De représentations de documents à programmes : l'hypothèse distributionnelle peut-elle vraiment être utilisée sur les langages de programmation ?[1]

Thibaut Martinet* Guillaume Cleuziou* Matthieu Exbrayat* Frédéric Flouvat**

*prénom.nom@univ-orleans.fr,
**prénom.nom@univ-amu.fr

Résumé. Les nombreux modèles d'apprentissage profond développés pour manipuler du code informatique s'appuient quasi-exclusivement sur des modèles dédiés au langage naturel et à son hypothèse distributionnelle. Nous proposons ici d'évaluer si cette hypothèse s'applique réellement au code informatique. Nous nous appuyons sur plusieurs méthodes d'exploration que nous appliquons à différentes variantes d'un modèle classique du traitement automatique des langues : doc2vec, modèle simple, facile à comprendre et à adapter. Entre autres contributions nous proposons un jeu de programmes permettant d'observer les phénomènes d'analogies entre codes, sur les plans syntaxiques et sémantiques.

1 Introduction

L'analyse des programmes informatiques a fait l'objet de recherches intensives, particulièrement au cours de la dernière décennie, grâce aux avancées majeures dans les domaines de l'apprentissage de représentations et de l'apprentissage profond. Ces méthodes trouvent leurs principales applications en ingénierie logicielle et en éducation, où elles aident les développeurs à améliorer la qualité de leur code et les enseignants à suivre les progrès des étudiants.

De nombreux modèles actuels pour le traitement des programmes informatiques sont inspirés par les modèles de langage naturel (Kanade et al., 2020; Feng et al., 2020; Wang et al., 2021; Cleuziou et Flouvat, 2021). Cependant, il n'est pas évident que les hypothèses faites sur le langage naturel, comme l'hypothèse distributionnelle du sens des mots (Harris, 1954), puissent être directement transposées aux langages de programmation. Cette hypothèse stipule que des mots de sens similaire apparaissent dans des contextes similaires, mais il n'est pas clair si cela s'applique également aux fragments de code.

La contribution de cet article est double. Premièrement, il propose une étude de l'hypothèse distributionnelle et son applicabilité aux langages de programmation. Deuxièmement, il présente un cadre dédié à l'évaluation des embeddings de programmes. Ce cadre permet d'évaluer la qualité des embeddings à partir de différentes perspectives, dont la visualisation des embeddings, la capacité à partitionner selon un critère externe, et la capture des analogies.

[1] Cet article est la version française et condensée d'un article accepté à la conférence ECAI 2024 (Martinet et al., 2024b)

L'hypothèse distributionnelle est-elle vraiment adaptée aux langages de programmation ?

2 Travaux connexes

Bon nombre de modèles, dans la littérature, sont issus du traitement automatique de la langue (TAL) et adaptés aux langages de programmation d'une façon ou d'une autre. Nous en avons recensé quatre catégories principales qui analysent une certaine distribution basée sur des contextes comme formulé dans l'hypothèse distributionnelle (Harris, 1954) : i) les modèles qui analysent la distribution d'éléments syntaxiques (généralement les tokens) directement dans le code source ; ii) les modèles qui analysent l'arbre de syntaxe abstraite (abstract syntax tree - AST) du programme (Alon et al., 2019; Cleuziou et Flouvat, 2021) ; iii) les modèles qui analysent une représentation du programme sous forme de graphe, flot de contrôle ou flot de données principalement (Ben-Nun et al., 2018). Pour les ASTs et les graphes, l'analyse concerne la distribution des noeuds, qui correspondent à des éléments d'une certaine granularité dans le code ; iv) les modèles qui analysent une trace de l'exécution du programme (Cleuziou et Flouvat, 2021). Ici, la représentation du programme va permettre de concentrer l'analyse sur une information spécifique liée au comportement du programme, comme l'évolution des variables ou la séquence des éléments syntaxiques rencontrés lors de l'exécution.

Ces différentes catégories peuvent ensuite être divisées en sous-catégories, en fonction de la granularité des éléments syntaxiques dont les modèles observent la distribution (généralement les tokens, les instructions, ou une combinaison des deux).

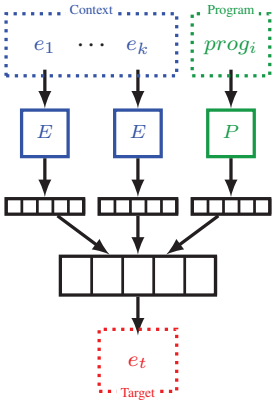
Par ailleurs, il existe peu d'évaluations qui mesurent directement la qualité des embeddings. L'écrasante majorité des articles évaluent plutôt les capacités d'apprentissage de leurs modèles dans leur globalité, sur des tâches de fine-tuning. Nous cherchons ici à mesurer l'information sémantique des programmes capturée par le modèle dans l'espace de représentation. Quelques évaluations vont dans ce sens, par exemple des explorations spatiales, et d'autres utilisent des modèles externes (Ben-Nun et al., 2018; Alon et al., 2019; Cleuziou et Flouvat, 2021).

3 Formalisation des hypothèses distributionnelles

Nous étudions six configurations, représentatives des hypothèses distributionnelles de l'état de l'art, caractérisées par : i) la granularité des *éléments* (ou fragments de code) considérés : tokens (t) ou instructions (i), et ii) la représentation du progamme (PR), définissant le *contexte d'occurence* dans lequel les *éléments* sont distribués : code source (S), trace d'exécution (T) ou arbre de syntaxe abstraite (A). Soit $\mathcal{H}_P^e$ l'hypothèse distributionnelle stipulant que *des éléments similaires de type* e *apparaissent dans des contextes similaires dans le format de représentation de programme* P. Le tableau de la figure 1 résume les 6 hypothèses étudiées.

4 Généralisation du modèle de langue

Notre étude s'appuie sur le modèle doc2vec, dérivé de word2vec (Mikolov et al., 2013), réseau de neurones très simple entraîné sur des séquences de mots (textes) de manière auto-supervisée. Sa variante CBOW s'attache à prédire un mot caché à partir de son contexte (mots précédents et suivants). A l'inverse, Skip-gram s'attache à prédire le contexte à partir du mot lui-même. Constatant que la distribution des mots varie d'un document à l'autre, Le et Mikolov (2014) ajoute au contexte la séquence dans laquelle le mot est observé, i.e. son document.



Hypothèse distributionnelle	**Source de données structurées** $\mathcal{S}$	**Éléments** $\mathcal{E}$	**Contexte** $\mathcal{C}$
$\mathcal{H}_S^t$	Code source S (séquences)	tokens t	éléments précédents et suivants dans la séquence
$\mathcal{H}_S^i$		instructions i	
$\mathcal{H}_T^t$	traces d'éxécution T (séquences)	tokens t	
$\mathcal{H}_T^i$		instructions i	
$\mathcal{H}_A^t$	ASTs (tokens)	noeuds de l'arbre	noeuds enfants
$\mathcal{H}_A^i$	ASTs (instructions)		

FIG. 1 – *À gauche : architecture Sec2vec, basée sur PV-DM (doc2vec), et entraînée à prédire un élément à partir de son contexte et du programme ; à droite : les 6 hypothèses distributionnelles étudiées.*

Ces modèles s'appuient de fait et uniquement sur l'hypothèse distributionnelle des mots. Doc2vec peut donc être vu comme un "outil d'extraction de sémantique" exploitant une hypothèse distributionnelle sur un jeu de données structurées. D'autres modèles plus puissants mais plus complexes introduisent des hypothèses supplémentaires tel le mécanisme d'attention.

Soit $\mathcal{E}$ un ensemble d'*éléments* (*e.g.* de mots), $\mathcal{S}$ un ensemble de données structurées sur $\mathcal{E}$ (*e.g.* du texte brut), et $\mathcal{C}$ une définition du *contexte* d'occurrences des éléments de $\mathcal{E}$ dans $\mathcal{S}$ (*e.g.* les k mots précédents et suivants). $(\mathcal{S}, \mathcal{E}, \mathcal{C})$2vec (sec2vec) est le modèle d'extraction de sémantique qui apprend les embeddings de $\mathcal{S}$ (et $\mathcal{E}$) à partir de la distribution des éléments de $\mathcal{E}$ dans les contextes $\mathcal{C}$ observés sur les sources $\mathcal{S}$ (hypothèse distributionnelle). Dans le cadre de notre étude nous instancions le modèle sec2vec pour les six hypothèses distributionnelles introduites en section 3, formant ainsi six triplets $(\mathcal{S}, \mathcal{E}, \mathcal{C})$ différents (tableau de la figure 1).

Dans le cas de code source ou de traces d'exécution, le contexte est constitué des éléments précédents et suivants, comme pour du texte. Pour les ASTs, il est constitué des noeuds enfants de l'élément considéré (qui est lui-même un noeud), du fait de la structure arborescente. L'extraction des tokens ou instructions depuis un code source ou une trace d'exécution est triviale, mais pour les AST le formalisme doit être adapté (Martinet et al., 2024a).

Nous notons $Sec2vec(\mathcal{H}_P^e)$ le modèle sec2vec appliqué aux éléments de type e dans la représentation P, utilisant le triplet correspondant $(\mathcal{S}, \mathcal{E}, \mathcal{C})$. De même nous notons $Sec2vec(\mathcal{H}^e)$ les modèles appliqués aux éléments e, et $Sec2vec(\mathcal{H}_P)$ ceux reposant sur la représentation P.

Dans les ASTs reposant sur les tokens, seules les feuilles contiennent des tokens, ce qui introduit un biais de nature entre les noeuds internes et les feuilles, dont les embeddings ne peuvent être appris. Nous avons recours à une variante de doc2vec, que nous appelons PV-SG, car elle repose sur la variante Skip-gram de word2vec. Nous la désignons par $Sec2vec^*(\mathcal{H})$.

Dans cet article nous utiliserons par défaut la version PV-DM de sec2vec, et ne présenterons les résultats de PV-SG (notés $Sec2vec^*(\mathcal{H})$) que lorsqu'une différence notable de performance est observée. De plus amples résultats sont disponibles en ligne (Martinet et al., 2024a).

5 Framework d'évaluation

Cette section présente l'une des principales contributions de l'article : l'évaluation d'espaces d'embeddings. L'objectif est d'évaluer et de comparer les hypothèses distributionnelles via les embeddings de programmes associés. Pour cela, nous nous appuyons sur des informations externes (non utilisées pendant l'entraînement) et sur des analogies censées être capturées par le modèle. A notre connaissance, il y a un véritable manque à ce niveau dans le domaine.

Toutes les évaluations ont été appliquées au jeu de données d'éducation NC5690 (Cleuziou et Flouvat, 2021) ; mais des résultats similaires ont été obtenus sur d'autres jeux de données (d'éducation également) incluant des codes en python et java (Martinet et al., 2024a). NC5690 contient 5690 programmes d'étudiants, répartis en 66 exercices.

Chaque expérience a été réalisée en faisant une validation croisée avec 5 échantillons (les valeurs moyennes et écarts-types sont indiqués dans les tableaux et courbes).

5.1 Implémentation

Sec2vec s'appuie sur l'implémentation de doc2vec de Gensim. Les différentes représentations des programmes en entrée de sec2vec ont été générées grâce à la bibliothèque tree-sitter. Des exemples d'AST et de traces artificielles sont disponibles dans Martinet et al. (2024a). Les modèles, fonctions d'évaluation et jeu de données d'analogies sont accessibles sur GitHub [2].

Les traces artificielles, représentant des séquences d'éléments (tokens ou instructions), ont été générées en parcourant l'AST en profondeur, en excluant ou répétant aléatoirement certains sous-arbres (p.ex. boucles). Une comparaison entre les traces réelles et artificielles a montré des résultats suffisamment similaires pour continuer avec la trace artificielle.

Notre objectif est de capturer la sémantique des programmes en tant qu'algorithmes. Or, certains aspects du code source brouillent légèrement cette sémantique, notamment les éléments liés à la langue naturelle, comme les noms de variables ou les commentaires. Aussi avons nous anonymisé toutes les parties de langue naturelle potentiellement perturbatrices : noms de fonctions, classes, variables, commentaires, et constantes (voir Martinet et al. (2024a)).

5.2 Evaluations s'appuyant sur une information externe

Ces premières évaluations utilisent une information externe correspondant à des catégories sémantiques fournies avec les données. Pour NC5690, cela correspond aux exercices.

Nous produisons une première visualisation qualitative, appelée "évolution cartographique", afin d'observer comment les espaces d'embeddings se structurent au fil de l'apprentissage en fonction des hypothèses distributionnelles étudiées. Chaque ligne dans la figure 2 correspond à un modèle (un sec2vec sur une hypothèse distributionnelle), chaque colonne représente la durée de l'entraînement (de 1 à 500 epochs), et chaque cellule contient une projection 2D, via t-SNE (Van der Maaten et Hinton, 2008), des embeddings de programmes. Les couleurs représentent les différentes catégories fournies avec les données (les exercices dans NC5690).

Partant d'une distribution plutôt uniforme au départ, les embeddings se regroupent en clusters (i.e. exercices) avec le temps. Nous observons cependant des performances moindres pour

2. https ://github.com/martinett/ProgramEmbeddingsEvaluationFramework

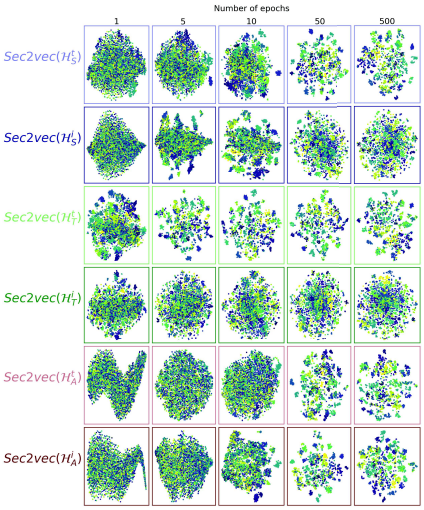
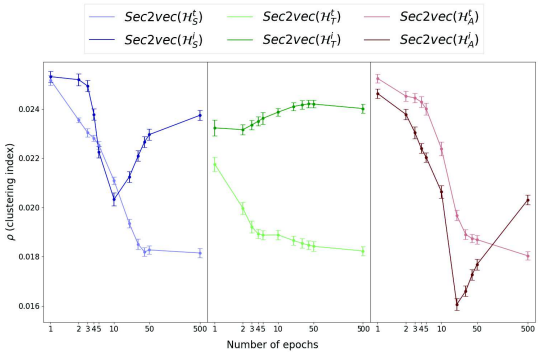


FIG. 2 – *Cartographie générée par t-SNE montrant l'évolution de l'espace d'embeddings pour différentes hypothèses distributionnelles (NC5690). Les couleurs représentent les exercices.*



$$\rho(\mathcal{D}, \Pi_{\mathcal{D}}) = \frac{\sum_{\pi \in \Pi_{\mathcal{D}}} \sum_{p_i, p_j \in \pi} dist(p_i, p_j)}{\sum_{p_i, p_j \in \mathcal{D}} dist(p_i, p_j)} \quad (1)$$

Avec $\Pi_{\mathcal{D}}$ l'ensemble des classes issues des données externes, $p_i \in \mathcal{D}$ un programme de $\mathcal{D}$, et $dist(p_i, p_j)$ la distance (typiquement euclidienne) entre deux embeddings.

FIG. 3 – *Evolution de l'indice de clustering ρ des exercices (le facteur externe) pour les différentes hypothèses sur le code source (bleu), les traces (vert) et les ASTs (marron).*

certains modèles basés sur les instructions ($Sec2vec(\mathcal{H}^i_S)$ et $Sec2vec(\mathcal{H}^i_T)$), mais pas pour celui basé sur l'AST ($Sec2vec(\mathcal{H}^i_A)$), qui semble mieux exploiter la structure du code. De plus, les modèles basés sur les traces d'exécution ($Sec2vec(\mathcal{H}^t_T)$ et $Sec2vec(\mathcal{H}^i_T)$) convergent plus rapidement car ils effectuent un nombre de prédictions par époque plus élevé.

Une évaluation quantitative de l'espace d'embeddings original, donc indépendante de la méthode de réduction de dimensions, complète ces visualisations. Pour cela, nous définissons un indice de clustering mesurant à quel point les embeddings de programmes sont séparés en clusters (i.e. exercices dans ces données). L'indice est calculé en fonction des distances entre programmes dans l'espace d'embedding (cf. formule de la figure 3), avec des valeurs comprises entre 0 (clusters bien séparés) et 1 (clusters qui se chevauchent).

La figure 3 confirme que les modèles basés sur les instructions ($Sec2vec(\mathcal{H}^i)$) atteignent rapidement leurs limites. Par ailleurs, même si les résultats sont au final similaires, les modèles basés sur les traces ($Sec2vec(\mathcal{H}_T)$) se stabilisent en 4 ou 5 époques, contre 40 à 50 pour ceux basés sur le code source ($Sec2vec(\mathcal{H}_S)$), confirmant ainsi leur rapidité de convergence.

Ces deux évaluations mettent donc en avant l'intérêt de considérer les tokens (plutôt que les instructions) pour extraire la sémantique des programmes. De plus, les traces d'exécution et les ASTs n'apportent pas de sémantique supplémentaire par rapport au code source, mais accélèrent artificiellement l'apprentissage des embeddings.

L'hypothèse distributionnelle est-elle vraiment adaptée aux langages de programmation ?

5.3 Evaluation basée sur les analogies de programmes

Certains jeux de données ne contiennent pas le type d'information que nécessitent les évaluations précédentes. Et même pour les jeux de données qui en contiennent, nous ne pourrions pas en comparer deux (ex. avec différents langages de programmation) puisque les résultats dépendent fortement du nombre de classes.

Nous proposons donc un nouveau type d'évaluation basé sur des analogies de programmes. L'évaluation par analogie est largement utilisée pour des modèles d'apprentissage de représentations de langue naturelle (Mikolov et al., 2013; Bakarov, 2018). Toutefois, il s'agit du premier travail adaptant cette approche aux programmes et mettant à disposition un jeu de données.

Notre évaluation par analogie fonctionne de manière similaire à celles utilisées pour les mots : deux paires de programmes liés (p_1, p_2) et (p_3, p_4) sont sélectionnées, leurs embeddings $vec(p_i)$ sont générés et le point $p' = vec(p_2) - vec(p_1) + vec(p_3)$ est calculé. Ensuite, il suffit de vérifier si p_4 est le plus proche voisin de p' parmi tous les programmes du jeu de données. Ces opérations permettent de vérifier si l'espace d'embeddings est suffisamment cohérent pour retrouver le quatrième programme de chaque analogie.

TAB. 1 – *Exemples d'analogies de programmes. Chaque ligne contient deux paires de programmes* (p_1, p_2) *et* (p_3, p_4)*, et nous voulons que* $\overrightarrow{p_1 p_2} \sim \overrightarrow{p_3 p_4}$.

	p1	**p2**	**p3**	**p4**
Syntaxique	*for elem → for index*		*for elem → for index*	

	def sum_list(l) :
	  res = 0
	  for elem in l :
	  res += elem
	  return res
		def sum_list(l) :
		  res = 0
		  for i in range(len(l)) :
		  res += l[i]
		  return res
			def display_list(l) :
			  for elem in l :
			  print(elem)
				def display_list(l) :
				  for i in range(len(l)) :
				  print(l[i])

Sémantique	*addition → multiplication*		*addition → multiplication*	

	def incr(x) :
	  return x + 1
		def double(x) :
		  return x * 2
			def incr_list(l) :
			  return [e+1 for e in l]
				def double_of_list(l) :
				  return [e*2 for e in l]

Nous définissons deux types d'analogies de programmes (cf. exemples du tableau 1). Les analogies syntaxiques relient des programmes équivalents sémantiquement (mêmes résultats) mais ayant des implémentations différentes. Les analogies sémantiques relient des programmes sémantiquement différents (résultats différents) mais présentant des similitudes syntaxiques.

Pour maximiser la couverture de l'espace d'embeddings, nous avons construit un jeu de données d'un millier d'analogies de programmes, dont 67% sont syntaxiques et 33% sont sémantiques. Nous avons ensuite calculé le ratio d'analogies retrouvées pour chaque modèle (i.e. les analogies pour lesquelles p_4 est le plus proche voisin du vecteur p').

Le tableau 2 montre que les modèles basés sur les instructions ne sont pas bons pour retrouver ces analogies, et ceux basés sur les tokens sont performants pour retrouver des analogies sémantiques, mais moins pour les analogies syntaxiques. Le modèle $Sec2vec(\mathcal{H}_A^t)$ n'apparaît pas compétitif, et la figure 4 (gauche) nous montre que les ASTs basés sur les tokens sont mieux exploités par l'architecture PV-SG (en pointillés) que PV-DM.

Par ailleurs, nous avons comparé l'évolution des performances des modèles en fonction du nombre d'epochs et d'iterations. Comme le montre la figure 4 (gauche), $Sec2vec(\mathcal{H}_T^t)$ est le premier à se stabiliser (vers 10 epochs), alors que $Sec2vec(\mathcal{H}_S^t)$ se stabilise vers 100 epochs, ce qui correspond au décalage déjà observé dans les évaluations s'appuyant sur une informa-

	Analogies syntaxiques	Analogies sémantiques
Source		
$Sec2vec(\mathcal{H}_S^t)$	**0.431** $\pm$0.011	**0.961** $\pm$0.005
$Sec2vec(\mathcal{H}_S^i)$	0.136 $\pm$0.015	0.001 $\pm$0.002
Trace		
$Sec2vec(\mathcal{H}_T^t)$	0.395 $\pm$0.011	0.955 $\pm$0.007
$Sec2vec(\mathcal{H}_T^i)$	0.124 $\pm$0.020	0.0 $\pm$0.0
AST		
$Sec2vec(\mathcal{H}_A^t)$	0.328 $\pm$0.008	0.551 $\pm$0.043
$Sec2vec(\mathcal{H}_A^i)$	0.141 $\pm$0.023	0.008 $\pm$0.006

TAB. 2 – *Evaluation des analogies par type après 500 epochs sur NC5690*

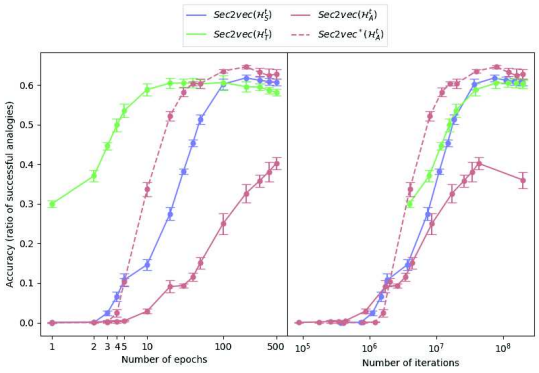


FIG. 4 – *Evaluation des analogies sur NC5690 en fonction du nombre d'epochs (g.) et du nombre d'itérations (dr.). A noter qu'une epoch correspond à autant d'itérations qu'il y a d'exemples en entrée.*

tion externe. En effet, les traces sont en réalité constituées de 10 traces pour mieux couvrir les possibilités d'exécution, et engendrent donc près de 10 fois plus d'exemples (et donc d'itérations) par epoch lors de l'apprentissage. Ce décalage disparaît lorsque nous affichons les mêmes résultats en fonction du nombre d'itérations (figure 4 droite).

6 Conclusion

A travers cet article nous avons souhaité contribuer à l'analyse et à la compréhension des modèles de langage appliqués au code informatique. Nous avons formulé plusieurs hypothèses distributionnelles pour les languages de programmation, considérant deux granularités d'éléments : tokens et instructions ; et trois contextes de représentation : code source, trace d'exécution et arbres de syntaxe abstraite (AST). Nous nous sommes appuyés sur un modèle simple, doc2vec, pour étudier l'impact de ces différentes hypothèses. Les expérimentations ont été menées sur des codes python et java, les deux langages montrant des comportements similaires.

Plusieurs méthodes d'évaluations ont été proposées, ainsi qu'un jeu de plus de milles analogies syntaxiques et sémantiques entre programmes, que nous prévoyons d'étendre pour couvrir plus largement l'espace d'embedding. Nous avons observé que travailler au niveau des instructions ne permettait pas de capturer la sémantique des programmes, et que les différentes hypothèses reposant sur des tokens se montraient globalement équivalentes.

Dans nos travaux futurs, nous prévoyons d'approfondir l'usage des AST, pour d'avantage exploiter la structure des programmes. Un passage des AST aux graphes est aussi envisagé. Enfin, nous souhaitons analyser des modèles à l'état de l'art, plus profonds que doc2vec, en amont de leur affinage à un problème particulier. Le domaine abordé dans cet article nous semble encore largement inexploré et riche de perspectives.

Références

Alon, U., M. Zilberstein, O. Levy, et E. Yahav (2019). code2vec : Learning distributed representations of code. *Proceedings of the ACM on Programming Languages 3*(POPL), 1–29.

Bakarov, A. (2018). A survey of word embeddings evaluation methods. *arXiv :1801.09536*.

Ben-Nun, T., A. S. Jakobovits, et T. Hoefler (2018). Neural code comprehension : A learnable representation of code semantics. *Advances in neural information processing systems 31*.

Cleuziou, G. et F. Flouvat (2021). Learning student program embeddings using abstract execution traces. In *14th International Conference on Educational Data Mining*, pp. 252–262.

Feng, Z., D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. (2020). Codebert : A pre-trained model for programming and natural languages. *arXiv :2002.08155*.

Harris, Z. S. (1954). Distributional structure. *WORD 10*(2-3), 146–162, doi: 10.1080/00437956.1954.11659520.

Kanade, A., P. Maniatis, G. Balakrishnan, et K. Shi (2020). Learning and evaluating contextual embedding of source code. In *International conference on machine learning*, pp. 5110–5121. PMLR.

Le, Q. et T. Mikolov (2014). Distributed representations of sentences and documents. In *International conference on machine learning*, pp. 1188–1196. PMLR.

Martinet, T., G. Cleuziou, M. Exbrayat, et F. Flouvat (2024a). Appendix of the paper "From document to program embeddings : can distributional hypothesis really be used on programming languages ?".

Martinet, T., G. Cleuziou, M. Exbrayat, et F. Flouvat (2024b). From document to program embeddings : can distributional hypothesis really be used on programming languages ? In *ECAI 2024*, pp. 2138–2145. IOS Press.

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

Van der Maaten, L. et G. Hinton (2008). Visualizing data using t-sne. *Journal of machine learning research 9*(11).

Wang, Y., W. Wang, S. Joty, et S. C. Hoi (2021). Codet5 : Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv :2109.00859*.

Summary

Many deep learning models have been applied to programming languages, all of them relying on natural language models and their underlying distributional hypothesis, but never questionning the relevance of this latter. In this paper we thus explore wether this hypothesis still stands for programming languages. Several methods are used, which we apply on variants of a well-known, easy to understand and to adapt model of natural language processing : doc2vec. Among other contributions, we propose a set of short programs that allow the observation of both syntactic and semantic analogies.