Détection d'entités quasi-dupliquées dans une base de connaissances avec PIKA

Maxime Prieur*, Guillaume Gadek*
Bruno Grilheres*

*Airbus Defence and Space
Elancourt, France
guillaume.gadek [at] airbus.com

Résumé. Cet article explore l'utilisation des modèles de réseaux de neurones adaptés aux graphes pour produire des représentations vectorielles des noeuds afin de résoudre le problème de la détection d'éléments similaires dans une base de connaissances. En s'appuyant sur des modèles pré-entraînés pour la similarité sémantique textuelle, notre méthode proposée, `PIKA`, agrège les caractéristiques hétérogènes (structurées et non structurées) d'une entité et de son voisinage pour produire un vecteur pouvant être utilisé dans différentes tâches telles que la recherche d'information ou la classification. Notre méthode apprend des poids spécifiques pour chaque type d'information apportée par une entité, ce qui nous permet de la traiter de manière inductive.

1 Introduction

Dans une base de connaissances, les éléments partageant des relations sont liés entre eux et peuvent ainsi être visualisés sous la forme d'un graphe de connaissances, très utile pour des tâches telles que l'analyse des réseaux sociaux, la prise de décision, la réponse automatique à des questions ou encore la recommandation de produits à l'utilisateur (Dai et al., 2020; Guo et al., 2020; Xiaohan, 2020). Lorsqu'une telle base de données est construite automatiquement (par l'analyse d'un flux continu de données), la tâche est nommée population automatique de bases de connaissances.

Dans une telle base, il existe un risque d'ajouter un élément déjà existant en raison de fautes de frappe dans les valeurs des attributs, de la mise à jour simultanée de la base de connaissances par plusieurs utilisateurs (humains ou IA) ou de l'ajout d'une entité avec des informations à jour sans se soucier de celle déjà présente dans la base de données. Ces cas sont très préoccupants : ils introduisent des redondances, polluent les données et ont un impact sur les performances des différentes tâches d'exploitation. Dans cet article, nous proposons une solution pour éviter ce type d'erreurs en détectant les entités similaires dans une base de connaissances avant d'insérer de nouveaux éléments.

À notre connaissance, ce problème n'a pas encore été entièrement résolu. Une solution appropriée devrait prendre en compte les différents types d'entités stockées dans une base de données (par exemple Personne, Lieu, etc...), être capable de s'adapter à un grand nombre

d'éléments (milliards) avec des descriptions hétérogènes et tirer parti à la fois de la description de l'entité (attributs) et de son voisinage (relations et attributs des voisins).

Dans cet article, nos principales contributions sont les suivantes :

— nous introduisons une étape de prétraitement pour agréger les attributs d'une entité dans un vecteur unique ;
— nous sommes les premiers à proposer un modèle GCN rapide et fiable, `PIKA`, capable de produire des embeddings de nœuds dans un graphe hétérogène, afin de retrouver des entités similaires ou quasi-dupliquées ;
— nous effectuons une évaluation et une comparaison approfondies du modèle que nous proposons par rapport aux méthodes GCN les plus récentes, en récupérant les clones générés à partir de la base de données KBP du TAC.

Le reste de cet article est structuré comme suit. La section 2 passe en revue les travaux récents concernant les méthodes de représentations vectorielles des entités d'une base de connaissances. La section 3 est une description détaillée de notre modèle `PIKA`. La section 4 formalise l'expérience menée pour mesurer la qualité de la détection de dupliqués. Enfin, la section 5 montre et décrit les résultats obtenus avant de discuter des pistes à explorer dans des travaux futurs.

2 État de l'art

2.1 Correspondance des entités

La tâche d'alignement d'entités (*entity matching*) consiste à déterminer la similarité entre deux enregistrements de la même base de données ou de bases de données différentes ; cette tâche peut être abordée de plusieurs manières.

Tout d'abord l'exploitation des attributs, par exemple en les projetant dans un espace vectoriel, simplifie la tâche de comparaison des entités (Mudgal et al., 2018). Deux autres modèles, Ditto et Entity Matching on Unstructured Data, proposés respectivement par Li et al. (2020) et Brunner et Stockinger (2020), ont choisi d'affiner des modèles de langage pré-entraînés tels que Roberta (Liu et al., 2019), en donnant en entrée la concaténation des deux enregistrements sérialisés. Une approche plus récente, HierMatcher (Fu et al., 2020), utilise des représentations au niveau du mot, de l'attribut et de l'entité.

Ces méthodes n'intègrent pas l'aspect *graphe* d'une base de connaissances puisqu'elles sont appliquées à des enregistrements atomiques non liés entre eux.

2.2 Embeddings basés sur la relation

Les méthodes d"obtention d'embeddings de nœuds utilisent le voisinage d'un nœud pour le définir ; les principales implémentations sont DeepWalk (Perozzi et al., 2014) et Node2Vec (Grover et Leskovec, 2016), inspirées de Word2Vec (Mikolov et al., 2013). Ils génèrent des marches aléatoires de longueur fixe autour de chaque nœud afin de construire un contexte utilisé pour produire l'incorporation du nœud.

Plus axée sur l'existence d'une relation entre deux enregistrements, l'approche "translationnelle" choisit de modéliser les liens comme des traductions d'une entité à une autre (Bordes

et al., 2013; Trouillon et al., 2016; Dettmers et al., 2018). Les embeddings sont calculés en utilisant une fonction de perte basée sur la marge réduisant la distance des triplets réels. Rotate, de Sun et al. (2019), traite les relations comme une rotation dans un espace vectoriel pour permettre de prendre en compte la symétrie, l'antisymétrie, l'inversion et la composition. Une fois encore, le problème de la transductivité se pose. De plus, garder comme objectif d'apprentissage les prédictions des relations n'est pas optimal pour la détection des similarités.

2.3 Réseaux de neurones sur les graphes : GNN

Le développement récent des architectures d'apprentissage profond, notamment des couches de convolution, a suscité un intérêt pour leur réutilisation dans le traitement des graphes (Kipf et Welling, 2017). Les informations autour du nœud peuvent être agrégées et mises à jour itérativement pour obtenir la représentation d'une cible, de manière inductive avec le modèle GraphSage (Hamilton et al., 2017). Ce type de méthode est généralement limité à une profondeur de 1 ou 2 couches en raison d'un problème de sur-lissage (Wu et al., 2019) et pour maintenir des temps de traitement faibles. Néanmoins, GraphSage fournit des poids appris réutilisables pour gérer les nouveaux nœuds.

Puisque tous les voisins n'ont pas la même influence sur une entité, GAT, un modèle GNN utilisant des projections linéaires et un mécanisme d'auto-attention, a été introduit (Veličković et al., 2017). En argumentant que l'attention utilisée par GAT n'est pas dynamique et peine à traiter l'influence réelle des nœuds, une modification de la séquence d'opérations du modèle précédent a été proposée pour résoudre le problème (Brody et al., 2021).

Les approches mentionnées ci-dessus ne tiennent pas compte de l'hétérogénéité du graphe, un aspect qui est couvert par HinSage, une adaptation de GraphSage implémentée dans StellarGraph [1]. HinSage regroupe et fait la moyenne des vecteurs du voisinage en fonction du type de relation ou de voisin. La méthode appelée HGT calcule le coefficient d'attention des voisins, le message passé par ceux-ci et fusionne le tout dans un nouveau vecteur. Toutes les opérations impliquent des projections linéaires spécifiques aux classes et aux relations. Cela rend son utilisation complexe dans le cas d'un graphe de connaissances en raison du grand nombre d'arêtes différentes.

2.4 Discussion

Notre examen des travaux connexes met en évidence le besoin d'une solution capable de :

— Tirer le meilleur parti des informations supplémentaires apportées par la nature des graphes de connaissances (c'est-à-dire le type de relations et la classe des nœuds) ;
— Effectuer la recherche assez rapidement pour être capable de proposer des entités similaires en temps réel ou presque parmi des milliers d'éléments ;
— Opérer de manière inductive pour permettre le traitement de nouveaux noeuds sans avoir à s'entraîner sur le modèle et à calculer une fois de plus les représentations de tous les noeuds ;
— Transformer le profil du noeud en un vecteur sans perte de contenu, dans le cas d'attributs épars ou riches.

Cet article vise à répondre à ces besoins en proposant une adaptation des modèles GCNs.

1. `https://stellargraph.readthedocs.io`

3 Description de la tâche et formalisation du modèle

3.1 Formalisation des données

Une base de connaissances est une structure de données utilisée pour stocker des entités hétérogènes avec des informations structurées. Cette structure peut être modélisée comme un graphe hétérogène attribué, $G = (V, E, \Lambda, M, A)$. Où, V est l'ensemble des nœuds constituant le graphe, $\epsilon_{i,j} \in E$ est un lien orienté entre deux nœuds, Λ est l'association de chaque entité à une classe, M est l'association d'une arête $\epsilon_{i,j}$ à son type et $\alpha(i) = \{\alpha(i)_1, ...\alpha(i)_k\} \in A$, est l'ensemble des attributs possédés par le nœud $v(i) \in V$. Un attribut est constitué d'un nom et de l'embedding de sa représentation textuelle, $\alpha(i)_j = (\alpha(i)_{j,name}, \alpha(i)_{j,val})$.

3.2 Architecture du modèle

Notre modèle est composé de deux parties : le calcul de vecteurs de profil, illustré dans la figure 1, et le modèle de graphe, en figure 2. Étant donné que les méthodes de graphe nécessitent un vecteur pour chaque entité, une première étape consiste à construire un vecteur résumant les informations du profil. La deuxième étape consiste en un modèle neuronal adapté à la nature des graphes de connaissances. Les deux parties peuvent être entraînées séparément.

Pour stabiliser et réduire le temps d'apprentissage, nous utilisons l'approche d'apprentissage k-head introduite par Vaswani et al. (2017) et réutilisée avec Veličković et al. (2017). Cette méthode consiste à entraîner plusieurs versions des paramètres du modèle en parallèle. Les représentations obtenues pour chaque tête sont ensuite moyennées en un seul vecteur.

3.3 Calcul d'entrée

Pour la première étape de calcul de vecteurs de profil, nous nous appuyons sur un encodeur pré-entraîné pour transformer les chaînes de caractères en vecteurs, comme Universal Sentence Encoder (Cer et al., 2018), ou encore Mpnet (Song et al., 2020) et DistilRoberta, un modèle plus léger adapté de Liu et al. (2019). Ces modèles ont été entraînés pour mesurer la similarité sémantique entre deux phrases. Pour les deux derniers modèles, nous avons utilisé les instances proposées par la bibliothèque *sentence-transformers*[2] ; elles ont été calculées en suivant le processus décrit par Reimers et Gurevych (2019). Nous avons choisi le Universal Sentence Encoder du fait de sa qualité comparable aux meilleurs, pour un temps de traitement nettement moindre.

Une fois que toutes les valeurs des champs constituant une entité sont encodées, il faut fusionner leurs informations. Tout d'abord, des coefficients spécifiques aux attributs, $w_{\psi(\alpha(i)_{j,name})}$, sont utilisés pour pondérer les vecteurs des caractéristiques d'un attribut, $\alpha(i)$. ψ est la fonction faisant correspondre le nom de l'attribut à son poids et $\alpha(i)_{j,name}$, le nom du j-ième attribut du nœud i. Ensuite, ces vecteurs pondérés sont agrégés. Pour cette étape, nous avons retenu l'*extreme-Pooling*.

L'équation 1 montre comment nous calculons l'intégration de profil, h_p, d'un nœud i avec k attributs, en utilisant l'extreme-pooling. Les poids et les caractéristiques des attributs ont des valeurs dans $\mathbb{R}^d$, d étant la longueur des vecteurs de caractéristiques. La $\bullet$ représente le produit de Hadamar.

2. https://www.sbert.net

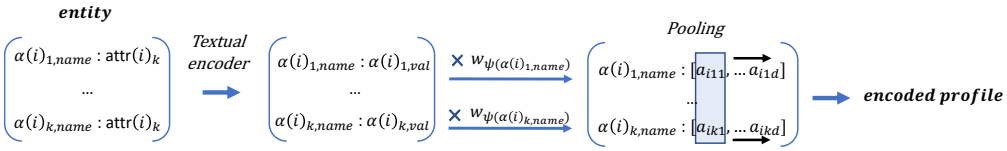

FIG. 1 – *Schéma de l'encodeur de profils*

Modèle	Dirigé	Relations hétérogènes	Types de nœuds	Attention	Entraînement itératif
GAT	✗	✗	✗	✓	✓
GATV2	✗	✗	✗	✓	✓
GATV2*	✓	✓	✓	✓	✓
Graphsage	✗	✗	✗	✗	✓
Graphsage*	✗	✗	✓	✗	✓
HinSage	✗	✓	✓	✗	✓
HGT	✓	✓	✓	✓	✓
PIKA-simple	✓	✓	✓	✓	✗
PIKA	✓	✓	✓	✓	✓

TAB. 1 – *Caractéristiques des modèles*

$$h_p(i) = extreme(\{w_{\psi(\alpha(i)_{j,name})} \bullet \alpha(i)_{j,val}\}) \quad for\ j \in \{1, ..., k\} \tag{1}$$

Puisque le profil ne dépend pas de la relation, une fois qu'il est calculé, il ne doit être mis à jour que lorsqu'une modification est apportée à l'entité elle-même. Cependant, l'utilisation de poids spécifiques par attribut rend impossible l'inférence sur des *types* de champs non vus. Dans le cas où les entités ont peu d'attributs, les vecteurs de pondération peuvent facilement être remplacés par des matrices.

3.4 Prise en compte des voisins par les réseaux de neurones à graphes

Nous avons l'intuition que la tâche serait mieux résolue en prenant en compte non seulement les profils des nœuds, mais aussi leurs voisinages. Pour ce faire, nous nous appuyons sur un GNN. En effet, l'état de l'art souligne la pertinence de quelques types d'architecture : GAT, GATV2, GraphSage, HinSage, HGT. Les aspects couverts par ces derniers modèles sont résumés dans la Table 1.

Nous adaptons ces modèles pour prendre en entrée les vecteurs du nœud cible et de tous ses voisins. Dans le cas d'un modèle prenant en compte le type du nœud et du voisinage, nous ajoutons une entrée correspondant à un vecteur d'indices. Ces indices associent le nœud cible à sa classe et les voisins à leur type de relation.

3.5 PIKA : *Performant Identification of clones in KnowledgeBase*

Nous présentons ici en détail PIKA ("Identification Performante de clones dans une Base de Connaissances"), le modèle qui a conduit aux meilleures performances. Il s'agit d'une adap-

tation de GAT, ajustée pour tirer parti des spécificités des graphes de connaissances. Les paragraphes suivants décrivent les opérations effectuées pour produire la représentation des nœuds.

$$h(i)_{i,j} = w_{\phi(i,j)} \bullet h_p(j) \tag{2}$$

Tout d'abord, chaque voisin du nœud cible est multiplié de manière indicielle par un poids spécifique au type de relation. $h_p(j)$ est la représentation du profil du nœud j, w le vecteur de poids (tous deux dans $\mathbb{R}^d$) et $\phi(i,j)$ est le type de relation entre le nœud i et le nœud j. ($\phi(i,j) = \phi(j,i)^{-1}$ est la relation inverse, telle que *father_of* et *child_of*). Pour permettre la prise en compte d'un grand nombre de types d'arêtes, nous avons utilisé des vecteurs, car les matrices nécessiteraient le calcul d'un trop grand nombre de paramètres. Les équations 3, 4, 5 reproduisent les opérations expliquées dans Veličković et al. (2017).

Dans l'équation 3, le coefficient d'attention $e(i)_{i,j}$ de chaque arête liée au nœud i est calculé en multipliant un vecteur d'attention $\beta \in \mathbb{R}^{d*2}$ à la concaténation des représentations des deux entités. Ces coefficients sont ensuite normalisés par Softmax dans l'équation 4.

$$e(i)_{i,j} = LeakyReLU(\beta.[h(i)_{i,j}||h(i)_{i,i}]) \tag{3}$$

$$\beta(i)_{i,j} = \frac{e^{e(i)_{i,j}}}{\sum_{k \in Neigh(i)} e^{e(i)_{i,k}}} \tag{4}$$

La dernière étape est une agrégation suivie d'une transformation non linéaire pour calculer $h(i)$ la représentation finale du nœud i. $Neigh(i)$ correspond au voisinage du nœud i, plus une relation avec lui-même. Comme plusieurs liens peuvent exister entre deux nœuds, le voisin concerné peut apparaître plusieurs fois dans l'équation 5.

$$h(i) = \sigma(\sum_{j \in Neigh(i)} \beta(i)_{i,j} \bullet h(i)_{i,j}) \tag{5}$$

Puisque le voisinage direct d'un nœud est suffisant pour différencier les paires clonées et non clonées, nous nous limitons à des modèles avec une seule couche convolutive, ce qui limite la mémoire utilisée et le temps de calcul de la représentation. Cela permet également de prendre en compte tous les voisins sans nécessité d'une stratégie d'échantillonnage comme celle utilisée dans Hamilton et al. (2017).

Contrairement à l'étape de calcul du vecteur résumant le profil, si un changement intervient au niveau des attributs, il est alors nécessaire de recalculer les représentations finales des nœuds voisins.

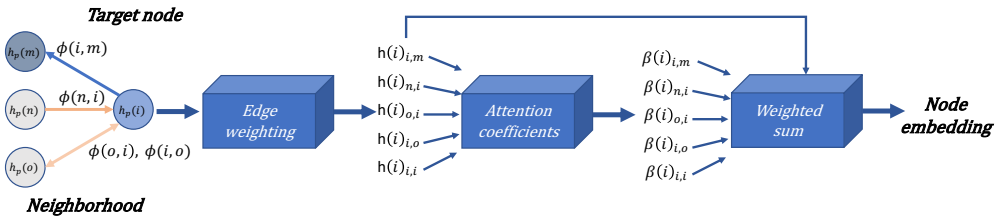

FIG. 2 – *Schéma du modèle* PIKA

3.6 Recherche de quasi-dupliqués dans les espaces vectoriels

Afin de retrouver les entités les plus similaires sur la base de leur embedding, nous utilisons un outil de recherche approximative du plus proche voisin : Annoy [3]. Inspiré des *kd-trees*, il utilise des projections aléatoires et associe un hyperplan aléatoire à chaque nœud de l'arbre de recherche. L'inconvénient de ces bibliothèques est la nécessité de mettre à jour l'index de recherche chaque fois qu'un nouveau nœud est ajouté à la liste des entités récupérables. Nous utilisons la similarité en cosinus pour comparer les vecteurs : c'est la mesure la plus courante pour traiter les embeddings.

3.7 Procédure d'entraînement

3.7.1 Mise en œuvre

Tous les modèles ont été implémentés à l'aide de Pytorch et entraînés en tant que classifieurs de manière supervisée, en suivant la méthode expliquée dans Reimers et Gurevych (2019). Cette approche consiste à entraîner un classificateur binaire avec des paires d'entités en entrée. Ce classifieur peut être utilisé en inférence pour effectuer la classification ; la dernière couche de prédiction peut être supprimée pour obtenir directement des vecteurs représentatifs, plus utiles pour la tâche de recherche d'entités. Notre procédure d'apprentissage modifie simplement la sortie donnée au classificateur tel que décrit initialement dans le cadre. Au lieu d'utiliser la concaténation des deux embeddings produits et leur différence absolue, nous ne conservons que la différence car elle produit de meilleurs résultats lors de nos expériences. En ce qui concerne le nombre d'attributs, seuls les 4 000 types d'attributs les plus présents sont pris en compte, ainsi que les 2 000 types d'arêtes (multipliés par deux puisque la direction du lien est considérée). La dimension des embeddings finaux est fixée à 512.

En ce qui concerne les modules de formation, nous utilisons la fonction de loss d'entropie croisée binaire pour aider à calculer le gradient :

$$loss = -\frac{1}{|N|}\sum_{i=1}^{|N|}[y_i log(p) + (1-y_i) log(1-p)] \tag{6}$$

$|N|$ est le nombre d'échantillons dans l'ensemble d'apprentissage N et p la probabilité que y_i soit prédit par le modèle. Pour mettre à jour les poids des modèles, nous utilisons l'optimiseur Adam (Kingma et Ba, 2017) et nous fixons le taux d'apprentissage à $lr = 10^{-3}$.

3.8 Procédure d'apprentissage itérative

Nous proposons une procédure d'apprentissage itérative qui, après chaque itération, ajoute les paires retournées par le modèle comme étant les plus similaires. Les éléments à partir desquels les requêtes les plus similaires sont effectuées sont ceux des ensembles de test et de validation initiaux, tandis que les instances retournées sont celles qui ne sont pas déjà dans l'un des ensembles. Cette procédure est utilisée pour l'étape d'encodage de profil ; l'utiliser pour l'étape d'agrégation de voisinage nécessiterait de générer d'abord une séparation train/test avec des précautions spécifiques (pour éviter qu'un même voisinage soit à la fois dans les ensembles

3. `https://github.com/spotify/annoy`

train et test). Les méthodes (dont PIKA) utilisant cette procédure itérative sont marquées dans le tableau 1.

4 Montage expérimental

4.1 Les jeux de données

4.1.1 Base de connaissances de référence TAC KBP

Afin de développer, tester et évaluer les modèles, le Linguistic Data Consortium (LDC) a développé la base de connaissances de référence TAC KBP. Cette base de connaissances est construite à partir d'articles de Wikipédia en anglais. Plus de $800\ 000$ entités sont réparties en quatre classes : Personne (**PER**), Entité géopolitique (**GPE**), Organisation (**ORG**) et Inconnu (**UKN**). Fourni comme un corpus textuel, TAC KBP peut cependant être modélisé comme un graphe de connaissances en utilisant comme relations les références à d'autres entités dans les textes descriptifs.

4.1.2 Génération de clones

TAC KBP étant considéré comme exempt d'entités dupliquées, il a été nécessaire de générer des clones à partir des instances de la base. Pour ce faire, un script génère à partir d'un tirage aléatoire d'une entité modèle (qui n'a pas déjà été utilisée) un clone avec des altérations de ses relations et attributs. Ces altérations reproduisent au mieux les erreurs constatées dans des cas d'utilisation réelle et ont des probabilités d'occurrence arbitraires.

En ce qui concerne les attributs, les valeurs des champs peuvent être permutées, les valeurs peuvent être tronquées de façon aléatoire, des changements/insertions de caractères peuvent se produire pour simuler des erreurs de frappe, des attributs peuvent être supprimés. En ce qui concerne les arêtes, certaines peuvent être supprimées, ajoutées avec une paire aléatoire de voisins et de prédicats ou le type de relation des arêtes existantes peut être remplacé. Pour éviter les biais dans l'évaluation, une entité ne peut être utilisée que pour produire un seul clone et un clone ne peut être utilisé comme modèle.

Pour tester de manière approfondie les méthodes, deux types de clones ont été générés. Le premier avec peu d'attributs supprimés (probabilité de suppression fixée à $0,2$), qualifié de **clones riches** et correspondant à une recherche de clones exacts. Le deuxième type, avec beaucoup de suppression (probabilité de suppression fixée à $0,5$) pour le cas d'une recherche par requête et sera qualifié de **clones pauvres**. Les autres modifications ont la même probabilité d'occurrence pour les deux types.

4.1.3 Distribution des ensembles de données

Comme le montre la figure 3, les ensembles de données sont constitués de paires de nœuds "$(entit_A, entit_B)$", associés à une étiquette "cloné" ou "non-cloné". Il y a respectivement 16 000, 9 600 et 6 400 paires dans les jeux de formation, de validation et de test. Les ensembles de données sont composés de 50% de paires clonées.

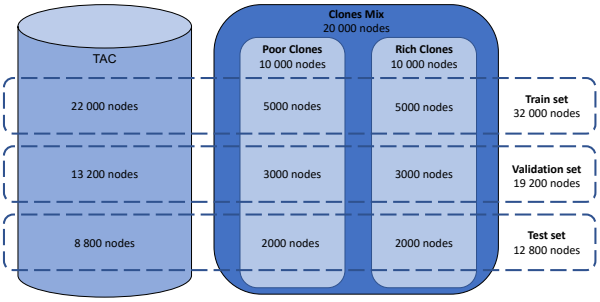

FIG. 3 – *Distribution of the dataset*

4.2 Procédure d'évaluation

Les méthodes sont évaluées sur la tâche de recherche d'entités similaires : comme dans un problème de recherche d'information, pour toutes les paires similaires dans l'ensemble de test, nous soumettons la première entité du tuple comme requête. On mesure ensuite la position de l'entité analogue dans la liste ordonnée par similarités avec l'embedding du nœud de la requête.

4.3 Métriques

Dans le cadre de la recherche d'entités similaires, nous avons utilisé le score $Hit@k$ (avec $k \in (1, 10, 100)$) et le MRR (Mean Reciprocal Rank). Le score $Hit@k$ mesure la capacité moyenne d'un système à retourner la cible dans les k premiers éléments, tandis que le MRR est utilisé pour évaluer la performance du système en général.

5 Résultats et discussion

Nous comparons notre méthode proposée `PIKA` avec des versions de GraphSage, HGT, HinSage, GAT et GATV2 que nous avons implémentées sous Pytorch. En plus d'être testées avec leurs implémentations standard, quelques modifications sont ajoutées pour prendre en compte les spécificités des KDBs. Nous mesurons également les performances de `PIKA` sans la procédure d'entraînement itérative sous le label `PIKA`-simple, afin d'évaluer la valeur ajoutée de cette procédure.

L'introduction de l'apprentissage itératif améliore réellement la qualité du modèle : `PIKA` surpasse clairement `PIKA`-simple, son implémentation non itérative. Ce gain est encore plus important lorsqu'on passe des modèles de base aux modèles modifiés pour prendre en compte les caractéristiques d'une KDB (notés par un *). En effet, on constate clairement un meilleur MRR pour les versions KDB, avec une exception pour GATV2-* sur les clones riches. L'amélioration est due, tout d'abord, au fait que l'orientation des arcs est incluse, ce qui est un paramètre déterminant pour les lieux par exemple (type GPE). De tels nœuds, qui sont souvent liés à une myriade de voisins entrants, seraient mieux représentés par leurs arêtes sortantes. Deuxièmement, nos méthodes différencient les poids en fonction de la classe du nœud dans

	Clones pauvres				Clones riches			
Modèle	MRR	Hit@1	Hit@10	Hit@100	MRR	Hit@1	Hit@10	Hit@100
GAT	0.257	0.206	0.356	0.534	0.532	0.472	0.631	0.764
GATV2	0.636	0.582	0.738	0.850	0.810	0.774	0.874	0.936
GATV2*	0.720	0.678	0.799	0.886	0.752	0.711	0.825	0.903
Graphsage	0.449	0.388	0.553	0.740	0.783	0.752	0.841	0.913
Graphsage*	0.484	0.432	0.579	0.764	0.820	0.793	0.871	0.928
HinSage	0.110	0.091	0.143	0.235	0.521	0.489	0.585	0.699
HGT	0.141	0.119	0.178	0.281	0.450	0.426	0.497	0.600
PIKA-simple	0.817	0.783	0.884	0.928	0.918	0.899	0.950	0.971
PIKA	**0.856**	**0.828**	**0.908**	**0.941**	**0.959**	**0.949**	**0.979**	**0.988**

TAB. 2 – *Résultats sur les clones pauvres (à gauche) et sur les clones riches (à droite). Les modèles avec une étoile "*" sont adaptés aux graphes de connaissance.*

le mécanisme d'auto-attention. Pour certains types, il est préférable de conserver davantage d'informations sur le nœud. Enfin, distinguer la nature des connexions permet une meilleure adaptabilité du modèle pour différentes topologies.

D'après les résultats précédents, nous pouvons également constater que les modèles non adaptés à la structure de la KDB ont du mal à s'appuyer sur les informations fournies par le voisinage, puisque leur MRR chute de 0,1 à 0,15 point lorsqu'on compare les résultats du même modèle sur les ensembles riches et pauvres. Compte tenu de la rareté des liens dans la base de connaissances (une médiane de trois arêtes par nœud et 14% des nœud ne possèdent aucune arête), nous pensons que les méthodes s'appuyant fortement sur le contexte seront encore plus performantes sur des graphes plus denses.

La proportion de faux positifs est encore trop élevée (indiquée par des scores de $Hit@1$ trop faibles) pour permettre un système de détection de similarité *automatique* : une validation humaine est encore nécessaire, mais est devenue plus efficace ($Hit@10$ et $Hit@100$ suffisants).

En ce qui concerne la contrainte de temps des méthodes d'embedding utilisant des graphes, nous mesurons une moyenne de $0,16\ s$ nécessaire pour calculer l'embedding d'une entité et retrouver l'élément le plus similaire dans notre base de connaissances. Cette courte durée rend réaliste l'intégration de PIKA dans des systèmes à l'échelle du Web.

6 Conclusion

Nous avons présenté un modèle en deux parties qui est capable de résumer les caractéristiques et le voisinage d'un nœud en un vecteur utilisable pour comparer les similarités et retrouver les entités proches en termes de caractéristiques et de relations. Comme les poids du modèle ne sont pas spécifiques aux nœuds mais aux types d'attributs et de relations, il est applicable à d'autres graphes avec les mêmes types d'attributs et d'arêtes.

Pour une application plus spécifique, notre méthode pourrait facilement être adaptée pour attribuer des poids plus élevés aux attributs sur lesquels il faut se concentrer. En soumettant des paires de clones avec des modifications spécifiques pendant l'entraînement, l'utilisateur serait en mesure de détecter des similarités uniquement sur des attributs ciblés.

Nous avons obtenu un modèle plus robuste aux paires d'entités ambiguës, en appliquant un apprentissage itératif. Néanmoins, l'intégration d'une entité complexe empêche une comparaison atomique efficace de ses attributs (comme le font les méthodes d'alignement d'entités) puisque l'information est lissée en un seul vecteur. De plus, les résultats ont montré que les entités recherchées, si elles ne sont pas retournées en première position, sont presque toujours dans les 100 premiers résultats. Ceci étant, nous pouvons facilement imaginer l'utilisation de notre modèle comme une étape de pré-filtrage sur une grande base de données avant d'effectuer une comparaison entité par entité ou d'utiliser une méthode plus précise mais plus lente sur moins de candidats.

Enfin, les valeurs numériques ne sont pas traitées de manière optimale par l'encodeur puisqu'elles apparaissent dans la base de données sous forme de chaînes de caractères. Les encodeurs de texte tels que USE ou Mpnet ne traitent pas spécifiquement les valeurs numériques : nous pensons qu'un traitement dédié à ces attributs est nécessaire afin d'atteindre une qualité acceptable pour la tâche de classification des clones.

Références

Bordes, A., N. Usunier, A. Garcia-Duran, J. Weston, et O. Yakhnenko (2013). Translating embeddings for modeling multi-relational data. *Advances in neural information processing systems 26*.

Brody, S., U. Alon, et E. Yahav (2021). How attentive are graph attention networks ? *arXiv preprint arXiv :2105.14491*.

Brunner, U. et K. Stockinger (2020). Entity matching with transformer architectures-a step forward in data integration. In *International Conference on Extending Database Technology, Copenhagen, 30 March-2 April 2020*. OpenProceedings.

Cer, D., Y. Yang, S.-y. Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Céspedes, S. Yuan, C. Tar, et al. (2018). Universal sentence encoder. *arXiv preprint arXiv :1803.11175*.

Dai, Y., S. Wang, N. N. Xiong, et W. Guo (2020). A survey on knowledge graph embedding : Approaches, applications and benchmarks. *Electronics 9*(5).

Dettmers, T., P. Minervini, P. Stenetorp, et S. Riedel (2018). Convolutional 2d knowledge graph embeddings.

Fu, C., X. Han, J. He, et L. S. 0001 (2020). Hierarchical matching network for heterogeneous entity resolution. In *IJCAI*, pp. 3665–3671.

Grover, A. et J. Leskovec (2016). node2vec : Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 855–864.

Guo, Q., F. Zhuang, C. Qin, H. Zhu, X. Xie, H. Xiong, et Q. He (2020). A survey on knowledge graph-based recommender systems. *IEEE Transactions on Knowledge and Data Engineering*.

Hamilton, W. L., R. Ying, et J. Leskovec (2017). Inductive representation learning on large graphs. In *NIPS*.

Kingma, D. P. et J. Ba (2017). Adam : A method for stochastic optimization.

Kipf, T. N. et M. Welling (2017). Semi-supervised classification with graph convolutional networks.

Li, Y., J. Li, Y. Suhara, A. Doan, et W.-C. Tan (2020). Deep entity matching with pre-trained language models. *arXiv preprint arXiv :2004.00584*.

Liu, Y., M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, et V. Stoyanov (2019). Roberta : A robustly optimized bert pretraining approach. *arXiv preprint arXiv :1907.11692*.

Mikolov, T., K. Chen, G. Corrado, et J. Dean (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv :1301.3781*.

Mudgal, S., H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, et V. Raghavendra (2018). Deep learning for entity matching : A design space exploration. In *Proceedings of the 2018 International Conference on Management of Data*, pp. 19–34.

Perozzi, B., R. Al-Rfou, et S. Skiena (2014). Deepwalk : Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 701–710.

Reimers, N. et I. Gurevych (2019). Sentence-bert : Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv :1908.10084*.

Song, K., X. Tan, T. Qin, J. Lu, et T.-Y. Liu (2020). Mpnet : Masked and permuted pre-training for language understanding. *arXiv preprint arXiv :2004.09297*.

Sun, Z., Z.-H. Deng, J.-Y. Nie, et J. Tang (2019). Rotate : Knowledge graph embedding by relational rotation in complex space.

Trouillon, T., J. Welbl, S. Riedel, Éric Gaussier, et G. Bouchard (2016). Complex embeddings for simple link prediction.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, et I. Polosukhin (2017). Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008.

Veličković, P., G. Cucurull, A. Casanova, A. Romero, P. Lio, et Y. Bengio (2017). Graph attention networks. *arXiv preprint arXiv :1710.10903*.

Wu, F., T. Zhang, A. H. de Souza Jr. au2, C. Fifty, T. Yu, et K. Q. Weinberger (2019). Simplifying graph convolutional networks.

Xiaohan, Z. (2020). A survey on application of knowledge graph. *Journal of Physics : Conference Series 1487*, 012016.

Summary

This paper explores the use of Graph Neural Network models producing node embeddings, in order to solve the not fully addressed problem of detecting similar items stored in a knowledge base. Leveraging pre-trained models for textual semantic similarity, our proposed method `PIKA` aggregates heterogeneous (structured and unstructured) characteristics of an entity and its neighborhood to produce an embedding vector that can be used in different tasks such as information retrieval or classification tasks. Our method learns specific weights for each information brought by an entity, enabling us to process it in an inductive fashion.