UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE APPLIQUÉES

PAR
MOUHAMADOU LAMINE SARR

EVALUATION COMPARATIVE D'ARCHITECTURES TRANSFORMER POUR LA PRÉDICTION DE DÉFAUTS LOGICIELS : ARBRES SYNTAXIQUES ABSTRAITS VERSUS MÉTRIQUES ORIENTÉES OBJET

Janvier 2026

Résumé

La prédiction de défauts logiciels(SDP) constitue un enjeu majeur pour l'industrie du développement logiciel, permettant d'identifier proactivement les composants susceptibles de contenir des bogues et d'optimiser l'allocation des ressources de test. Traditionnellement, cette prédiction repose sur des métriques orientées objet, notamment la suite de Chidamber et Kemerer (CK), qui ont démontré leur efficacité au fil de plusieurs décennies de recherche empirique.

Avec l'émergence de l'apprentissage profond, de nouvelles approches exploitant les caractéristiques syntaxiques du code source à travers les arbres syntaxiques abstraits (AST) ont été proposées, promettant de capturer des informations sémantiques complémentaires. Ce mémoire propose une analyse comparative rigoureuse entre les approches traditionnelles basées sur les métriques orientées objet, les méthodes modernes utilisant les représentations AST, ainsi que leurs combinaisons hybrides.

Les différentes configurations ont été évaluées sur plusieurs projets open

source Java à l'aide d'architectures d'apprentissage profond de type *encoder-only* et *encoder–decoder*, et selon des métriques d'évaluation standardisées. Les résultats ont montré que les métriques orientées objet conservent une supériorité notable en termes de performance prédictive, tandis que l'ajout d'informations syntaxiques via les AST n'apporte pas d'amélioration significative justifiant le surcoût computationnel associé.

Ces travaux confirment la valeur durable des approches fondées sur les métriques orientées objet pour la prédiction de défauts logiciels, tout en soulignant les limites actuelles des représentations syntaxiques et la nécessité d'explorer des approches multimodales plus intégrées.

Mots-clés : prédiction de défauts logiciels, métriques orientées objet, arbre syntaxique abstrait (AST), apprentissage profond, modèles Transformer, hybridation.

Abstract

Software defect prediction(SDP) remains a major challenge in the software engineering field, as it enables the proactive identification of components likely to contain bugs and optimizes the allocation of testing resources. Traditionally, this prediction relies on object-oriented metrics, notably the Chidamber and Kemerer (CK) metrics suite, which has demonstrated their effectiveness over several decades of empirical research.

With the rise of deep learning, new approaches have emerged that exploit the syntactic characteristics of source code through Abstract Syntax Trees (AST), aiming to capture complementary semantic information. This thesis presents a rigorous comparative analysis between traditional approaches based on object-oriented metrics, modern methods using AST representations, and their hybrid combinations.

The proposed configurations were evaluated on several open-source Java projects using deep learning architectures of the *encoder-only* and *encoder–decoder* types, assessed through standardized evaluation metrics. The

results indicate that object-oriented metrics maintain a clear predictive advantage, while the inclusion of syntactic information via AST representations does not provide a significant improvement justifying the associated computational cost.

These findings confirm the enduring value of metric-based approaches for software defect prediction while highlighting the current limitations of syntax-based representations and the need to explore more integrated multimodal approaches in future work.

Keywords : software defect prediction(SDP), object-oriented metrics, abstract syntax tree (AST), deep learning, Transformer models, hybridization.

Remerciements

Je tiens tout d'abord à exprimer ma profonde et sincère gratitude à mon directeur de recherche, **Monsieur Fadel Toure**, ainsi qu'à mon codirecteur, **Monsieur Mourad Badri**. Leur disponibilité constante, leur exigence scientifique et la qualité de leur encadrement ont été déterminantes dans la réalisation de ce mémoire. Leurs conseils éclairés et leur accompagnement rigoureux ont grandement contribué à la qualité de ce travail.

Je souhaite également adresser un hommage particulier à mon oncle, mentor et ami, **Mactar Diop**. C'est à lui que je dois mes premiers pas en informatique. Son soutien indéfectible, ses conseils avisés et la confiance qu'il m'a toujours accordée ont joué un rôle fondamental dans mon parcours académique et professionnel.

Ma reconnaissance va également à mon oncle **Moustapha Diop** ainsi qu'à sa famille. Leur accueil généreux et leur bienveillance depuis mon arrivée au Canada ont constitué un appui précieux et m'ont permis de poursuivre mes études dans un environnement stable et favorable.

Je souhaite enfin exprimer ma gratitude la plus profonde à ma famille. À mes parents, pour leurs bénédictions, leurs sacrifices et leurs encouragements constants, qui ont toujours été une source de motivation et de force.

À ma femme, **Fatoumata Kine Diop Thione**, je témoigne toute ma reconnaissance pour sa patience, ses sacrifices et son soutien sans faille tout au long de ces années d'études. Sa présence et son engagement à mes côtés ont été essentiels.

À ma fille, **Mame Khary Sarr**, dont la simple présence illumine mon quotidien et me rappelle chaque jour le sens et la valeur de mes efforts.

Table des matières

Liste des tableaux

Liste des abréviations

AdamW	Adam avec décroissance de poids
AI	Intelligence artificielle
API	Interface de programmation d'applications
AUC	Aire sous la courbe
BERT	Représentations bidirectionnelles d'encodeurs à partir de Transformers
CNN	Réseau de neurones convolutionnels
CV	Validation croisée
DL	Apprentissage profond
FN	Faux négatif
FP	Faux positif
GRU	Unité récurrente à portes
IDE	Environnement de développement intégré
IQR	Écart interquartile
JDT	Outils de développement Java
LR	Taux d'apprentissage

LSTM	Mémoire à long et court terme
ML	Apprentissage automatique
MSE	Erreur quadratique moyenne
NLP	Traitement automatique du langage naturel
PR	Précision–Rappel
QA	Assurance qualité
ReLU	Unité linéaire rectifiée
RNN	Réseau de neurones récurrents
ROC	Courbe caractéristique de fonctionnement du récepteur
SDK	Kit de développement logiciel
SDP	Software Defects Prediction
SE	Génie logiciel
SGD	Descente de gradient stochastique
SMOTE	Technique de suréchantillonnage synthétique des minorités
TN	Vrai négatif
TP	Vrai positif
UI	Interface utilisateur
UX	Expérience utilisateur

Table des figures

Chapitre 1

INTRODUCTION

1.1 Contexte

La qualité du logiciel constitue un enjeu fondamental en ingénierie logicielle, particulièrement dans les systèmes critiques où la moindre erreur peut engendrer des conséquences graves sur les plans économique, opérationnel et sécuritaire. Dans ce contexte, la détection précoce des défauts logiciels représente un levier essentiel pour assurer la fiabilité des systèmes logiciels, leur pérénité et réduire les coûts de maintenance (Hall et al., 2012).

Historiquement, la prédiction des défauts en orienté objet (OO)s'appuie principalement sur les métriques orientées objet, telles que la suite de Chidamber et Kemerer (Chidamber et al., 1994) incluant le couplage (CBO), la cohésion (LCOM), la complexité pondérée des méthodes (WMC), ou encore la profondeur d'héritage (DIT). Ces indicateurs fournissent une vision quantitative du code source basée sur sa structure architecturale. Bien qu'efficaces et largement validées empiriquement (Basili et al., 1996 ; Subramanyam et al., 2003), ces métriques restent limitées dans leur capacité à capturer la richesse syntaxique et les subtilités structurelles du programme.

Parallèlement, l'essor récent de l'apprentissage profond, particulièrement des architectures Transformer (Vaswani et al., 2017), a révolutionné le traitement des séquences en exploitant des mécanismes d'attention sophistiqués. Ces modèles ont démontré des succès remarquables en traitement du langage naturel (Devlin et al., 2018 ; Brown et al., 2020) et commencent à être adaptés au domaine logiciel (Z. Feng et al., 2020 ; Y. Wang et al., 2021). Leur capacité à capturer des dépendances à long terme et à traiter des séquences de longueur variable présente un potentiel significatif pour l'analyse de code source.

Pour exploiter cette richesse syntaxique, l'analyse du code via les arbres syntaxiques abstraits (AST) apparaît comme une approche prometteuse. Les AST permettent de modéliser le code source sous forme d'arbre hiérarchique, préservant fidèlement sa structure grammaticale tout en abstrayant les détails lexicaux superflus. Cette représentation capture naturellement les relations entre entités du code : déclarations de classes, invocations de méthodes, structures de contrôle, etc.

L'hypothèse centrale de cette recherche réside dans la complémentarité potentielle entre ces deux sources d'information : les métriques orientées objet, qui quantifient la complexité architecturale, et les AST, qui préservent la richesse syntaxique. Cette synergie, exploitée via des architectures Transformer adaptées, pourrait conduire à des modèles de prédiction de défauts plus performants et robustes.

1.2 Problématique

Malgré trois décennies de recherche en prédiction de défauts logiciels, plusieurs défis fondamentaux persistent. Les métriques orientées objet traditionnelles, bien que robustes et largement utilisées, ne capturent qu'une partie limitée des informations contenues dans le code source (Malhotra, 2015). En particulier, elles négligent les aspects sémantiques,

les structures syntaxiques complexes ainsi que les interactions fines entre les différentes entités du programme.

À l'inverse, les approches basées sur l'exploitation directe du code source à l'aide de techniques d'apprentissage profond offrent la possibilité d'apprendre des représentations plus riches. Toutefois, ces approches se heurtent à des difficultés importantes liées à la représentation du code et à la généralisation des modèles (Allamanis et al., 2018). La variabilité stylistique entre projets, la diversité des conventions de codage ainsi que la complexité structurelle du code contribuent à créer un espace de caractéristiques hétérogène difficile à modéliser efficacement.

Dans ce contexte, les représentations basées sur les arbres syntaxiques abstraits (AST) ont émergé comme une alternative prometteuse (White et al., 2016 ; Dam et al., 2018). Elles permettent de capturer explicitement la structure syntaxique du code. Cependant, les travaux existants restent souvent limités à des contextes intra-projet, ce qui soulève des interrogations quant à leur capacité de généralisation dans des scénarios inter-projets.

Par ailleurs, l'intégration conjointe de sources d'information hétérogènes, telles que les représentations syntaxiques et les métriques orientées objet, demeure un défi ouvert tant sur le plan architectural que méthodologique.

Dans ce cadre, les architectures de type Transformer (Vaswani et al., 2017) apparaissent comme une piste prometteuse. Leur mécanisme d'attention permet de modéliser des dépendances complexes au sein de séquences, ce qui les rend particulièrement adaptées à l'analyse de structures issues du code source.

Ainsi, la question centrale de cette recherche est formulée comme suit :

Dans quelle mesure les architectures Transformer peuvent-elles exploiter efficacement les arbres syntaxiques abstraits, seuls ou combinés aux métriques orientées objet, pour

améliorer la prédiction de défauts logiciels dans un contexte de généralisation inter-projets ?

Cette problématique soulève plusieurs enjeux. D'un point de vue théorique, elle interroge la transférabilité des modèles issus du traitement du langage naturel vers l'analyse de code. D'un point de vue méthodologique, elle nécessite l'identification de stratégies efficaces pour l'intégration de données hétérogènes. Enfin, d'un point de vue pratique, elle vise à déterminer les conditions dans lesquelles ces approches peuvent offrir un avantage par rapport aux méthodes traditionnelles.

Afin de répondre à cette problématique, cette étude propose et évalue quatre configurations basées sur des architectures Transformer : (i) une approche exploitant uniquement les AST, (ii) une approche basée uniquement sur les métriques orientées objet, (iii) une architecture hybride de type encoder-encoder, et (iv) une architecture encoder-decoder combinant ces deux sources d'information.

1.3 Questions de recherche

Afin de répondre à la problématique, cette étude s'articule autour de quatre questions de recherche visant une évaluation empirique systématique :

Q1 : Expressivité des représentations AST

Dans quelle mesure les arbres syntaxiques abstraits permettent-ils, à eux seuls, de représenter efficacement le code source pour la prédiction de défauts dans un contexte inter-projets ?

Q2 : Apport des approches hybrides

L'intégration des représentations syntaxiques (AST) avec les métriques orientées objet permet-elle d'améliorer significativement les performances par rapport aux approches uni-

modales ?

Q3 : Choix architectural

Quelle architecture Transformer (encoder-only, encoder-encoder, encoder-decoder) permet de mieux exploiter la complémentarité entre AST et métriques orientées objet ?

Q4 : Généralisation inter-projets

Dans quelle mesure les modèles proposés sont-ils robustes face à la variabilité des projets logiciels (domaines, styles de codage, tailles et complexité) ?

Ces questions permettent de structurer l'analyse expérimentale et de fournir une évaluation comparative rigoureuse des approches proposées.

1.3.1 Hypothèses de recherche

Afin de structurer l'analyse expérimentale, les hypothèses suivantes sont formulées :

— H1 : Les arbres de syntaxe abstraite (AST) fournissent une représentation pertinente pour la prédiction inter-projets.

— H2 : La combinaison des métriques orientées objet et des AST peut améliorer les performances de prédiction.

— H3 : L'architecture Encoder–Encoder est supérieure à l'architecture Encoder–Decoder.

1.4 Organisation du Mémoire

Ce mémoire est constitué de six chapitres contribuant à construire une réponse empirique et théorique à la problématique soulevée.

Le **chapitre 1** établit les fondements de la recherche en présentant le contexte scientifique, la problématique centrale, les questions de recherche spécifiques et l'organisation générale du travail.

Le **chapitre 2** propose un état de l'art structuré autour de quatre axes complémentaires : l'évolution des approches de prédiction de défauts, les métriques orientées objet et leurs limitations, l'émergence des représentations syntaxiques via les AST, et l'application des architectures Transformer en génie logiciel. Ce chapitre identifie les lacunes dans la littérature existante et positionne nos contributions spécifiques.

Le **chapitre 3** détaille les architectures Transformer développées pour cette recherche. Il présente les principes fondamentaux des mécanismes d'attention, puis décrit quatre configurations spécifiques : encoder-only pour AST seuls, encoder-only pour métriques seules, encoder-encoder combinant les deux modalités, et encoder-decoder pour leur intégration asymétrique. Les justifications techniques et les paramètres architecturaux sont explicités.

Le **chapitre 4** expose la méthodologie expérimentale complète. Il décrit la sélection et la préparation des données (projets Apache Ant, Camel, JEdit, Log4j et Xerces), les stratégies de prétraitement spécialisées pour chaque modalité, le protocole de validation inter-projets adopté, et les métriques d'évaluation retenues. Une attention particulière est portée à la reproductibilité et à la validité scientifique des procédures.

Le **chapitre 5** présente les résultats expérimentaux obtenus selon le protocole défini. Chaque configuration est analysée individuellement, suivie d'une synthèse comparative.

Les résultats obtenus mettent en évidence des observations partiellement contre-intuitives, notamment la performance persistante des métriques orientées objet par rapport aux représentations basées sur les AST, ainsi que l'absence d'amélioration systématique des approches hybrides. Ces observations motivent une analyse approfondie, orientée vers des explications théoriques et pratiques.

Le **chapitre 6** conclut le travail par une discussion critique des implications théoriques et pratiques des résultats. Il examine les menaces à la validité, propose des directions de recherche futures, et formule des recommandations pour les praticiens. Cette synthèse replace les contributions dans le contexte plus large de l'évolution des techniques de prédiction de défauts logiciels.

Chapitre 2

État de l'art

Introduction

La prédiction de défauts logiciels (Software Defect Prediction, SDP) a pour objectif d'anticiper, à partir d'artefacts de développement, les modules susceptibles d'être défectueux afin d'optimiser l'allocation des efforts de test et de maintenance. Les approches classiques s'appuient sur des métriques dérivées du code et du processus, tandis que les approches récentes exploitent des représentations apprises (apprentissage profond) et des modèles Transformers pré-entraînés sur de larges corpus de code. Cette section passe en revue (i) les synthèses et méta-analyses fondatrices, (ii) les métriques orientées objet (OO), (iii) les représentations syntaxiques du code (AST) et leurs dérivés, (iv) l'apprentissage profond appliqué à la SDP, (v) les modèles Transformers pré-entraînés pour le code, (vi) les approches hybrides/multi-modales, et (vii) les enjeux de généralisation et de validation inter-projets.

2.1 Panorama de la littérature et synthèse

Les revues systématiques ont structuré le champ en identifiant les facteurs d'influence et les bonnes pratiques d'évaluation. Hall et al. (2012) montrent, à partir d'une méta-analyse, l'importance de protocoles d'évaluation rigoureux (séparation stricte entraînement/test, mesures robustes, gestion du déséquilibre), soulignant la grande variabilité des performances selon les jeux de données et les métriques utilisées. Malhotra (2015) dresse une cartographie des techniques d'apprentissage (arbres de décision, SVM, réseaux neuronaux, etc.) et conclue à l'absence de « meilleur algorithme universel », plaidant pour des comparaisons contextualisées et des ensembles de données diversifiés. Radjenović et al. (2013) se concentrent sur les métriques de code (OO, complexité, taille) et mettent en évidence la redondance et la colinéarité entre certaines mesures, recommandant des sélections de variables et des analyses de sensibilité. Des synthèses plus récentes, telles que Liubchenko et al. (2023) et Abdu et al. (2022), intègrent les avancées en apprentissage profond et en pré-entraînement sur le code, soulignant le rôle croissant des représentations sémantiques apprises (embeddings) et des modèles Transformers.

2.2 Métriques orientées objet pour la SDP

Les métriques OO constituent l'un des piliers historiques de la SDP. La suite CK (*Chidamber & Kemerer*) – WMC, DIT, NOC, CBO, RFC, LCOM – a fourni un cadre conceptuel pour quantifier la complexité, le couplage et la cohésion des conceptions OO (Chidamber et al., 1994). Basili et al. (1996) validant empiriquement la capacité de ces métriques à prédire la qualité (défauts, effort), tandis que Briand et al. (2000) étudient systématiquement les liens entre mesures de conception et qualité observée. Subramanyam et al. (2003) confirment, sur des contextes industriels, la pertinence d'un sous-ensemble de ces métriques pour expliquer la variabilité des défauts. Plus récemment, les travaux sur les *bad*

smells (*code smells*) ont mobilisé l'apprentissage pour leur détection et leur corrélation avec la défectuosité (Moudache et al., 2022). Enfin, des études comparatives soulignent que la seule couche OO n'épuise pas l'information utile à la SDP et qu'un enrichissement par d'autres vues (processus, sémantique, syntaxe) améliore souvent la performance (Singh et al., 2013 ; Radjenović et al., 2013).

2.3 Représentations syntaxiques du code : AST et dérivés

Les arbres syntaxiques abstraits (AST) encodent la structure d'un programme et constituent une représentation pivot pour l'analyse de code (Aho et al., 2006). L'essor de l'« apprentissage pour le code » a stimulé des projections séquentielles ou chemin-basées des AST. Par exemple, Kovalenko et al. (2019) proposent *PathMiner* pour extraire des représentations chemin-basées (paths) capturant des dépendances structurelles. D'autres travaux apprennent des styles de programmation ou des motifs structuraux directement à partir du code source (Yee-King et al., 2020). Les chaînes de Markov d'ordre supérieur offrent un cadre pour modéliser des dépendances plus longues ; cependant, des résultats empiriques montrent que l'augmentation de l'ordre n'améliore pas systématiquement les performances, l'ordre 1 constituant souvent le meilleur compromis (Gorchakov et al., 2023). Enfin, l'AST s'est avéré discriminant au-delà de la SDP, notamment pour la détection d'abus ou de scripts malveillants (Rusak et al., 2018), ce qui suggère une transférabilité des signaux syntaxiques.

2.4 Apprentissage profond appliqué à la SDP

Les premières applications de l'apprentissage profond en génie logiciel ont porté sur des tâches similaires, telles que la détection de codes présentant des fonctionnalités proches (White et al., 2016). Ces travaux ont mis en évidence la capacité des représentations apprises automatiquement à surpasser les descripteurs manuels traditionnels. En SDP, (S. Wang et al., 2016) apprennent automatiquement des caractéristiques sémantiques pertinentes pour la prédiction de défauts, tandis que (J. Li et al., 2017) exploitent des réseaux convolutionnels (CNN) pour extraire des motifs à partir de représentations textuelles/structurées du code. (Dam et al., 2018) proposent une approche neuronale de bout-en-bout montrant des gains significatifs, à condition de calibrer soigneusement les hyperparamètres et de traiter le déséquilibre des classes—problématique récurrente en SDP.

2.5 Transformers et modèles pré-entraînés pour le code

L'introduction des architectures de type Transformer (Vaswani et al., 2017) a profondément fait évoluer les méthodes de représentation en apprentissage automatique. Dans le domaine du code source, la revue de la littérature proposée par Chen et al. (2021) recense un large éventail de tâches, telles que le résumé, la complétion, la réparation et la classification de code, tout en mettant en évidence l'impact du pré-entraînement multi-objectif sur les performances des modèles. *CodeBERT* (Z. Feng et al., 2020) et *CodeT5* (Y. Wang et al., 2021) illustrent l'efficacité d'un pré-entraînement *span-aware*/identifiant-sensible pour capturer la sémantique du code, réutilisable en aval (fine-tuning) pour la SDP et la détection de vulnérabilités. Des travaux applicatifs, tels que Y. Li et al. (2022), exploitent la capacité des modèles T5-like à réécrire du code pour la réparation de vulnérabilités. La littérature de synthèse sur la détection de vulnérabilités par apprentissage

profond souligne néanmoins des défis de généralisation, de bruit d'annotation et de biais de données (Chakraborty et al., 2021).

2.6 Approches hybrides et multi-modales

Pour surmonter les limites d'une seule vue (OO *vs* AST *vs* historique), plusieurs travaux explorent des architectures hybrides et des ensembles. Zhang et al. (2019) combinent des auto-encodeurs débruiteurs empilés avec des *ensembles* en deux étapes, obtenant des gains de robustesse. Les synthèses sur les méthodes d'ensembles confirment l'intérêt de la diversité des apprenants et des représentations (Sejm et al., 2017). Les approches multi-modales fusionnent métriques OO, sémantique du code et signaux de processus pour améliorer la SDP (Z. Li et al., 2019). Parallèlement, la communauté a étudié l'ingénierie de la qualité des modèles eux-mêmes (localisation de fautes dans les réseaux) (Mohan et al., 2018), préfigurant des cadres d'auditabilité des pipelines d'apprentissage pour la SDP.

2.7 Généralisation et validation inter-projets

Au-delà des performances intra-projet, la capacité à généraliser sur d'autres systèmes (validation inter-projets) est déterminante pour la transférabilité des modèles. Les travaux pionniers Zimmermann et al. (2009) montrent que les différences de domaine et de processus dégradent sensiblement la performance des modèles entraînés sur un projet et testés sur un autre. Turhan et al. (2009) et He et al. (2012) évaluent la valeur relative des données inter-entreprises, explorant des techniques de *filtering*, de sélection de voisins (projets similaires) et de normalisation pour atténuer le *dataset shift*. Ghotra et al. (2015) rappellent que les écarts de performance proviennent autant des choix mé-

thodologiques (prétraitement, métriques d'évaluation) que des algorithmes, plaidant pour des comparaisons équitables et reproductibles. Enfin, l'apprentissage de représentations plus générales (p. ex., *deep representation learning*) a été investigué pour la détection de vulnérabilités, avec des bénéfices en abstraction mais des risques de sur-apprentissage sur des artefacts de jeu de données (Russell et al., 2018).

2.8 Défis méthodologiques récurrents

Déséquilibre des classes. La proportion de modules défectueux est souvent faible, ce qui biaise l'exactitude et rend la SDP sensible aux seuils de décision. Les revues recommandent des métriques adaptées (F1, G-mean, AUC) et des stratégies de rééquilibrage (Hall et al., 2012 ; Malhotra, 2015).

Sélection et redondance des variables. Les métriques OO présentent des corrélations élevées ; des sélections guidées par la théorie et des analyses de variance/sensibilité sont nécessaires (Radjenović et al., 2013).

Reproductibilité et protocoles. Variabilité des splits, *leakage* potentiel, et hétérogénéité des jeux de données compliquent les comparaisons ; des *pipelines* transparents et des seeds fixés sont requis (Ghotra et al., 2015).

Transférabilité. Les écarts entre domaines, processus, langages et versions dégradent la performance inter-projets ; des normalisations, des adaptations de domaine et des pré-entraînements multi-projets sont des pistes prometteuses (Zimmermann et al., 2009 ; He et al., 2012).

Interprétabilité et explicabilité. Les modèles profonds (réseaux de neurones, ensembles) présentent une opacité intrinsèque qui limite la compréhension des décisions et freine l'adoption industrielle, particulièrement pour les approches multi-vues (OO, AST, historique). Des méthodes d'explicabilité post-hoc (SHAP, LIME), le développement de modèles intrinsèquement interprétables, et l'injection de connaissances symboliques constituent des pistes essentielles (Al-Smadi et al., 2023).

2.9 Synthèse et tendances

Trois tendances se dégagent :

1. **De l'artisanal vers l'apprenant :** descripteurs hand-crafted (OO) restent utiles et frugaux, mais les représentations apprises (CNN, Transformers) capturent des signaux sémantiques et structurels complémentaires (Dam et al., 2018 ; J. Li et al., 2017 ; Chen et al., 2021).
2. **Vers le multi-modale :** la combinaison de vues (OO + AST + historique/texte) et l'assemblage de modèles améliorent la robustesse (Z. Li et al., 2019 ; Zhang et al., 2019 ; Sejm et al., 2017).
3. **Du local au transférable :** le pré-entraînement sur de larges corpus de code et des protocoles inter-projets soigneusement conçus sont essentiels à la généralisation (Z. Feng et al., 2020 ; Y. Wang et al., 2021 ; Zimmermann et al., 2009).

2.10 Positionnement de la présente recherche

À la lumière de cet état de l'art, nous retenons que (i) les métriques OO fournissent une base explicable et peu coûteuse, (ii) les représentations AST apportent des signaux syntaxiques complémentaires, et (iii) les modèles Transformers, y compris en configuration

encoder-only, sont bien adaptés à l'encodage de longues séquences de tokens structurés (Vaswani et al., 2017 ; Chen et al., 2021). Nous posons l'hypothèse que la combinaison *AST-tokens + métriques OO* améliore la SDP par rapport à chaque modalité isolée, et qu'une validation inter-projets stricte est nécessaire pour évaluer la transférabilité (Zimmermann et al., 2009 ; He et al., 2012). Conformément aux recommandations méthodologiques des synthèses (Hall et al., 2012 ; Malhotra, 2015 ; Ghotra et al., 2015), notre protocole (chapitre. 4) adoptera des métriques adaptées au déséquilibre (F1, G-mean, Recall, Accuracy), des stratégies de rééquilibrage et des splits temporels/versionnels réalistes, en reportant l'ensemble des hyperparamètres et choix de prétraitement.

Conclusion

La SDP progresse selon trois axes complémentaires : l'amélioration des descripteurs (métrique OO versus représentations apprises), la fusion de modalités (multi-vue, ensembles), et la rigueur des protocoles (inter-projets, métriques adaptées). Les Transformers pré-entraînés ouvrent des perspectives fortes pour la généralisation, sous réserve d'une adaptation de domaine et d'une évaluation prudente. Notre travail s'inscrit dans cette trajectoire en explorant des Transformers alimentés par des séquences issues d'AST et enrichis par des métriques OO, avec une validation inter-projets visant la transférabilité.

Chapitre 3

ARCHITECTURE TRANSFORMER POUR LA PRÉDICTION DE FAUTES LOGICIELLES

3.1 Introduction

Les architectures de type Transformer, introduites par (Vaswani et al., 2017), ont révolutionné le domaine du traitement automatique du langage naturel grâce au mécanisme d'attention qui permet de capturer efficacement les dépendances à long terme dans les séquences. Dans le contexte de la prédiction des fautes logicielles, l'adaptation de cette architecture présente un intérêt particulier pour l'analyse des arbres syntaxiques abstraits (AST) et leur combinaison avec les métriques orientées objet.

Le choix exclusif des Transformers s'appuie sur leur capacité démontrée à traiter des sé-

quences de longueur variable et à capturer des relations complexes entre éléments distants, caractéristiques particulièrement pertinentes pour l'analyse de la structure syntaxique du code source (Chen et al., 2021 ; Y. Wang et al., 2021).

3.2 Principes fondamentaux des Transformers

3.2.1 Mécanisme d'attention

Le mécanisme d'attention constitue le cœur de l'architecture Transformer. Contrairement aux approches récurrentes traditionnelles, l'attention permet au modèle de considérer simultanément tous les éléments d'une séquence et de pondérer leur importance relative pour chaque position.

L'attention est calculée selon la formule :

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{3.1}$$

où Q, K et V représentent respectivement les matrices de requêtes (queries), clés (keys) et valeurs (values), et d_k la dimension des clés (Vaswani et al., 2017).

3.2.2 Attention multi-têtes

L'attention multi-têtes étend ce mécanisme en permettant au modèle de capturer différents types de relations simultanément :

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O \tag{3.2}$$

où chaque tête d'attention head_i est calculée comme :

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{3.3}$$

3.2.3 Encodage positionnel

Contrairement aux réseaux récurrents, les Transformers ne possèdent pas de notion intrinsèque de position. L'encodage positionnel compense cette limitation en ajoutant des informations de position aux représentations d'entrée :

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{3.4}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{3.5}$$

Cette approche sinusoïdale permet au modèle de distinguer l'ordre des tokens dans la séquence AST, information cruciale pour comprendre la structure du code (Vaswani et al., 2017).

3.3 Configuration Encoder-Only

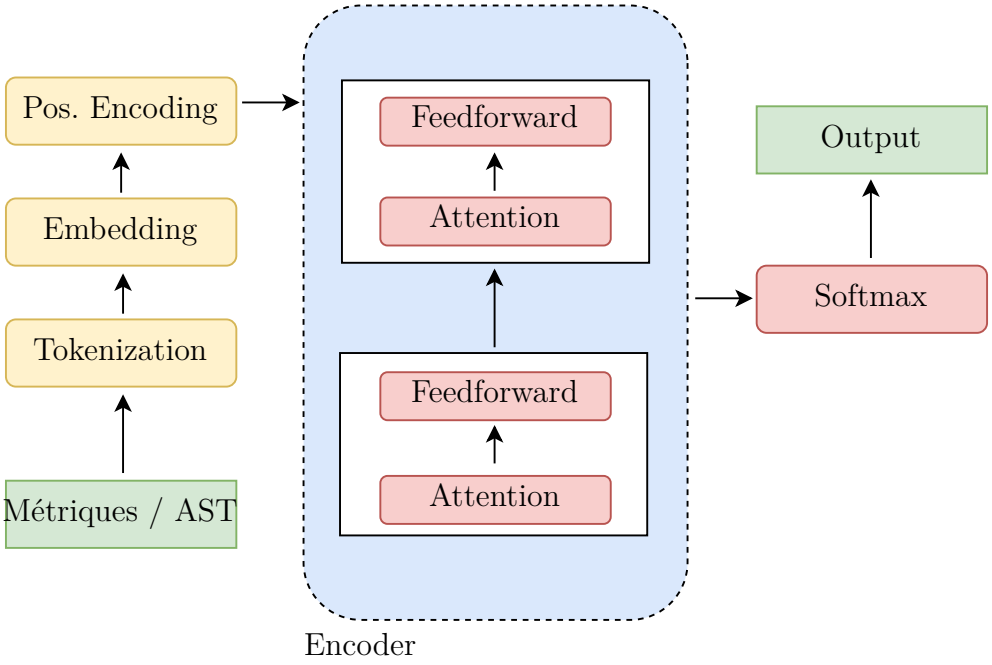

FIGURE 3.1 – Transformer Encoder-only — Metriques / AST

3.3.1 Architecture générale

La première famille de configurations explorées utilise uniquement la partie encodeur de l'architecture Transformer selon deux variantes : (i) traitement des séquences de tokens extraites des AST, et (ii) traitement des vecteurs de métriques orientées objet. Cette approche permet d'évaluer la contribution individuelle de chaque modalité d'information.

L'architecture de base se compose de :

— Une couche d'embedding qui transforme les entrées en vecteurs denses de dimension $d_{model} = 256$;

— N couches d'encodeur empilées (avec $N \in \{1, 2, 3\}$) ;

— Une couche de classification finale avec fonction softmax ;

3.3.2 Encoder-Only pour séquences AST

1. **Traitement des séquences syntaxiques**

   Les arbres syntaxiques abstraits sont préalablement linéarisés en séquences de tokens selon un parcours en profondeur. Chaque token correspond à un type de nœud spécifique (appel de méthode, déclaration de classe, structure de contrôle, etc.). La séquence résultante préserve l'information structurelle tout en étant compatible avec l'architecture séquentielle du Transformer.

   Les séquences sont normalisées à une longueur maximale de 512 tokens, correspondant à la moyenne observée dans notre jeu de données. Cette limitation permet un compromis entre expressivité et efficacité computationnelle.

2. **Mécanisme de prédiction AST**

   La couche de sortie utilise les représentations contextualisées produites par l'Encodeur. Un mécanisme de pooling global moyenné agrège les représentations de tous les tokens pour obtenir une représentation unique du fichier source :

$$h_{file}^{AST} = \frac{1}{L} \sum_{i=1}^{L} h_i^{AST} \tag{3.6}$$

   où h_i^{AST} représente la représentation du token i et L la longueur de la séquence AST.

3.3.3 Encoder-Only pour métriques orientées objet

Traitement des métriques orientées objet

La seconde variante de l'encoder-only traite directement le vecteur de métriques orientées objet. Ce vecteur de dimension fixe (correspondant au nombre de métriques disponibles) est projeté dans l'espace de dimension d_{model} .

Contrairement aux séquences AST de longueur variable, les métriques forment un vecteur de dimension constante. Chaque métrique (WMC, DIT, NOC, CBO, RFC, LCOM, etc.) est traitée comme une position spécifique dans une séquence courte et dense.

3.3.4 Comparaison des variantes encoder-only

Ces deux variantes permettent d'isoler l'apport de chaque modalité :

AST seuls : Capturent la richesse syntaxique et structurelle du code source, incluant les patterns complexes de contrôle de flux et les interactions entre entités.

Métriques seules : Fournissent une vue quantitative et agrégée de la complexité du code, basée sur des mesures établies en génie logiciel.

3.4 Configuration Encoder-Encoder pour données hybrides

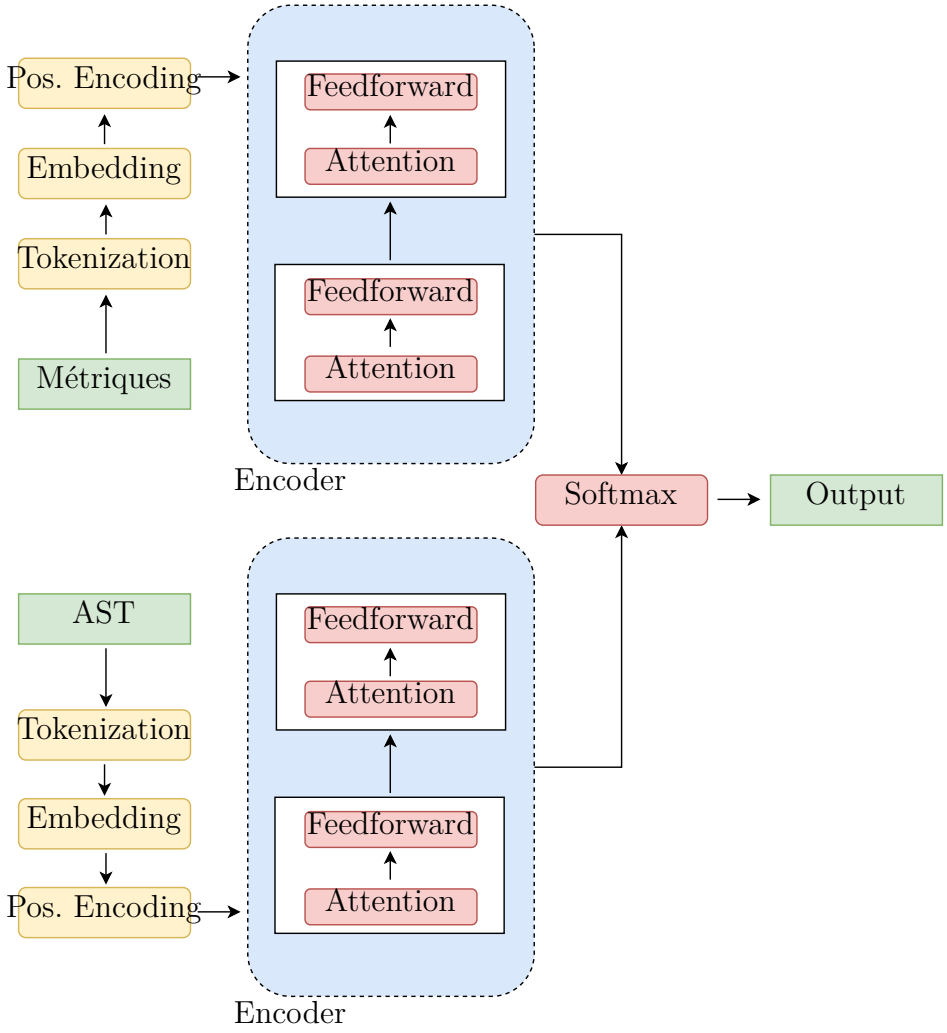

FIGURE 3.2 – Transformer Encoder-Encoder — Metriques + AST

3.4.1 Motivation et architecture

La troisième configuration étend les approches précédentes en intégrant les métriques orientées objet aux représentations syntaxiques. Cette approche encoder-encoder utilise deux flux d'encodage parallèles : un pour traiter les séquences AST et un autre pour les métriques numériques.

L'hypothèse sous-jacente est que les métriques orientées objet et les AST capturent des aspects complémentaires de la qualité du code : les premières quantifient la complexité structurelle tandis que les secondes préservent la richesse syntaxique.

3.4.2 Traitement dual des entrées

Encoder AST : Identique à la configuration précédente, cet encodeur traite les séquences de tokens syntaxiques et produit des représentations contextualisées.

Encoder Métriques : Un encoder spécialisé traite le vecteur de métriques orientées objet. Ce vecteur de dimension fixe (correspondant au nombre de métriques) est d'abord projeté dans l'espace de dimension d_{model} puis traité par un sous-ensemble de couches d'attention.

3.4.3 Fusion des représentations

Les représentations produites par les deux encoders sont fusionnées selon la stratégie de concaténation.

Les représentations des deux modalités sont concaténées puis passées à travers une couche dense :

$$h_{fused} = W[h_{AST}; h_{metrics}] + b \tag{3.7}$$

3.4.4 Architecture de classification

La classification finale s'appuie sur les représentations fusionnées. Une couche dense avec fonction d'activation ReLU précède la couche de sortie softmax :

$$p(faulty) = \mathsf{softmax}(W_2 \cdot \mathsf{ReLU}(W_1 \cdot h_{fused} + b_1) + b_2) \tag{3.8}$$

3.5 Configuration Encoder-Decoder pour données hybrides

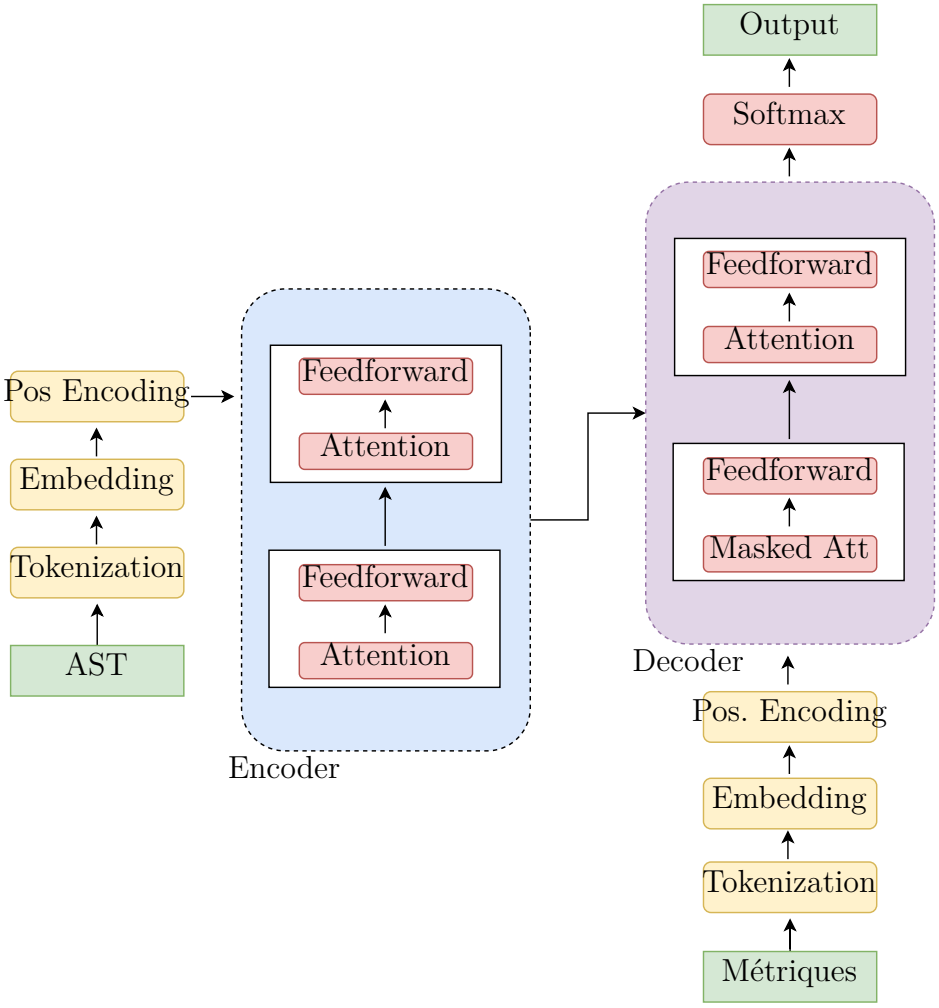

FIGURE 3.3 – Transformer Encoder-Decoder — AST + Metriques

3.5.1 Paradigme encoder-decoder

La quatrième configuration adopte le paradigme encoder-decoder complet, où l'encodeur traite les AST comme séquence d'entrée et le decodeur utilise les métriques orientées objet pour générer la prédiction finale. Cette approche asymétrique traite les deux modalités selon des rôles différents.

3.5.2 Encoder pour séquences AST

L'Encodeur suit la même architecture que dans les configurations précédentes, produisant des représentations contextualisées de la séquence AST d'entrée. Ces représentations encodent la structure syntaxique complète du fichier source.

3.5.3 Decoder guidé par les métriques

Le Decoder utilise les métriques orientées objet comme séquence de "décodage". Contrairement aux applications de traduction automatique, la séquence de sortie est déterministe et correspond au vecteur de métriques augmenté d'un token de classification.

3.5.4 Génération de la prédiction

Le Decoder génère progressivement une représentation enrichie qui culmine avec la prédiction de la classification. Cette approche séquentielle permet une intégration plus fine des informations métriques et syntaxiques.

3.6 Justification du choix architectural

3.6.1 Avantages des Transformers pour l'analyse de code

Le choix exclusif des architectures Transformer pour cette recherche repose sur plusieurs considérations techniques :

Parallélisation : Contrairement aux approches récurrentes, les Transformers permettent un traitement parallèle des séquences, réduisant significativement les temps d'entraînement sur les jeux de données volumineux.

Dépendances à long terme : Le mécanisme d'attention capture efficacement les relations entre éléments distants dans le code, comme les interactions entre méthodes définies dans des parties différentes d'un fichier.

Flexibilité architecturale : La modularité des Transformers facilite l'expérimentation avec différentes configurations (encoder-only, encoder-encoder, encoder-decoder) sans modifications fondamentales.

3.6.2 Comparaison avec les alternatives

Réseaux de neurones convolutionnels : Bien qu'efficaces pour capturer des patterns locaux, les CNN peinent à modéliser les dépendances à long terme caractéristiques de la structure du code.

Réseaux récurrents (LSTM, GRU) : Ces architectures souffrent de limitations computationnelles (traitement séquentiel) et de dégradation du gradient sur de longues

séquences.

Réseaux de neurones classiques : Inadaptés au traitement de séquences de longueur variable et incapables de capturer les relations séquentielles.

3.7 Conclusion

Ce chapitre a présenté quatre configurations d'architectures Transformer adaptées à la prédiction de fautes logicielles. Chaque configuration explore une stratégie différente d'exploitation et d'intégration des informations syntaxiques (AST) et métriques (orientées objet) :

1. **Encoder-only métriques** : Exploitation exclusive des métriques orientées objet ;
2. **Encoder-only AST** : Exploitation exclusive des représentations syntaxiques ;
3. **Encoder-encoder** : Traitement combinant les deux modalités ;
4. **Encoder-decoder** : Traitement asymétrique privilégiant l'interaction croisée ;

Ces architectures tirent parti des avantages intrinsèques des Transformers : capacité d'attention, traitement parallèle et flexibilité architecturale. Le chapitre suivant détaillera la méthodologie expérimentale permettant d'évaluer et de comparer ces quatre approches.

Chapitre 4

MÉTHODOLOGIE DE RECHERCHE

4.1 Introduction

Ce chapitre décrit en détail les données exploitées, les transformations appliquées avant l'entraînement, la construction des modèles de prédiction, le protocole de validation choisi ainsi que les métriques utilisées pour l'évaluation des performances. Une attention particulière est portée à la justification des choix méthodologiques, afin de montrer qu'ils reposent sur des considérations techniques et scientifiques rigoureuses.

Notre approche se distingue par l'exploration systématique de quatre configurations Transformer distinctes, permettant d'évaluer à la fois l'apport individuel et combiné des représentations syntaxiques (AST) et des métriques orientées objet dans un cadre expérimental contrôlé.

4.2 Présentation des données

4.2.1 Sélection des projets

Pour tester notre approche, nous avons utilisé des jeux de données issus de plusieurs projets logiciels open source largement étudiés dans la littérature (Jureczko et al., 2010 ; D'Ambros et al., 2010). Ces systèmes sont écrits en Java et présentent l'avantage d'être disponibles sous plusieurs versions successives, ce qui permet d'évaluer l'évolution de la qualité du code au fil du temps.

Les projets sélectionnés sont :

— **Apache Ant** : versions 1.4 à 1.7 - Outil de construction de projets Java ;
— **Apache Camel** : versions 1.0, 1.2, 1.4 et 1.6 - Framework d'intégration ;
— **JEdit** : versions 3.2, 4.0, 4.1, 4.2 et 4.3 - Éditeur de texte programmable ;
— **Log4j** : versions 1.0, 1.1 et 1.2 - Framework de logging ;
— **Xerces** : versions 1.1 à 1.4 - Parseur XML ;

Ce choix de projets offre une diversité représentative en termes de domaines d'application (outils de développement, frameworks, utilitaires), de tailles (de quelques milliers à plusieurs dizaines de milliers de lignes de code) et de complexités architecturales. Cette diversité est cruciale pour évaluer la capacité de généralisation des modèles proposés.

4.2.2 Caractéristiques des données

Les données combinent deux sources complémentaires d'information :

Métriques orientées objet : Extraites au niveau des classes Java, incluant la suite classique de Chidamber et Kemerer (Chidamber et al., 1994) : WMC (Weighted Methods per Class), DIT (Depth of Inheritance Tree), NOC (Number of Children), CBO (Coupling Between Objects), RFC (Response For a Class), et LCOM (Lack of Cohesion of Methods). Des métriques complémentaires sont également incluses : lignes de code (LOC), complexité cyclomatique, nombre de méthodes publiques, etc.

Arbres syntaxiques abstraits (AST) : Extraits directement du code source Java et transformés en séquences de tokens représentant la structure syntaxique des classes. Cette extraction préserve l'information hiérarchique tout en la rendant compatible avec les architectures séquentielles.

4.2.3 Labellisation des défauts

Dans le cadre de cette étude, la labellisation est réalisée par un processus de matching entre les noms de classes provenant de deux sources de données distinctes :

1. **Ensemble de données de métriques** : contenant les informations sur les défauts associés aux classes logicielles (étiquetées comme défectueuses ou non) ;
2. **Données AST (Abstract Syntax Tree)** : représentant la structure syntaxique du code ;

Le processus de labellisation suit la logique suivante : si une classe XY est identifiée comme défectueuse dans l'ensemble de données de métriques, alors la classe XY correspondante dans les données AST est automatiquement labellisée comme défectueuse.

Lors du croisement des jeux de données de métriques avec le code source correspondant, certaines incohérences apparaissent, notamment la présence d'entrées annotées (par

exemple, des classes de la forme `Outer$Inner`) qui ne correspondent pas à des fichiers `.java` distincts. Comme l'ont montré Jureczko et Madeyski, les jeux de données utilisés pour la prédiction de défauts intègrent non seulement des classes externes, mais également des classes imbriquées, représentées comme des entités indépendantes (Jureczko et al., 2010). Cette caractéristique rend l'appariement direct avec les fichiers sources non trivial. Peters et al. soulignent que de telles incohérences structurelles et de granularité peuvent introduire du bruit dans les expériences et affecter la validité des comparaisons entre approches (Peters et al., 2013). De son côté, Herbold insiste sur la nécessité d'aligner rigoureusement les unités d'analyse et les labels afin d'assurer la cohérence et la reproductibilité des résultats (Herbold et al., 2018). Enfin, Zimmermann et al. rappellent que la variabilité structurelle entre projets représente un facteur critique influençant la qualité des prédictions inter-projets (Zimmermann et al., 2009).

Dans ce contexte, nous avons adopté une approche stricte consistant à ne retenir que les fichiers réellement présents dans le code source de la version étudiée. Ce choix réduit légèrement la taille effective des ensembles, mais il garantit une correspondance sans ambiguïté entre le code analysé, les métriques extraites et les étiquettes associées, conformément aux recommandations de la littérature.

4.3 Architecture expérimentale

4.3.1 Configurations Transformer étudiées

Notre étude explore systématiquement quatre configurations d'architectures Transformer, chacune correspondant à une stratégie différente d'exploitation des données disponibles :

1. **Configuration 1 : Encoder-only AST**

   Cette configuration utilise uniquement la partie encoder de l'architecture Transformer pour traiter les séquences de tokens extraites des AST. Elle permet d'évaluer la capacité des représentations syntaxiques seules à prédire les défauts.

   Architecture : Encoder Transformer standard avec mécanisme d'attention multi-têtes, suivi d'une couche de pooling global et d'une tête de classification binaire.

   Entrées : Séquences de tokens AST de longueur maximale 512.

2. **Configuration 2 : Encoder-only Métriques**

   Cette variante traite exclusivement les vecteurs de métriques orientées objet en les adaptant au format séquentiel requis par l'architecture Transformer.

   Architecture : Encoder Transformer adapté pour traiter des vecteurs de caractéristiques de dimension fixe, avec projection initiale vers l'espace de représentation du modèle.

   Entrées : Vecteurs de métriques standardisées, traitées comme des séquences courtes avec encodage positionnel adapté.

3. **Configuration 3 : Encoder-encoder Hybride**

   Cette configuration emploie deux encodeurs parallèles : un pour les séquences AST et un autre pour les métriques, avec fusion des représentations avant la classification finale.

   Architecture : Double encodeur avec mécanismes de fusion explorés (concaténation).

Entrées : Traitement simultané des AST et métriques avec fusion des représentations contextualisées.

4. **Configuration 4 : Encoder-decoder Hybride**

   Cette approche utilise un encodeur pour traiter les AST et un decodeur qui exploite les métriques pour guider la génération de la prédiction finale.

   Architecture : Encoder standard pour les AST, Decoder utilisant les métriques comme séquence de décodage concaténée avec les représentations AST.

   Entrées : AST comme séquence d'entrée, métriques comme guide de décodage.

4.3.2 Paramètres architecturaux

L'ensemble des configurations partage les paramètres suivants :

— Dimension du modèle : $d_{model} = 512$;
— Nombre de têtes d'attention : $h \in \{2, 4, 8\}$;
— Nombre de couches : $N \in \{1, 2, 3\}$;
— Dimension des couches feed-forward : $d_{ff} = 64$;
— Fonction d'activation : ReLU ;
— Dropout : $\{0.2, 0.3\}$ pour la régularisation ;

Afin d'identifier la configuration la plus performante, plusieurs combinaisons de paramètres ont été évaluées en faisant varier le **nombre de têtes d'attention** ($h \in \{2, 4, 8\}$), le **nombre de couches** ($N \in \{1, 2, 3\}$) et le **taux de dropout** ($\{0.2, 0.3\}$). Chaque configuration a été testée selon le même protocole d'apprentissage inter-projets afin d'assurer la comparabilité des résultats. Les meilleures performances globales ont été obtenues avec

8 têtes d'attention, **2 couches** et un **taux de dropout de 0.2**, paramètres retenus pour l'ensemble des expérimentations finales.

4.4 Prétraitements et préparation des données

4.4.1 Traitement des arbres syntaxiques abstraits

1. **Extraction et linéarisation**

   L'extraction des AST s'appuie sur un script Python développé spécifiquement pour cette recherche utilisant la bibliothèque `javalang` pour parser le code source Java. Cette approche permet un contrôle précis sur les éléments syntaxiques extraits et leur représentation.

   Le processus d'extraction suit les étapes suivantes :

   (a) **Lecture du fichier source** : Chaque fichier Java est lu avec gestion automatique de l'encodage (fallback encoding) pour traiter les différents formats de caractères ;

   (b) **Analyse syntaxique** : Utilisation de `javalang.parse.parse()` pour construire l'arbre syntaxique complet ;

   (c) **Filtrage sélectif** : Extraction ciblée de quatre types de nœuds syntaxiques pertinents ;

   (d) **Gestion des erreurs** : Traitement des erreurs de syntaxe Java avec continuité du processus ;

   L'algorithme d'extraction parcourt l'AST et collecte spécifiquement :

- **Invocations de méthodes** (`MethodInvocation`) : Extraction du nom de la méthode via `node.member` ;
- **Déclarations de méthodes** (`MethodDeclaration`) : Extraction du nom de la méthode via `node.name` ;
- **Déclarations de classes** (`ClassDeclaration`) : Extraction du nom de la classe via `node.name` ;
- **Structures de contrôle** (`IfStatement`, `WhileStatement`, `ForStatement`) : Extraction du type de structure via `type(node).__name__` ;

Le choix de ces quatre types de nœuds AST a été motivé par leur fréquence dans le code et leur lien direct avec les interactions fonctionnelles.

Ce choix constitue cependant une simplification, dans la mesure où les nœuds liés aux structures de contrôle n'ont pas été inclus, ce qui pourrait limiter la capacité du modèle à capturer certains aspects de la complexité du code.

2. **Construction du vocabulaire**

   À partir de l'analyse de tous les fichiers source du corpus, nous avons construit un dictionnaire global de 11 154 tokens uniques. Chaque token correspond à un élément syntaxique spécifique identifié lors du parcours des AST. Cette approche garantit une couverture complète du vocabulaire syntaxique rencontré dans nos données.

   Le tableau 4.1 présente un échantillon représentatif de tokens extraits pour chaque catégorie :

3. **Normalisation des séquences**

   Pour assurer une compatibilité avec l'architecture Transformer et optimiser l'utilisation des ressources computationnelles, les séquences sont normalisées à une longueur maximale de 512 tokens. Cette valeur représente un compromis empiriquement validé :

TABLE 4.1 – Échantillon représentatif de tokens extraits du dictionnaire (4 par catégorie)

Catégorie	**Token**	**Index**
MethodInvocation	evaluateScript	257
	findInDir	265
	get	45
	put	63
MethodDeclaration	getUserProperties	0
	createJarSigner	1
	setJavafiles	2
	doCompilation	11
ClassDeclaration	Map	5191
	Type	1404
	Attribute	1550
	ExpressionBuilder	2057
ControlFlow	forDigit	1974
	errorWhileMapping	10937
	forClass	5826
	forName	1207

— **Séquences courtes** ($<$ 512 tokens) : Complétées par du padding avec des zéros (0) ;

— **Séquences longues** ($>$ 512 tokens) : Tronquées en conservant les 512 premiers tokens ;

Cette stratégie de troncature, bien qu'imparfaite, est justifiée par une contrainte de longueur séquentielle : la longueur moyenne des séquences de tokens par classe dans notre corpus est d'environ 500. Nous fixons donc la longueur d'entrée à 512 tokens (puissance de deux), ce qui couvre la plupart des cas tout en optimisant l'utilisation mémoire et le débit d'entraînement. Les séquences plus longues sont tronquées à 512 tokens et les plus courtes sont complétées par du *padding* (zéros).

4.4.2 Traitement des métriques orientées objet

1. **Nettoyage des données**

   Les métriques orientées objet sont utilisées directement sans traitement particulier des valeurs aberrantes. Seules les instances présentant des valeurs manquantes sont exclues du jeu de données pour garantir la cohérence des entrées.

2. **Normalisation et standardisation**

   Pour harmoniser les échelles hétérogènes des différentes métriques, nous appliquons une standardisation Z-score avec la méthode StandardScaler (Pedregosa et al., 2011) :

$$z = \frac{x - \mu}{\sigma} \tag{4.1}$$

   où x est la valeur originale, μ la moyenne et σ l'écart-type de la métrique considérée.

   Cette normalisation est appliquée séparément sur chaque ensemble d'entraînement, les paramètres (,) étant ensuite utilisés pour standardiser les ensembles de test correspondants.

3. **Encodage de la variable cible**

   La variable cible indiquant la présence ou l'absence de défauts (colonne "bug" ou "buggy") est encodée via la méthode One-Hot Encoding. Cette transformation crée deux variables binaires (classe 0 : non défectueux, classe 1 : défectueux) permettant l'utilisation de la fonction softmax pour la classification finale, évitant ainsi un cadre de régression inapproprié pour cette tâche de classification binaire.

4.4.3 Gestion du déséquilibre des classes

1. **Analyse du déséquilibre**

   Nos jeux de données présentent un déséquilibre significatif entre classes défectueuses et non-défectueuses, caractéristique typique des données de prédiction de défauts logiciels.

2. **Stratégie SMOTE**

   Pour traiter ce déséquilibre, nous employons la méthode SMOTE (Synthetic Minority Over-sampling Technique) (Chawla et al., 2002). SMOTE génère de nouvelles instances synthétiques de la classe minoritaire par interpolation linéaire entre une instance existante et ses k plus proches voisins :

$$x_{new} = x_i + \lambda \times (x_{neighbor} - x_i) \tag{4.2}$$

   où $\lambda \in [0, 1]$ est un facteur aléatoire et $x_{neighbor}$ un voisin sélectionné aléatoirement parmi les $k = 5$ plus proches.

   Application étendue : SMOTE est appliqué à la fois aux métriques orientées objet et aux séquences AST pendant la phase d'entraînement. Pour les AST, bien que l'interpolation dans l'espace discret des tokens ne soit pas directement applicable, nous utilisons SMOTE sur les représentations numériques correspondantes pour équilibrer les données d'entraînement.

   Validation de l'équilibrage : Après application de SMOTE, nous vérifions que le ratio des classes atteint approximativement 50/50, optimisant ainsi l'apprentissage des modèles.

4.5 Protocole de validation

4.5.1 Choix du protocole inter-projets

Notre étude adopte exclusivement un protocole de validation inter-projets, motivé par les considérations suivantes :

Réalisme applicatif : Le protocole inter-projets simule le scénario réaliste où un modèle entraîné sur des projets existants est appliqué à un nouveau projet non vu ;

Évaluation de la généralisation : Cette approche teste directement la capacité des modèles à généraliser au-delà des spécificités d'un projet particulier ;

Évitement du biais optimiste : Les protocoles intra-projet tendent à surestimer les performances en exploitant les similarités internes d'un même projet (Zimmermann et al., 2009) ;

4.5.2 Mise en œuvre du protocole

Pour chaque système cible dans notre corpus, nous appliquons la procédure suivante :

1. **Sélection du système test** : La dernière version disponible du système est isolée comme ensemble de test ;
2. **Construction de l'ensemble d'entraînement** : Toutes les versions des autres systèmes sont concaténées avec les versions précédentes du système de test pour former l'ensemble d'entraînement ;
3. **Évaluation finale** : Le modèle optimal est évalué sur l'ensemble de test isolé ;

Cette procédure est répétée pour chaque système, garantissant que tous les projets servent tour à tour de cible de test.

4.6 Métriques d'évaluation de performance

4.6.1 Matrice de confusion et métriques dérivées

L'évaluation s'appuie sur la matrice de confusion standard définie par quatre valeurs : VP (vrais positifs), VN (vrais négatifs), FP (faux positifs) et FN (faux négatifs). À partir de cette matrice, nous calculons les métriques suivantes :

1. **Exactitude (Accuracy)**

   L'exactitude mesure la proportion d'instances correctement classées :

$$Accuracy = \frac{VP + VN}{VP + VN + FP + FN} \tag{4.3}$$

   Bien qu'intuitive, cette métrique peut être trompeuse en présence de classes déséquilibrées, d'où la nécessité de métriques complémentaires.

2. **Rappel (Recall)**

   Le rappel (ou sensibilité) mesure la proportion de classes défectueuses correctement identifiées :

$$Recall = \frac{VP}{VP + FN} \tag{4.4}$$

Cette métrique est cruciale en prédiction de défauts car manquer un module défectueux peut avoir des conséquences graves.

3. **Précision (Precision)**

   La précision mesure la proportion de prédictions positives qui sont effectivement correctes :

$$Precision = \frac{VP}{VP + FP} \tag{4.5}$$

   Une précision élevée indique que le modèle génère peu de fausses alertes, aspect important pour l'acceptabilité pratique.

4. **F-mesure (F1-score)**

   La F-mesure combine précision et rappel via leur moyenne harmonique :

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4.6}$$

   Cette métrique fournit un indicateur équilibré particulièrement adapté aux classes déséquilibrées.

5. **Moyenne géométrique (G-mean)**

   La G-mean évalue l'équilibre entre la performance sur les classes positives et négatives :

$$G\text{-}mean = \sqrt{Recall \times Specificity} \tag{4.7}$$

où la spécificité est définie par :

$$Specificity = \frac{VN}{VN + FP} \tag{4.8}$$

4.7 Hyperparamètres et optimisation

4.7.1 Configuration d'optimisation

L'entraînement utilise l'optimiseur AdamW (Loshchilov et al., 2019) avec sa configuration par défaut, incluant un taux d'apprentissage de 0.001 et les coefficients de momentum standard ($\beta_1 = 0.9$, $\beta_2 = 0.999$). Aucune planification particulière du taux d'apprentissage n'est appliquée.

4.7.2 Critères d'arrêt et régularisation

Nombre maximum d'époques : 2500 époques maximum par configuration.

Early stopping : arrêt automatique si aucune amélioration de la perte de validation (`val_loss`) n'est observée pendant 500 époques consécutives (patience = 500).

Dropout : Taux de dropout variable ($\{0.2, 0.3\}$) dans toutes les couches pour prévenir le surapprentissage.

4.8 Environnement expérimental

4.8.1 Implémentation logicielle

Les expérimentations sont menées avec Python 3.8 et les bibliothèques suivantes :

— **PyTorch 1.12** : Framework d'apprentissage profond principal ;

— **Transformers :** Implémentation *from scratch* de l'encodeur Transformer Vaswani et al. (2017), sans modèle pré-entraîné ;

— **scikit-learn 1.1** : Préprocessing, métriques d'évaluation et validation croisée (Pedregosa et al., 2011) ;

— **imbalanced-learn 0.9** : Implémentation de SMOTE (Chawla et al., 2002) ;

— **Pandas 1.4** : Manipulation et analyse des données ;

— **NumPy 1.21** : Calculs numériques optimisés ;

— **Matplotlib/Seaborn** : Visualisation des résultats ;

4.8.2 Infrastructure expérimentale

Configuration matérielle :

— Processeur : Intel Core i5-12400F (6 cœurs, 12 threads) ;

— Mémoire vive : 16 Go DDR4-3200 ;

— Carte graphique : NVIDIA GeForce RTX 3060 (12 Go VRAM) ;

— Stockage : SSD NVMe 1 To ;

Système d'exploitation : Ubuntu 20.04 LTS avec pilotes NVIDIA 470.x pour l'accélération GPU.

Environnement de développement : Conda 4.14 avec des environnements isolés pour chaque expérimentation.

4.8.3 Considérations computationnelles

Utilisation GPU : Toutes les expérimentations exploitent l'accélération GPU CUDA pour réduire les temps d'entraînement.

Batch size : Taille de batch fixée à 64 échantillons, équilibrant l'utilisation de la mémoire et la stabilité d'entraînement.

Temps d'exécution estimé : Environ 2-3 heures par configuration et par projet cible, soit approximativement 240 heures d'expérimentation totale.

4.9 Considérations éthiques et reproductibilité

Pour garantir la reproductibilité complète de nos expérimentations :

— Les versions exactes de toutes les bibliothèques sont spécifiées ;
— Les jeux de données proviennent de répertoires publics standardisés ;

4.10 Conclusion

Ce chapitre a détaillé la méthodologie complète de notre recherche, depuis la sélection et préparation des données jusqu'aux protocoles d'évaluation. Notre approche se distingue

par :

1. Une exploration systématique de quatre configurations Transformer distinctes ;
2. Un protocole de validation inter-projets rigoureux ;
3. Un traitement spécialisé des modalités AST et métriques orientées objet ;
4. Une attention particulière à la reproductibilité et à la validité scientifique ;

Cette méthodologie robuste nous permet d'évaluer objectivement l'apport des différentes architectures proposées et de formuler des conclusions fiables sur l'efficacité des approches hybrides pour la prédiction de fautes logicielles. Le chapitre suivant présente les résultats obtenus selon ce protocole expérimental.

Chapitre 5

Résultats expérimentaux et analyse

5.1 Introduction

Ce chapitre présente les résultats expérimentaux obtenus selon la méthodologie décrite dans le chapitre précédent. Nous évaluons systématiquement les quatre configurations d'architectures Transformer proposées sur cinq projets Java open source, selon un protocole de validation inter-projets rigoureux.

Les expérimentations visent à répondre aux questions de recherche formulées en introduction :

1. Les AST peuvent-ils, à eux seuls, fournir une représentation suffisamment expressive pour prédire les défauts ?
2. Une approche hybride combinant AST et métriques orientées objet améliore-t-elle significativement les performances ?
3. Quelle architecture hybride (Encoder-Encoder vs Encoder-Decoder) est la plus efficace ?
4. Les modèles proposés sont-ils robustes face à la variabilité inter-projets ?

Chaque configuration est analysée individuellement avant une synthèse comparative qui met en évidence les forces et les faiblesses de chaque approche.

5.2 Analyse descriptive des données

Étant donné le volume important des statistiques descriptives détaillées, seules les valeurs résumées sont présentées dans ce chapitre. Les statistiques complètes — incluant les moyennes, minimums, maximums et variances pour l'ensemble des métriques orientées objet et des séquences AST — sont disponibles en libre accès dans le dépôt GitHub du projet afin d'éviter une surcharge de présentation dans le manuscrit. [1]

5.2.1 Statistiques descriptives des métriques orientées objet

TABLE 5.1 – Résumé des moyennes des métriques orientées objet par projet

Projet	WMC	CBO	RFC	LCOM	LOC	Défauts (%)
données d'entraînement	10,67	10,30	28,66	113,00	308,90	21,30
Ant-1.7	11,25	11,21	34,93	90,70	281,64	33,10
Camel-1.6	17,51	12,09	46,80	623,06	214,25	50,40
JEdit-4.3	12,87	8,23	34,53	320,36	172,50	2,00
Log4j-1.2	20,06	14,85	56,39	710,20	242,10	96,40
Xerces-1.4	18,66	13,13	49,91	690,49	225,56	11,90

Cette section présente les caractéristiques structurelles des projets utilisés dans les expérimentations. Les statistiques descriptives permettent d'évaluer quantitativement les différences entre le jeu d'entraînement concaténé et les jeux de test. Elles incluent les principales métriques orientées objet : la complexité moyenne (WMC), le couplage (CBO), la réponse pour une classe (RFC), la cohésion (LCOM), la taille moyenne des classes (LOC), ainsi que le taux de défauts en pourcentage.

1. https://github.com/hizhac/ast-oo.git

Le jeu d'entraînement global présente une complexité moyenne de 10,67, un couplage moyen de 10,30 et une réponse pour une classe moyenne de 28,66. La valeur moyenne de LCOM est de 113,0, et la taille moyenne des classes est de 308,9 lignes. Le taux de défauts est de 21,3 %, ce qui correspond à un déséquilibre modéré entre classes saines et défectueuses.

Les jeux de test présentent des écarts mesurables par rapport à ces valeurs. Pour Ant-1.7, WMC est de 11,25, CBO de 11,21 et RFC de 34,93. La taille moyenne des classes est de 281,64 lignes, et LCOM de 90,7. Le taux de défauts s'élève à 33,1 %.

Camel-1.6 montre des écarts plus importants : WMC = 17,51, CBO = 12,09, RFC = 46,80 et LCOM = 623,06. La taille moyenne des classes est de 214,25 lignes, et le taux de défauts atteint 50,4 %.

JEdit-4.3 présente une complexité moyenne de 12,87, un couplage de 8,23 et une taille moyenne des classes de 172,50 lignes. LCOM y atteint 320,36, soit une augmentation par rapport aux données d'entraînement. Le taux de défauts est très faible, à 2,0 %.

Log4j-1.2 présente les écarts les plus marqués : WMC = 20,06, CBO = 14,85, RFC = 56,39 et LCOM = 710,20. La taille moyenne des classes est de 242,10 lignes. Le taux de défauts atteint 96,4 %, ce qui constitue une distribution très différente de celle du jeu d'entraînement.

Enfin, Xerces-1.4 présente une complexité moyenne de 18,66, un couplage de 13,13, une réponse pour une classe de 49,91 et un LCOM de 690,49. La taille des classes est de 225,56 lignes, et le taux de défauts est de 11,9 %.

Ces statistiques montrent que les projets de test présentent des différences structurelles importantes, tant en termes de complexité et de cohésion que de distribution des défauts. Ces variations peuvent jouer un rôle dans la performance des modèles évalués dans les

sections suivantes.

5.2.2 Statistiques descriptives des séquences AST

TABLE 5.2 – Résumé des caractéristiques des séquences AST par projet

Projet	**Moyenne des tokens**	**Variance des tokens**	**Défauts (%)**
Données d'entraînement	5 000	$1{,}30 \times 10^7$ (13 021 734,7)	21,30
Ant-1.7	5 200–6 000	$1{,}02 \times 10^7$ (10 247 841,2)	33,10
Camel-1.6	6 000–7 000	$1{,}45 \times 10^7$ (14 518 440,5)	50,40
JEdit-4.3	3 500–4 000	$7{,}42 \times 10^6$ (7 420 315,8)	2,00
Log4j-1.2	6 500–7 000	$1{,}62 \times 10^7$ (16 218 304,7)	96,40
Xerces-1.4	4 500–5 000	$1{,}11 \times 10^7$ (11 146 932,6)	11,90

Les fichiers AST contiennent 512 tokens par instance, correspondant aux séquences dérivées des arbres syntaxiques abstraits. Les statistiques descriptives portent principalement sur la moyenne des tokens, leur variance, ainsi que le taux de défauts en pourcentage. Ces valeurs permettent de caractériser la diversité syntaxique des projets et d'apprécier les écarts entre le jeu d'entraînement et les jeux de test.

Dans le jeu d'entraînement concaténé, la moyenne des tokens se situe autour de 5 000, et la variance moyenne est de l'ordre de $1{,}30 \times 10^7$ (13 021 734,7). Le taux de défauts est de 21,3 %. Ces valeurs indiquent une diversité syntaxique importante répartie sur plusieurs projets.

Ant-1.7 présente des valeurs proches, avec une moyenne des tokens située entre 5 200 et 6 000, et une variance d'environ $1{,}00 \times 10^7$ (10 247 841,2). Le taux de défauts est de 33,1 %.

Camel-1.6 montre des écarts plus marqués : la moyenne des tokens se situe entre 6 000 et 7 000, et la variance est parmi les plus élevées, autour de $1{,}45 \times 10^7$ (14 518 440,5). Le taux de défauts est de 50,4 %.

JEdit-4.3 présente une moyenne nettement plus faible, autour de 3 500 à 4 000, et une variance plus modérée, d'environ 7,42 $\times$ 10^6 (7 420 315,8). Le taux de défauts est très faible, à 2,0 %.

Log4j-1.2 présente une moyenne de tokens souvent supérieure à 6 500, et une variance très élevée, autour de 1,62 $\times$ 10^7 (16 218 304,7). Le taux de défauts atteint 96,4 %.

Xerces-1.4 présente une moyenne située entre 4 500 et 5 000, avec une variance de l'ordre de 1,11 $\times$ 10^7 (11 146 932,6). Le taux de défauts est de 11,9 %.

Ces valeurs révèlent que les distributions syntaxiques des projets de test diffèrent nettement de celles du jeu d'entraînement, tant par la moyenne que par la dispersion des tokens. Ces écarts pourraient contribuer à la variabilité des performances du modèle AST-only.

5.2.3 Synthèse de la variabilité inter-projets

La comparaison des statistiques descriptives des projets met en évidence des écarts quantifiables entre le jeu d'entraînement et les jeux de test, tant au niveau des métriques orientées objet que des séquences AST. Pour les métriques, la complexité moyenne (WMC) du jeu d'entraînement est de 10,67, tandis qu'elle atteint 11,25 dans Ant-1.7, 17,51 dans Camel-1.6, 12,87 dans JEdit-4.3, 20,06 dans Log4j-1.2 et 18,66 dans Xerces-1.4. Ces valeurs montrent des écarts allant de $+$0,58 pour Ant-1.7 à $+$9,39 pour Log4j-1.2 par rapport au jeu d'entraînement.

Le couplage moyen (CBO), de 10,30 dans les données d'entraînement, atteint 11,21 dans Ant-1.7, 12,09 dans Camel-1.6, 8,23 dans JEdit-4.3, 14,85 dans Log4j-1.2 et 13,13 dans Xerces-1.4. Les écarts varient ainsi de –2,07 (JEdit-4.3) à $+$4,55 (Log4j-1.2). La réponse pour une classe (RFC), de 28,66 dans les données d'entraînement, est de 34,93 dans Ant-1.7, 46,80 dans Camel-1.6, 34,53 dans JEdit-4.3, 56,39 dans Log4j-1.2 et 49,91

dans Xerces-1.4, soit des écarts allant de +5,87 à +27,73.

La cohésion mesurée par LCOM varie également : 113,0 dans les données d'entraînement, 90,7 dans Ant-1.7, 623,06 dans Camel-1.6, 320,36 dans JEdit-4.3, 710,20 dans Log4j-1.2 et 690,49 dans Xerces-1.4. Ces valeurs montrent des écarts allant de –22,3 (Ant-1.7) à +597,2 (Log4j-1.2). La taille moyenne des classes, de 308,9 lignes dans les données d'entraînement, est de 281,64 dans Ant-1.7, 214,25 dans Camel-1.6, 172,50 dans JEdit-4.3, 242,10 dans Log4j-1.2 et 225,56 dans Xerces-1.4. Les écarts vont ainsi de –27,3 (Ant-1.7) à –136,4 (JEdit-4.3).

Les taux de défauts présentent une amplitude particulièrement large : 21,3 % dans les données d'entraînement contre 33,1 % dans Ant-1.7, 50,4 % dans Camel-1.6, 2,0 % dans JEdit-4.3, 96,4 % dans Log4j-1.2 et 11,9 % dans Xerces-1.4.

Les statistiques AST confirment également la variabilité inter-projets. La moyenne des tokens dans le jeu d'entraînement est d'environ 5 000, contre 5 200 à 6 000 dans Ant-1.7, 6 000 à 7 000 dans Camel-1.6, 3 500 à 4 000 dans JEdit-4.3, 6 500 à 7 000 dans Log4j-1.2 et 4 500 à 5 000 dans Xerces-1.4. Les variances montrent une dispersion similaire : $1{,}30 \times 10^7$ (13 021 734,7) dans les données d'entraînement, environ $1{,}00 \times 10^7$ (10 247 841,2) dans Ant-1.7, $1{,}45 \times 10^7$ (14 518 440,5) dans Camel-1.6, $7{,}42 \times 10^6$ (7 420 315,8) dans JEdit-4.3, $1{,}62 \times 10^7$ (16 218 304,7) dans Log4j-1.2 et $1{,}11 \times 10^7$ (11 146 932,6) dans Xerces-1.4.

Ces écarts chiffrés montrent que les projets de test ne présentent pas les mêmes distributions que le jeu d'entraînement, tant au niveau des métriques structurelles que des séquences syntaxiques. Les différences observées concernent la complexité, le couplage, la cohésion, la taille du code, la moyenne et la variance des tokens AST, ainsi que le taux de défauts. Ces variations constituent un décalage distributionnel qui pourrait avoir une influence sur la capacité des différents modèles à généraliser dans un contexte inter-projets.

5.3 Résultats et analyse – Modèle Encoder-Only AST

5.3.1 Résultats globaux

TABLE 5.3 – Performances de la configuration Encoder-Only AST

Projet	**G-mean (%)**	**F1-score (%)**	**Recall (%)**	**Accuracy (%)**
Ant-1.7	53,0	34,0	31,0	81,0
Camel-1.6	46,0	26,0	22,0	88,0
JEdit-4.3	37,0	3,0	18,0	77,0
Log4j-1.2	33,0	20,0	11,0	40,0
Xerces-1.4	14,0	3,0	2,3	53,0
Moyenne	**36,6**	**17,2**	**16,9**	**67,8**

La configuration Encoder-Only basée uniquement sur les séquences AST obtient des performances modestes en validation inter-projets. Le modèle atteint en moyenne un G-mean de 36,6 %, un F1-score moyen de 17,2 %, un rappel moyen de 16,9 % et une accuracy de 67,8 %. La disparité entre la précision globale et les autres métriques reflète un phénomène classique en contexte déséquilibré : lorsque le nombre de modules défectueux est faible dans certains projets, la prédiction majoritaire tend à privilégier la classe négative, ce qui augmente artificiellement l'accuracy.

Ces résultats suggèrent que les représentations syntaxiques isolées issues des AST restent sensibles aux variations structurelles et syntaxiques entre projets. Les sections suivantes examinent ces performances projet par projet, en s'appuyant sur les statistiques descriptives présentées précédemment.

5.3.2 Analyse des performances par projet

Sur Ant-1.7, le modèle obtient un G-mean de 53,0 %, un F1-score de 34,0 %, un rappel de 31,0 % et une accuracy de 81,0 %. Les statistiques descriptives indiquent que la moyenne des tokens AST dans ce projet se situe entre 5 200 et 6 000, contre environ 5 000 dans le jeu d'entraînement, tandis que la variance atteint environ $1{,}00 \times 10^7$ (10 247 841,2), proche de celle des données d'entraînement ($1{,}30 \times 10^7$, 13 021 734,7). Le taux de défauts (33,1 %) est également relativement proche de celui des données d'entraînement (21,3 %). Ces écarts modérés semblent permettre au modèle d'obtenir des performances supérieures à celles observées pour d'autres systèmes.

Pour Camel-1.6, le modèle obtient un G-mean de 46,0 %, un F1-score de 26,0 %, un rappel de 22,0 % et une Accuracy de 88,0 %. Ce projet se distingue par une moyenne des tokens AST comprise entre 6 000 et 7 000, nettement supérieure à celle des données d'entraînement, et une variance parmi les plus fortes, d'environ $1{,}45 \times 10^7$ (14 518 440,5). Son taux de défauts est également plus élevé, à 50,4 %. Ces différences chiffrées montrent que Camel-1.6 présente une distribution syntaxique plus éloignée de celle des données d'entraînement, ce qui correspond à une baisse de la performance du modèle.

JEdit-4.3 présente un G-mean de 37,0 %, un F1-score de 3,0 %, un rappel de 18,0 % et une Accuracy de 77,0 %. La moyenne des tokens y est comprise entre 3 500 et 4 000, soit une valeur inférieure à celle des données d'entraînement, et la variance est de $7{,}42 \times 10^6$ (7 420 315,8). Le taux de défauts est extrêmement faible, à 2,0 %, ce qui pourrait expliquer en grande partie les faibles valeurs de F1. En effet, la rareté des instances positives rend la tâche de classification plus difficile, même lorsque la variabilité syntaxique est moins élevée que dans d'autres projets.

Pour Log4j-1.2, le modèle obtient un G-mean de 33,0 %, un F1-score de 20,0 %, un rappel de 11,0 % et une Accuracy de 40,0 %. La moyenne des tokens dépasse fréquemment

6 500, et la variance atteint 1,62 $\times$ 10^7 (16 218 304,7), soit la valeur la plus élevée parmi les projets testés. Le taux de défauts est également très élevé, à 96,4 %, ce qui constitue un profil distributionnel très éloigné de celui des données d'entraînement. Combinés, ces écarts syntaxiques et distributionnels pourraient expliquer les performances limitées du modèle sur ce projet.

Sur Xerces-1.4, le modèle obtient un G-mean de 14,0 %, un F1-score de 3,0 %, un rappel de 2,3 % et une Accuracy de 53,0 %. La moyenne des tokens se situe entre 4 500 et 5 000, tandis que la variance atteint 1,11 $\times$ 10^7 (11 146 932,6). Le taux de défauts est de 11,9 %, inférieur à celui des données d'entraînement. Ce profil combine des variations syntaxiques importantes avec une distribution des classes qui reste déséquilibrée, ce qui semble correspondre aux faibles performances observées.

5.3.3 Discussion spécifique à l'approche AST-only

L'ensemble des résultats suggère que la configuration Encoder-Only fondée sur les AST est fortement influencée par la variabilité inter-projets. Les statistiques descriptives font apparaître des différences particulières entre les jeux de test et le jeu d'entraînement, tant au niveau des moyennes des tokens que de leurs variances. Ces écarts vont d'environ +200 à +2 000 en moyenne pour Ant-1.7, jusqu'à environ +1 500 à +2 500 pour Camel-1.6 et Log4j-1.2, tandis que JEdit-4.3 présente des valeurs inférieures d'environ –1 000 à –1 500. Les variances suivent des tendances similaires, variant de 7,42 $\times$ 10^6 (7 420 315,8) à 1,62 $\times$ 10^7 (16 218 304,7) selon les projets.

Par ailleurs, les taux de défauts, allant de 2,0 % à 96,4 %, créent des conditions d'apprentissage très différentes selon les projets. Lorsque le taux est faible, comme dans JEdit-4.3, le modèle manque d'exemples positifs. Lorsqu'il est très élevé, comme dans Log4j-1.2, il devient difficile de distinguer les classes. Ces éléments semblent pouvoir influer sur les performances.

Les travaux de White et al. (2016) ainsi que ceux d'Allamanis et al. (2018) indiquent que les modèles fondés uniquement sur les représentations syntaxiques sont sensibles aux variations structurelles et stylistiques du code. Les résultats obtenus ici, associés aux écarts observés dans les statistiques descriptives, semblent s'inscrire dans cette tendance, dans un contexte inter-projets.

5.4 Résultats et analyse – Modèle Encoder-Only Métriques

5.4.1 Résultats globaux

Table 5.4 – Performances de la configuration Encoder-Only Métriques

Projet	**G-mean (%)**	**F1-score (%)**	**Recall (%)**	**Accuracy (%)**
Ant-1.7	70,0	70,0	70,0	70,0
Camel-1.6	58,0	40,0	39,0	77,0
JEdit-4.3	67,0	6,0	45,0	72,0
Log4j-1.2	59,0	57,0	40,0	44,0
Xerces-1.4	50,0	43,0	29,0	44,0
Moyenne	**60,8**	**43,2**	**44,6**	**61,4**

Le modèle Encoder-Only entraîné uniquement sur les métriques orientées objet obtient des performances plus élevées que la configuration basée sur les AST. Le G-mean moyen est de 60,8 %, le F1-score moyen de 43,2 %, le rappel moyen de 44,6 % et l'Accuracy moyenne de 61,4 %. Ces valeurs pourraient indiquer une meilleure capacité de généralisation inter-projets, ce qui suggère que les descripteurs structurels issus des métriques orientées objet semblent moins sensibles aux variations inter-projets que les séquences AST, dans le cadre expérimental étudié.

5.4.2 Analyse des performances par projet

Sur Ant-1.7, le modèle atteint un F1-score de 70 %, accompagné d'un G-mean de 70 %, d'un rappel de 70 % et d'une Accuracy de 70 %. Les statistiques montrent que les valeurs structurelles de ce projet restent proches du jeu d'entraînement : WMC passe de 10,67 à 11,25, CBO de 10,30 à 11,21, et RFC de 28,66 à 34,93. La taille des classes diminue légèrement (de 308,9 à 281,6 lignes), et la cohésion varie de 113,0 à 90,7. Le taux de défauts est de 33,1 %, contre 21,3 % dans les données d'entraînement. Ces écarts, relativement faibles, pourraient expliquer la stabilité des performances observées.

Pour Camel-1.6, le modèle obtient un F1-score de 40 %, un G-mean de 58 %, un rappel de 39 % et une Accuracy de 77 %. Ce projet présente des écarts structurels marqués : WMC = 17,51, RFC = 46,80 et LCOM = 623,06, contre respectivement 10,67, 28,66 et 113,0 dans les données d'entraînement. La taille moyenne des classes diminue de 308,9 à 214,2 lignes, et le taux de défauts atteint 50,4 %. Malgré ces écarts, le modèle conserve des performances modérées, ce qui pourrait indiquer une certaine robustesse des métriques orientées objet face à des variations structurelles importantes.

JEdit-4.3 présente un F1-score de 6 %, un G-mean de 67 %, un rappel de 45 % et une Accuracy de 72 %. La complexité moyenne y est de 12,87, le couplage de 8,23, la taille moyenne des classes de 172,5 lignes et LCOM de 320,36. Le taux de défauts est très faible, à 2,0 %, ce qui entraîne un déséquilibre important entre classes saines et défectueuses. Cette asymétrie pourrait expliquer le faible F1-score malgré un rappel relativement élevé. Les métriques orientées objet permettent néanmoins au modèle de détecter une partie des rares instances défectueuses.

Sur Log4j-1.2, le modèle atteint un F1-score de 57 %, un G-mean de 59 %, un rappel de 40 % et une Accuracy de 44 %. Ce projet présente les écarts structurels les plus importants : WMC = 20,06, RFC = 56,39, CBO = 14,85 et LCOM = 710,20, contre 10,67, 28,66,

10,30 et 113,0 dans les données d'entraînement. La taille moyenne des classes diminue à 242,1 lignes, mais ces valeurs indiquent un niveau de complexité et de couplage nettement supérieur. Le taux de défauts atteint 96,4 %. Malgré ce contexte particulier, le modèle métriques-only parvient à conserver un niveau de performance élevé avec la mesure F1.

Pour Xerces-1.4, le modèle obtient un F1-score de 43 %, un G-mean de 50 %, un rappel de 29 % et une Accuracy de 44 %. La complexité moyenne y est de 18,66, la réponse pour une classe de 49,91, le couplage de 13,13 et LCOM de 690,49, soit des valeurs supérieures à celles des données d'entraînement. La taille des classes est de 225,6 lignes, et le taux de défauts est de 11,9 %. Ces différences pourraient en partie contribuer aux performances intermédiaires observées sur ce projet.

5.4.3 Discussion spécifique à l'approche métriques-only

Les performances observées suggèrent que les métriques orientées objet constituent une représentation plus stable que les AST dans un contexte inter-projets. Les écarts entre les valeurs structurelles du jeu d'entraînement et des projets de test, qu'il s'agisse de la complexité (WMC), du couplage (CBO), de la cohésion (LCOM) ou de la taille moyenne des classes (LOC), semblent influer la performance du modèle mais de manière moins marquée que pour les séquences AST. Les variations observées vont de +0,58 pour Ant-1.7 à +9,39 pour Log4j-1.2 en termes de complexité, et de –27,3 à –136,4 lignes pour la taille des classes. Les taux de défauts varient de 2,0 % à 96,4 %, ce qui constitue une amplitude importante, mais à laquelle le modèle reste relativement robuste.

Ces résultats suggèrent que les métriques structurelles capturent des propriétés générales du logiciel qui varient moins fortement d'un projet à l'autre que les représentations syntaxiques. Les performances obtenues dans cette configuration semblent ainsi indiquer une stabilité relativement supérieure en validation inter-projets.

5.5 Résultats et analyse – Modèle Hybride Encoder–Encoder

5.5.1 Résultats globaux

TABLE 5.5 – Performances de la configuration Hybride Encoder–Encoder

Projet	**G-mean (%)**	**F1-score (%)**	**Recall (%)**	**Accuracy (%)**
Ant-1.7	63,7	44,4	59,0	66,5
Camel-1.6	63,2	48,9	50,7	71,0
JEdit-4.3	60,8	6,5	63,6	58,3
Log4j-1.2	36,6	51,6	35,8	35,8
Xerces-1.4	40,0	32,6	22,8	39,9
Moyenne	**52,9**	**36,8**	**46,4**	**54,3**

La configuration Hybride Encoder–Encoder combine simultanément les représentations issues des métriques orientées objet et des séquences AST. Elle obtient un G-mean moyen de 52,9 %, un F1-score moyen de 36,8 %, un rappel moyen de 46,4 % et une Accuracy moyenne de 54,3 %. Ces valeurs situent cette approche entre les performances du modèle métriques-only et celles du modèle AST-only, ce qui reflète une exploitation partielle mais non optimale de l'information fournie par les deux modalités.

5.5.2 Analyse des performances par projet

Sur Ant-1.7, le modèle obtient un F1-score de 44,4 %, un rappel de 59,0 %, un G-mean de 63,7 % et une Accuracy de 66,5 %. Les statistiques descriptives montrent que les valeurs structurelles (WMC = 11,25 ; CBO = 11,21 ; RFC = 34,93) et syntaxiques (moyenne AST autour de 5 200–6 000) restent proches des données d'entraînement. Le taux de défauts est de 33,1 %, contre 21,3 % dans les données d'entraînement. Ces écarts modérés semblent expliquer que la combinaison des deux modalités procure un gain en rappel, tout en maintenant une performance globale stable.

Pour Camel-1.6, le modèle atteint un F1-score de 48,9 %, un rappel de 50,7 %, un G-mean de 63,2 % et une Accuracy de 71,0 %. Les écarts structurels sont importants : WMC = 17,51, RFC = 46,80, LCOM = 623,06 et moyenne AST entre 6 000 et 7 000, avec une variance élevée (1,45 $\times$ 10^7, 14 518 440,5). Le taux de défauts est de 50,4 %. Ce profil, plus complexe et plus hétérogène que les données d'entraînement, expliquerait pourquoi le modèle hybride dépasse la configuration Métriques-only et la configuration AST-only sur ce projet.

Sur JEdit-4.3, le modèle obtient un F1-score de 6,5 %, un rappel de 63,6 %, un G-mean de 60,8 % et une Accuracy de 58,3 %. La moyenne AST est comprise entre 3 500 et 4 000, et la variance est de 7,42 $\times$ 10^6 (7 420 315,8). Le taux de défauts est toutefois très faible, à 2,0 %, ce qui pourrait expliquer la forte disparité entre le rappel et le F1-score. Le modèle identifie une partie des rares instances défectueuses, mais la prédiction excessive de faux positifs entraîne un effondrement du F1-score.

Pour Log4j-1.2, la configuration hybride obtient un F1-score de 51,6 %, un rappel de 35,8 %, un G-mean de 36,6 % et une Accuracy de 35,8 %. Ce projet présente les écarts structurels les plus marqués : WMC = 20,06, RFC = 56,39, CBO = 14,85, LCOM = 710,20, moyenne AST supérieure à 6 500 et variance de 1,62 $\times$ 10^7 (16 218 304,7). Son taux de défauts atteint 96,4 %. Ces différences, tant syntaxiques que structurelles, expliqueraient la diminution du G-mean, tandis que le F1-score reste élevé en raison de la prédominance de la classe défectueuse.

Sur Xerces-1.4, le modèle obtient un F1-score de 32,6 %, un rappel de 22,8 %, un G-mean de 40,0 % et une Accuracy de 39,9 %. Les statistiques indiquent une complexité élevée (WMC = 18,66, RFC = 49,91), une cohésion faible (LCOM = 690,49), une taille de classe modérée (225,6 lignes) et une moyenne AST entre 4 500 et 5 000, avec une variance de 1,11 $\times$ 10^7 (11 146 932,6). Le taux de défauts est de 11,9 %. Ces valeurs témoignent d'un profil structurel et syntaxique différent des données d'entraînement, ce qui correspond aux performances intermédiaires observées.

5.5.3 Discussion spécifique à l'approche Encoder–Encoder

Les résultats obtenus avec la configuration Encoder–Encoder mettent en évidence une sensibilité notable du modèle aux écarts structurels et syntaxiques observés entre les jeux de test et le jeu d'entraînement. Les statistiques descriptives montrent que les projets dont les caractéristiques s'écartent peu des valeurs des données d'entraînement tendent à présenter des performances plus élevées. Par exemple, dans Ant-1.7, la complexité moyenne (WMC) passe de 10,67 dans les données d'entraînement à 11,25, la réponse pour une classe (RFC) de 28,66 à 34,93 et la moyenne des tokens AST se situe entre 5 200 et 6 000, soit des valeurs proches de celles du jeu d'entraînement. Le taux de défauts y est de 33,1 %, contre 21,3 % dans les données d'entraînement. Dans ce contexte, le modèle obtient un F1-score de 44,4 % et un rappel de 59,0 %, ce qui suggère que des écarts limités facilitent la généralisation.

Dans Camel-1.6, les métriques structurelles et les caractéristiques AST présentent des écarts plus importants. La complexité (WMC = 17,51) et la réponse pour une classe (RFC = 46,80) y sont nettement supérieures à celles des données d'entraînement, et la variance des tokens AST atteint $1{,}45 \times 10^7$ (14 518 440,5), contre $1{,}30 \times 10^7$ (13 021 734,7) dans le jeu d'entraînement. Le taux de défauts est également plus élevé (50,4 %). Dans ce contexte, le modèle obtient un F1-score de 48,9 % et un rappel de 50,7 %, soit des performances supérieures à celles rencontrées dans certains autres projets présentant des écarts structurels ou syntaxiques comparables. Cela semble indiquer que la qualité des résultats ne dépend pas uniquement d'un écart particulier, mais plutôt de la combinaison de plusieurs facteurs.

JEdit-4.3 montre un cas particulier, avec un taux de défauts très faible (2,0 %) et une moyenne AST nettement inférieure à celle des données d'entraînement, entre 3 500 et 4 000. Les valeurs de complexité et de cohésion diffèrent également : WMC = 12,87, LCOM = 320,36 contre 113,0 dans les données d'entraînement. Dans ces conditions, le

modèle obtient un F1-score de 6,5 % malgré un rappel de 63,6 %. La disparité entre ces deux mesures reflète principalement l'impact du faible nombre d'exemples défectueux, qui conduit à un nombre élevé de faux positifs et, en conséquence, à un F1-score plus faible.

Log4j-1.2 présente les écarts les plus marqués parmi les jeux de test, tant au niveau structurel que syntaxique. Les valeurs de WMC (20,06), RFC (56,39) et LCOM (710,20) sont nettement supérieures à celles des données d'entraînement, et la variance AST atteint $1{,}62 \times 10^7$ (16 218 304,7). Le taux de défauts y est de 96,4 %, soit une distribution très éloignée de celle rencontrée lors de l'entraînement. Dans ce contexte, le modèle obtient un F1-score de 51,6 % et une Accuracy de 35,8 %, valeurs qui reflètent les caractéristiques particulières de ce jeu de test. Le haut taux de défauts facilite en partie l'identification de la classe positive, tout en rendant l'Accuracy moins représentative.

Xerces-1.4 présente des valeurs intermédiaires en termes d'écarts : WMC = 18,66, RFC = 49,91, LCOM = 690,49, avec une moyenne AST entre 4 500 et 5 000 et une variance de $1{,}11 \times 10^7$ (11 146 932,6). Le taux de défauts y est de 11,9 %. Les performances obtenues — F1-score de 32,6 % et rappel de 22,8 % — correspondent à un profil qui combine des différences structurelles importantes avec une distribution moins extrême que dans Log4j-1.2 ou JEdit-4.3.

Dans l'ensemble, les résultats mettent en évidence que la configuration Encoder–Encoder est influencée par les écarts simultanés au niveau des métriques structurelles, des caractéristiques AST et des taux de défauts. Aucune relation directe ne peut être établie entre un indicateur particulier et la performance du modèle, mais les observations montrent que les projets dont les valeurs s'éloignent fortement de celles du jeu d'entraînement — qu'il s'agisse de mesures structurelles, syntaxiques ou distributionnelles — tendent à produire des performances plus variables. L'approche Encoder–Encoder permet toutefois, dans plusieurs cas, d'obtenir des résultats plus élevés que les modèles unimodaux, ce qui suggère que la combinaison parallèle des deux représentations peut contribuer à améliorer la performance lorsque les divergences entre les jeux d'entraînement et de test ne sont pas

trop importantes.

5.6 Résultats et analyse – Modèle Hybride Encoder–Decoder

5.6.1 Résultats globaux

TABLE 5.6 – Performances de la configuration Hybride Encoder–Decoder

Projet	**G-mean (%)**	**F1-score (%)**	**Recall (%)**	**Accuracy (%)**
Ant-1.7	71,7	54,0	68,7	73,5
Camel-1.6	51,8	35,2	37,3	62,5
JEdit-4.3	61,2	7,3	54,5	68,4
Log4j-1.2	42,9	39,1	24,6	26,7
Xerces-1.4	35,1	22,7	13,4	41,8
Moyenne	**52,5**	**31,7**	**39,7**	**54,6**

La configuration Hybride Encoder–Decoder intègre les séquences AST dans l'encodeur et les métriques orientées objet dans le décodeur. Cette fusion asymétrique repose sur un traitement syntaxique initial, auquel s'ajoutent les descripteurs structurels lors de la phase de décodage. Au niveau global, le modèle obtient un G-mean moyen de 52,5 %, un F1-score moyen de 31,7 %, un rappel moyen de 39,7 % et une Accuracy moyenne de 54,6 %. Ces performances sont inférieures à celles obtenues avec la configuration Encoder–Encoder, qui atteint un F1-score moyen de 36,8 %, mais supérieures à celles de la configuration AST-only (F1 moyen de 17,2 %). Cette position intermédiaire reflète la capacité du modèle à exploiter partiellement les deux modalités.

5.6.2 Analyse des performances par projet

Sur Ant-1.7, le modèle obtient un F1-score de 54,0 %, un rappel de 68,7 %, un G-mean de 71,7 % et une Accuracy de 73,5 %. Les caractéristiques d'Ant-1.7 se rapprochent des valeurs du jeu d'entraînement : WMC = 11,25 (contre 10,67), CBO = 11,21 (contre 10,30), RFC = 34,93 (contre 28,66) et moyenne AST entre 5 200 et 6 000 (contre environ 5 000 dans les données d'entraînement). La variance AST, $1{,}00 \times 10^7$ (10 247 841,2), reste également proche de celle des données d'entraînement ($1{,}30 \times 10^7$, 13 021 734,7). Le taux de défauts est de 33,1 %, contre 21,3 % dans les données d'entraînement. L'alignement partiel entre les distributions structurelles et syntaxiques semble faciliter la fusion séquentielle, ce qui correspond aux bonnes performances observées.

Camel-1.6 présente des écarts structurels et syntaxiques plus importants : WMC = 17,51, RFC = 46,80, LCOM = 623,06, et moyenne AST entre 6 000 et 7 000, avec une variance élevée ($1{,}45 \times 10^7$, 14 518 440,5). Le taux de défauts est de 50,4 %. Le modèle y obtient un F1-score de 35,2 %, un rappel de 37,3 %, un G-mean de 51,8 % et une Accuracy de 62,5 %. Ce résultat est inférieur à celui du modèle Encoder–Encoder (F1 = 48,9 %). Les différences importantes entre les caractéristiques des données d'entraînement et celles du projet, combinées à la fusion séquentielle, semblent affecter la stabilité des représentations apprises.

Pour JEdit-4.3, le modèle obtient un F1-score de 7,3 %, un rappel de 54,5 %, un G-mean de 61,2 % et une Accuracy de 68,4 %. Les statistiques AST indiquent une moyenne comprise entre 3 500 et 4 000, soit une valeur inférieure à celle des données d'entraînement, et une variance de $7{,}42 \times 10^6$ (7 420 315,8). Structurellement, la complexité (WMC = 12,87) et la cohésion (LCOM = 320,36) s'écartent également de celles des données d'entraînement. Le taux de défauts extrêmement faible, à 2,0 %, reste toutefois le facteur dominant : il entraîne un nombre important de faux positifs, ce qui réduit fortement le F1-score malgré un rappel élevé. Les performances sont similaires à celles obtenues avec

la configuration Encoder–Encoder (F1 = 6,5 %), ce qui montre que l'architecture choisie n'atténue pas l'effet de la distribution très déséquilibrée.

Sur Log4j-1.2, la configuration Encoder–Decoder obtient un F1-score de 39,1 %, un rappel de 24,6 %, un G-mean de 42,9 % et une Accuracy de 26,7 %. Ce projet présente les écarts structurels et syntaxiques les plus marqués : WMC = 20,06, RFC = 56,39, CBO = 14,85, LCOM = 710,20, moyenne AST supérieure à 6 500 et variance de $1{,}62 \times 10^7$ (16 218 304,7). Le taux de défauts est extrêmement élevé (96,4 %). Dans ce contexte, les performances sont clairement inférieures à celles de la configuration Encoder–Encoder (F1 = 51,6 %), ce qui suggère que la fusion séquentielle peine à intégrer deux modalités fortement divergentes de celles du jeu d'entraînement.

Pour Xerces-1.4, le modèle obtient un F1-score de 22,7 %, un rappel de 13,4 %, un G-mean de 35,1 % et une Accuracy de 41,8 %. Ce projet présente une complexité élevée (WMC = 18,66), une réponse pour une classe importante (RFC = 49,91), une cohésion faible (LCOM = 690,49) et une moyenne AST comprise entre 4 500 et 5 000, avec une variance de $1{,}11 \times 10^7$ (11 146 932,6). Le taux de défauts est de 11,9 %. Ces écarts pourraient expliquer les performances intermédiaires, situées entre celles observées pour JEdit-4.3 et Log4j-1.2.

5.6.3 Discussion spécifique à l'approche Encoder–Decoder

La configuration Encoder–Decoder montre une sensibilité marquée aux écarts entre les caractéristiques du jeu d'entraînement et celles des jeux de test. Les performances élevées obtenues sur Ant-1.7 (F1 = 54,0 %) correspondent à des écarts structurels et syntaxiques modérés, tandis que les baisses observées dans Camel-1.6 (F1 = 35,2 %) et Log4j-1.2 (F1 = 39,1 %) coïncident avec des différences nettement plus importantes, tant dans les métriques orientées objet que dans les moyennes et variances AST. Les statistiques montrent notamment une augmentation de la variance AST de $1{,}30 \times 10^7$ (13 021 734,7) dans les

données d'entraînement à 1,45 $\times$ 10^7 (14 518 440,5) dans Camel-1.6 et à 1,62 $\times$ 10^7 (16 218 304,7) dans Log4j-1.2, ainsi qu'une augmentation de la complexité (WMC) respectivement de $+$6,84 et $+$9,39.

L'impact du taux de défauts est également notable. Les projets dont les taux s'écartent fortement des données d'entraînement — 2,0 % pour JEdit-4.3 et 96,4 % pour Log4j-1.2 — produisent des comportements asymétriques en termes de précision et de rappel. Dans ces cas, l'intégration séquentielle des deux modalités ne permet pas d'équilibrer efficacement la classification.

De manière générale, bien que la configuration Encoder–Decoder exploite les deux types de représentations, ses performances montrent qu'elle reste sensible aux variations inter-projets, en particulier lorsque les écarts concernent simultanément la structure (métriques orientées objet), la syntaxe (AST) et la distribution des défauts.

5.7 Comparaison globale des approches

5.7.1 Classement général des approches

Les résultats globaux obtenus pour les quatre configurations évaluées permettent d'établir un classement des performances en validation inter-projets. La configuration basée exclusivement sur les métriques orientées objet obtient les meilleurs résultats en moyenne, avec un F1-score moyen de 43,2 %, un rappel moyen de 44,6 % et un G-mean de 60,8 %. La configuration hybride Encoder–Encoder occupe la deuxième position, avec un F1-score moyen de 36,8 %, un rappel moyen de 46,4 % et un G-mean de 52,9 %. Vient ensuite la configuration Encoder–Decoder, qui atteint un F1-score moyen de 31,7 %, un rappel moyen de 39,7 % et un G-mean de 52,5 %. Enfin, la configuration AST-only obtient un

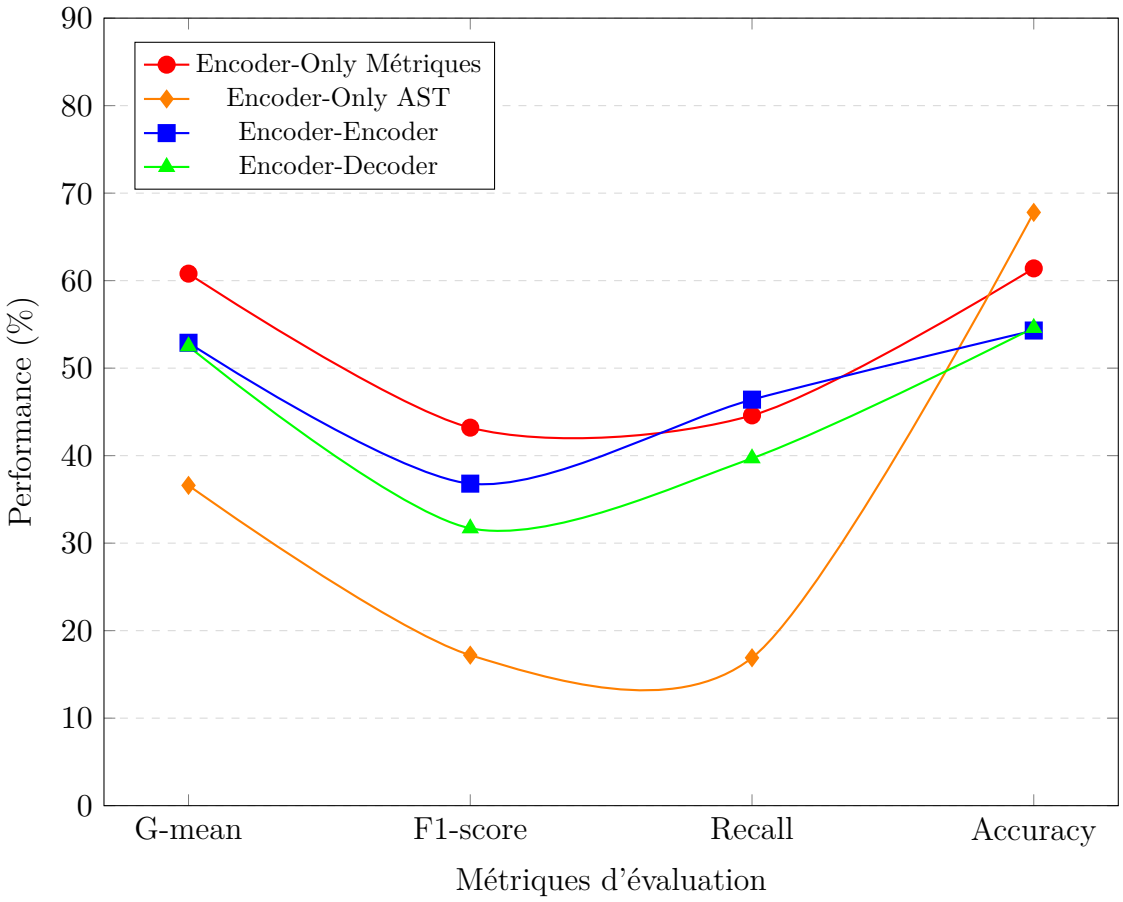

FIGURE 5.1 – Profils de performance des quatre configurations

F1-score moyen de 17,2 %, un rappel moyen de 16,9 % et un G-mean de 36,6 %.

Ce classement global montre que, sur les cinq projets testés, les représentations fondées sur les métriques orientées objet constituent les descripteurs les plus stables en validation inter-projets. Les deux configurations hybrides se situent entre les approches unimodales, ce qui indique une exploitation partielle de la complémentarité entre les descripteurs syntaxiques et structurels.

5.7.2 Analyse de la variabilité inter-projets

La variabilité observée entre les projets testés peut être reliée aux statistiques descriptives présentées en section 5.2. Les écarts structurels, qui concernent notamment la complexité (WMC), la réponse pour une classe (RFC), la cohésion (LCOM) et la taille moyenne des classes (LOC), varient de manière notable entre le jeu d'entraînement et les projets de test. Par exemple, WMC passe de 10,67 dans les données d'entraînement à 17,51 dans Camel-1.6, 20,06 dans Log4j-1.2 et 18,66 dans Xerces-1.4. La cohésion (LCOM) varie de 113,0 dans les données d'entraînement à 623,06 dans Camel-1.6, 320,36 dans JEdit-4.3 et 710,20 dans Log4j-1.2. De même, la taille moyenne des classes diminue de 308,9 lignes dans les données d'entraînement à 172,5 lignes dans JEdit-4.3 et à 214,3 lignes dans Camel-1.6. Les taux de défauts varient de 21,3 % dans les données d'entraînement à 2,0 % dans JEdit-4.3, 50,4 % dans Camel-1.6 et 96,4 % dans Log4j-1.2.

Les caractéristiques syntaxiques issues des AST présentent également des variations. La moyenne des tokens, d'environ 5 000 dans les données d'entraînement, atteint des valeurs comprises entre 6 000 et 7 000 pour Camel-1.6 et Log4j-1.2, tandis qu'elle descend entre 3 500 et 4 000 pour JEdit-4.3. Les variances AST vont de $7{,}42 \times 10^6$ (7 420 315,8) dans JEdit-4.3 à $1{,}62 \times 10^7$ (16 218 304,7) dans Log4j-1.2. Ces données montrent que chaque projet présente un profil spécifique, qui combine des écarts à la fois structurels et syntaxiques.

Les performances des modèles peuvent être interprétées à la lumière de ces écarts. Les projets dont les valeurs sont proches du jeu d'entraînement, comme Ant-1.7, tendent à produire des performances plus élevées dans l'ensemble des configurations. À l'inverse, les projets présentant des différences importantes au niveau structurel ou syntaxique, tels que Camel-1.6 et Log4j-1.2, montrent une baisse des performances, particulièrement pour les modèles basés sur les AST ou impliquant une fusion séquentielle. Enfin, les projets caractérisés par un taux de défauts très faible ou très élevé, comme JEdit-4.3 (2,0 %) et Log4j-1.2 (96,4 %), entraînent des comportements asymétriques en termes de précision et de rappel, indépendamment des configurations testées.

5.7.3 Forces et limites de chaque modalité

L'ensemble des résultats met en évidence des forces et des limites propres à chaque modalité. Les métriques orientées objet, fondées sur des caractéristiques structurelles globales du code, semblent offrir la meilleure stabilité en validation inter-projets. Elles conservent des performances correctes même lorsque les écarts structurels sont importants, comme dans Camel-1.6 (F1 = 40 %) ou Log4j-1.2 (F1 = 57 %). Les limites de cette approche apparaissent principalement dans les contextes où la distribution des classes est fortement déséquilibrée, comme dans JEdit-4.3 (2,0 % de classes défectueuses), où le F1-score chute à 6 % malgré un rappel de 45 %.

Les représentations AST capturent des propriétés syntaxiques fines du code, mais semblent montrer une sensibilité accrue aux variations inter-projets. Les performances obtenues dans des projets tels que Camel-1.6 (F1 = 26 %) ou Log4j-1.2 (F1 = 20 %) témoignent de cette difficulté. Les écarts observés dans les moyennes et variances AST indiquent que ces représentations varient plus fortement d'un projet à l'autre que les métriques orientées objet.

Les approches hybrides se situent entre les modèles unimodaux. La configuration

Encoder–Encoder, qui combine les deux modalités en parallèle, offre les meilleures performances hybrides, avec un F1-score moyen de 36,8 %. Elle exploite partiellement la complémentarité entre les caractéristiques syntaxiques et structurelles, ce qui permet d'améliorer les résultats dans certains projets, comme Camel-1.6 (F1 = 48,9 %). Toutefois, cette approche reste sensible aux écarts importants entre les jeux d'entraînement et de test.

La configuration Encoder–Decoder, qui intègre d'abord les AST puis les métriques orientées objet, présente une sensibilité encore plus marquée aux variations structurelles et syntaxiques. Les performances obtenues dans Camel-1.6 (F1 = 35,2 %) et Log4j-1.2 (F1 = 39,1 %) montrent que l'intégration séquentielle peut avoir plus de difficultés à équilibrer les deux modalités. Les écarts observés dans les moyennes et variances AST, ainsi que dans les valeurs structurelles, éclairent en partie ces comportements.

Réponses synthétiques aux questions de recherche

Les résultats obtenus dans ce chapitre permettent de répondre succinctement aux questions de recherche formulées en introduction.

Premièrement, l'utilisation exclusive des séquences AST n'a pas permis d'obtenir des performances suffisantes en contexte inter-projets. Le modèle AST-only atteint un F1-score moyen de 17,2 %, avec des valeurs très faibles sur certains projets, ce qui indique une représentation syntaxique trop sensible aux variations entre systèmes.

Deuxièmement, les approches hybrides n'ont pas conduit à une amélioration systématique des performances par rapport au modèle fondé uniquement sur les métriques orientées objet. Ce dernier obtient un F1-score moyen de 43,2 %, supérieur aux deux configurations hybrides (36,8 % et 31,7 %). Quelques gains ponctuels apparaissent, mais ils ne se généralisent pas à l'ensemble des projets testés.

Troisièmement, entre les deux architectures hybrides évaluées, la configuration Encoder–Encoder se montre globalement plus performante que l'architecture Encoder–Decoder. Avec un F1-score moyen de 36,8 %, elle offre de meilleurs résultats sur la majorité des projets, bien que l'inverse soit observé pour Ant-1.7.

Enfin, la robustesse des modèles face à la variabilité inter-projets demeure limitée. Les écarts structurels, syntaxiques et distributionnels observés entre le jeu d'entraînement et les jeux de test se traduisent par une variation notable des performances pour l'ensemble des configurations. Le modèle métriques-only reste le plus stable, mais aucune approche ne maintient des performances homogènes lorsque les caractéristiques du projet cible diffèrent fortement de celles du jeu d'entraînement.

5.8 Conclusion du chapitre

Ce chapitre a présenté les expérimentations menées dans un contexte de validation inter-projets, en évaluant quatre configurations basées sur les représentations syntaxiques (AST), structurelles (métriques orientées objet) et hybrides. Les statistiques descriptives ont mis en évidence des écarts importants entre les projets utilisés pour l'entraînement et ceux utilisés pour les tests, tant au niveau des mesures structurelles (par exemple, WMC variant de 10,67 dans les données d'entraînement à 20,06 dans Log4j-1.2) que des caractéristiques syntaxiques (variance AST allant de $7{,}42 \times 10^6$ (7 420 315,8) dans JEdit-4.3 à $1{,}62 \times 10^7$ (16 218 304,7) dans Log4j-1.2). Les taux de défauts ont également montré une large amplitude, de 2,0 % à 96,4 % selon les projets. Ces écarts ont constitué un élément déterminant dans l'interprétation des performances observées.

Les résultats suggèrent que l'approche Encoder-only fondée sur les métriques orientées objet est celle qui présente la meilleure stabilité inter-projets, avec un F1-score moyen de 43,2 % et un G-mean moyen de 60,8 %. Cette configuration conserve un niveau de per-

formance satisfaisant même dans des projets présentant des écarts structurels importants, comme Camel-1.6 ou Log4j-1.2. À l'inverse, le modèle AST-only obtient des performances plus variables, avec un F1-score moyen de 17,2 %, ce qui reflète la sensibilité des représentations syntaxiques aux différences entre projets, notamment lorsque les moyennes AST s'écartent fortement des valeurs données d'entraînement.

Les deux configurations hybrides se situent entre ces extrêmes. L'approche Encoder–Encoder, qui combine les représentations en parallèle, atteint un F1-score moyen de 36,8 % et montre des améliorations notables dans certains projets comme Camel-1.6, où elle dépasse les modèles unimodaux. L'approche Encoder–Decoder, fondée sur une fusion séquentielle des AST puis des métriques, obtient un F1-score moyen de 31,7 %, mais montre une plus grande sensibilité aux variations inter-projets, particulièrement dans les systèmes présentant des écarts importants ou des taux de défauts extrêmes.

L'ensemble des résultats met en évidence que la qualité de la généralisation dépend fortement de l'alignement entre les distributions d'entraînement et de test, qu'il s'agisse des valeurs structurelles, des caractéristiques syntaxiques ou du taux de défauts. Les modèles basés sur les métriques orientées objet bénéficient d'une variabilité inter-projets plus limitée et d'une représentation plus stable, tandis que les représentations AST sont davantage affectées par les différences syntaxiques. Les approches hybrides montrent un potentiel d'amélioration, mais leur efficacité dépendrait de la manière dont les deux modalités sont intégrées.

Ces observations fournissent des indications utiles pour les travaux futurs, en particulier pour l'exploration de mécanismes de fusion adaptés aux contextes où les divergences structurelles, syntaxiques ou distributionnelles entre projets sont marquées. Elles soulignent également l'intérêt d'approches capables de moduler dynamiquement la contribution des différentes modalités en fonction des caractéristiques du projet cible.

Chapitre 6

Discussion

6.1 Introduction

Ce chapitre discute les résultats obtenus dans le cadre des expérimentations inter-projets réalisées avec les quatre configurations retenues : *Encoder-only AST*, *Encoder-only Métriques*, *Hybride Encoder–Encoder* et *Hybride Encoder–Decoder*. L'objectif est d'interpréter les performances observées à la lumière des caractéristiques structurelles, syntaxiques et distributionnelles des projets étudiés, tout en identifiant les facteurs influençant la généralisation et les limites inhérentes aux modèles développés.

6.2 Analyse des performances globales

Les résultats montrent que les performances varient considérablement selon les projets et les configurations. Le modèle *Encoder-only Métriques* se distingue par sa stabilité inter-projets, avec un F1-score moyen de 43,2 % et un G-mean de 60,8 %. Cette configuration

constitue la référence la plus robuste parmi les quatre modèles évalués.

À l'inverse, le modèle *Encoder-only AST* affiche les performances les plus faibles (F1-score moyen de 17,2 %). Cette sensibilité pourrait être attribuée aux variations importantes des distributions syntaxiques entre les projets, notamment les différences de longueur moyenne des séquences de tokens AST et la variance associée.

Les configurations hybrides se situent entre ces deux extrêmes. L'architecture *Encoder–Encoder* atteint un F1-score moyen de 36,8 %, tandis que *Encoder–Decoder* obtient 31,7 %. Les performances hybrides restent donc inférieures au modèle basé uniquement sur les métriques orientées objet, mais supérieures à celles du modèle fondé uniquement sur les AST.

6.3 Discussion par configurations

6.3.1 Modèle Encoder-only AST

Les performances faibles observées pour la configuration *Encoder-only AST* pourraient s'expliquer en grande partie par l'hétérogénéité syntaxique entre les projets ainsi que par l'exclusion des structures de contrôle dans la représentation AST, ce qui limite la capture de la complexité logique du code. Les variations de la longueur moyenne des séquences AST, comprises entre 3 500 et 7 000 tokens selon les projets, ainsi que les différences de variance, pourraient influencer directement la capacité du modèle à généraliser. Les projets présentant une forte divergence syntaxique par rapport au jeu d'entraînement — tels que Camel-1.6 ou Log4j-1.2 — montrent des performances particulièrement faibles.

Le faible taux de défauts dans JEdit-4.3 (2,0 %) constitue également un facteur sus-

ceptible d'influencer les performances. Le modèle détecte relativement peu d'instances positives en raison de la rareté des exemples représentatifs, ce qui pourrait contribuer à un F1-score faible malgré un rappel non négligeable.

6.3.2 Modèle Encoder-only Métriques

Le modèle fondé sur les métriques orientées objet semble présenter une capacité de généralisation relativement plus forte. Les métriques OO, telles que WMC, RFC, CBO et LCOM, varient généralement moins fortement d'un projet à l'autre que les représentations AST, ce qui pourrait contribuer à la stabilité observée, même lorsque des écarts importants existent, comme dans Camel-1.6 (WMC = 17,51 ; LCOM = 623,06) ou Log4j-1.2 (WMC = 20,06 ; LCOM = 710,20).

Cependant, des limites apparaissent dans les cas où la distribution des classes est très déséquilibrée. Par exemple, dans JEdit-4.3 (2,0 % de classes défectueuses), le modèle obtient un F1-score faible malgré un rappel de 45 %, ce qui pourrait s'expliquer par un nombre élevé de faux positifs dans ce contexte particulier.

6.3.3 Modèle Hybride Encoder–Encoder

La configuration *Encoder–Encoder* permet de combiner en parallèle les informations extraites des séquences AST et des métriques orientées objet. Cette approche montre des améliorations de performance dans certains projets, notamment Camel-1.6 où elle atteint un F1-score de 48,9 %, supérieur à celui obtenu par les modèles unimodaux dans ce cas précis.

Cependant, cette architecture semble demeurer sensible aux écarts structurels, syn-

taxiques et distributionnels entre les projets. Les performances tendent à diminuer lorsque les divergences avec le jeu d'entraînement sont importantes, comme dans Log4j-1.2 ou JEdit-4.3, où la variabilité syntaxique et les taux de défauts pourraient contribuer à une réduction de la stabilité observée.

6.3.4 Modèle Hybride Encoder–Decoder

Dans cette configuration, les séquences AST sont traitées en premier lieu par l'encodeur, tandis que les métriques orientées objet sont intégrées dans le décodeur. Cette fusion séquentielle semble être plus sensible aux divergences entre les distributions du jeu d'entraînement et celles des jeux de test.

Les performances sont satisfaisantes lorsque les écarts restent modérés, comme dans Ant-1.7 (F1 = 54,0 %), mais tendent à diminuer lorsque les différences augmentent, comme dans Camel-1.6 (F1 = 35,2 %) ou Log4j-1.2 (F1 = 39,1 %). Ces observations suggèrent que l'intégration séquentielle de modalités hétérogènes peut être plus difficile à stabiliser lorsque les caractéristiques syntaxiques ou structurelles diffèrent fortement entre les projets, bien que d'autres facteurs non observés puissent également jouer un rôle.

6.4 Analyse des facteurs influençant la performance

6.4.1 Variabilité structurelle et syntaxique

Les résultats suggèrent que les écarts structurels et syntaxiques pourraient constituer des facteurs influençant la généralisation inter-projets. Les différences observées dans les valeurs de WMC, RFC, LCOM ou dans la longueur et la variance des tokens AST semblent

avoir un impact sur la performance des modèles, bien que cet effet puisse varier selon les configurations.

Par exemple, Camel-1.6 présente des écarts notables (WMC = 17,51 ; variance AST = $1{,}45 \times 10^7$), tandis que Log4j-1.2 cumule à la fois des divergences structurelles importantes et un taux de défauts très élevé (96,4 %). Ces deux systèmes montrent des performances plus limitées pour certaines configurations, ce qui pourrait indiquer une sensibilité accrue aux variations structurelles, syntaxiques ou distributionnelles.

6.4.2 Déséquilibre des classes

L'impact des taux de défauts apparaît particulièrement notable dans JEdit-4.3, où un taux de seulement 2,0 % pourrait limiter la capacité des modèles à identifier correctement les instances positives en raison de la rareté des exemples défectueux. À l'opposé, Log4j-1.2 présente un taux très élevé (96,4 %), ce qui semble modifier le comportement du modèle et peut influencer certains indicateurs, notamment l'accuracy.

6.4.3 Volume et diversité des données

Les projets disposant d'un volume modéré et d'une distribution intermédiaire des défauts, comme Ant-1.7, semblent généralement permettre une évaluation plus stable. À l'inverse, les projets présentant un faible volume ou des distributions extrêmes — JEdit-4.3, Log4j-1.2 — entraînent une variabilité accrue des performances.

6.4.4 Limitation des configurations explorées pour le modèle bi-modal

Du fait que seules quelques configurations du modèle bi-modal ont été explorées, le choix restreint d'architectures, de stratégies de fusion et d'hyperparamètres limite la capacité à établir un lien solide entre les questions de recherche et les résultats obtenus. Une exploration plus large des variantes possibles — par exemple différentes méthodes d'encodage des AST, d'autres mécanismes d'intégration des deux modalités ou un éventail plus riche d'hyperparamètres — aurait permis de vérifier la robustesse des conclusions. L'absence de cette diversité réduit donc la validité interne et constructe des résultats expérimentaux. Une amélioration future consisterait à intégrer explicitement les nœuds liés aux structures de contrôle afin de mieux capturer la complexité cyclomatique.

6.4.5 Variabilité non contrôlée de la taille des AST

Un autre facteur concerne la variabilité non contrôlée de la taille des AST. Les projets du corpus présentent des longueurs de séquences très différentes, ce qui introduit un niveau de variabilité important dans les représentations syntaxiques. Cette hétérogénéité pourrait ajouter un bruit difficile à compenser lors de l'apprentissage et contribuer aux écarts de performance observés entre les projets. L'absence de mécanismes de normalisation ou d'ajustement préalable — tels que le padding ou la restructuration consciente de l'arbre — limite la capacité à isoler l'effet réel des différences structurelles du code, puisque certaines divergences peuvent également découler de simples variations de taille.

6.5 Menaces à la validité

Dans le cadre de cette étude portant sur la prédiction des défauts logiciels à l'aide de modèles d'apprentissage profond basés sur les arbres de syntaxe abstraite (AST) et de métriques orientées objet, plusieurs menaces à la validité doivent être considérées. Ces menaces sont classées en quatre catégories : validité interne, validité externe, validité de construction et validité de conclusion.

6.5.1 Validité interne

Une première menace réside dans le choix des caractéristiques extraites des AST. En effet, seuls certains types de nœuds (par exemple, les invocations de méthodes) ont été considérés. Ce choix, bien que motivé par des considérations de simplification et de faisabilité, conduit à une représentation partielle de la structure du code.

En particulier, les nœuds associés aux structures de contrôle (tels que les instructions conditionnelles *if*, *else*, ou les boucles *while* et *for*) n'ont pas été intégrés dans le processus d'extraction. Or, ces structures sont directement liées à la complexité logique du programme et constituent un facteur important dans l'apparition des défauts logiciels. Cette limitation peut donc affecter la capacité du modèle à capturer des informations pertinentes.

Une autre menace concerne le paramétrage des modèles d'apprentissage profond, notamment ceux basés sur l'architecture Transformer. Les performances obtenues dépendent fortement des hyperparamètres (taille des représentations, nombre de couches, mécanisme d'attention, etc.). Bien que des choix raisonnables aient été effectués, il est possible que d'autres configurations auraient conduit à de meilleurs résultats.

Enfin, les étapes de prétraitement des données (tokenisation, encodage des AST, normalisation des métriques) peuvent introduire des biais. Toute erreur ou simplification excessive dans ces étapes peut influencer les performances finales du modèle.

6.5.2 Validité externe

Les expériences ont été menées sur un ensemble limité de projets logiciels issus de jeux de données publics. Bien que ces jeux de données soient couramment utilisés, ils ne couvrent pas l'ensemble des types de logiciels, des langages de programmation ou des pratiques de développement.

Par conséquent, les résultats obtenus peuvent ne pas être directement généralisables à d'autres contextes industriels, notamment pour des projets de grande envergure ou utilisant des architectures différentes.

De plus, l'hétérogénéité syntaxique entre les projets peut affecter la qualité des représentations basées sur les AST. Les différences dans les styles de codage, les conventions de nommage ou les structures de programme peuvent limiter la capacité du modèle à apprendre des motifs généralisables.

Enfin, l'approche proposée repose sur une représentation spécifique des AST. D'autres méthodes d'encodage, par exemple basées sur des graphes ou sur différents parcours de l'arbre, pourraient produire des résultats différents.

6.5.3 Validité de construction

Les métriques orientées objet utilisées capturent des aspects spécifiques du code tels que la complexité, le couplage et la cohésion, tandis que les représentations basées sur les AST capturent principalement la structure syntaxique. Ces deux types de représentation ne sont pas strictement équivalents, ce qui rend la comparaison indirecte.

De plus, la simplification des AST à un nombre limité de types de nœuds constitue une abstraction qui peut omettre des informations importantes. En particulier, l'absence des structures de contrôle limite la capacité de la représentation à capturer la complexité cyclomatique, reconnue comme un indicateur important des défauts logiciels.

6.5.4 Validité de conclusion

Bien que les résultats expérimentaux montrent que l'approche basée sur les AST et les modèles d'apprentissage profond n'est pas clairement supérieure aux approches basées sur les métriques orientées objet, ces conclusions doivent être interprétées avec prudence.

Tout d'abord, les différences de performance observées peuvent ne pas être statistiquement significatives. L'absence d'analyses statistiques approfondies, telles que des tests de significativité ou des intervalles de confiance, constitue une limite.

Ensuite, la taille des jeux de données et le déséquilibre potentiel entre classes (modules défectueux versus non défectueux) peuvent influencer les résultats et introduire des biais.

Enfin, certaines décisions méthodologiques, telles que le choix des caractéristiques AST ou des hyperparamètres des modèles, peuvent avoir un impact direct sur les performances et donc sur les conclusions tirées.

6.5.5 Synthèse

En résumé, bien que cette étude propose une comparaison rigoureuse entre des approches basées sur les AST et sur les métriques orientées objet, plusieurs limitations doivent être prises en compte. En particulier, la simplification de la représentation des AST, notamment l'absence des structures de contrôle, constitue une menace importante à la validité des résultats.

Ces éléments ouvrent des perspectives d'amélioration, notamment par l'intégration de représentations plus complètes des AST, l'utilisation d'autres architectures d'apprentissage, et la validation sur des ensembles de données plus diversifiés.

6.6 Impact sur les hypothèses

Les résultats présentés dans cette section doivent être interprétés à la lumière des menaces à la validité discutées précédemment, notamment en ce qui concerne la représentation partielle des AST et l'absence des structures de contrôle.

Les résultats obtenus sont analysés au regard des hypothèses de recherche formulées précédemment.

Hypothèse 1 : Les AST fournissent une représentation pertinente pour la prédiction inter-projets. Les résultats obtenus montrent que cette hypothèse n'est que partiellement vérifiée. Les performances de la configuration *Encoder-only AST* restent globalement faibles et fortement dépendantes des variations syntaxiques des projets, ce qui limite leur capacité à généraliser efficacement.

Hypothèse 2 : La combinaison des métriques OO et des AST peut améliorer les performances. Les performances hybrides indiquent que cette hypothèse n'est pas vérifiée de manière systématique. Si la configuration *Encoder–Encoder* permet une amélioration dans certains projets, elle ne surpasse pas de façon constante le modèle basé uniquement sur les métriques orientées objet.

Hypothèse 3 : L'architecture Encoder–Encoder est supérieure à Encoder–Decoder. Les résultats confirment cette hypothèse. La configuration *Encoder–Encoder* obtient des scores moyens supérieurs, bien qu'elle reste sensible à la variabilité inter-projets.

6.7 Conclusion

Les résultats suggèrent que la performance des modèles pourrait dépendre de manière significative des caractéristiques structurelles, syntaxiques et distributionnelles des projets. Dans notre étude, les métriques orientées objet apparaissent comme la modalité la plus stable, tandis que les séquences AST semblent particulièrement sensibles aux variations inter-projets. Les approches hybrides montrent un certain potentiel, mais n'entraînent pas d'amélioration systématique des performances dans le cadre expérimental retenu.

Ces observations soulignent la nécessité d'explorer de nouvelles stratégies de fusion, telles que des mécanismes d'attention croisée ou des modèles capables de pondérer dynamiquement les modalités en fonction des caractéristiques du projet cible.

Chapitre 7

Conclusion générale

Cette recherche avait pour objectif d'étudier, dans un protocole rigoureux de validation inter-projets, l'efficacité comparative de plusieurs architectures Transformer appliquées à la prédiction de défauts logiciels. L'étude a porté sur trois types de représentations : les métriques orientées objet, les séquences issues des arbres syntaxiques abstraits (AST), ainsi que deux formes d'intégration hybride de ces modalités. L'ensemble des expérimentations a été mené sur cinq projets Java open source, choisis pour leur diversité structurelle, syntaxique et distributionnelle.

Les résultats obtenus permettent de dégager plusieurs tendances générales. Tout d'abord, les modèles reposant exclusivement sur les métriques orientées objet ont, dans le cadre des données et des méthodes utilisées dans ce mémoire, présenté une certaine stabilité inter-projets. Cette tendance peut s'expliquer par la variabilité relativement plus faible des mesures structurelles d'un projet à l'autre, ce qui semble favoriser une meilleure généralisation. Il demeure toutefois important de souligner que cette observation n'établit pas une supériorité définitive des métriques OO : les performances restent liées aux caractéristiques spécifiques des projets utilisés, au degré de déséquilibre des classes et aux hyperparamètres retenus.

À l'inverse, les modèles fondés uniquement sur les AST ont montré une sensibilité plus marquée aux différences syntaxiques entre les projets. Les variations observées dans la longueur moyenne des séquences, leur variance et les taux de défauts semblent influencer significativement les performances. Cela suggère que, dans un contexte inter-projets, les représentations syntaxiques nécessitent probablement des mécanismes de normalisation ou d'intégration plus élaborés pour exploiter pleinement leur potentiel, plutôt que d'être considérées comme moins adaptées.

Les architectures hybrides évaluées dans cette étude ont présenté des performances intermédiaires. La configuration *Encoder–Encoder*, qui fusionne les informations syntaxiques et structurelles en parallèle, a montré des résultats intéressants dans certains cas, mais sans amélioration systématique par rapport aux modèles unimodaux. L'approche *Encoder–Decoder*, qui combine les modalités de manière séquentielle, s'est révélée plus sensible aux divergences entre projets, suggérant que la manière dont les deux sources d'information sont intégrées constitue un enjeu méthodologique important.

Dans l'ensemble, les résultats ne permettent pas de conclure de manière définitive à la supériorité d'une représentation sur une autre. Ils indiquent plutôt que, dans les conditions expérimentales testées, les métriques orientées objet constituent une base relativement stable, tandis que les représentations syntaxiques et les approches hybrides montrent un potentiel qui pourrait être mieux exploité avec des mécanismes de fusion plus avancés ou des stratégies d'adaptation de domaine.

Plusieurs perspectives de recherche se dégagent de ces travaux. Il serait notamment pertinent d'explorer des mécanismes de fusion plus expressifs, tels que l'attention croisée ou des représentations jointes apprises de manière contrastive. L'adaptation de domaine, incluant des techniques de normalisation inter-projets ou de sélection dynamique des projets sources, pourrait constituer un levier important pour améliorer la transférabilité des modèles. L'intégration d'autres artefacts logiciels — historique des modifications, informations textuelles, données du processus — représenterait également une piste prometteuse.

Enfin, l'utilisation de modèles Transformer pré-entraînés sur de larges corpus de code pourrait offrir une meilleure robustesse face à la variabilité inter-projets observée dans cette étude.

En somme, ce mémoire met en évidence des tendances concernant l'utilisation conjointe ou séparée des métriques orientées objet et des représentations syntaxiques pour la prédiction de défauts logiciels, tout en soulignant la nécessité de travaux complémentaires pour confirmer, approfondir ou nuancer les observations formulées. Les conclusions doivent être interprétées à la lumière des limites inhérentes au corpus utilisé, à l'hétérogénéité des projets et aux choix architecturaux réalisés. Les perspectives proposées ouvrent ainsi la voie à de nouvelles explorations méthodologiques visant à renforcer la robustesse et la transférabilité des approches d'apprentissage profond appliquées au génie logiciel.

Bibliographie

[1] Abdu, Ahmed et al. (2022). « Deep Learning-Based Software Defect Prediction via Semantic Key Features of Source Code—Systematic Survey ». In : *Mathematics* 10.17, p. 3120. doi : 10.3390/math10173120.

[2] Aho, Alfred V et al. (2006). *Compilers : Principles, Techniques, and Tools*. 2nd. Addison Wesley.

[3] Allamanis, Miltiadis et al. (2018). « A survey of machine learning for big code and naturalness ». In : *ACM Computing Surveys* 51.4, p. 1-37. doi : 10.1145/3212695.

[4] Basili, Victor R et al. (1996). « A validation of object-oriented design metrics as quality indicators ». In : *IEEE Transactions on Software Engineering* 22.10, p. 751-761.

[5] Briand, Lionel C et al. (2000). « Exploring the relationships between design measures and software quality in object-oriented systems ». In : *Journal of Systems and Software* 51.3, p. 245-273.

[6] Brown, Tom et al. (2020). « Language models are few-shot learners ». In : *Advances in Neural Information Processing Systems* 33, p. 1877-1901.

[7] Chakraborty, Saikat et al. (2021). « Deep learning based vulnerability detection : Are we there yet ? » In : *IEEE Transactions on Software Engineering* 48.9, p. 3280-3296.

[8] Chawla, Nitesh V et al. (2002). « SMOTE : synthetic minority over-sampling technique ». In : *Journal of Artificial Intelligence Research* 16, p. 321-357.

[9] Chen, Zhangyin et al. (2021). « Transformers for software engineering : A systematic literature review ». In : *ACM Transactions on Software Engineering and Methodology* 31.3, p. 1-37.

[10] Chidamber, Shyam R et al. (1994). « A metrics suite for object oriented design ». In : *IEEE Transactions on Software Engineering* 20.6, p. 476-493.

[11] D'Ambros, Marco et al. (2010). *The PROMISE repository of empirical software engineering data*. http://promise.site.uottawa.ca/SERepository. Accessed : 2023-12-01.

[12] Dam, Hoa Khanh et al. (2018). « A deep learning approach for software defect prediction ». In : *IEEE Transactions on Software Engineering* 45.2, p. 194-208.

[13] Devlin, Jacob et al. (2018). « BERT : Pre-training of Deep Bidirectional Transformers for Language Understanding ». In : *arXiv preprint arXiv:1810.04805*.

[14] Feng, Zhangyin et al. (2020). « CodeBERT : A pre-trained model for programming and natural languages ». In : *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing : Findings*, p. 1536-1547.

[15] Ghotra, Baljinder et al. (2015). « Revisiting the impact of classification techniques on the performance of defect prediction models ». In : *2015 IEEE/ACM*

37th IEEE International Conference on Software Engineering. T. 1. IEEE, p. 789-800.

[16] Gorchakov, Artyom V. et al. (2023). « Analysis of Program Representations Based on Abstract Syntax Trees and Higher-Order Markov Chains for Source Code Classification Task ». In : *Future Internet* 15.9, p. 314. doi : 10.3390/fi15090314. url : https://doi.org/10.3390/fi15090314.

[17] Hall, Tracy et al. (2012). « A systematic literature review on fault prediction performance in software engineering ». In : *IEEE Transactions on Software Engineering* 38.6, p. 1276-1304. doi : 10.1109/TSE.2011.103.

[18] He, Zhimin et al. (2012). « An investigation on the feasibility of cross-project defect prediction ». In : *Automated Software Engineering* 19.2, p. 167-199.

[19] Herbold, Steffen et al. (2018). « A Comparative Study to Benchmark Cross-project Defect Prediction Approaches ». In : *IEEE Transactions on Software Engineering* 44.9, p. 811-833. doi : 10.1109/TSE.2017.2724538.

[20] Jureczko, Marian et al. (2010). « Towards identifying software project clusters with regard to defect prediction ». In : *Proceedings of the 6th international conference on predictive models in software engineering*. ACM, p. 1-10.

[21] Kovalenko, Vladimir et al. (2019). « PathMiner : A Library for Mining of Path-Based Representations of Code ». In : *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, p. 13-17. doi : 10.1109/MSR.2019.00013.

[22] Li, Jian et al. (2017). « Software defect prediction via convolutional neural network ». In : *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, p. 318-328.

[23] Li, Yi et al. (2022). « VulRepair : a T5-based automated software vulnerability repair ». In : *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, p. 935-947.

[24] Li, Zheng et al. (2019). « Multi-modal deep learning for software defect prediction ». In : *IEEE Access* 7, p. 46595-46605.

[25] Liubchenko, Natalia et al. (2023). « Software defect prediction using machine learning : A systematic literature review ». In : *Information and Software Technology* 163, p. 107364. doi : 10.1016/j.infsof.2023.107364. url : https://doi.org/10.1016/j.infsof.2023.107364.

[26] Loshchilov, Ilya et al. (2019). « Decoupled weight decay regularization ». In : *International Conference on Learning Representations*.

[27] Malhotra, Ruchika (2015). « A systematic review of machine learning techniques for software fault prediction ». In : *Applied Soft Computing* 27, p. 504-518. doi : 10.1016/j.asoc.2014.11.023.

[28] Mohan, Konda Reddy et al. (2018). « DeepFault : fault localization for deep neural networks ». In : *International Conference on Fundamental Approaches to Software Engineering*. Springer, p. 171-191.

[29] Moudache, Sarra et al. (2022). « Bad smells detection using machine learning techniques : A systematic literature review ». In : *Journal of Software Engineering Research and Development* 10.1, p. 1-35.

[30] Pedregosa, Fabian et al. (2011). « Scikit-learn : Machine learning in Python ». In : *Journal of Machine Learning Research* 12, p. 2825-2830.

[31] Peters, Fayola et al. (2013). « Evaluating software defect prediction approaches : A benchmark and an extensive comparison ». In : *Empirical Software Engineering* 17.3, p. 530-577. doi : 10.1007/s10664-012-9191-4.

[32] Radjenović, Danijel et al. (2013). « Software fault prediction metrics : A systematic literature review ». In : *Information and Software Technology* 55.8, p. 1397-1418.

[33] Rusak, Gili et al. (2018). « POSTER : AST-Based Deep Learning for Detecting Malicious PowerShell ». In : *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. doi : 10.1145/3243734.3278496. url : https://arxiv.org/abs/1810.09230.

[34] Russell, Rebecca et al. (2018). « Automated vulnerability detection in source code using deep representation learning ». In : *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, p. 757-762. doi : 10.1109/ICMLA.2018.00120.

[35] Sejm, Marcin et al. (2017). « Ensemble methods for software defect prediction : A survey ». In : *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, p. 1135-1142.

[36] Singh, Yogesh et al. (2013). « Investigation of fault prediction capabilities of object-oriented metrics ». In : *2013 3rd IEEE International Advance Computing Conference (IACC)*. IEEE, p. 370-374.

[37] Al-Smadi, Yazan et al. (2023). « Reliable prediction of software defects using Shapley interpretable machine learning models ». In : *Egyptian Informatics Journal* 24.3, p. 100386. doi : 10.1016/j.eij.2023.05.011.

[38] Subramanyam, Ramanath et al. (2003). « Empirical analysis of CK metrics for object-oriented design complexity : Implications for software defects ». In : *IEEE Transactions on Software Engineering* 29.4, p. 297-310.

[39] Turhan, Burak et al. (2009). « On the relative value of cross-company and within-company data for defect prediction ». In : *Empirical Software Engineering* 14.5, p. 540-578.

[40] Vaswani, Ashish et al. (2017). « Attention is all you need ». In : *Advances in Neural Information Processing Systems* 30, p. 5998-6008.

[41] Wang, Song et al. (2016). « Automatically learning semantic features for defect prediction ». In : *IEEE Transactions on Software Engineering* 44.4, p. 324-343.

[42] Wang, Yue et al. (2021). « CodeT5 : Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation ». In : *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, p. 8696-8708. doi : 10.18653/v1/2021.emnlp-main.685.

[43] White, Martin et al. (2016). « Deep learning code fragments for code clone detection ». In : *Proceedings of the 31st IEEE/ACM International Conference on*

Automated Software Engineering (ASE). IEEE, p. 87-98. DOI : 10.1145/2970276.2970326.

[44] YEE-KING, Matthew et al. (2020). « Examining Student Coding Behaviours in Creative Computing Lessons using Abstract Syntax Trees and Vocabulary Analysis ». In : *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, p. 273-279. DOI : 10.1145/3341525.3387408.

[45] ZHANG, Yuming et al. (2019). « Software defect prediction based on stacked denoising autoencoders and two-stage ensemble learning ». In : *Information and Software Technology* 96, p. 94-111.

[46] ZIMMERMANN, Thomas et al. (2009). « Cross-project defect prediction : a large scale experiment on data vs. domain vs. process ». In : *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, p. 91-100.