Soutien à l'apprentissage de la programmation : conception et évaluation d'un indicateur sémantique

Julien Broisin, Clément Herouard

Soutien à l'apprentissage de la programmation : conception et évaluation d'un indicateur sémantique

Julien Broisin[1] and Clément Hérouard[2]

[1] Université Paul Sabatier, IRIT UMR 5505, 118 route de Narbonne, F-31062 Toulouse, France julien.broisin@irit.fr
[2] ENS, Rennes, France
clement@herouard.fr

Résumé. L'accompagnement des débutants dans l'apprentissage de la programmation connaît un véritable engouement depuis plus d'une décennie. La majorité des travaux se focalise sur une évaluation syntaxique du code pour étudier le comportement des apprenants et fournir des aides appropriées. Nous proposons dans cet article la conception d'un indicateur reflétant la proximité sémantique de deux codes source distincts, dans l'objectif d'exprimer la capacité d'un apprenant à résoudre un problème donné. Cet indicateur s'inspire d'une approche dédiée à la comparaison de chaînes de caractères, et applique une approche par apprentissage automatique pour déterminer certaines caractéristiques à partir d'un jeu de données obtenu durant un cours d'introduction à la programmation Shell suivi par 166 étudiants. Les premiers résultats sont encourageants : l'indicateur permet de classer correctement un code source d'un point de vue sémantique dans 58% des cas, et il est corrélé avec les évaluations sommatives réalisées par les enseignants.

Mots-clé : Apprentissage de la programmation · Analyse sémantique · Fouille de données éducatives · Apprentissage automatique.

Abstract. Supporting beginners in learning programming is of great interest for more than a decade. Most of the work focuses on a syntactic evaluation of the code produced by learners to study their coding behaviour and provide them with appropriate support. We propose in this article the design of an indicator reflecting the semantic proximity of two distinct source codes, in order to express the learner's capacity to solve a given problem. This indicator is based on an approach dedicated to strings comparison, and applies a machine learning approach to explore certain characteristics from a dataset gathered during an introductory Shell programming course involving 166 students. The first results are encouraging: the indicator makes it possible to correctly classify source code from a semantic point of view in 58% of cases, and is correlated with the summative evaluations carried out by teachers.

Keywords: Learning programming · Semantic analysis · Educational data mining · Machine learning.

1 Introduction

L'apprentissage de l'informatique est actuellement en plein essor. En France, la réforme du baccalauréat qui sera opérationnelle dès la prochaine rentrée scolaire prévoit une option Numérique et Sciences Informatiques en classes de première et terminale pour dispenser plus de 350 heures d'informatique à un élève de lycée. En Europe, notamment en Allemagne et au Royaume-Uni, des transformations éducatives profondes sont également initiées depuis 2016 pour favoriser le développement et l'apprentissage du numérique dans les écoles et préparer les apprenants à l'acquisition des compétences du 21ème siècle. Cette intégration de l'informatique dans les cursus de formation dès le plus jeune âge requière non seulement des enseignants préparés à cette nouvelle discipline, mais également des solutions technologiques pour accompagner les instructeurs et les apprenants dans leurs pratiques quotidiennes.

Dans cet objectif, la communauté de recherche en Environnements Informatiques pour l'Apprentissage Humain (EIAH) s'intéresse à la conception de systèmes dédiés à l'accompagnement de l'apprentissage de la programmation, comme en témoignent les approches d'analyse du comportement des apprenants [1, 2]. Une des approches reconnues dans la littérature consiste à proposer des systèmes qui analysent le code produit par les apprenants d'un point de vue syntaxique. Ces systèmes visent à fournir un soutien approprié et constructif aux apprenants en réalisant une évaluation syntaxique du code pour détecter les erreurs grammaticales et/ou éviter les erreurs de misconceptions [3]. Pourtant, un des résultats d'apprentissage primordiaux de la programmation concerne bien le développement de la capacité d'un individu à résoudre un problème [4]. Une approche pour atteindre cet objectif repose sur la conception de systèmes capables d'analyser un code source du point de vue sémantique, c'est-à-dire capable de refléter comment un problème a été résolu. Bien que certaines initiatives tentent d'aborder l'analyse sémantique de codes informatiques [5], les travaux sur ce sujet sont insuffisants et peu de solutions ont été proposées. Ainsi, des études supplémentaires sont nécessaires pour mieux comprendre le potentiel de cette approche lorsqu'il s'agit d'accompagner l'apprentissage de l'informatique.

Nous proposons alors dans cet article la conception d'un indicateur reflétant la proximité sémantique de deux codes source distincts, dans l'objectif de refléter la pertinence d'une production d'un apprenant au regard d'un problème donné. Les questions de recherche traitées dans l'étude sont donc les suivantes :
Question de recherche 1 : Comment concevoir un indicateur sémantique reflétant la capacité d'un apprenant à résoudre un problème ?
Question de recherche 2 : La qualité sémantique d'une production d'un apprenant est-elle corrélée avec sa performance académique ?

Pour répondre à la première question, nous adoptons une approche par comparaison d'arbres de syntaxe abstraite. Nous nous appuyons sur la distance d'édition formulée par Levenshtein, un algorithme qui a démontré son efficacité pour comparer des chaînes de caractères, et proposons une approche par apprentissage automatique pour déterminer certains facteurs nécessaires au calcul de la proximité sémantique. En ce qui concerne la seconde question de recherche,

nous menons un ensemble d'études statistiques sur des données collectées dans un contexte écologique d'apprentissage afin d'évaluer la précision de l'indicateur au regard de la perception humaine. En particulier, à partir de codes source produits par 166 étudiants, nous analysons la corrélation entre la valeur de notre indicateur calculée automatiquement pour l'ensemble des scripts d'une part, et les scores assignés manuellement par des enseignants d'autre part.

2 Approches d'analyse automatique de code source

Dans le contexte de l'apprentissage de la programmation, l'analyse automatique de code source est utilisée pour atteindre différents objectifs tels que l'amélioration des feedbacks fournis aux apprenants [6, 7], la prédiction de leur performance [8, 9], ou leur suivi pendant la tâche d'apprentissage [10].

Ces analyses automatiques sont réalisées par compilation du code [11], par recherche d'erreurs classiques au sein d'un code source [7], ou par l'écriture de test unitaires par les enseignants [12]. Ces différents travaux guident les apprenants dans l'écriture de programmes syntaxiquement corrects, mais ne permettent pas d'évaluer le code source à un niveau sémantique : l'évaluation syntaxique n'est pas suffisante pour rendre compte de la pertinence d'une production d'un apprenant vis-à-vis d'un problème posé par l'enseignant.

Dans la méta-review proposée par Ihantola et al. [13] qui ont étudié pas moins de 118 articles de recherche dans le domaine de la fouille données éducatives pour la programmation, le mot *semantics* est absent du papier. De même, une recherche des termes *semantic analysis programming* dans Google Scholar et Web of Science retourne bien un nombre important de résultats, mais aucun article ne traite réellement de l'analyse sémantique de code.

Les travaux que nous avons recensés qui se rapprochent d'une analyse sémantique sont ceux proposés par Bey et al. [5]. Afin de mettre en œuvre une notation automatisée du code des apprenants dans le cas des MOOC, les auteurs proposent de comparer le graphe de flot de contrôle correspondant au code produit par l'apprenant, avec un ensemble de graphes issus de solutions expertes qui résolvent le problème donné [14]. Si le graphe de l'apprenant est reconnu parmi la base de graphes experts, alors un score et un feedback sont retournés à l'apprenant ; s'il n'est pas reconnu, alors la production de l'apprenant est évaluée manuellement par un expert humain afin d'enrichir la base de graphes.

L'absence de travaux concernant l'analyse automatique d'un point de vue sémantique peut s'expliquer par le fait que la sémantique d'un programme n'est pas calculable. Nos travaux tentent de dépasser cette limite en proposant la conception d'un indicateur reflétant la pertinence d'un code source par rapport à un problème donné.

3 Spécification de l'indicateur sémantique

Notre objectif est de concevoir un indicateur capable d'évaluer la distance d'édition entre deux codes source du point de vue sémantique, et de satisfaire les con-

traintes suivantes : la tâche des enseignants doit se limiter à la production d'une solution unique aux problèmes posés, au contraire des méthodes par tests unitaires qui requièrent de leur part un effort important ; la distance doit diminuer au fur et à mesure que l'étudiant se rapproche de la solution, puisqu'un script incomplet peut être évalué même s'il ne répond pas entièrement au problème posé. Ainsi, si une solution à un problème est fournie par l'enseignant, la distance entre le code source correspondant et les productions des apprenants devrait fournir un nouvel éclairage sur leur capacité à résoudre le problème.

Le domaine du traitement automatique des langues a depuis longtemps étudié des problématiques liées à la correction orthographique où la sémantique est naturellement prise en compte [15]. Nous nous sommes inspirés de ces approches pour proposer la conception de notre indicateur, et en particulier de la distance d'édition entre deux chaînes de caractères, ou distance de Levenshtein [16]. Cette distance est définie par un coût calculé à partir du nombre minimum d'opérations (i.e., insertion/suppression d'un caractère, et substitution d'un caractère par un autre) nécessaires pour passer d'une chaîne à l'autre. Dans notre contexte, nous souhaitons identifier les opérations à effectuer pour passer d'un code source à un autre, et ainsi calculer une distance correspondant à la somme des coûts de chacune des opérations à réaliser. Dans le cas de la comparaison de chaînes de caractères, les coûts associés à chacune de ces opérations sont tous fixés à 1, à l'exception du coût de substitution d'un caractère par lui-même qui est nul. Dans le cas de la comparaison de codes source, les coûts de ces opérations ne sont pas tous égaux : la substitution de deux instructions visant les mêmes objectifs ne doit pas avoir le même coût que la substitution d'une commande par une autre ayant une fonctionnalité très différente. Le challenge de notre approche consiste donc à attribuer un coût à chacune des opérations selon l'importance des modifications à réaliser pour passer d'un code source à un autre.

La méthodologie mise en œuvre pour concevoir notre indicateur est alors la suivante : formalisation du code source sous la forme d'un arbre de syntaxe abstraite ; transformation de cet arbre en une chaîne d'instructions élémentaires ; identification des coûts d'insertion, de suppression et de substitution des instructions élémentaires ; spécification d'une méthode d'apprentissage automatique pour déterminer les valeurs de ces différents coûts ; évaluation des résultats.

3.1 Formalisation de code source

Nous avons adopté les arbres de syntaxe abstraite (Abstract Syntaxic Tree, AST) afin de proposer une description formelle des codes source. En effet, au contraire des arbres d'analyse, les arbres de syntaxe abstraite ne représentent pas les nœuds et les branches qui n'affectent pas la sémantique d'un programme.

Nous nous appuyons sur un parseur écrit en JavaScript qui lit un script et retourne l'arbre de syntaxe abstraite correspondant au format JSON ; un exemple de conversion d'un code source Bash en AST est donné par la Figure 1.

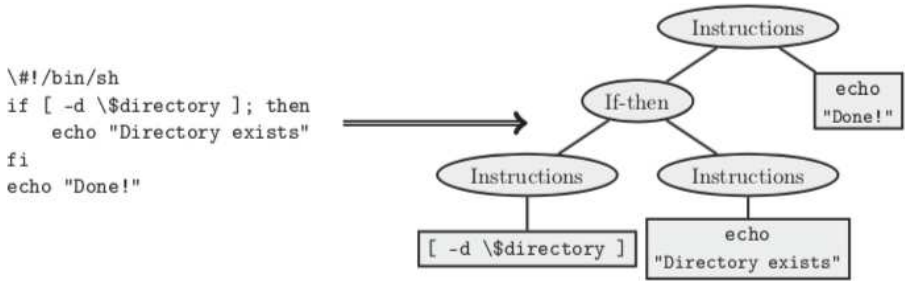

Fig. 1. Conversion d'un script en arbre de syntaxe abstraite.

3.2 Mise en œuvre de la distance d'édition

Ainsi, avec la représentation formelle proposée ci-dessus, la mesure de notre indicateur correspond à la distance d'édition entre deux arbres de syntaxe abstraite. Toutefois, afin d'adapter la distance de Levenshtein à notre contexte, nous souhaitons convertir l'AST généré à partir du code source en une chaîne de caractères ; du point de vue de la terminologie, nous nommons *tokens* les caractères résultant du processus de transformation de l'AST afin de ne pas les confondre avec ceux contenus dans le code source. Ainsi, nous allons effectuer un parcours en profondeur de l'AST pour créer la chaîne de tokens.

Production de la chaîne de tokens. Le langage Bash, objet de notre étude, propose 17 structures de contrôle correspondant chacune à un token différent. Un token peut par exemple être du type Command, du type Assignation pour les définitions de variables, ou du type While, Do ou Done pour signaler respectivement le début, le corps ou la fin de boucle. Notons que les tokens Command sont caractérisés par le nom de la commande et ses arguments. Un exemple de transformation d'un AST en chaîne de tokens est illustré par la Figure 2.

La distance entre deux chaînes de tokens est donc caractérisée par un certain nombre de coûts, ou paramètres, correspondant aux différentes opérations élémentaires d'insertion, de suppression et de substitution de chaque token. Les 17 tokens du langage Bash conduisent à la définition de 308 coûts : 17+17 coûts de création et de suppression d'un token, 16x17 coûts de substitution d'un token par un autre, 1 coût pour substituer un token Command en un autre dont le nom de la commande est différent, et enfin 1 coût pour substituer un token Command en un autre dont les noms de commande sont identiques et les arguments différents. Ce nombre de paramètres est trop élevé pour mettre en œuvre une méthode par apprentissage efficace. En effet, un nombre trop important de paramètres risque de diminuer la densité des tests et ainsi rendre la recherche de "bonnes" valeurs pour les paramètres plus difficile.

Réduction du nombre de paramètres. Une première opération pour réduire le nombre de paramètres à entraîner consiste à ignorer les tokens Until, Subshell

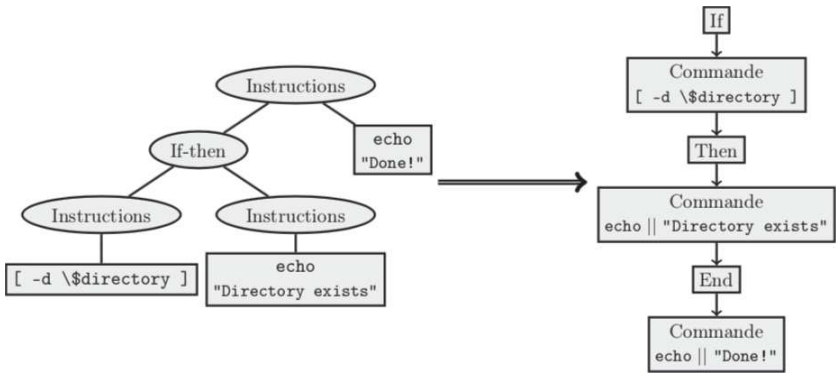

Fig. 2. Conversion d'un arbre de syntaxe abstraite en chaîne de tokens.

et Pipeline car les structures de contrôle correspondantes n'apparaissent pas dans le jeu de données de cette étude (cf. Section 4). Nous proposons également une symétrisation du problème en affectant des coûts égaux aux opérations d'insertion et de suppression d'un token ; de même, la substitution du token A par le token B a un coût égal à la substitution de B par A. Cette symétrisation rend la distance entre les scripts S_1 et S_2 égale à la distance entre S_2 et S_1, et permet de réduire le nombre de paramètres à 107. Enfin, nous posons une hypothèse plus lourde en considérant que seuls certains tokens peuvent se substituer à d'autres. Ainsi, seuls les groupes de tokens suivants peuvent se substituer entre eux : {Command, Assignation}, {If, Case}, {Then, Else, CaseItem}, et {For, While}. En effet, il est envisageable qu'un script contienne un Case là où un autre script emploie un If, par contre il est peu probable qu'une substitution d'un For par un If soit fréquente, ces deux instructions visant des objectifs très différents.

Ces simplifications diminuent considérablement le nombre de coûts puisque 23 paramètres doivent désormais être calculés. Notons θ le vecteur contenant ces 23 coûts. Notre indicateur, noté $d(S, C, \theta)$, est alors défini par la distance d'édition, sous les 23 paramètres θ, entre la chaîne de tokens d'un script S et celle d'un script C. Afin d'être en mesure de spécifier le critère d'apprentissage conduisant aux valeurs optimales de θ, nous présentons le jeu de données issu d'une situation d'apprentissage réelle qui a été utilisé pour réaliser cette étude.

4 Jeu de données

Le jeu de données fut obtenu dans un département Informatique d'un Institut Universitaire Technologique (IUT) durant un cours d'introduction à la programmation Shell suivi par 166 apprenants. Les étudiants découvrent des commandes courantes telles que echo, ls ou cat, la gestion des variables, ainsi que les structures du langage (e.g., for, while, if, case). Les étudiants mettent ensuite en

pratique ces notions lors de séances de travaux pratiques durant lesquelles ils produisent des scripts Bash pour tenter de solutionner une série de problèmes.

Le jeu de données comprend les productions des étudiants durant cinq séances de travaux pratiques d'une heure et demie chacune. A chaque enregistrement d'un script Bash par un étudiant, une copie de ce dernier accompagnée de son horodatage et de l'identifiant de l'étudiant est sauvegardée dans un répertoire afin d'enrichir le jeu de données. Ainsi, notre échantillon comprend 19232 scripts. Toutefois, des erreurs de nomenclature ou de numérotation des fichiers ont rendu 18% de ces scripts inexploitables : le jeu de données est constitué de 15794 scripts.

Cet échantillon comprend, pour un même exercice, plusieurs scripts produits par un même élève. Or les méthodes d'apprentissage sont efficaces lorsqu'elles s'appuient sur des données indépendantes les unes des autres. Alors nous ne considérons, par exercice et par étudiant, que le dernier script produit par l'apprenant car nous supposons qu'il est le plus abouti. Ce choix permet d'obtenir un jeu de données indépendantes, mais réduit l'échantillon à 2540 scripts.

Enfin, nous souhaitons classifier manuellement chaque script de l'échantillon dans deux catégories distinctes (i.e., *Correct* et *Incorrect*) exprimant leur justesse sémantique. Certains exercices dispensés lors de cette expérimentation ne permettent pas de concevoir une solution répondant totalement au problème posé, car certains problèmes concernent par exemple la compréhension de l'exécution d'un script fourni dans l'énoncé. Après cette opération de classification, l'échantillon final comprend 733 scripts répartis sur 11 exercices différents ; 397 scripts sont sémantiquement corrects, les autres étant sémantiquement incorrects.

Ce jeu de données a été divisé aléatoirement en un jeu d'apprentissage dont l'objectif est d'identifier les valeurs optimales du paramètre θ, et un jeu de test permettant d'évaluer la pertinence des résultats obtenus lors de la phase d'apprentissage. Ces deux échantillons contiennent le même nombre de scripts pour chaque exercice, et la proportion de scripts sémantiquement corrects et incorrects est préservée. De plus, notons qu'une solution a été fournie par un enseignant pour chacun des exercices.

5 Méthode d'apprentissage et résultats

Pour entraîner les bonnes valeurs des 23 coûts du paramètre θ, nous définissons le critère d'apprentissage $Score(\theta)$ à minimiser tel que :

$$Score(\theta) = \frac{\sum_{(S,C)\in Correct} \sqrt{d(S,C,\theta)}}{\sum_{(S,C)\in Incorrect} \sqrt{d(S,C,\theta)}} \quad (1)$$

où *Correct* (resp. *Incorrect*) est l'ensemble des paires composées des scripts corrects (resp. incorrects) du jeu d'apprentissage et des corrections associées.

La fonction *Score* est faible quand les scripts corrects sont associés à de faibles distances, et les scripts incorrects à de fortes distances. Nous sommons les racines carrées des distances pour réduire l'influence des fortes valeurs.

Nous pouvons remarquer une propriété de la fonction $d : \forall \lambda > 0, d(S, C, \lambda.\theta) = \lambda.d(S, C, \theta)$. Alors il est évident que $Score(\lambda\theta) = Score(\theta)$. Cela signifie que la fonction $Score$ est constante sur toutes les demies-droites partant de 0, et que l'exploration des différents coûts du paramètre θ, notés θ_i tels que $0.01 < \theta_i \leq 1$, est suffisante pour trouver les valeurs de θ_i minimisant $Score$.

Le jeu de données d'apprentissage a servi à initialiser l'algorithme de minimisation de descente de gradient. Le vecteur minimisant $Score$, noté $\overline{\theta}$, admet des valeurs de 0.01 (i.e., le minimum que nous avons fixé) presque partout sauf pour trois paramètres :

1. la création ou suppression d'un token Command, dont le coût est de 1 ;
2. la création ou suppression d'un token If, dont le coût est de 0.112 ;
3. la création ou suppression d'un token Function, dont le coût est de 0.022.

Nous disposons, à cette étape, du paramètre $\overline{\theta}$ qui renferme les valeurs des 23 coûts qui ont été obtenues par notre méthode d'apprentissage. Les tests menés dans la section suivante avec le paramètre $\overline{\theta}$ sur le jeu de données de test visent un double objectif : étudier la capacité de notre indicateur à différencier un script sémantiquement correct d'un script incorrect ; étudier la corrélation entre notre indicateur et des notes attribuées manuellement aux scripts par des enseignants.

6 Validation

6.1 Étude de la prédiction

Une première approche pour évaluer la pertinence de notre indicateur sous le paramètre θ consiste à évaluer sa capacité à classer automatiquement un script comme étant correct ou incorrect d'un point de vue sémantique. Dans une première étape, nous calculons la distance $d(S, C, \theta)$ entre chaque script S du jeu de données de test et la correction C du problème correspondant, afin d'obtenir une valeur de notre indicateur pour chaque script correct et incorrect.

L'étape suivante consiste à évaluer le modèle de prédiction résultant des différentes valeurs de l'indicateur. Nous utilisons pour cela la mesure ROC qui est très répandue dans le domaine des statistiques pour observer la performance d'un classifieur binaire. Lorsque le classifieur calcule une mesure m qui est comparée à un seuil σ pour prédire la classe, l'idée de la courbe ROC est de faire varier σ de 1 à 0 et, pour chaque valeur de σ, reporter dans un graphique le taux de faux positifs en abscisse, et le taux de vrais positifs en ordonnée. L'aire sous la courbe (AUC) indique alors la probabilité pour que le classifieur place un positif devant un négatif. Un classifieur qui ne se trompe jamais présente une AUC égale à 1, alors qu'un classifieur aléatoire est caractérisé par une AUC égale à 0.5.

Dans notre étude, si $d(S, C, \overline{\theta}) < \sigma$, alors le script est classé dans la catégorie *Correct* ; sinon le script est classé dans la catégorie *Incorrect*. La courbe ROC de la Figure 3 illustre le ratio de vrais positifs (i.e., fraction des scripts corrects qui sont effectivement détectés par le classifieur) en fonction du ratio de faux positifs (i.e., fraction des scripts incorrects qui sont mal classés par le classifieur). La

courbe ROC est sensiblement au-dessus de la bissectrice, l'aire sous cette courbe est égale à 0.585. Notre classifieur est donc légèrement meilleur qu'un classifieur aléatoire, mais ses performances ne sont pas très élevées ; nous émettons des hypothèses pour expliquer ces résultats dans la Section 7.

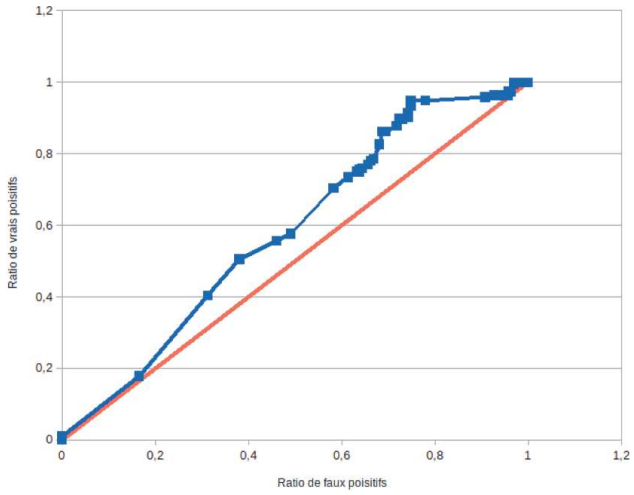

Fig. 3. Courbe ROC de notre classifieur (en bleu) et bissectrice du hasard (en rouge).

6.2 Étude de la corrélation avec une notation humaine

L'un des objectifs de notre indicateur est de refléter la progression de l'apprenant vers la solution au problème, c'est-à-dire que sa valeur est supposée diminuer au fur et à mesure qu'un script tend vers la solution. Nous étudions alors la corrélation entre la valeur de notre indicateur sous le paramètre $\overline{\theta}$ pour un script donné et la correction associée, et la note attribuée à ce script par un humain.

L'étude de cette corrélation est réalisée à l'aide d'un second jeu de données obtenu après l'examen terminal du cours décrit dans la Section 4. Parmi les 166 étudiants inscrits à ce cours, 163 ont participé à l'examen terminal qui consistait à produire un script répondant à un problème. Un enseignant a ensuite attribué un score sur 10 à chacun des 163 scripts. Après avoir éliminé les scripts grammaticalement incorrects, 105 scripts constituent ce second jeu de données.

La Figure 4 donne, pour chaque script, la valeur de l'indicateur en fonction de la note attribuée par l'enseignant. Ce graphe montre une corrélation négative : la corrélation de Kendall (non-linéaire) a une valeur égale à -0.319 (valeur p inférieure à $2.7 \cdot 10^{-6}$), la corrélation de Pearson (linéaire) ayant une valeur de -0.446 (valeur p égale à $1.9 \cdot 10^{-6}$). Nous avons donc une corrélation entre la note donnée par un humain et la valeur de notre indicateur. Cette corrélation n'est pas

extrêmement élevée, mais elle peut être suffisante pour estimer automatiquement la progression d'un étudiant vers la résolution d'un problème.

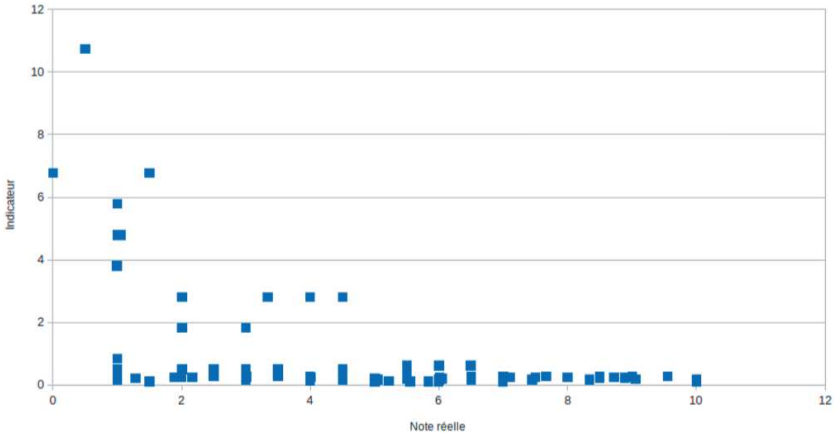

Fig. 4. Valeur de l'indicateur en fonction de la note attribuée par un enseignant.

7 Discussion

Notre indicateur permet de différencier un script sémantiquement correct d'un script incorrect, mais ses performances sont sensiblement supérieures à celles d'un classifieur aléatoire seulement. Une hypothèse pour expliquer ces performances émane du caractère hétérogène du jeux de données d'apprentissage, notamment en termes de taille des scripts qu'il contient. En effet, la distance calculée par notre indicateur tend à augmenter à chaque token du script. Donc plus un script est long, plus l'indicateur risque de retourner une valeur élevée (même si le script s'approche de la solution), alors qu'un script court et incorrect est évalué avec une faible valeur en raison du faible nombre de tokens.

L'étude de la corrélation réalisée sur le second jeu de données donne des indications sur une certaine qualité de généralisation de notre approche. En effet, contrairement au jeu de test issu du premier jeu de données, le jeu de scripts obtenu suite à l'examen terminal correspond à un exercice absent du jeu de données d'apprentissage. Notre méthode semble donc bien se généraliser à des exercices qui n'ont pas été rencontrés durant la phase d'apprentissage. Toutefois, cette corrélation n'est pas extrêmement forte et des scripts très mal évalués humainement obtiennent une valeur de notre indicateur quasiment nulle. Cela peut aussi s'expliquer par l'hypothèse formulée ci-dessus, puisque des scripts presque vides et donc incorrects sont mal notés par l'enseignant humain, alors que notre indicateur retourne une faible valeur. Des tests doivent être menés

pour étudier comment ajuster la méthode de calcul de l'indicateur, notamment en fonction du nombre de tokens du script représentant la solution au problème.

Des développements ont été initiés pour retourner cet indicateur aux apprenants pendant l'activité de programmation. Une première visualisation restitue, sous la forme d'une jauge, la valeur de l'indicateur à chaque tentative d'exécution du script. Une seconde visualisation expose la progression de l'apprenant dans la résolution du problème à partir des valeurs successives de l'indicateur pour un même script. Elle donne la variation de l'indicateur sous la forme d'un dégradé de couleurs pour fournir à l'étudiant une vue instantanée et facile à interpréter de l'évolution de la justesse de son script d'un point de vue sémantique. Ces outils destinés à supporter les apprenants seront expérimentés dès la prochaine année universitaire avec une nouvelle promotion d'étudiants d'IUT Informatique.

8 Conclusion

Nous avons cherché dans ce papier à concevoir un indicateur dont l'objectif est de servir de fondation à des systèmes de guidage automatique et à des outils d'awareness destinés à la fois aux apprenants et aux enseignants pendant la réalisation d'une activité pratique dédiée à l'apprentissage de la programmation. Si de nombreux travaux s'intéressent à la qualité syntaxique du code produit par les apprenants pour soutenir ces activités d'apprentissage, notre originalité consiste à étudier la qualité sémantique du code, c'est-à-dire à évaluer sa proximité avec la solution à un problème posé.

Ainsi, notre contribution principale concerne la conception d'un indicateur reflétant la capacité d'un apprenant à résoudre un problème. Pour répondre à la première question de recherche posée en introduction, nous avons adapté la distance de Levenshtein : à partir de deux chaînes de *tokens* représentant l'arbre de syntaxe abstrait des programmes correspondant, l'indicateur retourne une estimation de la distance d'édition séparant les deux scripts. Pour entraîner et tester cet indicateur, nous avons utilisé un jeu de données comprenant des scripts produits par des étudiants en situation d'apprentissage réelle. Les tests ont révélé que lorsqu'il s'agit de différencier des scripts sémantiquement corrects et incorrects, notre indicateur présente des performances légèrement supérieures à celles d'un classifieur aléatoire. D'autre part, nous avons observé une corrélation inverse entre la valeur de l'indicateur et le score attribué par une évaluation humaine : plus la notation humaine d'un programme est élevée, plus la distance entre ce programme et la solution du problème posé est faible.

Ces premiers résultats sont encourageants et laissent entrevoir l'opportunité de développer de nouveaux modèles et outils dédiés à l'analyse de l'apprentissage de la programmation. Toutefois, de nombreuses améliorations doivent être apportées et plusieurs pistes de recherche doivent être explorées pour améliorer la qualité de notre indicateur. En plus de données additionnelles nécessaires au raffinement des divers paramètres de notre indicateur, la généralisation de notre approche à différents langages de programmation doit être vérifiée, tout comme la prise en compte de programmes plus complexes.

References

1. Spacco, J., Denny, P., Richards, B., Babcock, D., Hovemeyer, D., Moscola, J., Duvall, R.: Analyzing student work patterns using programming exercise data. In: Proceedings of the 46th Technical Symposium on Computer Science Education, pp. 18–23. ACM (2015)
2. Sharma, K., Jermann, P., Dillenbourg, P.: Identifying Styles and Paths toward Success in MOOCs. In: Proceedings of the 8th International Conference on Educational Data Mining, pp. 408–411.
3. Taherkhani, A., Malmi, L.: Beacon-and schema-based method for recognizing algorithms from students' source code. Journal of Educational Data Mining, **5**(2), 69–101 (2013)
4. Saeli, M., Perrenet, J., Jochems, W. M., Zwaneveld, B.: Teaching programming in Secondary school: A pedagogical content knowledge perspective. Informatics in Education, **10**(1), 73–88 (2011)
5. Bey, A., Jermann, P., Dillenbourg, P.: A Comparison between Two Automatic Assessment Approaches for Programming: An Empirical Study on MOOCs. Educational Technology & Society, **21**(2), 259–272 (2018)
6. Denny, P., Luxton-Reilly, A., Carpenter, D.: Enhancing syntax error messages appears ineffectual. In: Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education, pp. 273–278. ACM (2014)
7. Karam, M., Awa, M., Carbone, A., Dargham, J.: Assisting Students with Typical Programming Errors During a Coding Session. In: Proceedings of the 7th International Conference on Information Technology, pp. 42–47. IEEE (2010)
8. Watson, C., Li, F. W., Godwin, J. L.: Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In: Proceedings of the 13th International Conference on Advanced Learning Technologies, pp. 319–323. IEEE (2013)
9. Ahadi, A., Lister, R., Lal, S., Leinonen, J., Hellas, A.: Performance and Consistency in Learning to Program. In: Proceedings of the 19th Australasian Computing Education Conference, pp. 11–16. ACM (2017)
10. Alammary, A., Carbone, A., Sheard, J.: Implementation of a smart lab for teachers of novice programmers. In: Proceedings of the 14th Australasian Computing Education Conference, pp. 121–130. ACM (2012)
11. Carter, E., Blank, G. D.: A tutoring system for debugging: status report. Journal of Computing Sciences in Colleges **28**(3), 46–52 (2013)
12. Sharma, K., Mangaroska, K., Trætteberg, H., Lee-Cultura, S., Giannakos, M.: Evidence for Programming Strategies in University Coding Exercises. In: Proceedings of the 13th European Conference on Technology Enhanced Learning, pp. 326–339. Springer (2018)
13. Ihantola, P., Vihavainen, A., Ahadi, A., Butler, M., Börstler, J., et al.: Educational data mining and learning analytics in programming: Literature review and case studies. In: 2015 ITiCSE on Working Group Reports, pp. 41–63. ACM (2015).
14. Aiouni, R., Bey, A., Bensebaa, T.: An Automated Assessment Tool of Flowchart Programs in Introductory Programming Course using Graph Matching. Journal of e-Learning and Knowledge Society, **12**(2), 141–150 (2016)
15. Beijering, K., Gooskens, C., Heeringa, W.: Predicting intelligibility and perceived linguistic distance by means of the Levenshtein algorithm. Linguistics in the Netherlands, **15**, 13–24 (2008)
16. Levenshtein, V., I.: Binary codes capables of correcting deletions, insertions, and reversals. Soviet Physics Doklady, **10**(8), 707–710 (1966)