',t')\in\Delta_-} \lfloor \gamma + s(h',r',t') - s(h,r,t) \rfloor_+ \tag{3.15}$$

Pour rappel, $\lfloor \cdot \rfloor_+$ désigne la partie positive, c'est-à-dire que $\lfloor x \rfloor_+$ vaut x si $x > 0$, et vaut zéro sinon. Pour interpréter cette perte, considérons un triplet valide $u \in \Delta_+$ et un triplet invalide $u' \in \Delta_-$. L'objectif d'un modèle est de maximiser le score de u et de minimiser celui de u'. Si on pose $\Delta s = s(u) - s(u')$ (c'est l'écart entre le score de u et celui de u'), cet objectif revient alors à maximiser Δs. Le terme de la somme dans l'équation 3.15 s'écrit :

$$j(u,u') = \lfloor \gamma - \Delta s \rfloor_+ \tag{3.16}$$

Ainsi, la perte associée à u et u' est nulle si $\Delta s \geq \gamma$, et vaut $\gamma - \Delta s$ sinon. L'objectif du modèle est donc de séparer les scores des triplets valides des scores des triplets invalides, et ce d'une marge au moins égale à γ. Ce choix de fonction de score est cohérent avec l'usage premier d'un modèle de plongement, qui est de prédire la validité d'un triplet inconnu : pour cet usage, l'enjeu est précisément de pouvoir distinguer les scores des triplets valides de ceux des triplets invalides.

ComplEx [114]

ComplEx est une extension de DistMult dans l'espace complexe. Comme RESCAL, il s'appuie sur une décomposition de la matrice d'adjacence $\mathbf{X}^{(r)}$ associée à la relation r pour en réduire la dimension.

Pour motiver la conception de ComplEx, les auteurs s'interrogent sur les fondements mathématiques de la factorisation de $\mathbf{X}^{(r)}$. Avant réduction de la dimension, la décomposition qui sert de base à DistMult peut s'écrire :

$$\mathbf{X}^{(r)} = \mathbf{E} \cdot \mathbf{W}_r \cdot \mathbf{E}^\top \tag{3.17}$$

Avec $\mathbf{E} \in \mathbb{R}^{n\times n}$ et $\mathbf{W}_r \in \mathbb{R}^{n\times n}$ une matrice diagonale : il s'agit de la diagonalisation d'une matrice réelle symétrique sur une base orthonormée. Toutefois, cette décomposition n'existe

que dans le cas particulier où $\mathbf{X}^{(r)}$ est symétrique : en effet, les valeurs propres d'une matrice réelle symétrique sont réelles, et leurs sous-espaces propres sont orthogonaux, donc le théorème spectral s'applique.

En revanche, la décomposition précédente n'est pas possible dans le cas général où $\mathbf{X}^{(r)}$ n'est pas symétrique. Dès qu'une relation r n'est pas symétrique, RESCAL effectue donc deux approximations successives : d'abord par la décomposition 3.17, et ensuite par la réduction de dimension (c'est-à-dire, lorsqu'on ne garde que les d premières composantes de $\mathbf{E}$ et $\mathbf{W}_r$). Pour éliminer la première approximation, on pourrait utiliser la décomposition en valeur singulière (SVD, pour *Singular Value Decomposition*), qui s'écrit :

$$\mathbf{X}^{(r)} = \mathbf{U} \cdot \mathbf{W}_r \cdot \mathbf{V}^\top \tag{3.18}$$

Avec $\mathbf{U}, \mathbf{V} \in \mathbb{R}^{n \times n}$. Dans ce cas, on a $\mathbf{U} \neq \mathbf{V}$ en général, on perd donc une propriété essentielle de la décomposition précédente : une entité possède une représentation unique qu'elle soit sujet ou objet dans la relation.

ComplEx vise à concilier ces deux exigences : obtenir une représentation unique des entités, indépendante de leur rôle dans le triplet – comme dans la décomposition en valeurs propres des matrices symétriques ; et pouvoir représenter des matrices non-symétriques, comme dans la décomposition en valeurs singulières. Pour cela, le modèle utilise la diagonalisation sur une base de vecteurs propres, comme l'équation 3.17, dans le cas général où $\mathbf{X}^{(r)}$ n'est pas symétrique : dans ce cas, $\mathbf{E}$ et $\mathbf{W}_r$ ne sont plus nécessairement réelles, et la décomposition s'écrit :

$$\mathbf{X}_r = \mathbf{E} \cdot \mathbf{W}_r \cdot \overline{\mathbf{E}}^\top \tag{3.19}$$

Avec $\mathbf{E} \in \mathbb{C}^{n \times n}$ désormais complexe, et $\mathbf{W}_r \in \mathbb{C}^{n \times n}$ une matrice diagonale complexe qui contient les valeurs propres de $\mathbf{X}^{(r)}$, classées dans l'ordre décroissant de leurs modules. En gardant alors les $d \times d$ premières coordonnées de $\mathbf{W}_r$ pour former les matrices $\mathbf{W}_r^{(d)} \in \mathbb{C}^{d \times d}$ et $\mathbf{E}^{(d)} \in \mathbb{C}^{n \times d}$, on obtient une approximation de $\mathbf{X}_r$ de dimension d :

$$\mathbf{X}_r \approx \mathbf{E}^{(d)} \cdot \mathbf{W}_r^{(d)} \cdot \overline{\mathbf{E}^{(d)}}^\top \tag{3.20}$$

La matrice $\mathbf{X}^{(r)}$ devant être réelle, son approximation en basse dimension est projetée dans $\mathbb{R}$ en ne considérant que la partie réelle de $\mathbf{E}^{(d)} \cdot \mathbf{W}_r^{(d)} \cdot \overline{\mathbf{E}^{(d)}}^\top$. Autrement dit, la fonction de score de ComplEx peut s'écrire :

$$\eta(h, r, t) = \Re(\mathbf{h}^\top \cdot \mathbf{W}_r \cdot \overline{\mathbf{t}}) \tag{3.21}$$

Avec $\mathbf{h}, \mathbf{t} \in \mathbb{C}^d$ des vecteurs complexes, et $\mathbf{W}_r$ une matrice complexe diagonale de dimension $d \times d$. Si on note η ce score et non s, c'est parce que ComplEx fait un choix de modélisation légèrement différent des modèles précédents. En effet, plutôt que de prédire directement des valeurs entre 0 et 1, on autorise les coefficients de $\Re(\mathbf{h}^\top \cdot \mathbf{W}_r \cdot \overline{\mathbf{t}})$ à prendre n'importe quelle valeur dans $]-\infty, +\infty[$, et on ramène ensuite le score dans $[0, 1]$ à l'aide de la fonction logistique : $\sigma : z \mapsto (1 + e^{-z})^{-1}$, soit une fonction de score finalement égale à $s = \sigma \circ \eta$:

$$s(h, r, t) = \sigma\left(\Re(\mathbf{h}^\top \cdot \mathbf{W}_r \cdot \overline{\mathbf{t}})\right) \tag{3.22}$$

Un avantage théorique de la fonction de score de ComplEx par rapport aux précédentes est qu'elle est capable d'être symétrique, antisymétrique ou ni l'un ni l'autre, selon la position de $\mathbf{W}_r$ dans l'espace complexe. Considérons d'abord le cas où $\mathbf{W}_r$ est réelle ($\mathbf{W}_r \in \mathbb{R}^{d \times d}$). On a :

$$\begin{aligned}
\eta(h, r, t) &= \Re(\mathbf{h}^\top \cdot \mathbf{W}_r \cdot \overline{\mathbf{t}}) && (3.23)\\
&= \Re\left((\overline{\mathbf{h}^\top \cdot \mathbf{W}_r \cdot \overline{\mathbf{t}}})^\top\right) && (3.24)\\
&= \Re\left(\overline{\mathbf{t}}^\top \cdot \overline{\mathbf{W}_r}^\top \cdot \overline{\mathbf{h}}^{\top\top}\right) && (3.25)\\
&= \Re\left(\mathbf{t}^\top \cdot \mathbf{W}_r \cdot \overline{\mathbf{h}}\right) && (3.26)\\
&= \eta(t, r, h) && (3.27)
\end{aligned}$$

3.24 est vraie car la partie réelle est invariante par conjugaison, et 3.26 parce que $\mathbf{W}_r$ est réelle et symétrique. On ramène alors le score dans l'intervalle $[0, 1]$, et on obtient :

$$s(h, r, t) = s(t, r, h) \tag{3.28}$$

Ainsi, s attribue le même score à un triplet et à son symétrique lorsque le plongement de la relation associée est réel, on peut donc modéliser plus fidèlement une relation symétrique. Si un triplet (h, r, t) est valide et présent à l'entraînement, le modèle sera entraîné pour maximiser $s(h, r, t)$; or, sous réserve que $\mathbf{W}_r$ soit réelle, on aura $s(t, r, h) = s(h, r, t)$ et donc on aura bien l'équivalence : (h, r, t) est valide $\iff$ (t, r, h) est valide.

Inversement, si $\mathbf{W}_r$ est imaginaire pure, c'est-à-dire $\mathbf{W}_r \in i\mathbb{R}$, on aura $\overline{\mathbf{W}_r} = -\mathbf{W}_r$; en injectant ce résultat dans l'équation 3.25, on aura :

$$\eta(h, r, t) = -\eta(t, r, h) \tag{3.29}$$

Ce qui, une fois ramené dans $[0, 1]$, donne :

$$s(h, r, t) = 1 - s(t, r, h) \tag{3.30}$$

Donc un plongement imaginaire pur modélisera des relations antisymétriques, puisqu'un score proche de 1 pour le triplet (h, r, t) impliquera nécessairement un score proche de 0 pour le symétrique (t, r, h).

Enfin, dans le cas général où $\mathbf{W}_r$ n'est ni réelle, ni imaginaire pure, la relation r modélisée ne sera ni symétrique, ni antisymétrique. Évidemment, il s'agit là de propriétés *théoriques* du modèle : en particulier, ces propriétés ne sont utiles qu'à la condition que le modèle soit capable de faire converger les plongements des relations symétriques vers l'espace réel, et les plongements des relations antisymétriques vers l'espace des imaginaires purs. Notons toutefois que la fonction de score est continue en $\mathbf{W}_r$: même si les conditions de symétrie ($\mathbf{W}_r \in \mathbb{R}^{d\times d}$) ou d'antisymétrie ($\mathbf{W}_r \in i\mathbb{R}^{d\times d}$) ne sont pas exactement respectées, la fonction de score peut s'approcher des propriétés de symétrie ($\eta(h, r, t) \approx \eta(t, r, h)$) ou d'antisymétrie ($\eta(h, r, t) \approx -\eta(t, r, h)$).

3.2.2 Modèles translationnels : TransE et ses variantes

On introduit ici une seconde approche, davantage géométrique, que l'on présente à travers l'exemple des modèles translationnels. Un modèle géométrique abandonne la référence au graphe vu comme un tenseur d'adjacence. Il est caractérisé par deux éléments : d'une part, des plongements vectoriels $\mathbf{E} = \{\mathbf{e}\}_{e\in\mathcal{E}} \subset \mathbb{R}^d$ représentant les entités ; d'autre part, une famille d'opérateurs géométriques $\{F_r\}_{r\in\mathcal{R}}$ qui combinent les plongements de deux entités. En appliquant l'opérateur F_r aux plongements $\mathbf{h}$ et $\mathbf{t}$, on obtient un nombre réel positif appelé l'*énergie* du triplet (h, r, t) et noté $E(h, r, t)$. L'objectif d'un modèle de plongement géométrique est de minimiser l'énergie des triplets valides, et de maximiser celle des triplets invalides.

Dans cette section, on examine une unique famille de modèles géométriques, les modèles translationnels. L'opérateur F_r d'un modèle translationnel est précisément la translation du sujet d'un triplet vers l'objet du triplet, la translation étant paramétrée par la relation considérée. Le premier de ces modèles a été TransE [39] ; il a inspiré un grand nombre de variantes dont on présente deux exemples : TransH [49] et TransD [115].

TransE [39]

L'idée fondamentale de TransE est de représenter une relation entre deux entités comme une translation entre leurs plongements. Une translation dans $\mathbb{R}^d$ est une opération linéaire qui consiste intuitivement à «déplacer» un point vers un autre. La distance et le sens de ce déplacement sont caractérisés par un vecteur $\mathbf{u} \in \mathbb{R}^d$; pour un tel vecteur $\mathbf{u}$, la translation associée s'écrit :

$$T_{\mathbf{u}} : \mathbf{x} \mapsto \mathbf{u} + \mathbf{x} \tag{3.31}$$

Dans TransE, chaque relation est associée à un vecteur $\mathbf{r}$ à d dimensions, et donc à une translation T_r. Un triplet (h, r, t) est valide si le translaté du plongement de h par T_r est proche ou égal au plongement de t :

$$(h, r, t) \text{ est valide } \iff T_r(\mathbf{h}) \approx \mathbf{t} \tag{3.32}$$

Les auteurs défendent l'idée que les relations hiérarchiques sont le type de relation fondamentale d'un graphe de connaissances, et que la translation est une approximation naturelle pour modéliser une hiérarchie. De plus, ils remarquent que les plongements lexicaux Word2Vec [40] traduisent naturellement certaines relations *one-to-one* comme des translations. Dans l'article original de Word2Vec [40], les auteurs remarquent la fameuse propriété de leur modèle :

$$\texttt{vec(Berlin) - vec(Germany)} \approx \texttt{vec(Paris) - vec(France)} \tag{3.33}$$

Ce qui revient à dire que la relation «être capital de» se traduit géométriquement par une translation entre le sujet et l'objet. Dans le cas de Word2Vec, il s'agit d'une propriété émergente du modèle, qui n'étais pas attendue par ses auteurs ; pour TransE, on impose au contraire cette propriété comme point de départ de la modélisation.

Mathématiquement, la relation $T_r(\mathbf{h}) \approx \mathbf{t}$ est quantifiée en mesurant la distance euclidienne entre le translaté $T_r(\mathbf{h})$ et l'objet $\mathbf{t}$, ce qui définit l'énergie associée au triplet :

$$E_\Theta(h, r, t) = d_{\text{euc}}(T_r(\mathbf{h}), \mathbf{t}) \tag{3.34}$$

Ce qui est plus généralement écrit :

$$E_\Theta(h, r, t) = \|\mathbf{h} + \mathbf{r} - \mathbf{t}\|_2 \tag{3.35}$$

Le nombre de paramètres est donc $d \times (n + m)$, linéaire en le nombre d'éléments du graphe.

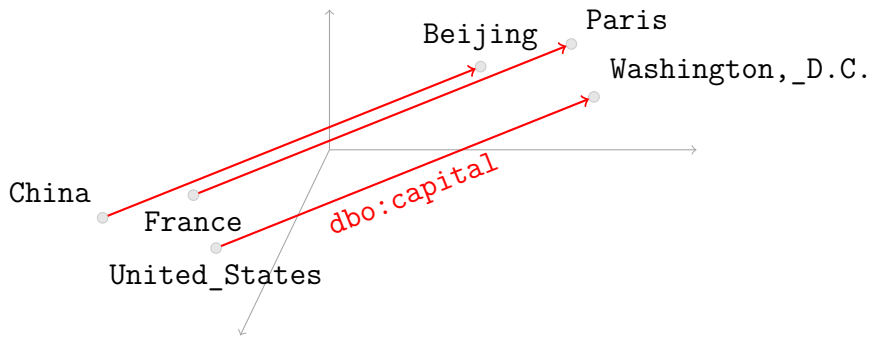


Figure 3.2 Modèle de plongement TransE. La relation `dbo:capital` est représentée par une translation, en rouge, qui relie les plongements des sujets aux plongements des objets.

La force de TransE réside dans sa simplicité théorique et son faible nombre de paramètres. Toutefois, cette simplicité se paye par une faible expressivité, qui limite sa capacité à modéliser des relations autres que *one-to-one* :

- relation *many-to-one* : si plusieurs entités $h_1, h_2, \ldots, h_k$ sont reliées à une même entité t par une relation r, alors les translatés $\mathbf{h_i} + \mathbf{r}$ doivent tous être égaux, donc $\mathbf{h_1} = \mathbf{h_2} = \ldots = \mathbf{h_k}$;
- relation *one-to-many* : comme au-dessus, si h est lié par r à plusieurs entités $t_1, \ldots, t_k$, alors nécessairement $\mathbf{t_1} = \mathbf{t_2} = \ldots = \mathbf{t_k}$;
- relations symétriques : si r est symétrique, c'est-à-dire $h \xrightarrow{r} t \implies t \xrightarrow{r} h$, alors la translation de la translation par r doit donner l'élément de départ, donc $\mathbf{r} = 0$

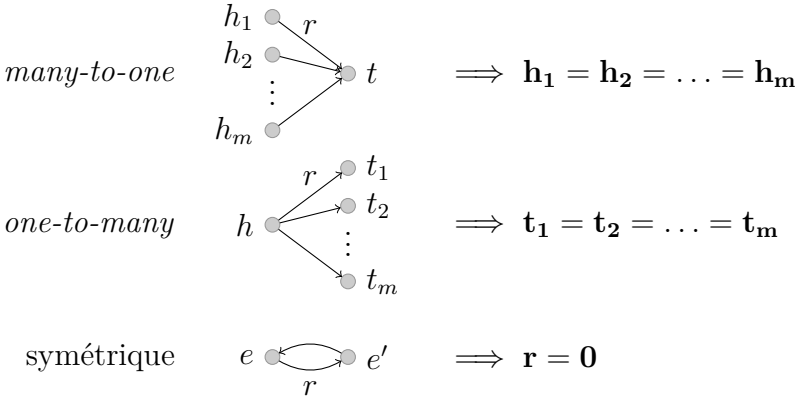


Figure 3.3 Limites du modèle TransE, pour les relations qui ne sont pas *one-to-one*.

Pour résoudre ces problèmes, plusieurs modèles basés sur TransE ont été proposés. Leur idée consiste à ajouter une étape préalable à la translation : le plongement d'une entité e est d'abord envoyé dans un nouvel espace, qui dépend de r et éventuellement de e, avant d'être

translaté dans ce nouvel espace.

Ainsi, on peut définir le cadre général suivant pour décrire un modèle translationnel :

— une fonction de transformation $F_{e,r} : \mathbb{R}^d \to \mathbb{R}^k$

— une fonction d'énergie $E_\Theta(h, r, t) = \|F_{h,r}(\mathbf{h}) + \mathbf{r} - F_{t,r}(\mathbf{t})\|_2$

Dans le cas de TransE, la fonction de transformation est la fonction identité :

$$\forall e, r, F_{e,r} : \mathbf{u} \mapsto \mathbf{u} \tag{3.36}$$

On présente maintenant deux modèles basés sur cette idée, le premier utilisant comme fonction de transformation la projection sur un hyperplan, le second une application linéaire qui dépend du triplet considéré.

TransH [33]

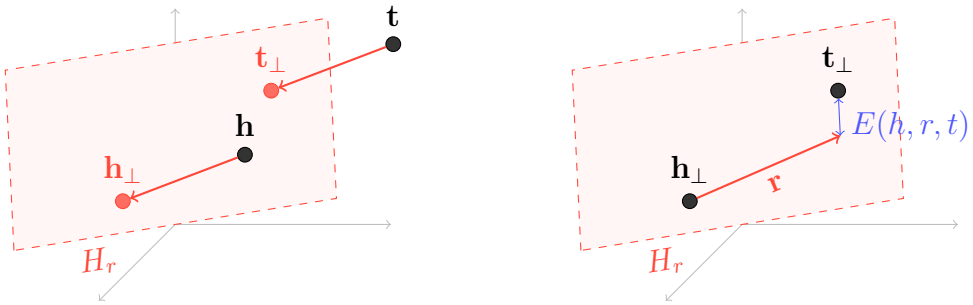


Figure 3.4 Principe général de TransH : pour calculer le score d'un triplet (h, r, t), on commence par projeter $\mathbf{h}$ et $\mathbf{t}$ sur l'hyperplan H_r, puis on translate $\mathbf{h}$. Le score du triplet est égal à la distance entre le translaté de $\mathbf{h}$ et $\mathbf{t}$.

Le premier de ces modèles est TransH, qui associe un hyperplan H_r à chaque relation r, et projette les plongements des entités sur cet hyperplan avant de procéder à la translation.

Un hyperplan H de $\mathbb{R}^d$ est un sous-espace vectoriel de $\mathbb{R}^d$, de dimension $d - 1$: c'est la généralisation en n dimensions d'un plan dans un espace de dimension 3. Un hyperplan H se caractérise par son vecteur normal $\mathbf{n}$: un vecteur $\mathbf{u}$ appartient à H si et seulement s'il est orthogonal à $\mathbf{n}$. On peut donc associer à tout vecteur non-nul $\mathbf{n}$ un hyperplan $H_\mathbf{n}$:

$$H_\mathbf{n} = \{\mathbf{x} \in \mathbb{R}^d : \mathbf{x} \cdot \mathbf{n} = \mathbf{0_d}\} \subset \mathbb{R}^d \tag{3.37}$$

On note $p_{\mathbf{n}}$ le projecteur orthogonal associé à un hyperplan $H_{\mathbf{n}}$, qui s'écrit : Il s'écrit :

$$p_{\mathbf{n}}(\mathbf{x}) = \mathbf{x} - (\mathbf{n}^\top \mathbf{x}) \cdot \mathbf{n} \tag{3.38}$$

Dans le cas de TransH, on apprend comme dans TransE des plongements vectoriels de dimension d pour toutes les entités et les relations du graphe, mais on y ajoute un vecteur normal $\mathbf{w}_r$ pour chaque relation r. Ce vecteur normal permet d'associer à chaque relation r un hyperplan H_r, selon l'équation 3.37. Pour une relation r donnée, et une entité e, on note $\mathbf{e}_\perp = p_{\mathbf{w}_r}(\mathbf{e})$ le projeté orthogonal du plongement de e sur l'hyperplan H_r. On impose de surcroît $\mathbf{r} \in H_r$ pendant l'entraînement.

L'énergie d'un triplet s'écrit alors :

$$E(h, r, t) = \|\mathbf{h}_\perp + \mathbf{r} - \mathbf{t}_\perp\|_2 \tag{3.39}$$

Si l'on remplace les projetés orthogonaux par leurs formules, on retrouve l'équation de TransE avec un terme correcteur $\mathbf{w}_r^\top \cdot (\mathbf{h} - \mathbf{t}) \cdot \mathbf{w}_r$:

$$E(h, r, t) = \left\|\mathbf{h} + \mathbf{r} - \mathbf{t} - \left(\mathbf{w}_r^\top \cdot (\mathbf{h} - \mathbf{t}) \cdot \mathbf{w}_r\right)\right\|_2 \tag{3.40}$$

TransD [115]

TransD est une seconde extension de TransE, dans laquelle les entités à translater sont transformées via une fonction de transformation linéaire qui dépend à la fois de la relation considérée et de l'entité à transformer. Outre les plongements usuels $\mathbf{e}, \mathbf{r}$ définis pour chaque entité $e \in \mathcal{E}$ et chaque relation $r \in \mathcal{R}$, le modèle TransD entraîne aussi des vecteurs de projection $\mathbf{e_p}$ et $\mathbf{r_p}$. Ces vecteurs servent à construire dynamiquement une matrice de transformation $M_{e,r}$:

$$\mathbf{M}_{e,r} = \mathbf{r_p} \cdot \mathbf{e_p}^\top + \mathbf{I_{d \times d}} \tag{3.41}$$

La fonction de transformation pour une entité e et un triplet r s'écrit alors :

$$F_{e,r} = \mathbf{M}_{e,r} \cdot \mathbf{e} \tag{3.42}$$

On peut donc voir le modèle comme ayant un espace générique $\mathbb{R}^d$ dans lequel les entités sont plongées, et réalisant une transformation linéaire de cet espace générique vers un espace spécifique à la relation considérée, tel que représenté à la figure 3.6.

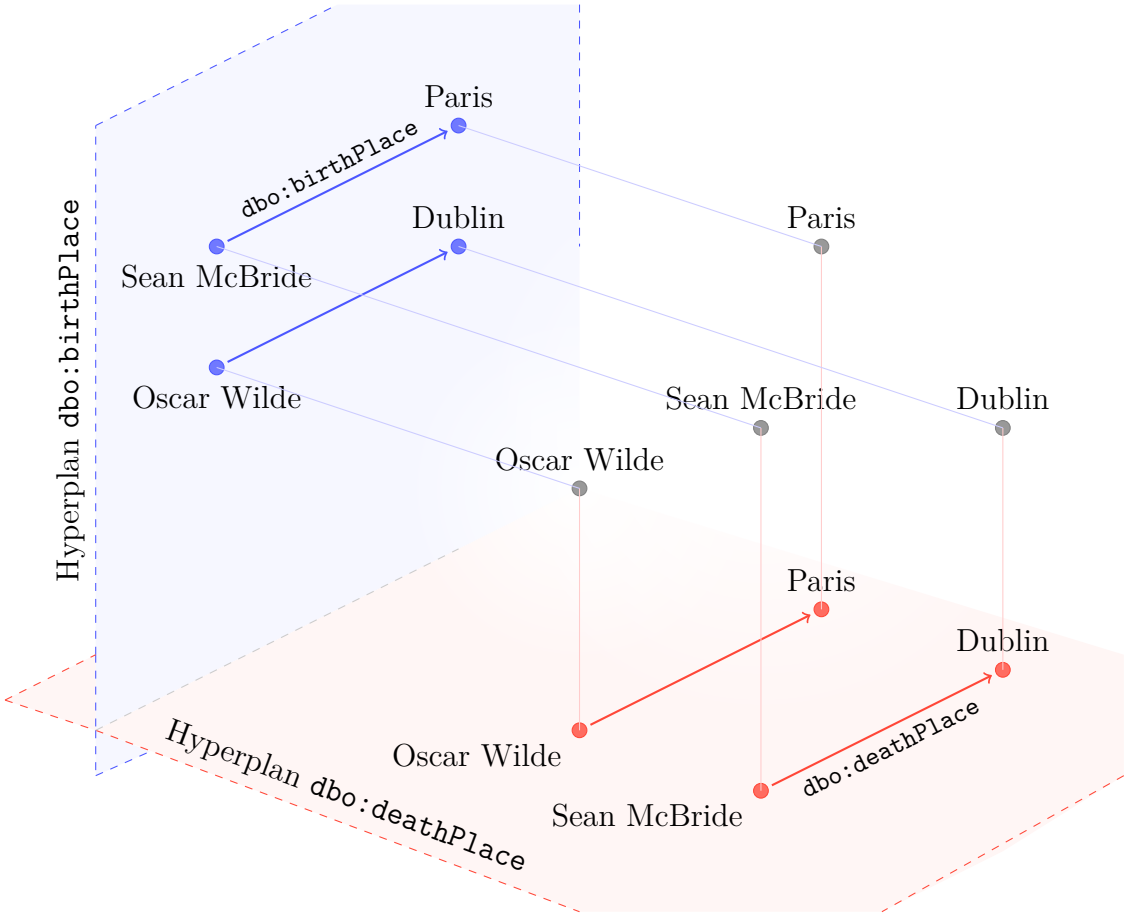


Figure 3.5 Exemple de deux relations `dbo:birthPlace` et `dbo:deathPlace`, ainsi que leurs hyperplans associés.

Soit finalement une fonction d'énergie :

$$E_\Theta(h,r,t) = \|(\mathbf{r_p} \cdot \mathbf{h_p}^\top + \mathbf{I_{d' \times d}}) \cdot \mathbf{h} + \mathbf{r} - (\mathbf{r_p} \cdot \mathbf{t_p}^\top + \mathbf{I_{d' \times d}}) \cdot \mathbf{t}\|_2 \tag{3.43}$$

À nouveau, cette équation se ré-écrit comme l'équation de TransE avec un terme correcteur $\mathbf{r_p} \cdot (\mathbf{h_p}^\top \cdot \mathbf{h} - \mathbf{t_p}^\top \cdot \mathbf{t})$:

$$E_\Theta(h,r,t) = \|(\mathbf{h} + \mathbf{r} - \mathbf{t}) + \mathbf{r_p} \cdot (\mathbf{h_p}^\top \cdot \mathbf{h} - \mathbf{t_p}^\top \cdot \mathbf{t})\|_2 \tag{3.44}$$

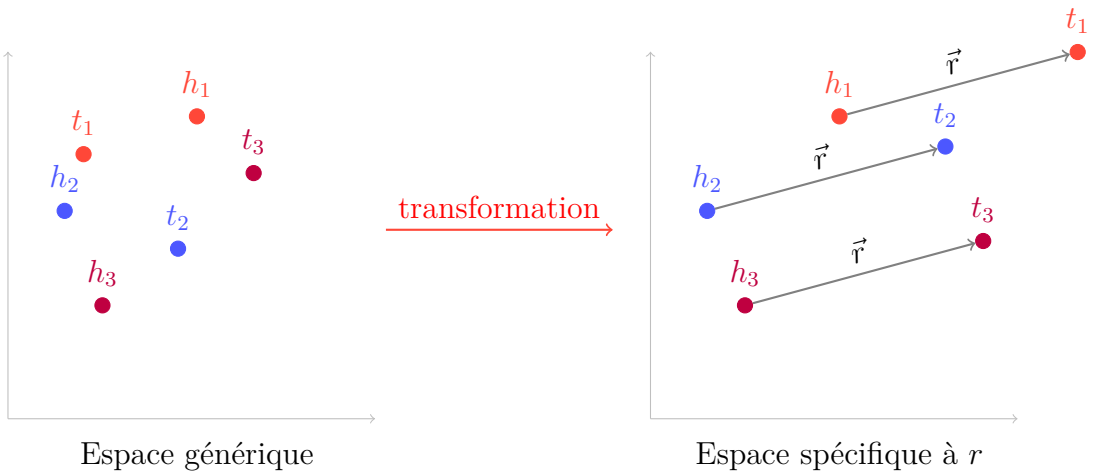


Figure 3.6 Représentation du modèle TransD, avec à gauche l'espace des plongements $\mathbf{e}$, et à droite l'espace des plongements transformés $M_{e,r} \cdot \mathbf{e}$ (tiré de [115]).

Tableau 3.1 Propriétés de quelques modèles à translation

Modèle	Entités	Relations	Transformation	Nombre de paramètres
TransE	$\mathbf{e} \in \mathbb{R}^d$	$\mathbf{r} \in \mathbb{R}^d$	Aucune	$d \times (m+n)$
TransH	$\mathbf{e} \in \mathbb{R}^d$	$\mathbf{r}, \mathbf{n_r} \in \mathbb{R}^d$	Projection sur un hyperplan	$d \times (2m+n)$
TransD	$\mathbf{e}, \mathbf{e_p} \in \mathbb{R}^d$	$\mathbf{r}, \mathbf{r_p} \in \mathbb{R}^d$	Transformation linéaire	$2d \times (m+n)$

3.2.3 Autres modèles

RDF2Vec [116]

Le modèle RDF2Vec s'inspire du modèle de plongement Word2Vec [40], très utilisé pour produire des plongements lexicaux.

Il fonctionne en deux étapes. Dans un premier temps, des chemins sont prélevés dans le graphe au moyen de marches aléatoires. Un *chemin* sur le graphe est une suite d'entités et de relations $(x_1, x_2, \ldots, x_T)$ telles que, pour tout $k \in \{1, 2, \ldots, \frac{T}{2}\}$, x_{2k-1} est une entité, x_{2k} est une relation, et $(x_{2k-1}, x_{2k}, x_{2k+1})$ est un triplet valide de $\mathcal{KG}$. Par exemple, la séquence `(dbr:Picasso, dbo:birthPlace, dbr:Málaga, dbo:country, dbr:Spain)` est un chemin de longueur 5, constitué des triplets `(dbr:Picasso, dbo:birthPlace, dbr:Málaga)` et `(dbr:Málaga, dbo:country, dbr:Spain)`. Dans un tel chemin, le *contexte* d'un élément x_i est l'ensemble des c éléments précédant x_i et des c éléments le suivant, avec c un paramètre appelé la *longueur* du contexte. On le note :

$$x_{t\pm c} = (x_{t-c}, x_{t-c+1}, \ldots, x_{t-1}, x_{t+1}, \ldots, x_{t+c}) \tag{3.45}$$

Pour chaque entité e du graphe, on effectue des marches aléatoires de longueur T au départ de e pour former un ensemble de séquences S_e : il suffit pour cela d'effectuer une recherche en profondeur à partir de e, et de s'arrêter lorsqu'on atteint une profondeur T. Ces marches aléatoires constituent le jeu d'entraînement.

Ensuite, ces chemins sont passés en entrée à un réseau de neurones. Deux variantes existent : dans la version C-BoW, le but de ce réseau est de prédire une entité en fonction des plongements de son contexte ; dans la version Skip-Gram, il s'agit au contraire de prédire le contexte d'une entité à partir du plongement de celle-ci.

C-BoW Dans cette variante, le réseau de neurones prend en entrée le contexte $x_{t\pm c}$ d'un élément ; son objectif est de prédire l'élément cible x_t à partir de ce contexte.

Tout élément $x \in \mathcal{E} \cup \mathcal{R}$ possède un plongement vectoriel $\mathbf{x}$ dans $\mathbb{R}^d$, obtenu par une couche de projection. Le contexte $x_{t\pm c}$ est représenté comme la moyenne des plongements de ses éléments :

$$\overline{\mathbf{x}_t} = \frac{1}{2c} \sum_{-c \leq i \leq c, i \neq t} \mathbf{x}_{t+i} \tag{3.46}$$

Le modèle attribue une probabilité que x_t soit effectivement égal à x, définie comme un softmax sur tout le vocabulaire $\mathcal{E} \cup \mathcal{R}$:

$$P(x|x_{t\pm c}) = \frac{\exp(\overline{\mathbf{x}_t}^\top \mathbf{x})}{\sum_{y \in \mathcal{E} \cup \mathcal{R}} \exp(\overline{\mathbf{x}_t}^\top \mathbf{y})} \tag{3.47}$$

Enfin, on fait glisser la fenêtre de contexte sur toute longueur de la séquence d'entraînement ; l'objectif du modèle est alors de maximiser la log-probabilité de la séquence complète :

$$\frac{1}{T} \sum_{t=1}^{T} \log P(x_t|x_{t\pm c}) \tag{3.48}$$

Cette architecture est représentée à la figure 3.7. En entrée, les éléments sont représentés par des vecteurs *one-hot* : si on note $(w_1, w_2, \ldots, w_V)$ les éléments de $\mathcal{E} \cup \mathcal{R}$, avec $V = |\mathcal{E} \cup \mathcal{R}|$, alors le vecteur *one-hot* de l'élément w_i est un vecteur de dimension V contenant des zéros et un unique 1 à la i-ème coordonnée.

Skip-Gram Skip-Gram opère à l'inverse de C-BoW : l'objectif est ici de prédire le contexte à partir de l'élément cible. On dispose à nouveau d'un élément cible x_t et de son contexte ; le modèle projette x_t dans $\mathbb{R}^d$ avec une couche de projection. Le modèle attribue alors à tout

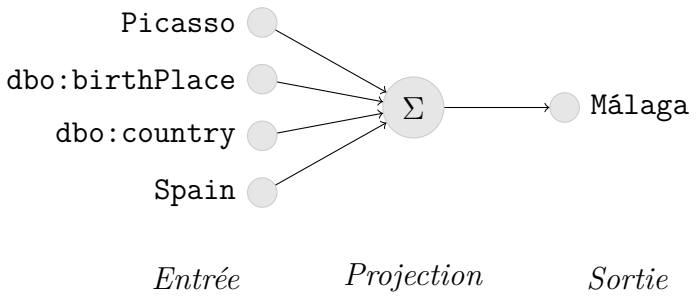


Figure 3.7 L'architecture du modèle RDF2Vec dans sa variante C-BoW.

élément x du vocabulaire $\mathcal{E} \cup \mathcal{R}$ une probabilité : la probabilité que x fasse partie du contexte de x_t. Cette probabilité s'exprime :

$$P(x|x_t) = \frac{\exp(\mathbf{x}^\top \mathbf{x}_t)}{\sum_{y \in \mathcal{E} \cup \mathcal{R}} \exp(\mathbf{y}^\top \mathbf{x}_t)} \tag{3.49}$$

On moyenne alors cette probabilité pour tous les éléments du contexte, pour chaque élément cible de la séquence, ce qui donne la log-probabilité moyenne du contexte, qui doit être maximisée par le modèle :

$$\frac{1}{T} \sum_{t=1}^{T} \sum_{-c \leq i \leq c, i \neq t} \log P(x_{t+i} \mid x_t) \tag{3.50}$$

Cette architecture est présentée à la figure 3.8 ; comme précédemment, l'élément d'entrée est représenté par son vecteur *one-hot*.

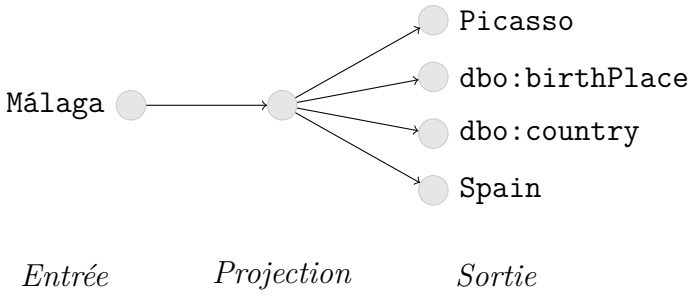


Figure 3.8 L'architecture du modèle RDF2Vec dans sa variante Skip-Gram.

À la fin de l'entraînement, il suffit de récupérer les poids de la couche cachée pour obtenir les plongements vectoriels des entités et des relations.

3.3 Séparabilité des plongements vectoriels

Puisque le but général de ce travail est d'identifier des groupes sémantiquement cohérents d'entités à partir de leur proximité géométrique, il nous est nécessaire d'évaluer la capacité d'un modèle à plonger les instances d'une même classe dans la même région de l'espace euclidien. On formalise cette exigence en définissant une *tâche de séparabilité*, dont le but est de mesurer si des groupes d'entités appartenant à deux classes différentes sont linéairement séparables.

L'hypothèse sous-jacente est la suivante : plus les classes sont nettement délimitées sur le plan géométrique, plus l'identification non-supervisée de ces classes (et donc l'extraction de taxonomie) sera facile. Pour savoir si deux classes sont bien délimitées, nous proposons de mesurer leur *séparabilité linéaire*, ou plus précisément la séparabilité linéaire des plongements vectoriels de leurs instances. Deux groupes de points sont linéairement séparables s'il est possible de faire passer un hyperplan entre les deux ; en deux dimensions, cela signifie que l'on peut trouver une droite telle que tous les points du premier groupe soient situés d'un côté de la droite, et tous les points du second groupe soient de l'autre côté. Notre idée est de mesurer si cette condition est respectée entre les plongements vectoriels de deux classes différentes.

La condition de séparabilité linéaire présente l'avantage d'être conceptuellement simple et facile à calculer, tout en imposant une contrainte forte sur les plongements. En effet, un modèle de plongement peut obtenir de bons résultats sur la complétion de triplets sans respecter cette condition. En revanche, un modèle qui vérifie cette condition de séparabilité garantit un lien direct entre similarité géométrique et similarité sémantique, ce qui devrait faciliter l'extraction de taxonomie.

3.3.1 Données

Pour entraîner nos modèles, nous avons utilisé la version de DBpédia publiée en octobre 2016, qui était la plus récente jusqu'à récemment, et dont on a exclu l'ontologie DBpédia [2] : en particulier, cela signifie qu'aucun triplet `rdfs:subClassOf` n'est présent dans les données. Le graphe complet contient 63,6 millions de triplets (63 550 208 exactement) impliquant 1 274 relations et 3,4 millions d'entités (3 388 747 exactement). Ces entités sont réparties en 589 classes [3], elle-mêmes distribuées sur sept niveaux de hiérarchie : au niveau 0, on trouve

2. Cette ontologie est accessible au format OWL dans le fichier `dbpedia_2016-10.owl`, disponible sur la page de téléchargement de DBpédia

3. On peut trouver la liste actualisée de ces classes à l'adresse : https ://mappings.dbpedia.org/server/ontology/classes/

la classe racine `owl:Thing` dont toutes les entités font partie ; au niveau 1, il y a vingt-quatre classes (parmi lesquelles `dbo:Agent`, `dbo:Place`, `dbo:Event`), etc.

L'entraînement se fait avec la librairie OpenKE [117]. On choisit une dimension de plongement de $d = 50$; pour les autres hyperparamètres, on utilise ceux fournis par les auteurs de chaque modèle.

3.3.2 Méthodes et variables d'analyse

Le principe général est le suivant : pour deux classes A et B, on prélève aléatoirement N instances de A et N instances de B ; on attribue aux instances de A l'étiquette 0 et à celles de B l'étiquette 1. Le résultat est un jeu de données $D = \{\mathbf{e_i}, y_i\}_{i=1,\dots,2N}$, avec $\mathbf{e}_i \in \mathbb{R}^d$ le plongement vectoriel de l'entité e_i, et $y_i \in \{0, 1\}$ l'étiquette associée à e_i. On sépare D en deux groupes : $D_{\text{entraînement}}$ contenant 75% des points, et D_{test} contenant les 25% restants. On entraîne alors une SVM linéaire sur $D_{\text{entraînement}}$, et on prédit ensuite les étiquettes des entités de D_{test}. Les étiquettes comparées peuvent alors être prédites aux étiquettes d'origine, selon les mesures usuelles de la classification automatique : précision, rappel, mesure $F1$.

Pour calculer ces métriques, notons A_{pred} l'ensemble des entités de D_{test} qui ont été attribuées à la classe A par le classificateur SVM, et A_{vrai} les entités de D_{test} qui appartiennent effectivement à la classe A (notons que le choix de A plutôt que B est arbitraire : remplacer l'un par l'autre revient à échanger précision et rappel). La précision est la proportion d'éléments étiquetés A qui appartiennent réellement à A :

$$p = \frac{|A_{\text{pred}} \cap A_{\text{vrai}}|}{|A_{\text{pred}}|}$$

Le rappel est la proportion d'éléments appartenant à A qui ont été correctement étiquetés A :

$$r = \frac{|A_{\text{pred}} \cap A_{\text{vrai}}|}{|A_{\text{vrai}}|}$$

La mesure F_1 est la moyenne arithmétique de la précision et du rappel :

$$F_1 = 2 \times \frac{p \times r}{p + r}$$

Dans nos expériences, on choisit $N = 1000$; si une classe possède moins que ces N instances, on utilise toutes les instances de cette classe.

Variables d'analyse

La difficulté qu'il y a à séparer une classe A d'une classe B dépend beaucoup de A et de B : intuitivement, il est plus difficile de distinguer un `dbo:College` d'une `dbo:University` qu'un `dbo:Aircraft` d'une `dbo:Person`. De plus, on peut imaginer que la taille de la classe (c'est-à-dire le nombre d'instances qui en font partie) influe aussi sur la difficulté, comme c'est souvent le cas en apprentissage automatique. Nous avons donc mis au point trois métriques pour évaluer la difficulté *a priori* de la tâche de séparation pour toute paire de classes (A, B). La première de ces métriques est la distance lexicale entre les classes, mesurée grâce à des plongements lexicaux ; la deuxième est une distance taxonomique entre les classes, basée sur la distance entre les classes dans la taxonomie ; la troisième est un indicateur de la fréquence des deux classes dans la base de connaissances.

Ces métriques permettent d'interpréter les résultats de séparabilité obtenus et d'identifier les modèles qui parviennent le mieux à séparer les classes proches ou les classes rares. On les détaille dans les trois paragraphes suivants.

Distance lexicale La première de nos métriques est une distance entre les classes, basée sur la proximité sémantique des noms de ces classes. Ici, l'intuition est que, si deux classes sont désignées par des mots aux sens proches, alors les classes elles-mêmes doivent être proches. Pour une classe X quelconque, on commence par séparer les mots qui composent son URI : dans DBpédia, la convention de nommage pour les classes est le *CamelCase*, donc l'entité `SportsTeamMember` est séparée en `"sports"`, `"team"` et `"member"`. On utilise alors des plongements lexicaux pour représenter chacun de ces mots par un vecteur de $\mathbb{R}^d$, et on moyenne ensuite ces vecteurs pour obtenir une représentation vectorielle $\mathbf{X}$ de la classe X de départ. On peut alors définir la distance entre deux classes A et B comme la distance euclidienne entre leurs représentants :

$$d_{\text{lex}}(A, B) = \|\mathbf{A} - \mathbf{B}\|_2 \tag{3.51}$$

Les plongements lexicaux utilisés sont entraînés avec CBOW [118] sur le Common Crawl, un corpus en anglais comprenant 600 milliards de mots et deux millions de types distincts. Avec ces plongements, les classes `dbo:River` et `dbo:Stream` ont une distance lexicale proche de zéro, alors que `dbo:Guitarist` et `dbo:MilitaryConflict` ont une distance lexicale élevée.

Distance taxonomique On introduit également une distance entre les classes basée sur leur distance dans la taxonomie de départ. Puisqu'une taxonomie est un arbre, il existe un

unique chemin (non-orienté) de longueur minimale entre n'importe quelle paire de classes A et B dans la taxonomie. En effet, un arbre est un graphe connexe minimal : connexe, donc au moins un chemin existe entre A et B ; minimal, donc il ne peut exister qu'un chemin, sinon on pourrait supprimer une arête d'un des deux chemins sans perdre la connexité, ce qui contredirait la minimalité du graphe. On note donc $\text{path}(A, B) = \{(A \to c_1), (c_1 \to c_2), \ldots, (c_k \to B)\}$ ce chemin. Pour une arête $e = (c_i \to c_j)$ donnée, on définit également sa profondeur $\text{depth}(e)$ comme le nombre d'arêtes entre elle et la racine de l'arbre : une arête connectée directement à la racine a une profondeur nulle ; une arête connectée à un successeur immédiat de la racine a une profondeur égale à 1, etc. On peut alors définir la distance taxonomique comme :

$$d_{\text{tax}}(A, B) = \sum_{e \in \text{path}(A,B)} \frac{1}{2^{\text{depth}(e)}} \tag{3.52}$$

Cette distance taxonomique est la longueur du chemin entre A et B, pondérée par la profondeur des arêtes qui composent ce chemin. Le poids d'une arête est $1/2^{\text{depth}(e)}$ et diminue donc avec sa profondeur : en effet, dans une taxonomie, la profondeur d'une arête est un indicateur de la spécificité de l'axiome de subsumption associé à cette arête. Les axiomes associés aux arêtes de profondeur 0 sont des axiomes très généraux, comme par exemple `dbo:Place` $\sqsubseteq$ `owl:Thing` («un endroit est une chose») ou `dbo:Agent` $\sqsubseteq$ `owl:Thing` («un agent est une chose»). À la profondeur 1, les axiomes sont déjà plus précis – par exemple, `dbo:Person` $\sqsubseteq$ `dbo:Agent` («une personne est un agent»). À des profondeurs plus élevées, les axiomes deviennent très spécifiques, comme par exemple `dbo:MotorRace` $\sqsubseteq$ `dbo:Race` à la profondeur 4. Aussi, deux classes séparées par une arête de faible profondeur sont sémantiquement plus distantes que deux classes séparées par une arête de profondeur élevée. Par exemple, les classes `dbo:NascarDriver` et `dbo:RacingDriver` sont plus proches l'une de l'autre que les classes `owl:Thing` et `dbo:AnatomicalStructure`, bien qu'il n'y ait qu'une arête d'écart dans les deux cas.

Quant au choix d'une décroissance spécifiquement égale à $\frac{1}{2^k}$, il confère à notre distance une propriété intéressante : une classe A est plus proche de ses sous-classes que de n'importe quelle autre classe. Vérifions-le en prenant A une classe de profondeur d, et A' une sous-classe

de A de profondeur $d' > d$. Alors la distance entre A et A' s'écrit :

$$\begin{aligned} d_{\text{tax}}(A, A') &= \sum_{k=d}^{d'-1} 2^{-k} \\ &= 2^{-d} \cdot \sum_{k=0}^{d'-d-1} 2^{-k} \\ &= \frac{1}{2^{d-1}} \left(1 - \frac{1}{2^{d'-d}}\right) \end{aligned}$$

Or la distance entre A et sa superclasse immédiate B est :

$$d_{\text{tax}}(A, B) = \frac{1}{2^{d-1}}$$

Et $d' - d \geq 1$, donc $d_{\text{tax}}(A, A') < \frac{1}{2^{d-1}} = d_{\text{tax}}(A, B) \leq d_{\text{tax}}(A, B')$ pour toute classe B' qui n'est pas sous-classe de A. On retrouve bien la propriété énoncée.

Mesure de fréquence Une entité e impliquée dans de nombreux triplets sera vue plus souvent pendant la phase d'entraînement, et aura donc un plongement vectoriel plus fiable. Comme ce plongement vectoriel dépend à son tour des plongements des entités et des relations qui sont connectées à e, il est possible que les entités appartenant à des classes rares soient impliquées dans des relations rares elles aussi et connectées à des entités rares ; cela produirait des plongements vectoriels moins fiables que la moyenne. Pour vérifier cette hypothèse, on évalue également l'influence de l'effectif des classes (*i.e.* le nombre d'instances qui appartiennent à cette classe) sur les scores de séparabilité. Pour obtenir une mesure synthétique de la fréquence d'une paire de classes (A, B), on utilise la moyenne harmonique des fréquences de A et de B. L'utilisation de la moyenne harmonique plutôt que de la moyenne arithmétique usuelle permet de mieux refléter les déséquilibres entre les fréquences de A et de B : si N_A est la fréquence de la classe A et N_B celle de la classe B, et que N_A est supérieur à N_B d'un ordre de grandeur, alors la moyenne arithmétique de N_A et N_B a le même ordre de grandeur que N_A, et n'indique donc pas que B est rare par rapport à A.

3.3.3 Résultats

On évalue six modèles de plongement sur la tâche de séparabilité : TransE, TransH, TransD, DistMult, ComplEx, RDF2Vec. La séparabilité est calculée sur 10 000 paires de classes issues de DBpédia. On donne les scores de séparabilité moyens pour chaque modèle dans le tableau 3.2 ; on agrège également les résultats pour différentes valeurs de distance lexicale et

taxonomique dans la figure 3.9, et pour différentes fréquences de classes dans la figure 3.10.

Tableau 3.2 Précision, rappel et mesure $F1$ moyens pour différents modèles de plongement sur la tâche de séparabilité. En gras, le meilleur résultat pour chaque métrique.

Modèle	Précision	Rappel	$F1$
ComplEx	90.4	89.9	89.7
DistMult	92.5	91.6	91.6
RDF2Vec	**99.7**	**99.7**	**99.7**
TransE	99.4	99.1	99.2
TransH	93.6	92.1	92.5
TransD	85.0	83.1	83.5

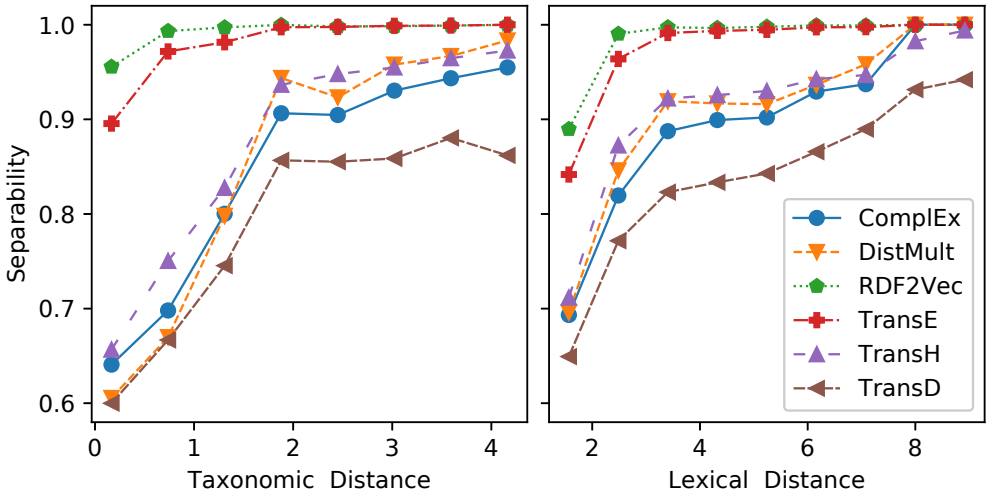


Figure 3.9 Séparabilité moyenne entre deux classes en fonction de leur distance taxonomique (à gauche) et de leur distance lexicale (à droite), pour différents modèles de plongement.

Comme attendu, le score de séparabilité est dépendant de la distance lexicale et taxonomique entre les classes, et ce pour tous les modèles. Tous les modèles sauf TransD parviennent à séparer correctement les classes distantes (distance lexicale supérieure à 8, ou distance taxonomique supérieure à 3) ; en revanche, seuls deux modèles parviennent à conserver des scores élevés sur l'essentiel de l'intervalle : RDF2Vec et TransE, avec un avantage net pour RDF2Vec dans les faibles distances. La séparabilité moyenne de ces deux modèles diminue pour les classes très proches (distance taxonomique inférieure à 1, distance lexicale inférieure à 2), mais elle demeure supérieure à 85%. Tous les autres modèles obtiennent des résultats nettement inférieurs.

Dans la figure 3.10, on observe que les classes rares sont plus difficilement séparables que les classes fréquentes, quoique tous les modèles n'aient pas la même sensibilité à la fréquence :

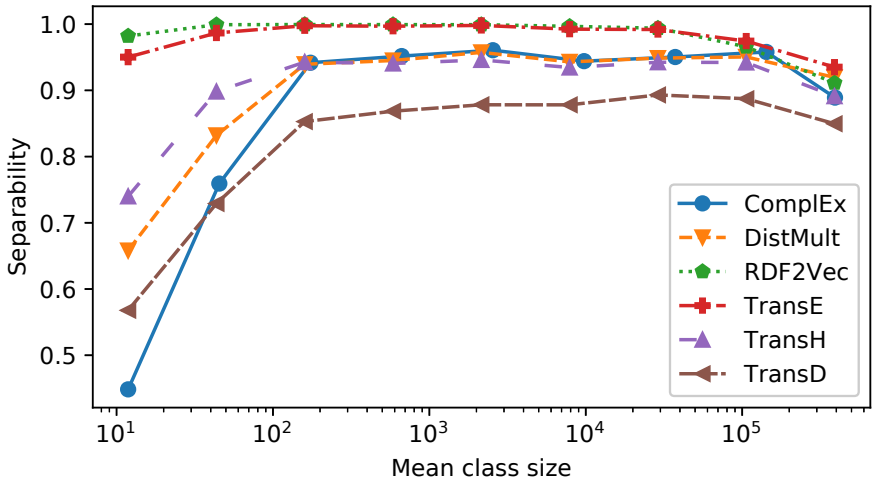


Figure 3.10 Séparabilité moyenne entre deux classes en fonction de la fréquence de ces classes, pour différents modèles de plongement.

RDF2Vec et TransE n'ont qu'une légère baisse de score pour les classes rares (moins de 10^2 instances en moyenne), qui peut partiellement être causée par le faible nombre d'échantillons sur lesquels l'algorithme SVM est entraîné. En revanche, les autres modèles ComplEx, DistMult, TransH et TransD y sont très sensibles. Cette sensibilité ne s'explique pas par une corrélation entre les paires rares et les paires proches, puisque les coefficients de Spearman et de Pearson entre les variables de distance et de fréquence sont proches de zéro.

En résumé, seuls deux modèles obtiennent une mesure $F1$ moyenne supérieure à 95% sur notre tâche de séparabilité : RDF2Vec et TransE, avec respectivement 99,7% et 99,2%. Dans le chapitre suivant, on aura l'occasion de vérifier si ces scores de séparabilité élevés se traduisent effectivement par des scores élevés pour l'extraction de taxonomie.

CHAPITRE 4 EXTRACTION DE TAXONOMIE

Au chapitre précédent, on s'est intéressé à la capacité des modèles de plongement à représenter des entités de même type dans des régions de l'espace bien délimitées. Ici, l'objectif est d'utiliser cette propriété pour reconstituer une taxonomie sur les types à partir de plongements vectoriels de graphe.

L'extraction de taxonomie se fait en deux étapes. Tout d'abord, on utilise la proximité géométrique entre plongements vectoriels pour effectuer un regroupement hiérarchique ascendant sur ces plongements, ce qui permet d'extraire une structure d'arbre sur des groupes d'entités. Ensuite, des types connus sont injectés dans cet arbre, afin de transformer la hiérarchie sur les groupes d'entités en une taxonomie sur les types. Pour réaliser cette injection des types dans l'arbre, nous proposons deux méthodes capables d'associer des types à des groupes d'entités en tenant compte de la structure d'arbre existante.

La section 4.1.1 est consacrée à l'énoncé du problème. Dans la section 4.2, nous proposons deux variantes d'une même méthode pour résoudre ce problème, et nous les évaluons à la section 4.3. Enfin, dans la section 4.4, nous étudions l'influence des différents hyperparamètres.

4.1 Présentation générale

4.1.1 Énoncé du problème

Soit $\mathcal{KG} \subseteq \mathcal{E} \times \mathcal{R} \times \mathcal{E}$ un graphe de connaissances. Pour une entité $e \in \mathcal{E}$, on dit que e est de type t si le triplet $(e, \texttt{rdf:type}, t)$ appartient à $\mathcal{KG}$. Dans la littérature, les types sont souvent appelés des *classes* ou des *concepts* : ici, on préfère l'usage de *type* pour éviter des conflits de notation avec les *clusters*, qui seront introduits dans la section 4.2.1. Notons qu'une entité peut avoir plusieurs types : ainsi, `dbr:Charles_Baudelaire` est à la fois de type `dbo:Poet`, `dbo:Writer` et `dbo:Agent`.

On note $\mathcal{T}$ l'ensemble des types contenus dans le graphe. L'enjeu est de construire une taxonomie T sur $\mathcal{T}$, c'est-à-dire un ensemble d'axiomes de subsumption $\{(t_i \sqsubset t'_i)\}_{i=1,\dots}$. Un axiome $t \sqsubset t'$ signifie que t' est un type plus général que t, et donc que tout élément de type t est aussi de type t'. Dans ce cas, on dit que t est un *sous-type* de t', et que t' est un *supertype* de t. Pour être valide, une taxonomie T doit vérifier une structure d'arbre, c'est-à-dire respecter deux conditions : chaque type doit avoir au plus un supertype, et T ne doit pas contenir de cycle, donc ne doit contenir aucune séquence $t_0, t_1, \dots, t_k$ telle que $t_0 \sqsubset t_1 \sqsubset \dots \sqsubset t_k \sqsubset t_0$.

L'objectif est d'extraire l'information taxonomique contenue dans la géométrie des plongements vectoriels du graphe. On suppose donc avoir accès à un jeu de données $\mathcal{D} = \{(\mathbf{e_i}, t_i)\}_{i=1,\dots,N}$, constitué de N plongements vectoriels typés, c'est-à-dire de N paires $(\mathbf{e_i}, t_i) \in \mathbb{R}^d \times \mathcal{T}$ où $\mathbf{e_i}$ représente le plongement vectoriel de l'entité e_i, et t_i son type.

4.1.2 Idée générale

Dans le problème de l'extraction de taxonomie, on est amené à manipuler des données sur deux plans : le premier est celui des types (`dbo:Person`, `dbo:Athlete`, `dbo:Artist`, ...) reliés entre eux par des liens de subsumption (par exemple, `dbo:Athlete` $\sqsubset$ `dbo:Person`) ; le second est celui des instances et des groupes d'instances, dans lequel on dispose de liens de proximité entre instances, grâce aux plongements vectoriels. Ces deux plans sont liés : les entités sont liées aux types par la relation `rdf:type`, ce qui permet de représenter un type dans l'espace des entités par l'ensemble des instances de ce type, et de traduire l'axiome `dbo:Athlete` $\sqsubset$ `dbo:Person` par une relation d'inclusion entre les instances de `dbo:Athlete` et celles de `dbo:Person`. En pratique, cette liaison est imparfaite : il y a des entités dont les types sont inconnus, erronés ou arbitraires.

Si l'on dispose d'une taxonomie, on peut naturellement déduire une hiérarchie sur les groupes d'entités : ainsi, les entités de type `dbo:Agent` peuvent être divisées en deux sous-groupes, celui des `dbo:Organisation` et celui des `dbo:Person` ; les entités de ce dernier groupe peuvent à leur tour être divisées en plusieurs sous-groupes (`dbo:Athlete`, `dbo:Artist`, etc.). Inversement, les plongements vectoriels d'entités fournissent un moyen pour créer une hiérarchie sur les groupes d'entités : il suffit pour cela d'appliquer un algorithme de regroupement hiérarchique sur ces plongements. Un tel algorithme regroupe les plongements en groupes, sur la base de leur proximité géométrique ; on obtient une division des plongements en groupes géométriquement similaires, eux-mêmes divisés en sous-groupes, et ainsi de suite. L'idée de notre méthode est d'utiliser cette hiérarchie sur les groupes pour induire une hiérarchie sur les types, autrement dit une taxonomie.

La section 4.2 décrit en détail la méthode proposée et ses deux variantes. La section 4.3 évalue cette méthode et discute l'influence de chaque paramètre.

4.2 Méthode proposée

La méthode proposée ici se résume en trois étapes : d'abord, on effectue un regroupement hiérarchique des plongements vectoriels afin d'extraire un arbre de clustering, c'est-à-dire une hiérarchie sur les groupes d'entités (section 4.2.1) ; ensuite, les types des entités sont injectés

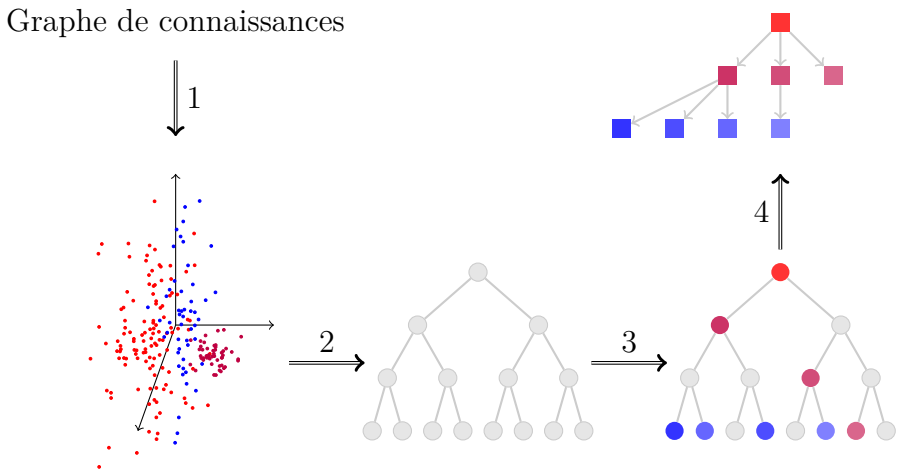


Figure 4.1 Principe général de la méthode d'extraction de taxonomie : (1) les entités du graphe sont plongées dans un espace vectoriel, (2) ces plongements vectoriels sont regroupés hiérarchiquement, (3) les types des entités sont injectés dans la hiérarchie, puis finalement (4) l'association de la hiérarchie et des types est transformée en taxonomie.

dans cet arbre de clustering (section 4.2.2) ; enfin, une taxonomie est extraite en combinant les types et la structure d'arbre (section 4.2.3).

4.2.1 Regroupement hiérarchique

La première étape consiste à exécuter un algorithme de regroupement (ou *clustering*) hiérarchique ascendant sur les plongements vectoriels des entités du jeu de données $\mathcal{D}$. On commence avec N groupes (ou *clusters*) singletons, chacun contenant l'une des N entités de départ. À chaque étape, deux des clusters existants sont fusionnés pour former un nouveau cluster. Les clusters à fusionner sont choisis de sorte à minimiser un certain critère : ce critère peut être la distance moyenne entre les entités des clusters, la distance minimale ou maximale entre ces entités, ou la variance du nouveau cluster. L'algorithme s'arrête lorsqu'il ne reste plus qu'un seul cluster qui contient toutes les entités ; ce cluster est appelé cluster racine. Le résultat de cet algorithme est un arbre binaire, dont la racine est le cluster racine contenant toutes les entités, et dont les feuilles sont les N clusters singletons. L'algorithme 1 contient le pseudo-code correspondant.

Notations

On note $\mathcal{C}$ l'ensemble des clusters, X l'arbre de clustering obtenu, et `racine` la racine de l'arbre – c'est-à-dire, le cluster contenant toutes les entités. On peut écrire $X = (V_X, E_X)$,

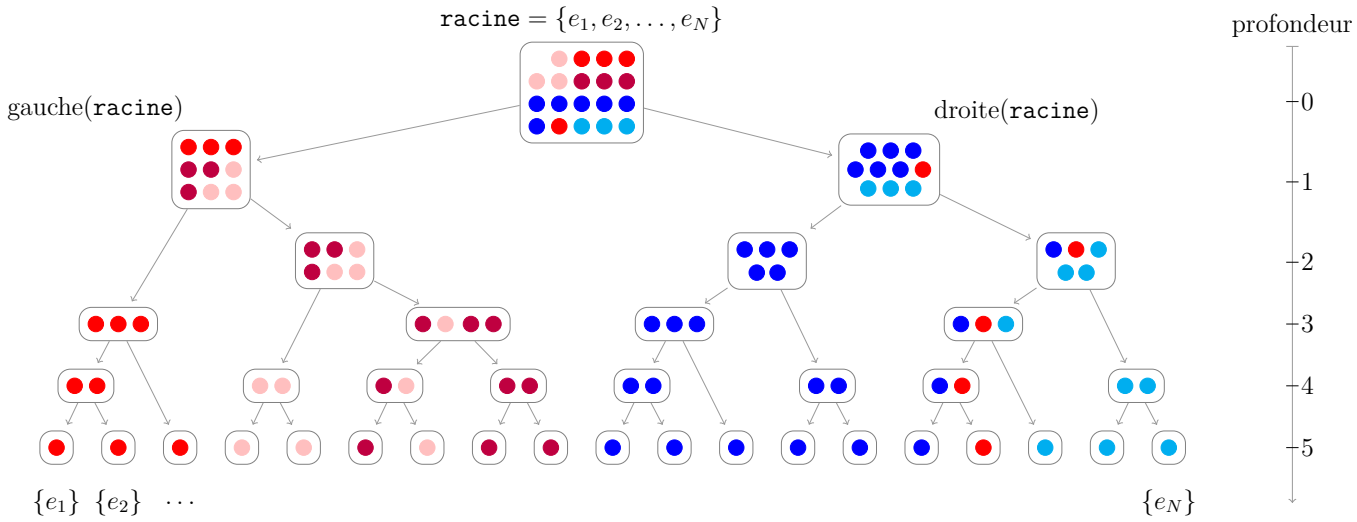


Figure 4.2 Exemple d'arbre de clustering obtenu par regroupement hiérarchique ascendant. La couleur des points représente leur étiquette y_i et n'est pas utilisée lors du regroupement.

avec $V_X = \mathcal{C}$ l'ensemble des sommets de X, et $E_X = \{(C_i, C_i')\}_{i=1,2,\ldots} \subseteq \mathcal{C}^2$ l'ensemble des arêtes de X.

Pour deux clusters C et C', on dit que C est un *prédécesseur* de C' si C appartient à l'unique chemin entre la racine et C', et on a alors $C' \subseteq C$. Dans ce cas, on appelle C' un *successeur* de C. De plus, on dit que C est un prédécesseur *strict* de C' si C est un prédécesseur de C' avec $C \neq C'$, auquel cas on a $C' \subset C$. La relation $\subset$ définit un ordre partiel sur l'arbre X.

Puisque X est une hiérarchie sur les clusters, il vérifie la propriété suivante :

$$\forall C_1, C_2 \in X, C_1 \cap C_2 \neq \emptyset \implies (C_1 \subseteq C_2 \vee C_2 \subseteq C_1) \tag{4.1}$$

Dit autrement, deux clusters sont soit disjoints, soit inclus l'un dans l'autre ; l'inclusion partielle de l'un dans l'autre n'est pas possible.

Pour un cluster C donné, on définit $X[C]$ le *sous-arbre de racine* C comme l'arbre contenant C et tous ses successeurs :

$$X[C] = (\{C' \in X : C' \subseteq C\}, \{(C_1, C_2) \in E_X : C_2 \subset C_1 \subseteq C\}) \tag{4.2}$$

Enfin, on définit gauche(C) et droite(C) respectivement les sous-clusters gauche et droit de C, succ(C) l'ensemble des successeurs de C, et succ$(C)^*$ l'ensemble des successeurs stricts de C.

Critère de fusion

Plusieurs critères sont envisageables pour choisir les clusters à fusionner lors du regroupement. Ici, on en considère quatre : le saut minimum, le saut moyen, le saut maximum, et la distance de Ward. Chacun de ces critères suppose que l'on ait défini une distance d entre les vecteurs à regrouper : cette distance peut être la distance euclidienne, la distance cosinus ou encore la distance de Manhattan.

Notons $\mathcal{C}_t$ l'ensemble des clusters à l'étape t. Un critère de fusion est en réalité une distance D entre les clusters, ou, vu autrement, un *coût* de fusion ; à chaque étape, on fusionne les clusters A et B qui ont la plus faible distance entre eux – autrement dit, on cherche à minimiser le coût de fusion tout au long de la construction de l'arbre. Ainsi, à l'étape t, on détermine les deux clusters à fusionner selon la formule :

$$A_t, B_t = \underset{A,B \in \mathcal{C}_t}{\arg\min}\, D(A, B) \tag{4.3}$$

On crée alors un nouveau cluster $C_{t+1} = A_t \cup B_t$, et on retire les clusters A_t et B_t ; le nouvel ensemble de clusters s'écrit donc :

$$\mathcal{C}_{t+1} = \{A_t \cup B_t\} \cup \mathcal{C}_t \setminus \{A_t, B_t\} \tag{4.4}$$

Et on ajoute à l'arbre de clustering les arêtes (A_t, C_{t+1}) et (B_t, C_{t+1}).

Algorithme 1 : Algorithme de regroupement hiérarchique ascendant

Entrée : un nuage de points $\{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_N\}$
Sortie : un arbre de clustering X
Paramètres : critère de fusion D
$t \leftarrow 0$;
$\mathcal{C}_0 \leftarrow \{\{\mathbf{e}_1\}, \ldots, \{\mathbf{e}_N\}\}$;
$X = (\mathcal{C}_0, \emptyset)$;
tant que $|\mathcal{C}_t| > 1$ **faire**
    `// déterminer les clusters à fusionner`
    $A_t, B_t \leftarrow \arg\min_{A,B \in \mathcal{C}_t} D(A, B)$;
    $C_{t+1} \leftarrow A_t \cup B_t$;
    `// mise à jour des clusters`
    $\mathcal{C}_{t+1} \leftarrow \{C_{t+1}\} \cup \mathcal{C}_t \setminus \{A_t, B_t\}$;
    `// mise à jour de l'arbre`
    $X \leftarrow (V_X \cup \{C_{t+1}\}, E_X \cup \{(A_t, C_{t+1}), (B_t, C_{t+1})\})$;
    $t \leftarrow t + 1$;
retourner X

On définit maintenant les quatre critères de fusion considérés dans la suite.

Saut minimum La distance entre A et B est la distance *minimale* entre un point de A et un point de B :

$$D(A, B) = \min_{\mathbf{x}\in A, \mathbf{y}\in B} d(\mathbf{x}, \mathbf{y}) \tag{4.5}$$

Saut moyen Ici, la distance entre A et B est la distance *moyenne* entre leurs entités :

$$D(A, B) = \frac{1}{|A| \times |B|} \sum_{\mathbf{x}\in A} \sum_{\mathbf{y}\in B} d(\mathbf{x}, \mathbf{y}) \tag{4.6}$$

Saut maximum Cette fois, la distance entre A et B est la distance *maximale* entre un point de A et un point de B :

$$D(A, B) = \max_{\mathbf{x}\in A, \mathbf{y}\in B} d(\mathbf{x}, \mathbf{y}) \tag{4.7}$$

Distance de Ward Le critère de Ward consiste à minimiser l'augmentation d'inertie lors de la fusion de A et de B. L'inertie d'un cluster C est la somme des carrés des distances entre ses entités et son centroïde :

$$I(C) = \sum_{\mathbf{x}\in C} \left[d\left(\mathbf{x}, g(C)\right)\right]^2 \tag{4.8}$$

Avec $g(C)$ le centroïde du cluster C, parfois appelé *centre de gravité* et défini par :

$$g(C) = \frac{1}{|C|} \sum_{\mathbf{y}\in C} \mathbf{y} \tag{4.9}$$

L'augmentation d'inertie lors de la fusion de A et de B est égale à l'inertie de $A \cup B$, à laquelle on retranche l'inertie de A et de B (puisque ces deux clusters disparaissent lors de la fusion) :

$$\begin{aligned} \Delta I &= I(A \cup B) - I(A) - I(B) \\ &= \sum_{\mathbf{x}\in A\cup B} d(\mathbf{x}, g(A \cup B))^2 - \sum_{\mathbf{x}\in A} d(\mathbf{x}, g(A))^2 - \sum_{\mathbf{x}\in B} d(\mathbf{x}, g(B))^2 \end{aligned} \tag{4.10}$$

En pratique, lorsqu'on utilise le critère de Ward, on choisit d'utiliser la distance euclidienne d_{euc}, ce qui permet de réécrire l'équation 4.10 de la façon suivante :

$$\Delta I = \frac{|A| \times |B|}{|A| + |B|} \left\|g(A) - g(B)\right\|_2^2 \tag{4.11}$$

La preuve complète de ce résultat est laissée au lecteur ; elle s'obtient en décomposant $\sum_{\mathbf{x}\in A\cup B} \left\|\mathbf{x} - g(A \cup B)\right\|_2^2$ en deux sommes $\sum_{\mathbf{x}\in A} \left\|(\mathbf{x} - g(A)) + (g(A) - g(A \cup B))\right\|_2^2$ et

$\sum_{\mathbf{x}\in B} \|(\mathbf{x} - g(B)) + (g(B) - g(A \cup B))\|_2^2$, et en exprimant les distances à l'aide du produit scalaire usuel.

En stockant les centroïdes des clusters tout au long de l'algorithme, cette équation permet de calculer le coût de fusion de deux clusters sans devoir recalculer l'inertie de $I(A \cup B)$ à chaque étape. Ainsi, le coût de fusion pour le critère de Ward est finalement :

$$D(A, B) = \frac{|A| \times |B|}{|A| + |B|} \|g(A) - g(B)\|_2^2 \tag{4.12}$$

4.2.2 Association type-cluster

Au terme du regroupement hiérarchique, on dispose d'un côté d'une hiérarchie sur des groupes d'entités (les clusters), et de l'autre une liste de types non hiérarchisée (les étiquettes du jeu de données). L'idée centrale de cette étape est d'utiliser la structure sur les entités pour créer une structure sur les types. Pour cela, on injecte les types dans l'arbre de clustering : soit en associant à chaque type un unique cluster, qui le représente au mieux ; soit en associant à chaque type plusieurs clusters avec des scores d'association. Ces deux méthodes sont exposées dans les sections suivantes.

méthode de liaison injective (MLI)

Pour pouvoir associer les types aux clusters pertinents, il faut mesurer à quel point un cluster C donné représente un type t. Pour ce faire, on peut utiliser la précision et le rappel de ce cluster pour t. Posons $N_{C,t} = |e \in C, type(e) = t|$ le nombre d'entités de type t contenues dans le cluster C, $N_t = |e \in \mathcal{D}, type(e) = t|$ le nombre d'entités de type t dans l'ensemble des données, et $N_C = |C|$ le nombre total d'entités dans le cluster C. On peut alors définir la précision, le rappel et la mesure F_1 :

$$p(t, C) = \frac{N_{C,t}}{N_C} \tag{4.13}$$

$$r(t, C) = \frac{N_{C,t}}{N_t} \tag{4.14}$$

$$F_1(t, C) = 2 \cdot \frac{p(t, C)r(t, C)}{p(t, C) + r(t, C)} = 2 \cdot \frac{N_{C,t}}{N_C + N_t} \tag{4.15}$$

La mesure F_1 varie entre 0 et 1 : 0 indique que le cluster C ne contient aucune entité de type t ; 1 signifie que le cluster contient uniquement des entités de type t et que toutes les entités de type t sont contenues dans le cluster.

Idéalement, on souhaiterait associer chaque type t au cluster qui maximise sa mesure F_1, et

donc choisir $m^*(t) = \arg\max_C F_1(t, C)$, mais on pourrait alors avoir deux types associés au même cluster. Or, si deux types t_A et t_B sont associés au même cluster C, alors tout type associé à un successeur de C aurait à la fois t_A et t_B comme prédécesseurs, et la taxonomie résultante serait donc invalide, selon les conditions énoncées en section 4.1.1. Il faut donc garantir une association univoque entre types et clusters. Formellement, cela signifie que la fonction d'association $m^* : \mathcal{T} \to \mathcal{C}$ entre les types et les clusters doit être injective :

$$\forall t, t' \in \mathcal{T}, t \neq t' \implies m^*(t) \neq m^*(t') \tag{4.16}$$

Pour une injection $m : \mathcal{T} \to \mathcal{C}$, on définit un score $J(m)$ qui mesure la qualité de l'association type-cluster définie par m. Ce score est simplement la mesure F_1 globale induite par m :

$$J(m) = \sum_{t \in \mathcal{T}} F_1(t, m(t)) \tag{4.17}$$

Notre association optimale est alors :

$$m^* = \underset{\substack{m:\mathcal{T}\to\mathcal{C} \\ m \text{ injective}}}{\arg\max} \sum_{t \in \mathcal{T}} F_1(t, m(t)) \tag{4.18}$$

L'équation 4.18 définit un problème d'affectation linéaire, pour lequel une solution optimale existe dans tous les cas (et est unique si les mesures F_1 sont uniques pour chaque cluster). Cette solution optimale peut être trouvée par l'algorithme de Jonker-Volgenant, avec une complexité de $O(|\mathcal{C}|^3)$ dans le pire des cas [119]. Toutefois, l'utilisation de l'heuristique *Asymmetric Greedy Search* [120] permet de réduire cette complexité, tout en trouvant des solutions dont le score est supérieur à 99.9% du score optimal, en moyenne.

méthode de liaison multiple (MLM)

L'approche précédente aboutit à une fonction de décision binaire : un axiome ($t_i \sqsubset t_j$) est prédit ou non, sans intermédiaire possible. Ceci parce que la fonction arg max de l'équation 4.18 n'est pas continue. En conséquence, la taxonomie prédite est sensible à des légers changements dans les données d'entrée.

Dans cette version, on explore une version lissée de l'approche précédente : l'association entre types et clusters est conservée, mais elle est lissée en remplaçant la fonction arg max par une fonction softmax. Ainsi, un type n'est plus associé à un unique cluster ; on lui attribue plutôt un vecteur de probabilités qui, pour chaque cluster, mesure si ce cluster le représente bien. Ensuite, comme précédemment, cette association type-cluster est utilisée

pour transformer la hiérarchie des clusters en une hiérarchie des types. Par rapport au cas précédent, la condition d'injectivité de l'association disparaît : une étape supplémentaire est nécessaire pour transformer le graphe de subsumption en une taxonomie valide. Ainsi, la discontinuité identifiée précédemment ne disparaît pas, mais elle est déplacée : au lieu d'intervenir lors de l'extraction des axiomes de subsumption, elle apparaît lors de la création de la taxonomie prédite à partir de ces axiomes.

Comme précédemment, notre hypothèse est que chaque type t est représenté par un cluster de l'arbre. Contrairement au cas précédent toutefois, on suppose que la localisation de ce cluster est entachée d'incertitude. Cette incertitude a plusieurs sources. Premièrement, l'arbre de clustering obtenu est imparfait (les clusters ne sont pas parfaitement homogènes) car les plongements vectoriels utilisés sont bruités. Ce bruit a lui-même plusieurs origines : en premier lieu, le graphe de connaissances initial est incomplet ; certaines informations sont absentes, inconnues ou mal renseignées ; ensuite, la procédure d'entraînement est stochastique, à la fois pour la génération des exemples négatifs Δ_- et pour la mise à jour des paramètres par descente de gradient ; enfin, les modèles de plongement sont parfois incapables de représenter certaines situations du graphe (par exemple, les relations antisymétriques ne sont pas modélisées par DistMult). Deuxièmement, les types qui servent d'étiquettes aux entités peuvent contenir des erreurs ou des ambiguïtés. Par exemple, les classes `dbo:Politician` et `dbo:OfficeHolder` de DBpedia sont très proches, et l'attribution d'un type ou d'un autre à une entité est souvent arbitraire. Troisièmement, les données d'entrée sont tirées aléatoirement, et un seul de leur type est conservé dans le jeu d'entraînement. Ce tirage aléatoire peut lui aussi entraîner des fluctuations. Toutes ces raisons se cumulent et justifient de modéliser le cluster représentant un type t par une *variable aléatoire* Z_t, plutôt que de lui attribuer une valeur fixe $m(t)$ comme dans le cas précédent.

Formellement, on introduit une famille de variables aléatoires $\mathcal{Z} = (Z_t)_{t\in\mathcal{T}}$, indépendantes et à valeur dans $\mathcal{C}$. On interprète la variable Z_t comme le cluster représentant t parmi tous les clusters de l'arbre de clustering. Il nous faut attribuer une loi de probabilité à Z_t, c'est-à-dire quantifier la probabilité que le type t soit représenté par un cluster C donné. Pour cela, on transforme les mesures F_1 en probabilités, à l'aide d'une fonction softmax. Pour un type t et un cluster C, on définit un score d'association $\sigma(t, C)$:

$$\sigma(t, C) = \frac{e^{\beta F_1(t,C)}}{\sum_{C'\in\mathcal{C}} e^{\beta F_1(t,C')}} \tag{4.19}$$

Pour un type t donné, la somme des scores d'association avec chaque cluster vaut 1, ce qui

permet d'envisager ce score comme une probabilité :

$$\forall t \in \mathcal{T}, \sum_{C \in \mathcal{C}} \sigma(t, C) = 1 \tag{4.20}$$

Le paramètre $\beta > 0$ contrôle la concentration de la distribution autour du maximum. Le cas $\beta \to 0$ attribue à tous les clusters la même probabilité, tandis que le cas $\beta \to \infty$ donne toute la masse de probabilité au cluster de F_1 maximum, ce qui se rapproche de l'approche précédente. Le choix de la valeur de β est discuté dans la section 4.4.

On peut alors définir la loi de la variable aléatoire Z_t, en posant pour $C \in \mathcal{C}$:

$$P(Z_t = C) = \sigma(t, C) \tag{4.21}$$

Par construction du softmax, la position la plus probable pour Z_t est le cluster qui a la mesure F_1 la plus élevée ; toutefois, des clusters qui ont une mesure F_1 proche de la valeur maximale se verront également attribuer des probabilités significatives. L'utilisation d'une exponentielle et un choix de $\beta \gg 1$ permet à l'inverse de faire approcher de zéro les valeurs de F_1 éloignées du maximum, et donc d'éviter les localisations aberrantes de Z_t.

On peut alors interpréter l'axiome $t' \sqsubset t$ comme un événement aléatoire, qui s'exprime en fonction des événements $(Z_t = C)$ et $(Z_{t'} = C)$, pour tous $C \in \mathcal{C}$. En effet, l'axiome $t' \sqsubset t$ signifie que les entités de type t' sont également de type t, et donc que le cluster $Z_{t'}$ représentant t' est un sous-cluster (immédiat ou non) de Z_t le cluster représentant t, ce qui se traduit mathématiquement par la relation $(Z_{t'} \subset Z_t)$. Pour exprimer l'événement $(Z_{t'} \subset Z_t)$, il suffit d'examiner les valeurs possibles de Z_t et $Z_{t'}$, et de ne considérer que les cas ou l'un est inclus dans l'autre. On peut donc décomposer $(Z_{t'} \subset Z_t)$ en union disjointe d'événements $(Z_t = C, Z_{t'} = C')$ pour toutes les paires de clusters C, C' telles que $C' \subset C$:

$$(Z_{t'} \subset Z_t) = \bigsqcup_{\substack{C, C' \in \mathcal{C} \\ C' \subset C}} (Z_t = C, Z_{t'} = C') \tag{4.22}$$

En utilisant cette décomposition et l'indépendance des Z_t, on peut écrire :

$$P(t' \sqsubset t) = P(Z_{t'} \subset Z_t) \tag{4.23}$$
$$= \sum_{\substack{C,C' \in \mathcal{C} \\ C' \subset C}} P(Z_t = C, Z_{t'} = C')$$
$$= \sum_{\substack{C,C' \in \mathcal{C} \\ C' \subset C}} P(Z_t = C) \cdot P(Z_{t'} = C') \tag{4.24}$$
$$= \sum_{\substack{C,C' \in \mathcal{C} \\ C' \subset C}} \sigma(t, C) \cdot \sigma(t', C') \tag{4.25}$$

On obtient donc une formule qui attribue une probabilité à chaque axiome de subsumption en fonction des scores d'association type-cluster. Par rapport au cas précédent, la validité d'un axiome n'est plus évaluée de manière binaire, mais peut prendre n'importe quelle valeur entre 0 et 1. Cela permet, entre autres, plus de souplesse dans l'étape suivante de reconstruction de taxonomie : en éliminant les axiomes trop incertains, on minimise le nombre d'axiomes faux qui sont extraits, quitte à ne pas extraire certains axiomes vrais ; inversement, en conservant des axiomes incertains, on extrait davantage d'axiomes, au risque d'extraire certains axiomes faux.

Un exemple de calcul est illustré à la figure 4.3, dans le cas de deux classes A et B et d'un arbre de clustering de taille 4 et de profondeur 2.

Formulation récursive Pour calculer ces probabilités pour chaque t, t' sans sommer sur toutes les paires de clusters, on peut concevoir un algorithme récursif de type «diviser pour régner». Pour cela, il nous faut exprimer $P(t' \sqsubset t)$ sous forme récursive. On introduit donc une nouvelle quantité, $P_C(t' \sqsubset t)$: c'est, comme dans l'équation 4.23, la probabilité de l'événement $t' \sqsubset t$ (c'est-à-dire de l'événement $Z_{t'} \subset Z_t$), mais restreint au cas où Z_t et $Z_{t'}$ appartiennent au sous-arbre $X[C]$.

$$P_C(t' \sqsubset t) = P(Z_{t'} \subset Z_t \wedge Z_t, Z_{t'} \in X[C]) \tag{4.26}$$

Or appartenir au sous-arbre $X[C]$ est équivalent à être un successeur de C, on peut donc

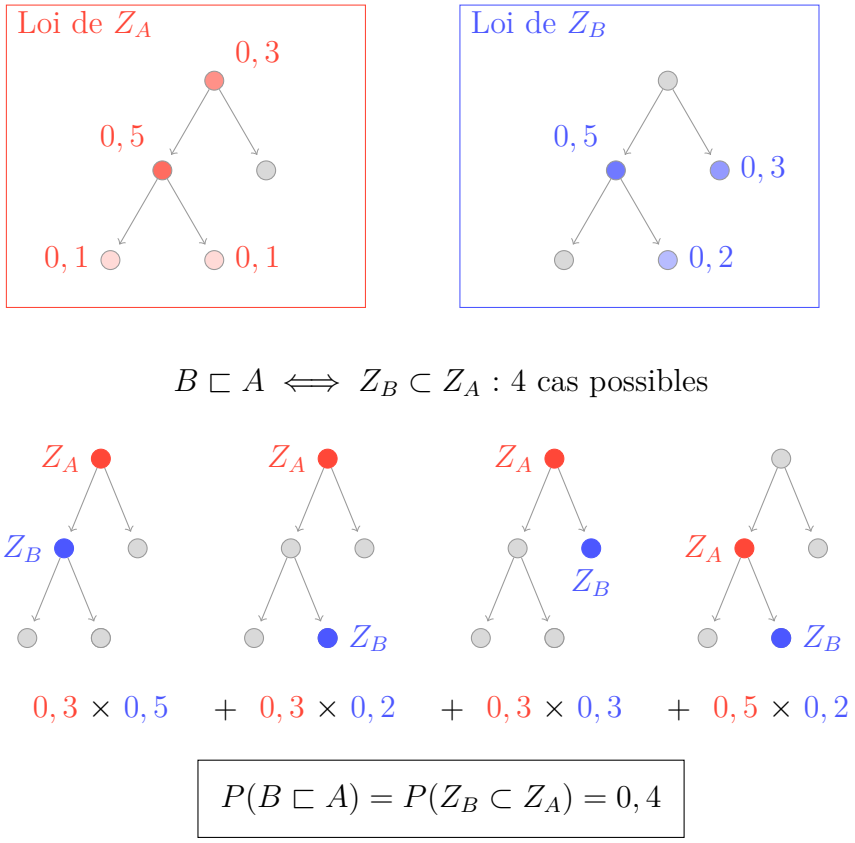


Figure 4.3 Exemple de calcul de la probabilité de l'axiome $(B \sqsubset A)$ à partir de la loi de Z_A (*en haut à gauche*) et de la loi de Z_B (*en haut à droite*).

réécrire l'événement $(Z_{t'} \subset Z_t) \wedge (Z_t, Z_{t'} \in X[C])$ sous la forme $Z_{t'} \subset Z_t \subseteq C$. D'où :

$$\begin{aligned} P_C(t' \sqsubset t) &= P(Z_{t'} \subset Z_t \subseteq C) \\ &= \sum_{\substack{C,C' \in \text{succ}(C) \\ C' \subset C}} P(Z_t = C) \cdot P(Z_{t'} = C) \end{aligned} \tag{4.27}$$

Comme le sous-arbre $X[\texttt{racine}]$ est égal à l'arbre complet, on a $P_{\texttt{racine}}(t' \sqsubset t) = P(t' \sqsubset t)$, et $P(t' \sqsubset t)$ est la quantité que l'on cherche à calculer. De plus, si C est une feuille, alors le sous-arbre $X[C]$ contient un unique élément, il est donc impossible d'avoir $Z_{t'} \subset Z_t \subseteq C$, et donc $P_C(t' \sqsubset t) = 0$ quels que soit t et t'. Ainsi, on connait la valeur de P_C sur les feuilles de l'arbre, et on souhaite calculer $P_{\texttt{racine}}$: il suffit pour cela d'exprimer P_C en fonction de $P_{\text{gauche}(C)}$ et $P_{\text{droite}(C)}$. On pourra alors calculer $P_{\texttt{racine}}(t \sqsubset t')$ récursivement, et on aura donc les probabilités $P(t \sqsubset t')$ recherchées.

Soit $L = \text{gauche}(C)$ et $R = \text{droite}(C)$ les sous-clusters gauche et droit de C. On peut

remarquer qu'un successeur de C est soit un successeur de L, soit un successeur de R, soit C lui-même. Autrement dit, en notant $\sqcup$ l'union disjointe :

$$\mathrm{succ}(C) = \mathrm{succ}(L) \sqcup \mathrm{succ}(R) \sqcup \{C\} \tag{4.28}$$

En suivant cette observation, on peut donc partager la somme 4.27 en trois :

$$\begin{aligned} P_C(t' \sqsubset t) = &\sum_{\substack{C_A, C_B \in \mathrm{succ}(L) \\ C_B \subset C_A}} P(Z_t = C_A) \cdot P(Z_{t'} = C_B) \\ &+ \sum_{\substack{C_A, C_B \in \mathrm{succ}(R) \\ C_B \subset C_A}} P(Z_t = C_A) \cdot P(Z_{t'} = C_B) \\ &+ \sum_{C' :\, C' \subset C} P(Z_t = C) \cdot P(Z_{t'} = C_B) \\ = P_L(t' \sqsubset t) &+ P_R(t' \sqsubset t) + P(Z_t = C) \cdot \sum_{\substack{C' : \\ C' \subset C}} P(Z_{t'} = C_B) \end{aligned} \tag{4.29}$$

Cette équation fait apparaître le lien recherché entre P_C d'un côté, et P_L et P_R de l'autre. On peut l'écrire de façon plus concise sous forme matricielle. Pour cela, définissons $n_T = |\mathcal{T}|$ le nombre de types distincts, $\mathbf{Q_C}$ une matrice de dimension $n_T \times n_T$ telle que sa coordonnée (i, j) contienne $P_C(t_j \sqsubset t_i)$, et $\mathbf{p_C} = (P(Z_{t_i} = C))_{i=1,\dots,n_T}$ le vecteur de probabilités du cluster C, de dimension n_T.

Avec ces notations, notre objectif est de calculer $\mathbf{Q_{racine}}$, et donc d'obtenir une formule récursive reliant $\mathbf{Q_C}$ à $\mathbf{Q_L}$ et $\mathbf{Q_R}$. L'équation 4.29 se réécrit matriciellement :

$$\mathbf{Q_C} = \mathbf{Q_L} + \mathbf{Q_R} + \mathbf{p_C} \cdot \left(\sum_{C' :\, C' \subset C} \mathbf{p_{C'}} \right)^\top \tag{4.30}$$

À ce stade, l'équation 4.30 est proche de la formulation récursive cherchée, mais il reste encore à éliminer la somme $\sum_{C' :\, C' \subset C} \mathbf{p_{C'}}$, parce qu'elle dépend de tous les successeurs stricts de C, et pas seulement de C, L ou R. On note donc $\mathbf{s_C}$ cette somme :

$$\begin{aligned} \mathbf{s_C} &= \sum_{C' :\, C' \subset C} \mathbf{p_{C'}} \\ &= \sum_{C' \in \mathrm{succ}(C)^*} \mathbf{p_{C'}} \end{aligned} \tag{4.31}$$

Cette somme est un vecteur de dimension n_T ; sa i-ème coordonnée représente la probabilité pour le type t_i d'être associé à un successeur strict de C, c'est-à-dire $P(Z_{t_i} \subset C)$. Or un

successeur strict de C est soit L ou R, soit un successeur strict de L ou R, ce qui s'exprime mathématiquement par :

$$\mathrm{succ}(C)^* = \{L\} \sqcup \{R\} \sqcup \mathrm{succ}(L)^* \sqcup \mathrm{succ}(R)^* \tag{4.32}$$

On peut donc réécrire $\mathbf{s_C}$ comme :

$$\begin{aligned} \mathbf{s_C} &= \mathbf{p_L} + \mathbf{p_R} + (\mathbf{s_L} + \mathbf{s_R}) \\ &= (\mathbf{p_L} + \mathbf{s_L}) + (\mathbf{p_R} + \mathbf{s_R}) \end{aligned} \tag{4.33}$$

Finalement, on définit la probabilité pour chaque type d'être associé à un successeur non-strict de C :

$$\mathbf{u_C} = \mathbf{p_C} + \mathbf{s_C} \tag{4.34}$$

Cela permet de reformuler ainsi l'équation 4.33 :

$$\mathbf{s_C} = \mathbf{u_L} + \mathbf{u_R} \tag{4.35}$$

On a désormais nos équations de récurrences : 4.30 pour $\mathbf{Q_C}$, 4.34 pour $\mathbf{u_C}$ et 4.35 pour $\mathbf{s_C}$. Il reste à calculer les cas de base, c'est-à-dire les valeurs de $\mathbf{Q}, \mathbf{u}, \mathbf{s}$ pour les clusters feuilles. Si C est une feuille, la probabilité pour un type t d'être associé à un de ses successeurs stricts est nulle, puisque C n'a aucun successeur strict. Pour la même raison, la probabilité pour qu'un axiome de subsumption $t' \sqsubset t$ soit vérifié dans le sous-arbre $X[C]$ est également zéro. Ainsi $\mathbf{s_C}$ et $\mathbf{Q_C}$ sont tous les deux nuls pour chaque cluster feuille. Quant à $\mathbf{u_C}$, sa valeur sur les feuilles peut être calculée par l'équation 4.34.

On peut finalement écrire les équations de récurrence pour $\mathbf{s_C}, \mathbf{Q_C}, \mathbf{u_C}$:

$$\mathbf{Q_C} = \begin{cases} \mathbf{0}_{n_T, n_T} & \text{si } C \text{ est une feuille} \\ \mathbf{p_C} \cdot \mathbf{s_C}^\top + \mathbf{Q}_{\text{gauche}(C)} + \mathbf{Q}_{\text{droite}(C)} & \text{sinon} \end{cases} \tag{4.36}$$

$$\mathbf{s_C} = \begin{cases} \mathbf{o}_{n_T} & \text{si } C \text{ est une feuille} \\ \mathbf{u}_{\text{gauche}(C)} + \mathbf{u}_{\text{droite}(C)} & \text{sinon} \end{cases} \tag{4.37}$$

$$\mathbf{u_C} = \mathbf{p_C} + \mathbf{s_C} \tag{4.38}$$

Pour implémenter cet algorithme efficacement, on utilise la programmation linéaire. On note $D_C = (\mathbf{s_C}, \mathbf{Q_C}, \mathbf{u_C})$ l'ensemble des quantités calculées récursivement. Les clusters sont parcourus dans l'ordre inverse d'un parcours en profondeur, ce qui garantit que, lorsqu'on vi-

site un cluster C, tous ses successeurs ont déjà été visités. Pour chaque cluster C, soit C est une feuille, auquel cas on calcule D_C à l'aide des formules d'initialisation précédentes ; soit C n'est pas une feuille, auquel cas ses deux sous-clusters gauche(C) et droite(C) ont déjà été visités et les quantités $D_{\text{gauche}(C)}$ et $D_{\text{droite}(C)}$ sont stockées dans le cache. Dans ce cas, on retire $D_{\text{gauche}(C)}$ et $D_{\text{droite}(C)}$ du cache (chaque cluster est visité une seule fois dans les équations de récurrence, donc plus aucun calcul ne dépend d'eux), on les utilise pour calculer D_C selon les équations de récurrence précédentes, et on ajoute D_C au cache. Le dernier cluster visité est `racine` : lorsque l'algorithme termine, le cache contient uniquement $D_{\texttt{racine}} = (\mathbf{s}_{\texttt{racine}}, \mathbf{Q}_{\texttt{racine}}, \mathbf{u}_{\texttt{racine}})$. On peut alors retourner la matrice de probabilité finale $\mathbf{Q}_{\texttt{racine}}$. Sa coordonnée (i, j) contient la probabilité de l'axiome $(t_j \sqsubset t_i)$. Pour un arbre binaire équilibré, cet algorithme garantit un cache de taille maximale $O(|\mathcal{T}|^2 \log_2(|\mathcal{C}|))$, avec $|\mathcal{T}| \ll |\mathcal{C}|$.

4.2.3 Construction de la taxonomie

Une fois les types associés aux clusters, la taxonomie est construite en transformant la hiérarchie sur les clusters en une hiérarchie sur les types. Dans le cas de la méthode de liaison injective, l'injectivité de l'association permet de réaliser facilement cette transformation. La taxonomie prédite $\hat{T}$ contient tous les types $t \in \mathcal{T}$, et elle contient l'axiome $t' \sqsubset t$ si et seulement si les clusters $m(t')$ et $m(t)$ associés à t et t' sont successeurs l'un de l'autre $(m(t') \subset m(t))$, sans qu'il existe un cluster étiqueté entre $m(t)$ et $m(t')$:

$$\hat{T} = (\mathcal{T}, \{t' \sqsubset t : m(t') \subset m(t) \text{ et } \forall C \in \mathcal{C}, m(t') \subseteq C \subseteq m(t) \implies C \notin m(\mathcal{T})\}) \tag{4.39}$$

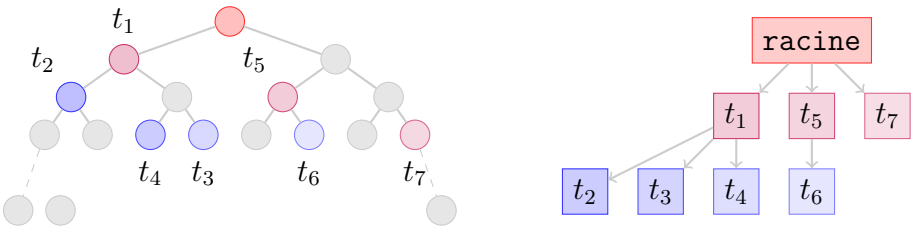


Figure 4.4 Extraction de taxonomie à partir d'un arbre de clustering étiqueté (méthode de liaison injective).

Pour rappel, $m(\mathcal{T})$ désigne l'image de $\mathcal{T}$ par m, c'est-à-dire l'ensemble des clusters qui sont associés à un type :

$$m(\mathcal{T}) = \{C \in \mathcal{C}, \exists t \in \mathcal{T}, m(t) = C\} \tag{4.40}$$

Autrement dit, pour chaque type t', on récupère le cluster associé $m(t')$, et on remonte vers

la racine en cherchant le premier prédécesseur de $m(t')$ qui soit associé à un type. Si un tel prédécesseur $m(t)$ existe, on ajoute à $\hat{T}$ l'axiome de subsumption $t' \sqsubset t$. Ce processus est décrit à la figure 4.4.

Dans le cas de la liaison multiple, on récupère une liste d'axiomes de subsumption $t' \sqsubset t$ qui ne constituent pas nécessairement une taxonomie, et ce pour deux raisons. D'une part, la condition d'injectivité qui existe pour la méthode de liaison injective n'existe pas ici, si bien que l'on peut extraire à la fois les axiomes $t \sqsubset t'$ et $t' \sqsubset t$. D'autre part, même dans le cas idéal – et théorique – où l'algorithme d'association donnerait une probabilité 1 aux axiomes effectivement valides et 0 aux axiomes invalides, le résultat ne serait pas un arbre mais la clôture transitive d'un arbre. En effet, par construction de l'algorithme, on n'extrait pas uniquement l'axiome $t \sqsubset t'$ pour l'immédiat parent de t, mais tous les axiomes $t \sqsubset t_i$ pour les axiomes t_i tels que $t \sqsubset t_1 \sqsubset t_2 \ldots \sqsubset t_k$. Cela revient à supprimer la clause $\forall C \in \mathcal{C}, m(t') \subseteq C \subseteq m(t) \implies C \notin m(\mathcal{T})$ de l'équation 4.39. Pour ces deux raisons, il est nécessaire d'opérer un filtrage avant de pouvoir obtenir une taxonomie valide : d'une part en éliminant les axiomes susceptibles de créer des cycles dans la taxonomie, d'autre part en calculant la réduction transitive (l'opération inverse de la clôture transitive).

Pour cela, on construit un graphe G des axiomes à conserver. Chaque axiome extrait est examiné, par ordre de probabilité décroissante. Pour chaque axiome α, si ajouter α à G crée un cycle, on rejette α. Sinon, on l'ajoute effectivement à G. Par construction, le graphe G ainsi obtenu est un graphe orienté acyclique (ou DAG en anglais, pour *direct acyclic graph*) : il est bien acyclique car le graphe de départ est vide donc acyclique, et car il reste acyclique à chaque étape ; il est orienté car l'axiome de subsumption $t \sqsubset t'$ est différent de l'axiome inverse $t' \sqsubset t$, il y a donc bien une orientation des arêtes de G.

Ensuite, on calcule la réduction transitive G_{red} du graphe G : c'est le plus petit graphe G_{red} tel qu'il existe un chemin de t vers t' dans G_{red} si et seulement si il existe un chemin de t vers t' dans G. Enfin, pour tout type t qui a un degré entrant supérieur à 1, c'est-à-dire qui est impliqué dans au moins deux axiomes $t \sqsubset t'$ et $t \sqsubset t''$, on supprime tous les axiomes de la forme $t \sqsubset t'$ à l'exception de celui de probabilité maximale. Le résultat est la taxonomie extraite $\hat{T}$. Le processus est représenté à la figure 4.5.

4.3 Évaluation et discussion

4.3.1 Données

Comme dans le chapitre précédent, on utilise le graphe DBpedia, dans sa version de 2016-10, dont on élimine les entités impliquées dans un seul triplet. En plus du graphe de connaissances

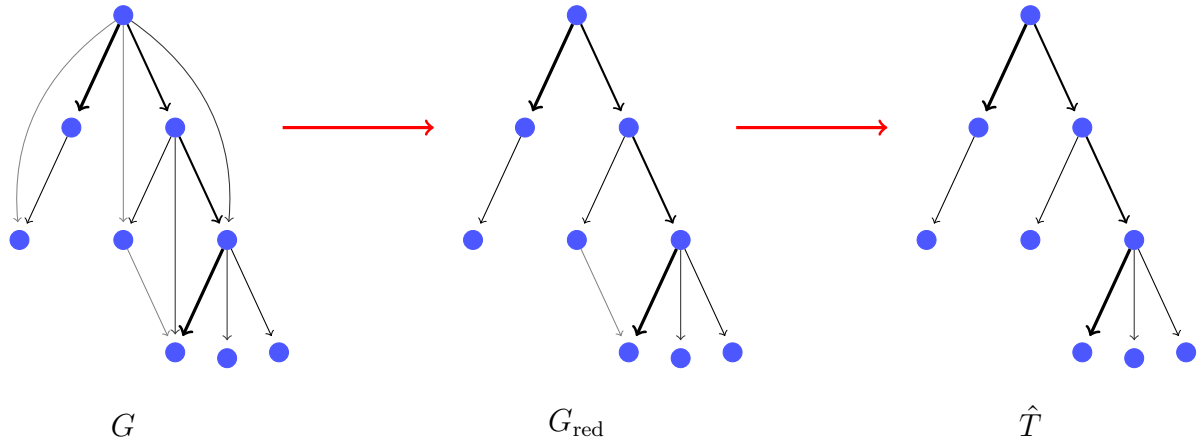


Figure 4.5 Extraction de taxonomie à partir d'une liste pondérée d'axiomes (méthode de liaison multiple) : un graphe acyclique est construit avec les axiomes les plus probables (gauche), sa réduction transitive est calculée (milieu), puis le graphe est transformé en arbre en retirant les arêtes de probabilités minimales (droite). L'épaisseur des arêtes représente la probabilité.

proprement dit, DBpedia dispose d'une ontologie [1], qui contient 455 classes ayant au moins une instance. Toutes les classes héritent d'une classe racine `owl:Thing` ; on appelle *profondeur* d'une classe sa distance à la racine, c'est-à-dire son nombre de superclasses : ainsi, `owl:Thing` a une profondeur de 0, et la profondeur maximale d'une classe est 6. Pour évaluer notre approche, on n'utilise pas toute l'ontologie DBpedia, mais une version simplifiée restreinte aux classes fréquentes. Pour extraire cette sous-taxonomie, on fixe la profondeur maximale à $p_{\max} = 3$, le nombre maximal de sous-classes par classe à $n_c = 7$ et le nombre minimal d'éléments par classe à $n_e = 500$. En partant de la classe racine `owl:Thing`, on ajoute récursivement les sous-classes S d'une classe C, par ordre décroissant d'effectif (on commence par ajouter les sous-classes qui contiennent le plus d'éléments), à condition que :

— S contienne au moins n_e entités
— la profondeur de S soit au plus $p_{\max}$
— C ait moins de n_c sous-classes

Cette taxonomie sera nommée par la suite DBpedia-Fréq. Elle contient sept classes de niveau 1, qui sont `Agent`, `Event`, `MeanOfTransportation`, `Place`, `Species`, `SportsSeason`, `Work` ; 25 classes de niveau 2, et 52 classes de niveau 3. Cette taxonomie, quoique simplifiée, couvre 75% des triplets `rdf:type` de DBpedia. Les valeurs des paramètres $p_{\max}, n_c, n_e$ ont été choisies empiriquement de sorte à atteindre ce seuil de couverture.

1. qui peut être consultée à l'adresse mappings.dbpedia.org/server/ontology/classes/ dans une version légèrement mise à jour, ou téléchargée au format OWL à l'adresse downloads.dbpedia.org/2016-10/dbpedia_-2016-10.owl

Une fois que l'on dispose de cette taxonomie de référence DBpedia-Fréq, on prélève aléatoirement dans le graphe des instances de chacune des classes de la taxonomie ; on obtient un jeu de données $\mathcal{D}$ comportant $N = 51418$ instances, chacune étant ensuite représentée par un plongement vectoriel de dimension $d = 50$. L'objectif est alors de reconstituer DBpedia-Fréq à partir de $\mathcal{D}$.

4.3.2 Méthode d'évaluation

On évalue notre méthode en faisant varier le modèle de plongement employé, le type de regroupement hiérarchique et la méthode d'association type-cluster. Les modèles de plongement expérimentés sont TransE, TransD, TransH, ComplEx, DistMult et RDF2Vec. Le regroupement hiérarchique se fait avec un des quatre critères de liaison cités (saut minimum, moyen, maximum, critère de Ward) et l'une des trois distances suivantes : euclidienne, cosinus, L_1. La méthode d'association est soit la méthode de liaison injective (MLI), soit la méthode de liaison multiple (MLM). On compare ces résultats avec ceux obtenus par la méthode TIEmb [30] (présentée à la section 2.2.2, page 26) ; pour cette méthode également, on expérimente avec les distances euclidienne, cosinus et L_1.

Pour comparer une taxonomie prédite $\hat{T}$ avec la taxonomie de référence de DBpedia-Freq, notée ici T^*, on mesure la différence entre les axiomes de subsumption prédits et les axiomes de référence en calculant la précision (nombre d'axiomes prédits qui sont vrais), le rappel (nombre de vrais axiomes qui sont correctement prédits), et la mesure F_1 (moyenne géométrique de la précision et du rappel). On calcule également la clôture transitive de $\hat{T}$ et T^* : la clôture transitive d'un arbre T est le graphe T^+ qui contient les mêmes sommets que T et dont les arêtes sont définies par $t \sqsubset t' \iff \exists t_1, \dots t_k, (t \sqsubset t_1), (t_1 \sqsubset t_2), \dots, (t_k \sqsubset t') \in T$. C'est donc le graphe obtenu en reliant un sommet t à tous ses prédécesseurs dans T. Une fois ces clôtures transitives calculées, on évalue à nouveau la différence entre les axiomes de $\hat{T}^+$ et ceux de $(T^*)^+$ en terme de précision, de rappel et de mesure F_1. Ces deux modes d'évaluation sont appelés respectivement *évaluation directe* et *évaluation transitive*.

Ces deux modes d'évaluation – directe et transitive – sont nécessaires pour évaluer correctement la proximité d'une taxonomie prédite avec une taxonomie de référence, car ils ne mesurent pas la même chose. Prenons le cas où le modèle d'extraction prédit correctement la plupart des axiomes mais commet des erreurs sur les axiomes de haut niveau (proches de la racine, par exemple en prédisant à tort l'axiome `dbo:Person` $\sqsubset$ `dbo:Event`). Comme la majorité des axiomes sont corrects, la mesure F_1 directe est élevée ; toutefois, après clôture transitive, l'erreur de haut niveau se propage dans tout le sous-arbre : dans notre exemple, toutes les sous-classe de `dbo:Person` seront incorrectement considérées comme

des sous-classes de `dbo:Event`, ce qui fera chuter la précision et donc la mesure F_1. À l'inverse, considérons le cas où un modèle extrait correctement les axiomes de haut niveau mais ne parvient pas à reproduire les axiomes de plus bas niveau – par exemple en prédisant `dbo:SoccerPlayer` $\sqsubset$ `dbo:Person` et `dbo:Athlete` $\sqsubset$ `dbo:Person` au lieu de `dbo:SoccerPlayer` $\sqsubset$ `dbo:Athlete` $\sqsubset$ `dbo:Person`. Dans ce cas, la mesure F_1 transitive est élevée, car la majorité des axiomes transitifs de référence sont correctement prédits, mais la mesure F_1 directe est basse, car les axiomes directs manquent.

4.3.3 Résultats

Les résultats principaux sont présentés dans le tableau 4.1 pour les modèles ComplEx, DistMult, RDF2Vec et TransE. On y présente les résultats pour les méthodes MLI et MLM avec, pour chacune, deux variantes de regroupement hiérarchique : l'une avec distance cosinus et critère du saut moyen, l'autre avec distance euclidienne et critère de Ward. Pour les modèles MLM, les paramètres sont fixés à $\beta = 100$ et $t = 0.1$ (ces choix sont discutés dans la section 4.4.2). Pour la méthode TIEmb, on se restreint également aux distances cosinus et euclidienne. Les résultats pour les autres modèles de plongement, métriques et critères de liaison sont globalement moins bons : par souci de concision, on les omet ici ; ils sont présentés à l'annexe B.

Du point de vue du F_1 moyen, le meilleur modèle est MLM avec TransE et distance cosinus ; ce modèle obtient d'ailleurs le F_1 le plus élevé sur chacune des trois évaluations. Ce modèle est suivi, dans l'ordre décroissant, par MLI avec TransE et distance cosinus, MLM avec TransE et distance euclidienne, TIEmb et les distances cosinus puis euclidiennes, et MLI avec TransE et distance euclidienne. Tous ces modèles obtiennent un F_1 moyen supérieur à $0,75$.

En moyenne, la méthode MLM obtient des résultats légèrement supérieurs à MLI, et supérieurs à TIEmb. TIEmb est inférieur aux méthodes MLM et MLI sur tous les modèles de plongement à l'exception de RDF2Vec, où il obtient de meilleurs scores.

Si l'on évalue les modèles de plongement sur leur capacité à intégrer géométriquement l'information taxonomique, la hiérarchie entre modèles est nette : TransE obtient les meilleurs résultats, quelle que soit la méthode d'extraction ou la distance utilisée ; RDF2Vec couplé à TIEmb obtient de bons résultats pour l'évaluation directe, ce qui indique une capacité à intégrer correctement les hiérarchies locales, mais commet des erreurs de haut niveau qui grèvent ses résultats moyens ; ComplEx et DistMult sont nettement inférieurs à TransE, avec un léger avantage pour ComplEx.

Tableau 4.1 Évaluation de notre approche et de TIEmb sur DBpedia-Freq, pour différents modèles de plongement. p, r, F_1 désignent respectivement la précision, le rappel et la mesure F_1. *cos* et *euc* indiquent les distances cosinus et euclidienne. Les résultats de la section *Moyenne* sont obtenus en calculant la moyenne des évaluations directe et transitive.

			Directe			Transitive			Moyenne		
Méthode	Plongements	Distance	p	r	F_1	p	r	F_1	p	r	F_1
TIEmb	ComplEx	cos	0.38	0.38	0.38	0.28	0.57	0.37	0.33	0.48	0.38
		euc	0.39	0.42	0.4	0.34	0.8	0.48	0.37	0.61	0.44
	DistMult	cos	0.26	0.27	0.26	0.19	0.44	0.27	0.23	0.36	0.27
		euc	0.37	0.41	0.39	0.33	0.79	0.47	0.35	0.60	0.43
	RDF2Vec	cos	0.77	**0.84**	0.81	0.43	0.87	0.57	0.60	0.86	0.69
		euc	0.70	0.77	0.73	0.30	0.73	0.42	0.50	0.75	0.58
	TransE	cos	0.77	0.72	0.74	0.83	0.89	0.86	0.80	0.81	0.80
		euc	0.76	0.75	0.76	0.7	0.9	0.79	0.73	0.83	0.78
Liaison injective (MLI)	ComplEx	cos	0.53	0.5	0.52	0.78	0.7	0.74	0.66	0.60	0.63
		euc	0.51	0.44	0.47	0.87	0.52	0.65	0.69	0.48	0.56
	DistMult	cos	0.55	0.47	0.5	0.73	0.57	0.64	0.64	0.52	0.57
		euc	0.43	0.36	0.39	0.75	0.54	0.63	0.59	0.45	0.51
	RDF2Vec	cos	0.60	0.44	0.50	0.88	0.50	0.63	0.74	0.47	0.57
		euc	0.60	0.53	0.56	0.89	0.64	0.74	0.74	0.58	0.65
	TransE	cos	0.82	0.8	0.81	0.97	0.89	**0.93**	**0.90**	0.85	0.87
		euc	0.73	0.67	0.7	**0.99**	0.68	0.8	0.86	0.68	0.75
Liaison multiple (MLM)	ComplEx	cos	0.52	0.48	0.5	0.89	0.7	0.79	0.71	0.59	0.65
		euc	0.51	0.44	0.47	0.84	0.6	0.7	0.68	0.52	0.59
	DistMult	cos	0.5	0.44	0.47	0.87	0.65	0.74	0.69	0.55	0.61
		euc	0.48	0.42	0.45	0.86	0.65	0.74	0.72	0.37	0.49
	RDF2Vec	cos	0.61	0.36	0.45	0.84	0.39	0.53	0.77	0.51	0.61
		euc	0.61	0.47	0.53	0.92	0.55	0.69	0.74	0.58	0.65
	TransE	cos	**0.83**	0.81	**0.82**	0.93	**0.93**	**0.93**	0.88	**0.87**	**0.88**
		euc	0.78	0.77	0.77	0.98	0.87	0.92	0.88	0.82	0.85

4.3.4 Discussion

On peut trouver étonnante la nette supériorité du modèle TransE, alors qu'il s'agit du plus simple des modèles testés. Pourtant, la simplicité des hypothèses géométriques derrière TransE force précisément le modèle à plonger les entités de même type dans la même région de l'espace vectoriel. En effet, pour deux entités h, h' de type t, on a :

$$\begin{aligned}\|\mathbf{h} - \mathbf{h'}\|_2 &= \|(\mathbf{h} + \mathbf{r}_{\text{IS\_A}} - \mathbf{t}) - (\mathbf{h'} + \mathbf{r}_{\text{IS\_A}} - \mathbf{t})\|_2 \\ &\leq \|\mathbf{h} + \mathbf{r}_{\text{IS\_A}} - \mathbf{t}\|_2 + \|\mathbf{h'} + \mathbf{r}_{\text{IS\_A}} - \mathbf{t}\|_2 \\ &\leq E(h, r_{\text{IS\_A}}, t) + E(h', r_{\text{IS\_A}}, t)\end{aligned}$$

Puisque $(h, r_{\text{IS\_A}}, t)$ et $(h', r_{\text{IS\_A}}, t)$ sont deux triplets valides, les énergies $E(h, r_{\text{IS\_A}}, t) + E(h', r_{\text{IS\_A}}, t)$ sont minimisées pendant l'entraînement, et la distance entre $\mathbf{h}$ et $\mathbf{h'}$ est également minimisée. On avait déjà noté cette propriété dans la section 3.2.2 sur les limites du modèle TransE pour les relations *many-to-one* : ici, dans le cas de l'extraction de taxonomie, cette limitation se transforme en point fort. De plus, nos observations permettent de valider une des hypothèses derrière TransE : ses concepteurs expliquaient que la translation était une bonne représentation pour les relations hiérarchiques, sans pouvoir en apporter de preuve ; ici, les bons résultats de TransE apportent un argument empirique à cette intuition.

Dans le chapitre précédent, on concluait que RDF2Vec et TransE étaient les meilleurs modèles en termes de séparabilité, dans cet ordre. Ici, nous devons constater que la séparabilité ne permet pas entièrement de préjuger des performances des modèles sur la tâche d'extraction de taxonomie. En effet, on observe bien une supériorité de RDF2Vec et TransE par rapport aux autres, mais c'est ici TransE qui domine nettement, tandis que RDF2Vec est à peine au-dessus de ComplEx et DistMult. Une piste d'explication viendrait du regroupement hiérarchique : si on observe les arbres de clustering produits par RDF2Vec et TransE, on constate que RDF2Vec donne des arbres plus déséquilibrés que TransE, et présente notamment une tendance à fusionner des clusters de tailles très inégales. Cette «aptitude au regroupement» n'est pas mesurée par la séparabilité, et pourrait expliquer l'inversion de la hiérarchie entre TransE et RDF2Vec.

Finalement, notons que, si notre approche obtient de bons résultats sur DBpedia-Freq, elle est encore incapable de traiter la taxonomie DBpedia complète. En effet, on est confronté à une tension entre gestion des classes rares et taille des données : d'un côté, si l'on maintient un jeu de données d'entrée de 50 000 entités, alors les classes rares de DBpedia sont insuffisamment représentées dans les données et l'algorithme ne parvient pas à les placer correctement dans la taxonomie prédite. À l'inverse, si l'on augmente la taille des données

d'entrée, la complexité de l'algorithme de regroupement augmente de façon cubique, et le temps de calcul devient prohibitif. Dans la section 6.2, nous esquissons une solution à ce problème. Enfin, au chapitre suivant, nous proposons une solution dans le cas où l'on autorise l'accès aux données du graphe, et pas seulement aux plongements vectoriels.

4.4 Hyperparamètres

Cette section discute le choix des principaux hyperparamètres du modèle, c'est-à-dire des paramètres qui ne sont pas appris à partir des données mais fixés avant l'entraînement. Les deux paramètres du regroupement hiérarchique, distance et critère de liaison, sont communs aux méthodes MLI et MLM ; ils sont traités dans la section suivante. Les deux autres sections sont consacrées à des paramètres spécifiques à MLM : la base du softmax β et le seuil de probabilité t.

4.4.1 Regroupement hiérarchique

L'étape de regroupement hiérarchique consiste à regrouper des entités géométriquement proches : cela suppose donc de définir une distance d entre entités. Dans notre cas, on a expérimenté trois distances classiques de $\mathbb{R}^n$: la distance euclidienne, la distance (ou plus proprement dissimilarité) cosinus, et la distance L_1. Cette dernière, parfois nommée distance de Manhattan, s'écrit, pour deux vecteurs $\mathbf{u} = (u_1, u_2, \ldots, u_n)$ et $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ de $\mathbb{R}^n$:

$$\|\mathbf{u} - \mathbf{v}\|_1 = \sum_{i=1}^{n} |u_i - v_i| \tag{4.41}$$

Le deuxième paramètre du regroupement est le critère de liaison à utiliser. Les critères considérés ici sont détaillés dans la section 4.2.1 et sont au nombre de quatre : saut minimum, moyen et maximum, et distance de Ward, ce dernier critère étant propre à la distance euclidienne. Cela donne donc 10 combinaisons possibles pour l'étape de regroupement hiérarchique.

Pour évaluer les performances moyennes de chaque combinaison de paramètres, on calcule la mesure F_1 de chaque combinaison, moyennée sur l'ensemble des modèles de plongements, et ce, pour les deux modes d'extraction MLI et MLM, et pour les deux modes d'évaluation direct et transitif. Le résultat est tracé à la figure 4.6.

On observe que les deux meilleures combinaisons sont, dans tous les cas, la distance cosinus avec saut moyen (abrégé en «cos-moy» dans la suite) et la distance euclidienne avec critère de Ward (abrégé en «euc-ward»). Pour la méthode MLI, cos-moy est nettement supérieur à

euc-ward dans les deux modes d'évaluation ; pour la méthode MLM, les résultats des deux combinaisons sont très proches.

Enfin, notons que la plupart de nos essais avec le critère du saut minimum n'ont pas abouti. Le saut minimum tend à renforcer les plus gros clusters à chaque étape [87], ce qui produit des arbres-peignes, ou *arbres dégénérés*. Un arbre-peigne est un arbre tel que chaque cluster est soit un cluster singleton, soit la fusion d'un cluster singleton avec un autre cluster (c'est donc l'opposé d'un arbre équilibré). Confrontés à cette structure particulière, nos deux modèles MLI et MLM obtiennent des résultats nuls dans la plupart des cas.

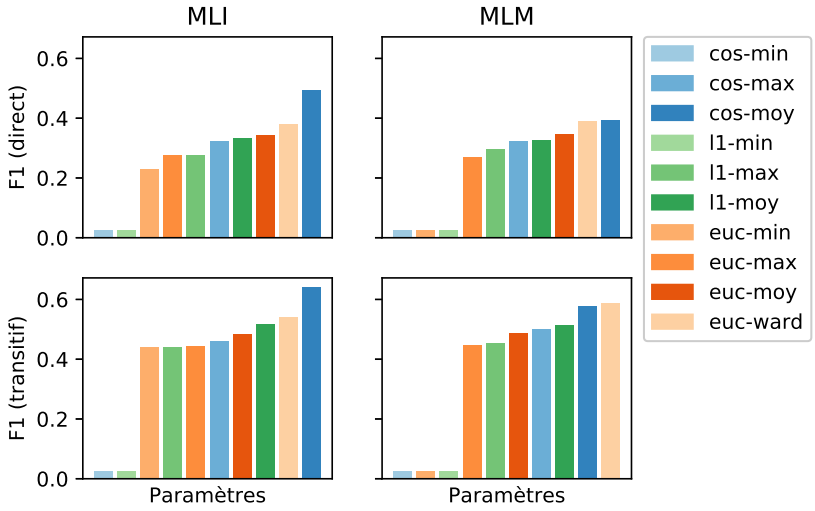


Figure 4.6 Mesures F_1 obtenues pour différentes combinaisons de distances (euclidienne, cosinus, L_1) et de critères de liaison (*min* pour saut minimal, *max* pour saut maximal, *moy* pour saut moyen, *ward* pour distance de Ward). Les scores sont moyennés sur tous les modèles de plongement, et calculés selon l'évaluation directe (*en haut*) et transitive (*en bas*), pour les deux méthode de liaison injective (MLI, *à gauche*) et de liaison multiple (MLM, *à droite*).

4.4.2 Base du softmax et seuil de probabilité

Les deux autres hyperparamètres sont propres à la méthode de liaison multiple : il s'agit de la base β du softmax, et du seuil de probabilité t. La base du softmax contrôle le piqué des distributions de probabilité $P(Z_t = C)$ autour des clusters qui ont la plus forte mesure F_1 : une valeur élevée concentre l'essentiel de la masse de probabilité sur le meilleur cluster, tandis qu'une valeur nulle distribue cette masse à tous les clusters. Le seuil de probabilité t est utilisé pour filtrer la liste d'axiomes de subsumption lors de l'étape de reconstruction taxonomique : seuls les axiomes $(B \sqsubset A)$ qui vérifient $P(B \sqsubset A) > t$ sont conservés et ajoutés

au graphe acyclique G. Dans toute la suite, le modèle de plongement utilisé est TransE et le regroupement se fait avec la distance cosinus et le critère du saut moyen, puisque ce sont les paramètres qui donnent le meilleur F_1 sur toutes les évaluations.

Un jeu d'évaluation X est constitué d'un jeu de données d'entrée $\mathcal{D}$, contenant des plongements vectoriels et les types correspondants, et d'une taxonomie de référence T^*. Pour des paramètres fixés, on peut extraire une taxonomie $\hat{T}$ à partir de $\mathcal{D}$, comparer cette taxonomie à la référence T^*, et calculer les métriques précédentes : précision p, rappel r et F_1, de façon directe et transitive.

Pour un jeu d'évaluation X, on choisit comme métrique d'évaluation la moyenne des deux mesures F_1 : c'est la mesure la plus synthétique, puisque le F_1 est une moyenne de la précision et du rappel, et que le F_1 moyen est lui-même la moyenne du F_1 direct et du F_1 transitif ; on le note $F_1^M(X, \beta, t)$ pour mettre en évidence la dépendance en les paramètres t et β, ainsi qu'en les données d'évaluation X. On peut alors chercher les valeurs optimales des paramètres :

$$\beta^*(X), t^*(X) = \underset{\substack{t\in[0,1]\\ \beta\in\mathbb{R}_+}}{\arg\max}\, F_1^M(X, \beta, t) \tag{4.42}$$

Ces paramètres optimaux peuvent par exemple être estimés avec une *grid search* : on discrétise l'espace de recherche, ce qui nous donne un nombre fini de combinaisons de paramètres possibles, et on évalue l'algorithme sur chacune de ces combinaisons. Par exemple, t varie dans $[0, 1]$ et β dans $\mathbb{R}_+$: on peut diviser le premier intervalle en $\{0, 0.25, 0.5, 0.75, 1\}$ et le second en $\{0, 100, 500, 1000\}$, soit cinq valeurs possibles pour t et quatre pour β, donc vingt combinaisons à essayer au total. Cette recherche peut être améliorée en tirant aléatoirement des couples de $[0, 1] \times \mathbb{R}_+$, ce qui permet généralement de mieux couvrir l'espace de recherche [121].

En pratique, on veut pouvoir appliquer notre approche sur des données d'entrée pour lesquelles on ne dispose pas de taxonomie de référence, puisque le but est précisément d'extraire une telle taxonomie. C'est un problème non-supervisé, on ne peut donc pas utiliser de validation croisée. De plus, les valeurs de β^* et t^* optimales calculées pour un certain jeu de données ne sont *a priori* pas optimales pour un autre jeu de données. Plutôt que calculer β^* et t^*, on souhaite plutôt trouver des valeurs «raisonnables» de β et de t, autrement dit des valeurs permettant, en moyenne, de ne pas trop s'éloigner de l'optimum.

Pour cela, on choisit plusieurs jeux d'évaluation $X_1, X_2, \ldots, X_r$ (par exemple tirés de plusieurs graphes de connaissances pour lesquels on dispose d'une taxonomie de référence connue), et on définit un ensemble d'évaluation $\mathcal{X} = (X_1, X_2, \ldots, X_r)$ qui regroupe ces jeux de référence. On cherche alors des valeurs de t et de β qui permettent d'approcher le score F_1 optimal sur

chacun des $X \in \mathcal{X}$:

$$\hat{\beta}, \hat{t} = \underset{\substack{t \in [0,1] \\ \beta \in \mathbb{R}_+}}{\arg\max} \frac{1}{|\mathcal{X}|} \sum_{X \in \mathcal{X}} \frac{F_1^M(X, \beta, t)}{F_1^M(X, \beta^*(X), t^*(X))} \tag{4.43}$$

Le quotient $\frac{F_1^M(X,\beta,t)}{F_1^M(X,\beta^*(X),t^*(X))}$ permet d'exprimer les performances du modèle pour t et β comme une fraction des performances optimales – en effet, l'enjeu ici n'est pas les performances absolues du modèle, mais l'écart entre les performances maximales possibles et les performances atteintes avec β et t. On appelle cette quantité le *score* associé aux paramètres β et t, qui varie entre 0 et 1 :

$$S(\beta, t; X) = \frac{F_1^M(X, \beta, t)}{F_1^M(X, \beta^*(X), t^*(X))} \tag{4.44}$$

Et l'on cherche à maximiser le score moyen associé à un couple de paramètres :

$$\bar{S}(\beta, t) = \frac{1}{|\mathcal{X}|} \sum_{X \in \mathcal{X}} S(\beta, t; X) \tag{4.45}$$

Alternativement, on pourrait choisir de maximiser le pire des scores, plutôt que la moyenne des scores. Le score minimal d'un couple de paramètres est le pire score obtenu sur l'ensemble des jeux de référence :

$$S_{\min}(\beta, t) = \min_{X \in \mathcal{X}} S(\beta, t; X) \tag{4.46}$$

En comparaison du score moyen, maximiser le score minimal permet de se prémunir contre des valeurs de paramètres qui fonctionnent bien dans le cas général mais donnent de très mauvais résultats dans certains cas :

$$\hat{\beta}, \hat{t} = \underset{\substack{t \in [0,1] \\ \beta \in \mathbb{R}_+}}{\arg\max} S_{\min}(\beta, t) \tag{4.47}$$

Pour construire l'ensemble $\mathcal{X}$, on construit aléatoirement cinq sous-taxonomies de DBpedia de différentes tailles et de différentes profondeurs ; chacune de ces sous-taxonomies permet de former un jeu d'évaluation. On obtient donc cinq jeux d'évaluation $X_1, \ldots, X_5$, qui constituent l'ensemble d'évaluation $\mathcal{X}$.

L'enjeu des paragraphes suivants est double : d'une part, trouver des valeurs satisfaisantes pour $\hat{\beta}$ et $\hat{t}$, et d'autre part, comprendre l'effet de β et de t sur la taxonomie extraite.

Choix de β et t

On applique ici notre méthode d'extraction de taxonomie aux cinq sous-taxonomies de $\mathcal{X}$, pour vingt valeurs de β réparties linéairement dans l'intervalle $[1, 200]$. Pour chacune des valeurs de β, on fait varier t dans l'intervalle $[0, 1]$ (à nouveau en séparant l'intervalle en vingt valeurs espacées linéairement), ce qui nous permet d'obtenir empiriquement le seuil optimal $t^*(\beta)$. On obtient ainsi un tableau reliant chaque couple de paramètres $(\beta, t) \in [0, 200] \times [0, 1]$ et chaque sous-taxonomie X_k au score associé $S(\beta, t; X_k)$.

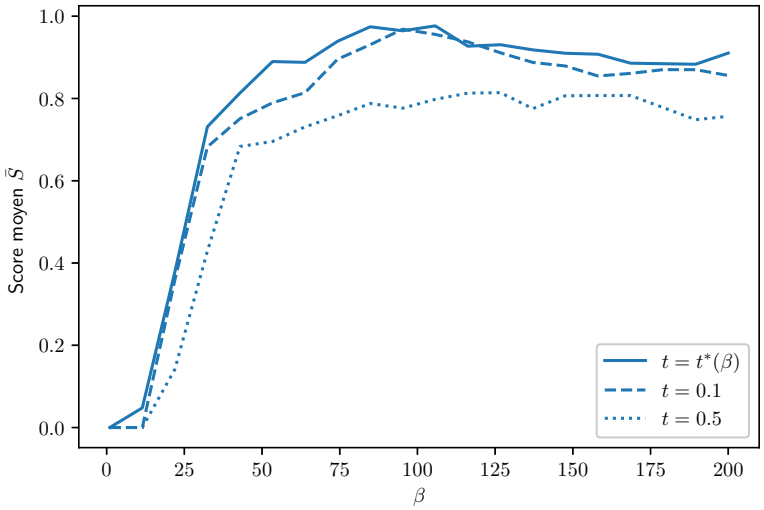


Figure 4.7 Résultats de l'extraction de taxonomie pour différentes valeurs du paramètre β, pour le seuil optimal $t^*(\beta)$ et pour deux seuils fixes $t = 0.1$ et $t = 0.5$. L'ordonnée représente le score moyen $\bar{S}(\beta, t)$, qui est la moyenne des mesures F_1^M sur les cinq sous-taxonomies de $\mathcal{X}$, exprimées comme une fraction de l'optimum.

On trace alors les résultats obtenus, pour trois valeurs de t : deux valeurs fixes, $t = 0.1$ et $t = 0.5$, et la valeur optimale $t = t^*(\beta)$ qui indique le meilleur score que l'on puisse espérer atteindre. Le résultat est présenté dans la figure 4.7. Pour les trois valeurs de t, on observe le comportement suivant : la mesure F_1 moyenne est nulle pour $\beta = 1$, augmente abruptement, atteint un maximum autour de $\beta = 100$, puis diminue progressivement.

Pour mieux comprendre les variabilités de $S(\beta, t; X)$ d'un jeu de données à l'autre, on affiche la courbe des scores en fonction de β pour chacune des cinq sous-taxonomies. Le résultat est présenté dans la figure 4.8. On observe des disparités importantes d'une taxonomie à l'autre, pour les trois valeurs de t.

Dans le cas $t = t^*$, le score maximal est obtenu dès $\beta = 50$ pour l'une des taxonomies, et

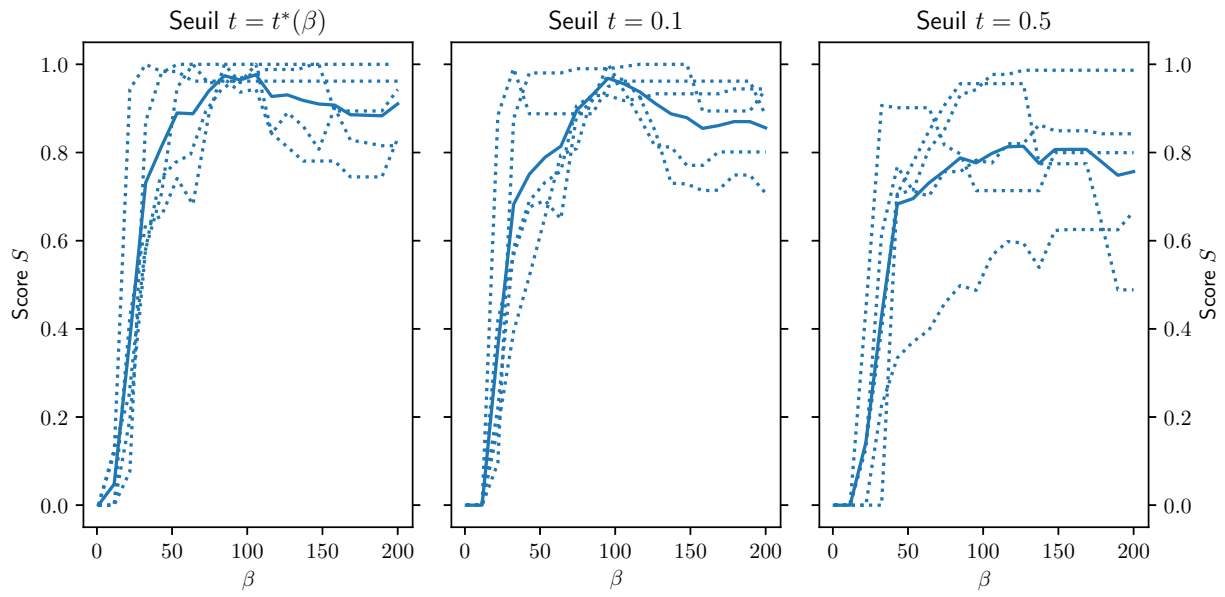


Figure 4.8 Scores obtenus pour différentes valeurs de β dans l'intervalle $[0, 200]$, pour le seuil optimal $t^*(\beta)$ (*à gauche*), et pour deux seuils fixes 0.1 et 0.5. Chaque ligne pointillée représente les résultats pour une sous-taxonomie X_i ; la ligne continue représente la moyenne sur les cinq sous-taxonomies.

pour $\beta = 150$ pour une autre. Dans certains cas, les résultats diminuent sitôt le maximum atteint ; dans d'autres, on observe au contraire un long plateau. Toutefois, pour toutes les taxonomies, le score atteignable en $\beta = 100$ est proche du score maximal, c'est-à-dire que $S(\beta = 100, t = t^*(\beta); X_i)$ est proche de $S(\beta^*, t^*; X_i)$ pour tous les $X_i \in \mathcal{X}$. Cela donne un score moyen $\bar{S}(\beta = 100, t = t^*(\beta))$ égal à $97,7\%$, et un score minimal $S_{\min}(\beta, t)$ égal à $94,4\%$.

Pour $t = 0.1$, le comportement est globalement le même que dans le cas précédent. À nouveau, on observe d'une part des disparités de comportement entre les taxonomies, et d'autre part un rapprochement des courbes autour de $\beta = 100$; choisir cette valeur de β conduit à un score moyen de 96.9% et à un score minimal de 92.9%. Dans les deux cas $t = t^*$ et $t = 0.1$, prendre comme critère le score moyen ou le score minimal est indifférent : la valeur de β qui maximise l'un est également celle qui maximise l'autre.

Le cas $t = 0.5$ est différent. Une nouvelle fois, on observe une augmentation abrupte du score dans l'intervalle $[0, 50]$, mais les disparités entre taxonomies sont fortement accrues. De plus, les scores sont significativement inférieurs au cas précédent ; le score ne dépasse pas 90% sur trois des cinq taxonomies. Enfin, les scores maximaux pour chaque taxonomie sont atteints en des points éloignés, ce qui grève le score moyen. Ce dernier atteint un maximum en $\beta = 125$ et vaut alors 81.4% ; le score minimal, lui, atteint son maximum en $\beta = 150$ et vaut alors

62.5%.

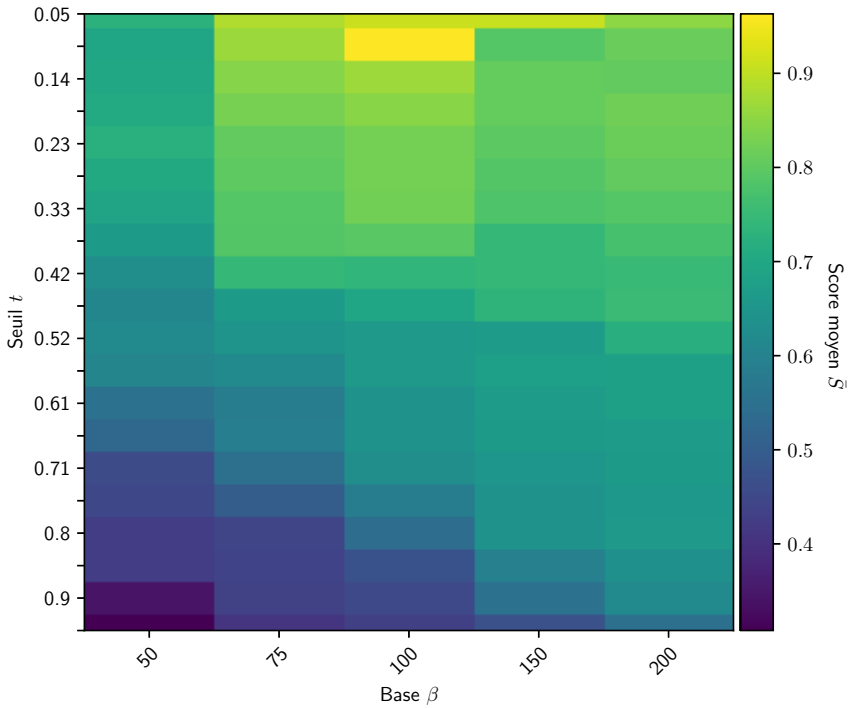


Figure 4.9 Scores moyens obtenus sur cinq sous-taxonomies pour différentes valeurs de β et de t.

Finalement, pour explorer d'autres valeurs du seuil t, on affiche directement le score moyen $\bar{S}(\beta, t)$ pour vingt valeurs de t dans $[0, 20]$ et pour $\beta \in \{50, 75, 100, 150, 200\}$. Les résultats sont affichés à la figure 4.9. À nouveau, les meilleurs résultats sont observés pour $\beta = 75$ ou $\beta = 100$ et pour des seuils faibles. On observe un maximum du score moyen $\bar{S}$ pour les valeurs $\beta = 100, t = 0.1$. De toutes ces observations, on utilisera par défaut les valeurs $\hat{\beta} = 100$ et $\hat{t} = 0.1$.

Cependant, cette recherche d'hyperparamètres demeure très limitée. D'une part, on s'est restreint aux plongements TransE avec distance cosinus et critère du saut moyen : d'autres modèles de plongement ou d'autres paramètres de regroupement pourraient conduire à d'autres valeurs de $\hat{\beta}$ et $\hat{t}$. De plus, les cinq sous-taxonomies de référence sont toutes issues de DBpedia, et ne représentent pas nécessairement l'ensemble des graphes de connaissances. Aussi, les valeurs données ici ne doivent pas être considérées comme des choix définitifs.

Influence de β

Pour observer qualitativement l'effet de β sur les probabilités des axiomes de subsumption, on affiche les superclasses les plus probables de quelques classes choisies : `dbo:Athlete`, `dbo:Song`, `dbo:Event` (en français : Athlète, Chanson, Événement). Par superclasse la plus probable d'une classe x, on entend la classe y telle que la probabilité $P(x \sqsubset y)$ soit supérieure à toutes les autres probabilités $P(x \sqsubset y')$. Le résultat est affiché dans le tableau 4.2. Dans la taxonomie de référence, `dbo:Athlete` et `dbo:Song` ont deux superclasses (`dbo:Person`, `dbo:Agent` et `dbo:MusicalWork`, `dbo:Work` respectivement), tandis que `dbo:Event` n'en a aucune.

Tableau 4.2 Les trois superclasses les plus probables de `dbo:Athlete` (*en haut*), de `dbo:Song` (*au milieu*), et de `dbo:Event` (*en bas*), avec les probabilités p associées, pour différentes valeurs de β entre 1 et 100 (une ligne pour chaque valeur de β, les préfixes `dbo:` sont omis par souci de concision).

	1		2		3	
β	classe	p	classe	p	classe	p
1	Agent	0.00	Event	0.00	Species	0.00
5	Event	0.00	Agent	0.00	Species	0.00
10	Event	0.02	SportsSeason	0.01	Document	0.01
30	Agent	0.97	Person	0.00	Species	0.00
50	Agent	1.00	Person	0.02	Organisation	0.00
100	Agent	1.00	Person	0.41	Organisation	0.00

	1		2		3	
β	classe	p	classe	p	classe	p
1	Agent	0.00	Event	0.00	Species	0.00
5	Event	0.00	Agent	0.00	Species	0.00
10	Event	0.02	SportsSeason	0.01	Document	0.01
30	Agent	0.25	MusicalWork	0.12	Species	0.06
50	Work	0.87	MusicalWork	0.67	Single	0.20
100	Work	1.00	MusicalWork	0.81	Single	0.14

	1		2		3	
β	classe	p	classe	p	classe	p
1	Agent	0.00	Species	0.00	Place	0.00
5	Agent	0.00	Species	0.00	Document	0.00
10	SportsSeason	0.01	Document	0.00	Agent	0.00
30	Work	0.00	Eukaryote	0.00	Place	0.00
50	Work	0.00	Place	0.00	ArchitecturalStructure	0.00
100	Person	0.00	Organisation	0.00	ArchitecturalStructure	0.00

Étudions d'abord les deux premiers tableaux. Dans les deux cas, on observe des résultats

incohérents pour $\beta < 30$: d'une part, les superclasses les plus probables ne correspondent pas aux vraies superclasses ; d'autre part, les probabilités de toutes les superclasses sont très proches de zéro, ce qui ne permet pas de distinguer les axiomes pertinents des autres. Lorsque β augmente (ici $\beta = 30$), les vraies superclasses commencent à émerger. Pour `dbo:Athlete`, les deux superclasses les plus probables sont bien les vraies superclasses ; l'une d'elles (`dbo:Agent`) est prédite avec probabilité 0.97, ce qui est satisfaisant ; en revanche, l'autre (`dbo:Person`) est prédite avec une probabilité très faible. Il faut attendre $\beta = 100$ pour discriminer nettement les classes valides (`dbo:Agent` et `dbo:Person`, prédites avec probabilités 1 et 0.41 respectivement) des classes invalides (`dbo:Organisation`, avec probabilité 0.0). Les mêmes observations s'appliquent à `dbo:Song` : les vraies superclasses émergent et se détachent progressivement des fausses superclasses. Dans ce cas, on note aussi que `dbo:Single`, qui n'est pas une superclasse de `dbo:Song` mais une classe sémantiquement proche, se voit attribuer une probabilité relativement élevée (entre 0.1 et 0.2).

Enfin, dans le cas de `dbo:Event`, le modèle prédit correctement des probabilités nulles pour tous les axiomes de subsumption possibles, et ce, quelle que soit la valeur de β. Cela correspond bien à la réalité, puisque `dbo:Event` ne possède aucune superclasse.

Si l'on examine maintenant les taxonomies extraites pour différentes valeurs de β, on obtient la figure 4.10 (seules quelques classes sont représentées, pour faciliter l'affichage). Comme prévu, le cas $\beta = 1$ n'aboutit à rien : on obtient un arbre dégénéré. Quand β augmente, les deux classes de niveau 1 (`dbo:Place` et `dbo:Agent`) sont identifiées, mais l'une est incorrectement considérée comme sous-classe de l'autre. Aucune hiérarchie ne se distingue sur les autres classes. Puis, pour $\beta = 50$, les deux classes principales sont correctement séparées, et leurs sous-classes attribuées sans erreur. Une hiérarchie commence à se dessiner dans les sous-classes de `dbo:Place`, avec d'un côté les lieux habités (`dbo:PopulatedPlace` et `dbo:Region`), et de l'autre les espaces naturels (`dbo:NaturalPlace`, `dbo:Mountain`, `dbo:Volcano`). Enfin, pour $\beta = 100$, la taxonomie est correctement extraite, avec une seule erreur (`dbo:Volcano` n'est pas sous-classe de `dbo:Mountain` dans la taxonomie de référence, même si cette classification aurait une certaine logique). Ces observations illustrent comment l'émergence progressive des superclasses quand β augmente, décrite dans les paragraphes précédents, se traduit par une structuration croissante de la taxonomie extraite.

Seuil de probabilité

Pour comprendre l'effet du seuil t, on s'intéresse à l'évolution de la précision, du rappel et du F_1 lorsque t varie entre 0 et 1, pour différentes valeurs de β. On calcule la moyenne de ces trois métriques sur les cinq sous-taxonomies utilisées précédemment. Les résultats sont

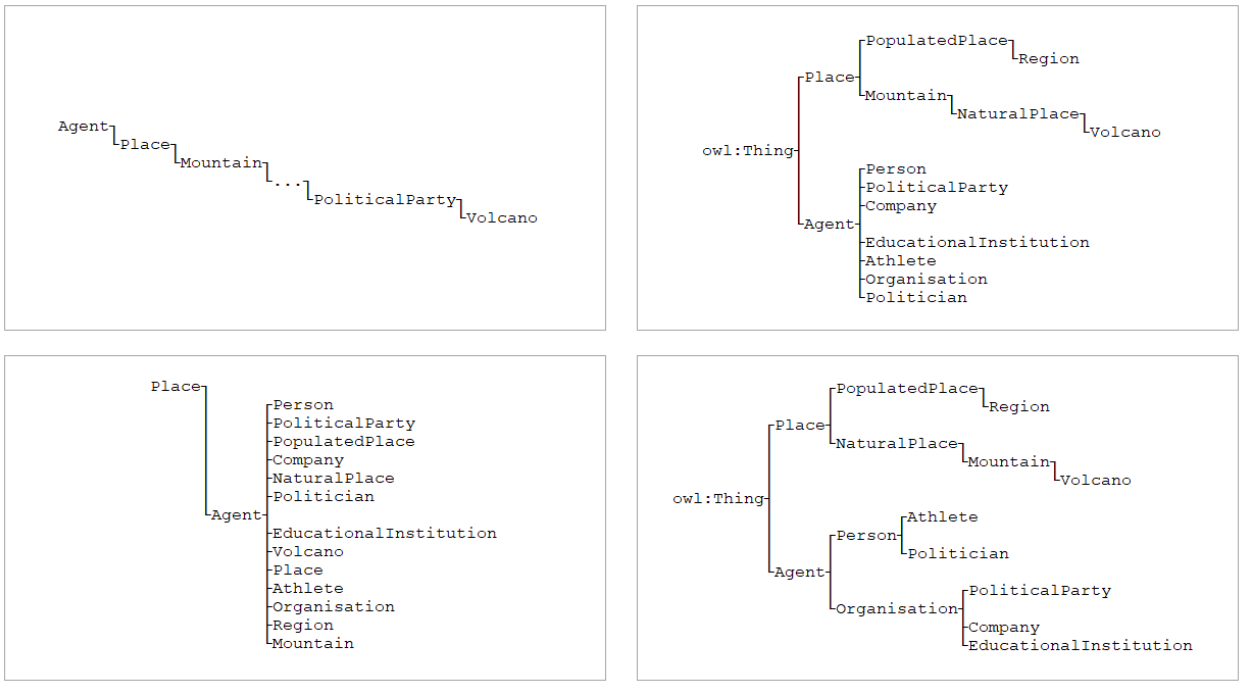


Figure 4.10 Exemples de taxonomies partielles pour $\beta = 1$ (*en haut à gauche*), $\beta = 15$ (*en bas à gauche*), $\beta = 50$ (*en haut à droite*) et $\beta = 100$ (*en bas à droite*). Note : la classe `owl:Thing` n'est pas présente dans les données, elle est ajoutée automatiquement lorsqu'il y a plus d'une classe sans parents dans la taxonomie extraite.

affichés dans la figure 4.11.

En moyenne, on observe les effets suivants : la précision augmente avec t, tandis que le rappel diminue. Ces deux effets étaient attendus : en augmentant le seuil, on élimine les axiomes les plus incertains, c'est-à-dire ceux qui ont le plus de chance d'être faux. La précision, autrement dit la proportion d'axiomes extraits qui sont vrais, augmente donc. Mais, parmi les axiomes éliminés par l'augmentation du seuil, certains étaient vrais : la proportion d'axiomes vrais qui sont effectivement extraits, c'est-à-dire le rappel, diminue.

Ces tendances sont plus ou moins marquées selon β : la précision augmente d'autant plus avec t que β est faible ; de même, la diminution du rappel avec t semble plus rapide pour les valeurs faibles de β. Selon les valeurs de β, ces deux tendances (augmentation de la précision et diminution du rappel) se compensent plus ou moins tant que t est faible. Dans les cinq cas, le F_1 observé sur l'intervalle $t \in [0, 0.4]$ est proche du F_1 maximal (10% d'écart au maximum). Toutefois, passé un certain seuil proche de $t = 0.5$, l'augmentation de la précision ne suffit plus à compenser la diminution du rappel et la mesure F_1 chute alors, quelle que soit la valeur de β.

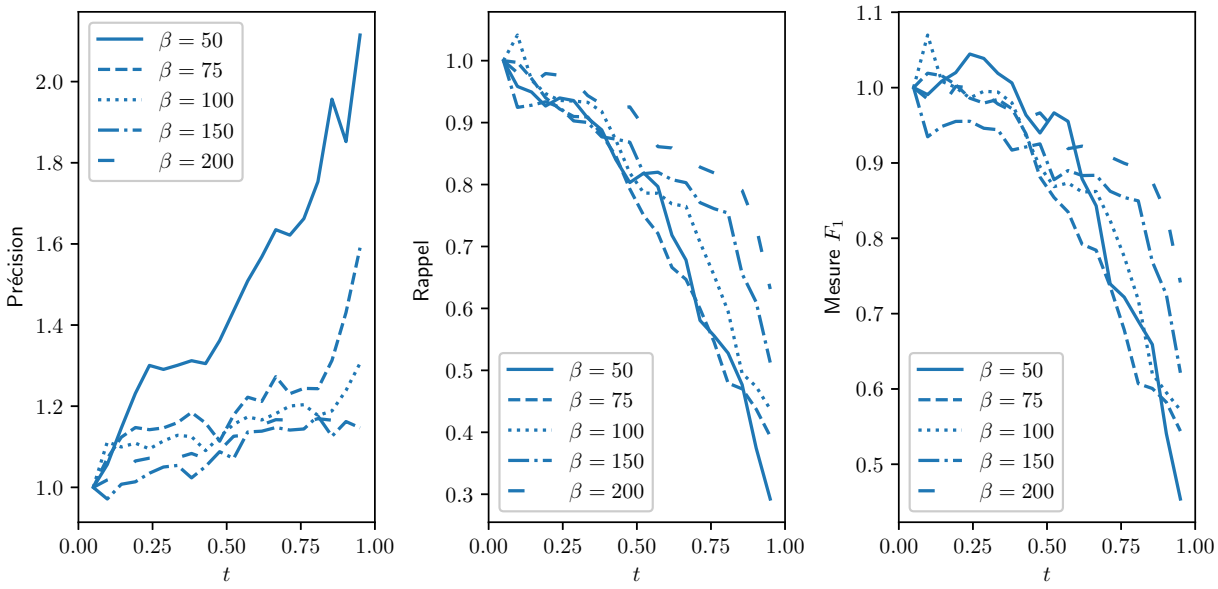


Figure 4.11 Précision, rappel et mesure F_1 moyens, pour différentes valeurs du seuil t et différentes valeurs de β. Pour faciliter la comparaison des tendances, les résultats sont normalisés de façon à avoir chaque ligne commençant à 1 en $t = 0$.

Pour affiner ces observations, on fixe $\beta = 100$, et on affiche le détail des résultats pour chacune des cinq taxonomies X_i dans la figure 4.12. On observe également une grande hétérogénéité d'une taxonomie à l'autre. En particulier, l'augmentation de la précision avec t est très nette pour l'un des modèles, mais moins pour les autres. De plus, la diminution du rappel est très rapide (dès $t > 0.05$) pour deux des modèles ; pour les cinq autres, le rappel est assez stable dans l'intervalle $[0, 0.5]$, et diminue ensuite. Cette hétérogénéité s'observe aussi pour la mesure F_1 : dans un cas, elle est stable dans l'intervalle $[0.1, 0.7]$ et diminue ensuite régulièrement ; dans deux autres cas, elle est stable sur $[0.1, 0.5]$, diminue vers un plateau sur $[0.5, 0.75]$, et diminue encore après 0.75 ; dans les deux derniers cas, elle diminue rapidement dès que t s'écarte de 0. La diversité des comportements d'une taxonomie à l'autre peut s'expliquer par la diversité des taxonomies elles-mêmes. Celles-ci ont en effet été construites avec un nombre de classes et une profondeur variable, dans l'espoir de mieux refléter la diversité des taxonomies du monde réel.

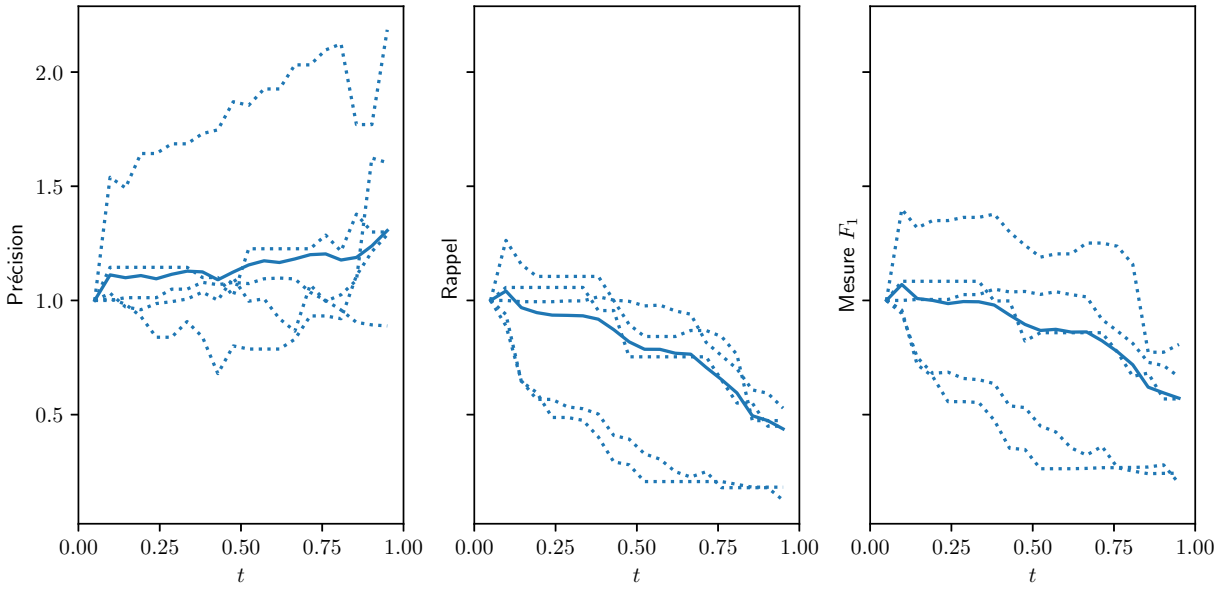


Figure 4.12 Précision, rappel et mesure F_1 mesurés sur cinq sous-taxonomies, pour différentes valeurs du seuil t et pour $\beta = 100$. Chaque ligne pointillée représente les résultats pour l'une des taxonomies, tandis que la ligne pleine est la moyenne des résultats sur les cinq taxonomies. Pour faciliter la comparaison des tendances, les résultats sont normalisés de façon à avoir chaque ligne commençant à 1 en $t = 0$.

CHAPITRE 5 EXTRACTION DE TAXONOMIE EXPRESSIVE

Dans le chapitre précédent, on a montré qu'il était possible de reconstituer une hiérarchie sur les types à partir d'un ensemble de plongements vectoriels typés mais non hiérarchisés. Toutefois, la taxonomie ainsi extraite est limitée aux types déjà connus, et suppose donc une liste de types fixée à l'avance. Ici, au contraire, on cherche à extraire une taxonomie expressive, c'est-à-dire une hiérarchie sur des classes qui peuvent être soit des types existants, soit des types émergeant des données et décrits par des axiomes logiques. On cherche également à décrire les types existants au moyen d'axiomes.

On adopte pour cela un point de départ beaucoup moins restrictif, puisqu'on s'autorise l'accès à tout le graphe de connaissances, et on cherche à combiner les modèles de plongement, qui permettent de détecter des régularités à partir de considérations géométriques, à la richesse des données structurées contenues dans le graphe. L'objectif poursuivi est double : identification de nouveaux types, et caractérisation des types existants. On cherchera ainsi à illustrer le potentiel des méthodes d'extraction taxonomique basées sur le regroupement non-supervisé de plongements vectoriels, par opposition aux méthodes qui manipulent directement des types et à celles qui regroupent les plongements de façon supervisée.

5.1 Motivation et principes généraux

Dans l'approche précédente, la restriction imposée sur les données d'entrée $\mathcal{D}$ a deux conséquences majeures qui rendent l'extraction de taxonomie délicate. On les présente ici, et on propose une solution pour les éviter dans le cas où, comme ici, on s'autorise l'accès à l'intégralité du graphe.

Examinons le cas d'un graphe de connaissances constitué pour moitié d'agents, dont la moitié sont des personnes. Supposons que les personnes soient divisées en dix sous-classes de taille égale (athlètes, artistes, scientifiques, etc.), et les athlètes en vingt sous-classes de taille égale (joueurs de hockey, gymnastes, golfeurs, etc.). Dans un ensemble d'entités $\mathcal{D}$ tirées aléatoirement, un quart des entités seront des personnes, $1/40^{\text{ième}}$ seront des athlètes, et donc seulement $1/800^{\text{ième}}$ seront des joueurs de hockey. Si l'on tient compte des déséquilibres de tailles qui existent dans les graphes réels, ces chiffres peuvent varier. Toutefois, cet exemple illustre la difficulté de l'extraction de taxonomie à partir d'un ensemble de données *fixe* : plus on s'enfonce dans la taxonomie, moins on dispose d'exemples, et cette diminution est exponentielle.

D'autre part, un graphe de connaissances extrait de manière automatique ou semi-automatique est bruité ; ses plongements vectoriels le sont aussi. Le regroupement hiérarchique est donc imparfait : si l'on suppose qu'à chaque division d'un cluster en deux sous-clusters, une certaine proportion des entités est assignée au mauvais cluster, alors on assiste à une propagation des erreurs de clustering dans l'arbre, ce qui rend les clusters profonds délicats à interpréter et étiqueter.

Sur la base de ces constats, nous proposons donc une méthode de prélèvement-regroupement, qui consiste à :

a. prélever aléatoirement des entités dans le graphe en s'appuyant sur les informations que l'on a déjà extraites, afin d'obtenir des entités de départ de plus en plus homogènes et de plus en plus spécifiques. On évite ainsi la diminution exponentielle du nombre d'entités par type ;

b. regrouper ce nouveau groupe d'entités. Ce faisant, on a moins besoin de s'enfoncer loin dans l'arbre de clustering, et on limite ainsi la propagation des erreurs.

Chaque étape de prélèvement-regroupement peut être vue comme une tâche d'extraction de taxonomie sur un sous-ensemble du graphe complet ; les sous-taxonomies ainsi extraites s'ajoutent les unes aux autres pour former une taxonomie unique. Lors des premières étapes, on extrait des taxonomies très générales sur les classes de haut niveau, puis on augmente progressivement la spécificité des entités prélevées, et donc la spécificité des sous-taxonomies extraites. Cette idée est décrite dans la section 5.2.1.

À ce principe général s'ajoute une méthode pour parcourir les arbres de clustering (section 5.2.2) et les étiqueter avec des axiomes expressifs (section 5.2.3). Puisque les plongements vectoriels sont géométriquement proches au sein d'un cluster, les entités associées sont sémantiquement proches : on mène donc l'extraction d'axiomes cluster par cluster ; les liens entre clusters permettent de définir un ensemble d'exemples positifs et un ensemble d'exemples négatifs sur lesquels mener une recherche inductive d'axiomes.

Dans la section 5.3, on applique notre méthode sur DBpedia, et on discute les résultats obtenus.

Remarque DBpedia, comme la plupart des graphes de connaissances, fonctionne sous l'hypothèse du monde ouvert : si un triplet qui n'est pas présent dans le graphe est soit faux, soit simplement manquant. Aussi, dans la suite, lorsqu'on écrit qu'une entité x ne vérifie pas un prédicat logique P, ou plus rapidement $\neg P(x)$, il s'agit simplement d'un raccourci pour signifier que l'information «x vérifie le prédicat P» est soit fausse, soit manquante dans le graphe.

Notations On utilise les notations usuelles de la théorie des graphes : un graphe G est représenté par une paire (V, E) avec V ses sommets, et $E \subseteq V \times V$ ses arêtes. En particulier, si G est un arbre de clustering, l'arête $(C, C') \in E$ indique une relation d'inclusion $C \subset C'$; si G est une taxonomie, l'arête $(A, B) \in E$ indique une subsumption $A \sqsubseteq B$.

5.2 Méthode proposée

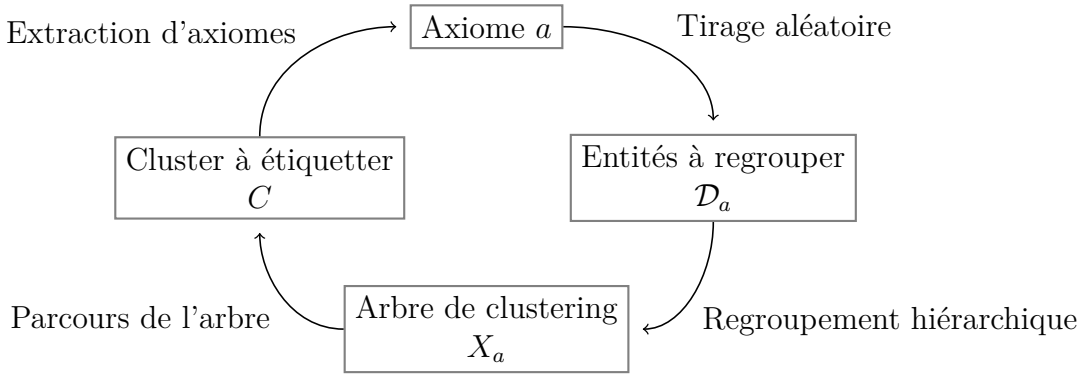


Figure 5.1 Principe général de la méthode d'extraction de taxonomie : à partir d'un axiome initial a, on prélève aléatoirement des entités qui vérifient a, on les regroupe hiérarchiquement, et on parcourt l'arbre obtenu pour extraire de nouveaux axiomes capables de décrire les clusters. On répète ces étapes pour construire progressivement une taxonomie complète du graphe ; l'initialisation se fait avec $a = \top$.

La méthode d'extraction de taxonomie expressive commence par une phase de regroupement hiérarchique. Dans la méthode précédente, on avait comme seules données d'entrée un ensemble fixé d'entités typées $\mathcal{D}$. Ici au contraire, on s'autorise l'accès à tout le graphe, donc l'ensemble $\mathcal{D}$ sur lequel s'opère le regroupement hiérarchique est variable et change au cours de l'exécution de l'algorithme. À chaque étape t, on constitue un nouveau jeu de données $\mathcal{D}_t$. On effectue alors un regroupement hiérarchique sur les plongements des entités de $\mathcal{D}_t$, puis on étiquette les clusters obtenus avec des axiomes logiques (l'extraction d'axiomes à partir de clusters est détaillée dans la section 5.2.3). Chaque nouvel axiome extrait sert alors à créer un nouveau jeu de données $\mathcal{D}_{t+1}$, sur lequel on répète l'étape précédente, ce qui permet d'étendre itérativement la taxonomie prédite. L'algorithme 2 présente les grandes étapes de cette méthode, et la figure 5.1 les résume.

5.2.1 Prélèvement aléatoire et regroupement hiérarchique

L'idée est d'extraire des axiomes itérativement, et de les organiser au fur et à mesure au sein d'une taxonomie $\hat{T}$. On dispose donc de la taxonomie $\hat{T}$ en cours de construction, et

Algorithme 2 : Algorithme d'extraction de taxonomie expressive. Il consiste en deux étapes principales – l'une de prélèvement et de regroupement d'entités, l'autre d'extraction d'axiomes – qui sont répétées de façon à construire récursivement la taxonomie $\hat{T}$. Les différentes fonctions utilisées sont détaillées dans la suite de la section.

Entrée : un graphe de connaissances $\mathcal{KG}$
Sortie : une taxonomie expressive $\hat{T}$
Paramètres : nombre d'entités à prélever à chaque étape N_e

```
// taxonomie construite récursivement :
```
$\hat{T} \leftarrow (\{\top\}, \varnothing)$;
```
// file d'axiomes à traiter :
```
$\mathcal{AT} \leftarrow \{\top\}$;
tant que $\mathcal{AT}$ *est non-vide* **faire**
  $a \leftarrow$ premier axiome de la file $\mathcal{AT}$;
  `// 1 : prélèvement/regroupement`
  $\mathcal{D}_a \leftarrow$ `Preleve(`$a, \mathcal{KG}, N_e$`)` ;
  $X_a \leftarrow$ `Regroupe(`$\mathcal{D}_a$`)` ;
  `// 2 : extraction d'axiomes`
  $T_a \leftarrow$ `EtiquetteArbre(`X_a`)` ;
  pour *chaque nouvel axiome* b *de* T_a **faire**
    ajouter b à $\mathcal{AT}$;
  $\hat{T} \leftarrow$ `Fusionne(`$\hat{T}, T_a$`)` ;
retourner X

d'une file d'axiomes à traiter $\mathcal{AT}$. La file d'axiomes à traiter contient des axiomes que l'on a extraits précédemment. On sait donc qu'ils représentent des classes du graphe, on connaît leur superclasse, mais on ne connaît pas encore leurs sous-classes : l'objectif est d'identifier leurs sous-classes et de les ajouter à $\hat{T}$.

Pour la première étape, $\mathcal{AT}$ est initialisée à $\{\top\}$: le concept universel $\top$ est en effet le seul axiome dont on sache *a priori* qu'il représente une classe du graphe, puisqu'il est par définition vérifié par toutes les entités du graphe. Il sert donc de racine à la taxonomie. De plus, comme on ne connaît qu'une seule classe et aucun axiome de subsumption, la taxonomie $\hat{T}$ est réduite à un seul sommet $\top$ et ne contient aucune arête.

On va alors retirer le premier élément a de la file $\mathcal{AT}$. On prélève aléatoirement, dans le graphe $\mathcal{KG}$, N_e entités parmi celles qui vérifient a ; cette étape correspond à la fonction `Preleve` de l'algorithme 2. Dans notre implémentation, le graphe est représenté grâce à une librairie Python spécifique, mais dans d'autres contextes, le langage de requête SPARQL pourrait être utilisé à cette fin. Lors de la première étape, $\mathcal{AT}$ ne contient que l'élément $\top$, donc $a = \top$ et il suffit alors de prélever aléatoirement des entités du graphe. Les plongements de ces entités donnent un nuage de points $\mathcal{D}_a$ de dimension $N_e \times d$, avec d la dimension des

plongements.

Sur ce nuage de point $\mathcal{D}_a$, on effectue un regroupement hiérarchique, similaire à celui décrit dans la section 4.2.1, qui correspond à la fonction `Regroupe` de l'algorithme 2. Ici, on a choisi les paramètres de regroupement qui donnaient les meilleurs résultats pour l'extraction de taxonomie non-expressive (voir la section 4.3.3 au chapitre précédent), c'est-à-dire la distance cosinus comme mesure de similarité, et le saut moyen comme critère de liaison. Autrement dit, on fusionne les clusters qui ont la plus faible distance cosinus moyenne entre leurs éléments. À l'étape t, notons $\mathcal{C}_t$ les clusters existants ; les clusters C_1 et C_2 à fusionner sont choisis selon l'équation :

$$C_1, C_2 = \underset{C_1, C_2 \in \mathcal{C}_t}{\arg\min} \frac{1}{|C_1| \times |C_2|} \sum_{\mathbf{e_1} \in C_1} \sum_{\mathbf{e_2} \in C_2} d_{\cos}(\mathbf{e_1}, \mathbf{e_2}) \tag{5.1}$$

Le résultat est un arbre binaire, noté X_a, dont la racine est $\mathcal{D}_a$ tout entier, et dont les feuilles sont les N_e éléments de $\mathcal{D}_a$.

Cet arbre de clustering X_a est alors parcouru, étiqueté et transformé en une taxonomie partielle T_a, éventuellement vide : cette étape de parcours et de transformation correspond à la fonction `EtiquetteArbre` de l'algorithme 2, et est décrite en détail dans la section suivante.

Lors de la première étape, l'axiome de départ $\top$ possède des sous-classes et la taxonomie extraite T_a n'est donc pas vide. On ajoute alors T_a à $\hat{T}$, à l'emplacement de a, ce qui correspond à la fonction `Fusionne` de l'algorithme 2. Tous les nouveaux axiomes contenus dans T_a sont alors ajoutés à la file d'axiomes à traiter $\mathcal{AT}$.

On peut alors répéter ce procédé, jusqu'à épuisement de $\mathcal{AT}$: tant que $\mathcal{AT}$ n'est pas vide, on retire son premier élément a, on prélève des entités parmi celles qui vérifient l'axiome a pour former le nuage de points $\mathcal{D}_a$, on regroupe ces points en un arbre de clustering X_a, et on en extrait une taxonomie partielle T_a. Si T_a est vide, on poursuit la recherche avec d'autres axiomes de $\mathcal{AT}$. Sinon, on ajoute les nouveaux axiomes contenus dans T_a à la file d'axiomes à traiter, et on ajoute T_a à $\hat{T}$.

Donnons un exemple d'exécution de l'algorithme sur DBpedia. On commence par prélever des entités aléatoirement dans tout le graphe. Après regroupement et étiquetage, on identifie les classes `dbo:Agent`, `dbo:Work`, `dbo:Event`, `dbo:Species`, `dbo:Place` : ces classes sont ajoutées à $\hat{T}$ en tant que sous-classes de $\top$, et elles sont également ajoutées à $\mathcal{AT}$. À l'étape suivante, l'axiome de départ est le premier axiome de $\mathcal{AT}$, c'est-à-dire $a =$ `dbo:Agent`. On prélève alors des instances de `dbo:Agent`, on regroupe leurs plongements, et on étiquette l'arbre de clustering, ce qui permet d'identifier deux nouveaux axiomes `dbo:Person` et `dbo:Organisation`.

Ces deux axiomes sont ajoutés à $\mathcal{AT}$, ainsi qu'à $\hat{T}$ comme sous-classes de `dbo:Agent`. La recherche continue ensuite avec les autres axiomes de $\mathcal{AT}$: d'abord `dbo:Work`, puis `dbo:Event`, etc. Au bout d'un certain nombre d'étapes, les classes feuilles ont été atteintes, et il n'est plus possible d'extraire de nouveaux axiomes : la file $\mathcal{AT}$ se vide progressivement. Une fois qu'elle ne contient plus aucun axiome, l'algorithme s'arrête, et $\hat{T}$ est renvoyée.

5.2.2 Parcours et étiquetage de l'arbre

Dans cette section, on se donne une fonction d'extraction d'axiomes α, telle que $\alpha(C)$ est un axiome logique qui qualifie le cluster C, si un tel axiome existe, et qui renvoie un symbole spécial `indéfini` dans le cas contraire. Un exemple d'une telle fonction est donné dans la section suivante.

On effectue un parcours de l'arbre X_a, en excluant la racine a qui est déjà étiquetée : pour un cluster C, on calcule $\alpha(C)$ pour trouver l'axiome associé à C. Si $\alpha(C) \neq$ `indéfini`, c'est-à-dire qu'on a trouvé une signification logique $\alpha(C)$ au cluster C, on interrompt la recherche, et on ajoute $\alpha(C)$ à la file d'axiomes à visiter $\mathcal{AT}$. Sinon, C n'a pas de signification logique accessible, et on poursuit la recherche dans ses sous-clusters. Toutefois, au fur et à mesure qu'on s'enfonce dans l'arbre de clustering, la taille des clusters diminue, et donc la valeur statistique des axiomes extraits diminue également : on fixe donc une profondeur maximale D au-delà de laquelle la recherche s'arrête. On note finalement $L_\alpha(a)$ l'ensemble des clusters étiquetés rencontrés lors de la recherche.

On construit alors la sous-taxonomie extraite T_a à partir de X_a : T_a contient tous les axiomes extraits, c'est-à-dire les étiquettes de X_a, et sa structure reflète la structure entre les clusters, la subsumption entre axiomes correspondant à l'inclusion des clusters associés :

$$\begin{aligned} T_a = (&\{\alpha(C) : C \in L_\alpha(a)\}, \\ &\{(\alpha(C), \alpha(C')) : C, C' \in L_\alpha(a) \text{ et } C \subset C' \text{ et} \\ &\forall D \text{ cluster de } X_a, (C \subset D \subset C' \implies \alpha(D) = \texttt{indéfini})\}) \end{aligned} \tag{5.2}$$

Il s'agit du même principe que l'extraction de taxonomie de la méthode de liaison injective présentée à l'équation 4.39 de la section 4.2.2 : le parent d'un cluster étiqueté C est le premier cluster étiqueté C' qui contient C.

Enfin, si au moins l'une des feuilles de X_a n'est pas couverte par un axiome $\alpha(C)$, c'est-à-dire s'il existe au moins une branche allant des feuilles à la racine qui ne contient pas d'étiquette, c'est qu'une partie de l'arbre n'a pas été décrite par un axiome, et qu'il reste potentiellement

de nouveaux axiomes à extraire. Dans ce cas, un nœud spécial `<?>` est ajouté à T_a et relié directement à a. La signification logique de ce nœud s'écrit :

$$\texttt{<?>} = a \wedge \left(\bigwedge_{C \in L_\alpha(a)} \neg\alpha(C) \right) \tag{5.3}$$

Soit, en langage courant, l'ensemble des éléments qui vérifient a mais ne vérifient aucun des sous-axiomes $\alpha(C)$ de a. Cette définition purement négative n'est pas d'une grande utilité dans une taxonomie expressive, puisqu'elle exprime simplement la tautologie $C \vee \neg C = \top$. On utilise donc le symbole `<?>` pour signifier que la recherche n'est pas encore finie pour a et qu'il reste des sous-axiomes de a à trouver.

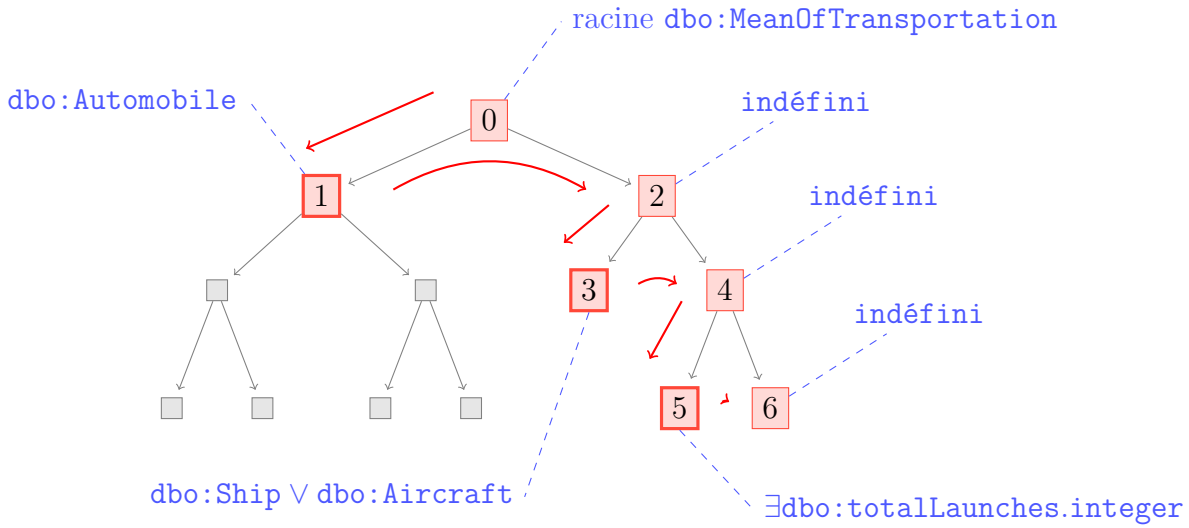
On donne un exemple de ce parcours à la figure 5.2. Dans cet exemple, l'axiome de départ a est la classe `dbo:MeanOfTransportation`, et la profondeur maximale de recherche est fixée à 4. L'arbre de clustering est parcouru en profondeur (figure 5.2a), et trois sous-axiomes de a sont extraits : `dbo:Automobile`, `dbo:Ship`$\vee$`dbo:Aircraft` et $\exists$`dbo:totalLaunches.integer`. On peut voir que toutes les entités ne sont pas couvertes par un axiome : en effet, le cluster 6 n'est pas étiqueté. Cela signifie potentiellement que les trois axiomes extraits ne couvrent pas l'intégralité des instances de `dbo:MeanOfTransportation` et qu'il reste donc de nouveaux concepts à découvrir : on ajoute donc le nœud spécial `<?>` (figure 5.2b).

Si $T_a = (\varnothing, \varnothing)$, c'est-à-dire qu'aucun axiome n'a été trouvé en parcourant l'arbre X_a, alors a n'aura pas de sous-axiome dans la taxonomie extraite et restera une feuille de $\hat{T}$. Sinon, on remplace le nœud a de $\hat{T}$ par la sous-taxonomie T_a. Cette fusion de $\hat{T}$ et T_a correspond à la fonction `Fusionne` de l'algorithme 2, dont le code est décrit dans l'algorithme 4. Une fois $\hat{T}$ et T_a fusionnées, on répète l'étape précédente, tant que la file d'axiomes $\mathcal{AT}$ n'est pas vide.

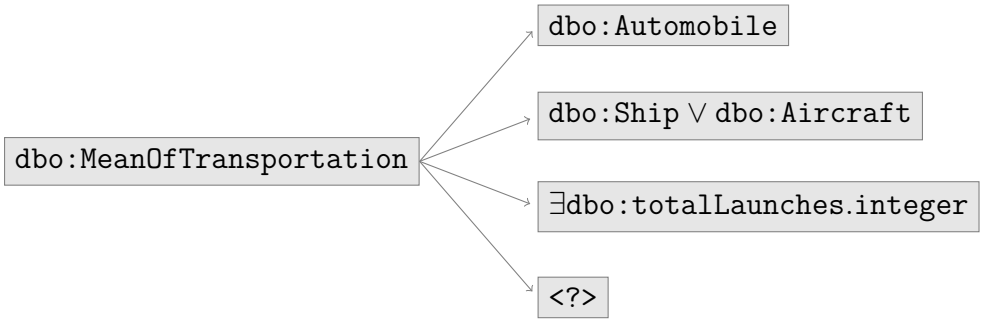
Extraction mono-niveau ou multi-niveaux Comme l'illustre la figure 5.3, notre algorithme de parcours d'arbre peut fonctionner avec deux modes d'extraction : mono-niveau ou multi-niveaux. Dans le premier cas, dès qu'on identifie un axiome pour le cluster C, on interrompt la recherche dans les sous-clusters. La taxonomie T_a extraite à partir de X_a ne contient donc qu'un seul niveau, elle a donc pour sommets les axiomes extraits $\alpha(C_1), \ldots, \alpha(C_k)$ pour tous les $C_1, \ldots, C_k$ étiquetés, reliés directement à la racine a. La taxonomie T_a s'écrit donc simplement :

$$T_a = (\{\alpha(C) : C \in L_\alpha\}, \{(\alpha(C), a) : C \in L_\alpha\}) \tag{5.4}$$

Ainsi, dans la figure 5.3a, on extrait $A \sqsubset \top$ et $B \sqsubset \top$ à l'étape une, puis $A_1 \sqsubset A$, $A_2 \sqsubset A$, $A_3 \sqsubset A$ à l'étape deux.

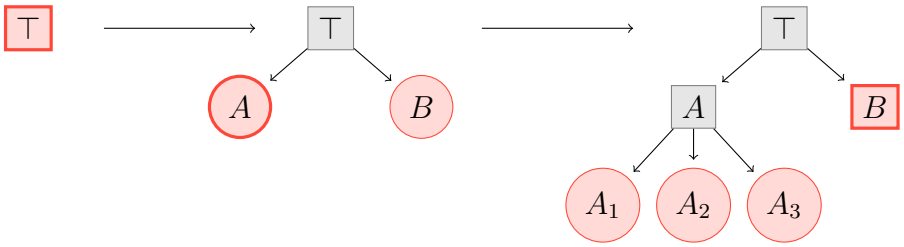


(a) Parcours du graphe X_a

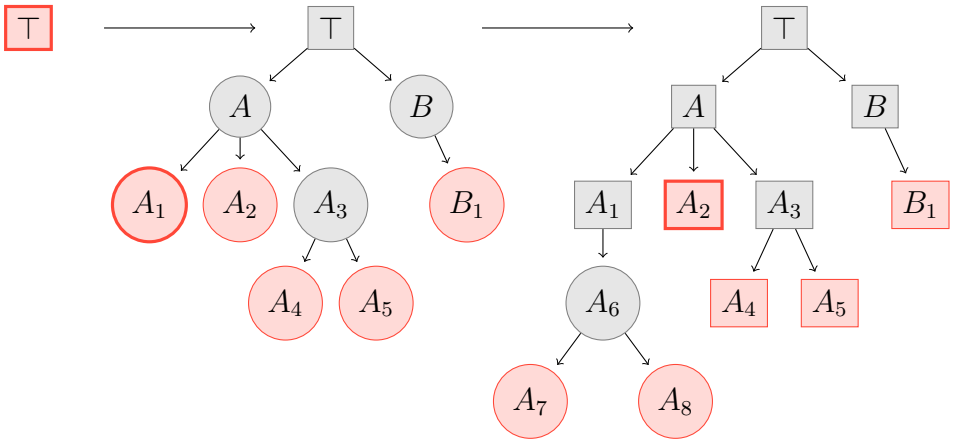


(b) Résultat obtenu T_a

Figure 5.2 Un exemple d'étiquetage d'un arbre de clustering X_a, avec a = `dbo:MeanOfTransportation`. *En haut*, chaque nœud représente un cluster de X_a ; on représente en rouge les clusters qui sont visités au cours de l'extraction (les flèches rouges indiquent le parcours effectué, et le chiffre l'ordre de visite des clusters), et en gris ceux qui ne sont pas visités. En bleu, on représente l'axiome associé à chaque cluster visité ; on indique par une bordure rouge épaisse les clusters qui sont effectivement étiquetés et qui apparaissent dans la taxonomie T_a. *En bas*, on représente la taxonomie partielle obtenue T_a.



(a) Extraction mono-niveau



(b) Extraction multi-niveaux

Figure 5.3 Représentation de $\hat{T}$ au cours des trois premières étapes de l'algorithme, pour une extraction mono-niveau (*en haut*) et une extraction multi-niveaux (*en bas*). Dans les deux cas, $\hat{T}$ est réduite au concept universel $\top$ lors de l'initialisation. Dans le premier cas, on ajoute un unique niveau à chaque étape ; dans le second, on s'autorise à en ajouter plusieurs. On représente en rouge les axiomes non-encore explorés, qui appartiennent donc à la file d'axiomes non-traités $\mathcal{AT}$; la bordure épaisse représente le prochain axiome à explorer ; les cercles indiquent les axiomes qui viennent d'être extraits.

À l'inverse, dans le cas multi-niveaux, on poursuit la recherche dans les sous-clusters de C, même quand C est étiqueté, et ce, jusqu'à atteindre la profondeur maximale D. La taxonomie T_a a donc potentiellement plusieurs niveaux. Cette approche est illustrée à la figure 5.3b : trois niveaux sont extraits lors de la première étape, et deux autres lors de la seconde. À chaque étape, seules les feuilles de T_a sont ajoutées à la file d'axiomes $\mathcal{AT}$.

L'approche multi-niveaux permet de réduire le nombre d'étapes de plongement-regroupement nécessaires – et donc potentiellement de diminuer la durée d'extraction totale, tout en tirant au maximum parti de la structure d'arbre de X_a. À l'inverse, dans le cas mono-niveau, la structure d'arbre sert principalement à déterminer dynamiquement le nombre de clusters pertinents. Dans nos essais, on utilisera uniquement la méthode mono-niveau (quitte à exécuter un plus grand nombre d'étapes), car elle est moins sensible aux changements de paramètres

et paraît globalement plus robuste. L'algorithme 3 contient le pseudo-code de cette approche.

Algorithme 3 : Pseudo-code pour la fonction `EtiquetteArbre` de l'algorithme 2. Cette fonction parcourt l'arbre de clustering X_a et cherche à étiqueter les clusters avec des axiomes, et renvoie une taxonomie partielle sur ces axiomes. La fonction `TrouveAxiomes` est décrite dans la section 5.2.3.

Entrée : un arbre de clustering X_a
Sortie : une taxonomie partielle T_a
Paramètres : profondeur maximale du parcours D
Fonction `EtiquetteArbre`(X_a) **:**

```
racine ← racine de X_a ;
NonVisités ← { sous-clusters de racine } ;
L_α ← ∅ ;
ajouteNoeudSpecial ← Faux ;
tant que NonVisités n'est pas vide faire
    C ← premier élément de NonVisités ;
    α(C) ← TrouveAxiomes(C, δ) ;
    si α(C) n'est pas indéfini alors
        ajouter C à L_α ;
    sinon
        si la profondeur de C est inférieure à D alors
            ajouter les sous-clusters de C à NonVisités ;
        sinon
            ajouteNoeudSpecial ← Vrai ;
V_a ← {α(C)}_{C∈L_α} ;
E_a ← {(α(C), racine) : C ∈ L_α} ;
si ajouteNoeudSpecial est vrai alors
    ajouter <?> à V_a;
    ajouter (<?>, racine) à E_a ;
T_a ← (V_a, E_a) ;
retourner T_a
```

Le cas des nœuds spéciaux `<?>` Dans l'étape précédente, le cas où l'axiome de départ a est l'axiome spécial `<?>` est traité un peu différemment du cas général. Les données d'entrée sont toujours tirées aléatoirement, suivant la formule 5.3 ; le regroupement hiérarchique et l'extraction de la sous-taxonomie T_a suivent une procédure identique. Ensuite, on rattache T_a à $\hat{T}$. La taxonomie $\hat{T}$ en cours d'extraction contient alors toujours le nœud spécial `<?>`, qu'on ne souhaite pas garder : on modifie alors $\hat{T}$ en rattachant directement les successeurs directs de `<?>` avec le prédecesseur direct de `<?>`, on ré-écrit donc les chemins $\alpha \to$ `<?>` $\to \beta$ en $\alpha \to \beta$, et on supprime `<?>` de $\hat{T}$.

Algorithme 4 : Pseudo-code pour la fonction `Fusionne` de l'algorithme 2. Cette fonction ajoute la taxonomie T_2 à la taxonomie T_1, en traitant éventuellement le cas où a (l'axiome qui a servi à construire T_2) est un nœud spécial `<?>`.

Fonction `Fusionne`(T_1, T_2) **:**
  $a \leftarrow$ axiome racine de T_2 ;
  $(V_1, E_1) \leftarrow T_1$;
  $(V_2, E_2) \leftarrow T_2$;
  $V \leftarrow V_1 \cup V_2$;
  $E \leftarrow E_1 \cup E_2$;
  si *a est de la forme* `<?>` **alors**
    pour $\alpha, \beta \in V$ *tels que* $\alpha \rightarrow a \rightarrow \beta$ **faire**
      `// on relie directement beta à alpha`
      $E \leftarrow (E \setminus \{(\beta, a), (a, \alpha)\}) \cup \{(\beta, \alpha)\}$;
    $V \leftarrow V \setminus \{a\}$;
  $T \leftarrow (V, E)$;
  retourner T

5.2.3 Extraction d'axiomes

L'étape de regroupement hiérarchique décrite précédemment suppose l'existence d'une méthode pour étiqueter automatiquement un cluster avec un axiome logique. Cette méthode doit estimer si un cluster correspond effectivement à un groupe d'entités pertinent, et, le cas échéant, lui attribuer un axiome logique capable de décrire les entités contenues dans ce cluster. Si le cluster n'est pas jugé pertinent, la méthode doit renvoyer `indéfini`.

La recherche d'un axiome A pour un cluster C ne peut pas se baser uniquement sur les entités du cluster C : en effet, on cherche un axiome qui qualifie C et *uniquement* C. On doit donc s'assurer que les entités qui ne font pas partie de C ne vérifient pas A, ce qui implique de disposer d'un ensemble d'exemples négatifs, en plus des exemples positifs que sont les entités de C. Un axiome doit ainsi remplir deux conditions : **couverture** (l'axiome est vérifié par la majorité des éléments du cluster) et **spécificité** (l'axiome n'est pas ou peu vérifié hors du cluster).

Toutefois, si l'on choisit comme exemples négatifs l'ensemble des entités qui ne sont pas dans C, on doit à nouveau considérer l'intégralité du graphe ; on perd donc l'intérêt d'avoir réduit l'espace de recherche grâce au tirage aléatoire et au regroupement, et on retrouve la difficulté inhérente à l'extraction d'axiomes dans un graphe de grande taille. On pourrait choisir de tirer aléatoirement des exemples négatifs, mais on risque alors de diminuer la spécificité des axiomes extraits. Par exemple, supposons que l'on cherche à étiqueter un sous-cluster de `dbo:Writer` : on souhaitera alors extraire des axiomes précis, capables par exemple de

distinguer les poètes des romanciers ou des dramaturges. Pourtant, si on compare les entités d'un tel sous-cluster à des entités quelconques du graphe, il est probable qu'un simple axiome $\exists \texttt{dbo:isWriterOf}.\top$ suffise à obtenir une très bonne spécificité. Les exemples négatifs doivent donc être de plus en plus proches des exemples positifs à mesure que l'on s'enfonce dans la taxonomie.

Or, l'arbre de clustering fournit justement, pour chaque cluster C, un groupe d'entités proche mais disjoint de C : il s'agit du cluster *voisin*. L'arbre étant binaire, chaque cluster (hors clusters feuilles) est partitionné en deux sous-clusters ; ces deux sous-clusters sont dits *voisins* l'un de l'autre. Deux clusters voisins ont le même cluster parent, ce qui indique une proximité entre eux ; ils sont disjoints l'un de l'autre, ce qui peut indiquer une différence entre leurs entités. Notre approche consiste donc à expliquer, au moyen d'un axiome logique, la division d'un cluster parent en deux sous-clusters. La méthode d'extraction choisie est simple, et se base sur des statistiques d'occurrence au sein d'un cluster, dans la lignée de la méthode SSI présentée dans la section 2.2.2. Toutefois, d'autres algorithmes d'extraction d'axiomes pourraient être utilisés.

Dans les paragraphes qui suivent, on formalise les notions de couverture et de spécificité esquissées ici, et on présente le cadre de notre méthode d'extraction et son fonctionnement.

Couverture, spécificité et score de partition

Soit $C = \{e_1, e_2, \ldots, e_n\} \subseteq \mathcal{E}$ un cluster contenant n entités. Si C n'est pas une feuille, alors il a deux sous-clusters gauche et droit, notés L et R, et contenant respectivement n_1 et n_2 entités, avec $n = n_1 + n_2$. En notant $\sqcup$ l'union disjointe, on a donc :

$$C = L \sqcup R \tag{5.5}$$

Pour un axiome logique A et une entité $x \in \mathcal{E}$, on note $A(x)$ si x vérifie l'axiome A. On se propose d'expliquer la séparation du cluster C en ses deux sous-clusters, c'est-à-dire d'identifier des axiomes qui sont vrais dans L mais pas dans R. Si on trouve un tel axiome, on pourra alors l'utiliser comme étiquette pour le cluster L. Si on n'en trouve pas, L portera l'étiquette `indéfini`. Pour étiqueter R, il suffit d'inverser les rôles de L et R.

On peut voir ce problème comme une recherche inductive d'axiomes à partir d'un ensemble d'exemples positifs $E^+ = L$ et d'un ensemble d'exemples négatifs $E^- = R$.

Expliquer la division de C en $L \sqcup R$ nécessite de trouver un axiome A tel que A soit valide

pour tous les éléments de L et pour aucun élément de R :

$$(\forall x \in L, A(x)) \wedge (\forall x \in R, \neg A(x)) \tag{5.6}$$

Soit, de manière équivalente :

$$\forall x \in C = L \sqcup R, A(x) \oplus (x \in R) \tag{5.7}$$

Avec $\oplus$ l'opérateur «ou exclusif». L'équation 5.7 signifie qu'une entité de C ne peut pas à la fois être dans R et vérifier A, et elle ne peut pas non plus être dans L sans vérifier A. Cette équation correspond au cas optimal où il existe un axiome qui divise parfaitement C en L et R : en pratique, la plupart des clusters ne peuvent pas être parfaitement divisés, et il nous faut donc mesurer à quel point on s'écarte de l'optimalité. Pour cela, on commence par définir la *précision* d'un axiome A par rapport à un ensemble d'éléments $E \sqsubset \mathcal{E}$ quelconque comme la proportion d'éléments de E qui vérifient A :

$$\text{prec}(A, E) = \frac{|\{x \in E, A(x)\}|}{|E|} \tag{5.8}$$

On mesure alors la capacité d'un axiome A à expliquer la division $C = L \sqcup R$ avec deux métriques : d'une part, sa *couverture*, définie comme la proportion d'éléments de L qui vérifient A ; d'autre part, sa *spécificité*, qui indique la proportion d'éléments de R qui ne vérifient pas A. Les deux doivent être proches de 1. Ces deux métriques se calculent à partir de la précision comme suit :

$$\text{cov}_{L \sqcup R}(A) = \text{prec}(A, L) \tag{5.9}$$

$$\text{spe}_{L \sqcup R}(A) = 1 - \text{prec}(A, R) \tag{5.10}$$

On combine ces deux mesures en un seul indicateur synthétique, que l'on appelle le *score de partition* de A, à l'aide d'une moyenne pondérée :

$$\text{part}_{L \sqcup R}(A) = \frac{|L| \cdot \text{cov}_{L \sqcup R}(A) + |R| \cdot \text{spe}_{L \sqcup R}(A)}{|L| + |R|} \tag{5.11}$$

Dans la suite, on omettra l'indice $L \sqcup R$ lorsqu'il n'y a pas d'ambiguïté.

On peut vérifier que l'on retrouve bien l'intuition derrière l'équation 5.7. Notons n, n_1, n_2 les effectifs des clusters C, L et R respectivement. La proportion d'éléments qui vérifient la

condition de séparation 5.7 est donnée par :

$$\text{xor}(A) = \frac{|\{x \in C : A(x) \oplus (x \in R)\}|}{n} \tag{5.12}$$

Par définition de l'opérateur ou exclusif, on peut écrire :

$$n \cdot \text{xor}(A) = |\{x \in C : (A(x) \wedge \neg(x \in R) \vee (x \in R \wedge \neg A(x))\}| \tag{5.13}$$

Puis, comme C est l'union disjointe de L et R, si $x \in C$, alors $\neg(x \in R)$ est équivalent à $x \in L$ et on a donc :

$$\begin{aligned} n \cdot \text{xor}(A) &= |\{x \in L : (A(x)\} \sqcup \{x \in R : \neg A(x))\}| \\ &= |\{x \in L : (A(x)\}| + |\{x \in R : \neg A(x))\}| \\ &= |\{x \in L : (A(x)\}| + |\{x \in R\} \setminus \{x \in R : A(x))\}| \\ &= n_1 \cdot \text{prec}(A, L) + n_2 - n_2 \cdot \text{prec}(A, R) \\ &= n_1 \cdot \text{cov}(A) + n_2 \cdot \text{spe}(A) \end{aligned}$$

Soit finalement :

$$\text{xor}(A) = \frac{n_1 \cdot \text{cov}(A) + n_2 \cdot \text{spe}(A)}{n_1 + n_2} \tag{5.14}$$

Propriétés de cov **et** spe On vérifie facilement que les métriques cov et spe vérifient les inégalités suivantes, pour n'importe quels axiomes A et B :

$$\text{cov}(A \wedge B) \leq \text{cov}(A) \tag{5.15}$$

$$\text{cov}(A \vee B) \geq \text{cov}(A) \tag{5.16}$$

$$\text{spe}(A \wedge B) \geq \text{spe}(A) \tag{5.17}$$

$$\text{spe}(A \vee B) \leq \text{spe}(A) \tag{5.18}$$

Ici et dans la suite, on note $\wedge$ et $\vee$ les opérateurs de conjonction et de disjonction de la logique descriptive habituellement notés $\sqcap$ et $\sqcup$, afin d'éviter une confusion avec l'union disjointe.

On tire de ces inégalités les conclusions suivantes : on peut généraliser un axiome A (c'est-à-dire augmenter sa couverture) en ajoutant une clause de disjonction, d'après l'équation 5.16 ; on peut le spécialiser (c'est-à-dire augmenter sa spécificité) en ajoutant une clause de conjonction, d'après l'équation 5.17. On voit aussi qu'on ne peut pas améliorer simultanément la couverture et la spécificité à l'aide de ces clauses.

Construction d'axiomes complexes par améliorations successives d'axiomes atomiques

On dispose d'un moyen pour évaluer la capacité d'un axiome A à expliquer la partition d'un cluster en deux sous-clusters. Il reste à définir une procédure pour construire de tels axiomes. Pour cela, on définit d'abord des types d'axiomes primitifs, appelés des *axiomes atomiques* ou simplement *atomes* dans la suite ; on extrait, pour chaque cluster, une liste d'axiomes atomiques, puis on combine ces atomes au moyen de conjonctions et de disjonctions pour produire des axiomes plus complexes.

On considère les types d'axiomes atomiques suivants :

- les classes C, qui correspondent aux concepts nommés de la logique descriptive, tels que `dbo:Agent`, `dbo:Person` ou `dbo:Place` ;
- les restrictions de la forme $\exists R.C$ avec C une classe, par exemple $\exists$`dbo:locatedIn`.`dbo:Country`, qui contient toutes les entités situées dans un pays ;
- les restrictions de la forme $\exists R.\{v\}$ avec $v \in \mathcal{E}$ une entité, tel que $\exists$`dbo:locatedIn`.$\{$`dbr:Canada`$\}$ pour représenter l'ensemble des entités situées au Canada ;
- les restrictions de la forme $\exists R.t$, avec t représentant les littéraux d'un type t donné, comme $\exists$`dbo:birthDate`.`xsd:date` l'ensemble des entités dont la date de naissance est du type `xsd:date` ;

Lorsqu'on combine ces axiomes comme dans le paragraphe suivant, on obtient la logique descriptive $\mathcal{EL}$, à laquelle s'ajoute l'union $\vee$.

On note à nouveau C le cluster d'entrée, L et R ses deux sous-clusters, et on cherche à étiqueter le sous-cluster L. Comme précédemment, pour étiqueter R, il suffit d'inverser les rôles de L et R.

Extraction des atomes Pour chaque entité x du cluster d'entrée, on extrait l'ensemble des triplets (x, r, y) dont x est le sujet. Si r est la relation `rdf:type`, alors y représente une classe, et le triplet (x, r, y) est transformé en l'axiome atomique y. Si y est un littéral, on extrait son type t, et on obtient l'axiome atomique $\exists r.t$. Autrement, on liste les classes $C_1, C_2, \ldots, C_m$ dont y fait partie, et on extrait les axiomes atomiques $\exists r.\{y\}, \exists r.C_1, \ldots, \exists r.C_m$. Des exemples sont donnés dans le tableau 5.1.

On obtient ainsi une liste d'axiomes atomiques $\mathcal{A}_{\text{atomes}}$ pour l'entièreté du cluster ; cette liste est filtrée et seuls les atomes vérifiés par plus de $\delta_{\text{filtre}} = 10\%$ des entités du cluster sont conservés. Notons que cette étape est commune à la recherche d'axiomes de L et de R, elle peut donc être effectuée une seule fois.

Tableau 5.1 Exemples d'axiomes atomiques extraits pour trois triplets. Les préfixes `dbo:` sont omis par souci de concision.

s

Triplet	Atomes extraits
$(x, \texttt{rdf:type}, \texttt{Person})$	$\texttt{Person}$
$(x, \texttt{bornIn}, \texttt{Canada})$	$\exists\texttt{bornIn}.\texttt{Country}$, $\exists\texttt{bornIn}.\texttt{Place}$, $\exists\texttt{bornIn}.\top$, $\exists\texttt{bornIn}.\{\texttt{Canada}\}$
$(x, \texttt{birthDate}, \texttt{"1928-1-8"xsd:date})$	$\exists\texttt{birthDate}.\texttt{xsd:date}$

Amélioration des axiomes Ensuite, on génère une liste d'axiomes candidats $\mathcal{A}_{\text{candidats}}$ à partir de cette liste d'axiomes atomiques. Initialement, les axiomes candidats sont simplement les axiomes atomiques. On évalue chaque axiome a parmi les candidats, en calculant sa couverture, sa spécificité et son score. On compare alors ce score à un seuil δ, par exemple $\delta = 0.9$. Si $\text{cov}(a) < \delta$, l'axiome a ne couvre pas assez d'entités : on le généralise alors en ajoutant des disjonctions avec des axiomes atomiques (clauses OU). Pour cela, on génère un nouvel axiome candidat $a \vee b$ pour chaque axiome atomique b contenu dans $\mathcal{A}_{\text{atomes}}$, et on l'ajoute à la liste des axiomes candidats. D'après l'équation 5.16, on a bien $\text{cov}(a \vee b) \geq \text{cov}(a)$. À l'inverse, si $\text{spe}(a) < \delta$, alors l'axiome est insuffisamment spécifique : il est vérifié par trop d'éléments de R. On le spécialise en ajoutant des conjonctions (clauses ET). Pour tout axiome atomique b contenu dans $\mathcal{A}_{\text{atomes}}$, on génère donc l'axiome $a \wedge b$ et on l'ajoute à la liste des axiomes candidats. Si l'on a à la fois $\text{spe}(a) < \delta$ et $\text{cov}(a) < \delta$, alors l'axiome a ne permet pas d'expliquer la partition $C = L \sqcup R$ et il est retiré de la liste des candidats. Enfin, si $\text{cov}(a) > \delta$ et $\text{spe}(a) > \delta$, l'axiome est conservé tel quel et il n'est plus amélioré.

À chaque itération, on limite le nombre de candidats qui sont améliorés (c'est-à-dire étendus par des disjonctions ou des conjonctions) : seuls les N_{ax} axiomes candidats avec le plus haut score de partition sont conservés, avec N_{ax} un paramètre fixé empiriquement, typiquement $N_{\text{ax}} = 10$. De plus, si l'amélioration du score entre le candidat amélioré $a \vee b$ (ou $a \wedge b$) et le candidat initial a est inférieure à un gain minimal `gain_min`, alors le candidat amélioré est rejeté. Ce faisant, on évite d'allonger exagérément les axiomes pour des gains de score minimes. La recherche d'axiomes s'arrête lorsqu'il n'y a plus d'axiomes candidats à améliorer ou lorsque le nombre d'itérations dépasse une certaine limite `max_étapes`.

Le résultat de cet algorithme est un ensemble $\mathcal{A}(L)$ (éventuellement vide) d'axiomes capables de qualifier le sous-cluster L par opposition au sous-cluster R, avec les scores associés. Si $\mathcal{A}(L) = \varnothing$, aucun axiome satisfaisant aux critères n'a été trouvé, et le cluster L n'est donc

pas étiqueté : on renvoie l'étiquette `indéfini`. Sinon, on étiquette L avec l'axiome de plus haut score :

$$\alpha(L) = \arg\max_{a \in \mathcal{A}(L)}(\text{part}(a)) \tag{5.19}$$

L'algorithme 5 contient le pseudo-code correspondant à cette méthode d'extraction d'axiomes. On y désigne par `Evalue` la fonction qui, à un axiome A et deux ensembles E^+ et E^- contenant des exemples positifs et négatifs, associe le triplet ($\text{cov}_{E^+ \sqcup E^-}(A)$, $\text{spe}_{E^+ \sqcup E^-}(A)$, $\text{part}_{E^+ \sqcup E^-}(A)$).

Algorithme 5 : Pseudo-code pour extraire un axiome à partir d'une liste d'axiomes atomiques, et d'exemples positifs et négatifs.

Entrée : un ensemble d'exemples positifs E^+ et négatifs E^- (typiquement, un cluster cible et son voisin), et une liste d'atomes
Sortie : liste d'axiomes $\mathcal{A}$ vérifiés par les entités de E^+ mais pas par celles de E^-
Paramètres : nombre maximal d'étapes max_étapes, gain minimal gain_min, seuil de score δ, nombre d'axiomes gardés d'une étape à l'autre N_{ax}

```
Fonction ExtraitAxiome(E+, E−, atomes) :
    A ← ∅ ;
    // initialisation des candidats
    candidats ← ∅ ;
    pour atome dans atomes faire
        couv, spécif, score ← Evalue(atome, E+, E−) ;
        si score > δ alors ajouter atome à A;
        // spécificité insuffisante : on spécialise avec ET
        sinon si spécif < δ alors ajouter (atome, score, ET) à candidats ;
        // couverture insuffisante : on généralise avec OU
        sinon si couv < δ alors ajouter (atome, score, OU) à candidats ;
    // amélioration des candidats
    t ← 0 ;
    tant que t < max_étapes faire
        améliorés ← ∅ ;
        pour (axiome, scoreInit, OP) dans candidats faire
            pour atome dans atomes faire
                nouveauCandidat ← atome OP axiome ;
                couv, spécif, score ← Evalue (nouveauCandidat, E+, E−) ;
                si score > δ alors
                    // nouveauCandidat a un score suffisant : on le garde
                    ajouter nouveauCandidat à A ;
                    continuer
                sinon si couv < δ et spécif < δ ou score − scoreInit < gain_min alors
                    continuer
                // nouveauCandidat est meilleur que axiome, mais pas encore
                   suffisant : on poursuit l'amélioration
                sinon si spécif < δ alors ajouter (nouveauCandidat, score, ∧) à améliorés ;
                sinon si couv < δ alors ajouter (nouveauCandidat, score, ∨) à améliorés ;
        si améliorés est vide alors
            // aucun nouvel axiome n'a été trouvé : la recherche s'arrête
            interrompre
        candidats ← les N_ax meilleurs axiomes de améliorés ;
        t ← t + 1 ;
    retourner le meilleur axiome de A (ou indéfini si A est vide) ;
```

Seuil adaptatif

Dans l'algorithme présenté aux sections précédentes, le seuil d'extraction δ est fixé tout au long de la construction de la taxonomie. On expérimente également une variante de cet algorithme, dans laquelle le seuil δ varie au cours du temps : au départ, le seuil de validité des axiomes est fixé à une valeur initiale élevée $\delta = \delta_{\text{init}}$; puis, à chaque fois que la file d'axiomes $\mathcal{AT}$ est vidée, on diminue δ d'une quantité $d\delta$, on ré-initialise $\mathcal{AT} = \{\top\}$, et on recommence l'algorithme, jusqu'à atteindre un seuil minimal $\delta_{\min}$.

Partir avec un seuil bas dès le début ne permet pas de discriminer efficacement les axiomes valides des axiomes invalides ; à l'inverse, conserver un seuil élevé tout au long de l'algorithme conduit à écarter des axiomes valides : en effet, les graphes de connaissances étant incomplets et bruités – certaines relations manquent, des triplets peuvent être erronés – un axiome valide peut être imparfaitement vérifié. La technique du seuil adaptatif permet de contourner en partie cette limitation, en imposant d'abord un seuil haut qui permet de créer une ossature fiable pour la taxonomie, puis en relâchant ce seuil pour aggréger de nouveaux axiomes plus incertains.

Dans nos expérimentations, les valeurs de $\delta_{\text{init}} = 0.9, d\delta = 0.1, \delta_{\min} = 0.5$ semblent donner les résultats les plus équilibrés.

5.3 Évaluation et discussion

L'évaluation d'une taxonomie expressive est plus difficile que celle d'une taxonomie non-expressive, car il n'existe pas de référence à laquelle la comparer. On propose ici deux modes d'évaluation. En premier lieu, on restreint les types d'axiomes recherchés aux seules classes nommées, ce qui nous ramène à l'extraction de taxonomie non-expressive et nous permet de mener une évaluation quantitative, sur le modèle du chapitre précédent. En second lieu, on analyse qualitativement les axiomes extraits.

Dans toute la suite, on travaille sur le graphe DBpedia décrit au chapitre précédent, auquel on a retiré les triplets `rdfs:subClassOf`. On utilise les plongements vectoriels TransE, car ce sont eux qui obtiennent les meilleurs résultats sur la tâche non-expressive présentée précédemment.

5.3.1 Évaluation quantitative par comparaison avec l'ontologie existante

L'évaluation quantitative sur la tâche non-expressive permet de vérifier que notre approche identifie correctement les hiérarchies entre les classes nommées ; identifier ces hiérarchies est un préalable nécessaire à l'extraction de nouvelles classes et à l'extraction d'axiomes plus

complexes.

On utilise donc la méthode décrite plus haut, mais on restreint les axiomes atomiques aux seules classes nommées C, et on fixe `max_étapes` à 0, ce qui revient à interdire les clauses ET et OU. On effectue 300 étapes de prélèvement-regroupement, on utilise un seuil adaptatif δ allant de $0,9$ à $0,5$ par pas de $0,05$, et on considère une profondeur maximale $D = 4$. Le résultat est accessible en ligne, à l'adresse labowest.ca/sdb2020/non_expressive.html.

On compare la taxonomie obtenue $\hat{T}$ à la taxonomie de référence de DBpedia T^*. Lorsqu'on élimine les entités impliquées dans un seul triplet, et que l'on ne garde que les classes ayant au moins une entité, on obtient une taxonomie de référence contenant 455 classes. Cette comparaison se fait selon les modalités décrites à la section 4.3.3 : on calcule la précision, le rappel et le F_1 entre l'ensemble des axiomes de subsumption contenus dans $\hat{T}$ et l'ensemble des axiomes de subsumption contenus dans T^* – c'est l'évaluation *directe*. On recalcule ensuite ces trois métriques pour les clôtures transitives de $\hat{T}$ et T^* – c'est l'évaluation *transitive*. On moyenne alors chacune des trois métriques sur les deux modes d'évaluation, ce qui donne l'évaluation *moyenne*. Les résultats sont présentés dans le tableau 5.2.

Tableau 5.2 Évaluation de la taxonomie non-expressive extraite à partir de DBpedia, en comparaison de l'ontologie de DBpedia.

	Précision	Rappel	F_1
Directe	68,94	68,79	68,87
Transitive	98,28	85,99	91,72
Moyenne	83.61	77,39	80,30

On obtient un F_1 de 68,9% sur l'évaluation directe, et de 91,7% sur l'évaluation transitive. La précision transitive est très élevée (98,2%), ce qui indique que l'algorithme commet peu d'erreurs de haut niveau. En revanche, l'écart entre cette précision transitive et la précision directe indique que l'algorithme ne parvient pas à extraire toute la hiérarchie et a tendance à «aplatir» la taxonomie ; autrement dit, il n'extrait pas suffisamment de niveaux de hiérarchie. Cela est corroboré par l'examen manuel de la taxonomie produite. Celle-ci contient seulement deux axiomes aberrants (`dbo:TimePeriod` $\sqsubset$ `dbo:Economist`, et `dbo:PersonFunction` $\sqsubset$ `dbo:Astronaut`). En revanche, un type d'erreur fréquent est la prédiction de $A \sqsubset C$ et $B \sqsubset C$ à la place de $A \sqsubset B \sqsubset C$, soit effectivement un aplatissement de la hiérarchie. De plus, un certain nombre de classes, et particulièrement de classes rares, sont rattachées directement à la racine.

5.3.2 Analyse qualitative des axiomes obtenus

On discute maintenant des axiomes obtenus lors de l'extraction de taxonomie expressive. Un exemple de taxonomie expressive est présenté à l'adresse labowest.ca/sdb2020/expressive.html (en anglais). Cette taxonomie a été obtenue en utilisant un score adaptatif variant de $\delta = 0,9$ à $\delta = 0,5$ avec un pas de $0,05$, une extraction mono-niveau avec profondeur maximale $D = 4$ et 350 étapes de prélèvement-regroupement. Une fois ces 350 étapes effectuées, on effectue 100 étapes en se restreignant à la recherche de classes nommées, avec un seuil bas $\delta = 0,15$ et une profondeur maximale $D = 10$, afin d'ajouter les classes rares ou mal définies à la taxonomie extraite.

Lorsqu'on trouve plusieurs bons candidats $\alpha_1(C), \ldots, \alpha_k(C)$ pour étiqueter un cluster C, on choisit toujours l'axiome avec le score le plus élevé $\alpha^*(C) = \arg\max_{\alpha_i(C)} \mathrm{part}(\alpha_i(C))$, mais on garde également en mémoire les autres candidats, à condition qu'ils aient un score au moins égal à 90% du meilleur score. Si ces candidats alternatifs existent, le cluster est affiché en bleu sur la page Web indiquée, et les candidats alternatifs sont affichés au survol de la souris. Ces candidats peuvent être utilisés à des fins d'analyse, pour mieux comprendre les informations que l'algorithme extrait ; ils pourraient aussi servir pour enrichir l'ontologie (par exemple, en ajoutant les axiomes $\alpha^*(C) \equiv \alpha_i(C)$ ou $\alpha_i(C) \sqsubseteq \alpha^*(C)$) ou pour identifier les relations associées à un concept spécifique.

Dans l'exemple présenté, notre approche extrait 106 classes expressives qui s'ajoutent aux 455 classes nommées de l'ontologie DBpedia. Ce nombre varie beaucoup selon les paramètres choisis (en particulier, le seuil δ, le gain minimal `gain_min`, et la longueur maximale des axiomes `max_étapes`), et on peut obtenir des taxonomies avec très peu de classes expressives, ou au contraire avec davantage de classes expressives que de classes nommées.

Certaines de ces classes expressives subsument un ensemble de classes existantes. Par exemple, il se forme une classe `dbo:Film` $\vee$ `dbo:TelevisionShow`, également étiquetée par l'axiome expressif $\exists$`dbo:runtime.xsd:integer`$\vee\exists$`dbo:starring.dbo:Person`, qui regroupe les films et les séries télévisées. Parmi les regroupements qui paraissent cohérents mais qui n'existent pas dans l'ontologie DBpedia, on retrouve également les fusées et les navettes spatiales, associées notamment à la relation `totalLaunches` ; les romans et les bandes dessinées ; les épisodes de série et ceux de dessins animés ; les programmes télévisuels et radiophoniques ; les journaux quotidiens et les magazines ; les montagnes et les volcans, etc. Les regroupements de plus de deux classes sont plus rares : on trouve notamment la réunion des mammifères, reptiles, oiseaux, poissons dans une grande classe d'animaux excluant les insectes, ou différents types de plantes réunis avec les champignons. Certains regroupements sont plus étonnants, par exemple la réunion des œuvres écrites (`dbo:WrittenWork`) et des logiciels (`dbo:Software`), ou

celle des bateaux avec les avions ; d'autres enfin sont incompréhensibles et s'apparentent à des erreurs pures et simples, comme le regroupement au sein d'une même classe de `dbo:Document`, `dbo:Sound` et `dbo:TimePeriod`. De telles erreurs semblent toutefois rares.

À l'inverse, certaines classes expressives précisent des classes existantes, et permettent donc d'augmenter la spécificité de la taxonomie. Un exemple est fourni par la classe `dbo:Language` : dans DBpedia, cette classe possède une unique sous-classe `dbo:ProgrammingLanguage`. Dans notre taxonomie expressive, une nouvelle sous-classe a été identifiée : ∃`dbo:spokenIn`.`dbo:Place`, qui contient les langues naturelles. Cette nouvelle sous-classe émerge à partir des données, et permet de décrire un groupe cohérent d'entités qui n'avait auparavant pas sa propre classe.

Toutefois, la plupart de ces classes spécifiques sont difficiles à interpréter : ainsi, la classe ∃`dbo:latestReleaseDate`.`xsd:date`∧ ∃`dbo:operatingSystem`.`dbo:Software` (sous-classe de `dbo:Software`) semble être construite par opposition aux jeux vidéos, mais les axiomes qui la décrivent sont peu explicites.

On trouve également des classes d'une très grande spécificité, comme par exemple `dbo:Album` ∧ (∃`dbo:language`.{`dbo:Tamil_Language` ∨ ∃`dbo:language`.{`dbo:Telugu_Language`}), qui contient des albums musicaux en langue tamoule ou télougou, deux langues parlées dans le sud de l'Inde. On voit ici apparaître un risque, celui d'une spécificité variable de la taxonomie extraite : dans cet exemple, il n'est pas nécessairement justifié d'avoir une classe pour qualifier tel genre musical et pas tel autre. Ce risque est d'autant plus marqué que Wikipédia (et, par suite, DBpedia) est biaisée [1] ; une méthode d'extraction inductive risque de refléter ces biais, voire de les renforcer en les fixant dans une taxonomie.

Enfin, au-delà de la détection de nouvelles classes, la taxonomie extraite permet d'enrichir les classes existantes, notamment grâce aux candidats alternatifs mentionnés plus haut. L'examen de ces candidats alternatifs permet d'associer classes nommées et relations : en effet, ces candidats alternatifs indiquent qu'un même groupe d'entité pouvait être décrit à la fois par des classes nommées et des axiomes expressifs, ce qui indique une relation d'inclusion ou d'équivalence entre les deux. Ainsi, `dbo:Person` est lié à `dbo:birthDate` et `foaf:gender`, `dbo:SportsTeamMember` à ∃`dbo:squadNumber`.`xsd:nonNegativeInteger` et ∃`dbo:team`.`dbo:SportsTeam`, les instances de `dbo:WrittenWork` font partie de ∃`dbo:author`. `dbo:Person` ou de `dbo:PeriodicalLitterature`, etc. Identifier ces régularités peut permettre de guider la complétion du graphe, en repérant les relations manquantes dans le graphe, de détecter le type d'une entité à partir des relations qu'elle vérifie ou encore d'amé-

1. En 2015, [122] notait que la version anglaise de Wikipédia contenait des articles sur 18 000 acteurs états-uniens, 2 600 acteurs indiens et moins de 100 acteurs nigérians, alors que l'Inde et le Nigéria sont les deux plus gros producteurs de films au monde.

liorer l'extraction automatique de triplets RDF en sachant quelles sont les informations à extraire.

5.3.3 Discussion et limitations

En l'état, la taxonomie expressive qui est extraite par notre méthode n'est pas encore au niveau d'une taxonomie produite par des experts humains : elle contient des doublons, des axiomes peu informatifs ou faux, et n'identifie pas toutes les relations de subsumption. On ne peut pas encore s'affranchir totalement de la supervision humaine, qui reste nécessaire pour nettoyer et interpréter les axiomes extraits.

En revanche, notre méthode peut constituer un outil pour explorer un graphe de connaissances et aider à la création ou la mise à jour d'une taxonomie, en identifiant les hiérarchies entre classes, en généralisant ou en spécialisant des classes existantes, et en extrayant des descriptions logiques de ces classes. De manière générale, la taxonomie extraite donne un aperçu général du contenu d'un graphe, du point de vue de ses classes et de ses relations ; on peut ensuite passer aisément au niveau des instances, en prélevant des entités qui vérifient ou non certains axiomes.

Notons d'ailleurs que, par souci de simplicité et faute de temps, nous n'avons pas intégré à notre outil d'extraction d'axiomes certains aspects importants. En premier lieu, nous ne considérons pas les relations inverses, à savoir les relations R^{-1} telles que $x \xrightarrow{R^{-1}} y$ si $y \xrightarrow{R} x$. Or, la plupart des relations de DBpedia n'existent que dans un seul sens : par exemple, une œuvre est liée à son auteur par la relation `dbo:author`, mais il n'existe pas de lien inverse entre un auteur et ses œuvres. On ne pourra donc pas extraire un axiome du type $\exists \texttt{dbo:author}^{-1}$. `dbo:Poem`, qui aurait été susceptible de décrire les poètes. De même, une gestion minimale des *valeurs* des littéraux (et pas seulement de leurs types) pourrait permettre de gérer les dates, donc les périodes historiques, les chiffres de population, les données économiques ou autres, et donc affiner l'identification ou la description des classes.

CHAPITRE 6 CONCLUSION

6.1 Synthèse des travaux

Dans ce travail, nous nous sommes intéressés au problème de l'extraction de taxonomie à partir d'un graphe de connaissances. Plus particulièrement, notre idée était d'utiliser les plongements vectoriels issus d'un graphe de connaissances pour identifier des groupes d'entités cohérents, et établir des liens taxonomiques entre ces groupes. L'objectif est donc d'établir une hiérarchie entre les classes existantes, mais aussi de pouvoir découvrir et caractériser de nouvelles classes.

Pour cela, nous commençons par une étude des modèles de plongement vectoriel, sous l'angle de la *séparabilité* : nous mesurons la capacité des modèles de plongement à plonger des entités de types différents dans des régions différentes de l'espace. Cette tâche permet de nuancer l'état de l'art en matière de plongements vectoriels : en effet, sur les tâches d'évaluation usuelles que sont la prédiction de liens ou la complétion de triplets, TransE obtient des performances inférieures à des modèles plus expressifs comme ComplEx ; ici, la situation est inversée.

Nous abordons ensuite le problème de l'extraction de taxonomie à partir d'un graphe de connaissances, en se basant sur DBpedia. Puisque l'on veut pouvoir identifier de nouvelles classes, on se base sur un regroupement non-supervisé des plongements vectoriels : cela nous permet d'extraire une structure d'arbre sur les groupes d'entités sans aucune supervision. À partir de cette structure, on peut décliner l'approche pour obtenir une taxonomie expressive ou non-expressive.

Si l'on fait correspondre ces groupes à des types connus, on obtient une taxonomie sur ces types. Au chapitre 4, nous proposons deux méthodes pour établir cette correspondance, l'une qui détermine une injection optimale des types vers les groupes, l'autre qui propose une association multiple entre les uns et les autres. Nous montrons que ces deux méthodes sont capables d'égaler et même de dépasser une méthode basée sur un regroupement supervisé.

Au chapitre 5, on associe au contraire les groupes à des axiomes logiques expressifs, et on ajoute un mécanisme d'itération pour fiabiliser le résultat : d'une part, le regroupement hiérarchique est répété sur des groupes d'entités de plus en plus spécifiques, et d'autre part la taxonomie extraite est étendue petit à petit sur la base de ce regroupement. Nous obtenons ainsi une taxonomie expressive de DBpedia.

6.2 Limites et pistes d'améliorations

Nous identifions ici quelques limites des travaux présentés ici, et proposons des pistes pour les surmonter. Nous esquissons aussi des directions de recherche pour poursuivre ou prolonger le travail entamé dans ce mémoire.

Modèles de plongement Dans tous les travaux présentés ici, on s'est restreint à évaluer six modèles de plongement, entraînés avec les hyperparamètres par défaut. Augmenter le nombre de modèles entraînés et faire varier ces hyperparamètres constitue donc une première piste pour affiner et systématiser les résultats obtenus, à la fois pour la séparabilité et l'extraction de taxonomie.

Sur la question des hyperparamètres, [123] a montré que la recherche d'hyperparamètres était une étape cruciale pour la tâche de prédiction de lien ; si cette conclusion s'applique également à l'extraction de taxonomie, alors une recherche rigoureuse d'hyperparamètres s'impose : non seulement pour obtenir les meilleurs résultats possibles, mais aussi pour comprendre l'interaction entre les choix de modélisation effectués par un modèle de plongement donné, les valeurs des différents hyperparamètres et les données d'entrée.

Quant aux modèles de plongement, de nombreuses approches ont été proposées qui obtiennent de bons résultats sur les jeux d'évaluation usuels. Dans la lignée des modèles présentés ici, on peut citer des modèles algébriques comme HolE [124] et SimplE [108], des modèles géométriques, comme PTransE [125] qui étend TransE à des chemins dans le graphe, plutôt qu'à des triplets isolés, ou RotatE [126], qui choisit de représenter les relations par des rotations dans l'espace. D'autres approches incluent des modèles neuronaux, comme R-GCN [127], ProjE [128] ou ConvE [129], ou l'utilisation de géométries non-euclidiennes [60, 79].

Pour l'évaluation des modèles, qui fait l'objet du chapitre 3, notre tâche de séparabilité se révèle incomplète : idéalement, on souhaiterait trouver un mode d'évaluation qui reflète les performances des modèles sur l'extraction taxonomique. Or, RDF2Vec obtient de meilleurs résultats que TransE pour la séparabilité, alors que TransE est supérieur à RDF2Vec sur la tâche d'extraction. Pour remédier à ce problème, on pourrait adjoindre une tâche de *regroupement* à la tâche de séparabilité, pour vérifier le comportement des modèles de plongement vis-à-vis des algorithmes de regroupement. On peut par exemple tirer des entités appartenant à k classes différentes, puis appliquer un algorithme des k-moyennes sur les plongements de ces entités, et finalement mesurer si le partitionnement obtenu recoupe bien les k classes initiales.

Extraction de taxonomie sur les classes existantes Une première extension du travail effectué ici consisterait à essayer d'autres graphes de connaissances : on pourrait expérimenter avec DBpedia dans d'autres langues que l'anglais, avec d'autres graphes généralistes comme YAGO [47] ou Wikidata [48], ou avec des graphes spécialisés.

Notre approche pour l'extraction automatique de taxonomie sur les classes existantes est limitée par l'étape de regroupement. Pour des questions de performance, la taille des données à regrouper est limitée, ce qui, d'une part, empêche d'utiliser l'intégralité des entités du graphe, et d'autre part, entraîne la décroissance exponentielle du nombre d'entités par classe. Ces deux facteurs compliquent le traitement des classes rares ou très spécifiques. En l'état, notre méthode n'est donc pas applicable à tout le graphe DBpedia.

Une solution possible consisterait à itérer la méthode MLM avec des données d'entrée tirées aléatoirement dans le graphe : à chaque étape, on tire des entités aléatoirement, on les regroupe, et on calcule la probabilité des axiomes de subsumption. On peut alors maintenir une matrice contenant ces probabilités, et la mettre à jour à chaque étape. De plus, on peut utiliser la géométrie des plongements pour tirer les entités dans une région donnée de l'espace, ce qui permettrait d'augmenter la spécificité des données et donc de la taxonomie extraite – il s'agirait en fait d'adapter la méthode expressive présentée à la section 5.2.1 au cas où l'on n'ait pas accès au graphe complet.

Extraction de taxonomie expressive Notre méthode d'extraction d'axiomes présente plusieurs limites : d'une part, les axiomes atomiques sont limités et laissent de côté les relations inverses, la composition de relations, les restrictions universelles, etc. D'autre part, on combine à chaque étape un axiome (complexe ou non) avec un atome : on ne peut donc pas combiner un axiome complexe avec un autre axiome complexe, ce qui interdit par exemple des axiomes de la forme $(A \wedge B) \vee (C \wedge D)$. Il serait également intéressant d'expérimenter d'autres approches pour l'extraction d'axiomes : comme l'espace de recherche est un sous-ensemble d'entités de petite taille et sémantiquement cohérent, il devrait être possible d'employer des méthodes plus complexes et capables d'extraire des axiomes plus expressifs, comme [130] ou [131].

Quant aux faiblesses des taxonomies expressives extraites par notre approche, elles ont été discutées à la section 5.3.3 : les axiomes extraits sont inégalement informatifs, parfois redondants ou trop spécifiques. La piste d'amélioration la plus évidente consiste à expérimenter des valeurs différentes pour les paramètres. Ceux-ci sont en effet nombreux, à chaque étape : on peut modifier les modalités du tirage aléatoire, la méthode de regroupement, l'ordre de parcours de l'arbre de clustering et sa profondeur maximale, le mode d'extraction (mono-niveau

ou multi-niveaux), le nombre d'étapes et l'évolution du seuil δ. Toutefois, pour systématiser une telle recherche, il est nécessaire de concevoir une méthode d'évaluation qualitative fiable des taxonomies produites. Pour ce faire, on peut envisager la démarche suivante : identifier des tâches externes susceptibles de tirer profit d'axiomes logiques (par exemple, typage automatique d'entités, réponse automatique à des questions, ou autre), et comparer, sur ces tâches, les performances d'un modèle utilisant seulement le graphe aux performances d'un modèle utilisant à la fois le graphe et la taxonomie extraite. On obtient ainsi une évaluation indirecte de cette taxonomie, qui permet de mesurer son utilité pratique.

Enfin, puisque l'on dispose de plongements vectoriels pour les entités, et d'axiomes logiques pour qualifier ces groupes, une direction de recherche intéressante serait de chercher une représentation vectorielle des axiomes logiques, afin d'obtenir une représentation unifiée pour les entités, les relations, les classes et les axiomes. Cela suppose de traduire géométriquement des opérateurs logiques tels que la négation, la conjonction ou la disjonction ; ce problème est discuté par [106], et une solution est esquissée par [132].

RÉFÉRENCES

[1] I. Bounhas, N. Soudani et Y. Slimani, "Building a morpho-semantic knowledge graph for arabic information retrieval," *Information Processing & Management*, p. 102–124, 2019.

[2] L. Dietz, A. Kotov et E. Meij, "Utilizing knowledge graphs for text-centric information retrieval," dans *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, 2018, p. 1387–1390.

[3] R. Ying *et al.*, "Graph convolutional neural networks for web-scale recommender systems," dans *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, p. 974–983.

[4] H. Wang *et al.*, "Ripplenet : Propagating user preferences on the knowledge graph for recommender systems," dans *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, 2018, p. 417–426.

[5] X. Wang *et al.*, "Explainable reasoning over knowledge graphs for recommendation," dans *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, p. 5329–5336.

[6] Y. Zhang *et al.*, "Variational reasoning for question answering with knowledge graph," dans *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[7] D. Lukovnikov *et al.*, "Neural network-based question answering over knowledge graphs on word and character level," dans *Proceedings of the 26th international conference on World Wide Web*, 2017, p. 1211–1220.

[8] A. Saha *et al.*, "Complex sequential question answering : Towards learning to converse over linked question answer pairs with a knowledge graph," dans *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[9] G. Bakal *et al.*, "Exploiting semantic patterns over biomedical knowledge graphs for predicting treatment and causative relations," *Journal of biomedical informatics*, vol. 82, p. 189–199, 2018.

[10] R. T. Sousa, S. Silva et C. Pesquita, "Evolving knowledge graph similarity for supervised learning in complex biomedical domains," *BMC bioinformatics*, vol. 21, n°. 1, p. 6, 2020.

[11] E. Hyvönen, H. Rantala *et al.*, "Knowledge-based relation discovery in cultural heritage knowledge graphs," dans *Digital Humanities in Nordic Countries Proceedings of the Digital Humanities in the Nordic Countries 4th Conference*. CEUR-WS. org, 2019.

[12] W. Wilcke, V. de Boer et F. van Harmelen, “User-driven pattern mining on knowledge graphs : An archaeological case study,” dans *Benelearn 2017 : Proceedings of the Twenty-Sixth Benelux Conference on Machine Learning, Technische Universiteit Eindhoven, 9-10 June 2017*, 2017, p. 137.

[13] L. Heling *et al.*, “Building knowledge graphs from survey data : A use case in the social sciences (extended version),” dans *European Semantic Web Conference*. Springer, 2019, p. 285–299.

[14] S. Auer *et al.*, “Dbpedia : A nucleus for a web of open data,” dans *The semantic web*. Springer, 2007, p. 722–735.

[15] G. Töpper, M. Knuth et H. Sack, “Dbpedia ontology enrichment for inconsistency detection,” dans *Proceedings of the 8th International Conference on Semantic Systems*, ser. I-SEMANTICS ’12. New York, NY, USA : Association for Computing Machinery, 2012, p. 33–40. [En ligne]. Disponible : https://doi.org/10.1145/2362499.2362505

[16] M. Kejriwal et P. Szekely, “Supervised typing of big graphs using semantic embeddings,” dans *Proceedings of The International Workshop on Semantic Big Data*, ser. SBD ’17. New York, NY, USA : Association for Computing Machinery, 2017. [En ligne]. Disponible : https://doi.org/10.1145/3066911.3066918

[17] M. Herrero-Zazo *et al.*, “Dinto : using owl ontologies and swrl rules to infer drug–drug interactions and their mechanisms,” *Journal of chemical information and modeling*, vol. 55, n°. 8, p. 1698–1707, 2015.

[18] L. Otero-Cerdeira, F. J. Rodríguez-Martínez et A. Gómez-Rodríguez, “Ontology matching : A literature review,” *Expert Systems with Applications*, vol. 42, n°. 2, p. 949–971, 2015.

[19] C. Bhatia et S. Jain, “Semantic web mining : Using ontology learning and grammatical rule inference technique,” dans *2011 International Conference on Process Automation, Control and Computing*. IEEE, 2011, p. 1–6.

[20] Z. Li *et al.*, “An ontology-based web mining method for unemployment rate prediction,” *Decision Support Systems*, vol. 66, p. 114–122, 2014.

[21] A. Holzinger, “From machine learning to explainable ai,” dans *2018 World Symposium on Digital Intelligence for Systems and Machines (DISA)*, 2018, p. 55–66.

[22] F. A. Cardillo et U. Straccia, “Towards Ontology-based Explainable Classification of Rare Events,” avr. 2019, working paper or preprint. [En ligne]. Disponible : https://hal.archives-ouvertes.fr/hal-02104520

[23] A. Seeliger, M. Pfaff et H. Krcmar, “Semantic web technologies for explainable machine learning models : A literature review.” dans *PROFILES/SEMEX@ ISWC*, 2019, p. 30–45.

[24] W. T. Adrian, A. Ligęza et G. J. Nalepa, “Inconsistency handling in collaborative knowledge management,” dans *2013 Federated Conference on Computer Science and Information Systems*, 2013, p. 1233–1238.

[25] L. Zhou, “Ontology learning : state of the art and open issues,” *Information Technology and Management*, vol. 8, nº. 3, p. 241–252, 2007.

[26] P. Cimiano *et al.*, “Conceptual knowledge processing with formal concept analysis and ontologies,” dans *International Conference on Formal Concept Analysis*. Springer, 2004, p. 189–207.

[27] J. Zhang *et al.*, “A new rapid incremental algorithm for constructing concept lattices,” *Information*, vol. 10, nº. 2, p. 78, 2019.

[28] J. Völker et M. Niepert, “Statistical schema induction,” dans *Extended Semantic Web Conference*. Springer, 2011, p. 124–138.

[29] W. Zhang *et al.*, “Iteratively learning embeddings and rules for knowledge graph reasoning,” dans *The World Wide Web Conference*, ser. WWW ’19. New York, NY, USA : Association for Computing Machinery, 2019, p. 2366–2377. [En ligne]. Disponible : https://doi.org/10.1145/3308558.3313612

[30] P. Ristoski *et al.*, “Large-scale taxonomy induction using entity and word embeddings,” dans *Proceedings of the International Conference on Web Intelligence*. ACM, 2017, p. 81–87.

[31] F. Wu et D. S. Weld, “Automatically refining the wikipedia infobox ontology,” dans *Proc. 17th International Conference on World Wide Web*. ACM, 2008, p. 635–644.

[32] V. Shwartz, E. Santus et D. Schlechtweg, “Hypernyms under siege : Linguistically-motivated artillery for hypernymy detection,” dans *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics : Volume 1, Long Papers*. Valencia, Spain : Association for Computational Linguistics, avr. 2017, p. 65–75. [En ligne]. Disponible : https://www.aclweb.org/anthology/E17-1007

[33] R. Fu *et al.*, “Learning semantic hierarchies via word embeddings,” dans *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1 : Long Papers)*, 2014, p. 1199–1209.

[34] N. Gupta *et al.*, “Domain ontology induction using word embeddings,” dans *15th IEEE International Conference on Machine Learning and Applications*. IEEE, 2016, p. 115–119.

[35] M. Atzori et S. Balloccu, “Fully-unsupervised embeddings-based hypernym discovery,” *Information*, vol. 11, nº. 5, p. 268, 2020.

[36] J. Pocostales, “NUIG-UNLP at SemEval-2016 task 13 : A simple word embedding-based approach for taxonomy extraction,” dans *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*. San Diego, California : Association for Computational Linguistics, juin 2016, p. 1298–1302. [En ligne]. Disponible : https://www.aclweb.org/anthology/S16-1202

[37] M. A. Hearst, “Automatic acquisition of hyponyms from large text corpora,” dans *Proceedings of the 14th conference on Computational Linguistics*, vol. 2. ACL, 1992, p. 539–545.

[38] S. Roller, D. Kiela et M. Nickel, “Hearst patterns revisited : Automatic hypernym detection from large text corpora,” dans *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2 : Short Papers)*. Melbourne, Australia : Association for Computational Linguistics, juill. 2018, p. 358–363. [En ligne]. Disponible : https://www.aclweb.org/anthology/P18-2057

[39] A. Bordes *et al.*, “Translating embeddings for modeling multi-relational data,” dans *Advances in Neural Information Processing Systems*, 2013, p. 2787–2795.

[40] T. Mikolov *et al.*, “Distributed representations of words and phrases and their compositionality,” dans *Advances in Neural Information Processing Systems*, 2013, p. 3111–3119.

[41] X. Lv *et al.*, “Differentiating concepts and instances for knowledge graph embedding,” *arXiv preprint arXiv :1811.04588*, 2018.

[42] C. Zhang *et al.*, “Taxogen : Unsupervised topic taxonomy construction by adaptive term embedding and clustering,” dans *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, p. 2701–2709.

[43] L. Ehrlinger et W. Wöß, “Towards a definition of knowledge graphs,” *SEMANTiCS*, vol. 48, p. 1–4, 2016.

[44] R. Cyganiak, M. Lanthaler et D. Wood, “RDF 1.1 concepts and abstract syntax,” W3C, W3C Recommendation, févr. 2014, http ://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[45] “Uniprot : the universal protein knowledgebase,” *Nucleic acids research*, vol. 45, nº. D1, p. D158–D169, 2017.

[46] D. S. Wishart *et al.*, “Drugbank 5.0 : a major update to the drugbank database for 2018,” *Nucleic acids research*, vol. 46, nº. D1, p. D1074–D1082, 2018.

[47] F. M. Suchanek, G. Kasneci et G. Weikum, "Yago : A large ontology from wikipedia and wordnet," *Journal of Web Semantics*, vol. 6, n°. 3, p. 203–217, 2008.

[48] D. Vrandečić et M. Krötzsch, "Wikidata : a free collaborative knowledgebase," *Communications of the ACM*, vol. 57, n°. 10, p. 78–85, 2014.

[49] Z. Wang *et al.*, "Knowledge graph embedding by translating on hyperplanes," dans *Twenty-Eighth AAAI conference on artificial intelligence*, 2014.

[50] I. Horrocks, O. Kutz et U. Sattler, "The even more irresistible sroiq." *Kr*, vol. 6, p. 57–67, 2006.

[51] M. Krötzsch, "Owl 2 profiles : An introduction to lightweight ontology languages," dans *Reasoning Web International Summer School*. Springer, 2012, p. 112–183.

[52] M. Krötzsch, F. Simancik et I. Horrocks, "Description logics," *IEEE Intelligent Systems*, vol. 29, n°. 1, p. 12–19, 2013.

[53] P. Hitzler *et al.*, "OWL 2 web ontology language primer (second edition)," W3C, Rapport technique, déc. 2012, https ://www.w3.org/TR/2012/REC-owl2-primer-20121211/.

[54] J. Lehmann, "DL-learner : learning concepts in description logics," *Journal of Machine Learning Research*, vol. 10, p. 2639–2642, 2009.

[55] H.-C. Kuo, T.-H. Tsai et H. Jen-Peng, "Building a concept hierarchy by hierarchical clustering with join/merge decision," dans *9th Joint International Conference on Information Sciences*. Atlantis Press, 2006.

[56] G. Petrucci, M. Rospocher et C. Ghidini, "Expressive ontology learning as neural machine translation," *Journal of Web Semantics*, vol. 52, p. 66–82, 2018.

[57] S. Sritha et B. Mathumathi, "A survey on various approaches for taxonomy construction," *Indian Journal of Innovations and Developments*, vol. 5, p. 6, 2016.

[58] N. Li, Z. Bouraoui et S. Schockaert, "Ontology completion using graph convolutional networks," dans *International Semantic Web Conference*, 2019, p. 435–452.

[59] S. Faralli *et al.*, "The ContrastMedium algorithm : Taxonomy induction from noisy knowledge graphs with just a few links," dans *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, vol. 1, 2017, p. 590–600.

[60] M. Nickel et D. Kiela, "Learning continuous hierarchies in the lorentz model of hyperbolic geometry," dans *Proc. ICML*, 2018.

[61] F. Nikolaev et A. Kotov, “Joint word and entity embeddings for entity retrieval from a knowledge graph,” dans *European Conference on Information Retrieval*. Springer, 2020, p. 141–155.

[62] C. Napoles, M. Gormley et B. Van Durme, “Annotated gigaword,” dans *Proceedings of the Joint Workshop on Automatic Knowledge Base Construction and Web-Scale Knowledge Extraction*, ser. AKBC-WEKEX ’12. USA : Association for Computational Linguistics, 2012, p. 95–100.

[63] J. Smith *et al.*, “Dirt cheap web-scale parallel text from the common crawl,” dans *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1 : Long Papers)*, 2013, p. 1374–1383.

[64] G. Boleda, A. Gupta et S. Padó, “Instances and concepts in distributional space,” dans *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics : Volume 2, Short Papers ; 2017 Apr 3-7 ; Valencia, Spain. Stroudsburg (PA) : ACL ; 2017. p. 79-85.* ACL (Association for Computational Linguistics), 2017.

[65] J. Camacho-Collados *et al.*, “Semeval-2018 task 9 : Hypernym discovery,” dans *Proceedings of the 12th International Workshop on Semantic Evaluation (SemEval-2018) ; 2018 Jun 5-6 ; New Orleans, LA. Stroudsburg (PA) : ACL ; 2018. p. 712–24.* ACL (Association for Computational Linguistics), 2018.

[66] A. Zouaq et R. Nkambou, “Evaluating the generation of domain ontologies in the knowledge puzzle project,” *IEEE Transactions on knowledge and data engineering*, vol. 21, nº. 11, p. 1559–1572, 2009.

[67] E. Hovy, Z. Kozareva et E. Riloff, “Toward completeness in concept extraction and classification,” dans *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, 2009, p. 948–957.

[68] A. Maedche et S. Staab, “Discovering conceptual relations from text,” 06 2000.

[69] E. Drymonas, K. Zervanou et E. Petrakis, “Unsupervised ontology acquisition from plain texts : The ontogain system,” vol. 6177, 01 1970, p. 277–287.

[70] R. Snow, D. Jurafsky et A. Y. Ng, “Learning syntactic patterns for automatic hypernym discovery,” dans *Advances in neural information processing systems*, 2005, p. 1297–1304.

[71] V. Shwartz, Y. Goldberg et I. Dagan, “Improving hypernymy detection with an integrated path-based and distributional method,” dans *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1 : Long*

Papers). Berlin, Germany : Association for Computational Linguistics, août 2016, p. 2389–2398. [En ligne]. Disponible : https://www.aclweb.org/anthology/P16-1226

[72] P. D. Turney, "Mining the web for synonyms : Pmi-ir versus lsa on toefl," dans *European conference on machine learning*. Springer, 2001, p. 491–502.

[73] A. Zouaq, D. Gasevic et M. Hatala, "Towards open ontology learning and filtering," *Information Systems*, vol. 36, n°. 7, p. 1064–1081, 2011.

[74] M. Geffet et I. Dagan, "The distributional inclusion hypotheses and lexical entailment," dans *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, 2005, p. 107–114.

[75] J. Shang *et al.*, "Automated phrase mining from massive text corpora," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, n°. 10, p. 1825–1837, 2018.

[76] J. Weeds, D. Weir et D. McCarthy, "Characterising measures of lexical distributional similarity," dans *COLING 2004 : Proceedings of the 20th International Conference on Computational Linguistics*. Geneva, Switzerland : COLING, aug 23–aug 27 2004, p. 1015–1021. [En ligne]. Disponible : https://www.aclweb.org/anthology/C04-1146

[77] M. Baroni et A. Lenci, "How we blessed distributional semantic evaluation," dans *Proceedings of the GEMS 2011 Workshop on GEometrical Models of Natural Language Semantics*, 2011, p. 1–10.

[78] K. A. Nguyen *et al.*, "Hierarchical embeddings for hypernymy detection and directionality," dans *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark : Association for Computational Linguistics, sept. 2017, p. 233–243. [En ligne]. Disponible : https://www.aclweb.org/anthology/D17-1022

[79] M. Nickel et D. Kiela, "Poincaré embeddings for learning hierarchical representations," dans *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., 2017, p. 6338–6347.

[80] Z. Yu *et al.*, "Learning term embeddings for hypernymy identification," dans *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[81] A. T. Luu *et al.*, "Learning term embeddings for taxonomic relation identification using dynamic weighting neural network," dans *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas : Association for Computational Linguistics, nov. 2016, p. 403–413. [En ligne]. Disponible : https://www.aclweb.org/anthology/D16-1039

[82] I. Vendrov *et al.*, "Order-embeddings of images and language," *arXiv preprint arXiv :1511.06361*, 2015.

[83] L. Vilnis *et al.*, "Probabilistic embedding of knowledge graphs with box lattice measures," *arXiv preprint arXiv :1805.06627*, 2018.

[84] R. Aly *et al.*, "Every child should have parents : A taxonomy refinement algorithm based on hyperbolic term embeddings," *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019. [En ligne]. Disponible : http://dx.doi.org/10.18653/V1/P19-1474

[85] B. Dhingra *et al.*, "Embedding text in hyperbolic spaces," *arXiv preprint arXiv :1806.04313*, 2018.

[86] O.-E. Ganea, G. Bécigneul et T. Hofmann, "Hyperbolic entailment cones for learning hierarchical embeddings," *arXiv preprint arXiv :1804.01882*, 2018.

[87] B. S. Everitt *et al.*, *Cluster analysis*, 5e éd. John Wiley & Sons, 2011, ch. 4, p. 80–84.

[88] H. Zhang *et al.*, "Learning multimodal taxonomy via variational deep graph embedding and clustering," dans *Proceedings of the 26th ACM International Conference on Multimedia*, ser. MM '18. New York, NY, USA : Association for Computing Machinery, 2018, p. 681–689. [En ligne]. Disponible : https://doi.org/10.1145/3240508.3240586

[89] D. Tsarkov et I. Horrocks, "Fact++ description logic reasoner : System description," dans *International joint conference on automated reasoning*. Springer, 2006, p. 292–297.

[90] B. Glimm *et al.*, "Hermit : an owl 2 reasoner," *Journal of Automated Reasoning*, vol. 53, no. 3, p. 245–269, 2014.

[91] A. Cropper, S. Dumančić et S. H. Muggleton, "Turning 30 : New ideas in inductive logic programming," dans *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, C. Bessiere, édit. International Joint Conferences on Artificial Intelligence Organization, 7 2020, p. 4833–4839, survey track.

[92] S.-H. Nienhuys-Cheng et R. De Wolf, *Foundations of inductive logic programming*. Springer Science & Business Media, 1997, vol. 1228.

[93] L. De Raedt et K. Kersting, "Probabilistic inductive logic programming," dans *Probabilistic Inductive Logic Programming*. Springer, 2008, p. 1–27.

[94] A. Srinivasan, T. A. Faruquie et S. Joshi, "Data and task parallelism in ilp using mapreduce," *Machine learning*, vol. 86, no. 1, p. 141–168, 2012.

[95] Q. Zeng, J. M. Patel et D. Page, "Quickfoil : Scalable inductive logic programming," *Proc. VLDB Endow.*, vol. 8, no. 3, p. 197–208, nov. 2014. [En ligne]. Disponible : https://doi.org/10.14778/2735508.2735510

[96] R. Wille, “Restructuring lattice theory : an approach based on hierarchies of concepts,” dans *Ordered Sets*, I. Rival, édit. Dordrecht-Boston : Reidel, 9 1981, p. 445–470.

[97] P. Valtchev et R. Missaoui, “Building concept (galois) lattices from parts : generalizing the incremental methods,” dans *International Conference on Conceptual Structures*. Springer, 2001, p. 290–303.

[98] L. Nourine et O. Raynaud, “A fast algorithm for building lattices,” *Information processing letters*, vol. 71, n°. 5-6, p. 199–204, 1999.

[99] M. Farach-Colton et Y. Huang, “A linear delay algorithm for building concept lattices,” dans *Annual Symposium on Combinatorial Pattern Matching*. Springer, 2008, p. 204–216.

[100] M. Li et G. Wang, “Approximate concept construction with three-way decisions and attribute reduction in incomplete contexts,” *Knowledge-Based Systems*, vol. 91, p. 165–178, 2016.

[101] R. Bendaoud, A. Napoli et Y. Toussaint, “Formal concept analysis : A unified framework for building and refining ontologies,” dans *Knowledge Engineering : Practice and Patterns*, A. Gangemi et J. Euzenat, édit. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, p. 156–171.

[102] C. Zhang et S. Zhang, *Association rule mining : models and algorithms*. Springer, 2003, vol. 2307.

[103] R. Srikant et R. Agrawal, “Fast algorithms for mining association rules,” dans *Proc. VLDB Conference*, 1994, p. 487–499.

[104] X. Yuan, “An improved apriori algorithm for mining association rules,” *AIP Conference Proceedings*, vol. 1820, n°. 1, p. 080005, 2017.

[105] X. Lin, “Mr-apriori : Association rules algorithm based on mapreduce,” dans *2014 IEEE 5th international conference on software engineering and service science*. IEEE, 2014, p. 141–144.

[106] V. Gutiérrez-Basulto et S. Schockaert, “From knowledge graph embedding to ontology embedding ? an analysis of the compatibility between vector space representations and rules,” dans *Sixteenth International Conference on Principles of Knowledge Representation and Reasoning*, 2018.

[107] P. G. Omran, K. Wang et Z. Wang, “Scalable rule learning via learning representation.” dans *IJCAI*, 2018, p. 2149–2155.

[108] S. M. Kazemi et D. Poole, “Simple embedding for link prediction in knowledge graphs,” dans *Advances in neural information processing systems*, 2018, p. 4284–4295.

[109] D. Q. Nguyen, T. Nguyen et D. Phung, "A relational memory-based embedding model for triple classification and search personalization," dans *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, juill. 2020, p. 3429–3435.

[110] R. Socher *et al.*, "Reasoning with neural tensor networks for knowledge base completion," dans *Advances in neural information processing systems*, 2013, p. 926–934.

[111] B. Hoser *et al.*, "Semantic network analysis of ontologies," dans *European Semantic Web Conference*. Springer, 2006, p. 514–529.

[112] M. Nickel, V. Tresp et H.-P. Kriegel, "A three-way model for collective learning on multi-relational data." dans *Icml*, vol. 11, 2011, p. 809–816.

[113] B. Yang *et al.*, "Embedding entities and relations for learning and inference in knowledge bases," dans *Proc. 3rd International Conference on Learning Representations*, 2015.

[114] T. Trouillon *et al.*, "Complex embeddings for simple link prediction," dans *Proc. International Conference on Machine Learning*, 2016.

[115] G. Ji *et al.*, "Knowledge graph embedding via dynamic mapping matrix," dans *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1 : Long Papers)*, 2015, p. 687–696.

[116] P. Ristoski et H. Paulheim, "Rdf2vec : RDF graph embeddings for data mining," dans *International Semantic Web Conference*. Springer, 2016, p. 498–514.

[117] X. Han *et al.*, "OpenKE : An open toolkit for knowledge embedding," dans *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing : System Demonstrations*. Association for Computational Linguistics, 2018, p. 139–144.

[118] T. Mikolov *et al.*, "Advances in pre-training distributed word representations," dans *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

[119] R. Jonker et A. Volgenant, "A shortest augmenting path algorithm for dense and sparse linear assignment problems," *Computing*, vol. 38, n°. 4, p. 325–340, 1987.

[120] P. Brown *et al.*, "A heuristic for the time constrained asymmetric linear sum assignment problem," *Journal of Combinatorial Optimization*, vol. 33, n°. 2, p. 551–566, 2017.

[121] J. Bergstra et Y. Bengio, "Random search for hyper-parameter optimization," *The Journal of Machine Learning Research*, vol. 13, n°. 1, p. 281–305, 2012.

[122] M. Nickel *et al.*, “A review of relational machine learning for knowledge graphs,” *Proceedings of the IEEE*, vol. 104, nº. 1, p. 11–33, 2015.

[123] R. Kadlec, O. Bajgar et J. Kleindienst, “Knowledge base completion : Baselines strike back,” *arXiv preprint arXiv :1705.10744*, 2017.

[124] M. Nickel *et al.*, “Holographic embeddings of knowledge graphs.” dans *AAAI*, vol. 2, nº. 1, 2016, p. 3–2.

[125] Y. Lin *et al.*, “Modeling relation paths for representation learning of knowledge bases,” *arXiv preprint arXiv :1506.00379*, 2015.

[126] Z. Sun *et al.*, “Rotate : Knowledge graph embedding by relational rotation in complex space,” dans *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [En ligne]. Disponible : https://openreview.net/forum?id=HkgEQnRqYQ

[127] M. Schlichtkrull *et al.*, “Modeling relational data with graph convolutional networks,” dans *European Semantic Web Conference*. Springer, 2018, p. 593–607.

[128] B. Shi et T. Weninger, “Proje : Embedding projection for knowledge graph completion,” dans *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[129] T. Dettmers *et al.*, “Convolutional 2d knowledge graph embeddings,” dans *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[130] D. Carral *et al.*, “Vlog : A rule engine for knowledge graphs,” dans *The Semantic Web – ISWC 2019*, C. Ghidini *et al.*, édit. Cham : Springer International Publishing, 2019, p. 19–35.

[131] J. Lajus, L. Galárraga et F. Suchanek, “Fast and exact rule mining with amie 3,” dans *The Semantic Web*, A. Harth *et al.*, édit. Cham : Springer International Publishing, 2020, p. 36–52.

[132] J. Hao *et al.*, “Universal representation learning of knowledge bases by jointly embedding instances and ontological concepts,” dans *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, p. 1709–1719.

ANNEXE A ARTICLE

On inclut à la page suivante l'article *What is the Schema of your Knowledge Graph: Leveraging Knowledge Graph Embeddings and Clustering for Expressive Taxonomy Learning*, co-écrit avec Amal Zouaq et présenté à la conférence SBD 2020. Son contenu correspond aux chapitres 3 et 5 du présent mémoire.

What is the Schema of your Knowledge Graph? Leveraging Knowledge Graph Embeddings and Clustering for Expressive Taxonomy Learning

Amal Zouaq
amal.zouaq@polymtl.ca
Polytechnique Montréal
Montréal, Québec, Canada

Félix Martel
felix.martel@polymtl.ca
Polytechnique Montréal
Montréal, Québec, Canada

ABSTRACT

Large-scale knowledge graphs have become prevalent on the Web and have demonstrated their usefulness for several tasks. One challenge associated to knowledge graphs is the necessity to keep a knowledge graph schema (which is generally manually defined) that accurately reflects the knowledge graph content. In this paper, we present an approach that extracts an expressive taxonomy based on knowledge graph embeddings, linked data statistics and clustering. Our results show that the learned taxonomy is not only able to retain original classes but also identifies new classes, thus giving an up-to-date view of the knowledge graph.

CCS CONCEPTS

• **Information systems** → **Clustering**; • **Computing methodologies** → **Ontology engineering**.

KEYWORDS

Knowledge graph embeddings, expressive taxonomies, ontology learning, hierarchical clustering.

ACM Reference Format:
Amal Zouaq and Félix Martel. 2020. What is the Schema of your Knowledge Graph? Leveraging Knowledge Graph Embeddings and Clustering for Expressive Taxonomy Learning . In *Semantic Big Data (SBD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3391274.3393637

1 INTRODUCTION

The emergence of knowledge graphs as a structured knowledge representation has seen tremendous developments in academia and industry in the last years [7]. Knowledge graphs are composed of facts, consisting of triples in the form (head entity, property, tail entity). Knowledge graphs often rely on a manually-built schema that represents types or classes, which gives a well-defined meaning to the facts they contain. The recent development of knowledge graphs has given rise to several interesting applications for question answering, information retrieval, recommendations, and more generally for problem solving. In parallel, we have also witnessed the incredible success of neural networks that require continuous representations. Knowledge graph embeddings arise, among other reasons, from the interest of exploiting knowledge graphs in neural models. These embeddings map entities and relations to low-dimensional vector representations and the general assumption is that the proximity of some given embeddings in a vector space implies their semantic proximity. However, once a group of embeddings appears as a cluster, one challenge is to be able to label this cluster. Another challenge that is related to knowledge graphs in general is that their ontological structure or schema is defined manually a priori. Thus, it is hard to identify what type of new data is injected in the knowledge graph, whether some unknown classes of entities emerge, and how to query and formalize the given schema of a dynamic knowledge graph at any point in time. In this paper, we seek to answer the following research question: How can we derive an ontological structure from knowledge graph embeddings?

In particular, our goal is to investigate several knowledge graph embedding models and evaluate their appropriateness for the identification of a knowledge graph current taxonomy. Based on these embeddings, we propose a method that clusters automatically knowledge graph entities and rely on the statistical distribution of classes and properties to label each cluster. Next, the hierarchical clustering combined with these frequent axioms are used to generate an expressive taxonomy.

The paper is structured as follows. Section 2 presents a background and state of the art for the generation of schemas for knowledge graphs. In section 3, we present the various knowledge graph models used in this study and we describe an evaluation task on the ability of embedding models to separate different classes. Finally, we explain our methodology to generate an expressive taxonomy using hierarchical clustering, statistics and cluster labelling in section 4.

2 RELATED WORK

2.1 Knowledge Graphs Schema Extraction

The extraction of a knowledge graph structure or schema from data is not a new task and has been tackled under the name of ontology learning or taxonomy extraction for the last decade. This task can be decomposed into learning simpler components such as classes, taxonomical relations, properties, complex axioms or rules from various data sources such as corpora or knowledge graph triples. Ontology learning in its more general form has been used to generate an ontology from scratch [14, 21], to populate an ontology [4, 13] or to complete an ontology [8]. Various techniques have

SBD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-7974-8/20/06…$15.00
https://doi.org/10.1145/3391274.3393637

been proposed such as linguistic patterns [3], statistical distributions over linked data [12], formal concept analysis [5], inductive logic programming [2] or machine learning [14]. While knowledge graph representation learning has emerged in several applications such as knowledge base completion [11] or link prediction [17], it has seldom been used, to our knowledge, for the task of schema generation or mining. Given the aim of these models to produce low-dimensional vectors that group semantically-similar entities in the same regions of the space, this property can be effectively exploited by a clustering algorithm to identify classes of entities in a dynamic knowledge graph. In this paper, we focus on the comparison of various knowledge graph embedding models in pointwise Euclidean space and complex space for the task of separability, that is the ability of linearly separating instances of different classes based on these embeddings. We then use the best models to generate a taxonomy from a knowledge graph.

2.2 Taxonomy Induction

The use of low-dimensional knowledge graph embeddings has only been recently considered for the task of schema induction and more precisely taxonomy extraction. The most similar work to the one proposed in this paper, TIEmb [15], proposes the extraction of a taxonomy using the RDF2Vec embeddings of DBpedia. TIEmb determines the inclusion of a cluster inside another one based on the centroid and radius of these clusters. In this paper, we exploit hierarchical clustering to automatically identify these inclusion axioms. We also do not only generate a taxonomy of named classes but label clusters with complex definitions based on conjunctions and disjunctions of atomic concepts and restriction axioms. This allows us to discover new classes that were not previously defined in the original knowledge graph schema and to identify triples that are incomplete (e.g. with undefined property values).

3 KNOWLEDGE GRAPH EMBEDDINGS

3.1 Embedding Models

In this paper, we experiment with six embedding models that represent entities in Point-wise Euclidean space and Complex space: TransE [1], TransH [19], TransD [6], DistMult [20], ComplEx [18] and RDF2Vec [16].

TransE, TransH and TransD. The geometric model behind TransE [1] is that a relation r can be seen as a translation in the Euclidean space : a triple (h, r, t) is valid if $\mathbf{t}$ is the translation of $\mathbf{h}$ by the vector $\mathbf{r}$. This results in the following energy function:

$$E(h, r, t) = \|\mathbf{h} + \mathbf{r} - \mathbf{t}\|_2 \quad (1)$$

As analyzed in [19], the geometric assumptions behind TransE are very strong and thus cannot handle properly all types of relations. For example, if r is a reflexive relation (meaning that (h, r, t) implies (t, r, h)), then the model will be trained to have both $h+r \approx t$ and $t + r \approx h$, i.e. $h \approx t$ and $r \approx 0$. To overcome these limitations, many extensions have been proposed, including TransH [19] and TransD [6]. In these two models, a relation is still modelled as a translation in the vector space, but this translation does not operate directly on the entity embeddings. Entity embeddings are first transformed, then translated. In TransH, embeddings are projected to a hyperplane H_r specific to relation r. The hyperplane H_r is defined by its normal vector $w_r \in \mathbb{R}^d$ and is learnt jointly with h, r, t. The projection $u_\perp$ of a vector $u \in \mathbb{R}^d$ on H_r is defined by:

$$u_\perp = u - (w_r^\top u) \cdot w_r \quad (2)$$

And the energy function of a triple is:

$$E(h, r, t) = \|\mathbf{h}_\perp + \mathbf{r} - \mathbf{t}_\perp\|_2 \quad (3)$$

Instead of a projection, TransD uses a linear transformation M, with M a transformation matrix depending on both the entity and the relation. For each entity or relation $u \in \mathcal{E} \cup \mathcal{R}$, the model learns two vectors of R^d: the embedding u itself, and a projection vector u_p. Then, the transformation matrix for an entity e and a relation r is dynamically built as: $M_{e,r} = r_p \cdot e_p{}^\top + I^{d \times d}$. The entity embedding can then be transformed using $e_r = M_{e,r} \cdot e$ and the energy function of a triple is :

$$E(h, r, t) = \|\mathbf{h_r} + \mathbf{r} - \mathbf{t_r}\|_2 \quad (4)$$

DistMult. In TransE, TransH and TransD, the embeddings of h, r, t are added or subtracted to/from each other. DistMult [20] proposes a multiplicative combination of these embeddings, with a score function:

$$\sigma(h, r, t) = h^\top D_r t \quad (5)$$

With D_r a diagonal matrix. The authors indicate that such multiplicative composition gives more expressivity to the model while keeping the same number of parameters as TransE.

ComplEX. ComplEX [18] extends DistMult to the complex space: h, r, t are in C^d instead of R^d, with a score function defined as:

$$\sigma(h, r, t) = R\left(h^\top W_r \bar{t}\right) \quad (6)$$

The use of real, purely imaginary or complex vectors can make the score function symmetric, antisymmetric or neither symmetric nor antisymmetric, so it can better reflect the symmetry of the relation.

RDF2Vec. Finally, RDF2Vec [16] applies the idea of Word2Vec [10] to knowledge graphs: random walks (that is, random sequences of connected entities and relations) are sampled in the graph to create a training set. Then, these walks are fed to a shallow neural network whose objective function is to accurately predict an entity or a relation given its context.

3.2 A Comparison of Knowledge Graph Embeddings using Separability

Since our goal is to use geometric similarity between embeddings to identify groups of entities that are semantically similar, we want to assess the ability of a model to embed entities from the same class to the same region of the Euclidean space. We formalize this by defining a class separability task, which aims at measuring if embeddings from different classes can be linearly separated.

The setup for this task is as follows: for two classes A and B, we sample N entities from A and N entities from B, which creates a dataset $D = \{\mathbf{e_i}, y_i\}_{i=1,\dots,2N}$ with $\mathbf{e_i} \in \mathbb{R}^d$ the embedding vector of entity e_i, and $y_i = 0$ if e_i is from class A, 1 otherwise. A linear SVM is trained over 75% of these samples, and then used to predict the labels of the 25% remaining samples. The predicted labels are then compared to the actual labels, using standard classification metrics: precision, recall, F-score. For each class, we sample 1,000 instances

if the class has enough instances, otherwise we use all the instances from that class.

Linear separability for entities of different classes is a strong requirement. An embedding model can perform well on standard evaluation benchmarks such as link prediction or triple completion without necessarily fulfilling this requirement. However, an embedding model that enforces as much as possible this linear separability constraint guarantees a straightforward relation between geometric similarity and semantic similarity, which makes the taxonomic extraction step much easier.

The challenge of separating class A from class B varies greatly with A and B: intuitively, it is harder to distinguish between a `College` and a `University` than between an `Aircraft` and a `Person`. To evaluate the complexity of the separability of the various classes of an ontology, we setup an experiment with three metrics: lexical distance measured using word embeddings, taxonomic distance based on an available taxonomy and class frequency in the knowledge graph. The first two metrics measure how difficult the task is based on class labels (lexical distance) and taxonomic distance. The last metric aims at measuring the effect of the size (in terms of instances) of a given class on the ability to produce embeddings that are useful for the separability task.

Finally, we assess the average separability scores (in terms of precision, recall and F-score) of the various knowledge graph embedding models presented in section 3.

Lexical Distance. We introduce a distance over classes, based on the semantic proximity of their labels. For a given class X, we split its label into words (e.g. `SportsTeamMember` becomes "sports", "team", "member"), and average the embeddings[1] of these words, which gives a vector representation $\vec{\mathbf{X}}$ of X. Then, we define the lexical distance between two classes A and B as the Euclidean distance between their representations:

$$d_{\text{lex}}(A, B) = \|\vec{\mathbf{A}} - \vec{\mathbf{B}}\|_2 \tag{7}$$

Taxonomic Distance. We also introduce a distance between classes based on their distance in the original taxonomy (in our experiments, we used DBpedia). Since the taxonomy is a tree, there is a unique (non-directed) path between any pair of classes $(A,\ B)$. We define the taxonomic distance between A and B as the length of this path, weighted by the depth of the edges in the path:

$$d_{\text{tax}}(A, B) = \sum_{e \in \text{path}(A,B)} \frac{1}{2^{\text{depth}(e)}} \tag{8}$$

Where $\text{depth}(e)$ is the depth of the edge in the tree: if an edge is connected to the root, it has depth 0, if it is connected to an immediate child of the root, it has depth 1, and so on. The weight of an edge decreases with its depth, because depth is an indicator of specificity in a taxonomy: the deeper a class is in the taxonomy, the more specific it is. Thus, two classes separated by only one edge are semantically closer if they are deep in the tree (e.g `MotorRace` and `Race`) rather than close to the root (e.g `Person` and `Agent`). Using the specific weighting scheme $1/2^{\text{depth}(e)}$ has the additional property that a class A is closer to its subclasses than to its siblings or to its superclasses.

[1]We used word embeddings from https://fasttext.cc/docs/en/english-vectors.html, trained on Common Crawl [9]

Table 1: Average precision, recall and F-score for different embedding models on the separability task.

Model	Precision	Recall	F-score
ComplEx	90.4	89.9	89.7
DistMult	92.5	91.6	91.6
RDF2Vec	**99.7**	**99.7**	**99.7**
TransE	99.4	99.1	99.2
TransH	93.6	92.1	92.5
TransD	85.0	83.1	83.5

Class Frequency. An entity involved in many triples will be seen many times during training, and thus may have a more reliable embedding vector. Since the embedding vector of an entity depends on the embeddings of the relations and entities that are connected to it in the knowledge graph, it is possible that entities from rare classes are involved in rare relations and linked to rare entities, so their own embeddings are less reliable than others. To assess this claim, we also investigate the effect of class size on separability scores. To provide a synthetic number for the sizes of a pair (A, B), we use the harmonic mean of the sizes of A and B. We find that using harmonic instead of arithmetic mean better handles the size imbalance that can occur between two classes: if N_A the size of A is orders of magnitude greater than the size of B, their arithmetic mean has the same order of magnitude as N_A, and thus does not reflect that B is rare comparatively to A.

We evaluate the six embedding models on 10,000 pairs of classes from DBpedia. We give the average separability score of each model in Table 1, and plot it for various lexical and taxonomic distances in Figure 1 and for various class sizes in Figure 2.

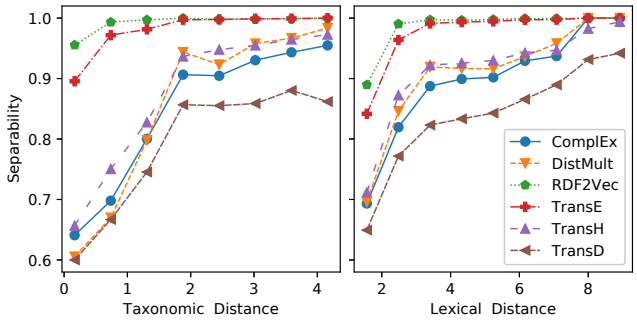


Figure 1: Average separability of two classes depending on the taxonomic distance (left) and lexical distance (right) between them, for different embedding models.

As expected, the separability score varies with the lexical and taxonomic distance between classes for all embedding models. All models but one handle correctly distant classes (lexical distance of 8 or above), but only two models have consistently high scores on most of the distance ranges: RDF2Vec and TransE, with RDF2Vec being slightly better than TransE especially for close classes. The average separability score of these two models drops below 95% when the lexical distance between classes is below 2, to reach 89% and 84% respectively.

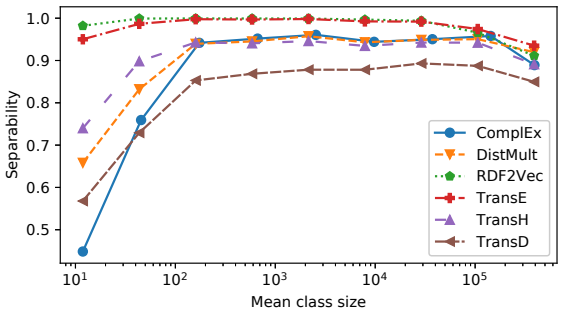


Figure 2: Average separability of two classes, depending on the harmonic mean of their number of instances, for different embedding models.

In Figure 2, we observe that rare classes are indeed less separable than more frequent classes, with some variability depending on embedding models. RDF2Vec is once again the best model, with TransE slightly behind. TransH, DistMult and ComplEx have close scores in the size range $[200, 10^5]$, but ComplEx is more sensitive to rare classes than DistMult, which is itself more sensitive than TransH. More surprisingly, we observe that scores also decrease for very frequent classes (mean size above 10^5 instances) for all embedding models. Both Pearson and Spearman's correlations between lexical distance and mean class size are close to zero, which rules out the possibility of a bias in the dataset. An explanation is that frequent classes are more likely to be parent of other classes in the dataset (for example, `dbo:Agent` is the second most frequent class of the dataset, and is a parent of half the classes in the DBpedia ontology), and separating a subclass from its parent is harder than separating sibling or unrelated classes. Finally, we observe a convergence of scores for frequent classes between all models, with TransE achieving the highest score.

In summary, two models achieve an average F-score above 95% on our separability task: RDF2Vec and TransE, with 99.7% and 99.2% respectively. In the remainder of this paper, we use TransE embeddings learned on the DBpedia knowledge base as the clusters that emerge using TransE appear more balanced in size.

4 EXPRESSIVE TAXONOMY LEARNING

Learning an expressive taxonomy is done by recursively repeating two steps: first, entity embeddings are clustered hierarchically, thus creating a structure over groups of semantically close entities. Then, clusters are labelled with axioms whose score is above a given threshold (see section 4.2). New entities are then sampled using these retained axioms, and the clustering and labelling steps are repeated with these new entities, until no relevant cluster is found.

4.1 Hierarchical Clustering

We start by sampling a set of n entities among those that verify a given axiom A_{start}. At first, $A_{\text{start}} = \top$, meaning that we sample among all entities in the knowledge graph. In later steps, and similarly to a bootstrapping approach, we sample entities that verify our newly extracted axioms.

These entities are first embedded into the Euclidean space $\mathbb{R}^d$. This $n \times d$ point cloud is then clustered using agglomerative clustering: at first, there are n clusters, each containing exactly one entity. Then, at each step, the two clusters that have the lowest average distance between their entities are merged together into a new cluster. The process is repeated until there is only one remaining cluster containing all entities. The result of this algorithm is a binary clustering tree, whose root is the whole dataset, and whose leaves are the entities. To avoid clusters that are too specific in the next step, we set a maximal depth D and remove clusters that are deeper than this. In our experiments, we used D in the range $[4-10]$. The purpose of this hierarchical clustering step is two-fold. First, it identifies groups of entities that are geometrically close, hence semantically coherent. This unsupervised, geometry-based class identification is a key point of our approach: the axiom mining step only occurs within groups of entities for which we already know that they are semantically similar. Second, it creates a hierarchy over these groups of entities, based only on the geometry of the embeddings. Once the axiom mining step is done, the clusters' hierarchy will naturally induce a hierarchy over axioms, that is, an expressive taxonomy.

4.2 Statistics based on Linked Data

Since relevant groups of entities have already been defined in the previous step, and since subsumption axioms are derived from the taxonomic structure of the clustering tree, the axiom mining step consists only in describing a cluster with one or several logical axioms. Here, we present a simple axiom mining method, but other approaches could be used. In this axiom mining scheme, we are interested in finding axioms that explain a clustering split.

Let C be a non-leaf cluster, L and R its left and right subclusters, that is $C = L \sqcup R$, with $\sqcup$ denothing the disjoint union. For an axiom A and an entity x, we write A(x) if x verifies axiom A. In order to explain the split between L and R, we want to extract an axiom A such that A is valid for all elements in L and none in R:

$$\forall x \in L, A(x) \wedge \forall x \in R, \neg A(x) \tag{9}$$

Or equivalently,

$$\forall x \in L \sqcup R, A(x) \oplus (x \in R) \tag{10}$$

With $\oplus$ denoting the exclusive or operation: an entity of $C = L \sqcup R$ cannot be in R and verify A at the same time, nor can it be in L without verifying A. We define the coverage of axiom A for the split $C = L \sqcup R$ as the proportion of elements in L that verify A, and the specificity of A as the proportion of elements in R that don't verify A. An optimal separating axiom has coverage and specificity equal to 1. We finally define a synthetic partition score for A as the arithmetic mean of coverage and specificity, weighted by the respective sizes of L and R. One can check that this partition score is indeed the proportion of elements $x \in C$ such that $A(x) \oplus x \in R$.

To find such an axiom, we first extract atomic axioms from a cluster, and then improve these atomic axioms by combining them with conjunctions, disjunctions and negations. In this paper, we only consider two types of atomic axioms: classes (concepts in description logic terminology), and existential restrictions. Three types of existential restrictions are considered: $\exists R.C$ with C a class, $\exists R.\{v\}$ with v an entity, and $\exists R.t$ with t representing litterals of

type t (e.g. `xsd:date`). For each entity x in our input cluster, we extract all the triples $(x,\ r,\ t)$ in the graph. If r is the `rdf:type` relation, then t is a class, and the triple $(x,\ r,\ t)$ is transformed into the atomic axiom t (concept axiom). Otherwise, the classes $C_1, \ldots, C_m$ of t are extracted, and the triple $(x,\ r,\ t)$ is transformed into atomic axioms $\exists R.\{t\}$ and $\exists R.C_1$, ..., $\exists R.C_m$. We end up with a list of atomic axioms for the entire cluster. As a last pruning step, we remove atoms that are verified by less than 10% of the entities.

Next, we generate candidate axioms using this set of atomic axioms. At first, candidate axioms are the atomic axioms themselves. Each candidate axiom is evaluated using the metrics of specificity and coverage. We experimentally define a score threshold δ, typically $\delta = 0.9$. If the coverage is below the threshold, the axiom does not cover enough entities. We thus start iteratively extending an axiom a by adding disjunctions (OR clauses): for each atomic axiom b that describes this cluster, we create a new candidate axiom $a \vee b$, which is in turn evaluated and extended if necessary. Conversely, if the specificity is below the threshold, the axiom does not separate well enough the two clusters, so we add conjunctions (AND clauses) as in the previous case. If both coverage and specificity are below the threshold, then the axiom is not a good candidate for explaining the split, and the search continues with other candidates. If both coverage and specificity are above the threshold, then the axiom is returned. At each step, only the N_{ax} candidate axioms with the highest scores are extended, to limit the size of the search space, where N_{ax} is a parameter. The search stops when the score does not improve enough from one step to the other, or when the number of steps is too high. This algorithm is described in Algorithm 1. For each cluster, the algorithm returns a list of axioms describing its left and right subclusters. Since at each step we consider equivalently atomic axioms and their negations, the algorithm is symmetric, so axioms in the right subcluster are the negation of axioms in the left subcluster.

ALGORITHM 1: Pseudo-code for iteratively improving a list of atomic axioms

```
Input: atoms, two clusters L and R
Output: list of axioms for the LR split
Parameters: max_steps, min_gain, score threshold δ, N_ax
found_axioms = ∅
to_improve = {None}
steps = 0
while steps <= max_steps do
    improved = ∅
    for axiom in to_improve do
        if axiom is None then
            OP = None
        else
            coverage, specificity, score = evaluate(a)
            if coverage < δ and specificity < δ then
                continue
            else if specificity < δ then
                OP = AND
            else if coverage < δ then
                OP = OR
        for atom in atoms do
            if OP is None then
                new_axiom = atom
            else
                new_axiom = atom OP axiom
            coverage, specificity, new_score =
              evaluate(new_axiom)
            if new_score > δ then
                found_axioms.push(new_axiom)
            else if new_score - score > min_gain then
                improved.push((new_axiom, new_score))
        if improved is empty then break
        to_improve := N_ax best axioms from improved
        steps := steps + 1
```

4.3 Taxonomy Learning

The final step is to extract the expressive taxonomy itself. For this, we inject the extracted axioms in the clustering tree. Clusters whose extracted axioms have low partition scores (below some threshold) are removed from the tree, meaning that they are not considered as a relevant level in the hierarchy, and their children are directly connected to their parents. If the extracted axioms do not cover all the n entities, it means that at least one subclass of A_{start} was not found, so we add a special class representing entities that verify A_{start} but none of the extracted axioms. For example, some cluster `Artist` might be divided into two labelled subclusters `Painter` and `Singer` and one unlabelled subcluster, leading to an extra class `Artist/...` of artists that are neither painters nor singers.

This results in a taxonomic tree $T(A_{\text{start}})$ over the entities that verify A_{start}. Since named classes were considered as atomic axioms, the resulting tree contains both named classes and expressive axioms. If $T(A_{\text{start}})$ is empty, there is no meaningful subclass of A_{start} and the search stops. Otherwise, for each new axiom B in $T(A_{\text{start}})$, we repeat the steps of sampling, clustering and axiom extraction: entities are sampled from B and clustered, axioms are extracted, and a taxonomy $T(B)$ is extracted. $T(B)$ is then inserted into $T(A_{\text{start}})$, and the search continues within $T(B)$. If $T(B)$ is empty, then there is no meaningful subclass of B, and the search stops. The algorithm can be further refined by decreasing the score threshold δ over time.

5 EVALUATION AND DISCUSSION

Assessing the quality of an expressive taxonomy is difficult because there is no gold standard available. In this section, we describe a quantitative evaluation of our approach on the "non-expressive" taxonomy extraction task, for which we have the DBpedia ontology as a gold standard, and we also provide a qualitative analysis of the expressive taxonomy that is obtained.

By restricting the extracted axioms to concepts (thus not considering existential relations), our algorithm can extract a taxonomy of named classes $\hat{T}$. As a gold standard, we use the DBpedia ontology[2] T^* and remove classes that have no instances, resulting in 455 classes. To compare $\hat{T}$ and T^*, we use standard evaluation metrics: precision (number of predicted axioms that are true), recall

[2]The latest version is available here: mappings.dbpedia.org/server/ontology/classes/

Table 2: Evaluation of the class subsumption axioms extracted, using DBpedia ontology as the gold standard. *Transitive* and *direct* indicates the evaluation with and without transitive closure respectively.

	Precision	Recall	*F*-score
Direct	68.94	68.79	68.87
Transitive	98.28	85.99	91.72

(number of true axioms that are predicted), and *F*-score (mean of precision and recall). Following [15], we also compute the transitive closure of $\hat{T}$ and T^*, and evaluate again the precision, recall and *F*-score. Results are shown in table 2. As we can see, the obtained F-score exceeds 90 percent, which indicates that DBpedia classes appear overall in the generated taxonomy but not necessarily at their same original depth in the DBpedia taxonomy.

We also provide a limited qualitative analysis of the expressive taxonomy[3]. Compared to the DBpedia taxonomy, which contains 455 classes, our taxonomy adds 106 expressive classes. We can notice that unpreviously defined classes emerge from the data. These emerging classes can subsume existing classes, such as the class $\texttt{Work} \wedge \exists \texttt{runtime}.\{\texttt{xsd:double}\}$ which contains all entities from $\texttt{Film} \vee \texttt{TVShow}$. We also obtain more specific subclasses such as $\texttt{Language} \wedge \exists \texttt{spokenIn}.\texttt{Place}$ that becomes a subclass of Language and is a sibling of ProgrammingLanguage.

We can also note that, at the first level, we are able to retrieve all the previously known DBpedia classes (Place, Event, Agent,...). Some classes that emerge highlight the need to complete the data instances. For example, the class Automobile is associated with the properties height, length and weight, but at least one of these properties is missing for 50% of its instances. Frequent classes appear as more accurately described than rare classes, as expected.

Overall, we believe that our generated taxonomy can dynamically reflect the content of a knowledge graph and that this approach could be used to extract up-to-date schemas. Some limitations include the difficulty to automatically determine the threshold at which to stop the decomposition of a cluster into further clusters and whether a class that emerges is valuable or only reflective of data incompleteness.

6 CONCLUSION

In this paper, we presented a methodology to extract an expressive taxonomy that represents the current content of a knowledge graph. This work is part of approaches that aim at automatically completing or creating knowledge graphs. With the evolution of representation learning based on graphs and distributional semantics based on dense vectors, we have shown that it is indeed possible to extract the taxonomy of a knowledge graph as reflected by data instances. Our future work will tackle class definitions and enrich the set of axioms considered in the generation of the knowledge graph schema. We also aim at demonstrating how the obtained taxonomy can highlight data errors, inconsistencies or incompleteness, and contribute to the interpretability of the knowledge graph. Finally, we also plan on generating mixed representations based not only on knowledge graph embeddings but also on modern language models for a more accurate schema learning.

[3]Our expressive taxonomy can be found at labowest.ca/sdb2020

ACKNOWLEDGMENTS

This research was supported by Apogée Canada, Canada First Research Excellence Fund program.

REFERENCES

[1] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. In *Advances in Neural Information Processing Systems*. 2787–2795.
[2] Lorenz Bühmann, Jens Lehmann, and Patrick Westphal. 2016. DL-Learner—A framework for inductive learning on the Semantic Web. *Journal of Web Semantics* 39 (2016), 15–24.
[3] Lara Haidar-Ahmad, Ludovic Font, Amal Zouaq, and Michel Gagnon. 2016. Entity typing and linking using sparql patterns and DBpedia. In *Semantic Web Evaluation Challenge*. Springer, 61–75.
[4] Junheng Hao, Muhao Chen, Wenchao Yu, Yizhou Sun, and Wei Wang. 2019. Universal representation learning of knowledge bases by jointly embedding instances and ontological concepts. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1709–1719.
[5] Simin Jabbari and Kilian Stoffel. 2018. FCA-Based Ontology Learning From Unstructured Textual Data. In *International Conference on Mining Intelligence and Knowledge Exploration*. Springer, 1–10.
[6] Guoliang Ji, Shizhu He, Liheng Xu, Kang Liu, and Jun Zhao. 2015. Knowledge graph embedding via dynamic mapping matrix. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 687–696.
[7] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and Philip S. Yu. 2020. A Survey on Knowledge Graphs: Representation, Acquisition and Applications. arXiv:cs.CL/2002.00388
[8] Na Li, Zied Bouraoui, and Steven Schockaert. 2019. Ontology completion using graph convolutional networks. In *International Semantic Web Conference*. Springer, 435–452.
[9] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. 2018. Advances in Pre-Training Distributed Word Representations. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*.
[10] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*. 3111–3119.
[11] Heiko Paulheim. 2017. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic web* 8, 3 (2017), 489–508.
[12] Heiko Paulheim and Christian Bizer. 2014. Improving the quality of linked data using statistical distributions. *International Journal on Semantic Web and Information Systems (IJSWIS)* 10, 2 (2014), 63–86.
[13] Rafael Peixoto, Thomas Hassan, Christophe Cruz, Aurélie Bertaux, and Nuno Silva. 2016. Hierarchical Multi-Label Classification Using Web Reasoning for Large Datasets. *Open Journal Of Semantic Web* (2016). https://doi.org/10.19210/1006.3.1.1
[14] Giulio Petrucci, Marco Rospocher, and Chiara Ghidini. 2018. Expressive ontology learning as neural machine translation. *Journal of Web Semantics* 52 (2018), 66–82. https://doi.org/10.1016/j.websem.2018.10.002
[15] Petar Ristoski, Stefano Faralli, Simone Paolo Ponzetto, and Heiko Paulheim. 2017. Large-scale taxonomy induction using entity and word embeddings. In *Proceedings of the International Conference on Web Intelligence*. ACM, 81–87. https://doi.org/10.1145/3106426.3106465
[16] Petar Ristoski and Heiko Paulheim. 2016. Rdf2vec: RDF graph embeddings for data mining. In *International Semantic Web Conference*. Springer, 498–514.
[17] Andrea Rossi, Donatella Firmani, Antonio Matinata, Paolo Merialdo, and Denilson Barbosa. 2020. Knowledge Graph Embedding for Link Prediction: A Comparative Analysis. arXiv:cs.LG/2002.00819
[18] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. 2016. Complex embeddings for simple link prediction. In *Proc. International Conference on Machine Learning*.
[19] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. 2014. Knowledge graph embedding by translating on hyperplanes. In *Twenty-Eighth AAAI conference on artificial intelligence*.
[20] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. 2015. Embedding Entities and Relations for Learning and Inference in Knowledge Bases. In *Proc. 3rd International Conference on Learning Representations*.
[21] Amal Zouaq, Dragan Gasevic, and Marek Hatala. 2011. Towards open ontology learning and filtering. *Information Systems* 36, 7 (2011), 1064–1081.

ANNEXE B RÉSULTATS COMPLETS POUR L'EXTRACTION DE TAXONOMIE

Le tableau B.1 présente les résultats complets de l'extraction de taxonomie, et inclut notamment les modèles, les métriques et les critères de liaison qui ont été omis à la section 4.3.3. Dans ce tableau, «cos», «euc», «l1» désignent respectivement les distances cosinus, euclidienne et L_1, tandis que «max», «min», «moy» et «ward» désignent le saut maximum, le saut minimum, le saut moyen et le critère de Ward.

Tableau B.1 Résultats complets pour l'extraction de taxonomie présentée à la section 4.3.3.

				Directe			Transitive		
Plongements	Méthode	Métrique	Liaison	p	r	F_1	p	r	F_1
ComplEx	MLI	cos	max	0.47	0.38	0.42	0.77	0.5	0.61
ComplEx	MLI	euc	max	0.4	0.36	0.38	0.73	0.53	0.62
ComplEx	MLI	l1	max	0.38	0.33	0.35	0.68	0.57	0.62
DistMult	MLI	cos	max	0.49	0.41	0.44	0.65	0.53	0.59
DistMult	MLI	euc	max	0.29	0.23	0.26	0.52	0.42	0.46
DistMult	MLI	l1	max	0.26	0.19	0.22	0.49	0.36	0.42
RDF2Vec	MLI	cos	max	0.3	0.23	0.26	0.58	0.35	0.44
RDF2Vec	MLI	euc	max	0.5	0.3	0.37	0.79	0.35	0.49
RDF2Vec	MLI	l1	max	0.29	0.31	0.3	0.48	0.54	0.51
TransD	MLI	cos	max	0.05	0.05	0.05	0.19	0.21	0.2
TransD	MLI	euc	max	0	0	0	0.11	0.22	0.15
TransD	MLI	l1	max	0.06	0.06	0.06	0.18	0.3	0.23
TransE	MLI	cos	max	0.76	0.64	0.69	0.92	0.64	0.75
TransE	MLI	euc	max	0.55	0.48	0.52	0.86	0.53	0.66
TransE	MLI	l1	max	0.72	0.64	0.68	0.94	0.58	0.72
ComplEx	MLM	cos	max	0.51	0.39	0.44	0.86	0.57	0.69
ComplEx	MLM	euc	max	0.42	0.34	0.38	0.81	0.57	0.67
ComplEx	MLM	l1	max	0.42	0.31	0.36	0.84	0.55	0.67
DistMult	MLM	cos	max	0.46	0.39	0.42	0.9	0.61	0.73
DistMult	MLM	euc	max	0.45	0.36	0.4	0.84	0.55	0.67
DistMult	MLM	l1	max	0.33	0.17	0.23	0.7	0.31	0.43

Tableau B.1 Résultats complets pour l'extraction de taxonomie présentée à la section 4.3.3.

				Directe			Transitive		
Plongements	Méthode	Métrique	Liaison	p	r	F_1	p	r	F_1
RDF2Vec	MLM	euc	max	0.61	0.36	0.45	0.84	0.39	0.53
RDF2Vec	MLM	cos	max	0.42	0.34	0.38	0.64	0.45	0.53
RDF2Vec	MLM	l1	max	0.39	0.38	0.38	0.52	0.53	0.53
TransD	MLM	cos	max	0.03	0.03	0.03	0.24	0.15	0.19
TransD	MLM	euc	max	0	0	0	0.12	0.15	0.13
TransD	MLM	l1	max	0	0	0	0.12	0.15	0.14
TransE	MLM	cos	max	0.7	0.59	0.64	0.88	0.65	0.75
TransE	MLM	euc	max	0.49	0.41	0.44	0.89	0.55	0.68
TransE	MLM	l1	max	0.71	0.66	0.68	0.95	0.68	0.79
TransH	MLM	cos	max	0.08	0.02	0.03	0.77	0.1	0.17
TransH	MLM	euc	max	0.17	0.02	0.03	1	0.06	0.11
TransH	MLM	l1	max	0.25	0.06	0.1	0.79	0.14	0.24
TransE	MLM	cos	max	0.54	0.69	0.6	0.88	0.69	0.77
TransE	MLM	euc	max	0.43	0.48	0.46	0.86	0.59	0.7
TransE	MLM	l1	max	0.57	0.67	0.62	0.95	0.68	0.79
DistMult	MLI	euc	min	0.26	0.2	0.23	0.34	0.59	0.44
TransE	MLM	cos	min	-	-	-	-	-	-
TransE	MLM	l1	min	-	-	-	-	-	-
ComplEx	MLI	cos	moy	0.53	0.5	0.52	0.78	0.7	0.74
ComplEx	MLI	euc	moy	0.54	0.45	0.49	0.71	0.63	0.67
ComplEx	MLI	l1	moy	0.54	0.47	0.5	0.76	0.65	0.7
DistMult	MLI	cos	moy	0.55	0.47	0.5	0.73	0.57	0.64
DistMult	MLI	euc	moy	0.42	0.36	0.39	0.64	0.55	0.59
DistMult	MLI	l1	moy	0.33	0.3	0.31	0.58	0.54	0.56
RDF2Vec	MLI	cos	moy	0.6	0.44	0.5	0.88	0.5	0.63
RDF2Vec	MLI	euc	moy	0.3	0.33	0.31	0.45	0.47	0.46
RDF2Vec	MLI	l1	moy	0.4	0.44	0.42	0.5	0.61	0.55
TransD	MLI	cos	moy	0.14	0.14	0.14	0.25	0.28	0.26
TransD	MLI	euc	moy	0.09	0.09	0.09	0.1	0.33	0.15
TransD	MLI	l1	moy	0.04	0.05	0.04	0.14	0.31	0.19
TransE	MLI	cos	moy	0.82	0.8	0.81	0.97	0.89	0.93

Tableau B.1 Résultats complets pour l'extraction de taxonomie présentée à la section 4.3.3.

				Directe			Transitive		
Plongements	Méthode	Métrique	Liaison	p	r	F_1	p	r	F_1
TransE	MLI	euc	moy	0.65	0.61	0.63	0.91	0.66	0.76
TransE	MLI	l1	moy	0.71	0.69	0.7	0.91	0.79	0.85
ComplEx	MLI	euc	ward	0.51	0.44	0.47	0.87	0.52	0.65
DistMult	MLI	euc	ward	0.43	0.36	0.39	0.75	0.54	0.63
RDF2Vec	MLI	euc	ward	0.6	0.53	0.56	0.89	0.64	0.74
TransD	MLI	euc	ward	0.08	0.08	0.08	0.17	0.19	0.18
TransE	MLI	euc	ward	0.73	0.67	0.7	**0.99**	0.68	0.8
ComplEx	MLM	cos	moy	0.52	0.48	0.5	0.89	0.7	0.79
ComplEx	MLM	euc	moy	0.53	0.44	0.48	0.86	0.65	0.74
ComplEx	MLM	l1	moy	0.53	0.44	0.48	0.8	0.63	0.71
DistMult	MLM	cos	moy	0.5	0.44	0.47	0.87	0.65	0.74
DistMult	MLM	euc	moy	0.47	0.42	0.45	0.8	0.7	0.74
DistMult	MLM	l1	moy	0.38	0.31	0.34	0.86	0.57	0.69
RDF2Vec	MLM	euc	moy	0.61	0.47	0.53	0.92	0.55	0.69
RDF2Vec	MLM	cos	moy	0.3	0.33	0.31	0.46	0.52	0.49
RDF2Vec	MLM	l1	moy	0.23	0.25	0.24	0.42	0.39	0.4
TransD	MLM	cos	moy	0.14	0.03	0.05	0.53	0.09	0.15
TransD	MLM	euc	moy	-	-	-	-	-	-
TransD	MLM	l1	moy	0.03	0.03	0.03	0.1	0.2	0.13
TransE	MLM	cos	moy	**0.83**	0.81	**0.82**	0.93	**0.93**	**0.93**
TransE	MLM	euc	moy	0.7	0.67	0.69	0.89	0.79	0.84
TransE	MLM	l1	moy	0.74	0.7	0.72	0.87	0.83	0.85
TransH	MLM	cos	moy	0.44	0.27	0.33	0.75	0.49	0.59
TransH	MLM	euc	moy	-	-	-	-	-	-
TransH	MLM	l1	moy	0.2	0.09	0.13	0.47	0.3	0.37
TransE	MLM	cos	moy	0.54	**0.89**	0.67	0.93	0.93	0.93
TransE	MLM	euc	moy	0.53	0.78	0.63	0.88	0.79	0.83
TransE	MLM	l1	moy	0.53	0.83	0.65	0.87	0.83	0.85
ComplEx	MLM	euc	ward	0.51	0.44	0.47	0.84	0.6	0.7
DistMult	MLM	euc	ward	0.48	0.42	0.45	0.86	0.65	0.74
RDF2Vec	MLM	euc	ward	0.58	0.55	0.56	0.91	0.68	0.78

Tableau B.1 Résultats complets pour l'extraction de taxonomie présentée à la section 4.3.3.

				Directe			Transitive		
Plongements	Méthode	Métrique	Liaison	p	r	F_1	p	r	F_1
TransD	MLM	euc	ward	0.03	0.03	0.03	0.23	0.16	0.19
TransE	MLM	euc	ward	0.78	0.77	0.77	0.98	0.87	0.92
TransH	MLM	euc	ward	0.34	0.16	0.22	0.94	0.28	0.43
TransE	MLM	euc	ward	0.53	0.78	0.63	0.97	0.87	0.91
ComplEx	TIEmb	cos	-	0.38	0.38	0.38	0.28	0.57	0.37
ComplEx	TIEmb	euc	-	0.39	0.42	0.4	0.34	0.8	0.48
ComplEx	TIEmb	l1	-	0.4	0.44	0.42	0.32	0.79	0.46
DistMult	TIEmb	cos	-	0.26	0.27	0.26	0.19	0.44	0.27
DistMult	TIEmb	euc	-	0.37	0.41	0.39	0.33	0.79	0.47
DistMult	TIEmb	l1	-	0.3	0.33	0.31	0.26	0.69	0.38
RDF2Vec	TIEmb	cos	-	0.77	0.84	0.81	0.43	0.87	0.57
RDF2Vec	TIEmb	euc	-	0.7	0.77	0.73	0.3	0.73	0.42
RDF2Vec	TIEmb	l1	-	0.7	0.77	0.73	0.3	0.73	0.42
TransD	TIEmb	cos	-	0.41	0.44	0.42	0.37	0.77	0.5
TransD	TIEmb	euc	-	0.27	0.3	0.28	0.18	0.51	0.26
TransD	TIEmb	l1	-	0.41	0.45	0.43	0.37	0.76	0.5
TransE	TIEmb	cos	-	0.77	0.72	0.74	0.83	0.89	0.86
TransE	TIEmb	euc	-	0.76	0.75	0.76	0.7	0.9	0.79
TransE	TIEmb	l1	-	0.75	0.77	0.76	0.56	0.9	0.69
TransH	TIEmb	cos	-	0.46	0.5	0.48	0.27	0.77	0.4
TransH	TIEmb	euc	-	0.39	0.42	0.4	0.19	0.58	0.28
TransH	TIEmb	l1	-	0.43	0.47	0.45	0.19	0.59	0.29