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
IVAN GABRIEL ROMERO RENGEL

PRÉDICTION DES CLASSES À TESTER DANS UN LOGICIEL ORIENTÉ OBJET: VERS UNE APPROCHE BASÉE SUR LA REPRÉSENTATION MATRICIELLE ET L'APPRENTISSAGE MACHINE

Septembre 2023

Résumé

Les logiciels larges et complexes de nos jours constituent un défi important en matière de maintenance, les tests et la résolution de bogues trouvés lors des phases de développement et de maintenance des systèmes, sont source d'une forte consommation de ressources et de temps.

En effet, durant tout leur cycle de vie, les produits logiciels sont appelés, entre autres, à être testés, l'identification des classes importantes pour être testées est un autre processus complexe, il y a des classes qui sont clés et d'autres moins importantes pour le fonctionnement d'un produit logiciel. Les classes plus importantes pourraient occasionner des problèmes plus graves s'il y a des méthodes ou de bouts de code qui ne sont pas testés convenablement, lors du développement de nouvelles fonctionnalités, des déploiements, etc. En conséquence, dans le processus de test, nous souhaitons découvrir le maximum de bugs, sinon le processus de correction poste de déploiement pourrait constituer un énorme gouffre de ressources.

Dans cette étude, nous tentons par une approche de représentation matricielle des systèmes orientés objet et des algorithmes d'apprentissage machine, d'identifier les classes candidates à tester en partant de leur point d'observation.

Mot clés: Convolutif, Réseaux neuronaux, Apprentissage de machine, Apprentissage profond; Prédiction des classes à tester, Métriques, Qualité.

Abstract

Large and complex software nowadays poses a significant maintenance challenge, testing and resolving bugs found during the development and maintenance phases of systems is a high resource and time-consuming.

Indeed, throughout their life cycle, software products are called, among other things, to be tested, the identification of important classes to be tested is another complex process, there are classes that are key and others less important for the operation of a software product. Larger classes could cause more serious problems if they contain untested methods or code in the moment of new feature development, deployments, etc. As a result, in the testing process, we may have more information at the time that the product had a flaw, otherwise this process could be a huge drain on resources.

In this study, we attempt by a matrix representation approach of object oriented systems and machine learning algorithms, to identify the candidate classes to be tested starting from their point of observation.

Key words: Convolutional; Neural Network; Deep Learning; Prediction of Classes to Test, Metrics, Quality.

Remerciements

Je me rends en premier lieu à Dieu, maître des temps et des circonstances pour sa grâce dans ma vie.

À mon directeur de recherche Fadel Touré, qui m'a fait confiance et surtout pour son encadrement, leur connaissance et expérience enrichissante, je trouve ici l'expression de mon profond respect et de ma profonde gratitude.

Je tiens aussi à rendre hommage à mes parents HECTOR ROMERO (d'heureuse mémoire) et ROSARIO RENGEL qui m'ont toujours encouragés à étudier et pour tous leurs bienfaits.

Je remercie également les membres de ma famille pour leurs encouragements et leur soutien indéfectible. Particulièrement à ma femme IDANIA MEJIAS et à mes enfants SAMUEL, PAOLA et IVANNA ROMERO MEJIAS pour toute forme d'assistance louable et substantielle.

J'adresse mes sincères remerciements à ce pays Canada, qui m'a permis, à moi et à ma famille, d'obtenir les moyens nécessaires, comme l'éducation, pour avoir une bonne qualité de vie. En définitive, à toute personne m'ayant apporté son aide d'une manière ou d'une autre pour l'accomplissement du présent travail, qu'elle trouve ici l'expression de ma reconnaissance.

Tables des matières

Liste des figures

Liste des tableaux

Liste des graphiques

Liste des symboles

CBO	(Coupling Between Objects), cette métrique compte pour une classe logicielle donnée, elle détermine le nombre de classes auxquelles elle est couplée et vice versa.
WMC	(Weighted Methods per Class), cette métrique donne la somme des complexités des méthodes d'une classe donnée, où chaque méthode est pondérée par sa complexité cyclomatique [1]. Seules les méthodes spécifiées dans la classe sont prises en compte.
LOC	(Lines Of Code per class), cette métrique compte pour une classe donnée son nombre de lignes de code source.
Intellij IDEA	Integrated DEvelopment Environment (IDE) pour languages JVM.
ROC	(Receiver Operating Characteristic), le ROC est un graphique montrant les performances d'un modèle de classification à tous les seuils de classification. Cette courbe trace deux paramètres, Taux de vrais positifs et Taux de faux positifs.
SMOTE	(Synthetic Minority Oversampling TEchnique) approche de sur-échantillonnage dans laquelle la classe minoritaire est sur-échantillonnée en créant des exemples synthétiques [2].
JUC	Java Unit Coverage.

Chapitre 1

Introduction

Les logiciels orientés objet sont composés d'unité conceptuelle de codes regroupant des données et des fonctions qui constituent les classes/modules, différentes études ont prouvé que la majorité des fautes sont concentrées dans quelques modules [3], des modules logiciels défectueux causent des pannes logicielles, augmentent les coûts de développement et de maintenance et diminuent la satisfaction des clients [4]. Ces constatations dénotent l'importance de tester précocement et rigoureusement les unités de manière individuelle (test unitaire) et de manière groupées (test d'intégration), afin d'améliorer la qualité du logiciel, la gestion du développement, la gestion des ressources et la qualité de processus dans sa globalité. L'objectif final est de réduire le temps de développement, de maintenance et d'effort de test pour obtenir une grande satisfaction du client.

Actuellement, il y a des métriques d'assurance de la qualité de logiciel (statique et dynamique), qui permettent d'évaluer entre autres, la complexité et le couplage reliés à différentes caractéristiques de la qualité d'un logiciel. Cependant, ces métriques ont montré des limitations dans les applications modernes orientées objet en raison de la présence d'artefacts orientés objet tels que le polymorphisme, les liaisons dynamiques, l'héritage et du code inutilisé [5].

Pendant cette étude, nous avons proposé notre propre métrique de logiciel sous forme de représentation matricielle basée sur la collaboration et les liens entre classes logicielles, dans le but de capturer la structure des liaisons existant dans le logiciel.

1.1 Objectif

L'objectif principal de cette étude est l'identification des classes qui doivent être testées unitairement en se basant sur une représentation matricielle des interactions et du couplage entre les classes logicielles. Cette approche repose sur le développement d'un plugiciel pour l'IDE Intellij qui permet l'extraction de l'information sur le couplage et d'autres liens entre les classes d'un système logiciel et la création d'une représentation matricielle de ces liens.

La représentation matricielle créée sera étiquetée par l'historique de ses données de tests unitaires basées sur deux métriques logicielles, ensuite un modèle prédictif sera utilisé permettant de suggérer les classes candidates à tester dans un système logiciel.

1.2 Problématique

Dans tout le cycle de vie d'un logiciel, le processus de détection des fautes est toujours présent, devant l'impossibilité de tester systématiquement toutes les classes dans les grands systèmes logiciels, l'identification des classes à tester prend une grande importance dans l'aide qu'elle peut apporter dans le processus pour identifier les modules, classes ou parties du code source à tester en priorité. Le processus de test unitaire est très complexe et il est relié à différents processus dans le cycle de développement du logiciel.

L'utilisation des modèles de prédiction basées sur l'apprentissage automatique pour l'identification des classes candidates dans le processus de test unitaire peut améliorer le temps du processus de développement et de test, avec une réduction des coûts pour l'entreprise. Par contre, les modèles créés ou développés ne sont pas simples à mettre en place et chacun présente ses avantages et inconvénients.

Dans cette étude, nous allons proposer une représentation matricielle basée sur des métriques de code source de relation entre les classes d'un système logiciel à partir de la création d'un plugiciel pour l'IDE Intellij, cette matrice pourrait être utilisée dans un modèle d'apprentissage profond pour la prédiction de classes candidates aux tests unitaires qui permettra d'aider dans le processus de décision de création de test unitaire.

1.3 Organisation du Mémoire

Ce travail est subdivisé en six principaux chapitres. Le premier chapitre est consacré à l'introduction générale, ce chapitre expose nos motivations et objectifs poursuivis dans ce travail de recherche.

Dans le deuxième chapitre, nous aborderons l'état de l'art dans le contexte de l'identification des classes candidates aux tests unitaires, basées sur la présentation des études existantes en la matière, leurs apports et limitations. Le troisième chapitre est consacré à la collecte des données et méthodes d'analyse, nous allons proposer notre question de recherche, les systèmes logiciels utilisés, les outils de collecte de données, le processus de la création du plugiciel, l'information des métriques de logiciels utilisés et les méthodes d'évaluation de performance.

Le quatrième chapitre sera consacré aux expérimentations utilisées et développées pour notre travail de recherche. Ce chapitre présentera également des informations sur l'apprentissage automatique, particulièrement les réseaux neuronaux et l'information sur le processus de la création du modèle prédictif.

Le cinquième chapitre présentera les résultats obtenus pour la représentation matricielle et le modèle d'apprentissage profond et les interprétations associées. Le sixième et dernier chapitre de ce travail mettra en exergue les menaces pour la validité, les limites et conclura notre étude.

Chapitre 2

État de l’art

L'analyse des classes candidates au test est un sujet exploré sous différentes études, nous pouvons mentionner les suivantes:

Fadel Toure, Mourad Badri et al. [6, 7] dans leur étude, ont mis en place une approche basée sur l'historique des informations logicielles pour prendre en charge la hiérarchisation des classes à tester. Ils ont d'abord analysé différents attributs de dix systèmes logiciels de code ouvert en Java pour lesquels des cas de test JUnit ont été développés pour plusieurs classes. Ils ont utilisé l’analyse de la moyenne et de la régression logistique pour caractériser les classes pour lesquelles les classes test JUnit ont été développées par les testeurs. Deuxièmement, ils ont utilisé deux classificateurs formés sur les valeurs de métriques et les informations de tests unitaires collectées à partir des systèmes sélectionnés, les classificateurs fournissent, pour chaque logiciel, un ensemble de classes sur lesquelles les efforts de tests unitaires doivent être concentrés. Les ensembles obtenus ont été comparés aux ensembles de classes pour lesquelles les classes de test JUnit ont été développées par les testeurs. Les résultats montrent que: (1) les valeurs moyennes des métriques des classes testées identifiées sont très différentes des valeurs moyennes des métriques des autres classes, (2) les attributs d'une classe influencent la décision du développer ou non une classe de test JUnit dédiée à celle-ci, et (3) les ensembles de classes suggérées par les classificateurs reflètent assez correctement la sélection des testeurs.

Mourad Badri et Fadel Toure [8] dans leur étude ont utilisé de manière empirique la relation entre les métriques de conception orientés objet, la testabilité des classes et l’effort de test unitaire, ils ont utilisé trois systèmes logiciels de code ouvert Java pour lesquels les tests JUnit existaient, les classes ont été classifiées, selon l’effort de test requis en deux catégories: haut et bas. Les algorithmes qu’ils ont utilisés sont la régression logistique univariée et multivariée pour connaître les relations entre les variables. La performance des modèles de prédiction a été évaluée en utilisant les courbes ROC. Les résultats montrent que la complexité, la taille, la cohésion et dans quelques situations le couplage sont de bons prédicteurs de l’effort de test unitaire des classes et que le modèle de régression multivariée est capable de prédire, avec une bonne précision, l’effort de test unitaire des classes.

Alexandre Boucher et Mourad Badri [9] ont proposé un modèle de prédiction de fautes, Multiple Risk Levels (MRL) qui permet de catégoriser les classes d’un système en plusieurs niveaux de risques et ne nécessitant aucune donnée sur les fautes pour être construit, les résultats obtenus démontrent que ce modèle permet de bien détecter les fautes dans un système logiciel. Cette recherche

est nommée dans notre état de l'art parce qu'à partir de la détection des fautes, les classes identifiées pourraient être candidates aux tests unitaires.

Yogesh Singh, Arvinder Kaur et collab. [10], ont étudié la prédiction d'effort de test avec des techniques de Réseaux de Neurones Artificiels (RNA), ces techniques sont présentées comme outils pour la prédiction des attributs de qualité. Dans leurs études, ils utilisent le RNA pour la prédiction de la qualité logicielle en utilisant des métriques orientés objet de Chidamber et Kemerer [11]. Pour la validation, ils ont utilisé les données publiques de la NASA. Ils ont trouvé une relation significative entre les métriques (CK) et l'effort de test mesuré en utilisant le nombre de lignes modifiées par classe logicielle. Le modèle a démontré qu'il est capable d'estimer l'effort de test dans les 35% de l'effort réel dans 72.54% des classes avec une Erreur Relative Absolue Moyenne (RMEA) de 0.25.

Mourad Badri et Fadel Toure [12] dans leur étude, se sont concentrés sur la façon de cibler automatiquement les classes, les candidats aux tests unitaires. Leur approche repose sur les algorithmes de classificateurs entraînés sur différentes informations de tests unitaires et les métriques de code source collectées à partir de différents systèmes logiciels.

Wyao Matcha, Fadel Toure et collab. [13] dans leur étude, se sont concentrés sur les tests unitaires des classes et en particulier sur la façon de suggérer automatiquement des classes appropriées pour les tests unitaires utilisant l'apprentissage profond et des informations sur les deux historiques des tests unitaires et métriques du code source.

Fadel Toure et Mourad Badri [14] ont étudié la façon de cibler automatiquement les classes candidates aux tests unitaires, leur objectif à long terme est de construire un outil collaboratif pour la communauté des développeurs. Cet outil collectera les métriques du code source et le test unitaire des classes, des informations provenant de différents projets afin d'améliorer les performances d'un classificateur centralisé, classificateur hébergé dans le nuage pour correspondre à la sélection par les testeurs des classes candidates aux tests unitaires. Pour les nouveaux systèmes en cours de développement, l'outil pourrait suggérer, après collecte des métriques de code source spécifiques, un ensemble de classes candidates aux tests unitaires. En raison du grand code source, la diversité et la quantité croissante de données auxquelles l'outil sera confronté, ils ont envisagé d'utiliser des modèles de réseaux de neurones profonds entraînés sur des ensembles de données des systèmes combinés pour explorer la précision des classificateurs. Les résultats de cette étude montrent qu'il était possible de faire correspondre correctement les classes candidates aux tests unitaires proposés par les testeurs. De plus, les résultats ont indiqué que les classes pourraient être bien prédits avec plus de 93 % des précisions.

En général, les études pour sélectionner les classes de tests les plus prometteurs pour détecter les fautes sont variées s'il y a des incertitudes sur l'impact des modifications dans le code ou si des liens de traçabilité entre le code et les tests ne sont pas disponibles.

Dans le prochain chapitre, nous poserons nos questions de recherche et exposerons les systèmes logiciels utilisés à partir des statistiques descriptives, en référence au processus de collecte

de données, nous allons expliquer le but de la création du plugiciel et l'utilisation des données déjà collectée basée sur des métriques logicielles et pour la méthode d'analyse, nous expliquerons les processus utilisés de façon générale.

Chapitre 3

Collecte de données et méthode d'analyse

3.1 Question de recherche

Nous nous sommes proposé de répondre aux questions de recherche suivante dans notre étude:

QR1: Comment pouvons-nous créer un plugiciel qui permettra de proposer des classes candidates pour le test à partir d'une analyse des classes d'un système logiciel basé sur ses relations d'héritage et des couplages afin de supporter les testeurs dans la phase des tests unitaires ?

À partir de l'environnement de développement logiciel IntelliJ, nous allons implémenter un plugiciel dans le but de bâtir une matrice relationnelle des classes avec des métriques logicielles d'héritage et de couplage pour pouvoir répondre à cette question.

QR2: Pourrait-on avoir une meilleure suggestion en utilisant cette représentation matricielle et l'apprentissage profond dans le choix des classes candidates durant la phase des tests unitaires ?

À partir, de l'utilisation de l'apprentissage profond et des informations historiques qui permettent d'étiqueter les classes, nous allons construire des bases de données d'entraînement et d'évaluation de performance, pour pouvoir répondre à cette question.

3.2 Systèmes logiciels sélectionnés

Les codes sources de 3 systèmes logiciels open source orientés objet développés en Java ont été extraits des référentiels publics et décrits ci-dessous. Pour chaque système, seul un sous-ensemble de classes a été testé à l'aide du cadre logiciel JUnit [15], ce cadre logiciel a été utilisé pour créer et écrire les tests unitaires automatisés pour les classes logiciels dans le langage Java.

- ***IO :*** Apache IO [16], est une librairie développée par Apache Software Foundation [17], c'est une librairie d'utilitaires standards permettant de manipuler les entrées/sorties des différents systèmes d'exploitation, la version utilisée était 2.11.0.
- ***JFC :*** JFreeChart [18], est une bibliothèque Java graphique qui permet l'intégration de graphiques de qualité professionnelle dans une application. JFreeChart supporte aussi différents types de sorties, y compris les composants Swing et JavaFX, il permet d'exporter les graphiques dans des formats variés d'images et graphiques vectoriels, la version utilisée était la 1.5.2

- ***MATH :*** Commons Math [19], est une librairie légère qui contient des fonctionnalités mathématiques et statistiques les plus communes qui ne sont pas disponibles dans la classe Math de Java, la version utilisée est la 3.6.1.

Pour cette recherche, afin de présenter les différentes propriété de nos trois systèmes sélectionnés, nous avons utilisé trois métriques de code source : LOC, WMC et CBO, ces métriques ont été sélectionnées, parce qu'elles ont reçu une attention considérable de la part des chercheurs et sont également de plus en plus adoptées par les références. De plus, ces mesures ont été validées dans de nombreuses études empiriques en tant qu'indicateurs significatifs de divers attributs de qualité logicielle.

Le tableau 3.1 résume certaines caractéristiques des systèmes analysés. Il donne, pour chaque système : (1) le nombre total de classes de code source, (2) le nombre total de lignes de code des classes de code source, (3) le nombre de classes pour lesquelles des cas de test JUnit ont été développés, (4) le nombre total de lignes de code des cas de test JUnit, (5) le pourcentage de classes de code source pour lesquelles des cas de test JUnit ont été développés, et (6) le pourcentage de lignes de code testées.

	(1)	(2)	(3)	(4)	(5)	(6)
IO	402	59261	211	26316	52,49%	44,41%
JFC	921	210789	369	48150	40,07%	22,84%
MATH	1801	284371	813	113242	45,14%	39,82%

Tableau 3.1 Quelques statistiques sur les systèmes logiciels sélectionnés

D'après le tableau 3.1, les colonnes 1 et 2 montrent que les trois systèmes sont différents en matière de lignes de code, IO a moins de lignes de code et MATH contient plus de lignes de code (59261 et 284371 respectivement), cette relation est la même en matière de nombre de classes (402 et 1801 respectivement). De plus, le système qui contient plus de classes testées est IO (52.49%) et le système avec moins de classes testées est JFC (40.07%), par contre, le système avec plus de lignes de codes testées est IO (44.41%) et le système avec moins de lignes de code testées est JFC (40.07%). En résumé, les trois systèmes sont différents les uns des autres en matière de nombre de classes et du taux de couverture des tests (en matière de code JUnit). Finalement, nous pouvons observer à partir du tableau 3.1 que le système IO a moins de classes et de lignes de code source, mais présente plus de classes et de lignes de code testées, MATH est le système plus grand en termes de classes et de lignes de code source. JFC est le système le moins couvert par les tests unitaires en termes de classes et de lignes de code testées.

Le tableau 3.2 résume les statistiques descriptives des métriques de code source sélectionnées pour les classes pour lesquelles des cas de test JUnit ont été développés. Nous pouvons observer pour le système MATH, les classes pour lesquelles les cas de test JUnit ont été développés sont plus complexes (WMC : 14.37) et plus grand (LOC : 139.29), la même tendance peut être observée pour le couplage (CBO: 7.88), entre les systèmes IO et JFC, les cas de test JUnit sont plus complexes pour IO (WMC: 11.75), mais les cas de test JUnit sont plus grands pour JFC (LOC: 130.49). La colonne d'écart type indique que les classes d'IO testées sont plus homogènes en matière de couplage (3.34), en matière de lignes de code et de complexité (156.04 et 7.59 respectivement) les classes de JFC sont plus homogènes.

		Obs.	Min.	Max.	Avg.	Std
IO	**CBO**	211	0	31	3.64	3.34
	LOC	211	1	2659	124.72	252.80
	WMC	211	0	220	11.75	20.77
JFC	**CBO**	369	0	297	6.55	15.95
	LOC	369	6	1247	130.49	156.04
	WMC	369	0	64	6.82	7.59
MATH	**CBO**	813	1	100	7.88	7.49
	LOC	813	2	1698	139.29	214.45
	WMC	813	0	283	14.37	24.99

Tableau 3.2 Statistiques descriptives des métriques de code source

Pour obtenir les statistiques descriptives, nous avons utilisé la technique de liaison préfixe/suffixe, comme d'autres auteurs [20, 21, 22], pour faire correspondre chaque classe de logiciel à sa classe de test JUnit, nous avons remarqué que les développeurs nomment généralement les classes de cas de test JUnit en ajoutant le préfixe (suffixe) "Test" ("TestCase") au nom des classes pour lesquelles les cas de test JUnit ont été développés. Seules les classes qui ont un tel mécanisme de correspondance de nom avec le nom de la classe de cas de test ont été incluses dans l'analyse. Les valeurs des métriques orientées objet ont été calculées à l'aide du plugiciel Metrics Reloaded Intellij [23].

3.3 Outils de collecte de données

Pour cette étude, nous avons développé un plugiciel pour l'IDE IntelliJ, le but de ce plugiciel est de créer un outil pour soutenir la priorisation des tests unitaires basés sur des informations de tests

unitaires et certaines métriques spécifiques, ce plugiciel permettra à l'utilisateur d'avoir une représentation matricielle entre les classes, les métriques que nous utilisons sont des relations de couplage et d'héritage entre les classes, les valeurs sont binaires, seulement nous allons avoir l'information s'il y a une relation de couplage ou d'héritage (1) ou pas (0). Cette représentation matricielle est stockée dans un fichier sous format csv.

` Les systèmes sélectionnés ont été testés avec le cadre logiciel JUnit, une typique utilisation de JUnit permet de tester chaque classe avec sa propre classe de test, cette classe de test sera exécutée par JUnit, le résultat sera les méthodes sans et avec erreurs.

Le score JUC est basé sur l'invocation de la classe de test unitaire, représentant pour chaque classe le pourcentage de lignes de code couvertes par l'ensemble des classes de tests unitaires [24].

Dans les classes testées, nous avons utilisé les métriques suivantes:

- *JUC-Tested* est une métrique basée sur l'invocation des classes de test unitaire, représentant pour chaque classe, le pourcentage de lignes de code logiciel couvert par l'ensemble des classes des tests unitaires [24], les valeurs binaires indique si la classe a été effectivement testée lors du test unitaire par l'analyse de la couverture de test.
- *Tested* est une métrique qui indique si une classe de logiciel a une classe de test unitaire dédiées ou pas [24].

Ensuite, un prétraitement des données est fait pour lier la représentation matricielle et la classification utilisant les métriques *JUC-Tested* et *Tested* dans chaque système logiciel sélectionné, les classes ont été évaluées et classifiées comme candidates au test unitaire (1) ou pas (0).

3.3.1 Métriques logicielles proposées

Dans notre étude, nous avons besoin de métriques qui ont les relations entre les différentes classes logicielles. Nous considérons les relations d'héritage et de couplage. Les métriques de logiciel que nous allons utiliser sont détaillées ci-dessous:

- *Is Inheritance Used (IIU)* : indique une relation d'héritage entre *Ci* et *Cj*. La valeur de IIU est binaire.
- *Is Class Called (ICC)* : indique l'appelle de la classe *Ci* par la classe *Cj*, il s'agit aussi de valeur binaire.

3.4 Collecte de données

Les étapes réalisées au cours du processus de collecte de données sont les suivantes.

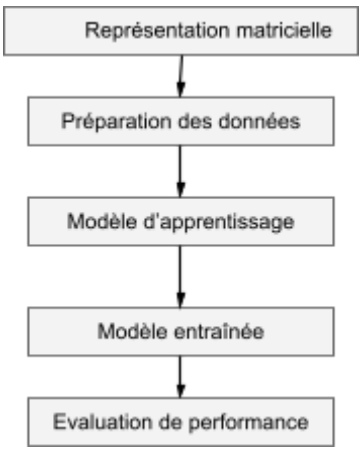


Fig 3.1 Étapes réalisées au cours du processus de collecte des données

3.4.1. Représentation matricielle

Dans cette étape, les données sont récoltées par le plugiciel dans le système logiciel, le résultat est une représentation matricielle stockée dans un fichier csv avec les objets de classes et les métriques de code source qui montrent la connexion (1= connecté, 0=pas connecté) entre elles.

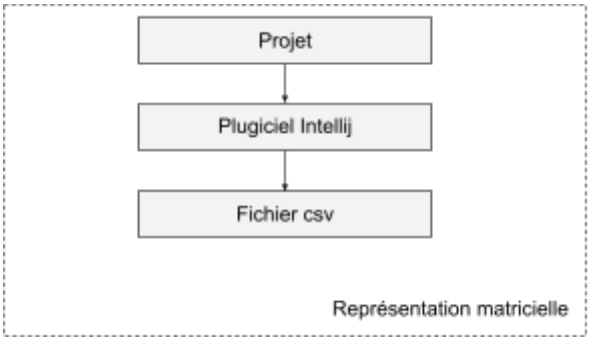


Fig 3.2 Sous-processus dans l'étape de représentation matricielle

3.4.2. Préparation des données

Les données sont nettoyées et sont liées aux autres données qui contiennent l'étiquette de catégories des tests unitaire, cette classification est faite séparément. Dans cette étape les données sont équilibrées (un ensemble de données est déséquilibré si les catégories de classification ne sont pas également représentées [2]), liées et séparées entre les données d'entraînement et de test.

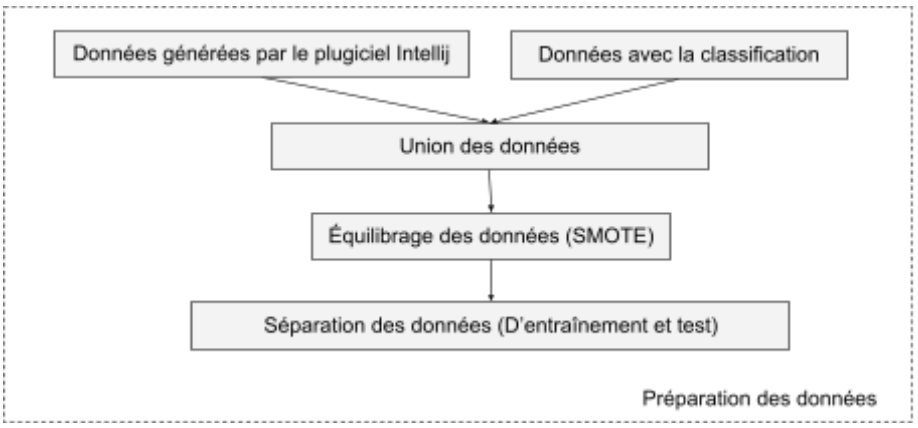


Fig 3.3 Sous-processus dans l'étape de préparation des données

3.4.3. Modèle d'apprentissage

Une fois que les données sont préparées et séparées entre données d'entraînement et de test, le pourcentage pris pour la séparation est 70 % entraînement et 30 % test respectivement, le modèle d'apprentissage qui sera détaillé dans le chapitre suivant va être créé.

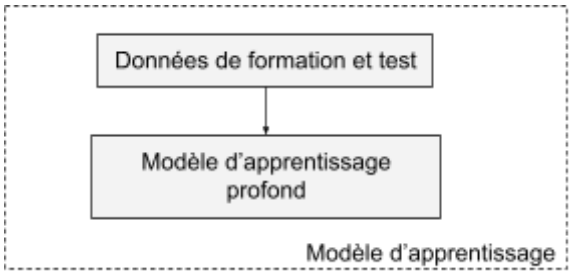


Fig 3.4 Sous-processus dans l'étape modèle d'apprentissage

3.4.4. Modèle entraîné

Le modèle d'apprentissage profond est entraîné, basé sur les données d'entraînement, avant d'entraîner le modèle, nous devions définir l'optimiseur, compiler et sélectionner le nombre de passages (epochs) optimal.

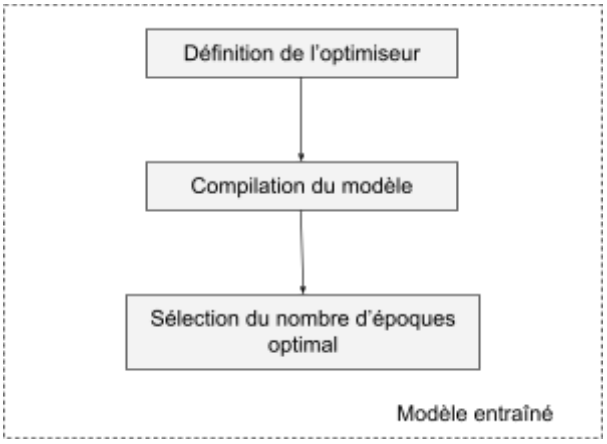


Fig 3.5 Sous-processus dans l'étape de modèle entraîné

3.4.5. Evaluation de performance

Avec le modèle entraîné, nous avons évalué les performances de classification en se basant sur quelques métriques d'évaluation de performance telles que : l'exactitude, la précision, la sensibilité, la spécificité, le F1-score et la moyenne géométrique [25].

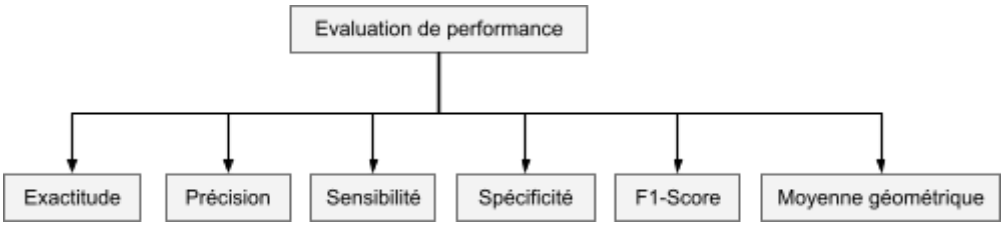


Fig 3.6 Sous-processus dans l'étape d'évaluation de performance

3.5 Création de plugiciel

Pour la création de notre plugiciel, nous avons utilisé DevKit [26] fourni pour développer des plugiciels dans IntelliJ. Il fournit son SDK personnalisé et un ensemble d'actions pour sa création.

Comme première étape, nous devions faire la configuration de l'environnement de développement pour la création d'un plugiciel dans Intellij. La configuration est conformée par les suivantes étapes [27]:

1. Ajouter dans la configuration de la plateforme le SDKs
2. Ajouter dans la configuration de la plateforme le Plugin SDK

Une fois que la configuration est faite, au moment de la création, choisir le projet de type *IDE Plugin*, la structure de notre projet est la suivante:

```
.IntelliJIDEAx0/
└── plugins
    └── sample.jar
        ├── com/foo/...
        │   ...
        └── META-INF
            ├── plugin.xml
            ├── pluginIcon.svg
            └── pluginIcon_dark.svg
```

Fig 3.7 Structure de projet d'un plugiciel sans dépendances [28]

Intellij IDEA fournit le concept d'action, une action est une classe dérivée de la classe abstraite *AnAction*, dont la méthode *actionPerformed()* est appelée lorsque son élément de menu ou son bouton de barre d'outils est cliqué. La classe action doit être configurée dans le fichier *plugin.xml* pour être appelé [29].

Notre plugiciel personnalisé créé en suite la matrice dans le format csv, la classe *CheckCodeInvocationVisitor()* qui étend la classe *JavaRecursiveElementVisitor()*, est utilisée avec les deux méthodes: *visitMethodCallExpression()* et *visitMethod()*.

La méthode *visitMethod()* est utilisée pour le calcul de la métrique *IsInheritanceUsed (IIU)*, dans cette méthode, l'algorithme parcourt toutes les méthodes héritées de la classe, pour chaque méthode héritée qui contient une classe *Ci* avec une classe *Cj* la valeur de la métrique va être 1.

La méthode *visitMethodCallExpression()* est utilisée pour le calcul de la métrique *IsClassCalled (ICC)*, dans cette méthode, l'algorithme parcourt toutes les méthodes qu'appelle la classe, pour chaque méthode *Cj* appelée par une classe *Ci* la valeur de la métrique va être 1.

La classe *Exporter* est utilisée dans le processus de génération du fichier csv. Avec l'installation de notre plugiciel, l'option *MatrixMetrics* apparaîtra dans la barre de tâche, avec une sous-option *Generate CSV File* pour la création de la représentation matricielle, à la fin du processus, une fenêtre avec un message de création sera affichée, ce message contiendra un lien pour télécharger le fichier.

Fig 3.8 Plugiciel personnalisé installé dans Intellij IDEA

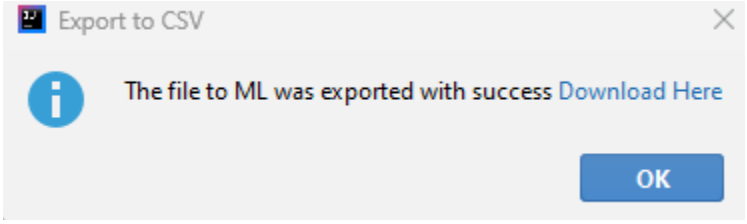


Fig 3.9 Fenêtre avec le message de création

Le fichier csv contiendra une représentation matricielle des lignes et colonnes avec les classes de l'application, l'intersection entre les classes, contiendra les deux valeurs binaires (*IIU, ICC*) que sont les deux métriques déjà mentionnées.

	GridArrangement	SparseGradient	EmpiricalDistribution	SmoothedDataHistogram	FileUtils
GridArrangement	0,0	0,0	0,0	0,0	0,0
SparseGradient	0,0	0,0	0,0	0,0	0,0
EmpiricalDistribution	0,0	0,0	0,0	0,0	0,0
SmoothedDataHistogram	0,0	0,0	0,0	0,0	0,0
FileUtils	0,0	0,0	0,0	0,0	0,1

Tableau 3.3 Exemple d'un fichier csv créé

Les données que nous allons utiliser pour la classification dans le modèle de prédiction ont été prises par les métriques *JUC-Tested* et *Tested*.

Resource	JUC	JUC-TESTED	TEST-CLASS	Tested	DTTested
org.apache.commons.io.ByteOrderMark	100	1	2	1	1
org.apache.commons.io.comparator.AbstractFileComparator	100	1	1	0	1
org.apache.commons.io.comparator.CompositeFileComparator	50	1	2	1	1
org.apache.commons.io.comparator.DirectoryFileComparator	0	0	2	1	1
org.apache.commons.io.comparator.ExtensionFileComparator	25	1	2	1	1

Tableau 3.4 Extrait du fichier csv utilisé pour l'étiquette de classification

3.6 Métriques de logiciel

Les métriques logicielles sont utilisées comme un instrument de mesure et de contrôle dans le processus de développement et de maintenance des logiciels [30].

Il existe plusieurs métriques logicielles proposées dans la littérature. Chidamber et Kemerer [11] ont proposé une suite de métriques orientées objet basées sur le code source largement utilisées dans les études en assurance qualité des logiciels. Cependant, ces métriques sont de types statiques, les métriques statiques sont capables de quantifier divers aspects de la complexité de la conception ou du code source d'un système logiciel, mais leur capacité à capturer certains comportements dynamiques d'une application est encore non prouvée, les fonctionnalités orientées objet comme le polymorphisme, la liaison dynamique, l'héritage et la présence courante de code non utilisé (code mort) dans les logiciels, rendent les métriques statiques imprécises, car ne reflétant pas précisément la situation d'exécution du logiciel [5].

Les métriques dynamiques sont la classe de métriques logicielles qui capturent le comportement dynamique des systèmes logiciels et sont généralement obtenues à partir de l'exécution du code ou des modèles exécutables [5].

Au cours du développement d'un logiciel, la qualité doit être prise en compte dès le début, un processus de qualité bien pensé va permettre d'obtenir un produit de qualité. La gestion de qualité est proposée comme un ensemble d'activités coordonnées pour orienter et contrôler un processus en matière de qualité et d'objectifs [31], pour atteindre cet objectif, le processus peut s'appuyer sur les métriques logicielles. Il existe différentes métriques pour les différentes approches de programmation, dans le contexte de la programmation orientée objet, des familles de métriques liées à la taille, à la complexité, à l'héritage, au couplage existent.

Les métriques dynamiques peuvent être classées en deux catégories: les métriques de contrôle et les métriques prédictives [31]. Les métriques prédictives sont normalement associées au produit logiciel. Avec l'aide des métriques prédictives, on est en mesure de déterminer les caractéristiques statiques et dynamiques du logiciel. Les métriques prédictives sont associées au logiciel lui-même et sont parfois appelées "métriques de produit" [31].

Les métriques de produit sont des métriques prédictives utilisées pour mesurer les attributs internes d'un système logiciel [31]. Parmi ces métriques nous pouvons trouver la taille du système, le nombre de lignes de code, les nombre de méthodes de chaque classe, le couplage entre classes, etc.

Une des métriques à considérer est le couplage entre objets CBO, le couplage est le degré de collaboration formelle entre les modules d'un logiciel, il pourrait être pris comme une mesure du degré de connexion entre deux modules [11].

Une valeur élevée pour cette métrique signifie que les classes ou modules sont très dépendants et qu'il est donc plus probable que la modification d'une classe affecte les autres classes d'un logiciel.

Les métriques de couplage servent dans différentes étapes du cycle de vie du logiciel, dont la majorité est évaluée par une analyse de code statique [32], par contre, comme nous avons remarqué, ces mesures statiques ne saisissent que certaines dimensions sous-jacentes du couplage, d'autres dépendances concernant au comportement dynamique d'un programme ne peuvent être déduites qu'à

partir des informations d'exécution, c'est-à-dire la qualité d'un produit logiciel va être influencé par son environnement opérationnel ainsi que par la complexité du code source.

3.6.1 Métriques statiques

Principalement ces métriques ont été définies à l'origine pour les programmes procéduraux et incorporées plus tard pour les systèmes orientés objet [5]. Les métriques statiques sont capables de quantifier divers aspects de la complexité de la conception ou du code source d'un système logiciel, mais leur capacité à prédire avec précision le comportement dynamique d'une application n'a pas encore été prouvée [5], le comportement dynamique d'un système logiciel doit être analysé par l'environnement d'exécution et par le code source pour avoir une bonne prédiction de l'état d'un système.

Métrique de couplage

CBO (Coupling Between Object) se rapporte à la notion selon laquelle un objet est couplé à un autre objet si l'un d'eux agit sur l'autre, c'est-à-dire des méthodes d'un qui utilise les méthodes ou les variables d'instance d'un autre. Comme indiqué précédemment, puisque les objets de la même classe ont les mêmes propriétés, deux classes sont couplées lorsque des méthodes déclarées dans une classe utilisent des méthodes ou des variables d'instance définies par l'autre classe [11].

3.6.2 Métriques dynamiques

Les métriques dynamiques sont collectées sur le programme en exécution. Ces métriques peuvent être collectées pendant les tests du système ou après la mise en service du système. Exemples: Le nombre de bogues, le temps nécessaire pour effectuer un calcul, etc. [31]

En général, les métriques dynamiques sont la classe des métriques logicielles qui capturent le comportement dynamique d'un système logiciel et sont généralement obtenus à partir des traces d'exécution du code ou des modèles exécutables [5].

Mitchell1 A. et Power J. F., ont décrit un ensemble de métriques de couplage orienté objet obtenues à l'exécution, dans leur étude, ils indiquent que les métriques de couplage d'exécution peuvent fournir une analyse qualitative intéressante et informative sur un logiciel et compléter les métriques de couplage statique existantes [33].

3.7 Méthodes d'évaluation de performance

3.7.1 Matrice de confusion

Une matrice de confusion est un tableau qui présente les résultats de la classification, elle donne le nombre de vrais/faux, positif/négatif. Dans notre étude un positif représente une classe à considérer pour le processus de test et un négatif représente une classe qui ne doit pas être considérée.

Classifié	Actuel	
	Classe Testé	Classe Non Testé
Classe Testé	Vrais positifs (VP)	Faux positifs (FP)
Classe Non Testé	Faux négatifs (FN)	Vrais négatifs (VN)

Tableau 3.5 Matrice de confusion

3.7.2 Exactitude

Cette métrique détermine le nombre de prédictions correctes sur toutes les prédictions, elle est décrite par la formule :

$$Exactitude = \frac{VP+VN}{FP+VP+FN+VN}$$

3.7.3 Précision

La précision détermine la qualité de la classification des prédictions vraies faites par le modèle, elle est décrite par la formule :

$$Précision = \frac{VP}{VP + FP}$$

3.7.4 Rappel ou Sensibilité

Le rappel décrit l'exactitude de la prédiction des classes positives dans l'ensemble des prédictions pertinentes. Différemment de la précision, il prend en considération les faux négatifs, elle est décrit par la formule :

$$Rappel\ ou\ Sensibilité = \frac{VP}{VP + FN}$$

3.7.5 Spécificité

La spécificité décrit l'exactitude de la prédiction des classes négatives faites par le modèle, elle est décrit par la formule :

$$Spécificité = \frac{VN}{VN + FP}$$

3.7.6 F-Measure

Le F-Measure est une métrique qui traduit l'équilibre entre la précision et la sensibilité. La mesure est 0 lorsque la précision ou la sensibilité est de 0. Cette mesure fonctionne cependant bien lorsque les données sont assez équilibrées, c'est-à-dire qu'il y a une quantité égale des données entre les classificateurs. La formule de cette mesure est la suivante [25] :

$$F - Mesure = \frac{2 * sensibilité * précision}{sensibilité + précision}$$

3.7.7 Moyenne Géométrique

La moyenne géométrique est une métrique qui mesure l'équilibre entre la performance de la classification dans les classes majoritaires et minoritaires. Une valeur basse est une indication d'une mauvaise performance dans la classification des cas positifs même si les cas négatifs sont correctement classés [25].

Les formules pour le calcule de la moyenne géométrique sont les suivantes [25] :

$$moyenne\ géométrique = \sqrt{Sensibilité * Spécificité}$$

Dans notre étude, nous avons considéré les niveaux suivants pour décrire les valeurs de la moyenne géométrique [34].

- *moyenne géométrique* < 0.5 signifie mauvaise classification;
- $0.5 \leq$ *moyenne géométrique* < 0.6 signifie faible classification;
- $0.6 \leq$ *moyenne géométrique* < 0.7 signifie classification acceptable;
- $0.7 \leq$ *moyenne géométrique* < 0.8 signifie bonne classification;
- $0.8 \leq$ *moyenne géométrique* < 0.9 signifie très bonne classification;
- *moyenne géométrique* ≥ 0.9 signifie excellente classification.

Dans le prochain chapitre, nous discuterons du processus d'expérimentation utilisé. Au début, nous expliquerons de façon générale l'apprentissage automatique et le réseau de neurones à convolution pour la création d' un modèle prédictif général qui permettra d'utiliser notre matrice relationnelle pour la prédiction des classes candidates dans le processus de test, finalement nous répondrons aux questions de recherche posées.

Chapitre 4

Expérimentations

4.1 Apprentissage automatique

L’apprentissage automatique pourrait être étudié dans le domaine de la computation à partir des programmes informatiques, on dit qu'un programme informatique apprend de l'expérience E par rapport à une classe de tâches T et à une mesure de performance P, si sa performance aux tâches dans T, mesurée par P, s'améliore avec l'expérience E [35].

4.2 Réseaux de neurones à convolution (RNC)

Les réseaux de neurones à convolution sont utilisés principalement dans le traitement de données de types images, puisque chaque image est une matrice à deux dimensions de pixels. Ces réseaux ont été inspirés par les processus biologiques où le modèle de connectivité entre les neurones ressemble au cortex visuel d’un animal.

4.2.1 Architectures des RNC

La structure typique des RNC est composé de:

- *Couches de convolutions* : Les couches convolutives sont utilisées pour extraire les différentes caractéristiques de l'entrée.
- *Couches de pooling* : L’objectif principal de ces couches est de réduire la taille des paramètres des couches convolutives afin de réduire les coûts de calcul.
- *Couches entièrement connectées* : Couche dans laquelle tous les nœuds contenus se connectent à tous les nœuds de la couche suivante.

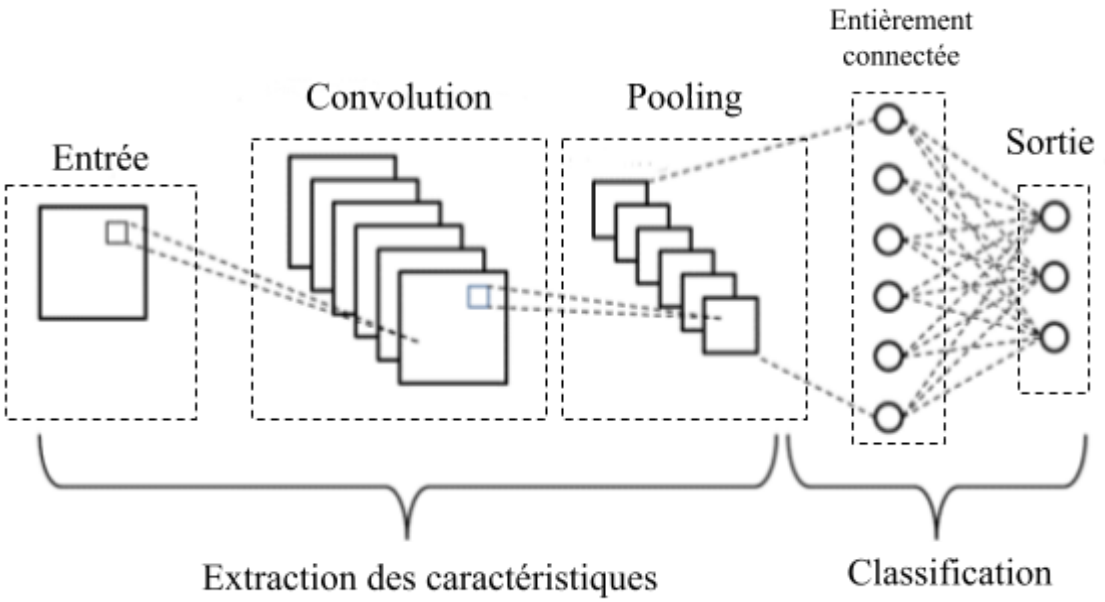


Fig 4.1 Architecture basique des RNC [36]

4.2.1.1 Dropout

Lorsque la couche de convolution de sortie est connectée à la couche pooling, cela peut occasionner un réajustement dans l'ensemble de données d'entraînement, ce qui a un impact négatif sur les performances du modèle lorsqu'il est utilisé sur de nouvelles données.

Pour surmonter ce problème, une couche de suppression aléatoire est utilisée dans laquelle quelques neurones sont supprimés du réseau neuronal pendant le processus de formation, ce qui réduit la taille du modèle: Il s'agit du dropout.

4.2.1.2 Les fonctions d'activations

Ces fonctions permettent essentiellement la décision au niveau de chaque neurone, de transmettre ou non le signal d'entrée aux couches suivantes. Il s'agit de fonctions mathématiques régulières qui déterminent la sortie, selon un seuil ou une règle.

4.2.1.3 La fonction de perte (LOSS)

La fonction de perte ou LOSS permet de quantifier la différence entre le résultat attendu et le résultat produit par le modèle d'apprentissage. Parmi les différentes fonctions de perte, il y a deux problématiques de classification multi-classe à mentionner:

- *Entropie croisée catégorielle (Categorical Cross-Entropy) [37]:* Elle est utilisée dans les scénarios de classification multiclasse, sa formule est la suivante :

$$H(p,q) = - \sum_{x \in classes} p(x) \, log \, q(x)$$

Où, p(x) est la vraie distribution de probabilité (one-hot) et q(x) est la distribution de probabilité prédite.

- *Entropie croisée catégorielle clairsemée (Sparse Categorical Cross-Entropy) [37]:* Elle a la même fonction de perte que l'entropie croisée catégorielle, la différence principale est la sortie attendue, si la sortie a le même format que l'entrée, la fonction de perte qui est utilisée est l'entropie croisée catégorielle. Si les sorties sont encodées dans un format entier, la recommandation est d'utiliser la fonction de perte d'entropie croisée catégorielle clairsemée.

4.3 Création du modèle prédictif

Pour la création du modèle, nous avons utilisé le langage de programmation python et l'outil Google Collab. Tous les processus et sous-processus ont été expliqués de façon générale dans le chapitre 3.

4.3.1 Source des données

Nous devions ouvrir chaque système logiciel dans Intellij et utiliser le plugiciel que nous avons développé pour générer la représentation matricielle associée.

4.3.2 Préparation des données

Pour chaque système logiciel, nous avons une représentation matricielle et nous avons les classes avec leurs étiquettes de classification stockées séparément.

Le processus de liaison des données est fait par un script, le résultat va être un ensemble des données avec la représentation matricielle étiquetée. Ensuite, pour chaque système logiciel nous avons calculé si les données sont équilibrés, à partir de la colonne qui contient les étiquettes (JUC-Tested et Tested), nous avons compté les numéros de lignes avec valeur 0 (Classes non candidates) et avec valeur 1 (Classes candidates) et nous avons calculé les statistiques descriptives.

	JUC-TESTED	**TESTED**
1	78	65
0	23	36

Tableau 4.1 Description de la quantité de données pour le système IO

	JUC-TESTED	TESTED
count	101	101
mean	0.772277	0.643564
std	0.421454	0.481335
25 %	1	0
50 %	1	1
75 %	1	1
min	0	0
max	1	1

Tableau 4.2 Statistiques descriptives des étiquettes de classification pour le système IO

Pour ces résultats, nous avons décidé d'équilibrer les données avec l'utilisation de la méthode SMOTE.

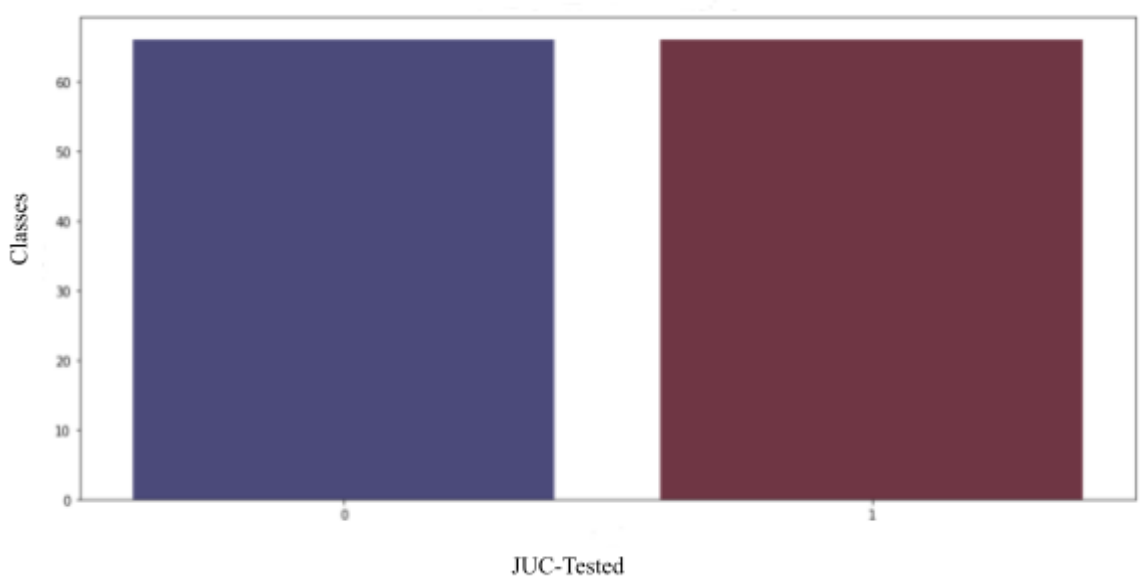


Fig 4.2 Données équilibrées pour l'étiquette JUC-TESTED pour le système IO

La séparation des données est faite avec soixante dix pourcent (70%) pour la validation et trente pourcent (30%) pour le test.

4.3.3 Développement du modèle d'apprentissage

Pour le développement du modèle, nous avons fait notre propre architecture, elle est illustrée dans la figure 4.3.

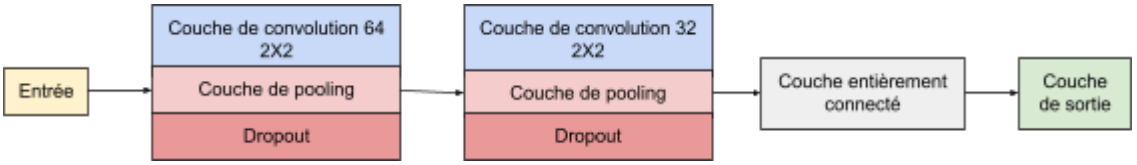


Fig 4.3 Diagramme du modèle d'apprentissage profond utilisé

L'entrée du modèle a été traitée pour avoir dans chaque classe l'ensemble des classes qui contiendrait la paire des valeurs binaires qui représente des métriques logicielles *IIU* et *ICC*, nous pouvons expliquer la matrice relationnelle avec la représentation mathématique suivant :

Si $E_{x,y} = P_{x,y}(IIU, ICC)$ et $M_{1,y} = [[E_{1,1}], [E_{1,2}], \ldots\ldots [E_{1,y}]]$ alors,

$M_{x,y} = \{[[E_{1,1}], [E_{1,2}], \ldots\ldots [E_{1,y}]], [[E_{2,1}], [E_{2,2}], \ldots\ldots [E_{2,y}]], \ldots\ldots\ldots [[E_{x,1}], [E_{x,2}], \ldots\ldots [E_{x,y}]]\}$

Où:

$E_{x,y}$: Élément dans la matrice relationnelle qui représente le paire *IIU* et *ICC* entre les classes C_x et C_y

$M_{1,y}$: Liste composé par les paires *IIU* et *ICC* entre la classe C_1 et les autres classes (C_1, C_2, C_3,...,C_y)

$M_{x,y}$: Matrice relationnelle composée par les paires *IIU* et *ICC* entre toutes les classes d'un logiciel

Pour une représentation graphique de $M_{1,y}$, nous allons illustrer la liste de la façon suivante: la valeur 0 (couleur jaune), la valeur 1 (couleur violette), le premier carré est IIU et le deuxième carre est ICC, $M_{1,y}$ sera :

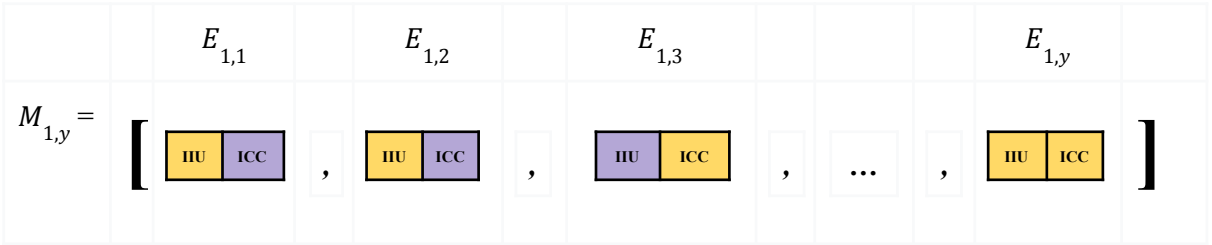


Fig 4.4 Liste composée par les paire IIU et ICC pour la classe C_1

4.3.4 Modèle entraînée

L'optimiseur utilisé est *Adam* avec un taux d'apprentissage de 0.001, la fonction de perte est *sparse_categorical_crossentropy*. Le nombre de passages (epochs) utilisés est de 20.

4.3.5 Evaluation de performance

L'évaluation de performance a été faite avec l'utilisation des modules suivants:

- `imblearn.metrics`, pour le calcul de la valeur de la spécificité.
- `sklearn.metrics`, pour le calcul de la matrice de confusion et des valeurs de la précision, la sensibilité, la F1 score et moyenne géométrique.

4.4 Rappel des questions de recherche

Les résultats obtenus dans ces expérimentations nous permettent de répondre à la question de recherche:

QR1: Comment pouvons-nous créer un plugiciel qui permettra de proposer des classes candidates pour le test à partir d'une analyse des classes d'un système logiciel basé sur ses relations d'héritage et des couplages afin de supporter les testeurs dans la phase des tests unitaires ?

Le but poursuivi étant de créer un plugiciel qui permettra de créer une matrice relationnelle avec les relations d'héritage et couplage entre les classes d'un système logiciel est faisable, il y a quelques points à considérer avant de commencer, le plus importants dans cette étude ont était la documentation de l'IDE et la performance du code pour éviter que l'IDE ne réponde pas en ce moment d'utilisation du plugiciel avec systèmes logiciels avec des milliers de classes.

QR2: Pourrait-on avoir une meilleure suggestion utilisant cette représentation matricielle et l'apprentissage profond dans le choix des classes candidates durant la phase des tests unitaires ?

Le but poursuivi étant d'avoir une meilleure prédiction avec deux métriques ou variables est satisfaisant.

Dans le prochain chapitre, nous discutons des résultats et de l' interprétation basée sur les systèmes logiciels utilisés et sur les deux métriques pour la classification des classes candidates à tester. Finalement, nous discutons des menaces à la validation pour donner une idée générale des scénarios des menaces en référence aux différents systèmes logiciels, le langage de programmation et le modèle prédictif.

Chapitre 5

Résultats et interprétation

5.1 Résultats

Les résultats obtenus, en tenant compte du fait que nous avons pris trente pour cent (30%) des données de test et nous avons fait exécuter le modèle pour chaque métrique *JUC-Tested* et *Tested* selon chaque système logiciel, la précision du modèle de prédiction varie entre 74.3% et 93.4% et le modèle fonctionne mieux avec le système logiciel *MATH*.

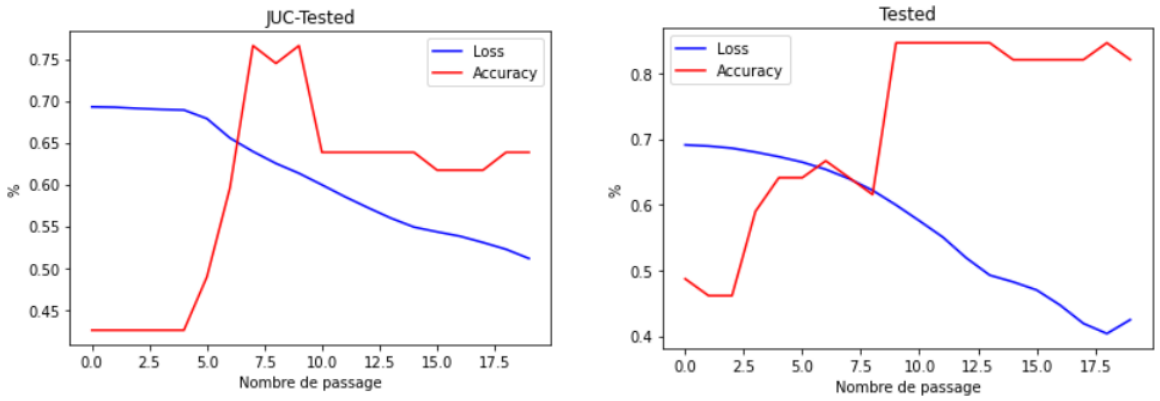
En prenant l'exactitude comme métrique à évaluer, le système logiciel *MATH* prend le meilleur résultat avec l'étiquette *JUC-Tested*, pour l'étiquette *Tested*, le système logiciel *IO* a le meilleur résultat.

Selon la métrique moyenne géométrique, le système logiciel *MATH* prend le meilleur résultat avec l'étiquette *JUC-Tested*, pour l'étiquette *Tested*, le système logiciel *IO* a le meilleur résultat.

	Métriques d'évaluation	**IO**	**MATH**	**JFREECHART**
JUC-Tested	**Exactitude**	0.6382	0.875	0.69
	Précision	0.917	1	0.75
	Sensibilité	0.407	0.783	0.584
	Spécificité	0.950	1	0.8
	Moyenne géométrique	0.622	0.885	0.684
	F1-Score	0.564	0.878	0.656
	Précision du modèle de prédiction	0.743	0.934	0.847
Tested	**Exactitude**	0.8205	0.612	0.64
	Précision	0.792	0.714	0.75
	Sensibilité	0.905	0.556	0.549
	Spécificité	0.722	0.692	0.759
	Moyenne géométrique	0.808	0.620	0.646
	F1-Score	0.8444	0.625	0.634
	Précision du modèle de prédiction	0.857	0.887	0.833

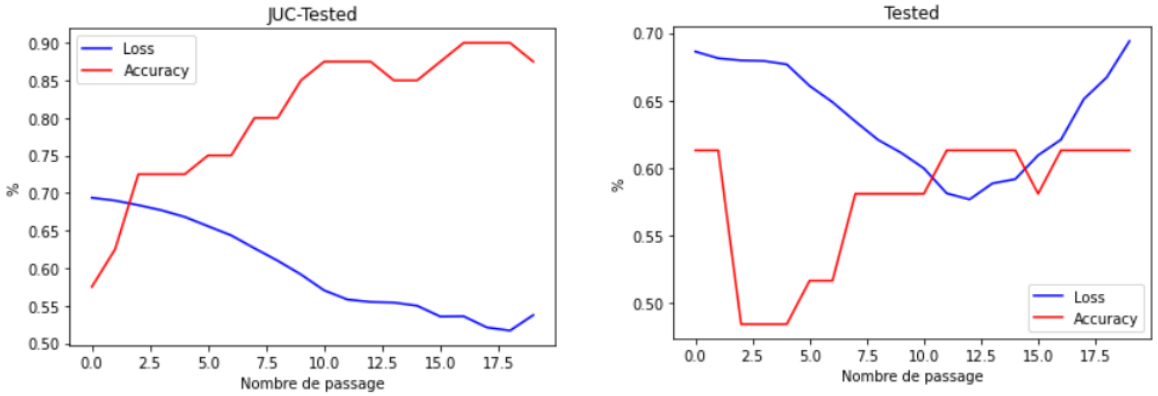
Tableau 5.1 Résultats de performance du modèle

Les résultats obtenus avec la librairie *IO,* par rapport au nombre de passages (20) et avec un 30% de données de test utilisé, montrent 63.83% de précision, 51.12% de perte, une erreur de 36.17% et une moyenne géométrique de 62.2% pour la métrique *JUC-Tested*, 82.05% de précision, 42.5% de perte, une erreur de 17.95% et une moyenne géométrique de 80.8% pour la métrique *Tested.* Dans le système logiciel *IO,* la métrique *Tested* pourrait être prise comme la métrique mieux classifiée pour être utilisée dans le modèle.



Graphique 5.1 Perte et Précision entre les étiquettes avec le système IO

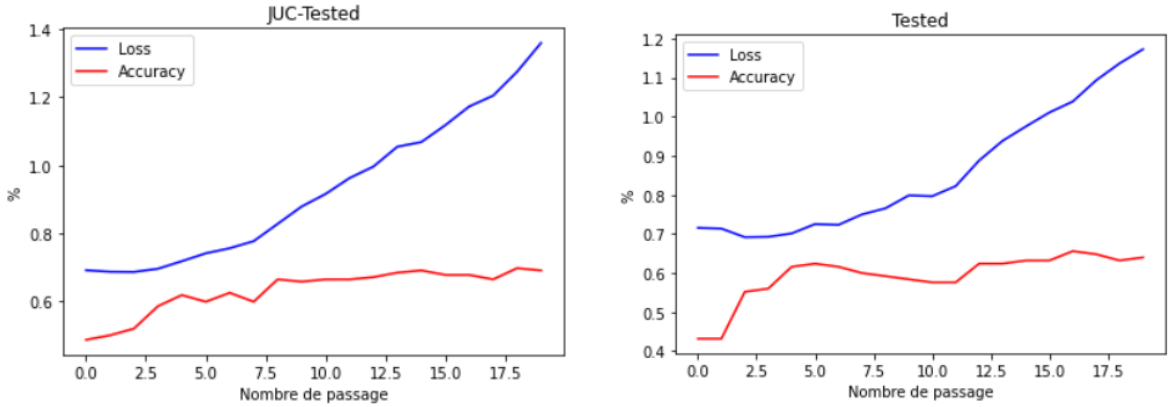
Les résultats obtenus avec la librairie *MATH,* par rapport au nombre de passages (20) et avec un 30% de données de test utilisé, montrent 87.5% de précision, 53.74% de perte, une erreur de 12.5% et une moyenne géométrique de 88.5% pour la métrique *JUC-Tested*, 61.29% de précision, 69.4% de perte, une erreur de 38.71% et une moyenne géométrique de 62% pour la métrique *Tested.*



Graphique 5.2 Perte et Précision entre les étiquettes avec le système MATH

Les résultats obtenus avec la librairie *JFREECHART,* par rapport au nombre de passages (20) et avec un 30% de données de test utilisé, montrent 69.08% de précision, 135.98% de perte, une erreur de 30.92% et une moyenne géométrique de 68.4% pour la métrique *JUC-Tested*, 64% de précision, 117.27% de perte, une erreur de 36% et une moyenne géométrique de 64.6% pour la métrique *Tested.*

Dans le système logiciel *JFREECHART,* la métrique *JUC-Tested* pourrait être prise comme la métrique mieux classifiée pour être utilisée dans le modèle.



Graphique 5.3 Perte et Précision entre les étiquettes avec le système JFREECHART

5.2 Discussion

Nous pouvons observer que les résultats sont variés, selon les systèmes logiciels et les étiquettes de test des logiciels utilisés pour la classification des classes (*JUC-Tested* et *Tested*), les possibilités d'optimisation sont grandes, cependant nous pouvons améliorer le modèle de différentes façons, soit par l'augmentation des métriques dans la représentation matricielle, c'est-à-dire, nous avons utilisé deux métriques, par contre si nous ajouterons plus métriques avec plus information sur le système logiciel ça pourrait améliorer le résultat, soit par l'optimisation des variables du modèle prédictif.

Les résultats obtenus sont satisfaisants avec les contraintes que nous avions, il est important de mentionner dans notre étude que les étapes de prétraitement, d'acquisition des données, réajustements et balancement des données sont essentielles pour obtenir une meilleure performance.

5.3 Menaces à la validité

Cette étude, au regard de bien d'autres études en génie logiciel, est confrontée à des menaces qui peuvent biaiser la validité de certains résultats.

Premièrement, cette étude couvre un ensemble de 3 différents systèmes logiciels, cela sous-entend que les conclusions liées à ces différents systèmes ne peuvent être généralisées sur l'ensemble d'autres systèmes qui n'ont pas fait l'objet de notre étude. Ce genre d'études doit être reproduit sur un grand nombre de systèmes pour pouvoir tirer des conclusions générales.

Deuxièmement, nous pouvons mentionner le langage de programmation Java est utilisé pour les systèmes logiciels, ce qui signifie que les caractéristiques d'extraction inhérentes aux métriques de logiciels de ces systèmes peuvent différer des autres langages de programmation, le langage Java est

un bon représentant des langages de programmation orientés objet, mais ces caractéristiques pourraient impacter sur la possibilité de généraliser nos conclusions à d'autres systèmes écrits dans des langages autres que Java.

Finalement, la dernière menace à laquelle on fait face concerne l'utilisation d'un modèle d'apprentissage profond sans vérifier d'autres algorithmes d'apprentissage automatique où nous pouvons augmenter la précision de la prédiction. Ceci peut donc affecter l'évaluation des performances de ces dernières.

Chapitre 6

Conclusion générale

6.1 Rappel de la problématique

Dans ce mémoire, nous avons étudié la problématique de la priorisation des classes candidates pour être inclus dans le processus de test. L'objectif a été abordé pour proposer un plugiciel pour l'environnement Intellij avec le but d'identifier des classes candidates au tester. Notre point de départ a été de créer une matrice relationnelle à partir d'un système logiciel et des métriques logicielles liées à l'héritage et au couplage. L'étude a été réalisée sur 3 systèmes logiciels, à partir de la matrice relationnelle, nous allons lier avec une table contenant l'information des mêmes classes, mais avec information des métriques comme *JUC-Tested* et *Tested*, ces métriques nous les utilisons comme étiquettes de classification pour créer le modèle prédictif.

Notre modèle prédictif est composé de différentes étapes. Dans le processus de préparation le résultat a été équilibré avec l'utilisation de la méthode SMOTE.

Pour le modèle d'apprentissage profond, nous avons utilisé une structure typique pour RNC, une couche de convolution, une couche de pooling et une couche entièrement connectée, cette structure typique a été utilisée deux fois dans notre modèle.

Dans le processus d'évaluation de performance, nous avons utilisé les métriques d'évaluation telles que l'exactitude, la précision, la sensibilité, la spécificité, et la moyenne géométrique.

6.2 Contribution

Durant cette étude, nous avons exploré les relations entre les métriques logicielles utilisées et les caractéristiques d'une classe pour être pris dans le processus de test unitaire. Ces études, menées sur trois systèmes logiciels open source, ont permis de montrer la possibilité de prédire les classes candidates à tester à partir de l'utilisation de la matrice relationnelle utilisée par un modèle d'apprentissage profond.

Cette recherche nous a permis de mettre en évidence la possibilité: (1) d'entraîner un réseau de neurones profond sur des données obtenues à partir d'une matrice relationnelle de relation des métriques de logiciel pouvant prédire les classes candidates à tester pour un système en cours de développement, (2) de créer et proposer un plugiciel comme outil pour aider dans le processus de développement et (3) d'avoir de résultat satisfaisant dans la prédiction à partir de l'utilisation de la matrice relationnelle.

6.3 Futures recherches

Ces résultats entrouvrent les possibilités de proposer des créations d'outils pour aider, à partir de l'utilisation des métriques de logiciel, à améliorer l'expérience de développement dans l'orientation de l'effort global des tests unitaires. En général, nous voulons proposer le développement des outils basés sur le nuage et les techniques d'analyse de métadonnées et des données (impliquant des algorithmes d'intelligence artificielle) pour bâtir une nouvelle génération d'outils de priorisation et d'orientation des tests qui pourraient être intégrés aux environnements de développement.

Bibliographie

[1] T. J. McCabe, “A Complexity Measure,” IEEE Transactions on Software Engineering, 1976.

[2] N. V. Chawla, K. W. Bowyer, L. O. Hall and W. P. Kegelmeyer, “Smote: synthetic minority over-sampling technique,” Journal of artificial intelligence research, 2002.

[3] N. E. Fenton and N. Ohlsson, “Quantitative analysis of faults and failures in a complex software system," IEEE Transactions on Software Engineering, Vol. 26, No. 8, pp. 797-814, 2000.

[4] A. G. Koru and H. Liu, "Building effective defect-prediction models in practice," IEEE Software, Vol. 22, No. 6, pp. 23-29, 2005.

[5] J. K. Chhabra and V. Gupta, “A Survey of Dynamic Software Metrics,” Journal of Computer Science and Technology, 2010.

[6] F. Toure, M. Badri and L. Lamontagne, “Investigating the Prioritization of Unit Testing Effort using Software Metrics,” Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering, 2017.

[7] F. Toure, M. Badri and L. Lamontagne, “Predicting different levels of the unit testing effort of classes using source code metrics: a multiple case study on open-source software,” Innovations in Systems and Software Engineering, 2018.

[8] F. Toure and M. Badri, “Empirical analysis of object-oriented design metrics for predicting unit testing effort of classes,” Journal of Software Engineering and Applications, 2012.

[9] A. Boucher and M. Badri, "An unsupervised fault-proneness prediction model using multiple risk levels for object-oriented software systems: An empirical study," Mémoire présentée à l'Université du Québec à troıs-rıvıères, 2018.

[10] Y. Singh, A. Kaur and R. Malhotra, “Predicting testing effort using artificial neural networks,” Lecture Notes in Engineering and Computer Science, 2008.

[11] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on Software Engineering, Vol. 20, No. 6, pp. 476-493, 1994.

[12] F. Toure and M. Badri, “Prioritizing Unit Testing Effort Using Software Metrics and Machine Learning Classifiers,” Proceedings of the 30th International Conference on Software Engineering and Knowledge Engineering, 2018.

[13] W. Matcha, F. Toure, M. Badri and L. Badri, “Using Deep Learning Classifiers to Identify Candidate Classes for Unit Testing in Object-Oriented Systems,” International Conference on Software Engineering and Knowledge Engineering, 2020.

[14] F. Toure and M. Badri, “Unit Test Effort Prioritization Using Combined Datasets and Deep Learning: A Cross-Systems Validation,” The 32nd International Conference on Software Engineering and Knowledge Engineering, 2020.

[15] “JUnit Framework”, https://junit.org/junit5/, Visité en juillet 2021.

[16] “Apache IO”, http://commons.apache.org/io/, Visité en juillet 2021.

[17] “Apache Software Foundation”, https://www.apache.org/ , Visité en juillet 2021.

[18] “JFreeChart”, http://www.jfree.org/jfreechart/, Visité en juillet 2021

[19] “Commons Math”, http://commons.apache.org/proper/commons-math/, Visité en juillet 2021.

[20] L. Badri, M. Badri and F. Toure, “Exploring Empirically the Relationship between Lack of Cohesion and Testability in Object Oriented Systems,” Advances in Software Engineering, Communications in Computer and Information Science, Vol. 117, Springer, Berlin, 2010.

[21] M. Bruntink and A.V. Deursen, “Predicting Class Testability using Object-Oriented Metrics,” 4th International Workshop on Source Code Analysis and Manipulation (SCAM), IEEE, 2004.

[22] A. Mockus, N. Nagappan and T. T. Dinh-Trong, “Test coverage and post-verification defects: a multiple case study,” Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 291– 301, 2009.

[23] “Metrics Reloaded”, https://plugins.jetbrains.com/plugin/93-metricsreloaded , Visité en juillet 2021.

[24] F. Toure and M. Badri, “Unit Testing Effort Prioritization Using Combined Datasets and Deep Learning: A Cross-Systems Validation,” The 32nd International Conference on Software Engineering and Knowledge Engineering, 2020.

[25] J. S. Akosa, “Predictive Accuracy: A Misleading Performance Measure for Highly Imbalanced Data,” Proceedings of the SAS Global Forum Conference, 2017.

[26] “Using DevKit”, https://plugins.jetbrains.com/docs/intellij/using-dev-kit.html, Visité en août 2021.

[27] “Setting Up a Development Environment”, https://plugins.jetbrains.com/docs/intellij/setting-up-environment.html#configuring-intellij-platform-sdk, Visité en août 2021.

[28] “Plugin Content”, https://plugins.jetbrains.com/docs/intellij/plugin-content.html#plugin-without-dependencies, Visité en août 2021.

[29] “Plugin Action”, https://plugins.jetbrains.com/docs/intellij/plugin-actions.html, Visité en août 2021.

[30] L. C. Santos, R. Saraiva, M. Perkusich, H. O. Almeida and A. Perkusich, “An empirical study on the influence of context in computing thresholds for Chidamber and Kemerer metrics,” The 29th International Conference on Software Engineering and Knowledge Engineering, 2017.

[31] I. Sommerville, “Software engineering,” 9th edition, ISBN-10, 137035152, 2011.

[32] L. C. Briand, J. W. Daly and J. K. W¨ust, “A Unified Framework for Coupling Measurement in Object-Oriented Systems,” IEEE Transactions on Software Eng., Vol. 25, No. 1 pp. 91–121, 1999.

[33] A. Mitchell and J. Power, “An empirical investigation into the dimensions of run-time coupling in Java programs,” Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java, 2004.

[34] A. Boucher and M. Badri, “Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison,” Information and Software Technology, 2018.

[35] T. M. Mitchell, “Machine learning,” Vol. 1, McGraw-hill New York, 1997.

[36] “Basic CNN Architecture”, https://www.upgrad.com/blog/basic-cnn-architecture/, Visité en avril 2022.

[37] “An Introduction to Neural Network Loss Functions”, https://programmathically.com/an-introduction-to-neural-network-loss-functions/, Visité en juillet 2021.

[38] “A Look at Precision, Recall, and F1-Score”, https://towardsdatascience.com/a-look-at-precision-recall-and-f1-score-36b5fd0dd3ec, Visité en mai 2022.

Annexe A

Code source du plugiciel pour l'IDE Intellij

MatrixMetricsAction.java

```
package com.idea;

import com.intellij.analysis.AnalysisScope;
import com.intellij.openapi.actionSystem.AnAction;
import com.intellij.openapi.actionSystem.AnActionEvent;
import com.intellij.openapi.actionSystem.PlatformDataKeys;
import com.intellij.openapi.progress.EmptyProgressIndicator;
import com.intellij.openapi.progress.ProgressManager;
import com.intellij.openapi.project.Project;
import com.intellij.openapi.roots.ProjectRootManager;
import com.intellij.openapi.ui.Messages;
import com.intellij.openapi.vfs.VirtualFile;
import com.intellij.psi.PsiElement;
import com.intellij.psi.PsiFile;
import com.intellij.psi.PsiManager;
import com.intellij.util.ResourceUtil;
import org.jetbrains.annotations.NotNull;

import java.net.URL;
import java.util.HashSet;
import java.util.Set;

public class MatrixMetricsAction extends AnAction {
   private Set<String> isInheritanceUsed = new HashSet<>();
   private Set<String> isClassCalled = new HashSet<>();
   private Set<String> classes = new HashSet<>();
   private final static String DEFAULT_TEST_EXPRESSION = "Test";

   public MatrixMetricsAction() {
   }

   @Override
   public void actionPerformed(@NotNull AnActionEvent anActionEvent) {
       Project project = anActionEvent.getData(PlatformDataKeys.PROJECT);
       AnalysisScope scope = new AnalysisScope(project);
```

```
        scope.setSearchInLibraries(false);
        generateCSVFile(project, scope);
    }

    private void generateCSVFile(Project project, AnalysisScope scope) {
        final String FILE_CSV = "matrixMetrics.csv";
        final String MENU_ITEM_EXPORT_CSV = "Export to CSV";
        final int EXPORT_TYPE_FILE = 1 /* Column Value: metric1Value, metric2Value
*/;
        final String[] columnNamesMetrics = {"IIU", "ICC"}; /* IIU = Is Inheritance
Used, ICC = Is Class Called */

        VirtualFile[] vFiles =
ProjectRootManager.getInstance(project).getContentSourceRoots();

        for(VirtualFile vFile: vFiles) {
            visitFiles(vFile, project);
        }

        String[] columnNamesClass = new String[classes.size()+1];

        columnNamesClass[0] = "";

        int metricIndex = 0;
        for(String className : classes) {
            columnNamesClass[metricIndex+1] = className;
            metricIndex++;
        }

        String[][] values = new String[classes.size()][classes.size()+1];
        String[] classNames = classes.toArray(new String[classes.size()]);

        String classRowName;
        String classColumnName;
        String isInheritanceUsed;
        String isClassCalled;
        for(int indexRow = 0; indexRow < classNames.length; indexRow++){
            values[indexRow][0] = columnNamesClass[indexRow+1];
            for(int indexColumn = 1; indexColumn < classNames.length+1;
indexColumn++){
                classRowName = classNames[indexRow];
                classColumnName = classNames[indexColumn-1];

                isInheritanceUsed = this.isInheritanceUsed.contains(classColumnName
+":" + classRowName) ? "1": "0";
```

```
                isClassCalled = this.isClassCalled.contains(classColumnName +":" +
classRowName) ? "1": "0";
                values[indexRow][indexColumn] =
                        isInheritanceUsed
                        + "," +
                        isClassCalled
                ;
            }
        }

        try {
            new Exporter(FILE_CSV, values, columnNamesClass, columnNamesMetrics,
EXPORT_TYPE_FILE);
            URL url = ResourceUtil.getResource(getClass(), "files", FILE_CSV);
            String filePath = url.getPath();
            Messages.showInfoMessage("<html>The file to ML was exported with success
<a href=\"" + filePath + "\">Download Here</a></html>", MENU_ITEM_EXPORT_CSV);
        } catch (Exception exception) {
            exception.printStackTrace();
        }
    }

    public void  visitFiles(VirtualFile file, Project project){
        PsiManager psiManager = PsiManager.getInstance(project);
        PsiFile jfile = psiManager.findFile(file);
        if(jfile !=null)
            visitJavaFile(jfile);

        if(file.getChildren() != null)
            for (VirtualFile f : file.getChildren()) {
               visitFiles(f, project);
            }

    }

    private String getSimpleNameOfClass(String allClassName){
        String[] classRowNames;
        classRowNames = allClassName.split("\\.");
        return classRowNames[classRowNames.length - 1];
    }

    private void visitJavaFile(PsiFile file) {
        for(PsiElement psjf: file.getChildren()) {
            if(psjf.toString().split(":").length > 1 &&
file.toString().split(":")[1].contains(".java")) {
```

```
                String className =
file.toString().split(":")[1].replace(".java","");
                if(!classes.contains(className) &&
!className.contains(DEFAULT_TEST_EXPRESSION)){
                    classes.add(className);
                }
                ProgressManager.getInstance().runProcess(() -> {
                    psjf.accept(new CheckCodeInvocationVisitor(className, classes));
                }, new EmptyProgressIndicator());
            }
        }

        if(CheckCodeInvocationVisitor.isInheritanceUsed.size() != 0) {
            this.isInheritanceUsed = CheckCodeInvocationVisitor.isInheritanceUsed;
        }
        if(CheckCodeInvocationVisitor.isClassCalled.size() != 0) {
            this.isClassCalled = CheckCodeInvocationVisitor.isClassCalled;
        }
    }
}
```

CheckCodeInvocationVisitor.java

```
package com.idea;

import com.intellij.psi.JavaRecursiveElementVisitor;
import com.intellij.psi.PsiClass;
import com.intellij.psi.PsiMethod;
import com.intellij.psi.PsiMethodCallExpression;

import java.util.Arrays;
import java.util.Collections;
import java.util.HashSet;
import java.util.Set;

public class CheckCodeInvocationVisitor extends JavaRecursiveElementVisitor {
    private static final Set<String> boilerplateMethods = new HashSet<String>();
    private String className;
    private Set<String> nameOfClasses;
    public static Set<String> isInheritanceUsed = new HashSet<>();
    public static Set<String> isClassCalled = new HashSet<>();

    public CheckCodeInvocationVisitor() {
    }

    public CheckCodeInvocationVisitor(String className, String[] nameOfClasses) {
        this.className = className;
        this.nameOfClasses = new HashSet<>(Arrays.asList(nameOfClasses));
    }
```

```
    public CheckCodeInvocationVisitor(String className, Set<String> nameOfClasses) {
        this.className = className;
        this.nameOfClasses = nameOfClasses;
    }

    static {
        Collections.addAll(boilerplateMethods, "toString", "equals", "hashCode",
"finalize", "clone", "readObject", "writeObject");
    }

    @Override
    public void visitMethodCallExpression(PsiMethodCallExpression expression) {
        String[] classCalled = expression.getText().split("\\.");
        if(classCalled.length > 1 && nameOfClasses.contains(classCalled[0])){
            isClassCalled.add(this.className + ":" + classCalled[0]);
            isClassCalled.add(classCalled[0] + ":" + this.className);
        }
        super.visitMethodCallExpression(expression);
    }

    @Override
    public void visitMethod(PsiMethod method) {
        final PsiMethod[] superMethods = method.findSuperMethods();
        if(superMethods.length != 0){
            for (PsiMethod superMethod : superMethods) {
                if(!superMethod.isConstructor() &&
!boilerplateMethods.contains(superMethod.getName())) {
                    String classParent =
superMethod.getParent().toString().split(":")[1];
                    isInheritanceUsed.add(this.className + ":" + classParent);
                    isInheritanceUsed.add(classParent + ":" + this.className);
                }
            }
        }

        super.visitMethod(method);
    }

    @Override
    public void visitClass(PsiClass aClass) {
        super.visitClass(aClass);
    }

}
```

Exporter.java

```
package com.idea;

import com.intellij.util.ResourceUtil;

import java.io.*;
import java.net.URL;

public class Exporter {
   private String[][] rows;
   private String[] columns;
   private int typeFile;

   private String[] columnNameMetrics;

   public Exporter(String filename, String[][] rows, String[] columns, int
typeFile) throws IOException {
       this.rows = rows;
       this.columns = columns;
       this.typeFile = typeFile;
       export(filename);
   }

   public Exporter(String filename, String[][] rows, String[] columns, String[]
columnNameMetrics, int typeFile) throws IOException {
       this.rows = rows;
       this.columns = columns;
       this.typeFile = typeFile;
       this.columnNameMetrics = columnNameMetrics;
       export(filename);
   }

   public void export(String fileName) throws IOException {
       URL url = ResourceUtil.getResource(getClass(), "files", fileName);
       String filePath = url.getPath();
       File file = new File(filePath);

       try {
           file.createNewFile();
       } catch (IOException e1) {
           e1.printStackTrace();
       }
       PrintWriter writer = new PrintWriter(file);
       try {
           export(writer);
       } finally {
           writer.close();
       }
   }

   private void export(PrintWriter writer) throws IOException {
       if(this.typeFile == 1) { // CSV File: cell = (value1,value2)
           int index = 0;
           for (String column : this.columns) {
               if (!column.equalsIgnoreCase("")) {
                   index++;
                   writer.print(column);
```

```
                if (index != this.columns.length - 1)
                    writer.print(';');
            } else {
                writer.print(column + ';');
            }
        }
        writer.println();

        for (String[] row : this.rows) {
            index = 1;
            for (String columnValue : row) {
                writer.print(columnValue);
                if (index < row.length)
                    writer.print(';');
                index++;
            }
            writer.println();
        }
        writer.println();
    }
  }
}
```

Annexe B

Source des données publique

Représentation matricielle librairie IO

https://raw.githubusercontent.com/ivangrr/dataset/master/matrixMetricsML-io-new.csv

Fichier avec les étiquettes JUC-Tested et Tested relié à la librairie IO

https://raw.githubusercontent.com/ivangrr/dataset/master/BD10Sys-io.csv

Représentation matricielle MATH

https://raw.githubusercontent.com/ivangrr/dataset/master/matrixMetricsML-math-new.csv

Fichier avec les étiquettes JUC-Tested et Tested relié à la librairie MATH

https://raw.githubusercontent.com/ivangrr/dataset/master/BD10Sys-math.csv

Représentation matricielle Librairie JFREECHART

https://raw.githubusercontent.com/ivangrr/dataset/master/matrixMetricsML-jfreechart-new.csv

Fichier avec les étiquettes JUC-Tested et Tested relié à la librairie JFREECHART

https://raw.githubusercontent.com/ivangrr/dataset/master/BD10Sys-jfreechart.csv

Code source du modèle prédictif

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D, Conv1D,
MaxPooling1D
from tensorflow.keras.optimizers import Adam
from random import randrange
# To calculate confusion matrix
from sklearn.metrics import confusion_matrix
# To calculates precision
from sklearn.metrics import precision_score
# To calculate recall or sensitivity
from sklearn.metrics import recall_score
# To calculate specificity or sensivity
from imblearn.metrics import specificity_score
# To calculate g-mean
import math
# To show classification report
from sklearn.metrics import classification_report
# To apply label encoding to normalize the data
from sklearn.preprocessing import LabelEncoder
# To balance data
from imblearn.over_sampling import SMOTE

# Reading data file
df = pd.read_csv("url représentation matricielle selon la librairie", sep=';')

# Reading label file
df_label = pd.read_csv("url fichier avec les étiquettes selon la librairie")

# For looping to match df and df_label data with the classification label "JUC-TESTED" or
"Tested"
classification_label = 'JUC-TESTED'
#classification_label = 'Tested'

for index, row in df.iterrows():
  is_in = False
  for index_label, row_label in df_label.iterrows():
    if(str(row_label["Resource"]).find(row["Unnamed: 0"]) != -1):
      is_in = True
      df.at[index,'label'] = row_label[classification_label]
  if(is_in == False):
      df = df.drop(labels=index, axis=0)

# Taking the dataframe label column
Y_data = df['label']
```

```
# Deleting the unnecessary columns, in our case is: Unnamed: 0 and label
df = df.drop(['Unnamed: 0', 'label'], axis=1)

# Reset dataframe (df) index
df = df.reset_index(drop=True)

# Creating a new dataframe(X_data) without label data
columns = list(df)
number_row = int(df.shape[0])
total = []

for r in range(number_row):
  total_column=[]
  for c in columns:
    txt = df[c][r]
    floats_list = []

    for item in txt.split(","):
      floats_list.append(float(item))

    total_column.append(floats_list)
  total.append(total_column)

X_data = np.array(total)


# Reshaping X_data to balance it
nsamples, nx, ny = X_data.shape
X_data = X_data.reshape((nsamples,nx*ny))

# Balance the dataset
oversample = SMOTE()
X_data, Y_data = oversample.fit_resample(X_data, Y_data)

# Visualize the Number of elements in each label
plt.figure(figsize=(15,7))
sns.countplot(x=Y_data, palette="icefire")
plt.title("Number of elements in each label")
Y_data.value_counts()

# Create labels_number variable to be used in the model
labels_number = Y_data.value_counts().count()

# Reshaping X_data to be used in the model
X_data = X_data.reshape(X_data.shape[0], int(X_data.shape[1]/2), 2)

# Splitting train and test data of dataframe
x_train, x_test, y_train, y_test = train_test_split(X_data, Y_data, test_size = 0.3,
random_state=2)
```

```
# Model
model = Sequential()
model.add(Conv2D(64, kernel_size=(2,2), padding='Same', activation='relu',
input_shape=(df.shape[1], 2, 1)))
model.add(MaxPooling2D(padding='same'))
model.add(Dropout(0.2))
#
model.add(Conv2D(filters = 32, kernel_size=(2,2), padding = 'Same', activation ='relu'))
model.add(MaxPooling2D(padding='same'))
model.add(Dropout(0.2))
#
model.add(Flatten())
model.add(Dense(labels_number, activation = "softmax"))
#
model.summary()

# Defining the optimizer
learning_rate_val = 0.001
optimizer = Adam(learning_rate=learning_rate_val)
# Compiling the model
model.compile(optimizer=optimizer, loss="sparse_categorical_crossentropy",
metrics=['accuracy'])

# Training model
epochs = 20  # for better result increase the epochs
history = model.fit(x_train, y_train, epochs=epochs, validation_data=(x_test, y_test))

# Showing Loss and Accuracy in the model
plt.plot(history.history['val_loss'], color='b', label="Loss")
plt.plot(history.history['val_accuracy'], color='r', label="Accuracy")
plt.title("JUC-Tested")
#plt.title("Tested")
plt.xlabel("Nombre de passage")
plt.ylabel("%")
plt.legend()
plt.show()

# Predicting the values from the validation dataset
Y_pred = model.predict(x_test)

# Converting predictions classes to one hot vectors
Y_pred_classes = np.argmax(Y_pred, axis=1)

# Computing the confusion matrix
confusion_mtx = confusion_matrix(y_test, Y_pred_classes)

# Showing the confusion matrix
f,ax = plt.subplots(figsize=(8, 8))
sns.heatmap(confusion_mtx, annot=True, linewidths=0.01,cmap="Greens",linecolor="gray", fmt=
'.1f',ax=ax)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix")
```

```
plt.show()

# Calculating classification report
model_classification_report = classification_report(
    digits=6,
    y_true=y_test,
    y_pred=Y_pred_classes)
print(model_classification_report)
```