Université de Montréal

Sur l'élaboration de meilleures techniques pour l'apprentissage auto-supervisé des représentations du code

par

Lucas Maes

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en informatique

Juillet 3, 2023

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Sur l'élaboration de meilleures techniques pour l'apprentissage auto-supervisé des représentations du code

présenté par

Lucas Maes

a été évalué par un jury composé des personnes suivantes :

Glen Berseth
(président-rapporteur)

Houari Sahraoui
(directeur de recherche)

Michalis Famelis
(membre du jury)

Résumé

Les représentations du code apprises par les modèles d'apprentissage profond sont une composante cruciale pour certaines applications en génie logiciel telles que la recherche de code ou la détection de clones. Les performances de ces applications dépendent de la qualité des représentations apprises par les modèles. De fait, des représentations possédant peu de bruit et contenant des informations avec un haut niveau d'abstraction, comme la sémantique fonctionnelle, facilitent la résolution de ces tâches. En effet, la recherche de code nécessite de comprendre les objectifs des morceaux de code pour les comparer avec une requête en langage naturel, tandis que la détection de clone exige de déterminer si deux morceaux de code ont la même sémantique fonctionnelle. La capacité des modèles à apprendre des représentations contenant de telles informations abstraites est donc cruciale pour la bonne résolution de ces tâches.

Cependant, il est toujours difficile pour les modèles de code d'apprendre des représentations abstraites indépendantes de la syntaxe, par exemple la sémantique fonctionnelle. Ce mémoire se consacre donc à l'élaboration de meilleures techniques pour l'apprentissage des représentations du code via l'apprentissage auto-supervisé. Plus spécifiquement, nous nous sommes concentrés sur deux tâches centrales dans l'automatisation du génie logiciel nécessitant un minimum de compréhension de la sémantique fonctionnelle, à savoir, la recherche de code et la détection de clones de type 4. Ce mémoire propose différentes approches à différents degrés d'entraînement. Le premier degré est le pré-entraînement et consiste à apprendre des représentations génériques du code adaptables à n'importe quels problèmes. Le second est le peaufinage, modifiant les représentations apprises pour un problème spécifique.

Tout d'abord, nous proposons un nouvel algorithme de pré-entraînement pour les modèles de code utilisant une méthode non contrastive régularisée adaptée de VICReg [**14**], permettant l'apprentissage de représentations génériques. Ensuite, nous proposons un nouvel objectif de peaufinage des modèles de code utilisant la distillation des connaissances d'un ensemble de modèles déjà peaufinés, appelés enseignants, sur un modèle étudiant, lui

permettant ainsi l'apprentissage de représentations plus abstraites.

L'ensemble des contributions vise à améliorer les représentations du code et à maximiser les performances des modèles d'apprentissage automatique pour le code, mais aussi à déterminer quel est le meilleur degré d'entraînement à adopter pour cela. Les résultats expérimentaux et les analyses menées dans ce mémoire sont préliminaires et ne permettent pas de tirer de conclusions définitives. Néanmoins, il est important de souligner que la deuxième contribution surpasse la méthode classique de peaufinage des modèles pour la recherche de code. De plus, les approches décrites proposent des pistes de directions de recherche innovantes et non conventionnelles.

Mots clés : Génie logiciel, apprentissage profond, apprentissage auto-supervisé, non contrastif, distillation, représentation du code, recherche de code, détection de clone.

Abstract

Code representations learned by deep learning models are a crucial component for certain software engineering applications such as code search or clone detection. The performance of these applications depends on the quality of the representations learned by the models. In fact, low-noise representations containing highly abstract information, such as functional semantics, facilitate the resolution of these tasks. Indeed, code search requires understanding the objectives of code snippets in order to compare them with a natural language query, while clone detection requires determining whether two code snippets have the same functional semantics. The ability of models to learn representations containing such abstract information is therefore crucial to the successful resolution of these tasks.

However, it is still difficult for code models to learn abstract representations that are independent of syntax, such as functional semantics. This thesis is therefore dedicated to developing better techniques for learning code representations via self-supervised learning. More specifically, we focus on two central tasks in software engineering automation requiring a minimum understanding of functional semantics, namely, code search and type 4 clone detection. This work proposes different approaches with different degrees of training. The first, pre-training, consists in learning generic code representations that can be adapted to any problem. The second is fine-tuning, modifying the representations learned for a specific problem.

First, we propose a new pre-training algorithm for code models using a regularized non-contrastive method adapted from VICReg [**14**] enabling the learning of generic representations. Secondly, we propose a new code model refinement objective using knowledge distillation of a set of already refined models, called teachers, on a student model allowing it to learn more abstract representations.

The aim of all these contributions is not only to improve code representations and maximize the performance of machine learning models for code, but also to determine the best degree of training to adopt for this purpose. The experimental results and analyses

carried out in this thesis are preliminary and do not allow to draw formal conclusions. Nevertheless, it is important to underline that the second contribution outperforms the classical model refinement method for code search. Moreover, the approaches described suggest innovative and unconventional research directions.

Keywords: Software engineering, deep learning, self-supervised learning, non-contrastive, distillation, code representation, code search, clone detection.

Table des matières

Liste des tableaux

Liste des figures

Liste des sigles et des abréviations

CE	Entropie croisée, de l'anglais *Cross-entropy*
JEA	Architecture de représentations jointes, de l'anglais *Joint Embeddings Architecture*
MLM	Modélisation du langage masqué, de l'anglais *Masked Language Modelling*
RNC	Non contrastif régularisé, de l'anglais *Regularized Non Contrastive*
MRR	Rang réciproque moyen, de l'anglais *Mean Reciprocal Rank*
MAP	Moyenne de moyenne de précision, de l'anglais *Mean Average Precision*

Remerciements

Tout d'abord, j'aimerais exprimer ma plus grande gratitude envers mon directeur de recherche, le Professeur Houari Sahraoui, ainsi que mes collaborateurs Aton Kamanda et Martin Weyssow, pour leur soutien incommensurable, leurs conseils et leur aide inestimable tout au long de ces deux années. Je leur suis énormément reconnaissant pour l'énergie et l'engagement qu'ils ont consacré à mon développement en tant que chercheur, me permettant de m'épanouir dans un cadre stimulant et passionnant pour mon développement intellectuel.

Ensuite, je souhaite adresser quelques mots à ma mère, dont la confiance et les encouragements à mon égard ont été une source de motivation inégalée. Je lui suis profondément reconnaissant d'avoir toujours cru en mes capacités et d'avoir toujours été présente pour me remonter le moral dans les moments difficiles.

Enfin, j'aimerais exprimer mes plus sincères remerciements aux membres du jury pour le temps et l'effort qu'ils consacreront à l'examen de ce mémoire. Leurs remarques constructives et leur rigueur dans l'évaluation de ce travail contribueront sans aucun doute à sa qualité, ainsi qu'à l'amélioration de ma rigueur et de ma capacité à mener des recherches académiques.

Chapitre 1

Introduction

1.1. Contexte

Avec l'avènement de l'informatique depuis les années 50, le logiciel a pris de plus en plus de place dans notre société [**20**]. Au fur et à mesure des années, les logiciels n'ont cessé de se complexifier de par leur taille, celle de leurs modules, celle de leurs procédures, ou encore par la complexité des conditions. Les coûts de production et de maintenance étant directement corrélés à la complexité de ces derniers [**104**], les ingénieurs logiciels ont donc très vite trouvé bénéfique d'automatiser des étapes lors de la conception de logiciels [**42**], réduisant ainsi le nombre d'erreurs d'un tel système complexe et, in fine, les coûts de production et de maintenance de celui-ci. Au cours des dernières années, l'apprentissage automatique s'est imposé comme une méthode efficace dans l'écosystème d'automatisation du développement de logiciels [**144, 137, 139, 4**], que ce soit pour la génération de code [**13, 138, 88, 80**], la détection de bogues [**108, 49**], la réparation de programmes [**146, 143**], la détection de clones [**140, 135, 151**] ou encore la recherche de code [**55, 86, 87**]. À cela s'ajoute que de plus en plus de gros acteurs dans le domaine du logiciel consacrent du temps et de l'argent à l'élaboration de modèles pour le génie logiciel. On peut citer, entre autres, Microsoft avec CodeBERT [**41**], OpenAI avec Copilot [**24**] et ChatGPT, DeepMind avec AlphaCode [**88**] et AlphaDev [**96**], ou encore Salesforce avec CodeT5 [**138**].

1.2. Problématique

Malgré la puissance et l'efficacité de l'apprentissage automatique en matière de production et de maintenance de logiciels [**137, 139, 4**], ces performances dépendent directement de la qualité des représentations apprises par le modèle d'apprentissage automatique utilisé [**17, 37, 128**]. En effet, plus les représentations du code apprises sont de qualité, plus il sera simple d'obtenir de bonnes performances sur une tâche donnée. Cela s'explique par le fait que la représentation d'une donnée avec un bruit minimal et un haut niveau d'abstraction

contient des informations génériques pour la résolution d'une multitude de problèmes [**18**]. Par exemple, une photo d'un chien interagissant avec un enfant pourrait contenir dans sa représentation une multitude d'informations haut niveau, telles que la race du chien, sa distance avec l'enfant, l'émotion du chien ou de l'enfant, etc. La réussite de certaines tâches d'automatisation du développement de logiciels grâce à l'apprentissage profond repose en grande partie sur la capacité des modèles à apprendre des représentations hautement abstraites contenant des informations abstraites telles que la sémantique fonctionnelle, c'est-à-dire le but du code, afin d'obtenir de bonnes performances [**123, 69**]. La recherche de code [**115, 8, 93, 45, 68, 125, 98, 109**], par exemple, nécessite de trouver le morceau de code le plus pertinent dans une base de données en se basant sur une requête formulée en langage naturel. Pour y parvenir, il est donc nécessaire de comprendre les objectifs des différents morceaux de code afin de les comparer avec ce qui est recherché. Un autre exemple de tâche est la détection de clones de type 4 [**129, 103, 112, 121, 48, 78, 7**]. Le but est de déterminer si deux morceaux de code partagent la même sémantique fonctionnelle ou non. Ceci démontre l'utilité de contenir une information aussi abstraite que la sémantique fonctionnelle dans les représentations du code apprises par les modèles. Il est important de noter qu'il existe également des clones de type 1 à 3. Cependant, ces derniers, se concentrent sur des similarités de code basées sur la structure, la syntaxe et les détails mineurs. Une méthode résolvant la détection de clones de type 4 ne nécessiterait donc pas de modifications majeures pour la détection de clones de types inférieurs à 4, car ces derniers ont les mêmes fonctionnalités, mais la structure, la syntaxe et des détails mineurs différents.

Cependant, des travaux antérieurs ont souligné la sensibilité des représentations de code aux petits changements syntaxiques tout en conservant une sémantique fonctionnelle identique, mettant donc en évidence une limitation des modèles de code dans leur capacité à saisir des représentations abstraites indépendantes de la syntaxe [**69, 61, 111, 152**]. Par conséquent, cette constatation ouvre la voie à nos contributions, visant à améliorer la capacité des modèles à capturer cette sémantique fonctionnelle.

1.3. Contributions

Via ce mémoire, nous proposons différentes contributions visant toutes à apprendre de meilleures représentations du code grâce à l'apprentissage auto-supervisé. L'apprentissage auto-supervisé est une méthode d'apprentissage automatique où un modèle apprend à partir de données non étiquetées en créant des étiquettes artificielles à partir des informations inhérentes aux données elles-mêmes. Cela s'avère très utile dans le domaine du code, où il est souvent difficile et coûteux d'obtenir des annotations précises pour un grand volume de données. L'apprentissage auto-supervisé permet de contourner cette limitation, par

exemple, en apprenant des représentations du code en prédisant le prochain jeton dans une séquence de code étant donné son contexte.

Nous nous sommes concentrés sur les deux tâches mentionnées dans la section 1.2, à savoir la recherche de code et la détection de clones de type 4. Dans le but d'améliorer la qualité de la sémantique fonctionnelle apprise par les réseaux de neurones, nous avons utilisé l'apprentissage auto-supervisé pour apprendre les représentations du code pour ces tâches dans un premier temps. En effet, ces deux tâches sont les plus utilisées par la communauté du génie logiciel pour évaluer la qualité de la sémantique fonctionnelle apprise par les réseaux de neurones [**69, 140, 68, 86, 87, 55, 56, 136**].

Plus spécifiquement, nous avons décidé de contribuer à plusieurs niveaux d'apprentissage, à savoir le pré-entraînement et le peaufinage. Le pré-entraînement vise à apprendre des représentations contenant des informations génériques pour la résolution de nombreux problèmes à partir de poids initialisés aléatoirement. Le peaufinage, quant à lui, part d'un modèle pré-entraîné et peaufine ses représentations en capturant les informations nécessaires pour résoudre une tâche spécifique. La figure 1.1 illustre ces deux niveaux d'apprentissage en montrant à gauche le pré-entraînement d'un modèle et à droite le peaufinage de ce dernier sur deux tâches différentes. La variation des couleurs de gauche à droite illustre les changements de poids en fonction du temps d'entraînement, le peaufinage commence avec les mêmes poids que ceux résultant du pré-entraînement.

Pour le premier niveau, c'est-à-dire le pré-entraînement, nous proposons (1) un nouvel algorithme, nommée `rncp-code`, utilisant une approche non contrastive régularisée et une méthode d'augmentation de données pour les modèles de code. L'utilisation d'une méthode non contrastive permet une plus grande expressivité dans l'espace des représentations apprises, car la "réduction dimensionnelle" est réduite [**47, 71, 66, 130, 12**]. Cela permet d'obtenir de meilleures représentations que les méthodes contrastives. Les approches non contrastives sont détaillées à la section 3.2.

Pour le deuxième niveau, le peaufinage, nous proposons (2) un nouvel algorithme de peaufinage d'un modèle déjà pré-entraîné dans un souci de gain de temps et de coût d'entraînement. En effet, nous souhaitions étudier si la qualité des représentations apprises par les modèles pré-entraînés, qui sont à l'état de l'art, pourrait constituer un point de départ suffisant pour améliorer les représentations dans la capture de la sémantique fonctionnelle sans repartir de zéro. Notre approche, nommée CodeMaster, est basée sur le concept de distillation de modèle, où les modèles déjà peaufinés jouent le rôle d'enseignant et le nouveau modèle celui d'étudiant. En plus de l'objectif de distillation traditionnel, notre contribution

ajoute un objectif de peaufinage visant à attribuer à chaque enseignant un score de confiance pour chaque ensemble de données observé, guidant ainsi de manière plus précise la distillation des connaissances.

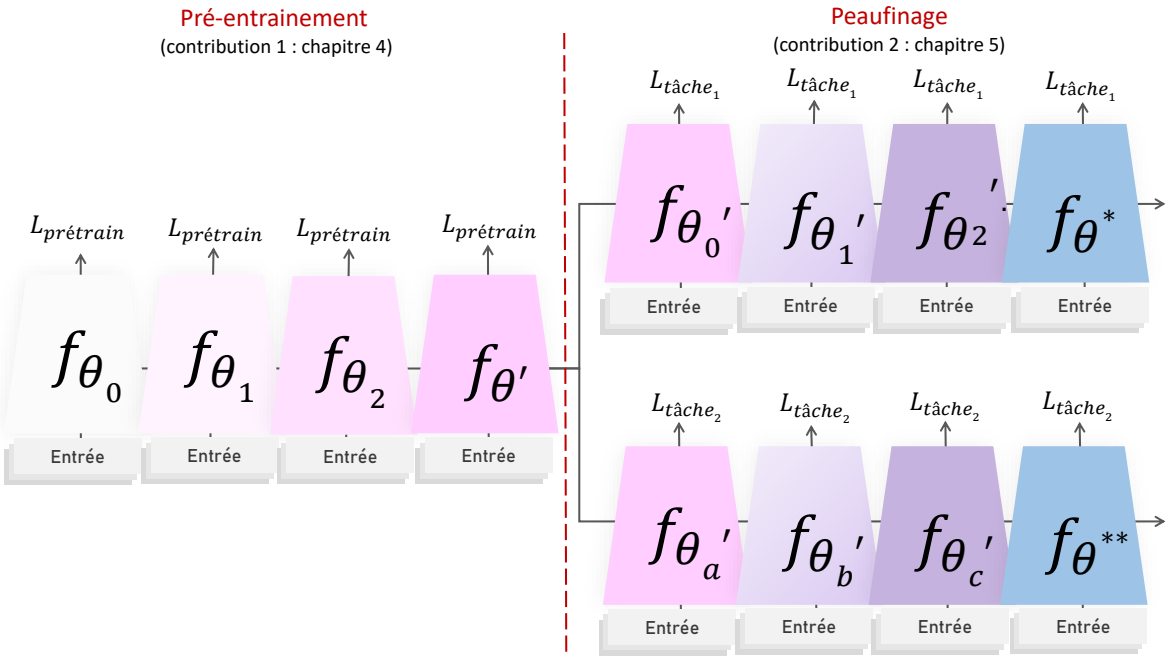


Fig. 1.1. Aperçu des différents niveaux d'entrainements pour l'apprentissage profond.

1.4. Structure de la thèse

Ce mémoire est décomposé de la manière suivante : le chapitre 2 constitue un rappel théorique des connaissances de base requises pour la bonne compréhension de ce mémoire, à savoir l'apprentissage des représentations, mais également comment utiliser ce type de modèle pour la recherche de code et la détection de clones de type 4.

Ensuite, le chapitre 3 détaillera les travaux connexes sur l'apprentissage de représentations pour le code, mais passera également en revue l'état de l'art sur la recherche de code et la détection de clones utilisant les systèmes d'apprentissage automatique.

Pour continuer, les chapitres 4 et 5 constitueront une explication claire et précise de l'approche conceptuelle et expérimentale adoptée lors de l'entraînement, du peaufinage et de l'évaluation des modèles utilisés. Ces chapitres sont chacun décomposés de la manière suivante : tout d'abord, une vue d'ensemble de la méthode est exposée et sert de courte introduction au lecteur. Ensuite, nous fournissons une explication détaillée et formelle de

l'approche utilisée. S'ensuit une section exposant les évaluations et les résultats obtenus pour cette idée. Enfin, la dernière section constitue une discussion sur les limites des approches exposées et sur les hypothèses, ainsi que sur les causes d'échec de ces approches, le cas échéant.

Finalement, le chapitre 6 est une conclusion de ce mémoire contenant également les limitations des approches proposées, mais également des pistes pour de potentiels travaux ultérieurs.

Chapitre 2

Généralités et définitions

Comme mentionné à la section 1.2, le problème traité dans ce mémoire est celui de l'apprentissage des représentations du code capturant des informations abstraites. Nous évaluons cette capacité sur deux tâches couramment utilisées par la communauté du génie logiciel, à savoir la recherche de code [**68**] et la détection de clones [**140**]. Ce chapitre décrit donc comment les modèles neuronaux sont entraînés pour ces tâches ainsi que leurs modes d'inférence.

2.1. Apprentissage des représentations

La majorité de l'apprentissage profond se résume à une problématique bien connue : l'apprentissage de bonnes représentations [**18**]. Une représentation est une transformation des données brutes qui facilite l'apprentissage et l'extraction de caractéristiques pertinentes des données originales. À noter que généralement, les représentations sont des vecteurs dans un espace de dimensions élevées, appelé *espace des représentations*. Généralement, le problème d'apprentissage des représentations se fait sans intervention humaine via l'utilisation d'un modèle d'apprentissage profond, autrement dit un réseau de neurones. La figure 2.1 illustre la transformation d'une donnée brute prise en entrée d'un réseau de neurones en sa représentation, c'est-à-dire un vecteur appartenant à l'espace des représentations.

Plus formellement, étant donné un jeu de données brutes $\mathcal{D} := \{x_1, x_2, ..., x_n\}$ où $x_i \in \mathbb{R}^d$, généralement dans un espace à très hautes dimensions d, le problème d'apprentissage de bonnes représentations correspond à trouver les paramètres θ d'une fonction paramétrique $f_\theta : \mathbb{R}^d \rightarrow \mathbb{R}^h$, où $d >> h$ de telle manière à ce qu'étant donné un point $x_i \in \mathcal{D}$, sa représentation $z_i = f_\theta(x_i)$ extrait des caractéristiques pertinentes des données brutes de $\mathcal{D}$. À noter que la fonction paramétrique f_θ est un réseau de neurones, plus communément appelé *encodeur*.

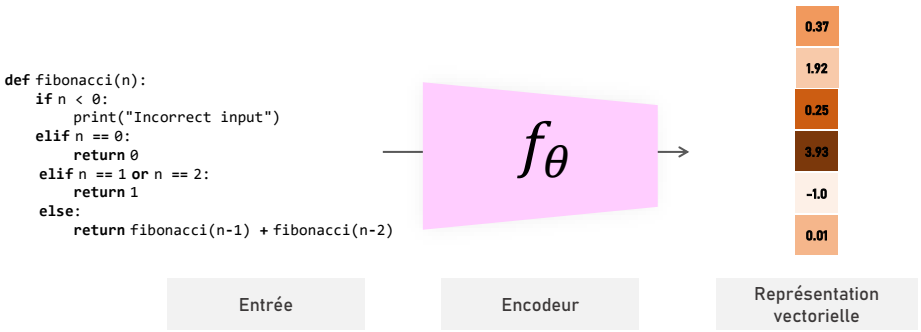


Fig. 2.1. Illustration de la transformation d'une donnée brute en entrée d'un encodeur en sa représentation vectorielle.

Pour entrainer un encodeur à produire les bonnes représentations correspondant aux entrées, il n'existe pas de manière directe de quantifier cela. Il est donc usuel de passer par des moyens alternatifs appelés fonctions de coûts, souvent dénotés $\mathcal{L}$. Les moindres carrés (MSE) par exemple, sont utilisés par les taches de régressions. D'autres fonctions d'objectifs existent pour d'autres tâches également, l'entropie croisée (CE), par exemple, est utilisée pour les taches de classifications. Le point commun entre tous ses objectifs est qu'ils sont souvent différentiables. En conséquence, la méthode la plus couramment utilisée est appelée la rétropropagation. Cette méthode s'appuie sur le fait qu'il existe une succession d'opérations entre les paramètres du modèle et la fonction de coût qui soient différentiables. Ce qui permet, en utilisant la règle de dérivation des fonctions composées, de calculer le gradient de la fonction de coût par rapport au poids $\nabla_\theta \mathcal{L}$. Ce dernier pointe dans la direction de l'espace pour augmenter le plus vite possible $\mathcal{L}$. Cependant, le but étant de minimiser $\mathcal{L}$, il est donc nécessaire de prendre d'utiliser l'opposé de la fonction de coût, $-\mathcal{L}$. L'entrainement d'un réseau de neurones f_θ consiste donc à minimiser $\mathcal{L}$ en adaptant les poids θ de ce dernier avec une règle de mise à jour du style :

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta \mathcal{L} \tag{2.1.1}$$

où α est un scalaire déterminant à quel point la mise à jour des poids sera importante.

2.2. Apprentissage profond pour le génie logiciel

Différentes tâches d'automatisation utilisant des réseaux de neurones sont possibles en génie logiciel. En supposant qu'un encodeur E ait déjà été entraîné sur du code, cette section décrit comment l'inférence peut se faire pour deux tâches de génie logiciel : la recherche de code et la détection de clones de type 4. À noter qu'une manière d'entraîner E spécifiquement pour ces deux tâches sera formalisée dans le chapitre 5.

2.2.1. Recherche de code

La recherche de code est une tâche consistant à trouver, parmi un ensemble de morceaux de code, celui le plus pertinent étant donné une requête en langage naturel.

En pratique, étant donné un encodeur E pré-entrainé sur du code et du langage naturel. La recherche de code consiste à encoder via E un ensemble $\{c_1, c_2, ..., c_n\}$ contenant n morceaux de code ainsi qu'une requête en langage naturel r. Admettons que les vecteurs représentations des codes et de la requête soient de dimensions h après avoir été encodé par E. Nous obtenons donc un ensemble de vecteurs que l'on peut représenter comme une matrice pour le code $C \in \mathbb{R}^{n \times h}$ et un vecteur pour la requête $z \in \mathbb{R}^h$.

Pour connaitre la proximité entre les morceaux de codes et la requête, on peut mesurer la similarité entre les vecteurs constituants C et z avec la distance cosinus entre les vecteurs dans l'espace des représentations. Pour rappel, la distance cosinus d entre deux vecteurs a et b est définie comme :

$$d(a, b) = \frac{a \cdot b}{\|a\| \cdot \|b\|} \tag{2.2.1}$$

Pour plus d'efficacité, cette opération peut se faire en une fois si l'on normalise chaque vecteur ligne composant C et z par leur norme respective et que l'on prend le produit Rz. Cela a pour résultat de nous donner un vecteur $s \in \mathbb{R}^n$ où

$$\forall i \in [n] : s_i := d(R_{i,:}, z) \tag{2.2.2}$$

Pour connaitre le code qui correspond le mieux à la requête. Il suffit de trouver, dans s , l'indice de l'élément le plus grand. Cela donnera l'indice du morceau de code qui a la plus grande similarité cosinus avec la requête. Plus formellement,

$$c^* := \arg\min_i s_i \tag{2.2.3}$$

où c^* est l'indice du code le plus similaire à la requête r.

En apprentissage automatique, la méthode utilisée pour entrainer un modèle E à la recherche de code n'est pas si différente de la manière pratique d'y parvenir. En effet, l'entrainement consiste à encoder via E un ensemble $\{c_1, c_2, ..., c_n\}$ contenant n morceaux de code ainsi qu'un ensemble $\{r_1, r_2, ..., r_n\}$ contenant n documentations chacune correspondante à son code. On obtient après avoir encodé des paquets de codes et de leurs documentations deux matrices $C \in \mathbb{R}^{n \times h}$ et $R \in \mathbb{R}^{n \times h}$. Ensuite, chaque vecteur ligne composant C et z sont normalisés par leur norme respective et le produit matriciel $RC^\top$ est calculé nous donnant une matrice S de dimensions $n \times n$ tel que :

$$\forall i, j \in [n] : S_{ij} := d(R_i, C_j) \tag{2.2.4}$$

où d est la similarité cosinus. Finalement, pour l'entropie croisée (CE) est utilisé comme fonction d'objectif pour comparer la matrice des scores S avec la matrice identité I_n.

$$\mathcal{L} = CE(S, I_n) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{n} I_{ij} \log(S_{ij}) \tag{2.2.5}$$

Cela a pour effet de forcer E à rapprocher les représentations des vecteurs constituants les éléments de la diagonale de S de 1 et éloigner les autres. C'est-à-dire les représentations du code et de leurs documentations respectives doivent avoir une similarité cosinus plus proche de 1 et les autres s'éloigner. La figure 2.2 illustre le processus général d'entrainement d'un modèle neuronal pour la recherche de code.

2.2.2. Détection de clone (Type 4)

La détection de clone de type 4 est un problème de classification visant à déterminer si deux morceaux de code a et b sont similaires au niveau de la sémantique fonctionnelle (cf. 1.2). En pratique, étant donné un encodeur E pré-entrainé sur du code, $z_1 := E(a), z_2 := E(b) \in \mathbb{R}^h$ les représentations respectives de a, b et $w \in \mathbb{R}^{2h}$ le vecteur correspondant aux poids d'un classificateur linéaire. La probabilité que a et b soient détectés comme des clones $(y = 1)$ est donnée par :

$$P(y = 1 | z_1, z_2) = \sigma(w \cdot (z_1 \oplus z_2)) \tag{2.2.6}$$

où $\oplus$ est l'opérateur de concaténation entre deux vecteurs et σ la fonction sigmoïde.

L'entrainement d'un système neuronal à détecter des clones ne change pas de son utilisation en pratique. Il suffit d'utiliser l'entropie croisée binaire (CE) comme fonction d'objectif pour comparer la probabilité résultante de la sigmoïde $\hat{y} := P(y = 1 | z_1, z_2)$ avec l'étiquette correspondant aux deux morceaux de codes a et b à savoir $y \in 0{,}1$.

$$\mathcal{L} = BCE(\hat{y}, y) = -\left[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right] \tag{2.2.7}$$

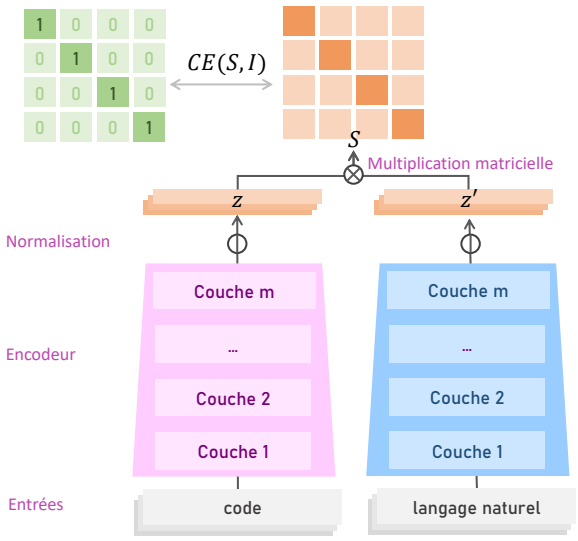


Fig. 2.2. Aperçu du processus générale d'entrainement de modèles neuronaux pour la recherche de code.

où BCE est l'entropie croisée binaire, qui est un cas particulier de l'entropie croisée (CE) avec seulement deux classes. La figure 2.3 illustre le processus général d'entrainement d'un modèle neuronal pour la détection de clone.

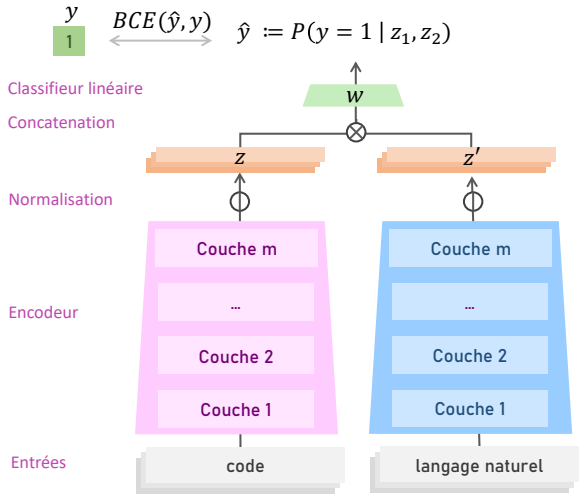


Fig. 2.3. Aperçu du processus générale d'entrainement de modèles neuronaux pour la détection de clone.

Chapitre 3

Travaux Connexe

3.1. Apprentissage contrastif

L'apprentissage contrastif est une méthode d'apprentissage auto-supervisé qui vise à apprendre une correspondance entre une distribution de points de données et un espace de représentations de telle sorte que l'extraction de caractéristiques lors de l'encodage des données soit bonne.

Cette méthode fonctionne en minimisant la distance entre les représentations de points de données similaires, paires *positives*, en maximisant la distance entre les points de données non similaires, appelées paires *négatives*, dans l'espace des représentations [**57**].

La fonction d'objectif *InfoNCE* proposée dans *Contrastive Predictive Coding* (CPC) [**105, 60**] est un objectif souvent utilisé dans les méthodes d'apprentissage contrastif, comme DeepInfoMax [**63**]. Contrairement à CPC, qui se concentre sur l'apprentissage de l'extraction de caractéristiques en prédisant le futur à partir du présent, d'autres méthodes comme SimCLR [**25, 26**] et MoCo [**58, 27**] utilisent une architecture de représentations jointes (JEA) avec un grand nombre d'exemples négatifs pour créer le contraste entre les données et apprendre une bonne transformation des données brutes vers l'espace des représentations. La figure 3.1 illustre une approche contrastive pour le code provenant de [**69**] utilisant MoCo [**58**]. L'approche consiste à minimiser la distance des représentations provenant de vues similaires et maximiser la distance entre celles dissimilaires, en utilisant un grand ensemble d'exemple négative pour cela.

L'apprentissage contrastif a obtenu des résultats impressionnants dans les domaines de la vision par ordinateur [**81, 58, 27, 26, 25**] et le traitement automatique du langage naturel [**102, 44**]. Cependant, certains travaux récents ont démontré que bien que les méthodes contrastives puissent éviter l'*effondrement complet*, un problème où toutes les données brutes

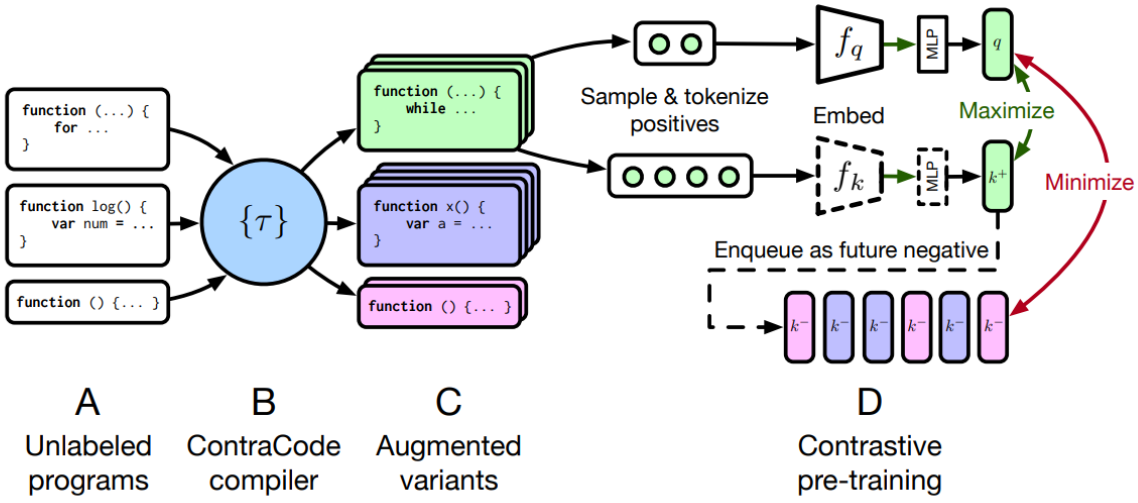


Fig. 3.1. Aperçu d'une approche contrastive pour le code. La figure provient de Jain et al.[**69**].

sont transformées en un même point de l'espace des représentations, elles peuvent souffrir de l'*effondrement dimensionnel*. Dans ce cas, la transformation apprise projette les données brutes dans un sous-espace vectoriel de l'espace des représentations. L'*effondrement* est un problème, car il limite la capacité du modèle à apprendre l'extraction de caractéristiques informatives dans l'espace des représentations et rend plus difficile la préservations de la topologie de la variété des données [**47, 71, 66, 130, 12**]. L'effondrement dimensionnel peut être dû à la difficulté d'apprendre une bonne transformation entre les données brutes et l'espace des représentations, car il existe une infinité de manières de créer des paires négatives.

3.2. Apprentissage non-contrastif

Contrairement à l'approche contrastive, les méthodes non-contrastives offrent une alternative pour apprendre la transformation entre les données brutes et l'espace des représentations, tout en tentant de conserver sa topologie. Ce type de méthode repose sur l'utilisation de paires positives de données (typiquement deux vues du même objet sémantique, par exemple deux photos d'un labrador) et sur des astuces pour éviter l'*effondrement complet* et réduire l'*effondrement dimensionnel*. Les méthodes non-contrastives ont produit des résultats impressionnants dans le domaine de la vision par ordinateur. BYOL [**54**] et SimSiam [**28**] ont utilisé un projecteur pour rompre la symétrie entre les encodeurs et ainsi éviter l'effondrement de l'espace de représentations. W-MSE [**38**] utilise la technique de *whitening*

pour éviter l'effondrement. Certaines méthodes non-contrastives, appelées *régularisées*, utilisent des termes de régularisation pour éviter l'effondrement. Par exemple, Barlow-Twins [**150**] utilise un terme de régularisation qui fait tendre la matrice de covariance vers la matrice identité afin de décorréler les caractéristiques apprises par l'encodeur. VICReg [**14**] améliore cet objectif d'apprentissage en ajoutant deux autres termes de régularisation, notamment un terme d'invariance qui contraint les représentations à être proches via un objectif de moindres carrés, et un terme de conservation de la variance entre les dimensions de chaque représentation au-dessus d'un certain seuil. Enfin, certains se servent du principe de réduction de taux [**149**] pour régulariser l'apprentissage des représentations. La figure 3.2 illustre une méthode non-contrastive provenant de [**150**]. Elle consiste à créer deux vues positives d'une image, calculer leurs représentations respectives, puis décorréler leurs caractéristiques en minimisant les éléments hors diagonale de la matrice de covariance des représentations. Il est à noter que tout cela se fait sans le moindre exemple négatif, contrairement aux méthodes contrastives décrites dans la section 3.1.

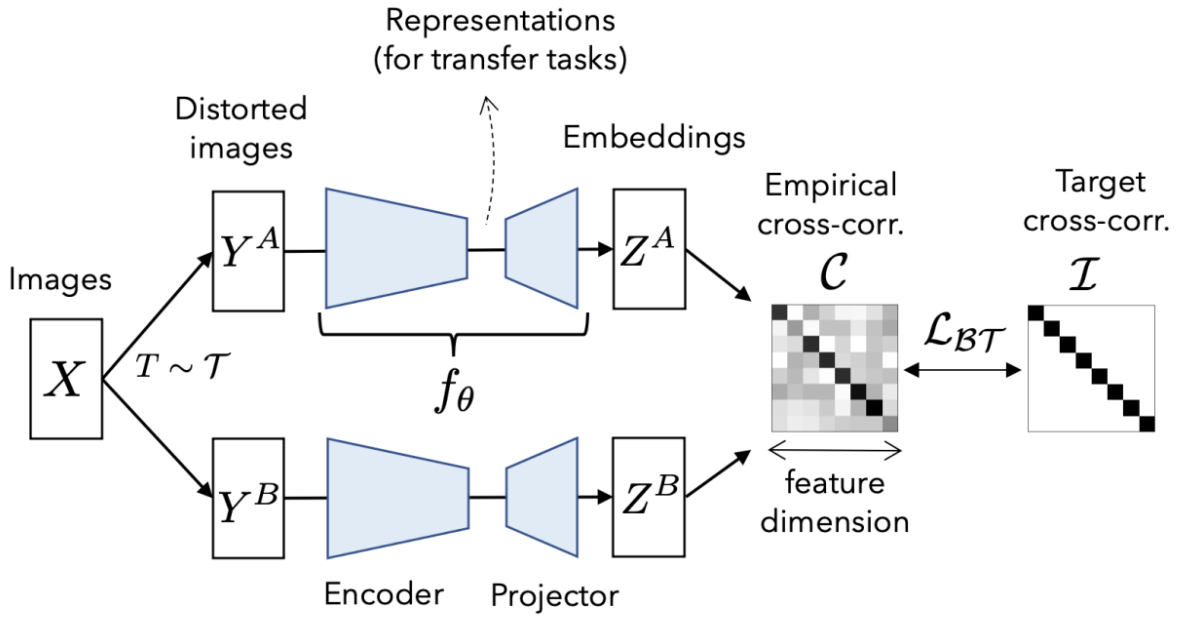


Fig. 3.2. Aperçu d'une approche non-contrastive. La figure provient de Zbontar et al.[**150**].

À noter que toutes ces méthodes ont en commun l'utilisation des mêmes techniques d'augmentation de données pour la vision par ordinateur introduites dans BYOL [**54**]. Cependant, certains travaux antérieurs ont tenté d'adapter ce succès à d'autres domaines tels que le traitement du langage naturel et le génie logiciel.

Pour le génie logiciel, [**83**] ont augmenté les données pour la recherche de code directement dans l'espace des représentations en appliquant des perturbations stochastiques aux vecteurs. ContraCode [**69**] utilise des transformations de code JavaScript telles que l'élimination de code non exécuté, le pré-calcul de constantes et la transformation des boucles *for* en boucles *while* pour augmenter les données d'entrée de leur modèle.

Plus récemment, certains travaux ont commencé à utiliser de grands modèles de langage pour augmenter les données textuelles par la génération de données synthétiques [**32**].

3.3. Distillation des connaissances

La distillation de connaissances est une méthode d'entraînement pour les modèles d'apprentissage profond consistant à transférer les connaissances d'un modèle enseignant à un modèle étudiant.

Initialement introduite par Hinton et al. [**62**], cette méthode avait pour objectif de compresser les connaissances de l'enseignant dans un modèle étudiant avec moins de paramètres. Cette approche a également été adaptée pour la recherche de code par Clement et al. [**30**], qui ont proposé un nouvel objectif de distillation basé sur une augmentation de données. L'approche la plus similaire à celle proposée dans la section 5 est sans doute celle de Li et al. [**85**], qui ont proposé une approche pour la recherche de code où chaque enseignant est spécialisé dans un langage de programmation. Cependant, nous nous différencions en introduisant le concept de coefficients de confiance, ce qui permet d'obtenir de meilleures performances, car tous les enseignants n'ont pas le même poids pendant l'entraînement. En effet, dans notre approche, les enseignants fournissent un signal d'entraînement proportionnel à la qualité de leurs prédictions. Shi et al. [**120**] ont adapté la distillation traditionnelle de Hinton et al. [**62**] pour compresser des modèles générant du code en un modèle de 3 méga-octets. Sau et al. [**118**] distillent les connaissances d'un enseignant en introduisant du bruit dans les prédictions de l'enseignant pour réduire le sur-apprentissage de l'étudiant. D'autres travaux ont exploré la distillation d'un ensemble d'enseignants en un modèle [**147**].

3.4. Apprentissage des représentations pour le code

Les représentations du code ont gagné une attention particulière au cours des récentes années [**34, 2**]. Deux tâches sont couramment utilisées pour évaluer la capacité d'abstraction apprise par les modèles d'apprentissage automatique. Il s'agit de la recherche sémantique de code [**68**] et de la détection de clones [**140**]. De nombreux travaux antérieurs se sont basés sur l'apprentissage auto-supervisé et contrastif pour obtenir des résultats impressionnants

sur ces deux tâches.

Tout d'abord, ContraCode [**69**], Corder [**22**], DISCO [**35**] ont introduit une approche contrastive utilisant des morceaux de code syntaxiquement différents, mais sémantiquement équivalents, respectivement en JavaScript et en Java. Cela est rendu possible grâce à des transformations de programmes basées sur des règles, telles que le renommage de variables, l'élimination de code mort et le précalcul de constantes, pour créer des augmentations de données. Code-MVP [**137**] utilise une approche similaire, mais y ajoute d'autres modalités comme l'arbre syntaxique (AST), les commentaires ou encore les graphes de flux de contrôle, mais uniquement pour le langage Python. OpenAI [**102**] se sert de l'objectif InfoNCE sur différentes tailles du modèle GPT-3 [**21, 24**], démontrant ainsi le pouvoir d'échelonnement des modèles, comme démontré auparavant [**74**].

Ensuite, CodeBERT [**41**] propose un modèle pré-entraîné basé sur BERT [**33**] qui se sert des données du *benchmark* CodeSearchNet [**68**] avec un objectif d'entraînement bimodal qui encode à la fois le code, mais aussi ses commentaires en langage naturel. L'objectif d'apprentissage de CodeBERT utilise à la fois la modélisation du langage masquée (MLM) et la détection de jetons remplacés (RTD). GraphCodeBERT [**56**] améliore CodeBERT en y ajoutant le graphe de flux de données en plus du code en entrée, mais aussi via une fonction de masquage guidée par le graphe. D'autres travaux récents ont continué d'améliorer l'apprentissage des représentations du code basés sur ces approches contrastives. SyncoBERT [**136**] propose un cadre d'apprentissage contrastif multimodal guidé par la syntaxe. En ajoutant deux nouveaux objectifs de pré-entraînement, à savoir la prédiction des identifiants et les arêtes dans l'arbre syntaxique du code (AST). Mais également, en ajoutant un objectif maximisant l'information mutuelle entre le code, les commentaires et l'arbre syntaxique. Plus récemment, UnixCoder [**55**] introduit un modèle pré-entraîné multimodal pour les langages de programmation. Celui-ci utilise un encodeur partagé entre toutes les modalités, mais les distingue avec des jetons spécifiques comme préfixe pour les données du modèle. CodeRetriever [**86**] mélange un objectif multimodal entre le code et sa documentation, ainsi qu'un objectif unimodal avec uniquement le code. Cette approche est un pré-entraînement contrastif qui utilise un autre modèle pré-entraîné pour trouver des codes similaires basés sur la correspondance entre leurs noms et leurs documentations, afin de créer des paires de codes similaires.

Finalement, SCodeR [**87**] se sert d'une approche contrastive utilisant les commentaires du code et leurs arbres syntaxiques pour créer des paires positives. À cela s'ajoute l'utilisation de deux discriminateurs qui prédisent la pertinence entre les paires positives et utilisent cela comme un signal d'entraînement pour créer des cibles souples.

3.5. Recherche de code traditionnelle

Avant l'avènement de l'apprentissage automatique, la recherche dans le domaine du génie logiciel s'intéressait depuis longtemps à la recherche de code, et la littérature sur ce sujet est riche [**115, 8, 93, 45, 68, 125, 98, 109**]. Plusieurs approches utilisaient des méthodes de recherche d'informations ou d'analyse de code. Par exemple, l'outil "grep" permet d'utiliser des expressions régulières et la correspondance de motifs pour rechercher dans les documents de code source [**76**]. D'autres travaux fondateurs, tels qu'Omega [**89**], CIA [**29**], CIA++ [**52**], Microscope [**5**], Rigi [**100**], et SCAN [**3**], utilisent une vue entité-relation-attribut d'un programme dans une vaste base de données de code source pour faire correspondre des requêtes à des noms de fonctions, d'appels ou de variables. Certains travaux séminaux, comme TXL [**31**], ASCENT [**46**], et REFINE [**79**] utilisent des outils de transformation de programmes pour faire correspondre, généralement via des arbres analysés, un langage dialectal en code source, transformant ainsi le problème de recherche de code en un problème de détection de clones. SCRUPLE [**107**] utilise une approche de correspondance de motifs avec des symboles dans une machine à états finis du code source. Sequoia [**15**] s'appuie sur la correspondance de motifs de graphes pour hiérarchiquement identifier des motifs de code de haut niveau dans le langage de programmation DOWTRAN. Additionnellement, certains travaux ont utilisé des modèles d'espace vectoriel se basant sur le comptage des occurrences de mots dans un document [**94, 117, 141**]. Des améliorations de ce genre de modèle ont été proposés en utilisant des mesures de similarités [**9, 75**] ou en liant les morceaux de codes par des termes similaires trouvés sur des forums en ligne [**77, 124**]. D'autres ont proposé une approche de recherche basé sur le calcul d'empreintes du code [**43, 114**]. Finalement, de récents travaux [**109**] utilisent le raisonnement équationnel en considérant l'ensemble des graphes de flux de données pour des fonctions équivalentes en utilisant un ensemble de règles de ré-écriture.

3.6. Détection de clones traditionnelle

Tout comme la recherche de code, la détection de clones n'a pas attendu l'émergence des réseaux neuronaux pour devenir un domaine d'intérêt au sein de la communauté du génie logiciel. La communauté a principalement utilisé des techniques telles que l'analyse de code et de texte, la recherche de motifs, et l'extraction d'informations pour aborder cette problématique. Certaines recherches antérieures [**19, 53, 95**] ont exploité des métriques logicielles, telles que les mesures de complexité de Halstead, pour établir des correspondances entre des morceaux de code. D'autres chercheurs [**70**] ont opté pour l'utilisation d'arbres d'exécution statique afin de créer des empreintes digitales des programmes, ce qui leur permet de déterminer la similitude entre deux programmes en se basant sur ces empreintes. Tamer [**64**] propose une méthode de détection de clones syntaxiques en se penchant sur une analyse

minutieuse de l'arbre syntaxique du code. D'autres chercheurs ont préféré s'appuyer sur des outils d'analyse de texte [**50, 72, 73, 91, 113, 134**] ou d'analyse de jetons [**51, 67, 101, 116, 133**] pour détecter les clones. Alors que certains se basent sur une transformation du code en chaînes de caractères pour calculer la similarité, d'autres extraient les jetons de code à l'aide d'un analyseur lexical pour effectuer cette comparaison. L'outil de détection de clones Dup [**10**] repose sur la représentation du code sous forme de séquences de lignes et identifie les clones en remplaçant les identificateurs de fonctions, de variables et de types par des identifiants spéciaux. En revanche, Duploc [**36**] élimine tous les espaces et les commentaires du code, puis détecte les clones en cherchant des motifs dynamiques au sein des chaînes de caractères du code source. D'autres chercheurs [**16**] proposent de détecter les clones en comparant des métriques telles que les hachages de sous-arbres des arbres syntaxiques initiaux du code source. Des métriques sont également employées pour comparer les clones au niveau des fonctions dans le cadre d'une représentation intermédiaire du code appelée langage de représentation intermédiaire (IRL) [**97**]. Enfin, SMC [**11**] adopte une approche combinée en utilisant à la fois des métriques et la détection de motifs dynamiques pour identifier les clones.

Chapitre 4

Pré-Entrainement Non-Contrastif pour les Représentations du Code

Vue d'ensemble

Cette section décrit `rncp-code`, une méthode de pré-entraînement pour apprendre des représentations du code basée sur VICReg [**14**], une méthode non-contrastive. Cette méthode a été conçue dans le but d'améliorer la capture de la sémantique fonctionnelle du code.

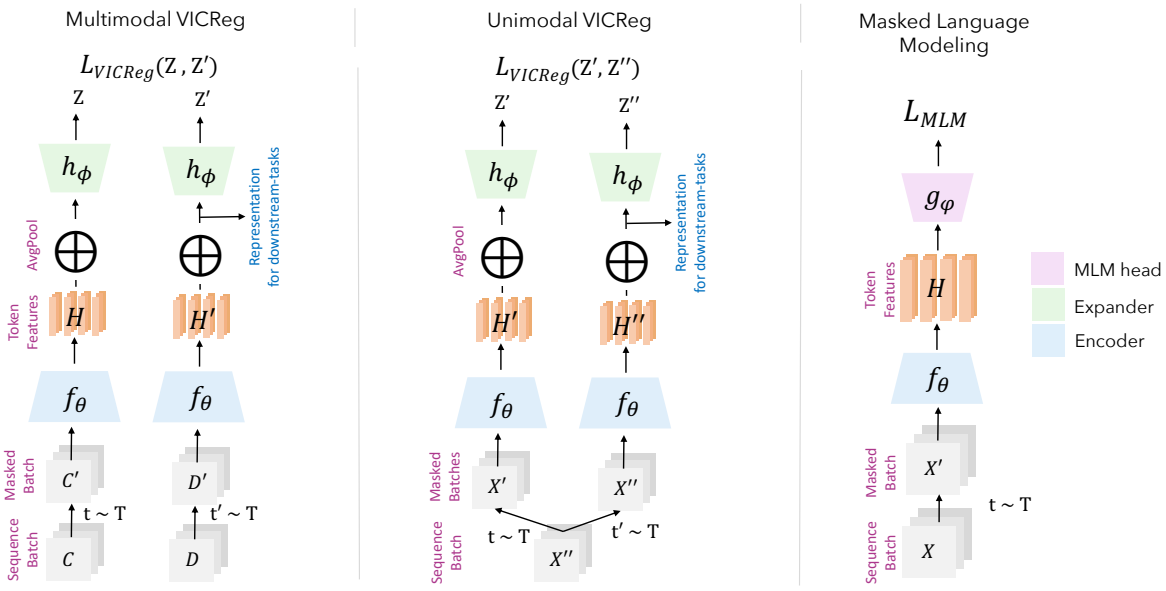


Fig. 4.1. `rncp-code` Illustration de l'architecture.

4.1. Approche

Dans notre méthode, nous avons utilisé une architecture de représentations jointes (JEA). Pour apprendre de bonnes représentations, nous avons utilisé l'architecture RoBERTA [**90**] comme encodeur pour la transformation des données brutes en représentations. Notre méthode se base principalement sur une approche non-contrastive régularisée, en plus de l'objectif par défaut des modèles de langues, à savoir la modélisation du langage masquée (MLM). Dans la suite de ce chapitre, nous utiliserons le nom de `rncp-code` pour parler de notre architecture.

Le but de `rncp-code` est d'apprendre des représentations invariantes à la syntaxe afin de se focaliser sur la sémantique fonctionnelle, comme discuté à la section 1.2. Pour ce faire, nous avons construit des paires d'exemples positifs en utilisant le code et sa documentation. Afin d'augmenter le nombre de données à notre disposition, nous avons masqué des parties du code et de sa documentation à chaque étape d'entraînement, mais nous avons également échangé aléatoirement les modalités entre les deux encodeurs. À l'aide de ces différentes astuces, nous espérions obtenir de meilleures performances. Le reste de ce chapitre sera consacré à détailler notre objectif de pré-entraînement, la manière dont les augmentations de données ont été appliquées, les résultats de l'évaluation de l'entrainement sur les taches de recherche de code et de détection de clones, ainsi qu'une discussion sur de potentielles hypothèses d'améliorations de la méthode. La figure 4.1 fournit une vision d'ensemble de l'approche considérée.

4.1.1. Modèle

Notre approche se base sur une architecture de représentations jointes (JEA). Étant donné une paire de code et sa documentation (x,y) échantillonnée depuis un jeu de données $\mathcal{D}$, deux transformations t et t' sont échantillonnées depuis une distribution $\mathcal{T}$ et appliquées de telle manière que $x' = t(x)$ et $y' = t'(y)$, créant ainsi une vue augmentée (x', y') de la paire originale. Notez que $\mathcal{T}$ représente la distribution de toutes les transformations possibles et sera détaillée dans la section 4.1.2. Par la suite, chaque séquence transformée de la paire est encodée par une branche de l'architecture.

Tout d'abord, chaque représentation est créée par l'encodage des séquences $v_x = f_\theta(x')$ et $v_y = f'_{\theta'}(y')$. Ensuite, chaque représentation est projetée dans un espace à très haute dimension par des extenseurs h_ϕ et $h'_{\phi'}$, de telle sorte que $\mathbf{z}_x = h_\phi(\mathbf{v}_x)$ et $\mathbf{z}_y = h'_{\phi'}(\mathbf{v}_y)$. Une fois les représentations étendues, l'objectif de VICReg [**14**] peut être calculé comme décrit à la section 4.1.3. Finalement, les représentations pré-encodées $v_x = f_\theta(x')$ et $v_y = f'_{\theta'}(y')$ sont

décodées par une tête de modélisation du langage g_ϕ afin de compléter les parties masquées du code et de la documentation.

4.1.2. Augmentation de données

À l'inverse des méthodes contrastives, les non-contrastives dépendent uniquement des paires d'exemples positifs. D'une part, dans un régime unimodal, cela est typiquement fait en augmentant un exemple en deux vues différentes, par exemple une photo de chien augmentée en ajoutant du bruit dans la colorimétrie de l'image. D'autre part, dans un régime multimodal, plusieurs perspectives du même objet sémantique sont déjà disponibles, car chaque modalité agit comme une vue spécifique de la donnée d'entrée, par exemple une photo de chien en train de jouer avec une balle et une description textuelle de l'image.

Des travaux antérieurs [**69, 35**] ont considéré une approche unimodale, utilisant uniquement le code source comme entrée pour apprendre les représentations. Cependant, générer des variations du code source requiert de construire son propre parseur spécifique au langage utilisé pour appliquer des transformations comme le renommage de variable, l'élimination de code mort, le pré-calcul de constante et le formatage du code. En conséquence, ces travaux n'ont utilisé qu'un seul langage de programmation pour leurs expériences.

D'autres travaux séminal ont considéré des approches multimodales, mais toujours en utilisant des parseurs spécifiques aux langages utilisés pour extraire des modalités comme l'arbre syntaxique (AST) ou le graphe de flux de contrôle [**55, 137**]. L'utilisation de tels parseurs peut être très coûteuse en termes de temps et de ressources, et n'est pas extensible en fonction du nombre de langages de programmation considérés.

Pour cette raison et guidés par de récents travaux [**86, 102**], nous soutenons qu'une approche multimodale, combinant à la fois le code et sa documentation, est une méthode plus efficace pour l'apprentissage de bonnes représentations dans le contexte des JEA. En effet, cette approche offre différentes perspectives d'un programme sans utiliser de parseurs, permettant ainsi de gérer différents langages de programmation sans avoir besoin de créer un parseur différent par langage. Afin d'améliorer encore davantage la capacité de l'encodeur à apprendre à être invariant à la syntaxe, nous avons augmenté les paires de code et de documentation via un masquage dynamique des jetons. Plus spécifiquement, étant donné une paire code-documentation, nous générons une version masquée de cette dernière en remplaçant un pourcentage des jetons de la séquence par un jeton `[MASK]` ou aléatoire.

4.1.3. Objectif de pré-entrainement

Dans le domaine de l'apprentissage des représentations pour le code, une pratique courante est l'utilisation de l'objectif contrastif InfoNCE pendant l'entraînement [**69, 22, 35, 102, 86, 87**]. Cependant, des travaux récents [**71**] ont montré que cet objectif pouvait conduire à l'*effondrement dimensionnel*, c'est-à-dire que les représentations apprises ne couvrent qu'un sous-espace vectoriel de l'espace des représentations possibles. Cela peut sérieusement limiter l'expressivité des représentations apprises. En revanche, les méthodes non contrastives régularisées (RNC) ont démontré un véritable potentiel pour réduire l'*effondrement dimensionnel* en couvrant un sous-espace vectoriel plus étendu de l'espace des représentations.

Pour ces raisons, nous proposons d'utiliser un objectif non contrastif régularisé, VICReg (*Variance-Invariance-Covariance Régularisation*), pour pré-entraîner notre encodeur. [**14**]

4.1.3.1. VICReg objectif. VICReg repose uniquement sur l'utilisation de pairs d'exemples positifs et vise à garder les représentations de ces pairs similaires tout en éviter l'effondrement via des termes de régularisations appliqués sur chaque représentation séparément. Les buts de ces termes sont (1) Rendre les représentations invariant à la vue d'entrée. (2) Maintenir la variance des dimensions pour chaque représentation au-dessus d'un certain seuil. (3) Décorréler chaque dimension de chaque représentation.

En utilisant la formalisation introduite dans le papier original [**14**], chaque représentation des modalités est traitée par paquet dénoté $Z^{(x)} = [z_1^{(x)}, \ldots, z_n^{(x)}]$ and $Z^{(y)} = [z_1^{(y)}, \ldots, z_n^{(y)}]$ pour les représentations du code et de la documentation respectivement. Chaque paquet contient n vecteur $z_i \in \mathbb{R}^d$ (avec $i \in [n]$ est l'indice du vecteur dans le paquet) résultant d'une passe avant à travers chaque encodeur et expanseurs de l'architecture à représentations jointes. L'objectif d'apprentissage de VICReg est défini comme la somme des trois termes de régularisations, à savoir le terme de variance, de covariance et d'invariance.

Le terme de régularisation de la variance, dénoté $v(Z)$, est défini comme :

$$v(Z) = \frac{1}{d}\sum_{j=1}^{d} \max(0, 1 - S(z^j, \epsilon)), \tag{4.1.1}$$

où $S(x, \epsilon) = \sqrt{\mathrm{Var}(x) + \epsilon}$ est l'écart type régularisée, d est la dimension du vecteur de représentation et ϵ une petite constante qui aide à stabilité numérique. Le but de ce terme de régularisation est de garder la variance entre chaque dimension d'une représentation aussi proche possible de 1, ce qui empêche l'*effondrement complet*, c'est-à-dire avoir tous les

entrée transformé en le même vecteur de représentation.

Ensuite, le terme de régularisation de la covariance, dénoté par $c(Z)$, est défini par :

$$c(Z) = \frac{1}{d}\sum_{i\neq j}[C(Z)]_{i,j}^2, \tag{4.1.2}$$

où $C(Z)$ est la matrice de covariance de Z, défini comme :

$$C(Z) = \frac{1}{n-1}\sum_{i=1}^{n}(z_i - \bar{z})(z_i - \bar{z})^T, \tag{4.1.3}$$

où $\bar{z} = \frac{1}{n}\sum_{i=1}^{n} z_i$, est le vecteur moyen de Z parmi les vecteurs lignes. Le terme de régularisation de la covariance favorise la décorrélation entre les différentes dimensions des représentations.

Finalement, le terme de régularisation de l'invariance, dénoté par $i(Z, Z')$ et défini comme:

$$i(Z^{(x)}, Z^{(y)}) = \frac{1}{d}\sum_{j=1}^{d}(z_j^{(x)} - z_j^{(y)})^2, \tag{4.1.4}$$

où les paquets $Z^{(x)}$ et $Z^{(y)}$ sont produits par des branches distinctes de l'architecture (cf. Fig. 4.1). Le terme d'invariance favorise différentes représentations d'un même objet sémantique à être similaire, encourageant donc les représentations à être invariant à la syntaxe du code.

L'objectif complet de VICReg peut être écrit comme :

$$\ell_{VICReg}(Z^{(x)}, Z^{(y)}) = \lambda i(Z^{(x)}, Z^{(y)}) + \mu[v(Z^{(x)}) + v(Z^{(y)})] + \nu[c(Z^{(x)}) + c(Z^{(y)})], \tag{4.1.5}$$

où λ, μ et ν sont des hyper-paramètres contrôlant l'importance de chaque terme de la fonction d'objectif.

4.1.3.2. Objectif de modélisation du langage masqué. Additionnellement à VICReg, nous utilisons également l'objectif traditionnel de modélisation du langage masqué (MLM), c'est-à-dire prédire ce qui se trouve derrière les jetons `[MASK]` en utilisant l'entropie croisée comme objectif. Cela encourage les encodeurs à compléter leurs vues partielles des données d'entrée en se basant sur leurs représentations partagées. L'objectif peut être défini comme suit :

$$\ell_{MLM}(X^{\text{mask}}) = -\sum_{x_i \in S_m} \log p(x_i | X^{\text{mask}}), \tag{4.1.6}$$

où S_m est la distribution softmax sur les sorties du décodeur et X^{mask} est une version masquée du paquet de séquences X où les jetons `[MASK]` ont été appliqués dynamiquement avec 15% de chance.

Notre objectif global de pré-entraînement $\mathcal{L}$ est la somme des objectifs VICReg et MLM et peut-être écrit comme suit :

$$\mathcal{L} = \sum_{(X,Y)\in\mathcal{D}} \sum_{t,t'\sim\mathcal{T}} \ell_{VICReg}(Z^X, Z^Y) + [\ell_{MLM}(X^{mask}) + \ell_{MLM}(Y^{mask})], \tag{4.1.7}$$

où Z^X et Z^Y sont des paquets de représentations correspondant au paquet de code X et à leur documentation Y transformés par t et t'. $\mathcal{T}$ représente la distribution des transformations possibles. Cet objectif est minimisé par rapport aux paramètres des encodeurs θ, θ', aux paramètres des expanseurs ϕ, ϕ', et aux paramètres des décodeurs ψ, ψ' pendant plusieurs époques. Une vue d'ensemble illustrée de l'objectif et de l'architecture est représentée à la Figure 4.1.

Il convient de noter que, bien que l'InfoNCE soit largement utilisée pour l'apprentissage des représentations du code, l'utilisation du critère RNC pourrait offrir des avantages significatifs par rapport aux méthodes d'apprentissage contrastives, comme exprimé par [**82**]. Cependant, bien qu'ils soient souvent considérés comme des objectifs concurrents, il est possible d'utiliser à la fois les objectifs contrastifs et RNC, ce qui serait une piste intéressante à explorer.

4.2. Expériences & Résultats

Comme mentionné précédemment, cf. section 1.2, la recherche de code et la détection de clones de type 4 constituent deux tâches souvent utilisées pour évaluer la capacité des modèles à capturer la sémantique fonctionnelle des modèles de code. En effet, elles nécessitent un minimum de compréhension de la sémantique du code pour pouvoir faire le lien entre le code et une requête en langage naturel ou entre un autre morceau de code syntaxiquement différent. Comme nos pairs, nous avons donc décidé d'utiliser ces tâches pour évaluer le modèle entraîné avec notre méthode. Cette section décrit et interprète donc les résultats obtenus. Pour chaque tâche, l'encodeur utilisé est un *transformer* [**132**] correspondant exactement à l'architecture utilisé par UnixCoder [**55**] (12 couches de *Transformer* avec des états cachés de dimensions 768 et 12 têtes d'attentions) pré-entrainé avec l'objectif de VICReg [**14**] pendant 100 époques sur des données issues de répertoire à source ouvert provenant de Github et avec $\lambda = 25, \mu = 25, \nu = 1$ comme valeurs pour les coefficients pondérant les différents termes de régularisations de l'objectif de pré-entrainement décrit à la section 4.1.3. À noter que toutes les expériences ont été réalisées sur un serveur contenant quatre cartes graphiques NVIDIA GeForce RTX 3090 chacune avec 24 Go de mémoire vives et avec le langage python via les bibliothèques PyTorch [**106**] et HuggingFace [**142**] pour l'implémentation des modèles. Pour les hyper-paramètres du pré-entrainement, ils sont identiques à ceux décrits pour l'encodeur de l'article de VICReg [**14**]. Pour toutes les tâches d'évaluation, les hyper-paramètres pour le peaufinage sont identiques et sont les suivants : le nombre d'époques est de 10, la taille de la séquence de code est de 256 et 128 pour le texte, la taille des paquets de données pour

l'entrainement et l'évaluation sont de 64, le taux d'apprentissage est de 0,00002. Il est aussi important de noter que le meilleur modèle est sélectionné après chaque époque et sur base de ses performances sur le jeu de validation en fonction de la métrique d'évaluation de la tâche (MRR, précision, rappel, MAP, etc.).

4.2.1. Recherche de code

Comme mentionné dans la section 2.2.1, la recherche de code consiste à trouver les morceaux de code les plus similaires dans une base de données étant donné une requête en langage naturel. Pour la recherche de code, nous avons décidé de l'évaluer sur trois différents *benchmarks* largement utilisés au sein de la communauté de la recherche de code, à savoir CosQA [**65**], Adv [**92**] et CodeSearchNet [**68**]. Nous avons utilisé le rang réciproque moyen (MRR) comme métrique d'évaluation pour ces tâches. Cette dernière permet de quantifier la capacité d'un système de recherche d'informations à proposer des informations pertinentes en premier dans une liste de résultats. Le MRR est une valeur comprise entre 0 et 100 où 100 correspond au meilleur scénario, c'est-à-dire proposer la meilleure information en premier résultat et 0 est le pire scénario, car l'information est proposée en dernier résulat. La table 4.1 rapporte les résultats pour chacun des jeux de données. Il est intéressant de noter que l'utilité d'évaluer sur CosQA et Adv est que ces jeux de données ont été créés explicitement pour être plus difficiles, car plus proches de données que l'on trouve dans la réalité. Ce qui explique la différence de MRR entre ces derniers et CodeSearchNet. À noter que les résultats comparés autres que ceux du `rncp-code` sont rapportés depuis des articles précédents.

Nous avons également considéré une tache de recherche de codes multilingue où les requêtes sont des morceaux de codes provenant d'autres langages. Pour ce faire, nous avons réutilisé les données fournies dans UnixCoder [**55**] se basant sur le jeu de données CodeNet [**110**]. Comme UnixCoder, nous avons rapporté la précision moyenne (MAP) pour chaque langage.

4.2.1.1. CodeSearchNet. Le jeu de données CodeSearchNet (CSN) contient 6 langages de programmation différents, à savoir Java, JavaScript, Go, Python, PHP et Ruby. La valeur rapportée dans la table 4.1 pour CSN est la moyenne parmi tous les langages. Une version détaillée du MRR pour chaque langage est décrite dans le Tableau 4.2. Les performances indiquées sont celles obtenues après peaufinage du modèle sur un langage. À noter que la manière dont les modèles ont été peaufinés est celle décrite à la section 2.2.1. On peut voir que `rncp-code` peine à atteindre les performances des modèles à l'état de l'art. Nous fournissons des hypothèses d'améliorations à la section 4.3. Les résultats comparés autres que ceux du `rncp-code` sont rapportés depuis des articles précédents.

	CosQA	AdvTest	CSN
RoBERTa [**90**]	-	-	61.7
CodeBERT [**40**]	64.7	27.2	69.3
GraphCodeBERT [**56**]	67.5	35.2	71.3
SyncoBERT [**136**]	-	38.1	74.0
UniXcoder [**55**]	70.1	41.3	74.4
SCodeR [**87**]	74.5	45.5	74.5
CodeRetriever [**86**]	**75.4**	**46.9**	**77.4**
`rncp-code` (notre méthode)	55.4	27.6	61.95

Tableau 4.1. Performance MRR (le plus élevé est le mieux) de `rncp-code` sur les jeux de données de recherche de codes CosQA, AdvTest et CodeSearchNet.

4.2.1.2. Recherche de codes multilingues en zéro-coup. Afin de quantifier plus en détail la capacité de notre approche à capturer la sémantique fonctionnelle, nous avons repris une expérience proposée par les créateurs d'UnixCoder. Le but de l'expérience est de retrouver les codes les plus similaires à un code fourni en requête au préalable. Bien que l'approche reste identique à celle décrite à la section 2.2.1, il est important de noter que dans cette expérience, il n'y a aucun peaufinage sur les données, mais directement une évaluation, ce que l'on appelle une approche zéro-coup. L'expérience prend une requête dans un langage parmi Python, Java et Ruby et recherche les candidats les plus similaires parmi une base de données de code dans ces mêmes langages. Il y a respectivement 11,744/15,594/23,530 morceaux de code écrits en Ruby/Python/Java. Ces candidats sont extraits du jeu de données CodeNet [**110**]. La table 4.3 rapporte la précision moyenne (MAP) pour chaque langage source et les différents langages cibles. On peut voir que `rncp-code` peine à atteindre les performances des meilleurs modèles. Cependant, il dépasse celles de CodeBERT [**40**], qui est le seul modèle autre que notre approche à ne pas être pré-entraîné avec des graphes. Cela pourrait donc fournir une piste sur l'importance de pré-entraîner les modèles de codes avec leurs arbres syntaxiques (AST) ou leurs graphes de flux de données comme modalités.

4.2.2. Détection de clone

Comme mentionné dans la section 2.2.2, la détection de clones est un problème de classification qui consiste à déterminer si deux morceaux de code ont la même sémantique fonctionnelle. Dans le domaine de la détection de clones, deux jeux de données sont majoritairement utilisés pour évaluer les modèles. Il s'agit de BigCloneBench [**127**] et POJ-104 [**99**]. Ces jeux de données ne contiennent que des morceaux de code écrits en Java.

Pour BigCloneBench, nous avons rapporté la précision, le rappel et le score F1 de chaque modèle. Pour POJ-104, nous avons utilisé la précision moyenne parmi R échantillons

	Ruby	**JavaScript**	**Go**	**Python**	**Java**	**PHP**	**Moy.**
RoBERTa [**90**]	58.7	51.7	85.0	58.7	59.9	56.0	61.7
ContraCode [**69**]	-	30.6	-	-	-	-	-
CodeBERT [**40**]	67.9	62.0	88.2	67.2	67.6	62.8	69.3
GraphCodeBERT [**56**]	70.3	64.4	89.7	69.2	69.1	64.9	71.3
SynCoBERT [**136**]	72.2	67.7	91.3	72.4	72.3	67.8	74.0
UniXcoder [**55**]	74.0	68.4	91.5	72.0	72.6	67.6	74.4
SCodeR [**87**]	**77.5**	**72.0**	**92.7**	74.2	74.8	69.2	76.7
CodeRetriever [**86**]	77.1	71.9	92.4	**75.8**	**76.5**	**70.8**	**77.4**
`rncp-code` (notre méthode)	64.7	54.1	86.5	60.3	59.3	54.4	61.95

Tableau 4.2. Performance MRR (le plus élevé est le mieux) de `rncp-code` sur le jeu de données CodeSearchNet sur six langages de programmation.

Source	**Ruby**			**Python**			**Java**			**Moy.**
Cible	Ruby	Python	Java	Ruby	Python	Java	Ruby	Python	Java	
CodeBERT [**40**]	13.55	3.18	0.71	3.12	14.39	0.96	0.55	0.42	7.62	4.94
GraphCodeBERT [**56**]	17.01	9.29	6.38	5.01	19.34	6.92	1.77	3.50	13.31	9.17
PLBART [**1**]	18.60	10.76	1.90	8.27	19.55	1.98	1.47	1.27	10.41	8.25
CodeT5-base [**138**]	18.22	10.02	1.81	8.74	17.83	1.58	1.13	0.81	10.18	7.81
UniXcoder [**55**]	**29.05**	**26.36**	**15.16**	**23.96**	**30.15**	**15.07**	**13.61**	**14.53**	**16.12**	**20.45**
`rncp-code` (notre méthode)	15.36	7.7	0.76	4.89	14.22	0.97	1.39	1.03	6.54	5.87

Tableau 4.3. Score MAP (%) (le plus élevé est le mieux) de `rncp-code` pour une tache de recherche code-à-code dans un contexte de zéro-coup.

(MAP@R) comme métrique d'évaluation. À noter que les résultats comparés, autres que ceux du `rncp-code`, sont rapportés depuis des articles précédents. La table 4.4 détaille les performances obtenues avec notre approche. On peut voir que `rncp-code` obtient des performances très maigres pour POJ-104, avec des résultats inférieurs à ceux de RoBERTA [**90**], qui est un modèle pré-entraîné uniquement avec du langage naturel. Cela dit, pour BigCloneBench, nous obtenons des performances légèrement inférieures à celles de CodeBERT, mais avec une légère meilleure précision.

Nous avons également rapporté une évaluation qualitative des clones trouvés dans CodeSearchNet en utilisant un morceau de code dans un certain langage comme requête.

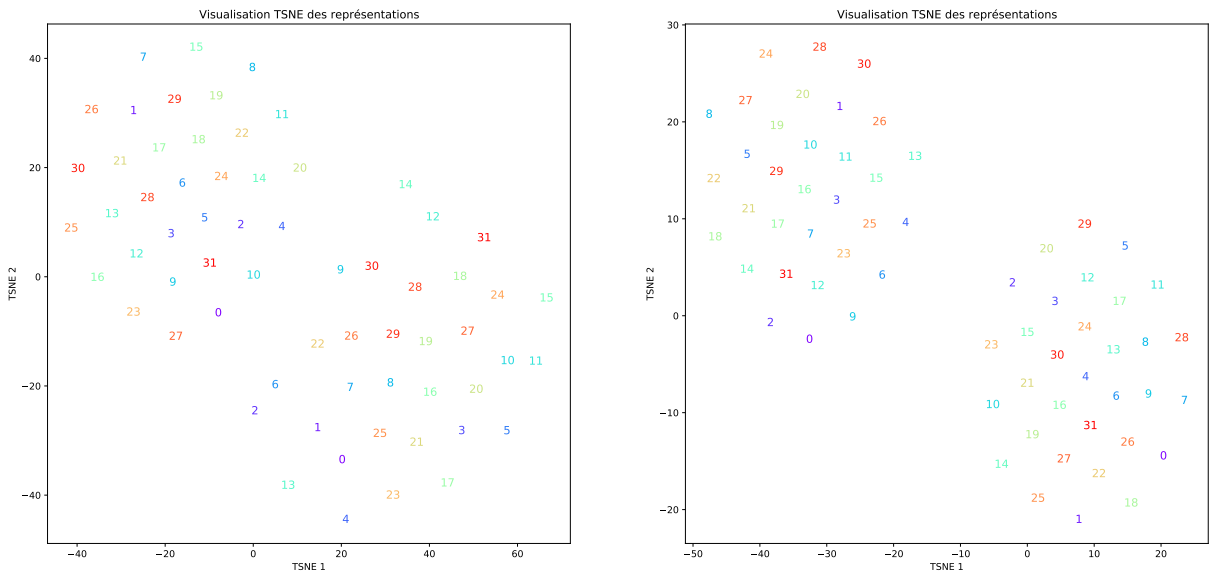
	POJ-104	**BigCloneBench**		
	MAP@R	Rappel	Précision	F1-score
RoBERTA [**90**]	76.67	95.1	87.6	91.3
CodeBERT [**40**]	82.67	94.7	93.4	94.1
GraphCodeBERT [**56**]	85.16	94.8	95.2	95.0
SYNCOBERT [**136**]	88.24	-	-	-
UniXcoder [**55**]	90.52	92.9	**97.6**	95.2
CodeRetriever [**86**]	88.85	-	-	-
SCodeR [**87**]	**92.45**	**96.2**	94.5	**95.3**
`rncp-code` (notre méthode)	73.78	94.0	94.0	94.0

Tableau 4.4. Résultats de `rncp-code` pour la détection de clone sur POJ-104 et BigCloneBench.

4.2.3. Évaluation qualitative

Après avoir réalisé une évaluation comparative quantitative de notre approche par rapport à l'état de l'art, nous avons également souhaité évaluer qualitativement nos méthodes. À cet effet, nous avons effectué une comparaison basée sur la similarité entre des paires de code et de documentation, en utilisant des visualisations t-SNE [**131**] pour UniXCoder et notre approche, comme illustré à la figure 4.2. Cela nous permet de visualiser des représentations en hautes dimensions dans des dimensions plus petites, ici 2, tout en préservant au maximum les relations et la structure entre ces dernières. Il est intéressant de noter que pour les deux méthodes, nous observons deux groupes distincts de représentations : l'un pour le code et l'autre pour la documentation.

En plus de cette évaluation qualitative, nous avons également évalué nos approches en identifiant les trois meilleurs morceaux de code proposés, classés par ordre de similarité, pour une requête spécifique. Dans notre cas, la requête était "Comment faire le tri à bulle". Cette



(a) UnixCoder (b) `rncp-code`

Fig. 4.2. Comparaison des graphiques t-SNE [**131**] des représentations de 30 paires code-documentation pour Unixcoder (à gauche) et notre approche (à droite). Une paire partage le même numéro pour sa documentation et son code.

évaluation nous a permis de comparer les performances et la pertinence de nos méthodes en termes de suggestions de code pertinents et adaptés à la requête donnée. La figure 4.3 démontre que UniXCoder retourne des résultats plus pertinent que notre approche pour cette requête, même si `rncp-code` comprend que la requête parle de tri.

4.3. Discussion

Malgré une idée conceptuelle prometteuse, la pratique semble tout autre. En effet, les résultats des expériences rapportés semblent indiquer que les méthodes non-contrastives régularisées ne sont pas la bonne solution pour pré-entraîner des modèles de code afin d'apprendre des représentations capturant des informations hautement abstraites comme la sémantique fonctionnelle. Cependant, nous émettons l'hypothèse que le problème ne provient pas de la méthode, qui a déjà fait ses preuves [**14**], mais d'autres facteurs qui peuvent causer des limites dans la qualité du pré-entraînement. Cette section vise donc à détailler nos hypothèses pour l'amélioration de `rncp-code`.

4.3.1. Hypothèses d'améliorations

4.3.1.1. Augmentation de données. La plus grosse hypothèse d'amélioration de notre travail est, sans aucun doute, l'algorithme d'augmentations de données. Nous pensons qu'il est largement préférable d'utiliser une méthode universelle pour l'augmentation des données de

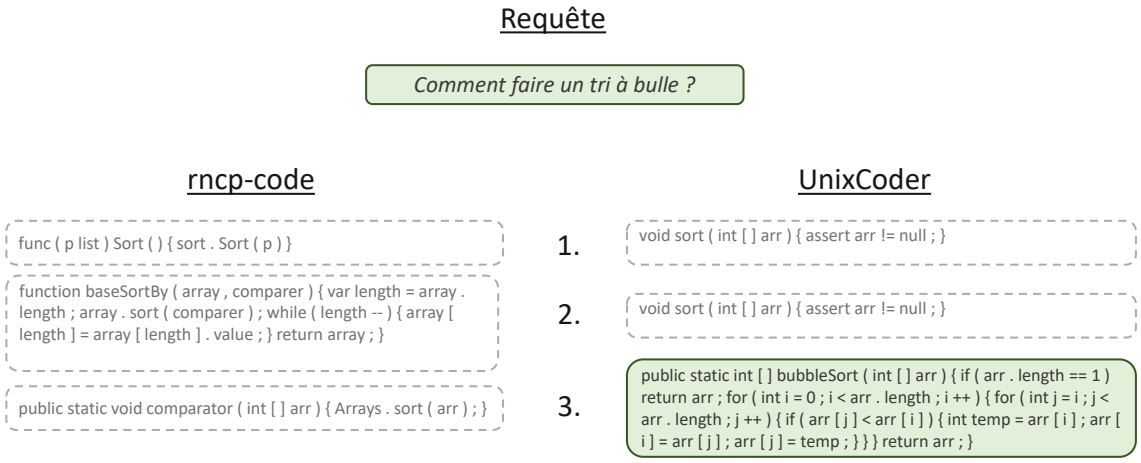


Fig. 4.3. Comparaison des trois meilleurs codes trouvés dans CodeSearchNet pour `rncp-code` et UnixCoder étant donné la requête *"Comment faire un tri à bulle ?"*.

code plutôt que d'élaborer un parseur par langage [**69, 35**]. En effet, les images et le texte ont déjà des transformations qui s'adaptent à n'importe quelle image et langage respectivement [**54, 122, 39**]. Le cas du code est un peu plus complexe, car à l'inverse du texte, certaines opérations faciles comme l'échange de la position de deux mots ne changent souvent pas le sens d'une phrase, ce qui n'est pas le cas du code qui est très dépendant de sa structure [**148**].

Notre première idée pour augmenter n'importe quelle donnée de code est de demander à ChatGPT de générer des codes sémantiquement équivalents au programme qu'on lui fournit en entrée, que ce soit dans d'autres langages ou dans le même. Cette façon d'augmenter les données fonctionne assez bien, mais est très coûteuse financièrement et temporellement, car l'API d'OpenAI est très lente pour générer les codes alternatives. Nous n'avons donc pas pu tester à nouveau `rncp-code` avec cette manière de faire de l'augmentation de données. Une illustration de la qualité de l'augmentation produite par ChatGPT est représentée par les figures 4.4, 4.5, 4.6, 4.7, 4.8 et 4.9. Notre autre idée se base sur de récents travaux visant à augmenter les données de code directement dans l'espace des représentations [**84**]. Cela permet d'avoir une méthode qui n'est pas dépendante du langage d'origine et peu coûteuse.

Tanslate the following program in 1. ruby, 2. java, 3. php, 4. go, 5. javascript. Only output the resulting codes without any explanation.

```python
def bubbleSort(arr):
    n = len(arr)
    swapped = False
    for i in range(n-1):
        for j in range(0, n-i-1):
            if arr[j] > arr[j + 1]:
                swapped = True
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

        if not swapped:
            return
```

Fig. 4.4. Illustration d'une requête à ChatGPT pour l'augmentation d'une fonction en python qui trie une liste avec le tri à bulle en différents langages de programmations.

1. Ruby:

```ruby
def bubble_sort(arr)
  n = arr.length
  swapped = false
  for i in 0..n-2
    for j in 0..n-i-2
      if arr[j] > arr[j + 1]
        swapped = true
        arr[j], arr[j + 1] = arr[j + 1], arr[j]
      end
    end

    if !swapped
      return
    end
  end
end
```

Fig. 4.5. Illustration de l'augmentation, par ChatGPT, de la requête illustré à la figure 4.4 en Ruby.

2. Java:

```java
public class BubbleSort {
  public static void bubbleSort(int[] arr) {
    int n = arr.length;
    boolean swapped = false;
    for (int i = 0; i < n-1; i++) {
      for (int j = 0; j < n-i-1; j++) {
        if (arr[j] > arr[j + 1]) {
          swapped = true;
          int temp = arr[j];
          arr[j] = arr[j + 1];
          arr[j + 1] = temp;
        }
      }

      if (!swapped) {
        return;
      }
    }
  }
}
```

Fig. 4.6. Illustration de l'augmentation, par ChatGPT, de la requête illustré à la figure 4.4 en Java.

3. PHP:

```php
function bubbleSort($arr) {
  $n = count($arr);
  $swapped = false;
  for ($i = 0; $i < $n-1; $i++) {
    for ($j = 0; $j < $n-$i-1; $j++) {
      if ($arr[$j] > $arr[$j + 1]) {
        $swapped = true;
        list($arr[$j], $arr[$j + 1]) = [$arr[$j + 1], $arr[$j]];
      }
    }

    if (!$swapped) {
      return;
    }
  }
}
```

Fig. 4.7. Illustration de l'augmentation, par ChatGPT, de la requête illustré à la figure 4.4 en PHP.

4. Go:

```go
func bubbleSort(arr []int) {
  n := len(arr)
  swapped := false
  for i := 0; i < n-1; i++ {
    for j := 0; j < n-i-1; j++ {
      if arr[j] > arr[j+1] {
        swapped = true
        arr[j], arr[j+1] = arr[j+1], arr[j]
      }
    }

    if !swapped {
      return
    }
  }
}
```

Fig. 4.8. Illustration de l'augmentation, par ChatGPT, de la requête illustré à la figure 4.4 en Go.

5. JavaScript:

```javascript
function bubbleSort(arr) {
  var n = arr.length;
  var swapped = false;
  for (var i = 0; i < n-1; i++) {
    for (var j = 0; j < n-i-1; j++) {
      if (arr[j] > arr[j + 1]) {
        swapped = true;
        [arr[j], arr[j + 1]] = [arr[j + 1], arr[j]];
      }
    }

    if (!swapped) {
      return;
    }
  }
}
```

Fig. 4.9. Illustration de l'augmentation, par ChatGPT, de la requête illustré à la figure 4.4 en JavaScript.

4.3.1.2. Qualité des données. De récents travaux ont démontré la nécessité d'avoir des jeux de données d'entraînement de qualité pour l'entraînement de systèmes neuronaux à la recherche de code [**126**]. Ces mêmes travaux ont montré que les commentaires de CodeSearchNet, utilisés comme requêtes, contenaient beaucoup de bruit, limitant la qualité de l'entraînement. De plus, CodeSearchNet ne contient que 2,8 millions de codes et leurs documentations, ce qui est relativement peu pour le pré-entraînement d'une méthode comme VICReg sur une architecture aussi imposante que les *Transformers*. Nous pensons donc qu'avec un jeu de données plus conséquent et de meilleure qualité, l'efficacité du pré-entraînement de `rncp-code` serait grandement accrue.

4.3.1.3. Architecture du modèle. Le papier original sur lequel notre idée s'appuie, VICReg [**14**], n'utilise pas de *Transformers* [**132**] comme architecture pour leur modèle pré-entraîné, mais plutôt différentes tailles de *ResNet* [**59**]. Ce choix a son importance, car chaque type d'architecture en apprentissage profond possède des biais inductifs différents, ce qui conduit à des résultats d'apprentissage différents pour un même jeu de données [**145**]. Une piste d'amélioration pour notre approche serait donc d'adopter une architecture avec des biais inductifs plus spécifiques au code. Cela pourrait inclure un entraînement avec plus d'époques, une taille de paquet plus grande et une taille de prédicteur plus large.

4.3.2. Lien avec la co-distillation

Notre approche pourrait être vue comme une forme de co-distillation [**6, 23, 28**]. Cette dernière consiste à former plusieurs réseaux d'étudiants pour qu'ils s'améliorent iterativement mutuellement sur la base de leurs propres connaissances. Dans notre approche, nous avons également deux réseaux d'étudiants qui apprennent différentes vues du même contenu sémantique en utilisant l'augmentation des données. Ils cherchent à améliorer leur compréhension en se mettant d'accord sur leurs connaissances partagées et en complétant la partie masquée de leurs entrées afin de mieux comprendre leur modalité. Une analogie pourrait être faite avec deux étudiants qui apprennent un sujet, par exemple un étudiant qui regarde des vidéos et l'autre qui lit un texte pour apprendre la biologie. Ils discutent et se mettent d'accord sur ce qu'ils ont appris, améliorant ainsi leur compréhension du domaine. De plus, ils sont testés après chaque vidéo ou texte en complétant les informations manquantes du contenu qu'ils viennent de traiter individuellement. Nous avons trouvé cette analogie intéressante à partager, car elle pourrait être utile pour trouver de nouvelles idées en vue de recherches ultérieures.

Chapitre 5

Peaufinage de Modèles de Code Basé sur la Distillation

Vue d'ensemble

Comme mentionné à la section 1.3, nous proposons de travailler à deux niveaux d'entraînement différents. Le premier se situe au niveau du pré-entraînement des modèles de code en proposant l'approche décrite au chapitre 4. Le deuxième, décrit dans ce chapitre, se concentre sur le peaufinage des modèles de code déjà existants. Nous proposons une nouvelle méthode de peaufinage, appelée CodeMaster, qui réutilise les modèles précédemment peaufinés sur des langages de programmation comme professeurs pour distiller (cf. section 3.3) le peaufinage d'un autre modèle étudiant.

5.1. Approche

À l'inverse de l'approche décrite à la section 4, la méthode proposée ici apprend des représentations spécifiques pour la recherche de code. En effet, étant une méthode de peaufinage, notre technique CodeMaster n'apprend pas des représentations génériques pour n'importe quelle tache de code, mais se spécialise plutôt dans les tâches considérées pour le peaufinage. CodeMaster peut être peaufiné à partir de n'importe quel modèle. Nous avons décidé de partir des poids d'UniXCoder [**55**] car c'est le modèle avec les meilleures performances qui a mis ses poids à disposition en téléchargement libre.

Cette approche de peaufinage découle d'une réflexion sur le fait que les modèles déjà disponibles obtiennent des performances impressionnantes sur les taches de code, comme le montrent les résultats à la section 4.2. En conséquence, ne serait-il pas préférable

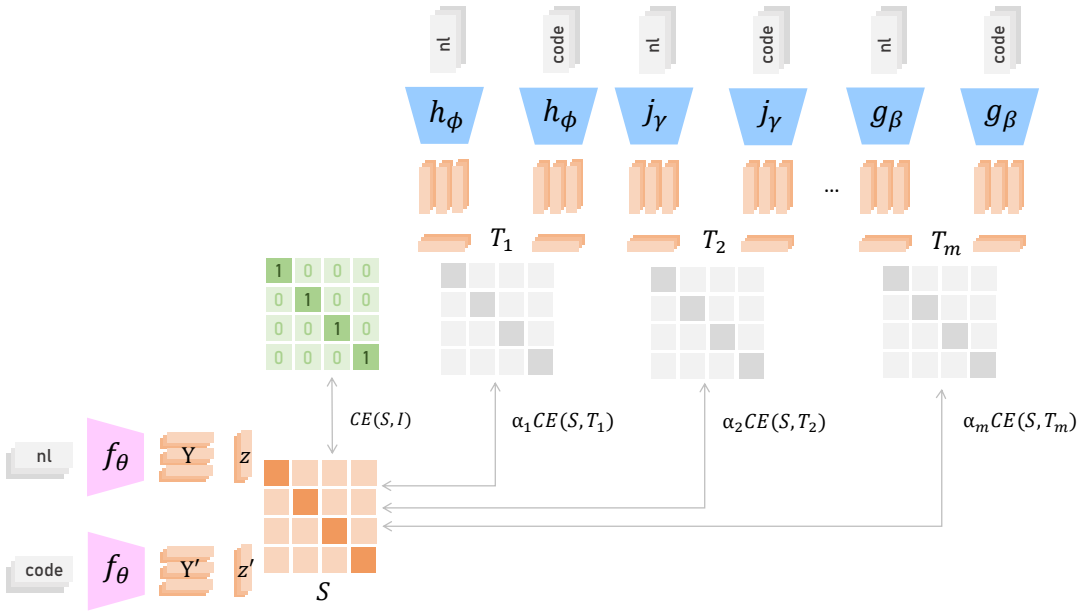


Fig. 5.1. Illustration générale de la nouvelle méthode de peaufinage CODEMASTER.

de se concentrer sur la façon d'améliorer ces modèles sans introduire de nouveaux pré-entraînements ? CODEMASTER tente de répondre par l'affirmative à cette question en créant des enseignants spécialisés pour différents langages de programmation, ce qui permet de former un ensemble d'experts avec des représentations spécifiques à chaque langage considéré. Ensuite, un nouveau modèle, initialisé avec les poids d'UniXCoder, est utilisé comme apprenant et sera peaufiné sur une tâche donnée, par exemple la recherche de code. Cependant, en plus de l'objectif habituel, cet apprenant sera guidé par le groupe d'experts grâce à un signal d'entraînement comparant les prédictions faites par chaque enseignant et celles de l'apprenant. Nous proposons également une méthode pour accorder plus ou moins de crédit au retour d'un enseignant en fonction de la qualité de ses prédictions, en utilisant un coefficient de confiance $\boldsymbol{\alpha}_i$. La figure 5.1 donne une vue d'ensemble de CODEMASTER.

5.1.1. Modèle

Étant donné un jeu de données $\mathcal{D}$ composé de morceaux de code et de leur documentation répartis parmi m langages de programmation, considérons $\mathbf{t} = [f_{\theta_1}, f_{\theta_2}, ..., f_{\theta_m}]^\top$ comme un vecteur de m modèles différents, chacun étant préalablement entraîné sur un langage spécifique parmi les m. Chaque modèle $\mathbf{t}_i$ sera qualifié d'enseignant pour le reste de cette section. En effet, ces enseignants serviront de guides lors du perfectionnement d'un étudiant f'_γ.

Considérons un ensemble de k codes C et leur documentation D. Tout d'abord, les représentations de C et D sont calculées par l'encodeur étudiant, f'_γ, créant ainsi un ensemble de représentations Y et Y' pour C et D respectivement, avec une représentation par jeton pour chaque séquence. Ensuite, ces représentations au niveau des jetons sont transformées en représentations au niveau des séquences, qui sont définies comme la moyenne des jetons d'une séquence donnée. Ainsi, Y et Y' sont convertis en un ensemble de représentations au niveau des séquences Z et Z'. Comme mentionné dans la section 2.2.1, la matrice de similarité S est ensuite calculée en normalisant les vecteurs des ensembles Z et Z', puis en effectuant le produit matriciel entre ces deux ensembles, donnant $S = ZZ'^\top$. Cette matrice S associe à chaque représentation de code $z \in Z$ et à la documentation $z' \in Z'$ la similarité cosinus d entre les deux représentations, de sorte que $\forall i,j \in [k] : S_{ij} := d(z_i, z'_j)$. Par imitation, le même processus est utilisé pour calculer la matrice des scores T_i pour chaque enseignant $\forall i \in [m] : T_i := H_i H_i'^\top$, où H_i et H'_i sont les ensembles de représentations au niveau des séquences pour l'enseignant $\mathbf{t}_i$. Tout cela est illustré par la figure 5.1.

5.1.2. Objectif de peaufinage

Comme décrit à la section 5.1.1, une fois toutes les matrices de scores calculées, l'objectif de peaufinage à minimiser est calculé et seuls les poids de l'étudiant sont optimisés par rapport à celui-ci. La fonction de coût est calculée comme la somme des entropies croisées de la matrice des scores de l'étudiant et celles des enseignants, chacune multipliée par un coefficient de confiance $\boldsymbol{\alpha}_i$. À cela s'ajoute l'objectif de peaufinage classique décrit à la section 2.2.1, à savoir l'entropie croisée entre la matrice des scores de l'étudiant et la matrice identité. Il est important de noter que pour utiliser l'entropie croisée, il faut que les deux objets fournis en entrée soient des distributions de probabilité. En conséquence, nous avons appliqué l'opération de softmax au préalable sur les vecteurs lignes de toutes les matrices de scores.

Notre objectif de peaufinage peut donc s'écrire sous la forme suivante :

$$\mathcal{L}_{\text{globale}} = \lambda \mathcal{L}_{\text{enseignant}} + (1 - \lambda)\mathcal{L}_{\text{étudiant}} \tag{5.1.1}$$

$$\mathcal{L}_{\text{enseignant}} = -\frac{1}{mk} \sum_{i}^{m} \boldsymbol{\alpha}_i \sum_{j}^{k} \sum_{h}^{k} T_{i_{jh}} \log S_{jh} \tag{5.1.2}$$

$$\mathcal{L}_{\text{étudiant}} = -\frac{1}{k} \sum_{i}^{k} I_{jj} \log S_{jj} \tag{5.1.3}$$

où T_i est la matrice de score du $i^{ème}$ enseignant, S_i la matrice de score de l'étudiant, $I \in \mathbb{R}^{k \times k}$ la matrice identité et λ un scalaire donnant plus ou moins de poids à l'objectif des enseignants. À noter que la manière dont les coefficients de confiance $\boldsymbol{\alpha}_i$ des enseignants sont calculés est décrite à la section 5.1.2.1.

5.1.2.1. Confiance des enseignants. Notre objectif de peaufinage introduit des coefficients devant chaque entropie croisée entre la matrice des scores de l'étudiant et celles des enseignants. Cela permet de donner plus de crédit aux enseignants sûrs de ce qu'ils prédisent et, réciproquement, de fournir moins de poids à ceux qui se trompent. Un coefficient de confiance pour un enseignant i est défini comme l'opposé de l'entropie croisée entre la matrice des scores T_i de l'enseignant et la matrice identité $I \in \mathbb{R}^{k \times k}$. Cela permet d'avoir un coefficient corrélé à la justesse des prédictions de la matrice des scores T_i. Il est important de noter que, pour obtenir une distribution des coefficients, une opération de softmax est appliquée, créant ainsi des coefficients dont les valeurs sont comprises entre 0 et 1 et dont la somme est équivalente à 1.

$$\boldsymbol{\alpha} = \text{softmax}([-\text{CE}(T_1, I), -\text{CE}(T_2, I), ..., -\text{CE}(T_m, I)]^\top) \tag{5.1.4}$$

où CE est l'entropie croisée entre T_i et I dont la définition est $\text{CE}(p,q) = -\sum_i p(i) \log q(i)$ et $\boldsymbol{\alpha}_i$ est le coefficient correspondant à $\mathbf{t}_i$.

5.2. Expériences & Résultats

Il est important de noter que cette méthode n'a pas conçu pour la détection de clones, car notre approche nous permet uniquement de peaufiner des modèles pour la recherche de code. Toutefois, nous avons tout de même souhaité tester si un modèle peaufiné pour la recherche de code, via notre approche, avait un impact sur les performances pour la détection de clones. Nous avons donc également rapporté les résultats d'un modèle peaufiné pour la recherche de code avec notre approche pour la détection de clones multilingues dans un contexte de zéro-coup, car cette dernière ne nécessite pas de peaufinage supplémentaire. Le modèle pré-entrainé utilisé pour les enseignants et pour peaufiner l'étudiant est l'encodeur de UnixCoder [**55**] (12 couches de *Transformer* avec des états cachés de dimensions 768 et 12 têtes d'attentions). À noter que toutes les expériences ont été réalisées sur un serveur contenant quatre cartes graphiques NVIDIA GeForce RTX 3090 chacune avec 24 Go de mémoire vives et avec le langage python via les bibliothèques PyTorch [**106**] et HuggingFace [**142**] pour l'implémentation des modèles.Pour toutes les tâches d'évaluation, les hyper-paramètres pour le peaufinage sont identiques et sont les suivants : le nombre d'époques est de 10, la taille de la séquence de code est de 256 et 128 pour le texte, la taille des paquets de données pour l'entrainement et l'évaluation sont de 64, le taux d'apprentissage est de 0,00002 et le

coefficient pour la fonction d'objectif λ est de 0.7. Il est aussi important de noter que le meilleur modèle est sélectionné après chaque époque et sur base de ses performances sur le jeu de validation en fonction de la métrique d'évaluation de la tâche (MRR ou MAP).

5.2.1. Recherche de code

	CosQA	**AdvTest**	**CSN**
RoBERTa [**90**]	-	-	61.7
CodeBERT [**40**]	64.7	27.2	69.3
GraphCodeBERT [**56**]	67.5	35.2	71.3
SyncoBERT [**136**]	-	38.1	74.0
UniXcoder [**55**]	70.1	41.3	74.4
SCodeR [**87**]	74.5	45.5	74.5
CodeRetriever [**86**]	**75.4**	**46.9**	**77.4**
CodeMaster (notre méthode)	70.4	41.0	73.93

Tableau 5.1. Performance MRR (le plus élevé est le mieux) de CodeMaster sur les jeux de données de recherche de codes CosQA, AdvTest et CodeSearchNet.

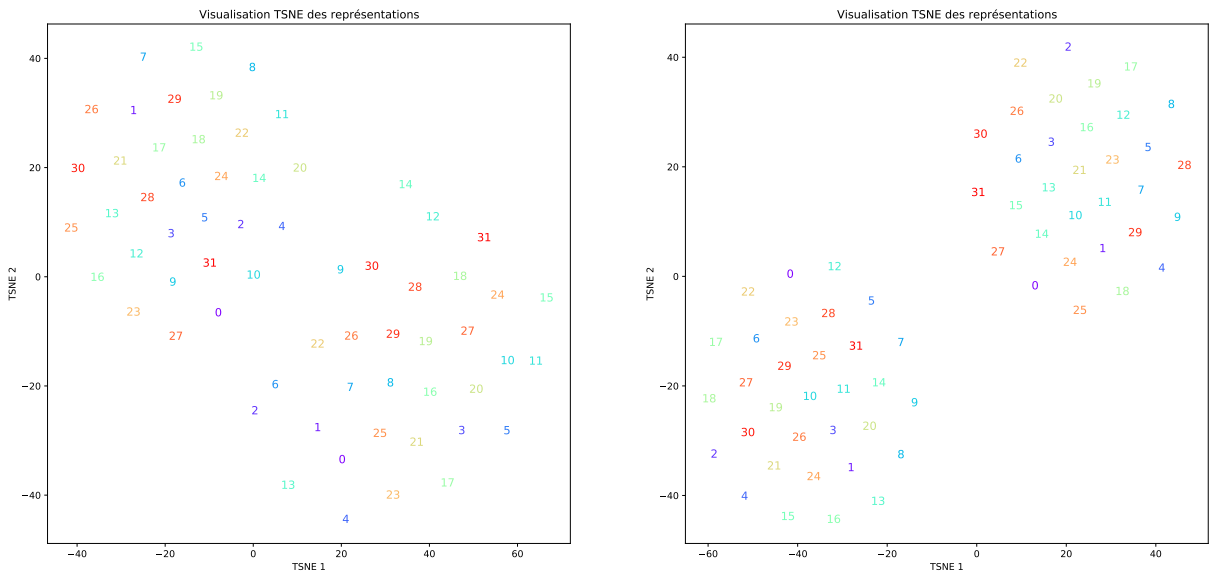
5.2.1.1. CodeSearchNet. À l'image de la section 4.2, nous avons évalué les performances de notre approche pour la recherche de code sur les jeux de données CodeSearchNet, CosQA, AdvTest, ainsi que dans l'expérience de recherche de code multilingues en zéro-coup. En effet, cela nous permet de quantifier dans quelle mesure notre modèle a bien peaufiné ses connaissances pour le jeu de données sur lequel il a été entraîné. D'autre part, cela nous permet de déterminer si les connaissances acquises généralisent à des données jamais vues, donnant ainsi une idée de l'abstraction et de la généralité des représentations apprises. Le tableau 5.2 présente les performances en MRR sur les trois jeux de données mentionnés précédemment. Nous pouvons constater que notre approche n'atteint pas les performances des méthodes proposant un pré-entraînement sur des bases de poids, à savoir SCodeR et CodeRetriever, qui sont des modèles à l'état de l'art. Cependant, nous obtenons de meilleurs résultats sur CosQA par rapport à UnixCoder. Le tableau 5.2 rapporte également les performances en MRR de notre approche sur CodeSearchNet, comparées aux modèles à l'état de l'art. Nous pouvons constater que notre approche obtient de meilleurs résultats que le modèle initial, UnixCoder, pour certains langages tels que Ruby, JavaScript et Go, mais pas pour les autres langages.

5.2.1.2. Recherche de codes multilingues à partir de zéro. Tout comme notre approche décrite dans le chapitre 4, nous avons évalué la capacité de notre méthode à déterminer si les connaissances acquises généralisent à des données jamais vues, ce qui permet d'avoir

une idée de l'abstraction et de la généralité des représentations apprises. L'objectif de l'expérience est de trouver les codes les plus similaires à un code donné en requête, sans peaufiner le modèle avec les données, c'est-à-dire en partant de zéro. Le tableau 5.3 présente la précision moyenne (MAP) pour chaque langage source et les différents langages cibles. Nous pouvons constater que notre approche, bien que peaufiné sur UnixCoder, obtient des performances inférieures à celles de ce dernier, mais supérieures aux autres approches. Une raison à cela est que CodeMaster, étant peaufiné avec du code et du langage naturel pour la recherche de code, aurait des difficultés lorsqu'il s'agit de rechercher uniquement des morceaux de code sans langage naturel.

	Ruby	**JavaScript**	**Go**	**Python**	**Java**	**PHP**	**Moy.**
RoBERTa [**90**]	58.7	51.7	85.0	58.7	59.9	56.0	61.7
ContraCode [**69**]	-	30.6	-	-	-	-	-
CodeBERT [**40**]	67.9	62.0	88.2	67.2	67.6	62.8	69.3
GraphCodeBERT [**56**]	70.3	64.4	89.7	69.2	69.1	64.9	71.3
SynCoBERT [**136**]	72.2	67.7	91.3	72.4	72.3	67.8	74.0
UniXcoder [**55**]	74.0	68.4	91.5	72.0	72.6	67.6	74.4
SCodeR [**87**]	**77.5**	**72.0**	**92.7**	74.2	74.8	69.2	76.7
CodeRetriever [**86**]	77.1	71.9	92.4	**75.8**	**76.5**	**70.8**	**77.4**
CodeMaster (notre méthode)	75.8	68.9	91.5	70.8	71.6	65.0	73.93

Tableau 5.2. Performance MRR (le plus élevé est le mieux) de CodeMaster sur le jeu de données CodeSearchNet sur six langages de programmation.



(a) UnixCoder (b) CodeMaster

Fig. 5.2. Comparaison des graphiques t-SNE [**131**] des représentations de 30 paires code-documentation pour Unixcoder (à gauche) et notre approche CodeMaster (à droite). Une paire partage le même numéro pour sa documentation et son code.

5.2.2. Évaluation qualitative

Comme pour le chapitre 4, nous avons également procédé à une évaluation qualitative des représentations apprises par CodeMaster. Dans cette évaluation, nous avons comparé ces représentations à celles obtenues par UniXCoder. Pour ce faire, nous avons utilisé des visualisations t-SNE afin de mettre en évidence la similarité entre les paires de code et de documentation, illustré à la figure 5.2. Il est intéressant de noter que, comme pour `rncp-code`, les deux méthodes créent deux groupes distincts de représentations : l'un pour le code et l'autre pour la documentation.

En plus de cette évaluation qualitative, nous avons également évalué les performances de CodeMaster en termes de suggestions de code pour la requête spécifique "Comment faire le tri à bulle". Nous avons comparé les résultats obtenus par CodeMaster avec ceux générés par UniXCoder. À l'inverse de notre approche décrite au chapitre 4, CodeMaster propose des codes pertinents en position plus élevée que UniXCoder pour la requête, comme l'illustre la figure 5.3.

	Ruby			**Python**			**Java**			**Moy.**
	Ruby	Python	Java	Ruby	Python	Java	Ruby	Python	Java	
CodeBERT [**40**]	13.55	3.18	0.71	3.12	14.39	0.96	0.55	0.42	7.62	4.94
GraphCodeBERT [**56**]	17.01	9.29	6.38	5.01	19.34	6.92	1.77	3.50	13.31	9.17
PLBART [**1**]	18.60	10.76	1.90	8.27	19.55	1.98	1.47	1.27	10.41	8.25
CodeT5-base [**138**]	18.22	10.02	1.81	8.74	17.83	1.58	1.13	0.81	10.18	7.81
UniXcoder [**55**]	**29.05**	**26.36**	**15.16**	**23.96**	**30.15**	**15.07**	**13.61**	**14.53**	**16.12**	**20.45**
CodeMaster (notre méthode)	26.54	19.4	8.38	14.77	26.25	7.5	8.38	4.78	13.88	14.43

Tableau 5.3. Score MAP (%) (le plus élevé est le mieux) de CodeMaster pour une tache de recherche code-à-code dans un contexte de zéro-coup.

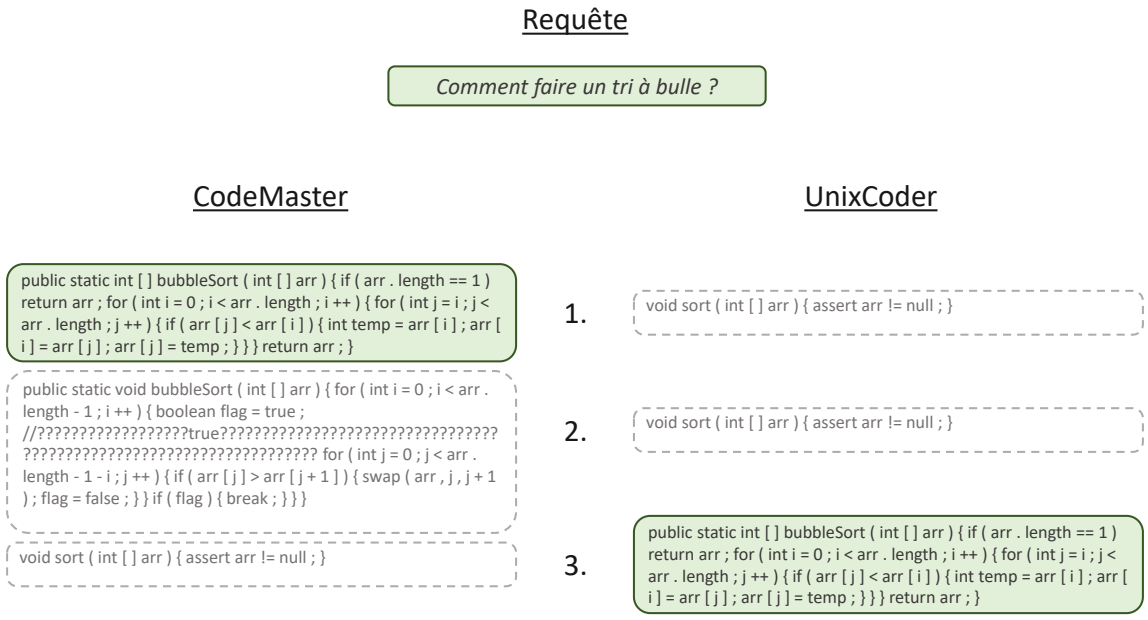


Fig. 5.3. Comparaison des trois meilleurs codes trouvés dans CodeSearchNet pour CodeMaster et UnixCoder étant donné la requête *"Comment faire un tri à bulle ?"*.

5.3. Discussions

Malgré des résultats prometteurs, CodeMaster ne parvient pas à surpasser les performances de toutes les méthodes à l'état de l'art. Cependant, il est important de noter que ces dernières proposent des pré-entraînements de modèles et non des peaufinages. Notre méthode de peaufinage sur UnixCoder [**55**] permet d'obtenir de meilleures performances que la méthode de peaufinage habituelle. Nous aurions également aimé évaluer notre méthode de peaufinage sur SCodeR [**87**] et CodeRetriever [**86**], mais ces derniers n'ont pas fourni de sauvegarde de leurs modèles pour l'évaluation. Malgré cela, nous avons plusieurs pistes d'amélioration pour notre approche.

5.3.1. Amélioration des coefficients

Nous pensons qu'une piste d'amélioration pour la méthode serait l'apprentissage des coefficients de confiance pour chaque enseignant. Cela pourrait se mettre en place de différentes manières. La première assez simple serait de définir chaque coefficient comme un paramètre appris lors de l'apprentissage. On peut encore améliorer cette méthode en mettant des contraintes, telles que la somme des coefficients doit être égale à 1. Notre dernière idée serait d'utiliser un autre modèle qui, étant conditionné sur la donnée en entrée, produirait une distribution des degrés de confiance pour des langages.

5.3.2. Amélioration de la distillation

Notre première façon d'améliorer la distillation pourrait être de distiller uniquement les connaissances de l'enseignant avec le plus gros coefficient de confiance. Cela permettrait de réduire le bruit dans le signal de distillation et peut-être aiderait l'étudiant à apprendre de meilleures représentations.

Une façon de rendre le peaufinage plus complexe et de potentiellement améliorer la distillation serait d'ajouter un terme à la fonction d'objectif, que nous avons appelé *accord neuronal*, permettant aux enseignants et à l'étudiant de se mettre d'accord sur le classement entre leurs représentations du code et des documentations. Plus précisément, la matrice de scores S décrite à la section 2.2.1 n'est plus calculée comme le produit des représentations du code et de leurs documentations. À la place, deux matrices de scores S_1 et S_2 sont calculées. La première est le produit des représentations du code d'un enseignant $\mathbf{t}_t$ et celui de l'étudiant, la deuxième est la même chose, mais avec les représentations des documentations. Le but de ce nouveau terme serait de rapprocher les classements faits par les enseignants et l'étudiant.

Chapitre 6

Conclusion et travaux futures

En conclusion, ce mémoire propose d'explorer de nouvelles techniques pour l'amélioration de l'apprentissage auto-supervisé des représentations du code. Cela en se concentrant sur les taches de recherche de code et de détection de clones de type 4, ces dernières nécessitant des représentations capturant des informations avec un haut niveau d'abstraction telles que la sémantique fonctionnelle. Les contributions principales de ce mémoire sont les suivantes : (1) Tout d'abord, la proposition d'un nouvel algorithme de pré-entraînement pour les modèles de code basé sur VICReg [**14**], une méthode non contrastive régularisée. (2) Ensuite, un nouvel objectif de peaufinage basé sur la distillation de connaissances à partir de modèles déjà peaufinés, en utilisant des coefficients de confiance pour attribuer des crédences proportionnelles à la qualité des prédictions des modèles.

Les résultats expérimentaux et les analyses menées dans ce mémoire ne démontrent pas des performances extraordinaires sur les différents jeux de données considérés. Cependant, les approches décrites proposent des pistes de directions de recherche innovantes et non conventionnelles. Nous sommes convaincus que davantage de recherches dans ces directions permettraient l'élaboration de techniques permettant d'obtenir de meilleures représentations de qualité et de meilleures performances dans des tâches clés de l'automatisation du génie logiciel. Ces résultats contribuent donc à l'avancement des connaissances dans le domaine de l'apprentissage automatique appliqué au code et ouvrent de nouvelles perspectives pour l'amélioration des techniques existantes.

L'évolution croissante de l'informatique, l'importance grandissante de la place du logiciel dans notre société et les récents progrès en apprentissage automatique soulignent l'importance et l'intérêt de la conduite de recherches dans le domaine du génie logiciel. Cela est appuyé par l'investissement croissant d'entreprises de premier plan dans ce domaine, ce qui ajoute du crédit au potentiel de l'apprentissage automatique pour améliorer la productivité,

la qualité et l'évolution des logiciels produits. Les travaux consignés dans ce mémoire tentent de contribuer à cet effort en explorant le potentiel et en offrant des perspectives prometteuses pour de nouvelles approches de l'apprentissage des représentations du code. Nous espérons que cela permettra d'ouvrir la voie à de nouvelles avancées dans le domaine et de fournir des pistes pour de futures recherches.

6.1. Limitations

Malgré les propositions de pistes intéressantes pour l'amélioration de l'apprentissage auto-supervisé des représentations du code, les travaux proposés cumulent plusieurs limitations, mais également plus de pistes d'amélioration pour de futurs travaux. Les sections 4.3 et 5.3 fournissent déjà des pistes et hypothèses d'amélioration pour les méthodes proposées.

Une grosse limitation de ce mémoire est le fait que les résultats de la première méthode ne soient pas du tout au niveau de l'état de l'art. Au vu des résultats compilés à la section 4.2, il semble difficile de croire au potentiel de cette méthode. Cependant, nous sommes convaincus que les méthodes non contrastives régularisées sont le futur de l'apprentissage auto-supervisé, et nous avons d'ores et déjà proposé une liste d'hypothèses et d'améliorations possibles pour cette méthode à la section 4.3. D'autres limitations de notre approche pour cette idée sont la capacité de calcul et le temps d'entraînement. En effet, VICReg a été entraîné pendant plusieurs centaines d'époques avec une taille de paquet de 512 et en utilisant 32 cartes graphiques. Nous n'avions pas de telles ressources à notre disposition, et nous pensons que plus de ressources computationnelles pourraient certainement améliorer également les performances de l'approche.

De même que pour la première contribution, notre objectif de peaufinage ne bat pas les performances de l'état de l'art en ce qui concerne les méthodes proposant un nouveau pré-entraînement. Cependant, en partant du meilleur modèle de l'état de l'art fournissant ses poids, à savoir UniXCoder [**55**], nous battons la méthode de peaufinage habituellement utilisée pour la recherche de code, suggérant ainsi le potentiel de notre approche. Cependant, nous avons également proposé une liste d'hypothèses d'améliorations possibles pour cette méthode, également à la section 5.3, dans le but de constituer une fonction objectif permettant de battre les résultats nécessitant un pré-entraînement avec seulement un peaufinage.

6.2. Travaux futures

Des directions de recherche futures pour la première méthode pourraient être de considérer plus de modalités comme l'arbre syntaxique du code (AST) dans l'apprentissage des représentations. L'utilisation de code similaire écrit dans différents langages de programmation semble également être une piste intéressante pour l'extraction de concepts abstraits relatifs au code. Nous émettons également l'hypothèse que la méthode de pré-entraînement proposée pourrait s'avérer utile pour d'autres tâches liées au code qui dépendent de la capacité des représentations à capturer la sémantique fonctionnelle des programmes. Par exemple, nous aimerions tester si notre méthode pourrait améliorer les modèles de traduction de programmes d'un langage vers un autre. En ce qui concerne notre objectif de peaufinage, une direction future pourrait être d'utiliser l'amalgamation des modèles [**119**], dans le but de combiner tous les modèles de code existants, entraînés avec toutes sortes de modalités, afin de distiller leurs connaissances dans un étudiant pour créer des représentations robustes et capturer des informations abstraites sur le code.

Références bibliographiques

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray et Kai-Wei Chang : Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

[2] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam et Bilal Maqbool : A systematic review on code clone detection. *IEEE access*, 7:86121–86144, 2019.

[3] R Al-Zoubi et A Prakash : Software change analysis via attributed dependency graphs. Rapport technique, Technical Report CSE-TR-95-91, Department of EECS, University of Michigan, 1991.

[4] Miltiadis Allamanis, Earl T Barr, Christian Bird et Charles Sutton : Learning natural coding conventions. *In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293, 2014.

[5] James Ambras et Vicki O'Day : Microscope: A program analysis system. *In Proc. of the 20th Hawaii International Conference on System Sciences*, pages 460–468, 1987.

[6] Rohan Anil, Gabriel Pereyra, Alexandre Passos, Robert Ormandi, George E Dahl et Geoffrey E Hinton : Large scale distributed neural network training through online distillation. *arXiv preprint arXiv:1804.03235*, 2018.

[7] Lerina Aversano, Luigi Cerulo et Massimiliano Di Penta : How clones are maintained: An empirical study. *In 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 81–90. IEEE, 2007.

[8] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi et Cristina Lopes : Sourcerer: a search engine for open source code supporting structure-based search. *In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 681–682, 2006.

[9] Sushil K Bajracharya, Joel Ossher et Cristina V Lopes : Leveraging usage similarity for effective retrieval of examples in code repositories. *In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 157–166, 2010.

[10] Brian S Baker : A program for identifying duplicated code. *In Proceedings of the 24th ACM/IEEE conference on Software engineering*, pages 49–57. IEEE Computer Society Press Los Alamitos, CA, USA, 1992.

[11] Magdalena Balazinska, Ettore Merlo, Benoit Dagenais, Benjamin Lagu'e et Kostas A Kontogiannis : Measuring clone based reengineering opportunities. *In Proceedings of the 6th International Symposium on Software Metrics*, pages 292–303, 1999.

[12] Randall Balestriero et Yann LeCun : Contrastive and non-contrastive self-supervised learning recover global and local spectral embedding methods. *In* Alice H. Oh, Alekh Agarwal, Danielle Belgrave et Kyunghyun Cho, éditeurs : *Advances in Neural Information Processing Systems*, 2022.

[13] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin et Daniel Tarlow : Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[14] Adrien Bardes, Jean Ponce et Yann LeCun : VICReg: Variance-invariance-covariance regularization for self-supervised learning. *In International Conference on Learning Representations*, 2022.

[15] Ira Baxter et Randall Matthias : Dow chemical awards contract extension to semantic designs for process control software reengineering. `http://www.semdesigns.com/Announce/DOW_PRWebRelease_May2017_14309535.pdf`, 2017.

[16] Ira D Baxter, Alex Yahin, Leonardo Moura, Marco Sant'Anna et Laura Bier : Clone detection using abstract syntax trees. *In Proceedings of the IEEE International Conference on Software Maintenance (ICSM '98)*, pages 368–377. IEEE, 1998.

[17] Yoshua Bengio, Aaron Courville et Pascal Vincent : Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013.

[18] Yoshua Bengio *et al.* : Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[19] H. L. Berghel et D. L. Sallach : Measurements of program similarity in identical task environments. *SIGPLAN Not.*, 19(8):65–76, aug 1984.

[20] Pearl Brereton, David Budgen, Keith Bennnett, Malcolm Munro, Paul Layzell, Linda MacCaulay, David Griffiths et Charles Stannett : The future of software. *Communications of the ACM*, 42(12):78–84, 1999.

[21] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell *et al.* : Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[22] Nghi DQ Bui, Yijun Yu et Lingxiao Jiang : Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. *In Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 511–521, 2021.

[23] Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé Jégou, Julien Mairal, Piotr Bojanowski et Armand Joulin : Emerging properties in self-supervised vision transformers. *In Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 9650–9660, October 2021.

[24] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman *et al.* : Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[25] Ting Chen, Simon Kornblith, Mohammad Norouzi et Geoffrey Hinton : A simple framework for contrastive learning of visual representations. *In International conference on machine learning*, pages 1597–1607. PMLR, 2020.

[26] Ting Chen, Simon Kornblith, Kevin Swersky, Mohammad Norouzi et Geoffrey E Hinton : Big self-supervised models are strong semi-supervised learners. *Advances in neural information processing systems*, 33:22243–22255, 2020.

[27] Xinlei Chen, Haoqi Fan, Ross Girshick et Kaiming He : Improved baselines with momentum contrastive learning. arxiv 2020. *arXiv preprint arXiv:2003.04297*, 2020.

[28] Xinlei Chen et Kaiming He : Exploring simple siamese representation learning. *In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15750–15758, 2021.

[29] Y-F Chen, Michael Y. Nishimoto et CV Ramamoorthy : The c information abstraction system. *IEEE Transactions on software Engineering*, 16(3):325–334, 1990.

[30] Colin B Clement, Chen Wu, Dawn Drain et Neel Sundaresan : Distilling transformers for neural cross-domain search. *arXiv preprint arXiv:2108.03322*, 2021.

[31] James R Cordy, Charles D Halpern-Hamu et Eric Promislow : Txl: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[32] Haixing Dai, Zhengliang Liu, Wenxiong Liao, Xiaoke Huang, Yihan Cao, Zihao Wu, Lin Zhao, Shaochen Xu, Wei Liu, Ninghao Liu, Sheng Li, Dajiang Zhu, Hongmin Cai, Lichao Sun, Quanzheng Li, Dinggang Shen, Tianming Liu et Xiang Li : Auggpt: Leveraging chatgpt for text data augmentation, 2023.

[33] Jacob Devlin, Ming-Wei Chang, Kenton Lee et Kristina Toutanova : Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[34] Luca Di Grazia et Michael Pradel : Code search: A survey of techniques for finding code. *ACM Computing Surveys*, 55(11):1–31, 2023.

[35] Yangruibo Ding, Luca Buratti, Saurabh Pujar, Alessandro Morari, Baishakhi Ray et Saikat Chakraborty : Towards learning (dis)-similarity of source code from program contrasts. *In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6300–6312, 2022.

[36] S Ducasse, M Rieger et S Demeyer : A language independent approach for detecting duplicated code. *In Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99)*, pages 109–118. IEEE, 1999.

[37] Dumitru Erhan, Aaron Courville, Yoshua Bengio et Pascal Vincent : Why does unsupervised pre-training help deep learning? *In Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 201–208. JMLR Workshop and Conference Proceedings, 2010.

[38] Aleksandr Ermolov, Aliaksandr Siarohin, Enver Sangineto et Nicu Sebe : Whitening for self-supervised representation learning. *In International Conference on Machine Learning*, pages 3015–3024. PMLR, 2021.

[39] Steven Y Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura et Eduard Hovy : A survey of data augmentation approaches for nlp. *arXiv preprint arXiv:2105.03075*, 2021.

[40] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang *et al.* : Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[41] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang et Ming Zhou : CodeBERT: A pre-trained model for programming and natural languages. *In Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, novembre 2020. Association for Computational Linguistics.

[42] Karen A Frenkel : Toward automating the software-development cycle. *Communications of the ACM*, 28(6):578–589, 1985.

[43] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann et Koushik Sen : Codehint: Dynamic and interactive synthesis of code snippets. *In Proceedings of the 36th International Conference on Software Engineering*, pages 653–663, 2014.

[44] Tianyu Gao, Xingcheng Yao et Danqi Chen : Simcse: Simple contrastive learning of sentence embeddings. *arXiv preprint arXiv:2104.08821*, 2021.

[45] ISABEL GARCÍA-CONTRERAS, JOSÉ F. MORALES et MANUEL V. HERMENEGILDO : Semantic code browsing. *Theory and Practice of Logic Programming*, 16(5-6):721–737, 2016.

[46] David Garlan, Linxi Cai et Robert L Nord : A transformational approach to generating application-specific environments. *In Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pages 68–77, 1992.

[47] Quentin Garrido, Yubei Chen, Adrien Bardes, Laurent Najman et Yann Lecun : On the duality between contrastive and non-contrastive self-supervised learning. *arXiv preprint arXiv:2206.02574*, 2022.

[48] Pratiksha Gautam et Hemraj Saini : Various code clone detection techniques and tools: a comprehensive survey. *In Smart Trends in Information Technology and Computer Communications: First International Conference, SmartCom 2016, Jaipur, India, August 6–7, 2016, Revised Selected Papers 1*, pages 655–667. Springer, 2016.

[49] Görkem Giray, Kwabena Ebo Bennin, Ömer Köksal, Önder Babur et Bedir Tekinerdogan : On the use of deep learning in software defect prediction. *Journal of Systems and Software*, 195:111537, 2023.

[50] Nils Göde et Rainer Koschke : Incremental clone detection. *In Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09)*, pages 219–228, 2009.

[51] Yaroslav Golubev, Viktor Poletansky, Nikita Povarov et Timofey Bryksin : Multi-threshold token-based code clone detection. *In Proceedings of the 28th International Conference on Software Analysis, Evolution and Reengineering (SANER'21)*, pages 496–500, 2021.

[52] Judith E. Grass : Object-oriented design archaeology with cia++. *Computing Systems*, 5(1):5–67, 1992.

[53] Sam Grier : A tool that detects plagiarism in pascal programs. *ACM Sigcse Bulletin*, 13(1):15–20, 1981.

[54] Jean-Bastien Grill, Florian Strub, Florent Altché, Corentin Tallec, Pierre Richemond, Elena Buchatskaya, Carl Doersch, Bernardo Avila Pires, Zhaohan Guo, Mohammad Gheshlaghi Azar *et al.* : Bootstrap your own latent-a new approach to self-supervised learning. *Advances in neural information processing systems*, 33:21271–21284, 2020.

[55] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou et Jian Yin : UniXcoder: Unified cross-modal pre-training for code representation. *In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland, mai 2022. Association for Computational Linguistics.

[56] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang et Ming Zhou : Graphcode{bert}: Pre-training code representations with data flow. *In International Conference on Learning Representations*, 2021.

[57] R. Hadsell, S. Chopra et Y. LeCun : Dimensionality reduction by learning an invariant mapping. *In 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742, 2006.

[58] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie et Ross Girshick : Momentum contrast for unsupervised visual representation learning. *In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9729–9738, 2020.

[59] Kaiming He, Xiangyu Zhang, Shaoqing Ren et Jian Sun : Deep residual learning for image recognition. *In Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[60] Olivier Henaff : Data-efficient image recognition with contrastive predictive coding. *In International conference on machine learning*, pages 4182–4192. PMLR, 2020.

[61] Jordan Henke, Goutham Ramakrishnan, Zi Wang, Aws Albarghouth, Somesh Jha et Thomas Reps : Semantic robustness of models of source code. *In 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 526–537. IEEE, 2022.

[62] Geoffrey Hinton, Oriol Vinyals, Jeff Dean *et al.* : Distilling the knowledge in a neural network.

[63] R Devon Hjelm, Alex Fedorov, Samuel Lavoie-Marchildon, Karan Grewal, Phil Bachman, Adam Trischler et Yoshua Bengio : Learning deep representations by mutual information estimation and maximization. *arXiv preprint arXiv:1808.06670*, 2018.

[64] Tiancheng Hu, Zijing Xu, Yilin Fang, Yueming Wu, Bin Yuan, Deqing Zou et Hai Jin : Fine-grained code clone detection with block-based splitting of abstract syntax tree. *In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 89–100, 2023.

[65] Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou et Nan Duan : Cosqa: 20,000+ web queries for code search and question answering. *arXiv preprint arXiv:2105.13239*, 2021.

[66] Weiran Huang, Mingyang Yi et Xuyang Zhao : Towards the generalization of contrastive self-supervised learning. *arXiv preprint arXiv:2111.00743*, 2021.

[67] Yu-Liang Hung et Shingo Takada : Cppcd: A token-based approach to detecting potential clones. *In Proceedings of the 14th International Workshop on Software Clones (IWSC'20)*, pages 26–32, 2020.

[68] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis et Marc Brockschmidt : Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[69] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E Gonzalez et Ion Stoica : Contrastive code representation learning. *arXiv preprint arXiv:2007.04973*, 2020.

[70] Hugo T. Jankowitz : Detecting plagiarism in student pascale programs. *The computer journal*, 31(1):1–8, 1988.

[71] Li Jing, Pascal Vincent, Yann LeCun et Yuandong Tian : Understanding dimensional collapse in contrastive self-supervised learning. *In International Conference on Learning Representations*, 2022.

[72] Toshihiro Kamiya : Ccfinderx: An interactive code clone analysis environment. 2021.

[73] Toshihiro Kamiya, Shinji Kusumoto et Katsuro Inoue : Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *In IEEE Transactions on Software Engineering*, volume 28, pages 654–670, 2002.

[74] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu et Dario Amodei : Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[75] Iman Keivanloo, Juergen Rilling et Ying Zou : Spotting working code examples. *In Proceedings of the 36th International Conference on Software Engineering*, pages 664–675, 2014.

[76] Brian W Kernighan et John R Mashey : The unix™ programming environment. *Software: Practice and Experience*, 9(1):1–15, 1979.

[77] Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein et Yves Le Traon : Facoy: a code-to-code search engine. *In Proceedings of the 40th International Conference on Software Engineering*, pages 946–957, 2018.

[78] Miryung Kim, Vibha Sazawal, David Notkin et Gail Murphy : An empirical study of code clone genealogies. *In Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196, 2005.

[79] G. Kotik et L. Markosian : Automating software analysis and testing using a program transformation system. *In Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*, TAV3, page 75–84, New York, NY, USA, 1989. Association for Computing Machinery.

[80] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese et Steven Chu Hong Hoi : Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

[81] Phuc H Le-Khac, Graham Healy et Alan F Smeaton : Contrastive representation learning: A framework and review. *Ieee Access*, 8:193907–193934, 2020.

[82] Yann LeCun : A path towards autonomous machine intelligence. *preprint posted on openreview*, 2022.

[83] Haochen Li, Chunyan Miao, Cyril Leung, Yanxian Huang, Yuan Huang, Hongyu Zhang et Yanlin Wang : Exploring representation-level augmentation for code search. *In Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 4924–4936, Abu Dhabi, United Arab Emirates, décembre 2022. Association for Computational Linguistics.

[84] Haochen Li, Chunyan Miao, Cyril Leung, Yanxian Huang, Yuan Huang, Hongyu Zhang et Yanlin Wang : Exploring representation-level augmentation for code search. *arXiv preprint arXiv:2210.12285*, 2022.

[85] Wen Li, Junfei Xu et Qi Chen : Knowledge distillation-based multilingual fusion code retrieval. *Algorithms*, 15(1):25, 2022.

[86] Xiaonan Li, Yeyun Gong, Yelong Shen, Xipeng Qiu, Hang Zhang, Bolun Yao, Weizhen Qi, Daxin Jiang, Weizhu Chen et Nan Duan : Coderetriever: Unimodal and bimodal contrastive learning. *arXiv preprint arXiv:2201.10866*, 2022.

[87] Xiaonan Li, Daya Guo, Yeyun Gong, Yun Lin, Yelong Shen, Xipeng Qiu, Daxin Jiang, Weizhu Chen et Nan Duan : Soft-labeled contrastive pre-training for function-level code representation. *arXiv preprint arXiv:2210.09597*, 2022.

[88] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago *et al.* : Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[89] Mark A Linton : Implementing relational views of programs. *ACM SIGSOFT Software Engineering Notes*, 9(3):132–140, 1984.

[90] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer et Veselin Stoyanov : Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[91] Wenjie Zhuang Na Meng Liuqing Li, He Feng et Barbara Ryder : Cclearner: A deep learning-based clone detection approach. *In Proceedings of the 33rd International Conference on Software Maintenance and Evolution (ICSME'17)*, pages 249–260, 2017.

[92] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang *et al.* : Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[93] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen et Satish Chandra : Aroma: Code recommendation via structural code search. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[94] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang et Jianjun Zhao : Codehow: Effective code search based on api understanding and extended boolean model (e). *In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 260–270. IEEE, 2015.

[95] Nazim H Madhavji : Compare: A collusion detector for pascal. *Technique et Science Informatiques*, 4(6):489–497, 1985.

[96] Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern *et al.* : Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.

[97] Jacques Mayrand, Craig Leblanc et Ettore M Merlo : Experiment on the automatic detection of function clones in a software system using metrics. *In Proceedings of the IEEE International Conference on Software Maintenance (ICSM '96)*, pages 244–253. IEEE, 1996.

[98] Alon Mishne, Sharon Shoham et Eran Yahav : Typestate-based semantic code search over partial programs. *SIGPLAN Not.*, 47(10):997–1016, oct 2012.

[99] Lili Mou, Ge Li, Lu Zhang, Tao Wang et Zhi Jin : Convolutional neural networks over tree structures for programming language processing. *In Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[100] Hausi A Müller, Brian Douglas Corrie et Scott Robert Tilley : *Spatial and visual representations of software structures: A model for reverse engineering*. University of Victoria, Department of Computer Science, 1991.

[101] Tasuku Nakagawa, Yoshiki Higo et Shinji Kusumoto : Nil: Large-scale detection of large-variance clones. *In Proceedings of the 29th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'21)*, pages 830–841, 2021.

[102] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy *et al.* : Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005*, 2022.

[103] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H Pham, Jafar Al-Kofahi et Tien N Nguyen : Clone management for evolving software. *IEEE transactions on software engineering*, 38(5):1008–1026, 2011.

[104] Edward E Ogheneovo *et al.* : On the relationship between software complexity and maintenance costs. *Journal of Computer and Communications*, 2(14):1, 2014.

[105] Aaron van den Oord, Yazhe Li et Oriol Vinyals : Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

[106] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai et Soumith Chintala : Pytorch: An imperative style, high-performance deep learning library, 2019.

[107] S. Paul et A. Prakash : A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

[108] Michael Pradel et Koushik Sen : Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25, 2018.

[109] Varot Premtoon, James Koppel et Armando Solar-Lezama : Semantic code search via equational reasoning. *In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language*

Design and Implementation, PLDI 2020, page 1066–1082, New York, NY, USA, 2020. Association for Computing Machinery.

[110] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker *et al.* : Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 2021.

[111] Goutham Ramakrishnan, Jordan Henkel, Zi Wang, Aws Albarghouthi, Somesh Jha et Thomas Reps : Semantic robustness of models of source code. *arXiv preprint arXiv:2002.03043*, 2020.

[112] Chanchal K Roy et James R Cordy : Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. *In 2008 16th iEEE international conference on program comprehension*, pages 172–181. IEEE, 2008.

[113] Chanchal K. Roy et James R. Cordy : Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. *In Proceedings of the 16th International Conference on Program Comprehension (ICPC'08)*, pages 172–181, 2008.

[114] Colin Runciman et Ian Toyn : Retrieving re-usable software components by polymorphic type. *In Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 166–173, 1989.

[115] Caitlin Sadowski, Kathryn T. Stolee et Sebastian Elbaum : How developers search for code: A case study. *In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 191–201, New York, NY, USA, 2015. Association for Computing Machinery.

[116] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy et Cristina V. Lopes : Sourcerercc: Scaling code clone detection to big code. *In Proceedings of the 38th International Conference on Software Engineering (ICSE'15)*, pages 1157–1168, 2015.

[117] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy et Cristina V Lopes : Sourcerercc: Scaling code clone detection to big-code. *In Proceedings of the 38th International Conference on Software Engineering*, pages 1157–1168, 2016.

[118] Bharat Bhusan Sau et Vineeth N Balasubramanian : Deep model compression: Distilling knowledge from noisy teachers. *arXiv preprint arXiv:1610.09650*, 2016.

[119] Chengchao Shen, Xinchao Wang, Jie Song, Li Sun et Mingli Song : Amalgamating knowledge towards comprehensive classification. *In Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3068–3075, 2019.

[120] Jieke Shi, Zhou Yang, Bowen Xu, Hong Jin Kang et David Lo : Compressing pre-trained models of code into 3 mb. *In 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[121] G Shobha, Ajay Rana, Vineet Kansal et Sarvesh Tanwar : Code clone detection—a systematic review. *Emerging Technologies in Data Mining and Information Security: Proceedings of IEMIS 2020, Volume 2*, pages 645–655, 2021.

[122] Connor Shorten et Taghi M Khoshgoftaar : A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.

[123] Jing Kai Siow, Shangqing Liu, Xiaofei Xie, Guozhu Meng et Yang Liu : Learning program semantics with code representations: An empirical study. *In 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 554–565. IEEE, 2022.

[124] Raphael Sirres, Tegawendé F Bissyandé, Dongsun Kim, David Lo, Jacques Klein, Kisub Kim et Yves Le Traon : Augmenting and structuring user queries to support efficient free-form code search. *In Proceedings of the 40th international conference on software engineering*, pages 945–945, 2018.

[125] Kathryn T. Stolee, Sebastian Elbaum et Daniel Dobos : Solving the search for source code. *ACM Trans. Softw. Eng. Methodol.*, 23(3), jun 2014.

[126] Zhensu Sun, Li Li, Yan Liu, Xiaoning Du et Li Li : On the importance of building high-quality training datasets for neural code search. *In Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1609–1620, New York, NY, USA, 2022. Association for Computing Machinery.

[127] Jeffrey Svajlenko et Chanchal K Roy : Evaluating clone detection tools with bigclonebench. *In 2015 IEEE international conference on software maintenance and evolution (ICSME)*, pages 131–140. IEEE, 2015.

[128] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow et Rob Fergus : Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[129] Robert Tairas : Clone detection and refactoring. *In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 780–781, 2006.

[130] Yuandong Tian, Xinlei Chen et Surya Ganguli : Understanding self-supervised learning dynamics without contrastive pairs. *In International Conference on Machine Learning*, pages 10268–10278. PMLR, 2021.

[131] Laurens Van der Maaten et Geoffrey Hinton : Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[132] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser et Illia Polosukhin : Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[133] Ming Wang, Pengcheng Wang, Kangqi Yin, Haoyu Cheng, Yun Xu et Chanchal K. Roy : Lvmapper: A large-variance clone detector using sequencing alignment approach. *In IEEE Access*, volume 8, pages 27986–27997, 2020.

[134] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu et Chanchal K. Roy : Ccaligner: A token based large-gap clone detector. *In Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, pages 1066–1077, 2018.

[135] Wenhan Wang, Ge Li, Bo Ma, Xin Xia et Zhi Jin : Detecting code clones with graph neural network and flow-augmented abstract syntax tree. *In 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE, 2020.

[136] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu et Xin Jiang : Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556*, 2021.

[137] Xin Wang, Yasheng Wang, Yao Wan, Jiawei Wang, Pingyi Zhou, Li Li, Hao Wu et Jin Liu : Code-mvp: Learning to represent source code from multiple views with contrastive pre-training. *arXiv preprint arXiv:2205.02029*, 2022.

[138] Yue Wang, Weishi Wang, Shafiq Joty et Steven CH Hoi : Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[139] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran et Denys Poshyvanyk : A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–58, 2022.

[140] Martin White, Michele Tufano, Christopher Vendome et Denys Poshyvanyk : Deep learning code fragments for code clone detection. *In 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98. IEEE, 2016.

[141] Wikipedia : Google code search – wikipedia, the free encyclopedia, 2019. [Online; accessed 24-August-2019].

[142] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest et Alexander M. Rush : Huggingface's transformers: State-of-the-art natural language processing, 2020.

[143] Chunqiu Steven Xia et Lingming Zhang : Conversational automated program repair. *arXiv preprint arXiv:2301.13246*, 2023.

[144] Frank F Xu, Uri Alon, Graham Neubig et Vincent Josua Hellendoorn : A systematic evaluation of large language models of code. *In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.

[145] Yufei Xu, Qiming Zhang, Jing Zhang et Dacheng Tao : Vitae: Vision transformer advanced by exploring intrinsic inductive bias. *Advances in Neural Information Processing Systems*, 34:28522–28535, 2021.

[146] Michihiro Yasunaga et Percy Liang : Break-it-fix-it: Unsupervised learning for program repair, 2021.

[147] Shan You, Chang Xu, Chao Xu et Dacheng Tao : Learning from multiple teacher networks. *In Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1285–1294, 2017.

[148] Shiwen Yu, Ting Wang et Ji Wang : Data augmentation by program transformation. *Journal of Systems and Software*, 190:111304, 2022.

[149] Yaodong Yu, Kwan Ho Ryan Chan, Chong You, Chaobing Song et Yi Ma : Learning diverse and discriminative representations via the principle of maximal coding rate reduction. *Advances in Neural Information Processing Systems*, 33:9422–9434, 2020.

[150] Jure Zbontar, Li Jing, Ishan Misra, Yann LeCun et Stéphane Deny : Barlow twins: Self-supervised learning via redundancy reduction. *In International Conference on Machine Learning*, pages 12310–12320. PMLR, 2021.

[151] Aiping Zhang, Liming Fang, Chunpeng Ge, Piji Li et Zhe Liu : Efficient transformer with code token learner for code clone detection. *Journal of Systems and Software*, 197:111557, 2023.

[152] Jingfeng Zhang, Haiwen Hong, Yin Zhang, Yao Wan, Ye Liu et Yulei Sui : Disentangled code representation learning for multiple programming languages. *In Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 4454–4466, 2021.