Apprentissage d'embeddings de codes pour l'enseignement de la programmation : une approche fondée sur l'analyse des traces d'exécution

Guillaume Cleuziou, Frédéric Flouvat

Apprentissage d'embeddings de codes pour l'enseignement de la programmation : une approche fondée sur l'analyse des traces d'exécution

Guillaume Cleuziou*,**, Frédéric Flouvat*

* ISEA, Université de la Nouvelle-Calédonie, BP R4, 98851 Nouméa, Nouvelle-Calédonie
prenom.nom@unc.nc
**Université d'Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France
prenom.nom@univ-orleans.fr

Résumé. Améliorer l'efficacité pédagogique des plateformes d'entraînement à la programmation est une problématique en pleine effervescence qui nécessite de construire des représentations fines et exploitables des programmes d'apprenants. Cet article présente une nouvelle approche pour l'apprentissage d'embeddings de programmes. Partant de l'hypothèse que la fonctionnalité d'un programme, mais aussi son "style", peuvent être capturés par l'analyse des traces d'exécutions, la méthode $code2aes2vec$ procède en deux étapes. Une première étape génère des séquences d'exécutions abstraites (AES) à partir de tests unitaires et des arbres syntaxiques abstraits (AST) des programmes soumis. La méthode $doc2vec$ est ensuite utilisée pour apprendre des représentations vectorielles condensées (embeddings) des programmes à partir de ces AES. Cette contribution donne également lieu à l'exploitation et la mise à disposition de nouveaux jeux de données réelles. Une première évaluation réalisée sur ces données montre que les embeddings générés par $code2aes2vec$ semblent capturer efficacement la fonctionnalité et le style des programmes.

1 Introduction

L'apprentissage de la programmation passe de plus en plus par l'utilisation de plates-formes d'entraînement en ligne. Classiquement, les apprenants y soumettent leur(s) code(s) et la plate-forme leur retourne les éventuelles erreurs syntaxiques ou fonctionnelles sur la base de cas de tests définis par l'enseignant. L'exploitation des données de ces plates-formes ouvre des perspectives excitantes en matière de suivi et d'aide à l'apprentissage de la programmation. Elles peuvent être utilisées par exemple pour identifier des étudiants en situation de décrochage, pour cibler des mauvaises pratiques ou pour propager des retours de l'enseignant. Ces fonctionnalités permettraient d'offrir à l'apprenant plus d'autonomie dans son apprentissage, et à l'enseignant d'être plus réactif et efficace dans ses interventions. Toutefois, cette exploitation nécessite une analyse fine des programmes soumis. Ces plates-formes d'entraînement doivent aller au-delà d'une simple analyse syntaxique du script, et permettre de considérer la sémantique associée.

Apprentissage d'embeddings de codes pour l'enseignement de la programmation

La fouille de textes a suscité beaucoup d'intérêts ces dernières années. La représentation des textes sous forme de vecteurs de réels, encore appelés "embedding", a notamment été au coeur de nombreux travaux récemment. Ces représentations permettent de projeter (ou 'plonger') tout un vocabulaire dans un espace vectoriel de faible dimension. Par ailleurs, avoir une telle représentation des mots permet ensuite d'exploiter une grande diversité de méthodes existantes (réseaux de neurones, SVM, clustering, etc.). Un des défis à ce niveau est de pouvoir capturer dans ces représentations les relations sémantiques sous-jacentes (ex. similarités, analogies). Les travaux de Mikolov et al. (2013) basés sur l'utilisation de réseaux neuronaux ont notamment été précurseurs en la matière. Leur méthode $word2vec$ est une des plus référencées dans le domaine. Son principe est de s'appuyer sur le lien entre un mot et son contexte (les mots apparaissant avant et après). Pour cela, ils proposent quelques architectures simples, efficaces et non-supervisées, pour apprendre des embeddings de mots à partir d'un corpus de textes. Par exemple, l'architecture *CBOW* apprend à prédire chaque mot à partir de son contexte. Les résultats obtenus ont notamment mis en avant la capacité de l'approche à extraire des relations sémantiques complexes à partir de simples opérations sur les projections $v() : v("king") - v("man") + v("woman") \approx v("queen")$, et $v("Paris") - v("France") + v("Italie") \approx v("Rome")$.

La transposition de ces approches aux codes informatiques n'est pas directe. Le code a certaines spécificités qu'il convient d'intégrer pour espérer avoir des représentations aussi riches (Allamanis et al., 2018). À la différence du texte, le code est exécutable, et une petite modification peut avoir un impact important sur son exécution. Un programme peut également appeler d'autres programmes pouvant eux-mêmes faire appel à d'autres programmes. Le contexte d'utilisation d'une instruction est aussi particulièrement important pour en déduire son rôle. Enfin, à la différence des textes, les arbres syntaxiques de programmes sont généralement plus profonds et composés de sous-structures de répétitions (boucles). Les approches existantes pour construire des embeddings de programmes n'intègrent que partiellement ces spécificités. Elles exploitent indépendemment les instructions (Azcona et al., 2019), les entrées/sorties (Piech et al., 2015), une partie des traces d'exécution (Wang et al., 2018) ou l'arbre syntaxique abstrait (AST) (Alon et al., 2019). Elles se focalisent plus sur la fonction du programme que sur son style. De plus, la majorité de ces approches sont supervisées et construisent des embeddings pour une tâche spécifique (ex. prédire les erreurs, prédire les fonctionnalités, etc.).

Face à ces limites, nous proposons la méthode $code2aes2vec$, exploitant instructions, structure du code et traces d'exécution, afin de construire des embeddings plus fins. La première étape de cette méthode consiste à génèrer des séquences d'exécutions abstraites (AES) à partir de cas de tests et des AST des programmes. La deuxième étape utilise la méthode $doc2vec$ [1] (Le et Mikolov, 2014) pour apprendre des embeddings de programmes. Contrairement aux approches existantes, nous proposons donc une approche générique et non-supervisée qui apprend des embeddings de programmes en intégrant des éléments de fonctionnalités, de style et d'exécution. Cet aspect est primordial par rapport à notre application car il faut être capable de différencier des programmes répondant à un même exercice (i.e. implémentant les mêmes fonctionnalités) mais de façons différentes (en terme de stratégie ou d'efficacité). Notre approche est validée sur deux jeux de données réelles, contenant plusieurs milliers de programmes Python issus de plates-formes pédagogiques.

1. méthode dérivée de $word2vec$ et permettant d'apprendre un embedding de document à partir des mots le composant.

Pour résumer, les principales contributions de ce travail sont :

1. la proposition d'une nouvelle représentation d'un programme, appelée séquence d'exécution abstraite (AES), permettant de capturer plus de sémantique,
2. l'exploitation de ces représentations avec $doc2vec$ pour construire des embeddings de programmes de façon non-supervisée,
3. la mise à disposition de la communauté de deux jeux de données enrichis, issus de plates-formes d'entraînement à la programmation.

La section à venir détaille les travaux existants dans le domaine et précise l'originalité de notre approche par rapport à ceux-ci. La section 3 présente notre approche en deux étapes : la construction des séquences d'exécutions abstraites et l'apprentissage des embeddings à partir de celles-ci. La section 4 est consacrée aux évaluations qualitatives et quantitatives des représentations apprises avant de dresser les nombreuses perspectives de ce travail (section 5).

2 Travaux connexes

L'apprentissage de représentations à partir de programmes est au cœur de nombreux travaux dernièrement, dont les principales motivations applicatives concernent le développement logiciel (débogage, découverte d'API, etc.) et l'enseignement (apprentissage de la programmation). Deux types d'embeddings sont plus particulièrement étudiés : les embeddings des éléments composants un programme (mots, tokens, instructions, ou appels de fonctions) (Nguyen et al., 2017; DeFreez et al., 2018; Henkel et al., 2018) et les embeddings de programmes (Piech et al., 2015; Wang et al., 2018; Azcona et al., 2019). La suite de cette section se focalise sur ce deuxième type d'approches, une revue plus large des méthodes existantes pouvant être trouvée dans Allamanis et al. (2018).

Tout comme en fouille de textes, une première approche consiste à représenter un programme comme un sac de mots (Salton et al., 1975). Toutefois, comme présenté dans l'étude comparative de Azcona et al. (2019), cette approche ne donne pas de bons résultats, car elle capture mal la sémantique sous-jacente. Les approches basées sur des réseaux de neurones donnent en revanche de meilleurs résultats dans leurs expérimentations. Piech et al. (2015) proposent une méthode construisant des embeddings de programmes d'étudiants qu'ils utilisent notamment pour propager automatiquement les remarques des enseignants. L'espace d'embeddings est construit à partir d'un réseau de neurones entraîné pour prédire la sortie d'un programme en fonction de son entrée. Il capture ainsi l'aspect fonctionnel du code. Pour tenter d'appréhender le style du programme, les auteurs utilisent un réseau de neurones récursif basé sur l'AST de chaque programme. Contrairement aux autres approches, les embeddings générés sont des matrices, et non des vecteurs, ce qui limite leur exploitation comme entrées d'algorithmes d'apprentissage. Les auteurs n'intègrent pas aussi les variables définies par les apprenants. Les résultats obtenus montrent que ces représentations capturent relativement bien les aspects fonctionnels du code mais plus difficilement le style. Dans Wang et al. (2018), les auteurs mettent en avant la limite des approches basées sur la syntaxe pour capturer la sémantique d'un programme. Face à cela, ils proposent de considérer à la place la trace issue de l'exécution du code, et plus particulièrement les valeurs des variables. Différentes représentations sont proposées et utilisées pour entraîner un réseau de neurones récurrent dont l'objectif est de prédire les erreurs faites par des étudiants dans un cours de programmation. L'embedding

des programmes est issu d'une des couches de ce réseau de neurones. Les auteurs mettent plus particulièrement en avant une représentation considérant la trace de chaque variable indépendamment et intégrant les dépendances entre variables dans la structure du réseau de neurones. En plus d'obtenir un embedding spécifique à cette tâche, cette solution nécessite donc de redéfinir l'architecture du réseau de neurones, et de refaire son entraînement, pour chaque exercice. Alon et al. (2019) proposent un réseau de neurones permettant de prédire le nom d'une méthode (i.e. sa fonction) à partir de son code. Pour cela, le programme est d'abord décomposé en une collection de chemins (d'une feuille à une autre) dans l'AST. Seuls les chemins les plus fréquents du jeu de données sont utilisés comme descripteurs. Les paramètres du réseau de neurones correspondent pour partie aux embeddings finaux et pour une autre partie à une pondération censée quantifier l'importance de chaque chemin (descripteur) pour la tâche de prédiction. Comme évoqué par les auteurs, cette approche nécessite une grande quantité de programmes en entrée. Par ailleurs, il n'est pas possible de prédire la fonction (et l'embedding) d'un programme dont les chemins n'apparaissent pas dans le jeu d'entraînement. Les embeddings produits capturent des informations et relations sémantiques sur la fonction du code, mais ignorent les différences de style.

3 Description de l'approche

Deux grandes stratégies d'apprentissage d'embeddings de programmes ressortent de l'étude bibliographique précédente : par l'observation des résultats de l'exécution du programme (Piech et al., 2015; Wang et al., 2018) ou alors par l'analyse du script (Azcona et al., 2019) et/ou de son AST (Alon et al., 2019). Notre approche se positionne à l'intersection de ces deux stratégies et vise ainsi à profiter conjointement des descriptions fonctionnelles et syntaxiques des programmes pour en induire des embeddings pertinents.

Nous proposons ainsi la méthode $code2aes2vec$ qui procède en deux étapes :

1. l'étape $code2aes$ représente le cœur de l'approche : elle consiste à traduire un programme sous forme d'une séquence d'exécution abstraite, ou AES (*Abstract Execution Sequence*), correspondant aux chemins de l'AST empruntés par le programme lors de son exécution sur des cas de tests prédéfinis ;
2. l'étape $aes2vec$ exploite l'approche $doc2vec$ de Le et Mikolov (2014) pour construire les embeddings des AES (et donc des programmes associés) en entraînant un réseau de neurones à prédire chaque "mot" d'une AES à partir de son contexte (fenêtre de quelques "mots" précédents/suivants) et du programme associé.

3.1 $code2aes$: construction de séquences d'exécutions abstraites

La traduction d'un programme en une AES nécessite de fournir, en complément du programme lui-même, une collection de cas de tests sur lesquels le programme sera exécuté afin d'en exploiter les traces. Pour le domaine d'application qui nous anime, à savoir l'éducation, des cas de tests sont généralement intégrés aux plates-formes d'entraînement pour évaluer les contributions soumises. A fortiori, le fait de laisser libre la construction des cas de tests offre à l'enseignant une possibilité d'introduire des choix de vérifications et ainsi d'orienter l'interprétation des programmes de ses apprenants en fonction de ses propres choix pédagogiques.

Par exemple, sur un exercice de recherche d'une valeur dans un tableau, un enseignant souhaitant mettre l'accent sur l'efficacité des algorithmes, peut choisir d'intégrer quelques cas de tests pour lesquels la valeur recherchée apparaît tôt lors du parcours. Ces cas de tests permettront ainsi de distinguer deux programmes (valides) par des AES plus ou moins longues selon que le programme stoppe ou non la boucle de parcours dès l'apparition de la valeur recherchée.

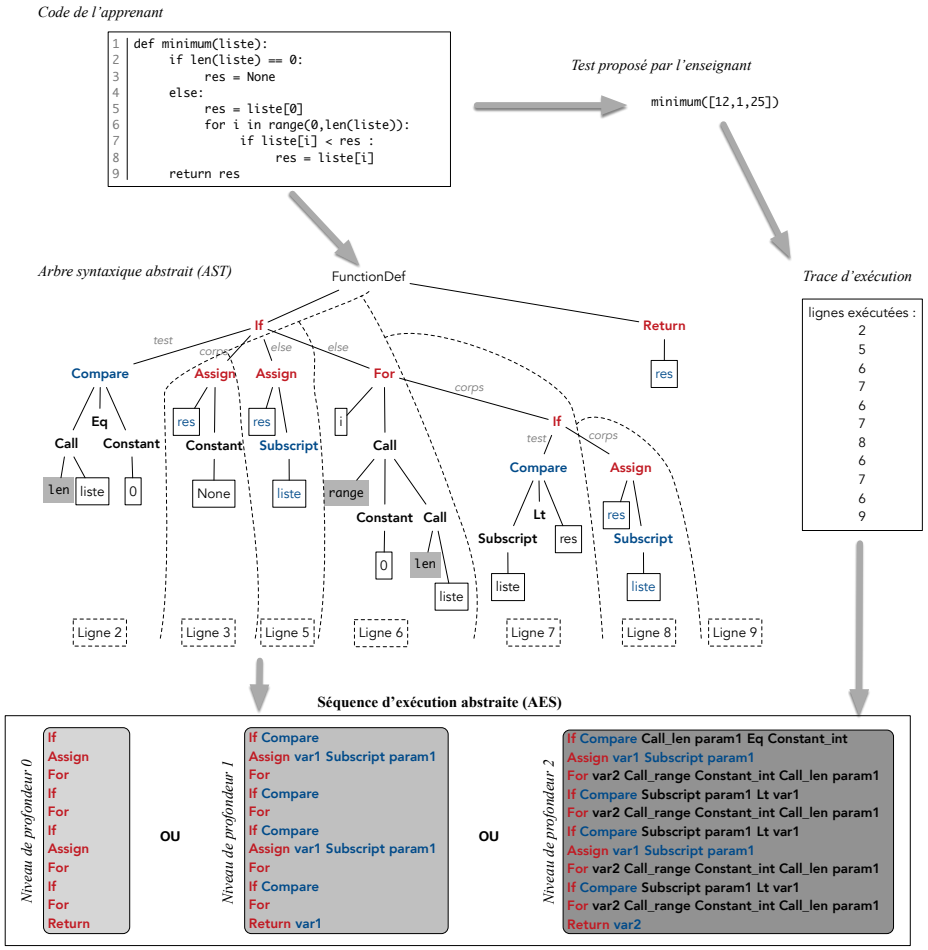

FIG. 1 – *Illustration du processus de construction des AES à partir d'un programme Python soumis en réponse à l'exercice 'rechercher le minimum dans une liste' et d'un cas de test.*

La Figure 1 illustre sur un exemple détaillé le processus de traduction d'un programme en une AES. Cet exemple considère en entrée le code soumis par un apprenant en réponse à l'exercice "*Écrire une fonction python qui retourne le minimum d'une liste passée en paramètre*". Ce programme donne lieu à un double traitement : d'une part l'AST est construit décrivant la structure syntaxique du programme en terme de structures de contrôle (`if-else`,

`for`, `while`), appels de fonctions (`call`), affectations (`assign`), etc ; d'autre part le code est exécuté sur un exemple (ici l'entrée `[12, 1, 25]`) et la trace est conservée, indiquant les lignes du programme successivement exécutées. L'AES est finalement construite par la mise en correspondance de ces deux niveaux d'information : syntaxique et fonctionnel. La séquence issue de la trace est traduite en une séquence de "mots" extraits des nœuds de l'AST.

Trois niveaux d'abstraction sont proposés selon la profondeur considérée dans l'AST :

— AES niveau 0 : chaque ligne du programme donne lieu à un unique mot correspondant au symbole de tête du sous-arbre associé dans l'AST (en rouge sur la figure),

— AES niveau 1 : chaque ligne du programme donne lieu à un ou plusieurs mots correspondant aux symboles de tête du sous-arbre associé et de ses principaux sous-arbres (en rouge+bleu sur la figure),

— AES niveau 2 : cette fois les sous-arbres sont parcourus en totalité de sorte que chaque ligne de la trace est traduite par une séquence de mots correspondant à tous les nœuds apparaîssant dans le sous-arbre associé (en rouge+bleu+noir sur la figure).

On observera pour les deux derniers niveaux que les noms des variables, des paramètres ainsi que les valeurs des constantes ont été normalisés de sorte à ne pas étendre artificiellement le "vocabulaire" considéré. Ainsi, la variable `res` est renommée `var1` et la variable `i` est renommée `var2`. L'AES finale, censée représenter le programme, est obtenue par la concaténation des AES partielles dérivées de chaque cas de test.

3.2 $aes2vec$: apprentissage des embeddings de programmes

La transposition de la méthode $doc2vec$ (Le et Mikolov, 2014) à l'apprentissage d'embeddings de programmes (plutôt que de documents) revient à considérer les programmes comme des documents textuels dont la séquence de mots est donnée par une AES. Comme pour les mots d'un texte, chaque "mot" d'une AES (nœud de l'AST) est porteur de sens ; le choix et l'ordre de ces mots au sein de l'AES permet d'appréhender le sens du programme, autrement dit ce que fait le programme (sa fonctionnalité) et sa manière d'opérer (son style).

De façon analogue à $doc2vec$, la méthode $aes2vec$ apprend des vecteurs représentant AES et mots par l'intermédiaire d'un réseau de neurones (Figure 2). Ce réseau est entraîné à prédire le mot courant w_i à partir de l'identifiant de l'AES, des mots précédents et des mots suivants. Pour cela, chaque AES est associée à une colonne d'une matrice D, et chaque mot à une colonne d'une matrice W. Ces vecteurs sont ensuite agrégés par somme, moyenne ou concaténation (il s'agit d'un hyperparamètre du réseau) pour prédire le mot courant en utilisant un classifieur multi-classes du type *softmax*. Les paramètres sont mis à jour par descente de gradient stochastique. Dans ce modèle d'apprentissage, chaque vecteur d'AES sera uniquement utilisé pour les prédictions des mots de cette AES, tandis que les vecteurs de mots sont communs à toutes les AES. La dimension des vecteurs (AES et mots) est fixée et correspond in fine à la dimension de l'espace de représentation (embeddings) souhaité. Une hypothèse forte de ce modèle est que l'AES elle-même contribue à la prédiction des mots qui la composent.

Une fois le modèle entraîné, la matrice D renfermera les embeddings des programmes utilisés pour cet entraînement (via leur AES). Le positionnement d'un nouveau programme dans cet espace d'embeddings consiste à inférer un nouveau vecteur colonne dans D à partir des mots de l'AES générée pour ce nouveau programme, les autres paramètres du modèle restant fixés (W ainsi que les paramètres du *softmax*).

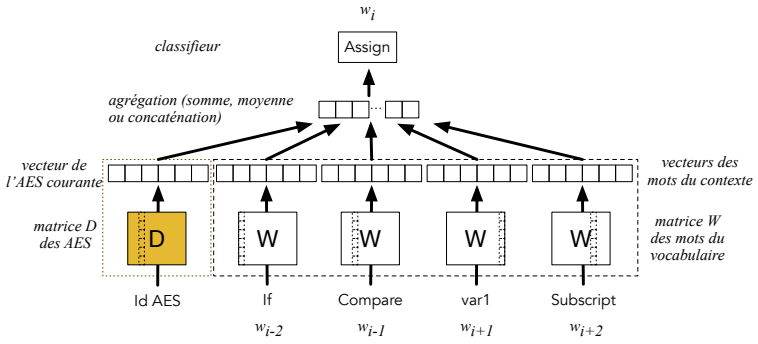

FIG. 2 – *Schéma du réseau de neurones $aes2vec$ (adapté de $doc2vec$) utilisé pour prédire un mot w_i à partir de l'identifiant de son AES d'origine et de son contexte (deux mots précédents et deux mots suivants).*

4 Évaluation de la méthode

Pour nos expérimentations, nous avons construit plusieurs jeux de données réelles, dont les caractéristiques sont décrites dans le tableau 1. Ils sont constitués de programmes Python soumis par les étudiants sur deux plates-formes d'entraînement dans le cadre de cours d'introduction à la programmation. Outre les programmes (documentés) de construction d'AES ($code2aes$), nous mettons également à disposition [2] ces trois corpus de programmes Python, les cas de tests associés ainsi que les AES construites sur chaque programme. L'ensemble des résultats présentés dans la suite de cette section peuvent ainsi être intégralement et aisément reproduits.

Jeu de données	NewCal.-1014	NewCal.-5690	Dublin-42487
Nb. programmes	1,014	5,690	42,487
Nb. programmes corrects	189	1,304	19,961
Nb. exercices	8	66	65
Nb. 'mots' AES-0	113,223 (20)	761,726 (44)	7,4 M (38)
Nb. 'mots' AES-1	226,682 (42)	1,7 M (57)	15,2 M (83)
Nb. 'mots' AES-2	690,019 (71)	3,9 M (113)	40,4 M (209)

TAB. 1 – *Présentation synthétique de trois jeux de données de programmes Python récoltés sur des plates-formes d'entraînement à la programmation. Le 'Nb. mots' indiqué correspond à la taille des corpus d'AES ; le nombre entre parenthèses indique la taille du 'vocabulaire'.*

— **NewCaledonia-5690** comporte les programmes réalisés en 2020 par une soixantaine d'étudiants de l'université de Nouvelle-Calédonie, sur une plate-forme d'entraînement à la programmation [3].

2. https ://github.com/GCleuziou/code2aes2vec.git
3. Plateforme développée et mise à disposition par le département informatique de l'IUT d'Orléans.

— **NewCaledonia-1014** correspond à une sous-partie de *NewCaledonia-5690* composée des contributions associées à 8 exercices sélectionnés pour leur diversité algorithmique (cf. tableau 2) et leur volumétrie équilibrée (100 à 150 programmes par exercice). Nous l'utiliserons comme jeu de données 'jouet' facilitant les analyses qualitatives.
— **Dublin-42487** comporte des programmes d'étudiants de l'université de Dublin, réalisés entre 2016 et 2019. Bien que le corpus originel, mis à disposition en juillet 2020 par Azcona et al. (2019), renferme près de 600,000 programmes (Python et Bash), nous en proposons ici un sous-ensemble enrichi semi-automatiquement de cas de tests (initialement non-fournis).

Exercice	Description
permutation	permuter les éléments d'une liste
minimum	rechercher le minimum dans une liste
compareChaines	comparer deux chaînes de caractères
quatrePlus100	renvoyer les quatre premières valeurs supérieures à 100 d'une liste en entrée
indiceOccurrence	renvoyer l'indice de la première occurrence d'un élément dans une liste
compareDates	comparer deux dates à partir de leur jour, mois et année
polynôme	calculer les racines d'un polynôme de degré 2
joursNuit	afficher une information sur le moment de la journée étant donnée une heure

TAB. 2 – *Énoncés des exercices du jeu de données NewCaledonia-1014.*

De manière générale, les données issues de l'éducation sont complexes. Les programmes peuvent contenir des erreurs, être de petite taille, ne pas répondre totalement aux fonctions prévues et être relativement redondants. Ces données ont des caractéristiques très différentes des jeux de données utilisés en développement logiciel.

Dans les expérimentations qui suivent, chaque jeu de données a été découpé en trois sous parties : entraînement (90%), validation (5%) et test (5%) ; l'ensemble de validation servant à sélectionner le meilleur modèle parmi ceux appris lors des différentes itérations ($aes2vec$). Sauf mention contraire, l'algorithme $aes2vec$ a été paramétré pour apprendre des embeddings de dimension 100, les fenêtres glissantes considèrent les 2 mots avant et les 2 mots après celui à prédire dans l'AES et l'apprentissage est réalisé sur 500 itérations.

Dans un premier temps, nous évaluons notre approche de manière qualitative sur le jeu de données *NewCaledonia-1014* constitué dans cet objectif. La Figure 3 (gauche) présente une visualisation des 912 programmes de l'ensemble d'entraînement, obtenue par projection non-linéaire via l'algorithme de réduction de dimensions t-SNE (LJPvd et Hinton, 2008). On observe que, bien que les embeddings soient appris de manière non-supervisée, la méthode $code2aes2vec$ permet d'apprendre, à partir d'un nombre relativement limité de données d'entraînement, un espace de représentation dans lequel les zones identifient des fonctionnalités distinctes des programmes. Ainsi, les vecteurs de programmes s'organisent assez naturellement en 8 groupes très fortement corrélés aux 8 exercices d'origine. De plus, l'organisation topologique de ces groupes respecte une logique sur l'algorithme inhérent aux programmes associés. Les exercices 'permutation' et 'minimum' sont proches dans l'espace et correspondent aux deux seuls exercices dont la consigne suggère un parcours complet d'une liste donnée en argument. Les exercices 'compareChaines', 'quatrePlus100' et 'indiceOccurrence', dans la partie supérieure de l'espace, nécessitent quant à eux de parcourir partiellement une liste.

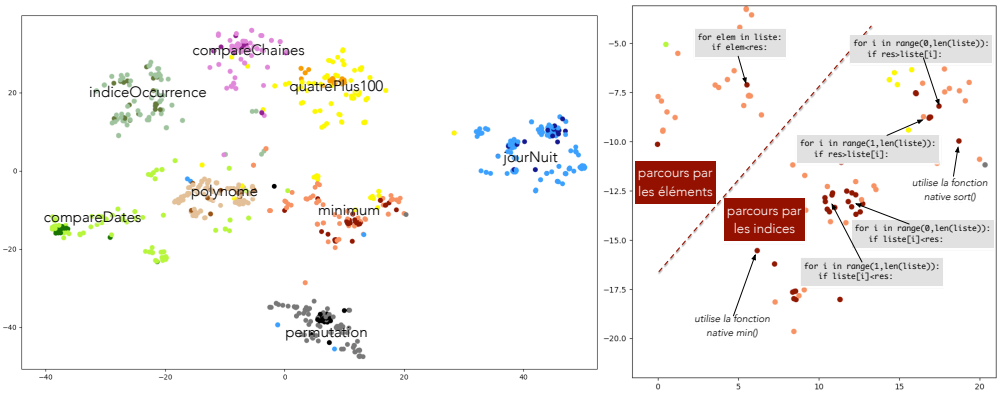

FIG. 3 – *Visualisation des embeddings de programmes obtenus par la méthode* $code2aes2vec$ *pour les 8 exercices du jeu de données NewCal.-1014. Les couleurs identifient les exercices, avec en clair les programmes incorrects et en foncé ceux corrects. La figure de gauche représente tous les embeddings et celle de droite détaille la zone associée à l'exercice 'minimum'.*

Enfin, les trois derniers exercices ne nécessitent pas de boucles et reposent uniquement sur l'utilisation d'instructions conditionnelles. L'exercice 'jourNuit' se distingue par l'usage attendu d'une fonction d'affichage (`print`) tandis que tous les autres exercices demandent de renvoyer une information (`return`). Cette caractéristique peut expliquer l"isolement' des programmes associés à cet exercice par rapport aux autres programmes.

La Figure 3 (droite) présente en détail l'espace des embeddings appris autour de l'exercice 'minimum'. Il est intéressant d'observer que dans cette projection, les styles de programmes sont clairement distingués, notamment les deux écritures d'une boucle `for` en Python (parcours d'une liste par ses indices vs. ses éléments). De façon très fine, les programmes sont également regroupés selon que leur boucle de parcours débute au premier élément de la liste (`range(0,...)`) où au deuxième (`range(1,...)`), après une initialisation du minimum au premier élément dans tous les cas. L'importance de l'ordre des mots pour la prédiction dans la méthode $aes2vec$ se manifeste ici par une distinction forte entre programmes selon l'ordre d'écriture de l'expression conditionnelle (`if liste[i]<res` vs. `if res>liste[i]`). Cette distinction peut sembler artificielle puisque ces deux expressions sont strictement équivalentes du point de vue de leur évaluation. Bien que équivalentes, la première écriture apparaît toutefois plus 'naturelle' que l'autre. Cette distinction a donc un intérêt d'un point de vue cognitif et donc pédagogique[4]. Nous attirons enfin l'attention du lecteur sur la mise à l'écart de certains programmes valides. Notamment l'existence d'un programme utilisant la fonction Python native `min`, ou encore celui utilisant une fonction de tri `sort`. Cette mise à l'écart revêt un intérêt applicatif crucial puisqu'elle laisse entrevoir la possibilité d'identifier des programmes valides qu'un enseignant souhaiterait pourtant rejeter ou du moins modérer car il les jugerait déviants par rapport à son objectif pédagogique. Plus généralement, cette analyse semble confirmer que les espaces d'embeddings appris par la méthode $code2aes2vec$ capturent correc-

4. Il serait aisé de 'gommer' ce phénomène en normalisant les expressions lors de l'étape $code2aes$; cette option pouvant être laissée à la discrétion de l'enseignant.

tement non-seulement la fonctionnalité des programmes mais également leur style. Leur qualité intrinsèque ouvre la voie à de nombreux usages pratiques susceptibles d'améliorer considérablement l'efficacité des plates-formes d'apprentissage de la programmation (détection de solutions atypiques, automatisation/propagation de commentaires fins, analyse des 'trajectoires' d'étudiants, analyse des typologies d'erreurs, etc.).

Dans un second temps, nous évaluons notre approche $code2aes2vec$ d'un point de vue quantitatif sur les trois jeux de données (Tableau 3). Nous considérons une tâche usuelle d'évaluation d'embeddings de programmes, à savoir la prédiction de sa fonctionnalité (i.e. l'exercice auquel il répond). Ainsi, pour chaque configuration envisagée (niveau d'AES, type d'agrégation), l'espace de représentation appris de façon **non-supervisée** avec la méthode $code2aes2vec$ sur les données d'entraînement, est utilisé par un simple classifieur 1-plus proche voisin (distance Euclidienne) pour prédire la classe des programmes de l'ensemble de test. La ligne *classification aléatoire* du Tableau 3 rend compte de la difficulté de la tâche en fonction du jeu de données. Nous reportons également les résultats obtenus par l'approche **supervisée** $code2vec$ (Alon et al., 2019) [5] exécutée avec les paramètres par défaut.

		NewCal.-1014	NewCal.-5690	Dublin-42487
classification aléatoire		*0.125*	*0.015*	*0.015*
$code2vec$ (Alon et al., 2019)		0.230	0.098	0.048
$code2aes2vec$ (somme)	AES-0	0.843	0.361	0.316
$code2aes2vec$ (concat.)	AES-0	0.980	0.505	0.460
$code2aes2vec$ (somme)	AES-1	0.980	0.677	0.486
$code2aes2vec$ (concat.)	AES-1	0.980	0.712	0.595
$code2aes2vec$ (somme)	AES-2	**1.0**	0.775	0.561
$code2aes2vec$ (concat.)	AES-2	**1.0**	**0.811**	**0.650**

TAB. 3 – *Évaluation quantitative et comparative des embeddings produits sur la tâche de reconnaissance de la fonctionnalité d'un programme (taux de bonne classification).*

On constate que le modèle $code2vec$ proposé récemment par Alon et al. (2019) ne peut être entraîné de manière satisfaisante sur aucun des trois jeux de données. Cela est dû notamment au grand nombre de paramètres mais aussi et surtout à la nécessité de disposer traditionnellement d'une quantité importante d'exemples pour l'entraînement des réseaux de neurones. Contrairement à la méthode $code2aes2vec$ qui s'appuie par exemple sur plusieurs millions d'entrées pour le corpus Dublin-42487 *via* l'usage des AES comme représentation intermédiaire, $code2vec$ ne dispose que des programmes comme entrées (soient 42,487).

Les résultats comparatifs obtenus avec $code2aes2vec$ pour différentes configurations confirment deux caractéristiques importantes de notre approche. D'une part une amélioration de la qualité des embeddings avec l'augmentation du niveau de détail des AES ; les AES de niveau 2 conduisant incontestablement aux meilleures représentations vectorielles de programmes. D'autre part, l'importance de préserver l'ordre des mots [6] du contexte dans la phase

5. Les autres méthodes présentées dans l'état de l'art n'ont pas pu être comparées faute d'implémentations opérationnelles disponibles.

6. Contrairement aux données textuelles, le vocabulaire des AES étant réduit, l'ordre d'apparition des mots est (au moins) aussi important que la nature même des mots.

d'agrégation de l'étape $aes2vec$; une agrégation par concaténation (concat.) conduisant systématiquement à de meilleurs modèles sur les tâches de prédiction non-triviales.

Enfin, d'autres comparaisons ont été réalisées, révélant des scores similaires dès que la dimension de l'espace dépasse 20, pour des fenêtres de mots évoluant de 1 à 10 (sous réserve d'une agrégation de type 'concaténation') ainsi qu'en augmentant la part de données de test. On observe également qu'il est possible d'améliorer encore ces scores en ayant recours à d'autres types de classifieurs (ex. SVM) comme alternatives au classifieur 1-plus proche voisin.

5 Conclusion et perspectives

Cet article étudie le problème de la construction de représentations vectorielles, ou embeddings, de programmes dans un contexte lié à l'éducation où la fonction est tout aussi importante que le style. Face à ce problème, nous proposons la méthode $code2aes2vec$ transformant le code en séquences d'exécutions abstraites puis en embeddings. Cette approche s'appuie sur une adaptation de la méthode $doc2vec$ aux programmes. Les évaluations menées valident la qualité des embeddings appris, capturant de manière fine la fonction et le style des programmes.

Les perspectives à ce travail sont nombreuses. Dans notre approche, tous les mots du programme ont le même poids dans la construction des embeddings. Ainsi, un programme correcte et un autre retournant une mauvaise valeur (ou déclenchant une erreur à la fin) pourront avoir des embeddings très proches, bien que fonctionnellement très différents. Cet aspect pourrait être intégré dans la construction de nos AES ou dans l'architecture du réseau de neurones utilisé pour générer des embeddings. Pour cela, il pourrait être aussi intéressant d'ajouter à nos AES les valeurs prises par les variables, à la manière de Wang et al. (2018) mais dans une approche multimodale générique. Une autre perspective serait de permettre à l'expert d'intégrer une partie de sa connaissance sur le langage. Comme discuté précédemment, certaines séquences d'instructions peuvent être équivalentes (p.ex. `if liste[i]<res:` vs. `if res>liste[i]:`). Des relations sémantiques entre mots peuvent aussi être connues (ex. la relation entre le `for` et le `while`). Ces connaissances permettraient de contraindre le réseau de neurones et de guider la construction des embeddings. L'implémentation actuellement disponible de notre approche permet de traiter des programmes en langage Python uniquement, ce qui limite les corpus exploitables. Il s'agirait alors d'étendre cette implémention pour traiter tout type de langage. Enfin, ces embeddings de programmes ouvrent un grand nombre de perspectives pour l'aide à l'enseignement. Par exemple, ils pourraient être utilisés pour identifier les typologies d'erreurs, les solutions alternatives, ou encore prédire les étudiants en situation de décrochage via l'analyse de leurs 'trajectoires'.

Références

Allamanis, M., E. T. Barr, P. Devanbu, et C. Sutton (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR) 51*(4), 1–37.

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.

Azcona, D., P. Arora, I.-H. Hsiao, et A. Smeaton (2019). user2code2vec : Embeddings for profiling students based on distributional representations of source code. In *Proceedings of the 9th International Conference on Learning Analytics & Knowledge*, pp. 86–95.

DeFreez, D., A. V. Thakur, et C. Rubio-González (2018). Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 423–433.

Henkel, J., S. K. Lahiri, B. Liblit, et T. Reps (2018). Code vectors : understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 163–174.

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

LJPvd, M. et G. Hinton (2008). Visualizing high-dimensional data using t-sne. *J Mach Learn Res 9*, 2579–2605.

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

Nguyen, T. D., A. T. Nguyen, H. D. Phan, et T. N. Nguyen (2017). Exploring api embedding for api usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 438–449. IEEE.

Piech, C., J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, et L. Guibas (2015). Learning program embeddings to propagate feedback on student code. In *Proceedings of the 32nd International Conference on Machine Learning*, ICML'15, pp. 1093–1102. JMLR.org.

Salton, G., A. Wong, et C.-S. Yang (1975). A vector space model for automatic indexing. *Communications of the ACM 18*(11), 613–620.

Wang, K., R. Singh, et Z. Su (2018). Dynamic neural program embeddings for program repair. In *International Conference on Learning Representations*.

Summary

Improving the pedagogical effectiveness of programming training platforms is a hot topic that requires the construction of fine and exploitable representations of learners' programs. This article presents a new approach for learning program embeddings. Starting from the hypothesis that the functionality of a program, but also its "style", can be captured by analyzing these traces of executions, the $code2aes2vec$ method proceeds in two steps. A first step generates abstract execution sequences (AES) from running tests and abstract syntax trees (AST) of the submitted programs. The $doc2vec$ method is then used to learn condensed vector representations (embeddings) of the programs from these AESs. This contribution also leads to the exploitation and diffusion of new real data sets. A first evaluation performed on these data sets shows that the embeddings generated by $code2aes2vec$ seem to efficiently capture the semantics and even the style of the programs.