Rapport de

Initiation à l'Innovation et à la Recherche

Kévin Nguyen

Rayane Hamani

Florian Dendoncker

Master 1 – Année universitaire 2020 - 2021

Université de Lille

Sommaire

Chapitre I : Notre sujet de recherche

I-1 Introduction

De nos jours, il existe bon nombre d'outils qui facilitent la rédaction numérique, que ça soit l'écriture de programmes ou bien la rédaction de rapports. Ces outils sont très utiles au quotidien et permettent de gagner du temps dans notre rédaction et de renforcer notre fil conducteur dans certains cas. En fait, ces outils sont même intégrés partout, de nos logiciels de traitement de textes à nos environnements de développement en passant par nos navigateurs et notre barre de recherche système. Nous vivons au quotidien avec ces outils et ceux-ci sont tellement puissants et s'améliorent tant de façon permanente que l'on pourrait se demander jusqu'où est-ce qu'ils seraient capables d'aller. Bien-sûr, afin qu'ils soient utilisables par tous, il faut que ces outils respectent certaines contraintes et se détachent de certains biais mais leurs possibilités d'évolution sont telles que l'on est en droit de rêver et surtout, d'expérimenter.

I-2 Problématique

Nous, développeurs, avons la chance de profiter de nombreuses extensions et built-in fonctionnalités dans nos environnements de développement. Mais parce-que notre passion repose sur un raisonnement personnel et un travail minutieux, il est difficile de laisser un tier nous assister entièrement dans cette tâche en même temps au même moment, d'autant plus si ce tier ne nous est pas supérieur et commet régulièrement des erreurs. Ainsi, on est en droit de se demander s'il ne serait pas à notre avantage d'être assisté par un tier (en l'occurrence ici une intelligence artificielle) si celui-ci est capable de nous comprendre et d'améliorer notre code sans modifier le comportement de notre programme, le tout sans que cela ne consomme trop de ressource. En clair :

« Est-ce raisonnable en coût et en productivité, de laisser une IA corriger notre code dynamiquement ? »

I-3 Plan

Pour répondre à cette question, nous allons tout d'abord commencer par regarder les outils déjà existants pour les développeurs, ceux qu'ils peuvent activer et désactiver pour les assister dans leur rédaction de code. Nous regarderons ensuite les projets qui ont déjà été réalisés par des chercheurs et des passionnés du domaine afin de profiter de leurs avancées pour notre projet.

Nous entrerons ensuite plus en détails sur comment répondre à notre problématique et nous verrons ensuite les limites de ce que nous pouvons apporter à la cause. Et enfin, nous émettrons un protocole d'expérimentation tout en essayant d'exclure le plus de biais possibles et en montrant les avantages et possibles inconvénients de notre protocole.

Enfin, nous conclurons sur notre problématique, l'intérêt de celle-ci, les applications potentielles d'une telle nouvelle technologie, etc…

Chapitre II : Ce qui existe déjà

II-1 Les outils mis à disposition aux développeurs

Il y a énormément de différents outils de développement qui sont compris dans les environnements de développement ou sont fournis sous forme d'extension. Pour en citer quelques-uns, il existe des outils d'auto-complétion, des outils de détection d'erreurs de syntaxes, des outils de débogage, des outils de refactorisation, etc…

Il est bon de noter que les outils de développement les plus utilisés sont ceux d'auto-complétion et sont pratiquement toujours déjà intégrés aux IDE.

Comme outils d'auto-complétion intégrés aux IDE les plus utilisés, on retrouve ceux des IDE : Eclipse, Atom, Visual Studio Code, etc...

Comme outils de débogage : CodeView, IDA Pro, GDB, JDB de Java, etc...

Comme outils de détection d'erreurs de syntaxes : CodeSonar, JSHint, etc...

Comme outils de refactorisation : Eclipse (intégré), IntelliJ, Jdeveloper, CodeRush, etc…

Ci-dessus sont des listes non-exhaustives d'outils existants.

II-2 Les projets ayant déjà été réalisés

Des projets ont déjà été réalisés dans le domaine, afin d'améliorer les performances de ces outils, d'améliorer leur précision, leur coût en ressources, etc... Par exemple, les chercheurs à l'origine de l'article 1 (voir la bibliographie) ont analysé la naturalité du langage Java afin d'améliorer l'outil d'auto-complétion de l'IDE Eclipse. Les chercheurs de l'article 2 ont créé un analyseur syntaxique qui corrige les erreurs faites dans des codes, en utilisant seulement ses deux premières suggestions. Les chercheurs de l'article 3 ont fusionné un analyseur syntaxique et les principes de naturalité de langage afin d'améliorer la précision des complétions de déclaration.

Grâce au soutien d'une communauté de développeurs et chercheurs actifs, une partie du travail quant à la réalisation d'une telle IA est sûrement déjà et faite.

Chapitre III : Ce que nous proposons

III-1 Notre proposition

L'idée derrière une IA qui « corrige » dynamiquement notre code, est justement que celle-ci code en même temps que nous. Elle doit être en mesure de corriger toutes les erreurs de syntaxes de l'utilisateur et lui permettre de coder plus efficacement grâce à l'auto-complétion. L'IA prendra donc la forme d'une extension compatible et installable sur tous les IDE les plus populaires afin de pouvoir être utilisable par la majorité des développeurs. Elle sera ainsi composée de trois tâches principales :

- L'analyse du code en temps réel.
- La prédiction des mots les plus probables.
- L'auto-complétion du mot une fois que celle-ci est presque quasi sûre.

Le principe de l'analyse de code en temps réel repose sur le fait de balayer la fonction et ligne de code en train d'être écrite à chaque espace ou ponctuation sauf la zone où se trouve le curseur. Les endroits corrigés seront surlignés et mis en évidence afin de permettre à l'utilisateur de voir les endroits avec lesquels l'IA a interagi.

La prédiction des mots les plus probables est une fonctionnalité commune à la plupart des outils d'auto-complétion tel qu'IntelliSense, etc… Dans le cas de cette IA, celle-ci devra réduire le goulot des mots probables au fur et à mesure que l'utilisateur écrit des lettres.

L'auto-complétion du mot est le fait que l'IA devra proposer un mot pendant l'écriture et finira donc directement le mot à la place de l'utilisateur, une fois que celle-ci est sûre du mot que celui-ci voulait écrire.

L'IA sera entrainée à l'aide du lexique des mots anglais de nombreux projets codés en différents langages. Afin qu'elle propose des mots correspondants au bon langage des projets sur lesquels elle sera testée, l'IA devra aussi être entrainée à différencier les différents langages.

III-2 La portée de notre contribution

Mettre en place un système tel que celui-ci est très complexe, car une correction automatique non-intrusive du code implique, afin de ne pas gêner l'utilisateur dans son travail, de ne jamais commettre d'erreur. Un tel travail est évidement hors de notre portée pour le moment. Les systèmes d'intelligence artificielle devront d'abord grandement s'améliorer avant de rendre un objectif tel que celui-ci atteignable. Mais ce concept reste intéressant à envisager tant il pourrait changer les méthodes de travail de beaucoup de développeurs.

Une possible contribution de notre part serait de continuer à entrainer les systèmes d'IA existants sur de nouveaux projets afin d'en améliorer l'efficacité et la précision, ou lui apprendre de nouveaux langages de programmation. Mais une amélioration pure du système d'apprentissage et de correction demandera sûrement des innovations encore non-développées à ce jour.

Une autre tâche à laquelle on pourrait assister à notre échelle, serait de mettre au point une série de protocoles expérimentaux qui permettraient d'évaluer l'IA une fois créée. Ces protocoles devraient être dispensées de tout biais afin d'émettre une évaluation et un jugement impartial de l'intelligence artificielle en test.

III-3 L'évaluation de l'IA

Une chose à garder en tête lors de la conception de tels systèmes est le coût en ressources. Entrainer une IA permet d'avoir moins à s'occuper de la conception de la fonction décisionnelle du correcteur mais implique aussi souvent des coûts en mémoire et en puissance de calcul très imposants. Il faudra donc faire attention à la puissance nécessaire pour faire tourner un tel système. Des protocoles de test devront être mis en place, une évaluation de la consommation du système chargé de la correction peut être effectué simplement en comparant les ressources consommées par l'IDE avec et sans le système de correction en test. Evaluer l'efficacité du programme est plus complexe, car celle-ci dépend du développeur utilisant le correcteur. En effet, même différents utilisateurs de même niveau ne feront pas les mêmes erreurs car chaque méthode de réflexion est unique. Une comparaison du nombre d'erreur contenues dans un programme entre une version avec et une version sans le correcteur pourrait être un indicateur. Des témoignages d'utilisateurs permettraient également de se faire une idée des points forts et points faibles du système afin de combler ses lacunes.

Le protocole de test, que nous avons pensé, est le suivant :

Soit 2 groupes de personnes. Un groupe utilisera l'IA, tandis que l'autre ne l'utilisera pas. Les 2 groupes travaillerons sur le même programme, dans le même langage et sur le même IDE. Le système d'exploitation et la machine physique seront identiques pour toutes. Toutes les personnes qui participeront au test devront avoir le même niveau algorithmique et la même expérience dans le langage qui servira au test. Ils devront également avoir utilisé le langage récemment. L'objectif de ce test est de mesurer la différence entre le temps moyen de réalisation du projet de chaque groupe. A noter cependant que les utilisateurs qui utiliseront l'IA devront déjà s'être familiarisés avec celle-ci avant l'expérience et pendant un certain temps afin de prendre l'habitude de l'utiliser. Aussi, différents outils de mesure de performances seront installés sur les machines afin de mesurer l'utilisation des ressources en temps réel de l'IDE.

La population visée pour participer à ce test est les programmeurs compétitifs. Ils remplissent tous les critères de sélection et pourront montrer une vraie différence de productivité.

Chapitre IV : Conclusion

IV-1 Conclusion

Un tel système est difficile à évaluer car il est pour l'instant impossible d'en analyser une version concrète.

Mais on peut supposer que malgré l'augmentation du coût en ressources et la complexité de sa mise en place, les bénéfices qu'il pourrait apporter sont tels qu'il deviendrait probablement un indispensable dans beaucoup d'IDE.

Une évolution progressive des outils de complétion vers un système similaire et de correction est sûrement plus probable que l'apparition soudaine d'un concept tel que celui-ci.

Chapitre V : Annexes

V-1 Résumé de notre article de recherche

Les techniques de modélisation statistique du langage ont été appliquées avec succès à de grands corpus de code source, ce qui a donné lieu à toute une série de nouveaux outils de développement de logiciels, tels que des outils de suggestion de code, d'amélioration de la lisibilité, de détection de bugs, etc... Tout cela, se faisant en faisant analyser des réseaux de neurones, des centaines de projets différents. Un problème majeur de ces techniques est que le code introduit un nouveau vocabulaire à un rythme bien plus élevé que le langage naturel, car les nouveaux noms d'identificateurs prolifèrent.

Tant les grands vocabulaires que les problèmes de non-vocabulaire affectent gravement les modèles de langage neuronal du code source, dégradant leurs performances et les rendant incapables de s'adapter à l'échelle. Dans cet article, nous abordons cette problématique en :

1) étudiant l'impact de divers choix de modélisation sur le vocabulaire résultant sur un corpus à grande échelle de 13 362 projets.

2) présentant un modèle de langage neuronal (NLM) de code source à vocabulaire ouvert pouvant être mis à l'échelle d'un tel corpus, 100 fois plus grand que dans les travaux précédents

3) montrant que de tels modèles dépassent l'état de l'art sur trois corpus de code distincts (Java, C, Python). À notre connaissance, ce sont les plus grands NLM de code qui ont été signalés.

V-2 Notre grille de comparaison

Sujet	Tokens	Éléments pris en compte dans la base de connaissance pour faire les prédictions	Critères de ranking sur ces éléments	Méthode d'apprentissage	Autocompletion de code	Correction automatique de la syntaxe	Adaptabilité vers d'autres languages
Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code	mots et sous-mots (commentaires inclus)	Fonctions locales et importées de bibliothèques Variables locales et importées de bibliothèques Bibliothèque générée à partir de l'entrainement	Scalabilité, perte d'informations, fréquence des mots	Supervisée	Oui	Oui	Très bonne adaptabilité sur d'autre projet
On the naturalness of software	mots et tout types	Modèle de langage n-gram basé sur une collection de programmes java et de programmes Ubuntu	Probabilité des n-gram	Non supervisée	Oui	Non	Il faut refaire tout le processus d'entrainement et tokenization pour d'autre langage
Syntax and sensibility: Using language models to detect and correct syntax errors	mots, séparateurs, operateurs, identificateur, littéraux	Modèles de syntaxe correcte extrait d'un entrainement sur 10000 programmes java	Similarité avec le code incorrecte analysé	Supervisée	Non	Oui	Il faut refaire tout le processus d'entrainement et tokenization pour d'autre language
Combining Program Analysis and Statistical Language Model for Code Statement Completion	un mot, une variable, un attribut, une appelle de méthode, un type objet d'une classe	Modèle de langage entraine, combine a un modèle d'occurrence pour la détermination du meilleur candidat	Calcul du meilleur candidat a la complétion via un modèle d'occurrence séparé	Supervisée	Oui	Oui	De base entrainé sur de multiples projets, de langage différent.

Deep-AutoCoder: Learning to Complete CodePrecisely with Induced Code Tokens	character comme token, pour former un vecteur + D son nombre de character par ligne de code	Character level LSTM et token level LSTM	3 couches (input/hidden/output) calcul de probabilité via un vecteur représentant une séquence de tokens	Supervisée	Oui	Non	Plus général, mais nécessite de refaire le travail d'analyse de code du langage
Neural Code Completion	mots et séquences de mots (nodes dans leur paper)	arbre de synthaxe abstrait	Une architecture NT2N est utilisée pour calcule l'élément suivant le plus probable	Supervisée	Oui	Non	Semble être ouvert à JS notamment mais aussi aux autres langages
Contextual Code Completion Using Machine Learning	mots et séquences de mots (vecteurs dans leur paper)	Code source Linux, un projet en C et Twisted, un projet Python de librairie de networking	Calcul de probabilité sous forme d'un vecteur de tokens	Non supervisée	Oui	Non	Testé en C et en Python. Semble s'adapter aux langages mais nécessite du training

V-3 Bibliographie

Article de recherche :

Big Code != Big Vocabulary : Open-vocabulary models for source code

Article 1 :

On the naturalness of software

Article 2 :

Syntax and sensibility : Using language models to detect and correct syntax errors

Article 3 :

Combining program analysis and statistical language model for code statement completion

Article 4 :

Deep-AutoCoder : Learning to complete code precisely with induced code tokens

Article 5 :

Neural code completion

Article 6 :

Contextual code completion using machine learning

Et aussi plusieurs recherches Google en lien avec les intelligences artificielles, leur fonctionnement, leur algorithme, leur validation, etc… Vraiment beaucoup de recherche pour le coup.